Geometric Search

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

Geometric search: overview

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:

```
cnt = STrange(L, R);
for (i = 0; i < cnt; i++)
  {
    x = STsearch();
    /* process x */
  }
```

Application: database queries

ST.h

```
void STinit();
void STinsert(Item);
Item STsearch();        ← no arg
 int STempty();
 int STrange(Key, Key)  ← new function
```

ST interface in C

insert B	B
insert D	B D
insert A	A B D
insert I	A B D I
insert E	A B D E I
insert A	A A B D E I
insert H	A A B D E H I
insert F	A A B D E F H I
range E to H	3
search	E
search	F
search	H

Range search (1D) implementations

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)

	insert	range	search
ordered array	N	lg N	1
hash table	1	N	N
BST	lg N	lg N	1

1D range search BST implementation

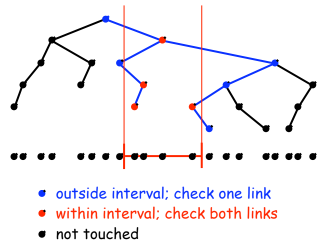
Recursively search all subtrees that could have keys in range

- 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

```
int count;
int BSTrangeR(link h, Key L, Key R)
  { int txL = (h->key >= L);   // not left of range
    int txR = (h->key <= R);   // not right of range
    if (txL && (h->l != z)) BSTrangeR(h->l);
    if (txL && txR) count++;
    if (txR && (h->r != z)) BSTrangeR(h->r);
  }
int BSTrange(Key L, Key R)
  { count = 0; BSTrangeR(head, L, R); }
```



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?

ST.h

```
void STinit();
void STinsert(Item);
Item STsearch();
 int STempty();
 int STrange(Key, Key)
```

ST interface in C

same as for 1D

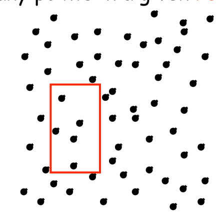
Geometric interpretation

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

init

- divide space into G-by-G squares
- create linked list for each square

insert

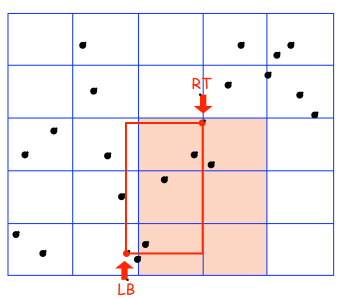
- use coordinates to index proper list
- add point to list

range

- use range coordinates to index squares that could have keys in range
- examine all records in all such squares
- if key is in the range, increment counter

```
typedef struct Node* link;
struct Node { Point p; link next; };
link grid[maxX/G][maxY/G];
int GRIDinit()
  { int i, j;
    for (i = 0; i < maxX/G; i++)
      for (j = 0; j < maxY/G; j++)
        grid[i][j] = NULL;
  }
int GRIDinsert(Point p)
  {
    link t = malloc(sizeof *t);
    t->p = p;
    t->next = grid[p.x/G][p.y/G];
    grid[p.x/G][p.y/G] = t;
  }
```

```
int count;
int GRIDrange(Point LB, Point RT)
  { int i, j;
    for (i = LB.x/G; i <= RT.x/G; i++)
      for (j = LB.y/G; j <= RT.y/G; j++)
        for (t = grid[i][j]; t != NULL; t = t->next)
          if ( t->p.x >= LB.x &&
               t->p.x <= RT.x &&
               t->p.y >= LB.y &&
               t->p.y <= RT.y ) count++;
  }
```



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

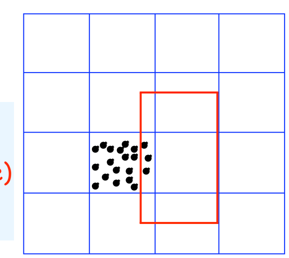
- too small: space, initialization cost too high
- too large: too many points per grid square
- rule of thumb: √N by √N grid (~N squares)

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

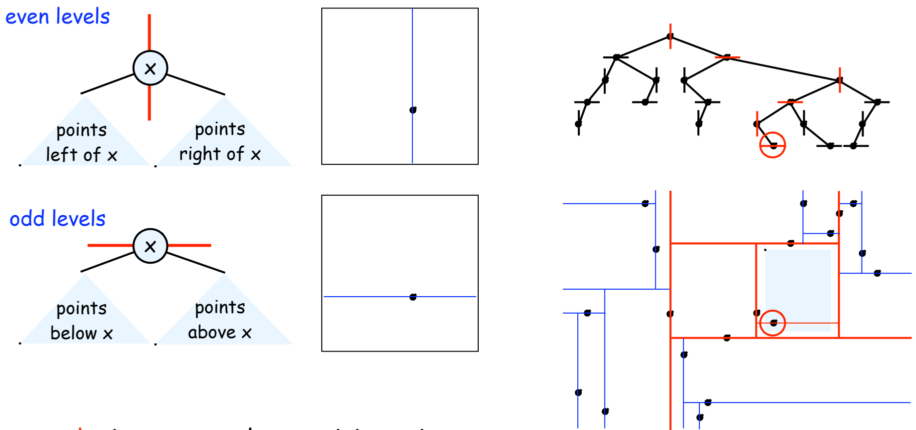


2D trees

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



search gives rectangle containing point

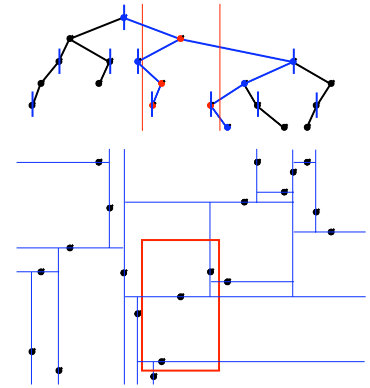
insert further subdivides plane

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

```
int count;
int TDTrangeR(link h, Point LB, Point RT, int sw)
  { int txL = (h->p.x >= LB.x);      ← not left
    int txR = (h->p.x <= RT.x);      ← not right
    int tyB = (h->p.y >= LB.y);      ← not below
    int tyT = (h->p.y <= RT.y);      ← not above
    t1 = sw ? txL : tyB; t2 = sw ? txR : tyT;
    if (t1 && (h->l != NULL))
      TDTrangeR(h->l LB, RT, !sw);
    if (txL && txR && tyB && tyT) count++;
    if (t2 && (h->r != NULL))
      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)

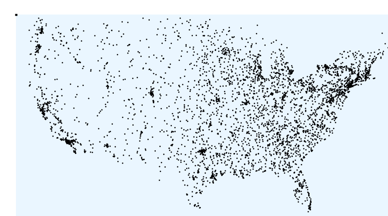
		insert	range	search
random points				
	unordered array	1	N	N
	kD tree	lg N	R + lg N	1
	grid	1	R	1
worst case points				
	kD tree	N	N	N
	grid	1	N	N
random order				
	grid	1	N	N
	2D tree	lg N	R + lg N	1

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

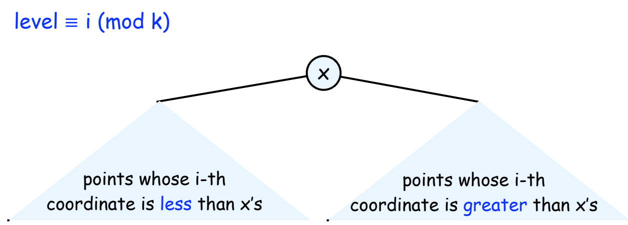
Good clustering performance is a primary reason to choose kD trees over grid methods

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



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!

Geometric intersection

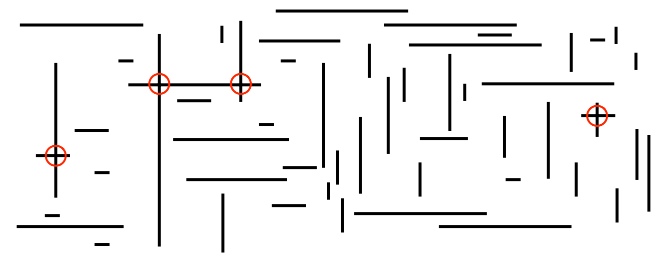
Problem: Find all intersecting pairs among a set of N geometric objects

Applications:

- CAD (stay tuned)
- games, movies, virtual reality

Simplest version:

- 2D
- all objects are horizontal or vertical line segments



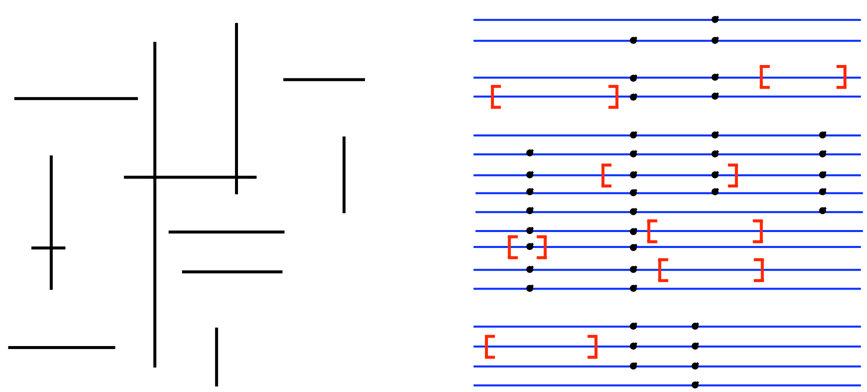
Solution approach extends to 3D and general objects

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 (!)



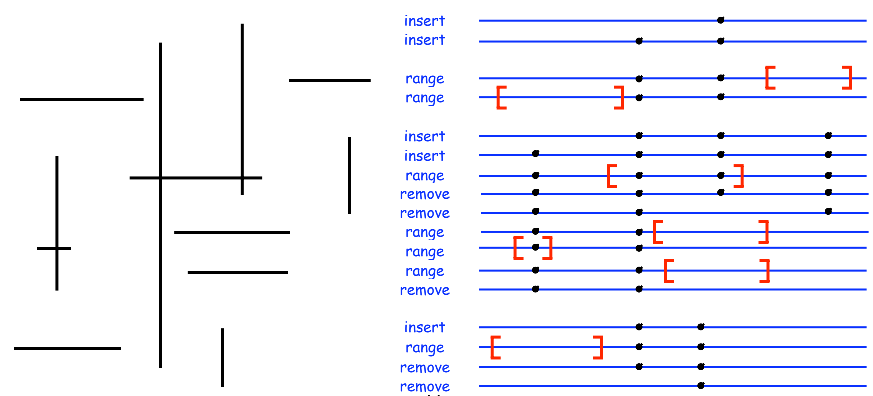
Sweep-line h-v intersection implementation

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



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

```
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);
  }
```

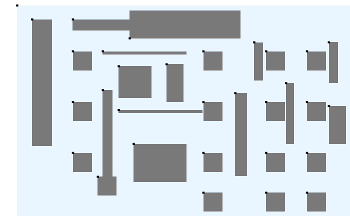
Running time:
- O(N) insert and delmax ops for PQ
- O(N) insert, delete, and range ops for ST

Total: O(N log N) ⬅
(with suitable ADT implementations)

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

Digression: algorithms and Moore's Law

Problem: Find intersections in N h-v rectangles
Solution: Slight modification to sweep-line h-v line intersection algorithm
Application: microprocessor design



early 1970s: microprocessor design became a geometric problem
- Very Large Scale Integration
- Computer-Aided Design
- design-rule checking

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

Quadratic algorithm: (compare each rectangle against all others)
- 197x: takes M days
- 197(x+1.5): takes (4M)/2 = 2M days (!!)

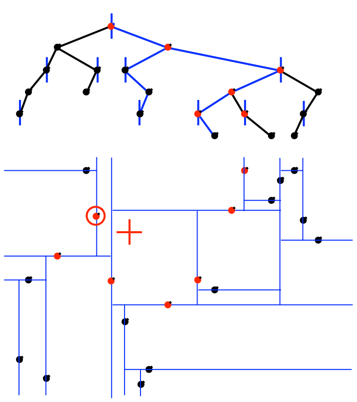
Need O(N log N) CAD algorithms to sustain Moore's Law

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) ?



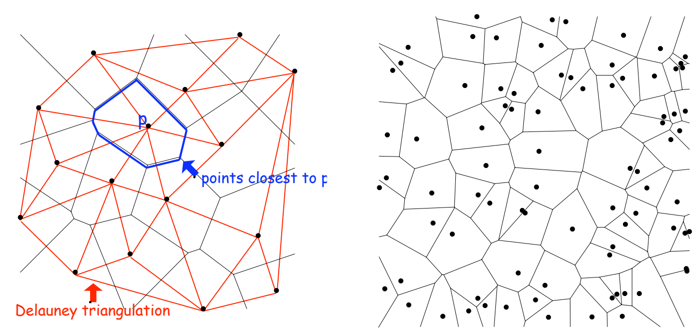
Voronoi diagram

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

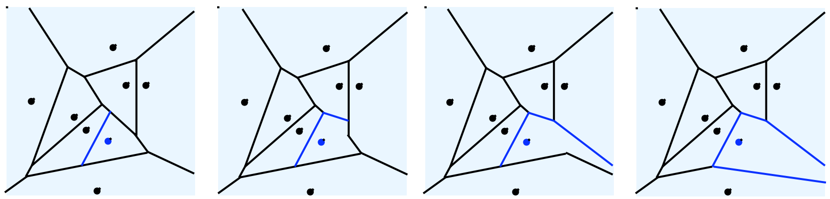


Challenge: compute the Voronoi

Adding a point to Voronoi diagram

Basis for incremental algorithms

Region containing point gives points to check to compute new Voronoi region boundaries



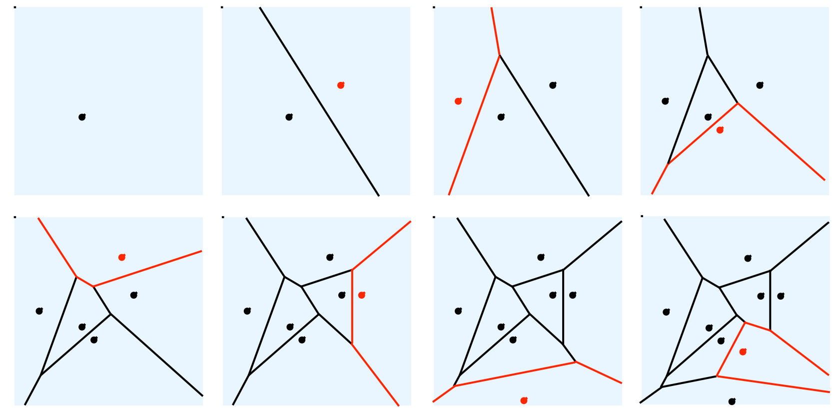
Main challenge in computing Voronoi: representing it

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

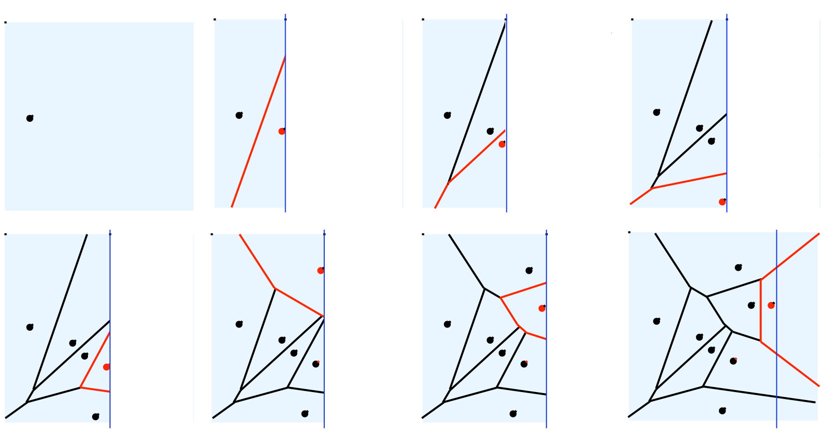


Running time: O(N log N)

Sweep-line Voronoi algorithm

Presort points on x-coordinate

Eliminates point location (as for convex hull)

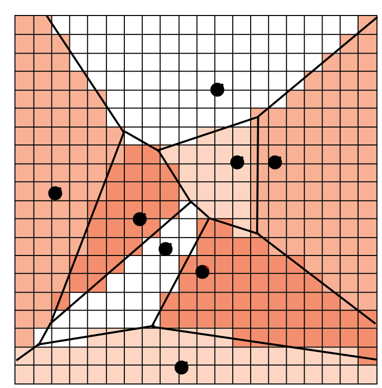


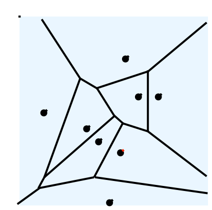
Discretized Voronoi diagram

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

	grid	2D tree	Voronoi diagram	intersecting lines
basis	√N h-v lines	N points	N points	√N lines
representation	2D array of N lists	N-node BST	N-node multilist	~N-node BST
cells	~N squares	N rectangles	N polygons	~N triangles
search cost	1	log N	log N	log N
extend to kD?	too many cells	easy	cells too complicated	use (k-1)D hyperplane