Competitive Programming

From Problem 2 Solution in O(1)

Computational Geometry

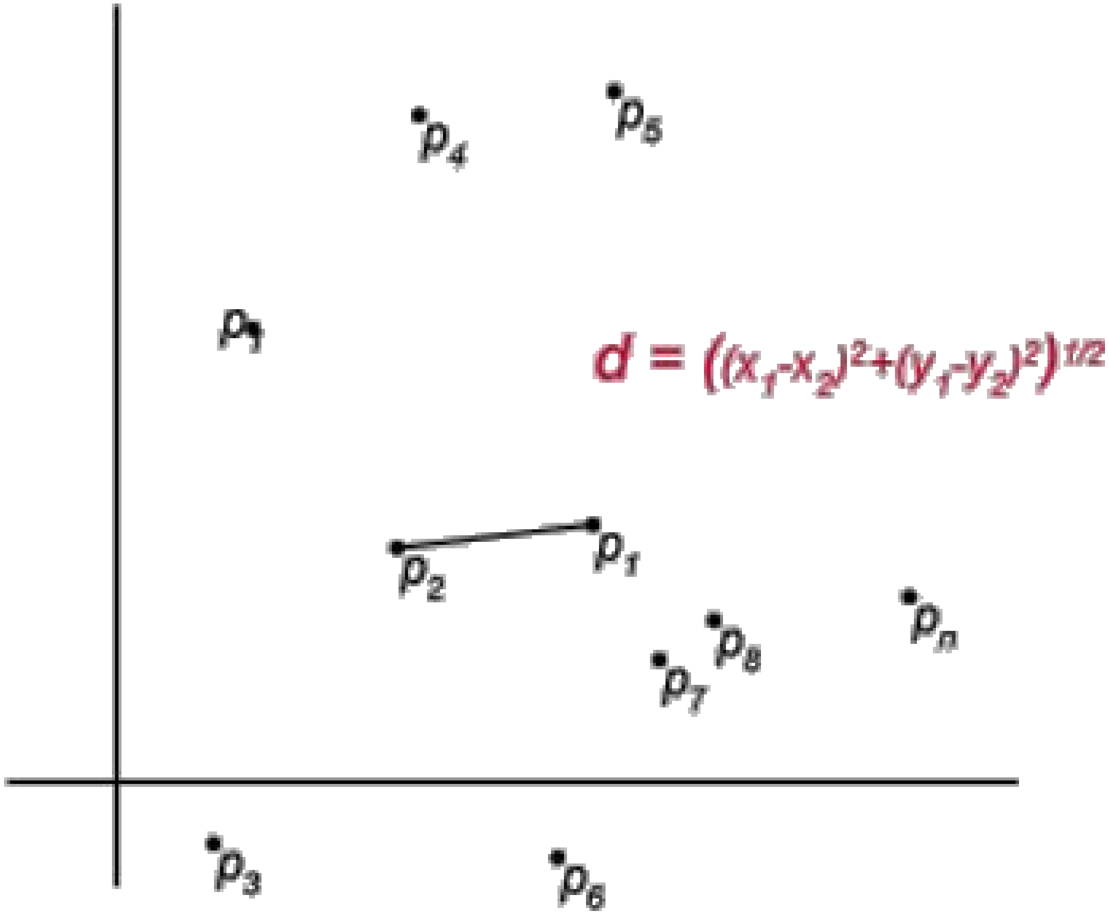
Line Sweep - Closest Pair

Mostafa Saad Ibrahim
PhD Student @ Simon Fraser University

Line Sweep

- Giving a plane of objects and some task
  - E.g. Set of segments to intersect
  - E.g. Set of rectangles to compute union
- One can consider every pair of objects!
  - However, every object (e.g. segment) interacts (e.g. intersect) with some surrounding ones **NOT** ALL
- Imagine a **vertical line** that is **swept** across the plane, specially **at discrete points** (events)
  - E.g. start/end of segment or a rectangle
  - We will use them to identify the **surrounding objects**

Closest Pair Problem



Given N points

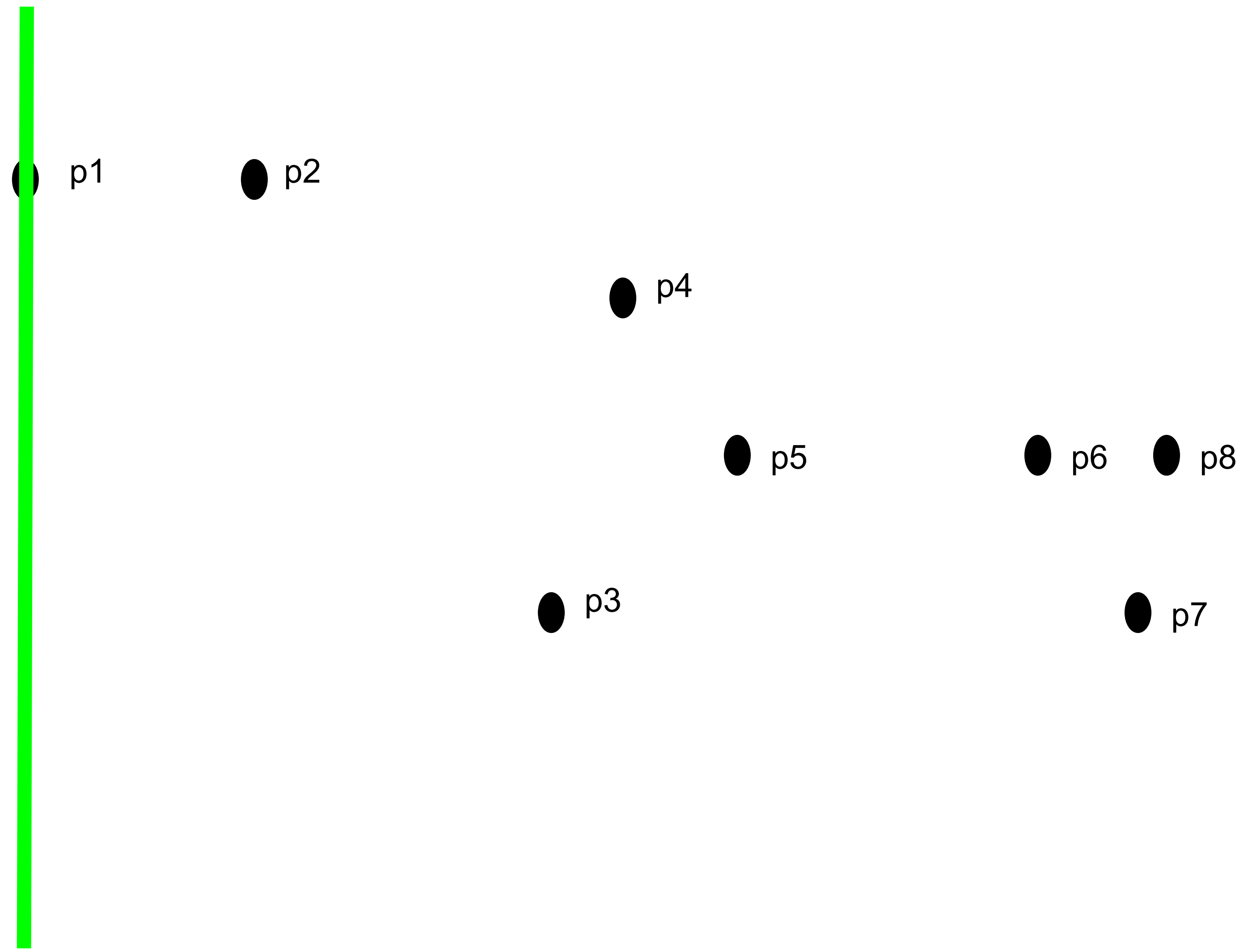
Find the closest pair of points among them using Euclidean Distance

Src: http://ramos.elo.utfsm.cl/~lsb/elo320/aplicaciones/aplicaciones/CS460AlgorithmsandComplexity/lecture1/COMP460%20Algorithms%20and%20Complexity%20Lecture%201.htm

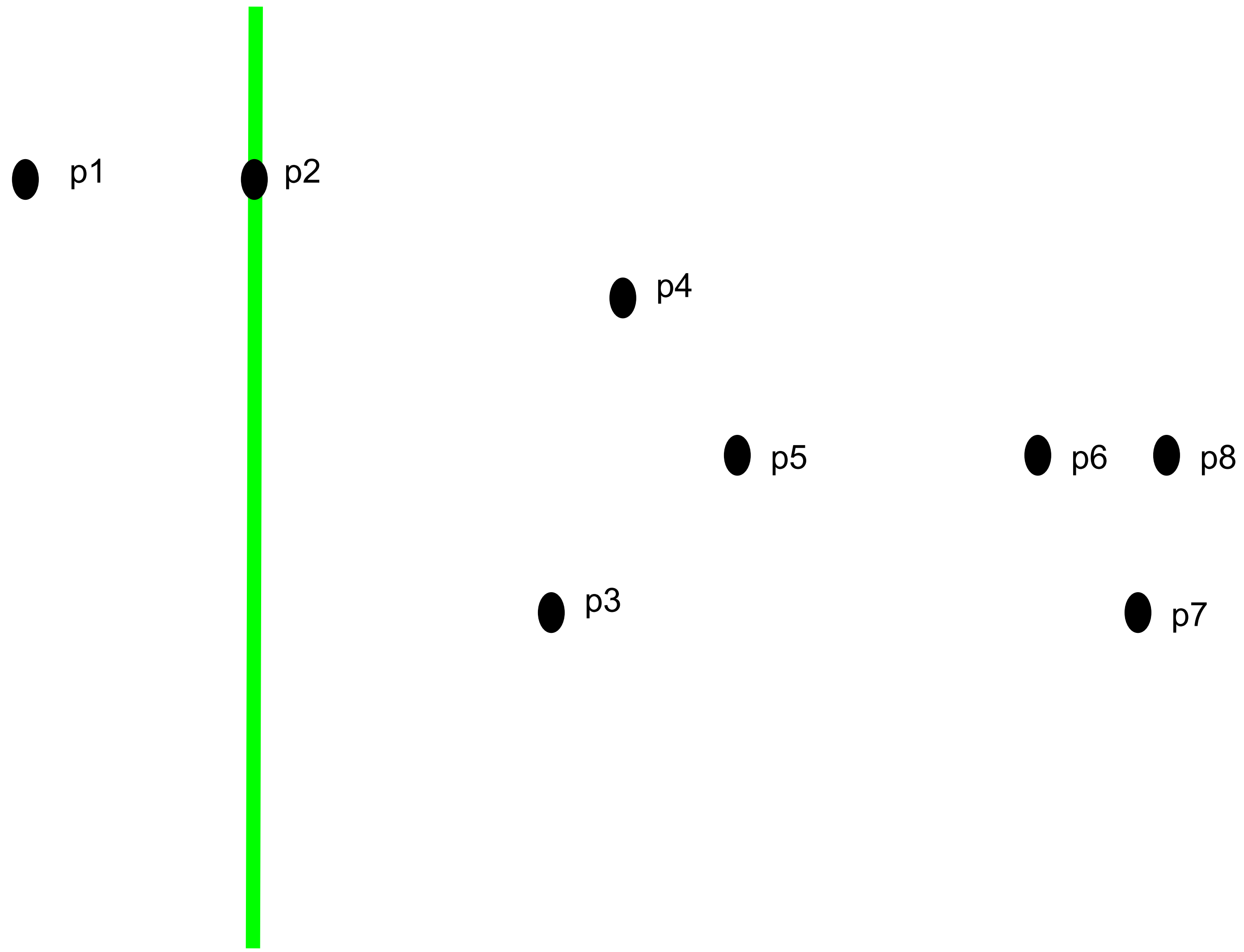
Closest Pair Problem

- A trivial approach is brute force O(n^2)
  - For every pair of points compute distance
  - Minimize over them
- Divide and Conquer Solution: O(nlogn)
  - Can be generalized to N-Dimensions
- Sweep Line Solution O(nlogn)
  - **Today Session**
  - Given a point, should we really consider every other point? Or jsut subset of them?
  - Vertical **Sweep line**...Points are the **events**
  - **Active Window** is based on current shortest distance

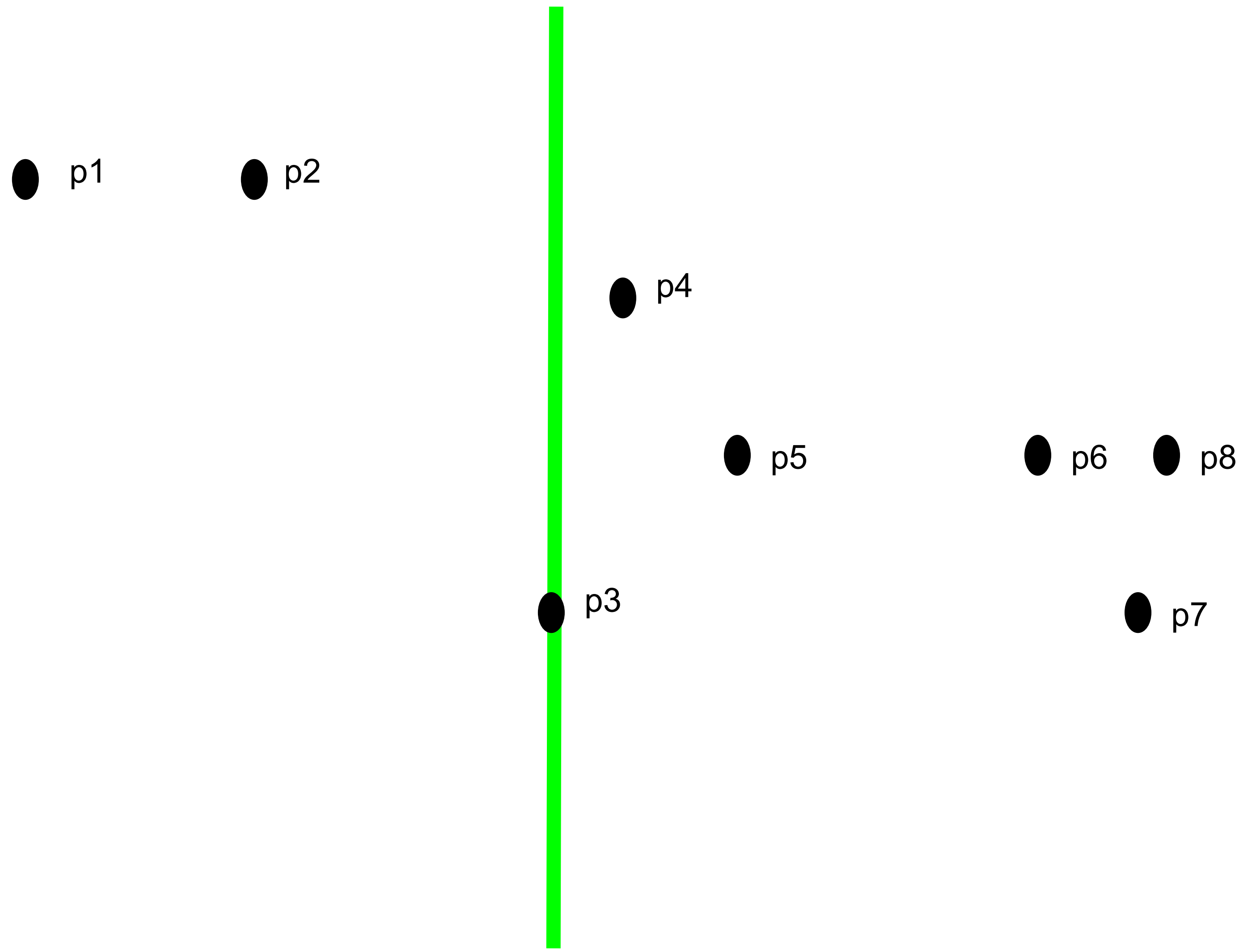
Vertical Sweep Line



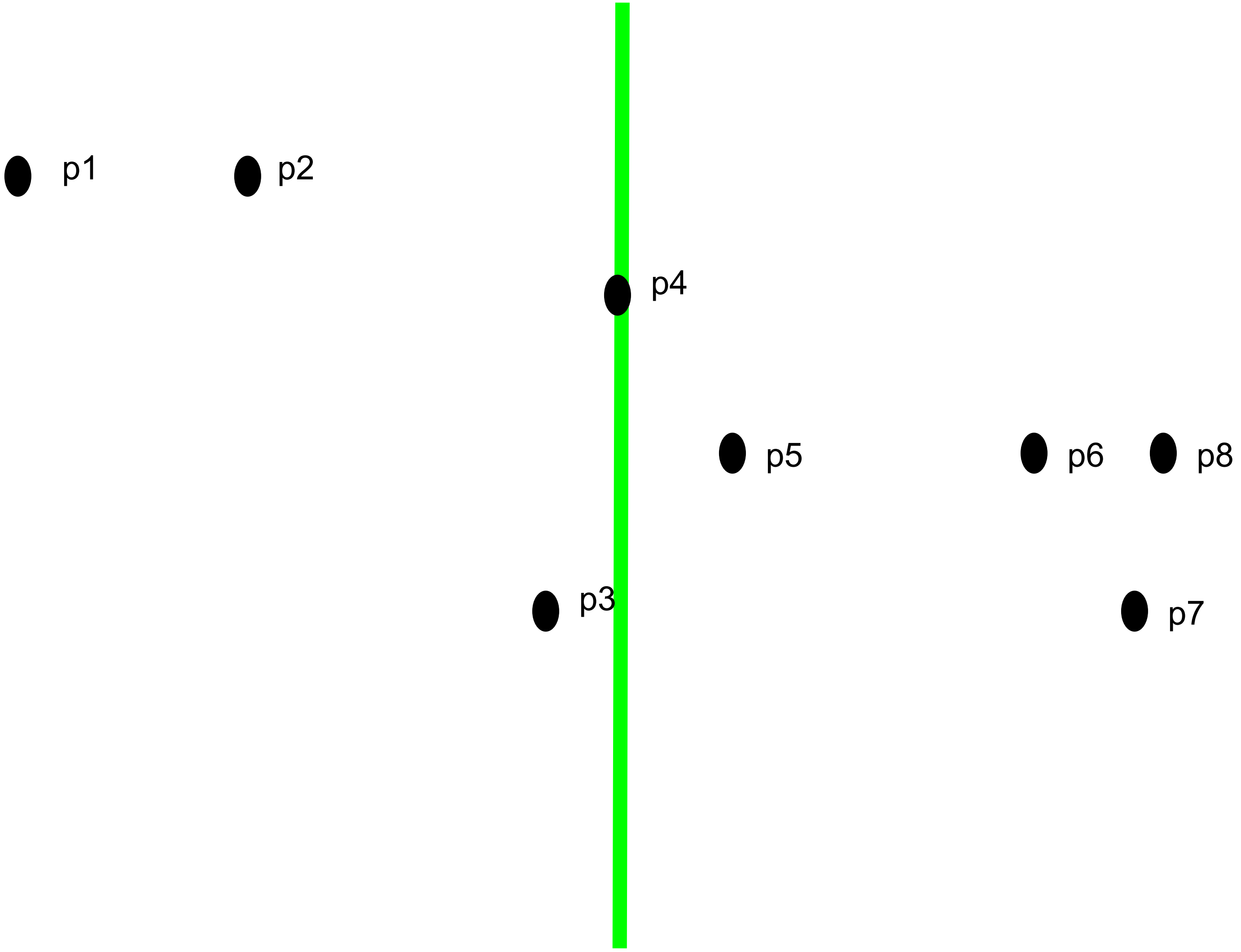
Vertical Sweep Line



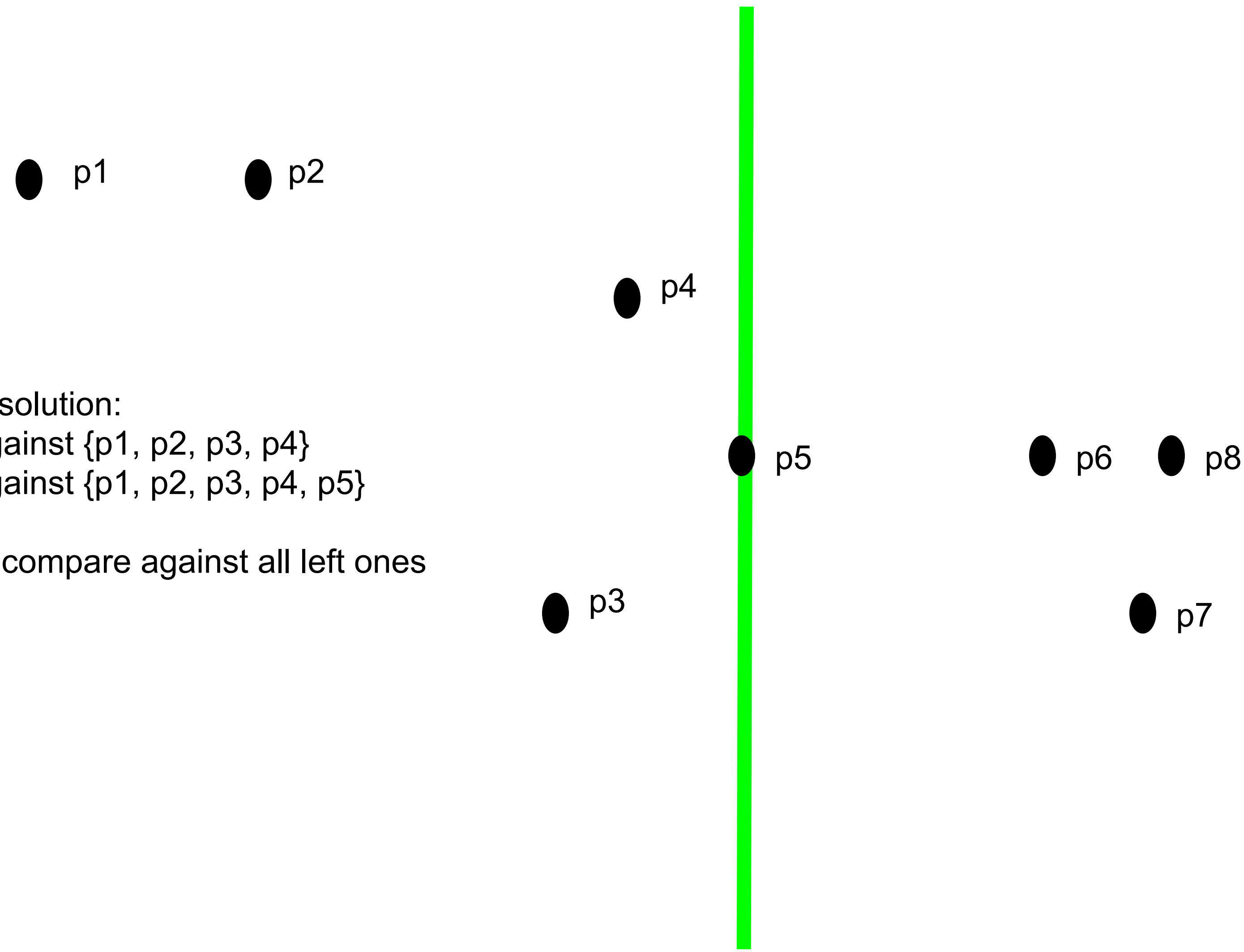
Vertical Sweep Line



Vertical Sweep Line



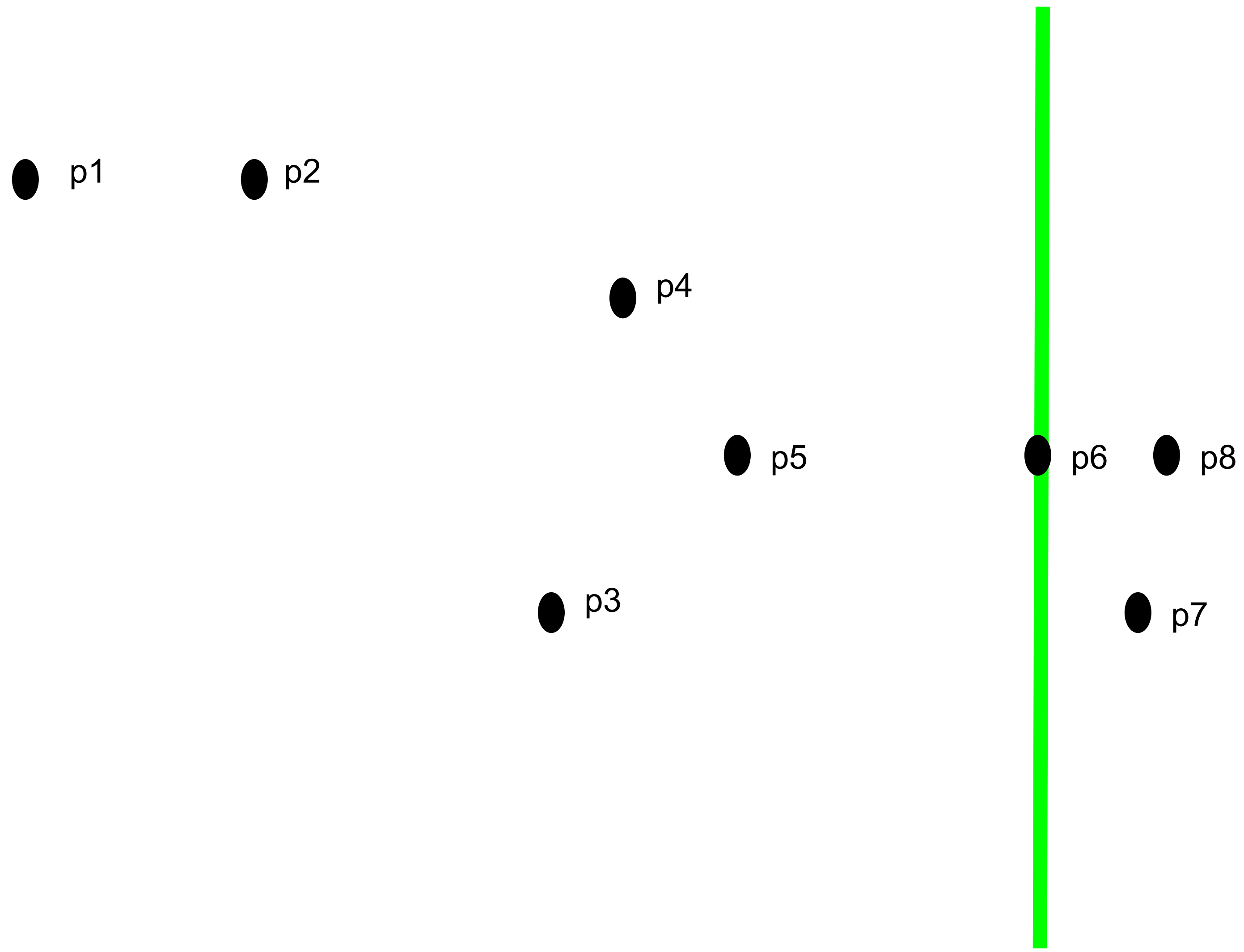
Vertical Sweep Line



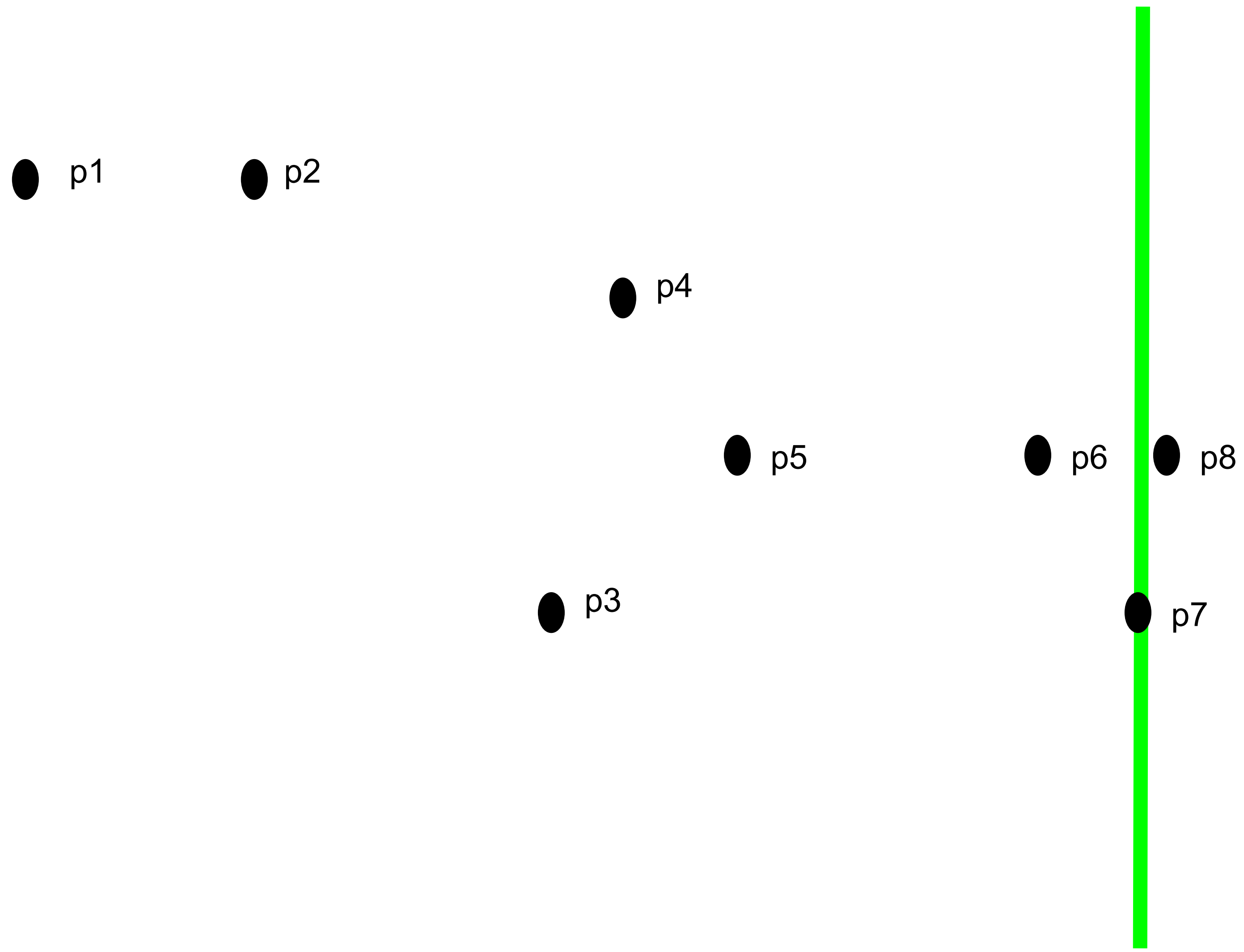
Note
In a brute force solution:
p5 compares against {p1, p2, p3, p4}
p6 compares against {p1, p2, p3, p4, p5}

e.g. every point compare against all left ones

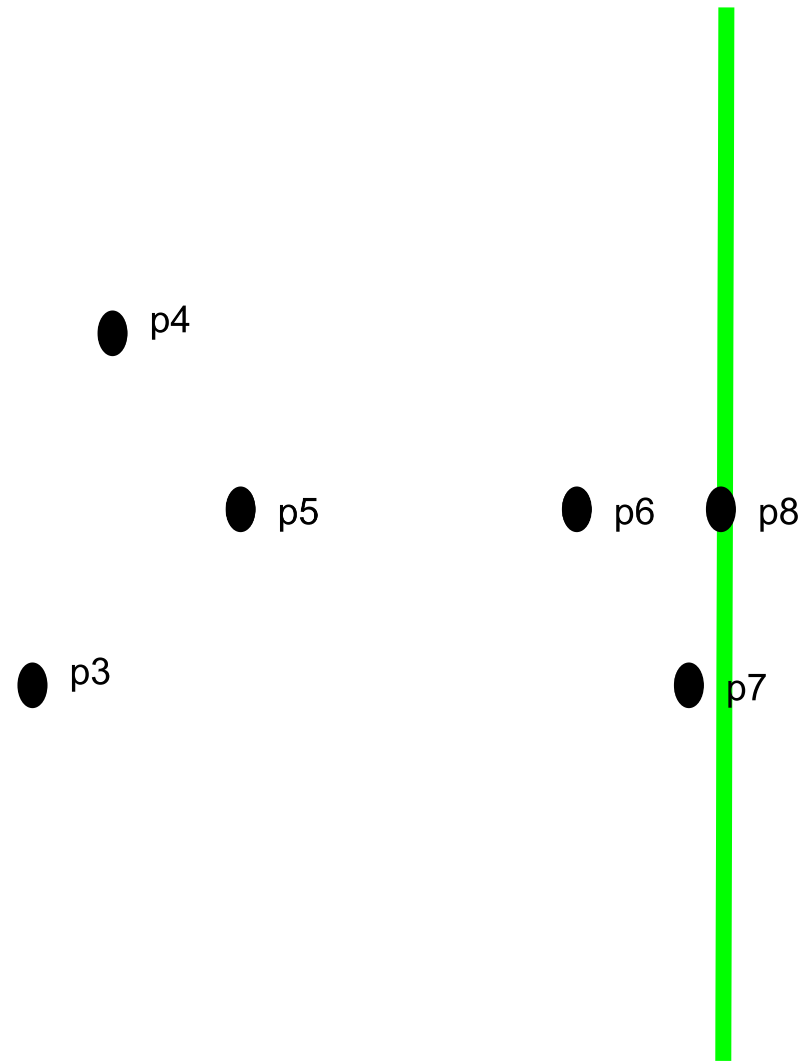
Vertical Sweep Line



Vertical Sweep Line



Vertical Sweep Line



Assume we reached some point e.g. p8
should we consider all the previous points?

No, few subset only (this is called **active set**)

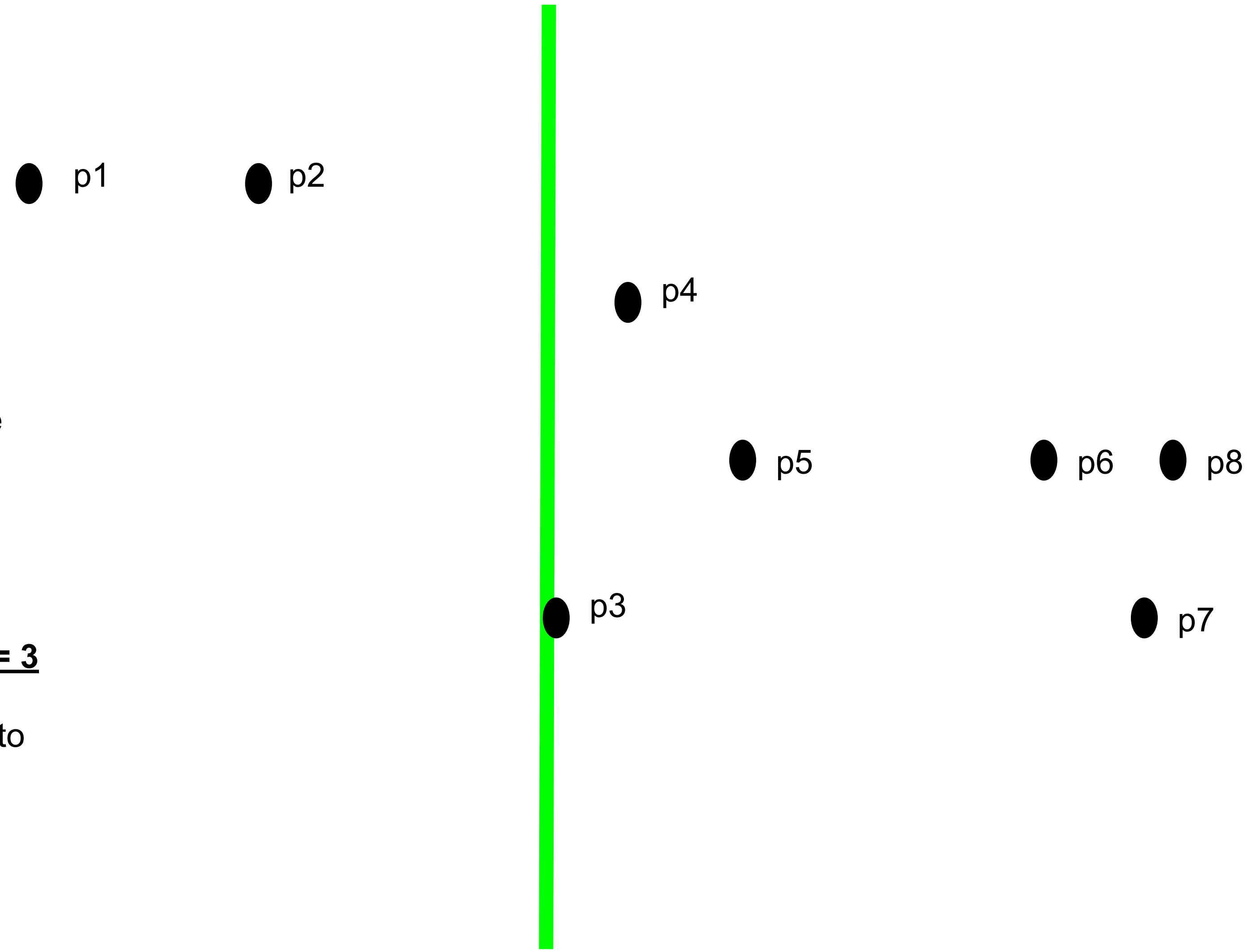
Like what? May be only p6 and p7?

Based on what? **Best distance so far**
e.g. best dsistance is (p4, p5) 2.5
So consider only points far with 2.5 units far from p8

Vertical Sweep Line

- We are sweeping on points from left to right
- Each point may update the shortest D so far
- How to compute your **active window**?
  - Considering every left point is O(n^2)!
- Assume current best distance is 5
- Then next point P only need to consider **half circle** centered at p to get points <= 5
  - Note, just ½ circle NOT full one
  - We r sweeping from left to right, we only know left points
  - Note, rectangle is almost a circle + little parts

Active window: Half Circle from P

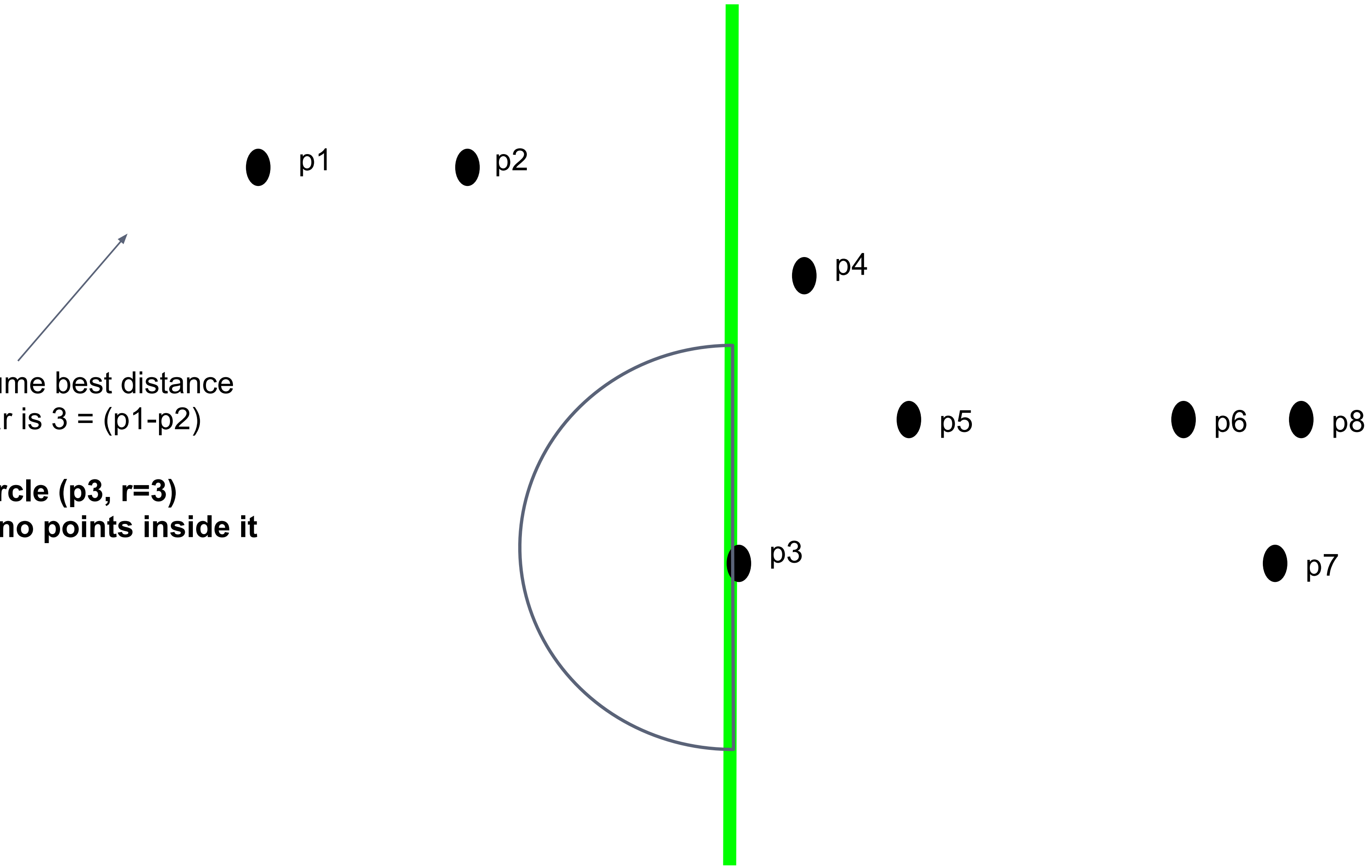


Assume best distance so far is 3 = (p1-p2)

Now we are at p3
p3 should see if any points are within
<u>half circle of radius = 3</u>

if any, see if distance to any < 5

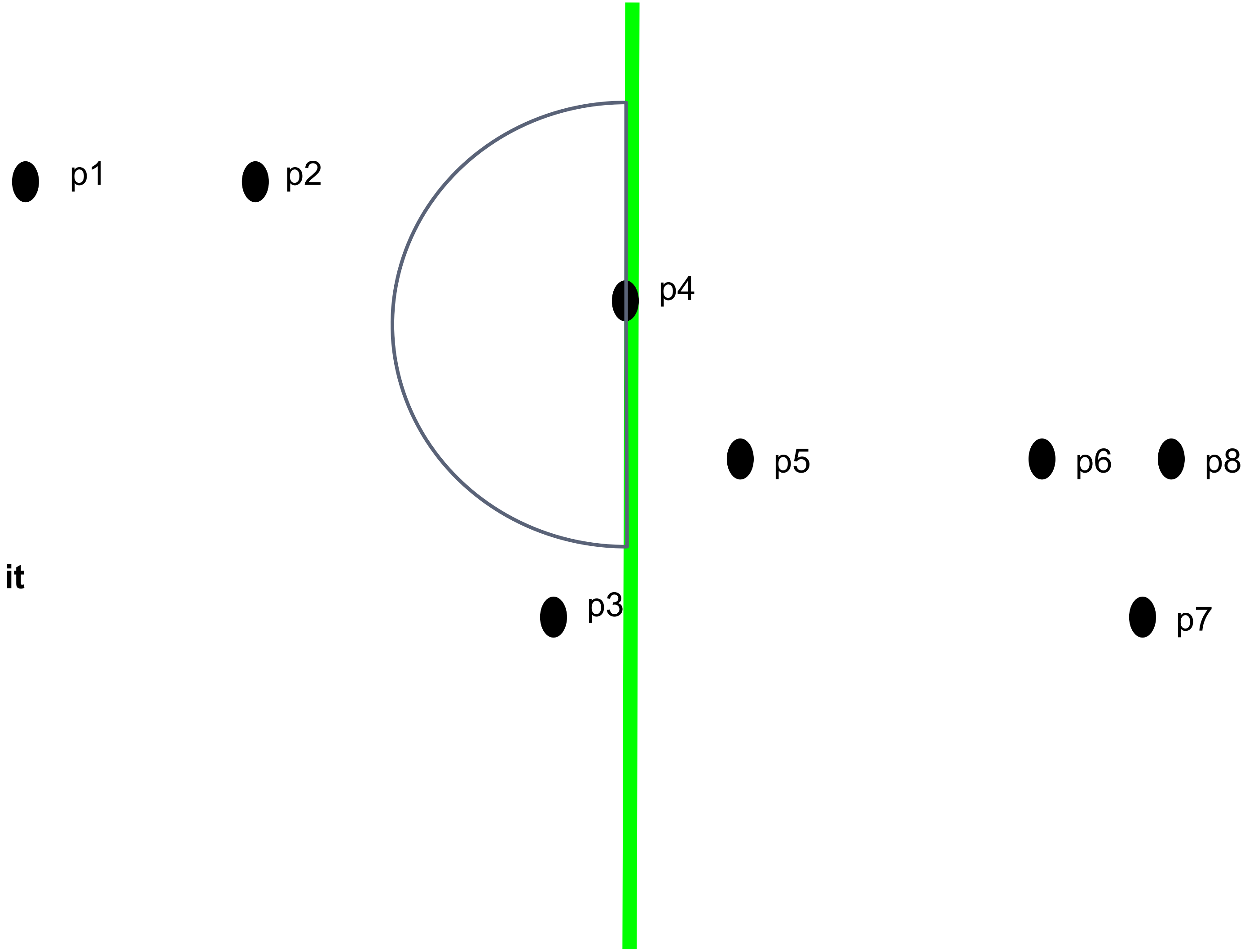
Active window: Half Circle from P



Assume best distance
so far is 3 = (p1-p2)

½ circle (p3, r=3)
has no points inside it

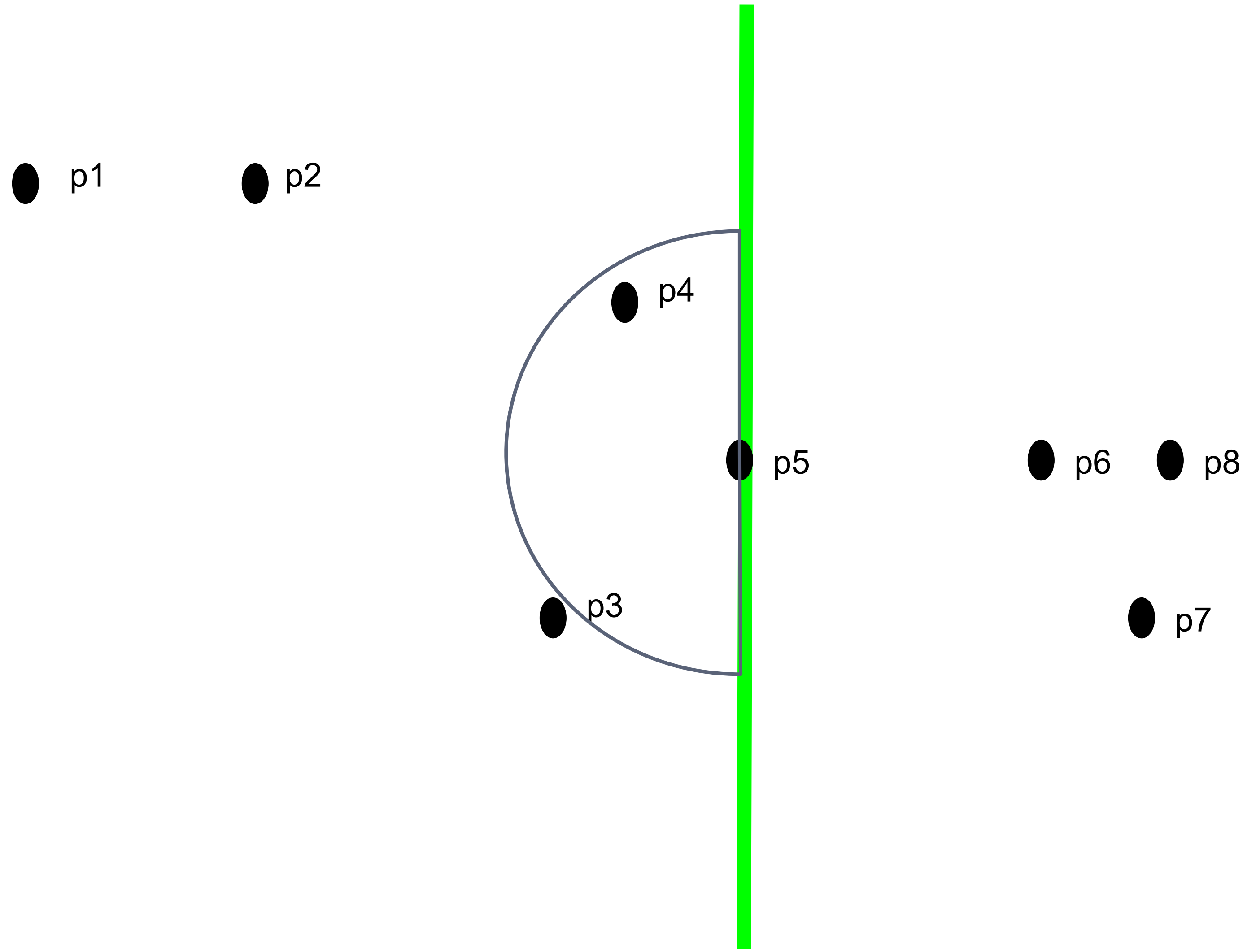
Active window: Half Circle from P



Assume best distance so far is 3 = (p1-p2)

½ circle (p4, r=3) has no points inside it

Active window: Half Circle from P

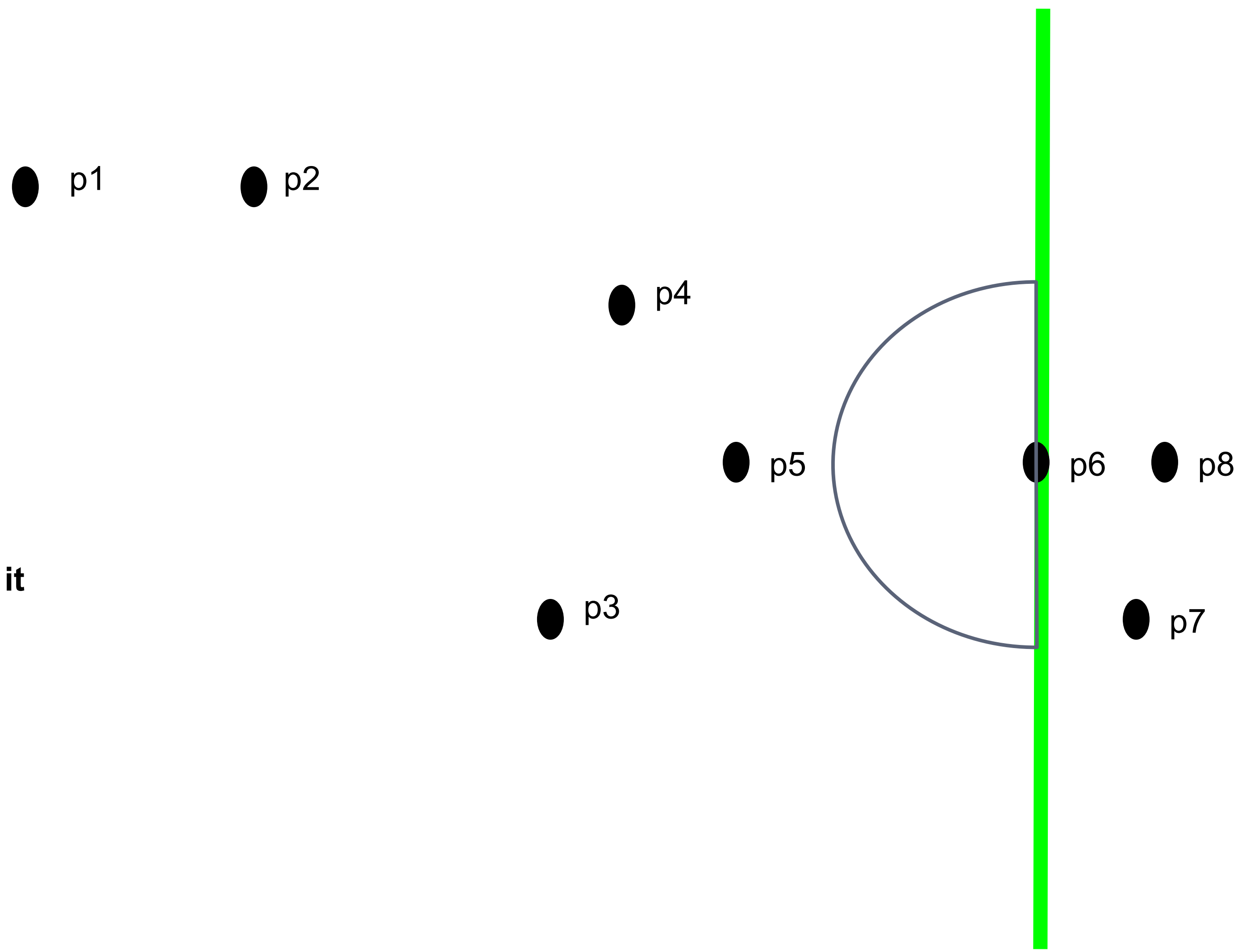


Assume best distance
so far is 3 = (p1-p2)

½ circle (p5, r=3)
has point p4
Dist(p5,p4)=2.5

Update bestD = 2.5

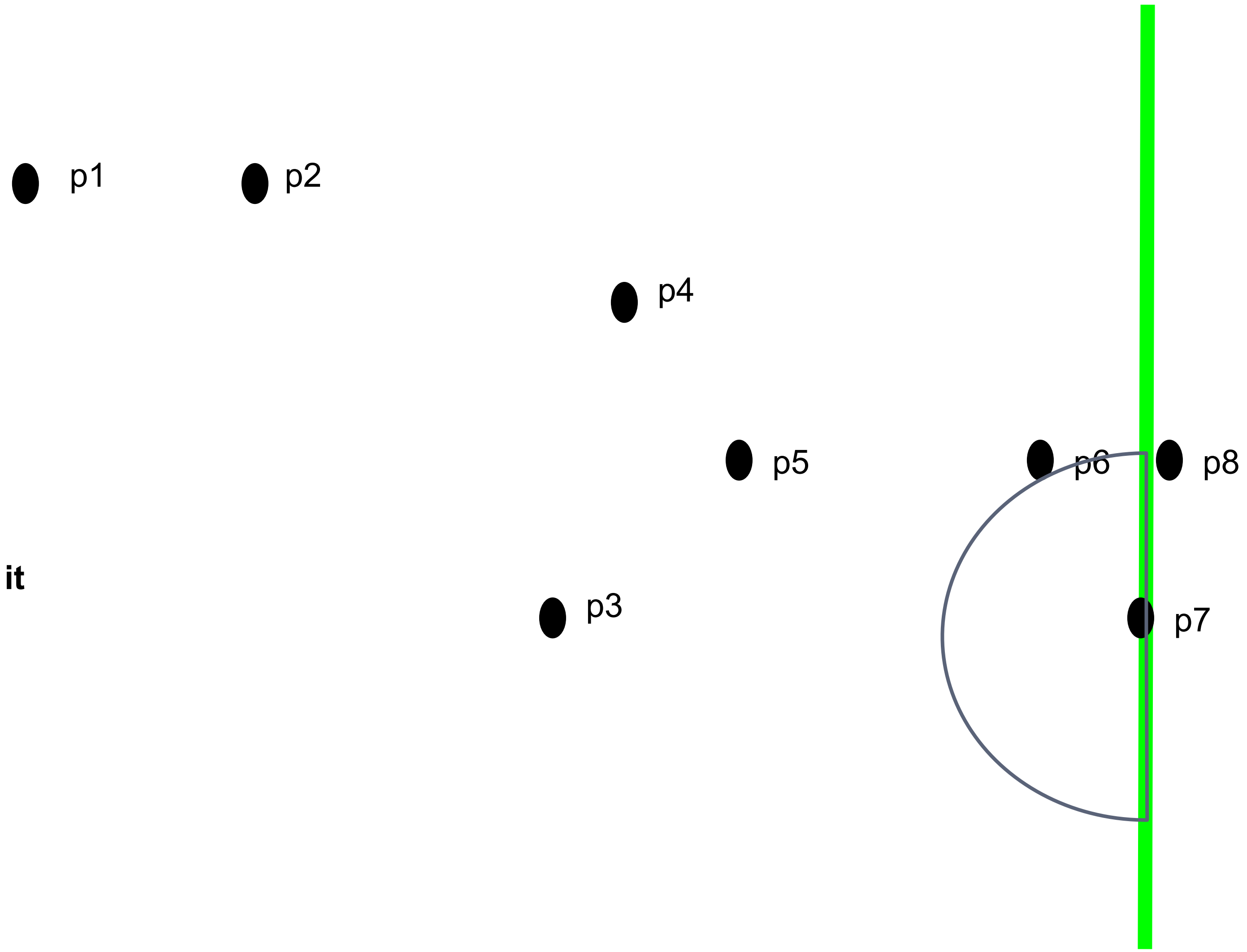
Active window: Half Circle from P



Assume best distance
so far is 3 = (p1-p2)

½ circle (p6, r=2.5)
has no points inside it

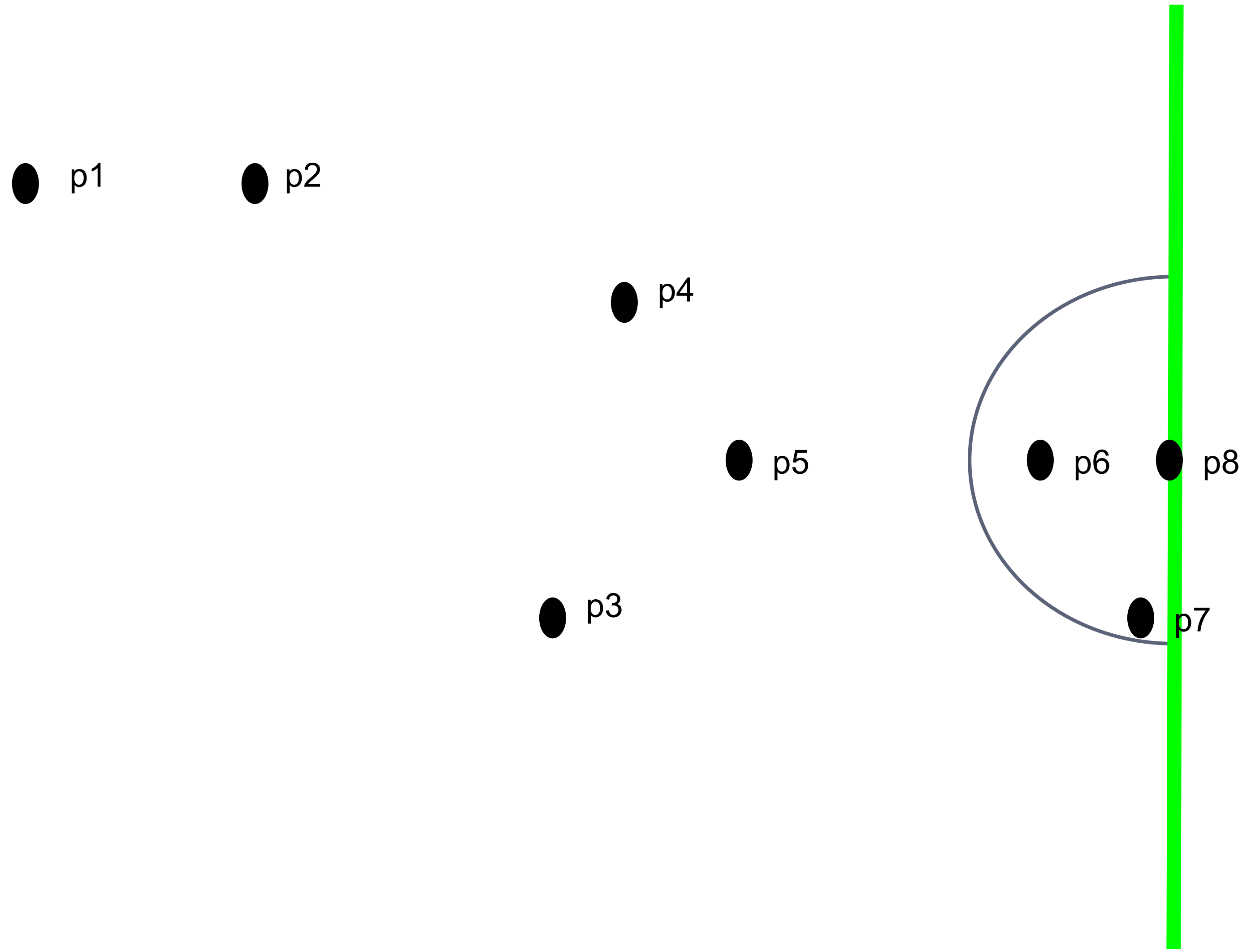
Active window: Half Circle from P



Assume best distance
so far is 3 = (p1-p2)

½ circle (p7, r=2.5)
has no points inside it

Active window: Half Circle from P

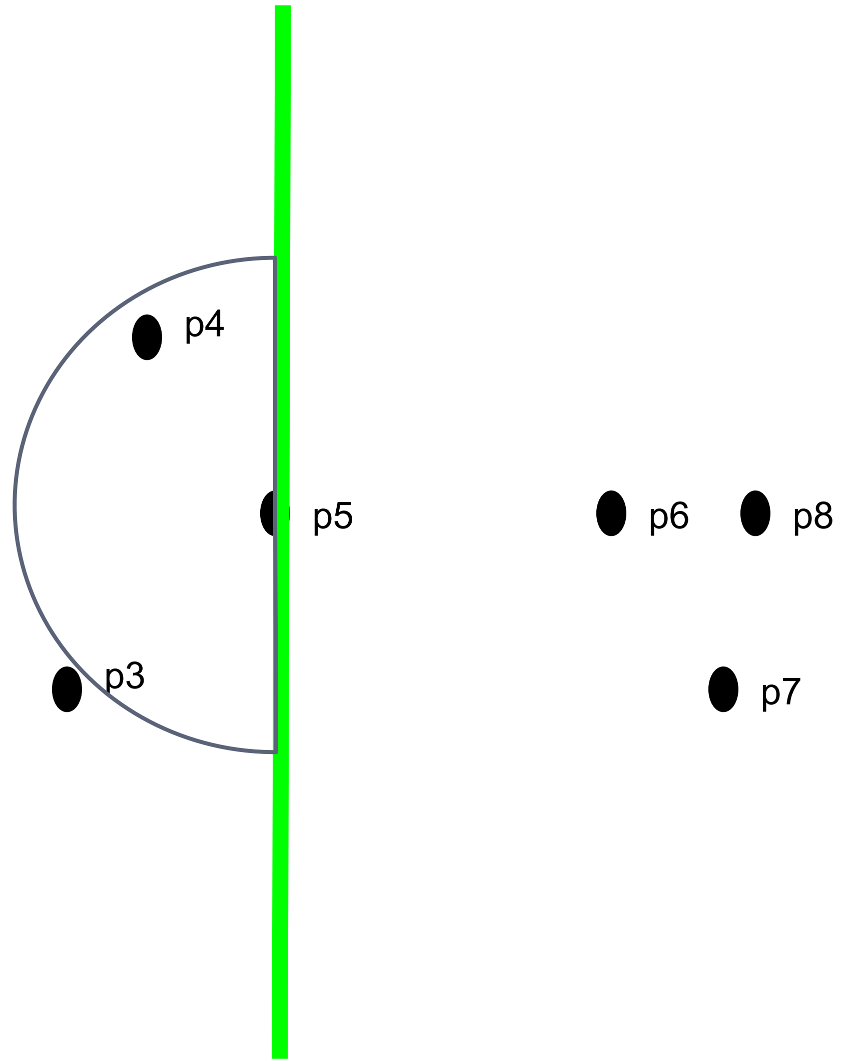


Assume best distance
so far is 3 = (p1-p2)

½ circle (p8, r=2.5)
has 2 points: p6, p7
Dist(p8,p6)=1.3
Dist(p8,p7)=2.2

Update BestD = 1.3

Half Circle vs Half Rectangle



The minimum shape to get closest euclidean distance is circle

But efficient active window based on ½ circle may be hard?

What about a half rectangle: r x 2r ?

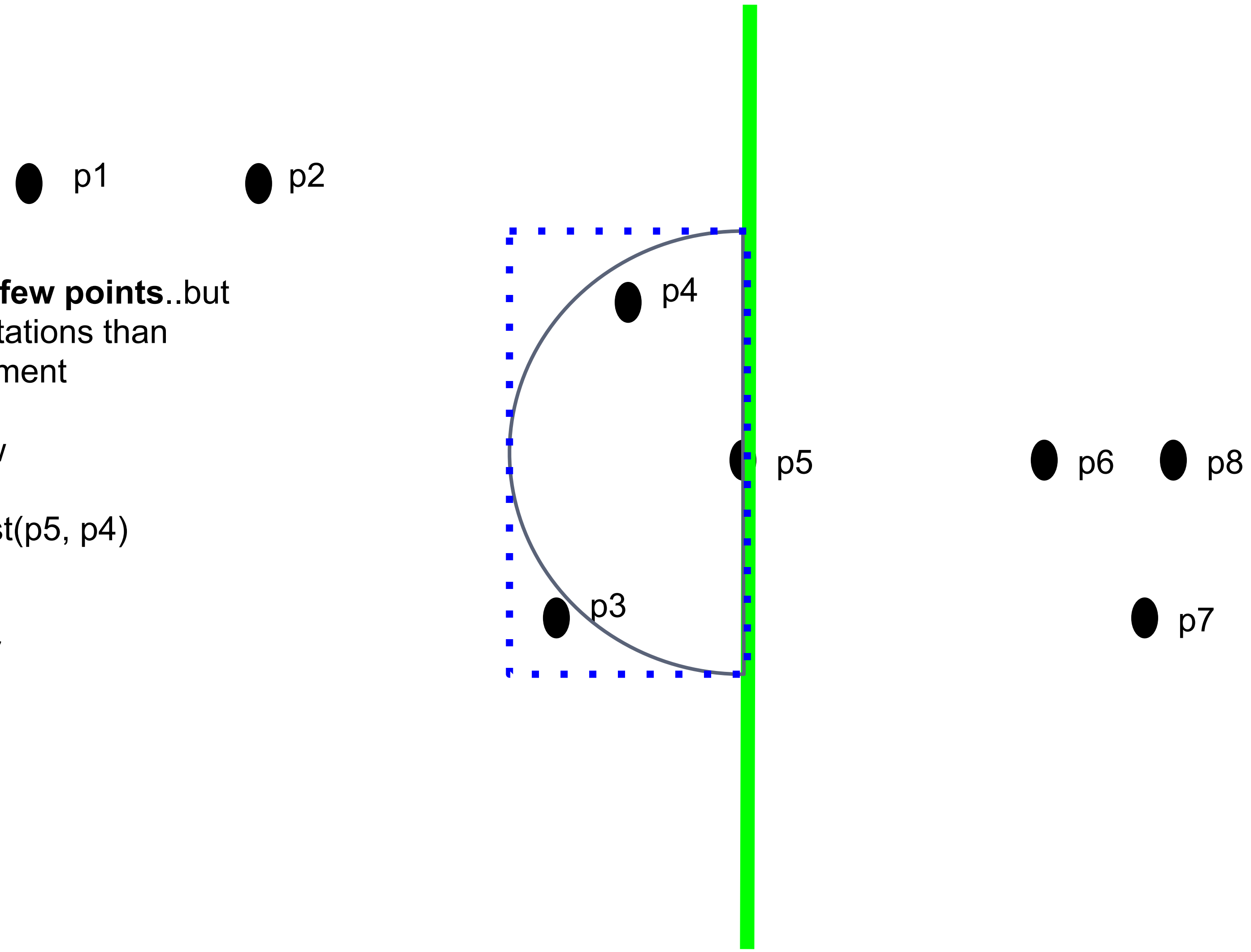
Half Circle vs Half Rectangle

It might include **extra few points**..but given its trivial computations than circle = great achievement

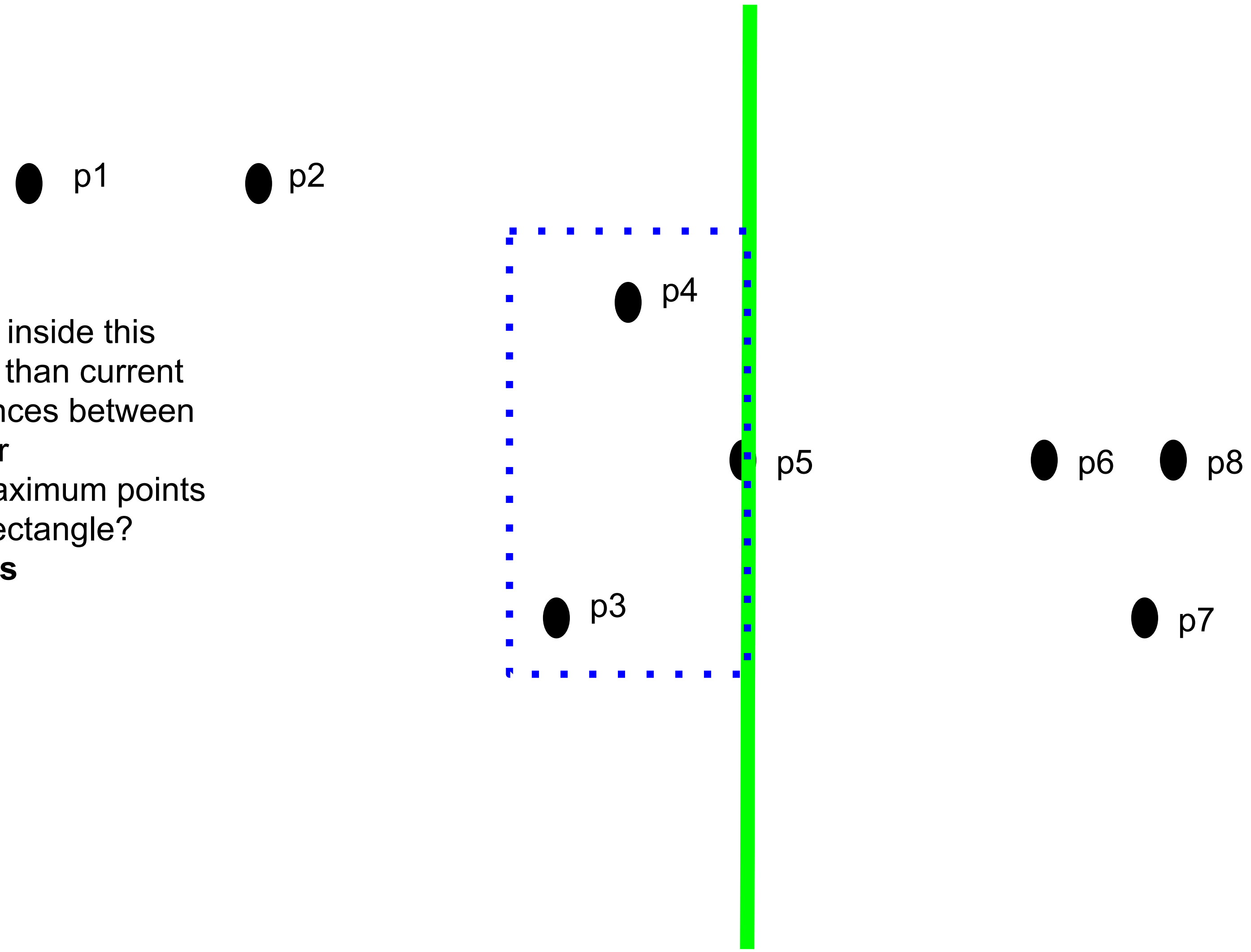
Now p5 active window
has both p4, p3
Note dist(p5, p3) > dist(p5, p4)

This rectangle is r * 2r

e.g. 3x6 rectangle



Half Rectangle



Observation:

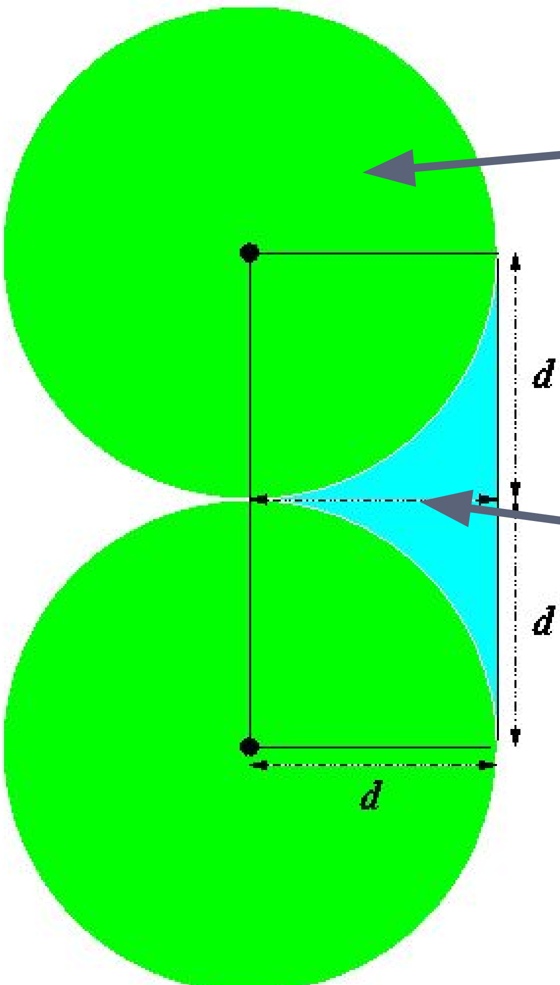
- We know points inside this rectangle (other than current point) has distances between themselves >= r
- What are the maximum points inside the r*2r rectangle?
- **at most 6 points**

Max points in the rectangle

- **Theorem**: A rectangle of width d and height 2d can contain **at most six points** such that any two points are at distance at least d
  - Divide dx2d to 2 rectangles: dxd (think 6 corners)
  - Add points in the current 6 boundaries
  - Draw circle around each point of length D (others can’t be inside the circle, only on its boundary)
  - Move the points wherever and try to add further points
  - You can’t …
  - hence maximum 6 points

Max points in the rectangle

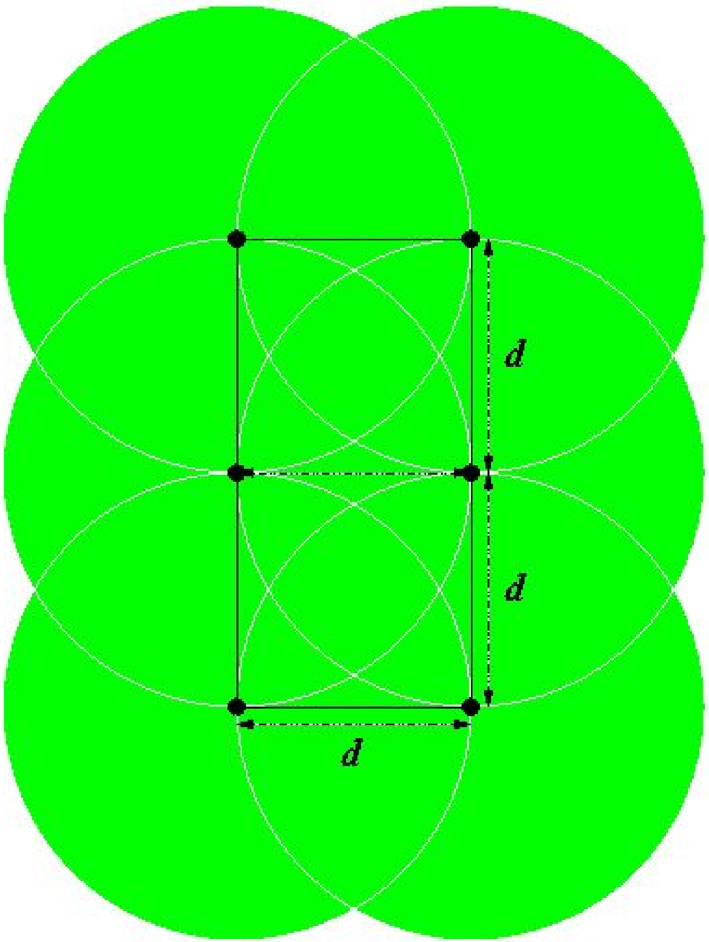
Other points shouldn't be inside the circle



We can add/move in this space

Src: http://www.cs.mcgill.ca/~cs251/ClosestPair/proofbox.html

Max points in the rectangle



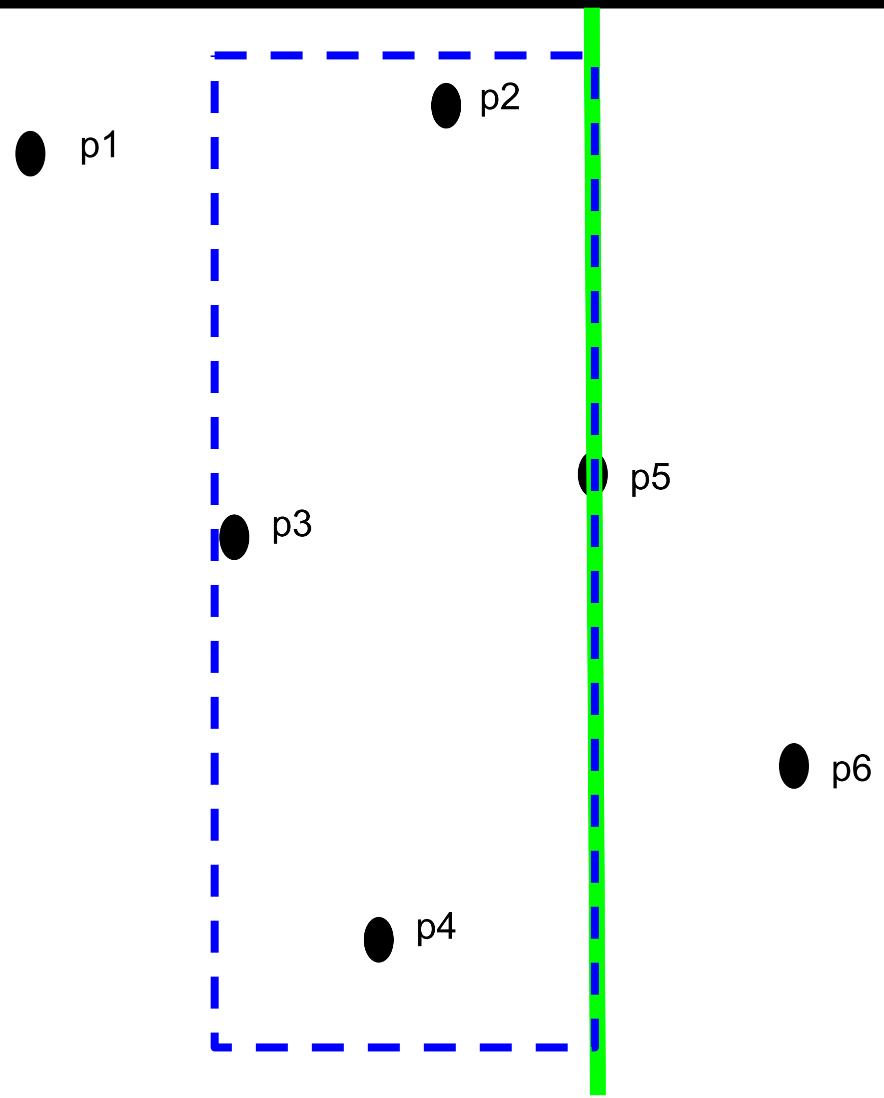
- Put 6 points on corners
- Draw 6 circles to limit others inside
- All space now **covered**
- One can’t add/move without being less than D

Src: http://www.cs.mcgill.ca/~cs251/ClosestPair/proofbox.html

Implementation

- Sorting the events is direct (sort X axis)
- The challenge is in the active window
- Assume finishing processing at point p
  - So far best distance is d
  - Assume having list LY of points on left of p with **x difference** from p to any of these points is d
  - E.g. this set is rectangle of **width d, but hieght = OO**
- To move to next sweep point q
  - Remove any point in LY with **x difference** to q > d
  - Do processing, then add q
  - Note: This list may be large, but its width is d

Implementation: big rectangle

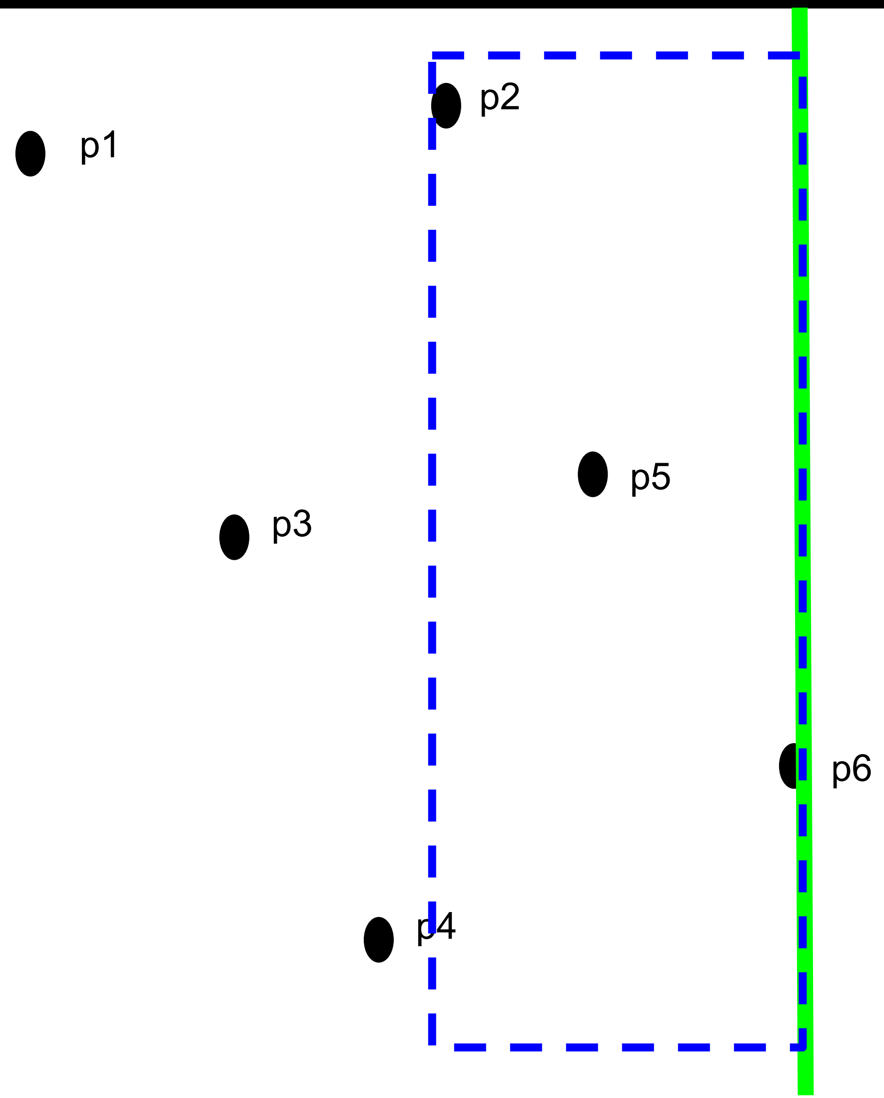


Assume we are done with p5
updated d = 4
LY = {p4, p3, p2} : sorted on Y

Now let's **move to p6**

- Filter LY
- Process it
- Add p6 to LY

Implementation: big rectangle



From p5
previous LY = {p4, p3, p5, p2}
previous d = 4
====

Dist(p4, p6) > 4 => remove
Dist(p3, p6) > 4 => remove
Dist(p6, p5) <= 4 => leave it
Dist(p6, p2) <= 4 => leave it

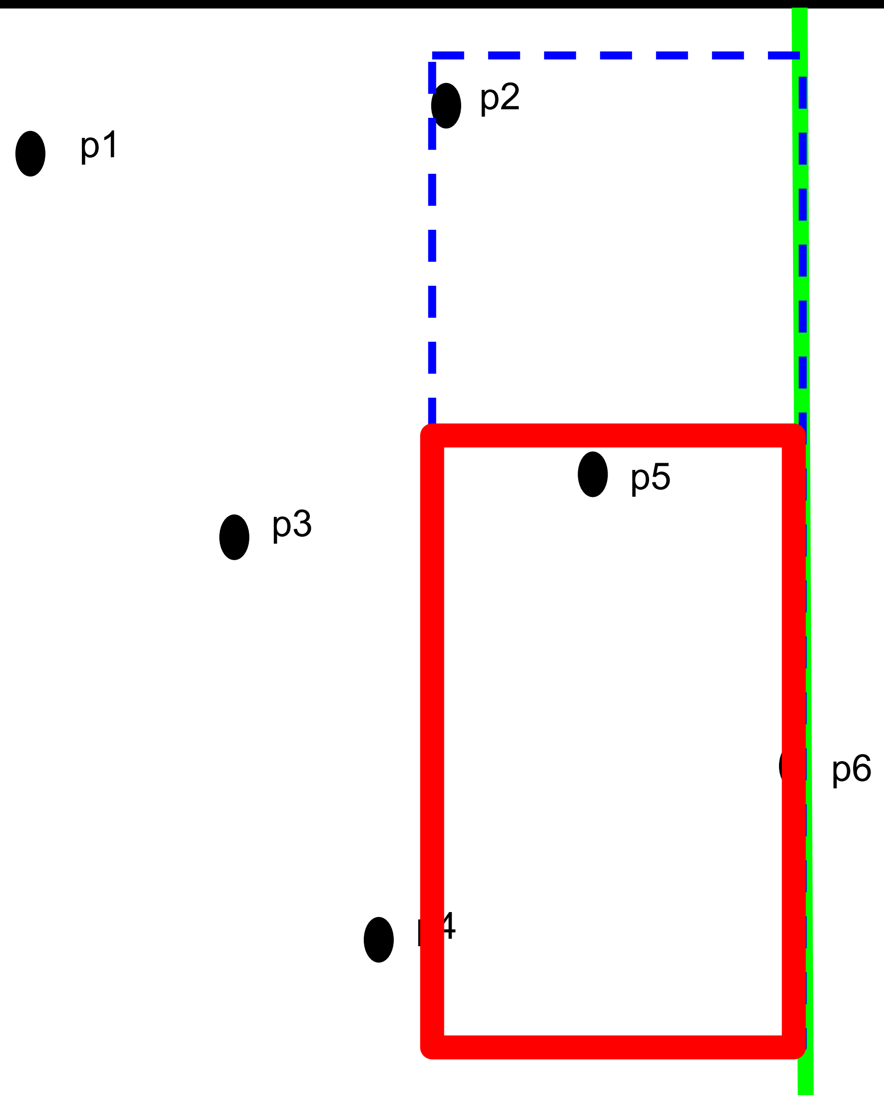
Updated L = {p5, p2}, d = 4
====

- Process P6
- Let new d = 3.5
- Add P6 to LY
- LY = {p6, p5, p2}
- Move to p7

Implementation

- Now we have current point q (q.x, q.y)
  - best distance d
  - points LY with x difference <=d: e.g. rectangle (d, OO)
- We need our active window?
  - Let LY sorted by its Y
  - Then binary search list to find range of (q.y-d, q.y+d)
  - That is d unit above q.y … and d unit below it
  - This window will have maximum 6 points
- In summary
  - Maintain the big rectangle of (d, OO) dimension
  - Binary search in the rectangle to find the (d, 2d) rectangle

Implementation: small rectangle



Recall at p6
- Updated LY = {p5, p2}
- Let new d = 4

Let
- p6.Y = 10
- Search LY for Y = {6, 14}
- This is the active window
- ActiveWindow = {p5}

Note red rectangle (active) is 4x8 rectangle

Implementation: Initialization

- Let d = OO (best distance so far, infinity)
- Let LX be sorted set on x-axis
  - Read input points and add in LX
- Let LY be sorted set on y-axis

Implementation: processing

- For every point p in LX
  - For every point q in LY
    - If $|p.x - q.x| > d \Rightarrow$ remove q from LY
    - Note: Get these points with iterating on left of p in LX
  - start = lower_bound(LY, p.y - d)
    - E.g. find **first** point with its q.y >= p.y-d
  - end = upper_bound(LY, p.y + d)
    - E.g. find first point with its q.y > p.y+d
  - For pos: from start to end
    - cur_dist = distance(p, LY[pos])
    - if(cur_dist < d) $\Rightarrow$ d = cur_dist
  - Add p to LY

Implementation

```
typedef complex<double> point;

#define X real()
#define Y imag()
#define vec(a,b)                ((b)-(a))
#define length(a)               (hypot((a).imag(), (a).real()))

struct cmpX {
  bool operator()(const point &a, const point &b) {
    if (dcmp(a.X, b.X) != 0)
      return dcmp(a.X, b.X) < 0;
    return dcmp(a.Y, b.Y) < 0;
  }
};

struct cmpY {
  bool operator()(const point &a, const point &b) {
    if (dcmp(a.Y, b.Y) != 0)
      return dcmp(a.Y, b.Y) < 0;
    return dcmp(a.X, b.X) < 0;
  }
};
```

Implementation

```
double closestPair(vector<point> &eventPts) {
  double d = OO;
  multiset<point, cmpY> activeWindow;
  sort(eventPts.begin(), eventPts.end(), cmpX());

  int left = 0;
  for (int right = 0; right < (int) eventPts.size(); ++right) {
    while (left < right && eventPts[right].X - eventPts[left].X > d)
      activeWindow.erase(activeWindow.find(eventPts[left++]));
    auto asIt = activeWindow.lower_bound(point(-OO, eventPts[right].Y - d));
    auto aeIt = activeWindow.upper_bound(point(-OO, eventPts[right].Y + d));
    for (; asIt != aeIt; asIt++)
      d = min(d, length(eventPts[right] - *asIt));
    activeWindow.insert(eventPts[right]);
  }
  return d;
}
```

Little complex data structure

- Let LXY be a map of x position to sorted list of all available y positions
  - E.g. in C++:
  - map<double, multiset<double> > pointsMap
  - That is for every input (x, y)
  - pointsMap[x].insert(y);
- Now, this structure allow us to:
  - Once search on x dimension to get the big rectangle
  - Then search on y dimension to get the small rectangle
- Overall smaller code, little more smarter

Implementation

```
#define foreach(a,s) for(auto a=(s).begin();a!=(s).end();a++)

double closestPair(map<double, multiset<double> > & pointsMap) {
  double d = OO;
  foreach(xsIt, pointsMap) foreach(ymIt, xsIt->second) // sweep on each point p
    {
      double x = xsIt->first, y = *ymIt;
      // Iterate on rectangle dx2d (max 6 points)
      // iterate on active set - X dimension (distance d)
      auto xeIt = pointsMap.upper_bound(x + d);
      for (auto xIt = xsIt; xIt != xeIt; xIt++) {
        double x2 = xIt->first;
        // iterate on active set - Y dimension (distance 2d)
        auto ysIt = xIt->second.lower_bound(y - d);
        auto yeIt = xIt->second.upper_bound(y + d);
        for (; ysIt != yeIt; ysIt++) {
          if (xsIt != xIt|| ymIt != ysIt)  // if NOT original (x,y)
            d = min(d, max( abs(x-x2), abs(y-*ysIt)));
        }
      }
    }
  return d;
}
```

Final Notes

- Line sweep is a technique (think like DP)
  - We can use it in many (advanced) problems
- It is all about vertical/horizontal sweep line + active window. Efficiency is important key
- Sometimes the active window need
  - Available balanced tree (such as **C++/Java set**)
  - Or Written such as AVL or **treap** (modify structure)
  - Another **nested sweep line** (e.g. horizontally)
  - Complex structure such as **segment tree**
- There are some variation (e.g. **radial** **sweep**)

تم بحمد الله

علمكم الله ما ينفعكم

ونفعكم بما تعلمتم

وزادكم علماً