COS 226 Lecture 15: Geometric search

Extend search ADT to geometric data

Range search
Intersections among geometric objects
Near neighbor search
Point location

Multidimensional problems

Trees
Divide-and-conquer
Discretized algorithms

Range searching

Possible addition to Search ADT:

```
.       void STinit();
.       void STinsert(recordType x);
. recordType STsearch(keyType v);
.        int STempty();
.        int STrange(keyType v1, keyType v2);
```

Count the number of records with key values
    between v1 and v2

Options to actually process the records
* pass a procedure to be called
    for each record in the range
* return list of records
    (possibly sorted)

Depends on how many records expected
    (count them first)

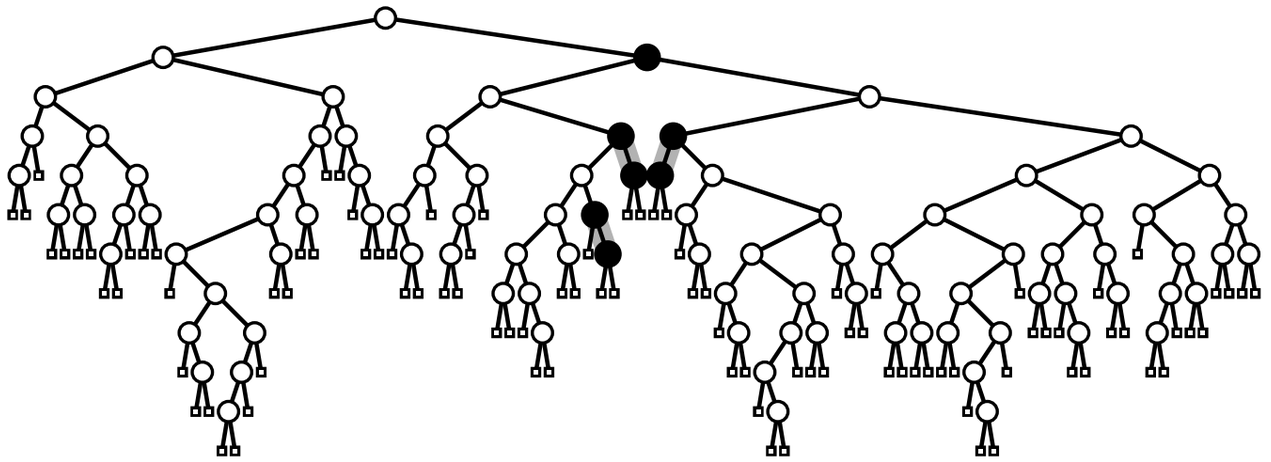
Implementations
    ARRAY: do binary search on both v1 and v2
        return difference between indices
    HASH TABLE: no easy algorithm
    BST, TRIE: recursive traversal works

BST 1D range searching

Recursive traversal

* eliminate subtrees with no keys in interval
* root may or may not be in interval
* traverse both subtrees if it is

```
int v1, v2, count = 0;
BSTrangeR(struct node *t)
   { int tx1 = (t->key >= v1);
     int tx2 = (t->key <= v2);
     if (tx1 && (t->l != z)) BSTrangeR(t->l);
     if (tx1 && tx2) count++;
     if (tx2 && (t->r != z)) BSTrangeR(t->r);
   }
```



2D Range searching

Same basic method works in higher dimensions (!!)
(discovered by an undergraduate)

interval in 1D is rectangle in 2D

Recursive traversal
* eliminate subtrees with no keys in rectangle
* root may or may not be in rectangle
* traverse both subtrees if it is

Corresponds to recursive subdivision of the plane
alternating horizontal and vertical lines

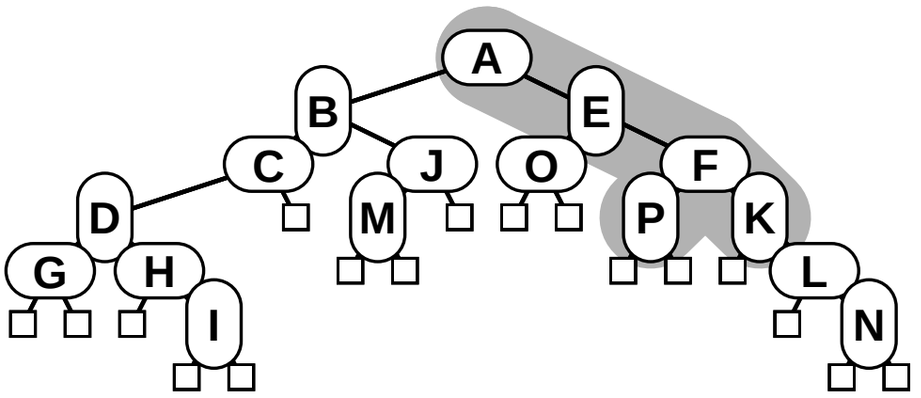
Trivial generalization to k dimensions

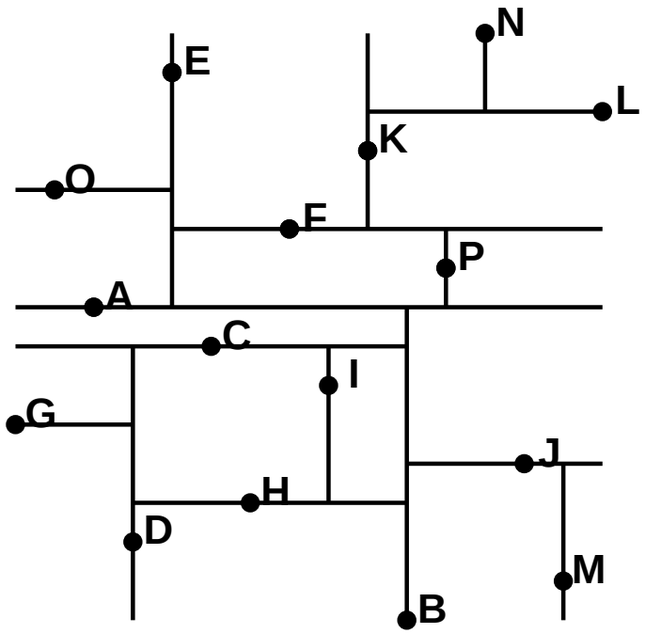
2D tree construction

Each node corresponds to an area
in the plane

Each internal node divides its area
into two subdivisions

Switch between horizontal and vertical
dividing lines

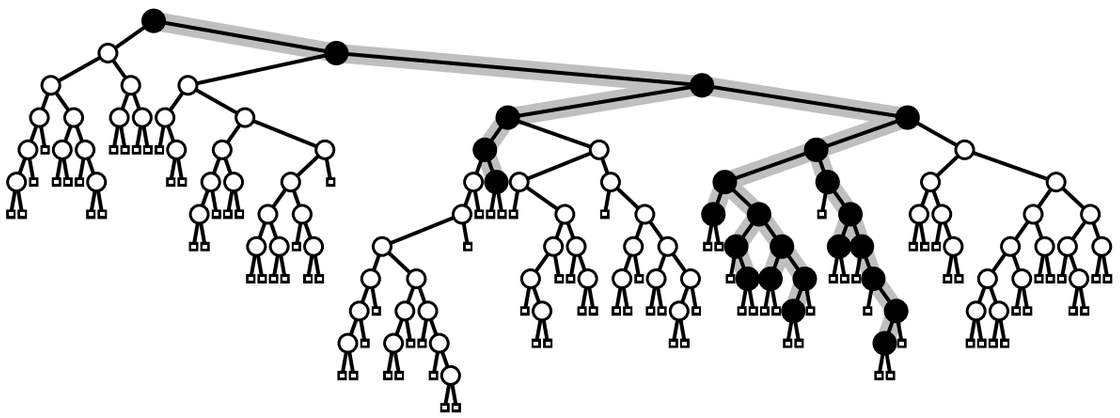


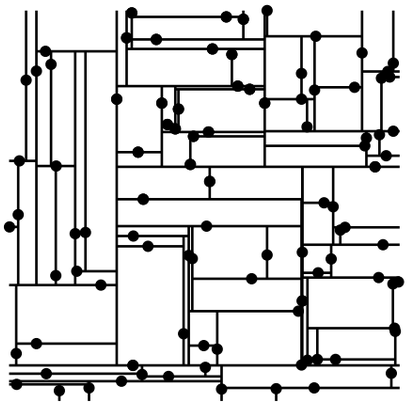


Quad tree: use 4-way tree
divide on both coordinates at once

2D tree range searching

```
int x1, y1, x2, y2, count = 0;
TDTrangeR(struct node *t, int d)
 {
   int t1,t2,tx1,tx2,ty1,ty2;
   if (t == z) return;
   tx1 = x1 < t->p.x; tx2 = t->p.x <= x2;
   ty1 = y1 < t->p.y; ty2 = t->p.y <= y2;
   t1 = d ? tx1 : ty1; t2 = d ? tx2 : ty2;
   if (t1 && (t->l != z)) TDTrangeR(t->l, !d);
   if (tx1 && tx2 && ty1 && ty2) count++;
   if (t2 && (t->r != z)) TDTrangeR(t->r, !d);
 }
```



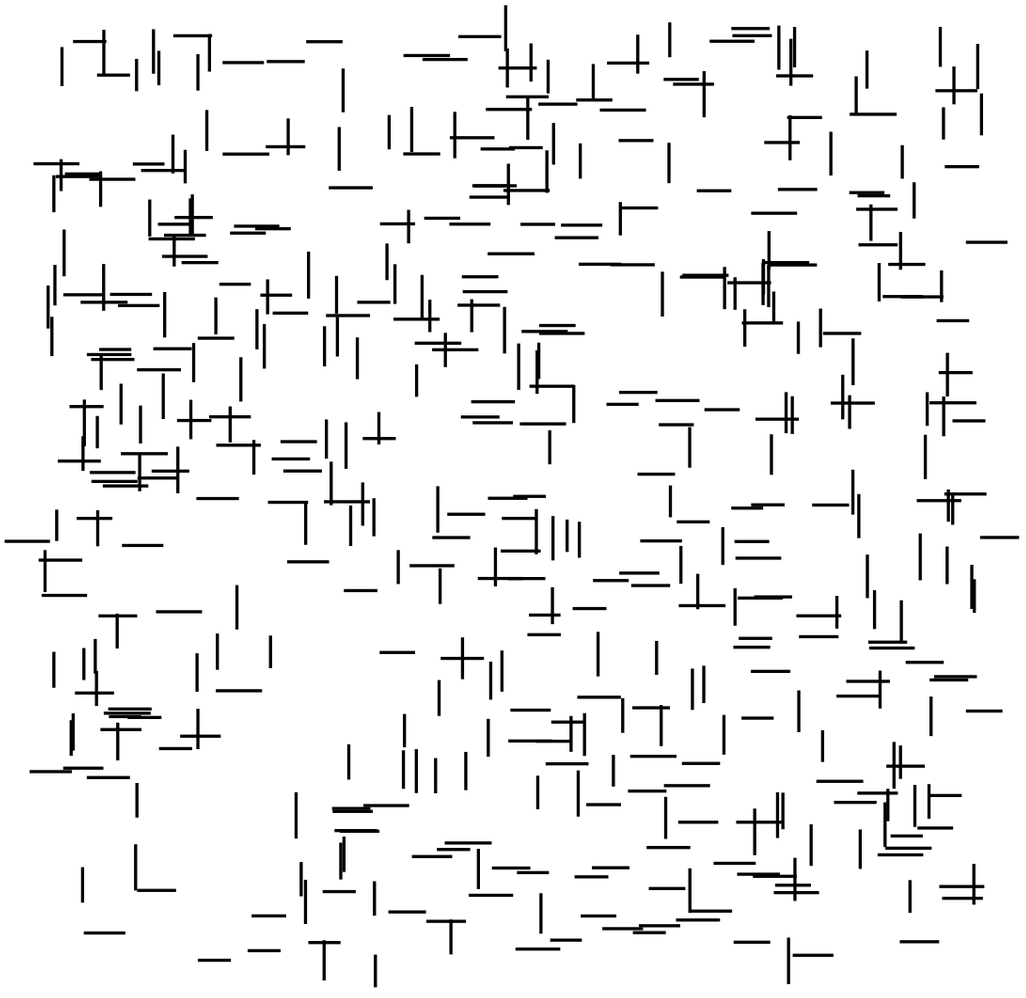


Manhattan line intersection problem

N lines, either horizontal or vertical

How many pairs intersect?

Call a given procedure for each intersection



As with other search problems

usually no harder to REPORT all intersections

(call a given procedure for each)

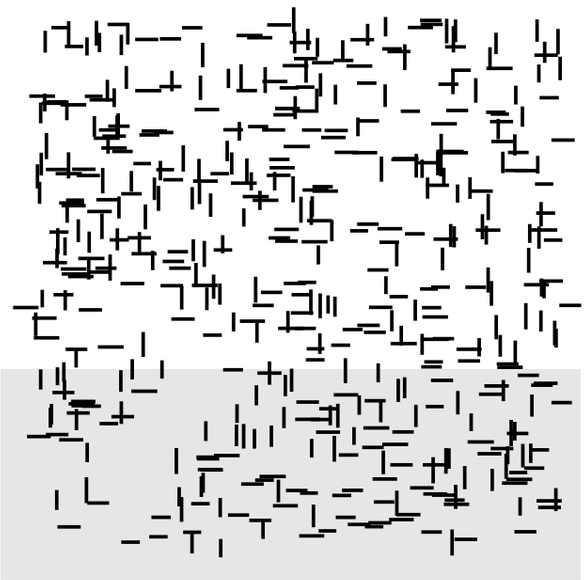
Manhattan line intersection

Dynamic SWEEP LINE algorithm

Horizontal line sweeps from bottom to top

At any given instant

* vertical data line represents "point"
* horizontal data line represents "interval"



There is an h-v intersection

if "point" is in "interval"

Reduces 2D line intersection problem

to 1D range searching!

Sweep line implementation

Uses both PQ and ST (with range search) ADT

Consider y coordinates of endpoints
in increasing order (use PQ)

Three types of "events"
- B: bottom of vertical line
- T: top of vertical line
- H: horizontal line

Maintain dynamic ST
where an item with key x means that
some vertical line intersects
the sweep line at x

B: INSERT x
T: DELETE x
H: RANGE (x1, x2)

Generalizes to give fast algorithm for
- * rectangles
- * general lines, circles
- * convex polygons

Generalizes to higher dimensions
- "sweep hyperplane"
- many applications

Near neighbor searching

Another possible addition to Search ADT:

```
.       void STinit();
.       void STinsert(recordType x);
. recordType STsearch(keyType v);
.        int STempty();
. recordType STnearest(keyType v);
```

Find the record with key value closest to v

Need a concept of "distance", not just "less"
easy if keys are numbers, or points in space

Implementations

- ARRAY: do binary search on v
  - compare with points on either side
- HASH TABLE: no easy algorithm
- BST, TRIE: recursive traversal works

1D BST near neighbor searching

Test search value for proximity with every node touche

Go down the both subtrees if close to the edge

```
void BSTnear(struct node *t)
   {
      if (t == z) return;
      if (dist(v, t->key) < min)
         { best = t; min = dist(v, best->key); }
      if (v < t->key || (v - t->key) < min)
         BSTnear(t->l);
      if (v > t->key || (t->key - v) < min)
         BSTnear(t->r);
   }
```

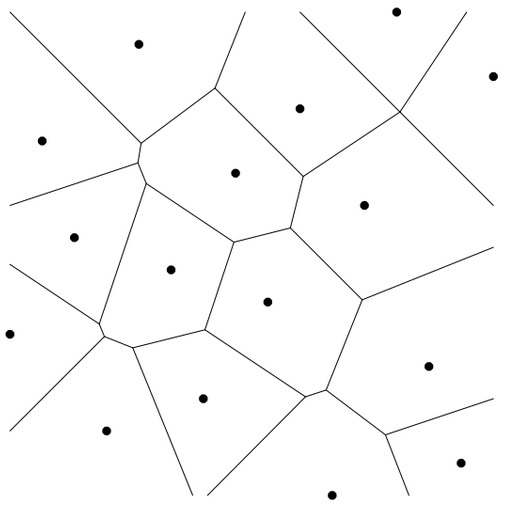
Multidimensional near neighbor searching:

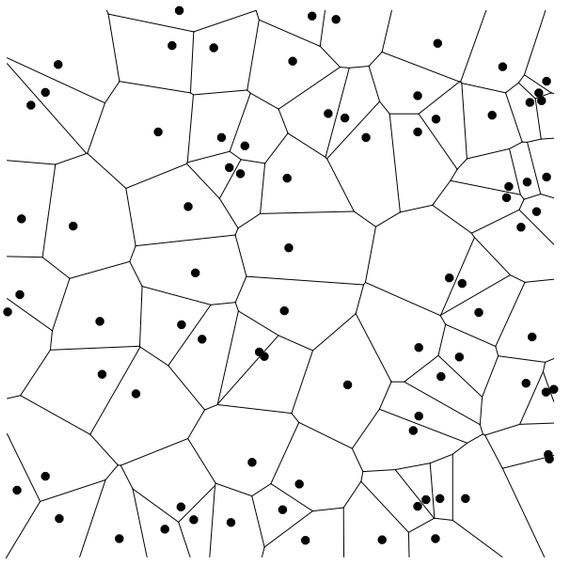
same algorithm, but switch dimensions
at each level, just as for range searching

Voronoi diagram

Given: set S of N points

VORONOI DIAGRAM: for each point x, identify
the set of points in the plane closer to x
than to any other point y in S





Ultimate "closest point" structure

Voronoi edges
perpendicular bisectors of point pairs

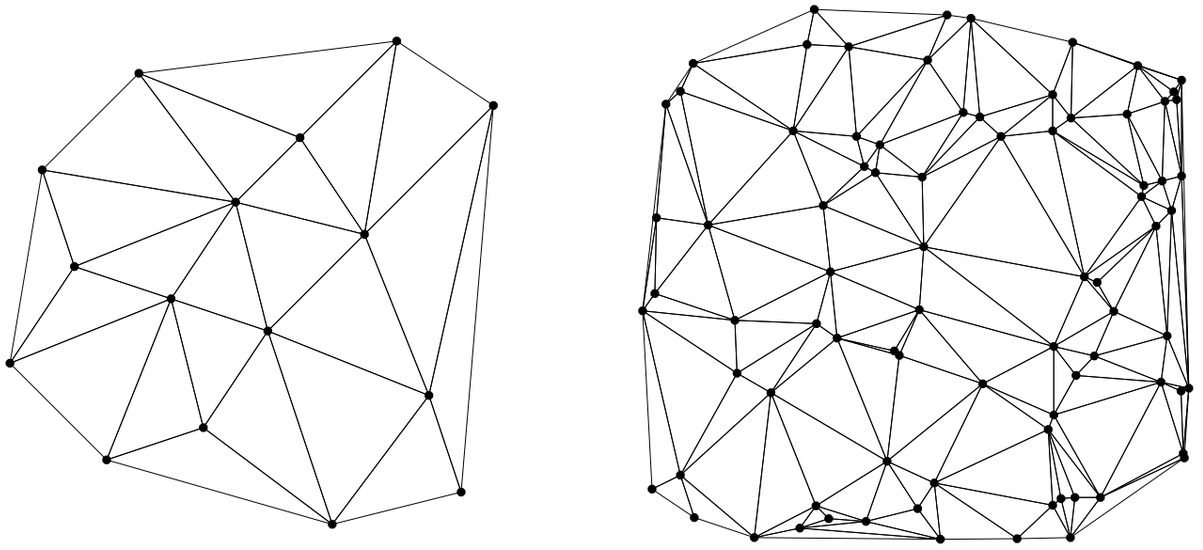
Voronoi points
centroids of point triple triangles

Challenge to compute
Representation?
Degenerate cases?

Delaunay triagulation

Given: set S of N points

DELAUNAY TRIANGULATION: connect x to
every point y in S with a Voronoi edge
separating x and y



Outer boundary is convex hull
Representation easier: no extra points

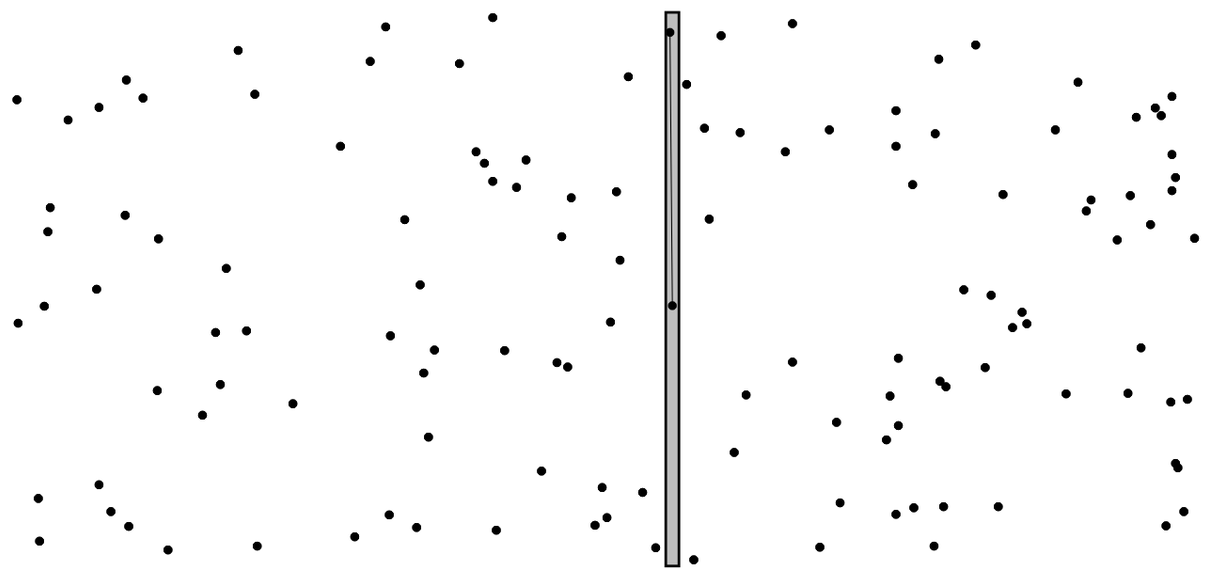
Voronoi diagram and Delauney triangulation
can be computed in N log N steps (!!)

Approaches
* divide and conquer
* sweep line
* randomize
* discretize

2D divide-and-conquer

Ex: To find the closest pair in N points

* sort on x
* divide into two sets of N/2 points
* find closest pair in each half
* find closest pair crossing boundary



After pairs in halves found, checking
boundary crossing seems easy
Actually need to sort on y (comes for free)
to make boundary check easy
Boundary check MUST be carefully done
because it terminates recursion

2D divide-and-conquer code

Use the same code (mergesort) to

* sort on x
* sort on y and work on boundary

Call this twice (change key from x to y):

```
struct node *sort(struct node *c, int N)
{ int i; struct node *a, *b; float middle;
   struct point p1, p2, p3, p4;
   a = c; if (c->next == z) return c;
   for (i = 2; i <= N/2; i++) c = c->next;
   b = c->next; c->next = z;
   if (pass == 2) middle = b->p.x;
   c = merge(sort(a, N/2), sort(b, N-(N/2)));
   if (pass == 2)
   {
      p1 = z->p; p2 = z->p; p3 = z->p; p4 = z->p;
      for (a = c; a != z; a = a->next)
         if (fabs(a->p.x - middle) < min)
         { check(a->p, p1); check(a->p, p2);
            check(a->p, p3); check(a->p, p4);
            p1 = p2; p2 = p3; p3 = p4; p4 = a->p; }
   }
   return c;
}
```

Same general strategy works to compute Voronoi

Grid methods

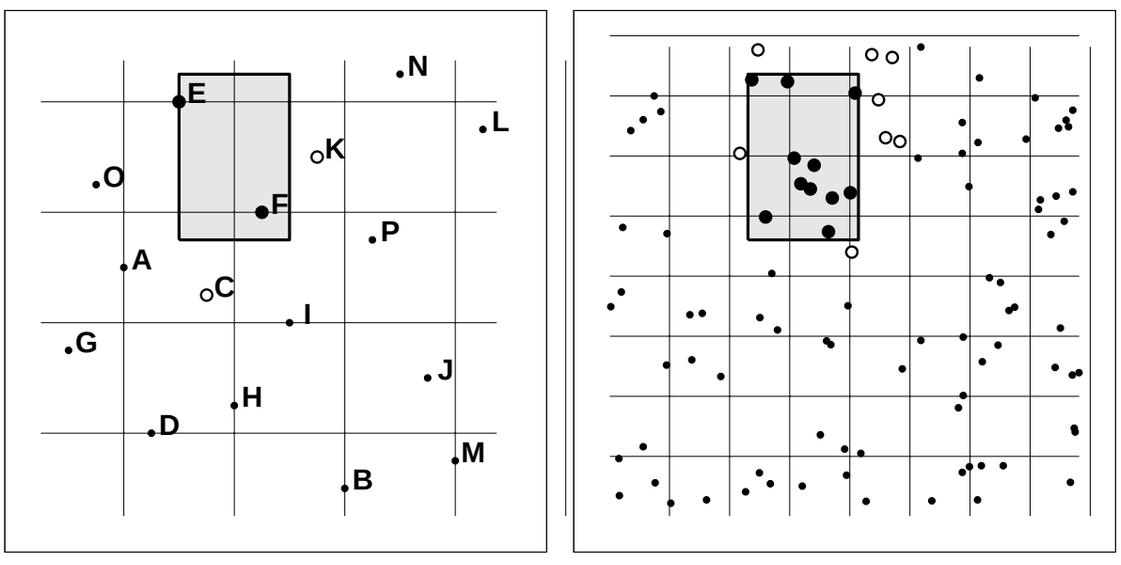
Grids : geometric search :: tries : search ADT

Grid method
* define uniform grid of fixed-size squares
* put points in lists associated with squares
* ignore points in faraway grid squares

Time-space tradeoff like MSD sort
* grid too fine: empty cells
* grid too coarse: lists too long

Use 2- or 3-level grids, or recurse ala quad trees

Ex: range searching



For graphics applications
ultimate grid is PIXEL ARRAY
leads to "discretized algorithms"

Point location problem

Ex: find state corresponding to point on map

Planar subdivision

* 2D tree planar decomposition
* N lines
* Voronoi diagram
* grid

Which division contains the given point?

Difficult in general

if only because of difficulty of representing planar subdivisions

Discretized line intersection

p-by-p bit raster, p^2 pixels

N lines

Draw rasterized verson of line
    report intersection if pixel already 1

Cost:
    p^2 to initialize pixels to 0
    number of pixels on lines

Cost dominated by p^2

Line intersection same cost
    as drawing blank picture!

Discretized Voronoi diagram

put 1 pixels on a priority queue
priority: distance to closest point

Algorithm:
* remove pixel from priority queue
* check all neighbor pixels
    * closer or same: ignore
    * farther: check pixel value
        * if 0, set to 1 and put back on pq
        * if 1, must be on a voronoi edge!

Time proportional to
    number of pixels on diagram
            times
    diameter of largest cell
    (plus time to initialize pixels to 0)

Idea:
    use or refine triangulation from
    discretized diagram to compute real
    diagram