Geometric Search

- range search
- space partitioning trees
- intersection search
Overview

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

This lecture. Intersections among objects.

Example problems.

• 1d range search in set of $N$ numbers.
• 2d orthogonal range search in set of $N$ points.
• Find all intersections among $N$ orthogonal line segments.
• Find all intersections among $N$ orthogonal rectangles.

2d orthogonal range search

orthogonal rectangle intersection
- range search
- space partitioning trees
- intersection search
1d range search

Extension of ordered symbol table.
• Insert key-value pair.
• Search for key $k$.
• Range search: find all keys between $k_1$ and $k_2$.
• Range count: number of keys between $k_1$ and $k_2$.

Application. Database queries.

Geometric interpretation.
• Keys are point on a line.
• Find/count points in a given interval.
1d range search: implementations

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).

Red-black BST. All operations fast.

Parameters. \( N = \text{number of keys}; \ R = \text{number of keys that match}. \)

Running time is output sensitive
(number of matching keys can be \( N \))

<table>
<thead>
<tr>
<th>data structure</th>
<th>insert</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>( R + \log N )</td>
</tr>
<tr>
<td>hash table</td>
<td>1</td>
<td>( N )</td>
<td>( N )</td>
</tr>
<tr>
<td>red-black BST</td>
<td>( \log N )</td>
<td>( \log N )</td>
<td>( R + \log N )</td>
</tr>
</tbody>
</table>

Order of growth of running time for 1d range search
1d range search: BST implementation

**Range search.** Find all keys between $k_1$ and $k_2$.
- 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).

**Proposition.** Running time is proportional to $R + \log N$ (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.
• Range count: number of keys that lie in a 2d range.

Applications. Networking, circuit design, databases.

Geometric interpretation.

• Keys are point in the plane.
• Find/count points in a given h-v rectangle.

rectangle is axis-aligned
2d orthogonal range search: grid implementation

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:** \( 1 + \frac{N}{M^2} \) 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 data structure: \( N \).
- Insert point: 1.
- Range search: 1 per point in range.
Grid implementation. Fast, simple solution for well-distributed points.

Problem. Clustering a well-known phenomenon in geometric data.

• Lists are too long, even though average length is short.
• Need data structure that *gracefully* adapts to data.
**Clustering**

*Grid implementation.* Fast, simple solution for well-distributed points.

**Problem.** Clustering a well-known phenomenon in geometric data.

**Ex.** USA map data.

![Map of USA with scattered points]

13,000 points, 1000 grid squares

- Half the squares are empty
- Half the points are in 10% of the squares
› range search
› space partitioning trees
› intersection search
Space-partitioning trees

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

Grid. Divide space uniformly into squares.
Quadtree. Recursively divide space into four quadrants.
2d tree. Recursively divide space into two halfplanes.
BSP tree. Recursively divide space into two regions.
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.

Grid
Quadtree
2d tree
BSP tree
**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!

```java
public class QuadTree {
    private Quad quad;
    private Value val;
    private QuadTree NW, NE, SW, SE;
}
```
Quadtree: larger example

Quadtree: 2d orthogonal range search

Range search. Find all keys in a given 2d range.
• Recursively find all keys in NE quadrant (if any could fall in range).
• Recursively find all keys in NW quadrant (if any could fall in range).
• Recursively find all keys in SE quadrant (if any could fall in range).
• Recursively find all keys in SW quadrant (if any could fall in range).

Typical running time. $R + \log N$. 
**N-body simulation**

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

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

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

[YouTube Video: "N-body similation"](http://www.youtube.com/watch?v=ua7YIN4eL_w)
Barnes-Hut algorithm for 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

- 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

**kd range search.** Orthogonal range search in \( k \)-dimensions.

**Main application.** Multi-dimensional databases.

3d space. Octrees: recursively subdivide 3d space into 8 octants.

100d space. Centrees: recursively subdivide 100d space into \( 2^{100} \) centrants???

Raytracing with octrees
Recursively partition plane into two halfplanes.
2d tree implementation

Data structure. BST, but alternate using x- and y-coordinates as key.
- Search gives rectangle containing point.
- Insert further subdivides the plane.
2d tree: 2d orthogonal range search

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.
• Adapts well to high-dimensional and clustered data.
• Discovered by an undergrad (Jon Bentley) in an algorithms class!
- range search
- space partitioning trees
- intersection search
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 line segment intersection search: sweep-line algorithm

Sweep vertical line from left to right.
• $x$-coordinates define events.
• $h$-segment (left endpoint): insert $y$-coordinate into ST.
Orthogonal line segment intersection search: sweep-line algorithm

Sweep vertical line from left to right.

• $x$-coordinates define events.
• $h$-segment (left endpoint): insert $y$-coordinate into ST.
• $h$-segment (right endpoint): remove $y$-coordinate from ST.
Orthogonal line segment intersection search: sweep-line algorithm

Sweep vertical line from left to right.

- $x$-coordinates define events.
- $h$-segment (left endpoint): insert $y$-coordinate into ST.
- $h$-segment (right endpoint): remove $y$-coordinate from ST.
- $v$-segment: range search for interval of $y$-endpoints.
Orthogonal line segment intersection search: sweep-line algorithm

Sweep line reduces 2d orthogonal line segment intersection to 1d range search.

**Proposition.** The sweep-line algorithm takes time proportional to $N \log N + R$ to find all $R$ intersections among $N$ orthogonal segments.

- Put $x$-coordinates on a PQ (or sort). $N \log N$
- Insert $y$-coordinates into ST. $N \log N$
- Delete $y$-coordinates from ST. $N \log N$
- Range searches. $N \log N + R$

Efficiency relies on judicious use of data structures.

**Remark.** Sweep-line solution extends to 3d and more general shapes.
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 $\Rightarrow$ one new pair of adjacent segments.
- Intersection $\Rightarrow$ swap adjacent segments.
Efficient implementation of sweep line algorithm.

- Maintain PQ of important $x$-coordinates: endpoints and intersections.
- Maintain set of segments intersecting sweep line, sorted by $x$.
- Time proportional to $R \log N + N \log N$.

Implementation issues.

- Degeneracy.
- Floating-point precision.
- Must use PQ, not presort (intersection events are unknown ahead of time).
Orthogonal rectangle intersection search

**Goal.** Find all intersections among a set of $N$ orthogonal rectangles.

**Non-degeneracy assumption.** All $x$- and $y$-coordinates are distinct.

**Application.** Design-rule checking in VLSI circuits.
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 = orthogonal rectangle intersection search.
"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.
Orthogonal rectangle intersection search

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

- **Sweep line**: sort rectangles by \( x \)-coordinates and process in this order, stopping on left and right endpoints.
- **Maintain set of \( y \)-intervals** intersecting sweep line.
- **Left endpoint**: search set for intersecting \( y \)-intervals; insert \( y \)-interval.
- **Right endpoint**: delete \( y \)-interval.
Interval search trees

public class IntervalST<Key extends Comparable<Key>, Value>

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IntervalST()</td>
<td>create interval search tree</td>
</tr>
<tr>
<td>void put(Key lo, Key hi, Value val)</td>
<td>put interval-value pair into ST</td>
</tr>
<tr>
<td>Value get(Key lo, Key hi)</td>
<td>return value paired with given interval</td>
</tr>
<tr>
<td>void remove(Key lo, Key hi)</td>
<td>remove the given interval</td>
</tr>
<tr>
<td>Iterable&lt;Value&gt; intersects(Key lo, Key hi)</td>
<td>return all intervals that intersect the given interval</td>
</tr>
</tbody>
</table>

```
(4, 8)     (5, 11)     (15, 18)     (17, 19)     (20, 22)
```

(7, 10)
**Interval search trees**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>IntervalST()</code></td>
<td>create interval search tree</td>
</tr>
<tr>
<td><code>void put(Key lo, Key hi, Value val)</code></td>
<td>put interval-value pair into ST</td>
</tr>
<tr>
<td><code>Value get(Key lo, Key hi)</code></td>
<td>return value paired with given interval</td>
</tr>
<tr>
<td><code>void remove(Key lo, Key hi)</code></td>
<td>remove the given interval</td>
</tr>
<tr>
<td><code>Iterable&lt;Value&gt; intersects(Key lo, Key hi)</code></td>
<td>return all intervals that intersect the given interval</td>
</tr>
</tbody>
</table>

**Non-degeneracy assumption.** No two intervals have the same left endpoint.
Interval search trees

Create BST, where each node stores an interval \((lo, hi)\).

- Use left endpoint as BST key.
- Store max endpoint in subtree rooted at node.

Suffices to implement all ops efficiently!
Finding an intersecting interval

To search for any interval that intersects query interval \((lo, hi)\):

```
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;
```

Ex. Search for \((9, 10)\).
Finding an intersecting interval

To search for any interval that intersects query interval \((lo, hi)\):

\[
\text{Node } x = \text{root}; \\
\text{while } (x \neq \text{null}) \\
\{ \\
\quad \text{if } (x.\text{interval}.\text{intersects}(lo, hi)) \\
\quad \quad \text{return } x.\text{interval}; \\
\quad \text{else if } (x.\text{left} == \text{null}) \quad x = x.\text{right}; \\
\quad \text{else if } (x.\text{left}.\text{max} < lo) \quad x = x.\text{right}; \\
\quad \text{else} \quad x = x.\text{left}; \\
\}\\\n\text{return } \text{null};
\]

**Case 1.** If search goes right, then no intersection in left.

**Pf.**

- \((x.\text{left} == \text{null}) \Rightarrow \text{trivial.}\)
- \((x.\text{left}.\text{max} < lo) \Rightarrow \text{for any interval } (a, b) \text{ in left subtree of } x,\)
  
  we have \(b \leq \text{max} < lo.\)

definition of max  
reason for going right
Finding an intersecting interval

To search for any interval that intersects query interval \((lo, hi)\):

```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 there is either an intersection in left subtree or no intersections in either.

**Pf.** Suppose no intersection in left. Then for any interval \((a, b)\) in right subtree of \(x\), \(hi < c \leq a \Rightarrow\) no intersection in right.
Interval search tree: analysis

Implementation. Use a red-black BST to guarantee performance.

- Insert interval: \( \log N \) in the tree, \( \log N \) in theory.
- Find interval: \( N \) in the tree, \( \log N \) in theory.
- Delete interval: \( N \) in the tree, \( \log N \) in theory.
- Find any interval that intersects \((lo, hi)\): \( N \) in the tree, \( \log N \) in theory.
- Find all intervals that intersect \((lo, hi)\): \( R \log N \), \( R + \log N \) in theory.

Order of growth of running time for \( N \) intervals.
Move a vertical "sweep line" from left to right.

- **Sweep line:** sort rectangles by $x$-coordinates and process in this order, stopping on left and right endpoints.
- **Maintain set of rectangles that intersect the sweep line in an interval search tree (using $y$-intervals of rectangle).**
- **Left endpoint:** interval search for $y$-interval of rectangle; insert $y$-interval.
- **Right endpoint:** delete $y$-interval.
Rectangle intersection search: costs summary

Sweep line reduces 2d orthogonal rectangle intersection search to 1d interval search.

**Proposition.** Sweep line algorithm takes time proportional to $N \log N + R$ to find $R$ intersections among a set of $N$ rectangles.

- Put $x$-coordinates on a PQ (or sort). $N \log N$
- Insert $y$-intervals into ST. $N \log N$
- Delete $y$-intervals from ST. $N \log N$
- Interval searches for $y$-intervals. $N \log N + R$

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

<table>
<thead>
<tr>
<th>problem</th>
<th>example</th>
<th>solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>1d range search</td>
<td><img src="image" alt="1d range search" /></td>
<td>BST</td>
</tr>
<tr>
<td>kd orthogonal range search</td>
<td><img src="image" alt="kd orthogonal range search" /></td>
<td>kd tree</td>
</tr>
<tr>
<td>1d interval search</td>
<td><img src="image" alt="1d interval search" /></td>
<td>interval search tree</td>
</tr>
<tr>
<td>2d orthogonal line segment intersection</td>
<td><img src="image" alt="2d orthogonal line segment intersection" /></td>
<td>sweep line reduces to 1D range search</td>
</tr>
<tr>
<td>2d orthogonal rectangle intersection</td>
<td><img src="image" alt="2d orthogonal rectangle intersection" /></td>
<td>sweep line reduces to 1D interval search</td>
</tr>
</tbody>
</table>