Geometric Search

range search
intersections of geometric objects
near-neighbor search
point location

Types of data
- points, lines, planes; polygons, circles, ...

SETS of N objects

Problems extend to higher dimensions
- good algorithms also extend to higher dimensions

Higher level intrinsic structures arise (ex: convex hull)

Basic problems
- range search
- intersections
- near neighbor search

Range search (1D)

Useful extension to symbol-table ADT
for records with numeric keys
- create
- insert
- search
- test if empty
- range search: how many records have key values that fall within a given range?

Change semantics of search
- require initial call to range search (count items in successful search)
- return items in successive search calls

Typical client code:
```c
int cnt = STrange(L, R);
for (i = 0; i < cnt; i++)
  x = STsearch(); /* process x */
```

Ordered array
- slow insert
- binary search on both interval endpoints for range
- increment and test index for search

Hash table
- no reasonable algorithm (key order lost in hash)

BST
- search on both endpoints for range (need threads for fast search)

<table>
<thead>
<tr>
<th></th>
<th>insert</th>
<th>range</th>
<th>search</th>
</tr>
</thead>
<tbody>
<tr>
<td>ordered array</td>
<td>N</td>
<td>lg N</td>
<td>1</td>
</tr>
<tr>
<td>hash table</td>
<td>1</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>BST</td>
<td>lg N</td>
<td>lg N</td>
<td>1</td>
</tr>
</tbody>
</table>

Application: database queries
Recursively search all subtrees that **could have** keys in range

- if key at root is **within** range
  - increment global counter
  - search both subtrees
- if key at root is **left** of range, no need to search left subtree
- if key at root is **right** of range, no need to search right subtree

Slightly simpler logic:
- **not left** implies within or right, so search left
- **not right** implies within or left, so search right

```c
int BSTrangeR(link h, Key L, Key R)    { int txL = (h->key >= L);       int txR = (h->key <= R);      if (txL && (h->l != z)) BSTrangeR(h->l); if (txR && (h->r != z)) BSTrangeR(h->r);    }  int BSTrange(Key L, Key R)    { count = 0; BSTrangeR(head, L, R); }
```

**1D range search BST implementation**

**Range search (2D)**

Useful extension to symbol-table ADT for records with 2-dimensional keys

- **create**
- **insert**
- **search**
- **test if empty**
- **range search**: how many records have key values that fall within a given range?

**Geometric interpretation**

1D range search
- keys are points on the line
- how many points in a given interval?

2D range search
- keys are points in the plane
- how many points in a given rectangle?

**2D range search grid implementation costs**

**Classic example (see Sedgewick Chapter 3)**

- **array**: constant-time access to list by indexing
- **list**: O(N) space for sets of varying size (total size N)

Choose grid square size to tune performance

- too small: space, initialization cost too high
- too large: too many points per grid square
- rule of thumb: √N by √N grid (~N squares)

**Time costs:**

- **initialize**: O(N) to initialize lists
- **insert**: O(1) provided points evenly distributed
- **range**: O(1) per point in range (same provision)

Simple, fast solution for well-distributed points

BUT can be slow (points might all be in same square)

Need more flexible data structure
Recursive search structure for 2D keys (points in the plane)

Standard BST, but alternate using x and y coordinates as key

Corresponds to planar subdivision useful for many geometric algorithms

Even levels
- Points left of x
- Points right of x

Odd levels
- Points below x
- Points above x

Search gives rectangle containing point
Insert further subdivides plane

2D range search 2D tree implementation

Recursively search all subtrees that could have keys in range
- If key at root is in the range, increment counter
- At even level
  - If root's key is left of or within range, search right subtree
  - If root's key is right of or within range, search left subtree
- At odd level
  - If root's key is above or within range, search lower subtree
  - If root's key is below or within range, search higher subtree

```c
int count;
int TDTrangeR(link h, Point LB, Point RT, int sw)
    { int txL = (h->p.x >= LB.x);       int txR = (h->p.x <= RT.x);      ...        TDTrangeR(h->r LB, RT, !sw);    }
int BSTrange(Key LB, Key RT)
    { count = 0; BSTrangeR(head, LB, RT, 0); }
```

Range search (2D) implementations

Grid
- Clustering worst case

kD tree
- BST search for range
  - (need threads for fast search)

<table>
<thead>
<tr>
<th></th>
<th>insert</th>
<th>range</th>
<th>search</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>random points</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>unordered array</td>
<td>1</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>kD tree</td>
<td>lg N</td>
<td>R + lg N</td>
<td>1</td>
</tr>
<tr>
<td>grid</td>
<td>1</td>
<td>R</td>
<td>1</td>
</tr>
<tr>
<td><strong>worst case points</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>kD tree</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>grid</td>
<td>1</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td><strong>random order</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>grid</td>
<td>1</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>2D tree</td>
<td>lg N</td>
<td>R + lg N</td>
<td>1</td>
</tr>
</tbody>
</table>

Clustering

Geometric data is seldom uniformly random

Example: USA map data
- 80000 points, 20000 grid squares
- Half the grid squares are empty
- Half the points have >10 others in same grid square
- 10 percent have >99 others in same grid square

Clustering is a well-known phenomenon even in random data
Problems worsen in higher dimensions

Good clustering performance is a primary reason to choose kD trees over grid methods
**kD trees**

Recursive search structure for kD keys (points in k-dimensional space)
Standard BST, but cycle through dimensions for key coordinates
Corresponds to spatial subdivision useful for many geometric algorithms

\[ \text{level} = i \mod k \]

- Points whose i-th coordinate is less than x's
- Points whose i-th coordinate is greater than x's

Search gives kD parallelopiped containing point
Insert further subdivides space

Efficient, simple data structure for processing kD data

*Note*: 2D and kD trees were discovered by an undergraduate in an algorithms class!

---

**Fast algorithm for h-v line intersection**

Use horizontal sweep line moving from top to bottom
- Vertical line segment in data is a point on the sweep line
- Horizontal line segment in data is an interval on the sweep line
- h-v intersection when points within interval

Reduces 2D h-v line intersection to 1D range searching (!)

---

**Geometric intersection**

Problem: Find all intersecting pairs among a set of N geometric objects

Applications:
- CAD (stay tuned)
- Games, movies, virtual reality

Simplest version:
- 2D
- All objects are horizontal or vertical line segments

Solution approach extends to 3D and general objects

---

**Sweep-line h-v intersection implementation**

Use priority queue ADT on y to simulate sweep line movement
Use range search ADT on x to simulate sweep line contents

Three types of events
- Top of vertical: insert x coordinate onto the sweep line
- Bottom of vertical: remove x coordinate from the sweep line
- Horizontal: range search on endpoints
Use priority queue ADT on y to simulate sweep line movement
Use range search ADT on x to simulate sweep line contents

```
PQinit(); STinit();
for (i = 0; i < N; i++)
PQinsert(lines[i]);
while (!PQempty())
{
    t = PQdelmax();
    if (horizontal(t))
        { cnt = STrange(t.p0.x, t.p1.x);
          for (i = 0; i < cnt; i++) intersection(t, STsearch());
        }
    else if (top(t)) STinsert(t);
    else if (bottom(t)) STdelete(t);
}
```

Running time:
- O(N) insert and delmax ops for PQ
- O(N) insert, delete, and range ops for ST
Total: O(N log N) (with suitable ADT implementations)

Same basic idea extends to handle arbitrary geometric shapes (!!)

**Near neighbor search**

Another useful extension to symbol-table ADT for records with metric keys
- create
- insert
- test if empty
- near neighbor search: which record has a key that is nearest to a given key?

Need concept of distance (not just less)

kD trees provide fast, elegant solution
- recursively search subtrees that could have near neighbor (may search both)
- \(O(\log N)\)

**Voronoi diagram**

Ultimate near-neighbor search structure
- **Voronoi region**: set of all points closest to a given point
- **Voronoi diagram**: planar subdivision delineating Voronoi regions (note: Voronoi edges are perpendicular bisector segments)
- **Delauney triangulation**: dual of Voronoi diagram (includes convex hull!) edge p-q in Delauney iff p-q bisector segment in Voronoi

Challenge: compute the Voronoi
Adding a point to Voronoi diagram

Basis for incremental algorithms
Region containing point gives points to check to compute new Voronoi region boundaries

Main challenge in computing Voronoi: representing it
Use multilist associating each point with its Voronoi neighbors

Randomized incremental Voronoi algorithm
Add points (in random order)
• find region containing point use near-neighbor algorithm or (with work) Voronoi itself
• update neighbor regions, create region for new point
Running time: $O(N \log N)$

Sweep-line Voronoi algorithm
Presort points on $x$-coordinate
Eliminates point location (as for convex hull)

Discretized Voronoi diagram
Use grid approach to answer near-neighbor queries in constant time
Approach 1: provide approximate answer (to within grid square size)
Approach 2: keep list of points to check in grid squares
Computation not difficult (move outward from points)
### Summary

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

<table>
<thead>
<tr>
<th>Basis</th>
<th>Grid</th>
<th>2D Tree</th>
<th>Voronoi Diagram</th>
<th>Intersecting Lines</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Representation</strong></td>
<td>( \sqrt{N} ) h-v lines</td>
<td>N points</td>
<td>N points</td>
<td>( \sqrt{N} ) lines</td>
</tr>
<tr>
<td><strong>Cells</strong></td>
<td>2D array of N lists</td>
<td>N-node BST</td>
<td>N-node multilist</td>
<td>~N-node BST</td>
</tr>
<tr>
<td><strong>Search Cost</strong></td>
<td>( \sim N ) squares</td>
<td>N rectangles</td>
<td>N polygons</td>
<td>~N triangles</td>
</tr>
<tr>
<td><strong>Extend to kD?</strong></td>
<td>too many cells</td>
<td>easy</td>
<td>cells too complicated</td>
<td>use (k-1)D hyperplane</td>
</tr>
</tbody>
</table>