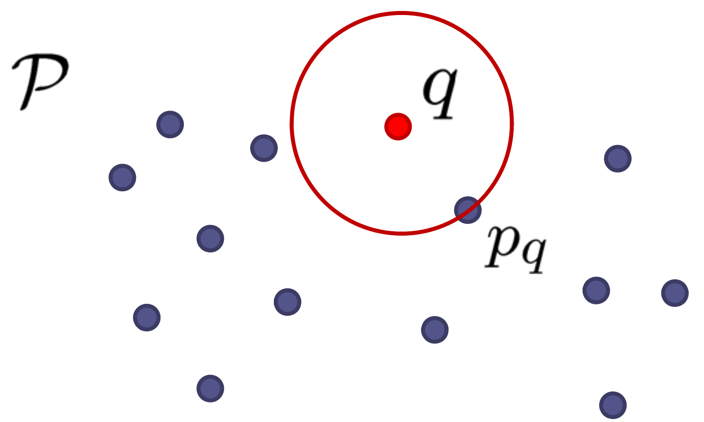
Nearest Neighbor Search

CS164

Problem Definition

- Given a set $\mathcal{P}$ of points in space and a query point q find its nearest neighbor in $\mathcal{P}$:

$$p_q = \operatorname*{argmin}_{p \in \mathcal{P}} d(p, q)$$



Problem Definition

- Given a set $\mathcal{P}$ of points in space and a query point q find its nearest neighbor in $\mathcal{P}$:

$$p_q = \operatorname*{argmin}_{p \in \mathcal{P}} d(p, q)$$

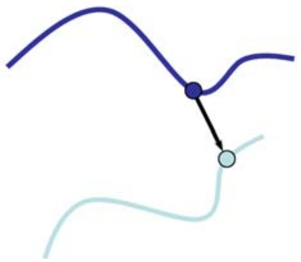
- In Euclidean space:

$$p_q = \operatorname*{argmin}_{p \in \mathcal{P}} \|p - q\|_2^2$$

- Can solve in $O(|\mathcal{P}|)$ with brute force search.
- Want sublinear query time with reasonable preprocessing and storage requirements.

Why is this important?

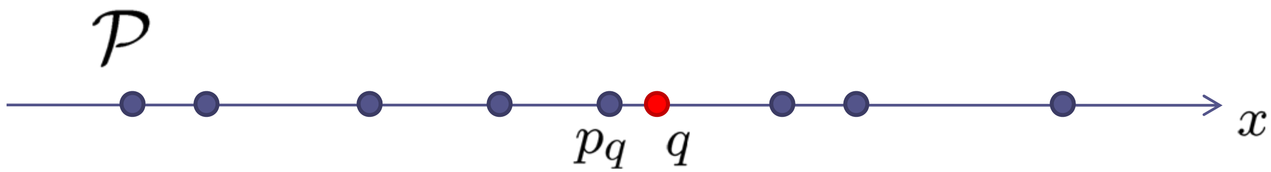
- Alignment: part of the ICP algorithm.



- Collision Detection: are the 2 shapes too close?
- Normal Estimation, surface reconstruction, rendering… Many, many others

Let's start in 1D.

- Given a collection of points on the line (*x* axis), and a query point. Find its nearest neighbor.

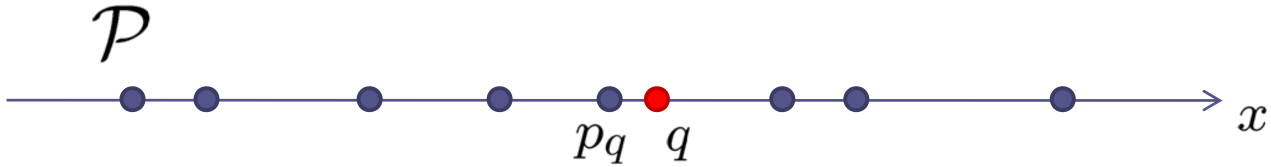


- Note that points have an ordering. So find the interval: $p_i \leq q \leq p_{i+1}$ and $p_q = \underset{p_i, p_{i+1}}{\operatorname{argmin}} |q - p|$.

- This is binary search!

Let's start in 1D.

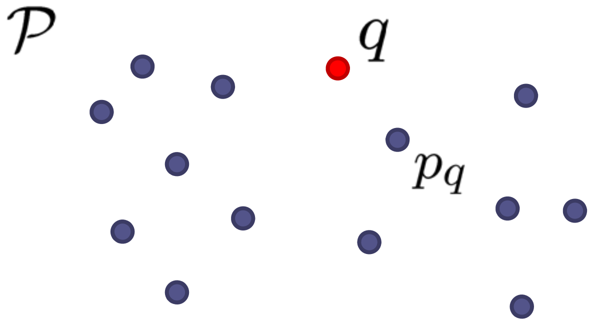
- Given a collection of points on the line (*x* axis), and a query point. Find its nearest neighbor.



- Preprocessing:
  - Sort the points in $O(n \log n)$ time.
- Answering a query:
  - Binary search to find the interval, and report the closest of the two points. $O(\log n)$ time.
- Perfect method!

In 2D things get complicated.

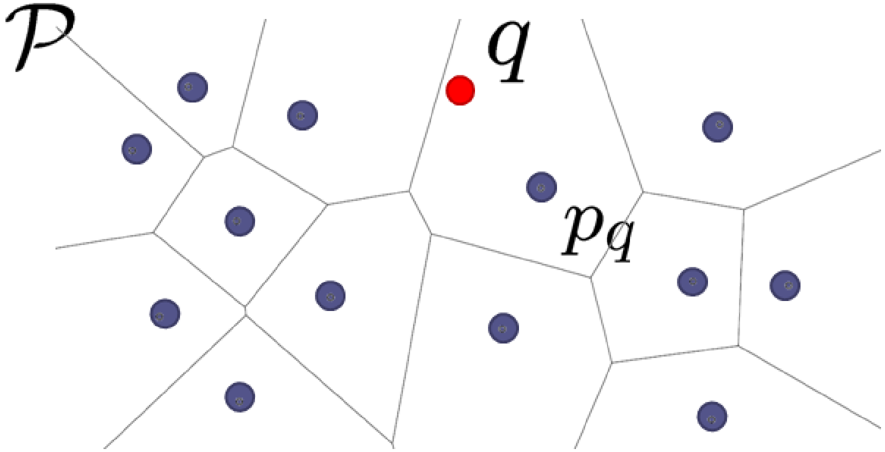
- Given a collection of points and a query point. Find its nearest neighbor.



- Points do not have a natural ordering. Closest points along each axis can be different.
- However, can extend a similar intuition.

In 2D things get complicated.

- Remember the Voronoi diagram:



- Partition the space into “influence regions.”
- Finding the nearest point = finding the region that contains the query point.

Point Location in 2D. Kirkpatrick's Algorithm.

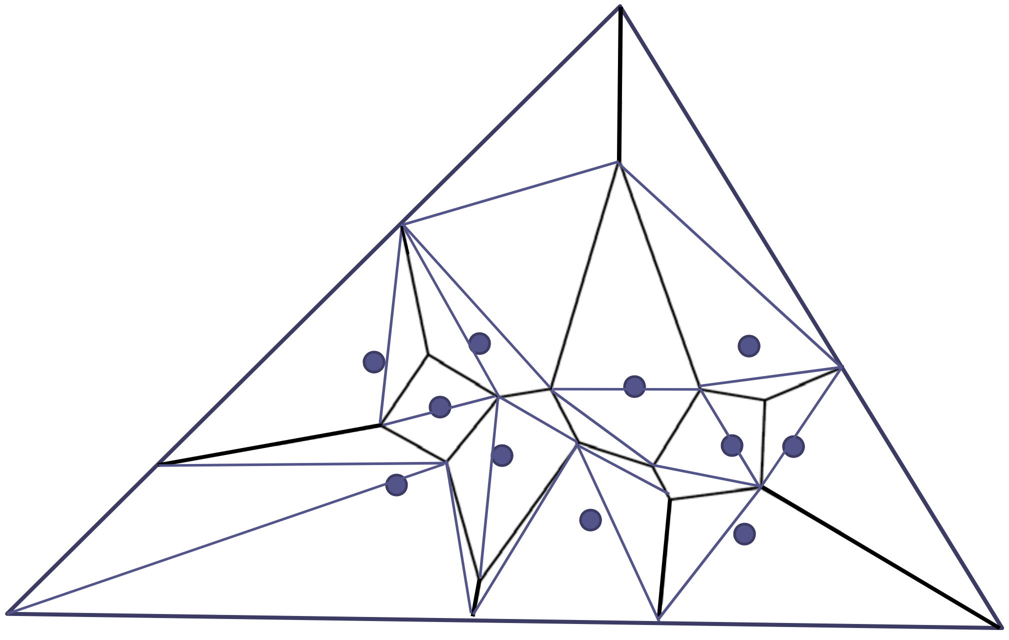
- Achieves optimal $O(\log n)$ query time with $O(n)$ storage and $O(n \log n)$ preprocessing.

- Assume that the planar subdivision is a triangulation. Locating a point inside a set of triangles.

Point Location in 2D. Kirkpatrick's Algorithm.

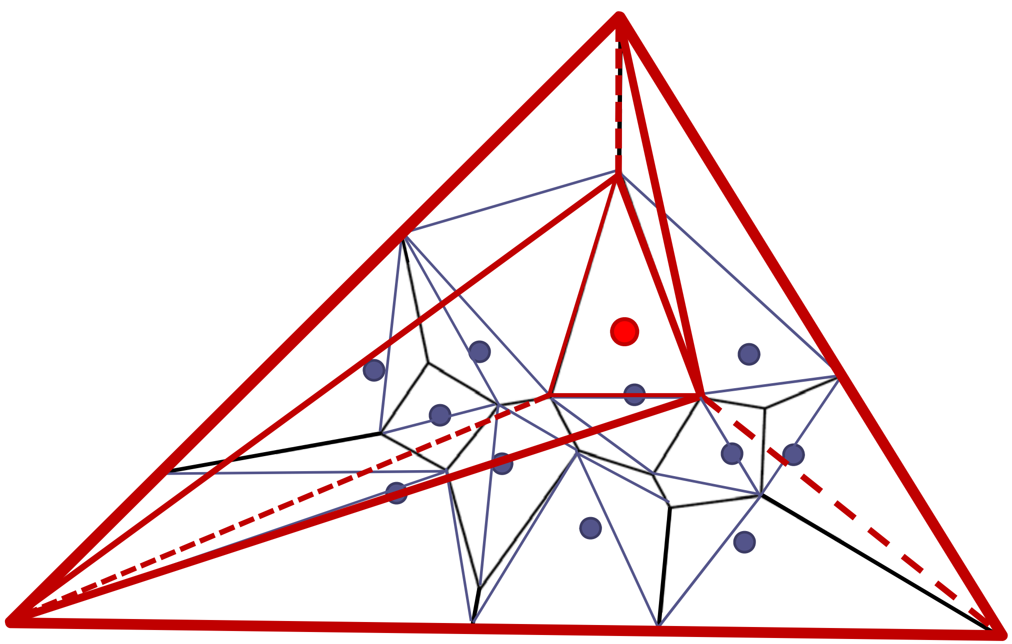
- Convert Voronoi diagram into a triangulation.

1. Compute the bounding triangle
2. Inside each face, pick a vertex and connect others to it. Possible since faces are convex.



Point Location in 2D. Kirkpatrick's Algorithm.

- Main Idea: Binary search on triangles.
- Create a hierarchy, such that answering a query involves descending in $O(\log n)$ steps.



Point Location in 2D. Kirkpatrick's Algorithm.

- Creating the hierarchy:
  - Start with all triangles: T_0.
  - Delete a constant fraction of vertices, and retriangulate to get T_1.
  - Make sure that every triangle in T_1 overlaps a constant number of triangles in T_0.

iterate

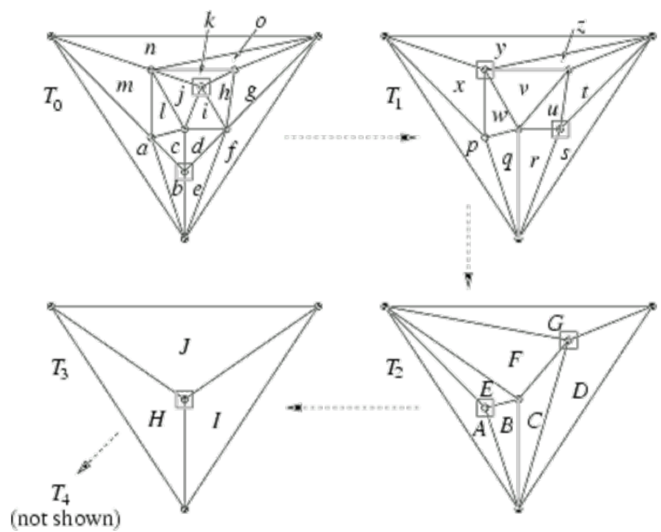


Image by D. Mount

Point Location in 2D. Kirkpatrick's Algorithm.

- By construction the number of triangulations is $O(\log n)$.

- Answering a point location query:
  - Traverse the hierarchy from T_k to T_0.
  - Find which triangle in T_{i-1} contains q : $O(1)$ time.
  - Will terminate in $O(\log n)$.

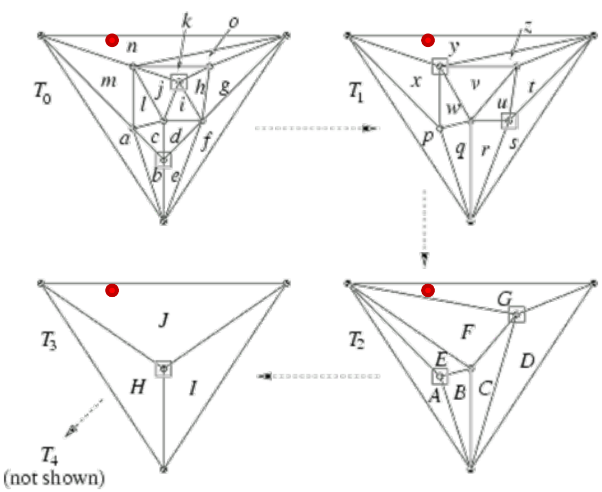
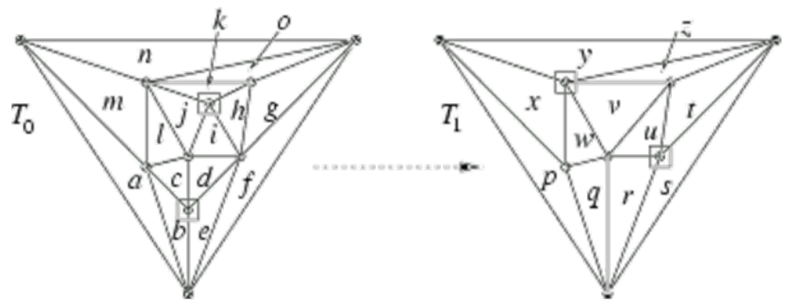


Image by D. Mount

Point Location in 2D. Kirkpatrick's Algorithm.

- Main question:
  - Can we always find a good set of vertices to delete?

- Main observation:
  - Consider set of independent vertices with bounded degree.

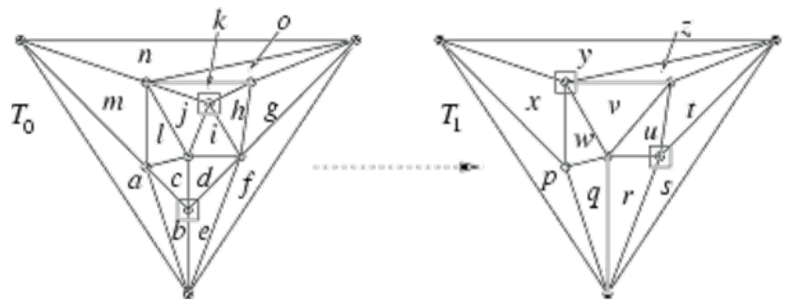


  - Removing each, creates a hole of size at most d.
  - New triangles will intersect at most d old ones.

Point Location in 2D. Kirkpatrick's Algorithm.

- Main question:
  - Can we always find a good set of vertices to delete?

- Main lemma:
  - Consider set of independent vertices with degree less than d.



  - If $d = 8$ there are at least $n/18$ independent vertices whose degree is at most d. Can find them greedily in $O(n)$.
  - Follows from the fact that in a planar graph, the average degree is at most 6. Many points of degree under 8.

Point Location in 2D.

- Problems:
  - At each step, reducing the size by $1/18$, would like $1/2$.
  - More practical algorithms exist: take CS268!
  - In $\mathbb{R}^3$ no method is known with $O(n)$ space and $O(\log n)$ query time.
  - Complexity of the Voronoi diagram: $O(n)$ in $\mathbb{R}^2$, but grows quickly with dimension: $\Theta(n^{\lfloor \frac{d+1}{2} \rfloor})$.
  - Need a more practical algorithm!

Nearest Neighbor in 3D.

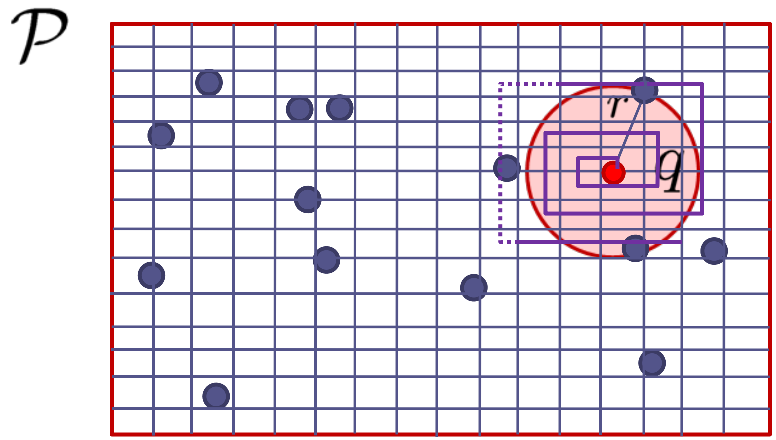
- Simple, yet practical methods, perhaps at the expense of worst case running times.

- First simple Idea:

  Partition the space into a grid, such that each cell contains a constant number of points.

Voxel Grids in 2D/3D

- Partition the space into a grid with a constant number of points per cell.



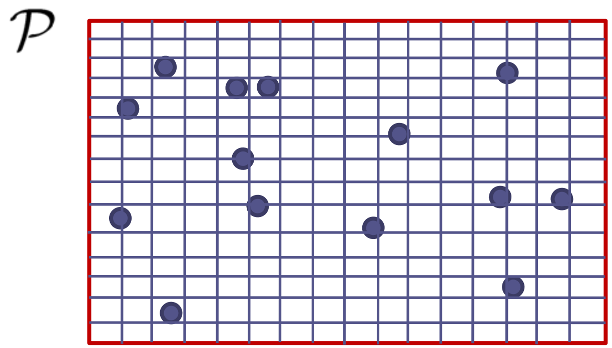
- Answering a query:

 1. Locate the cell containing q. $O(1)$ time.

 2. Perform a spiral search of neighboring cells within distance r from q. Update r as you go.

Voxel Grids in 2D/3D

- Easy to implement: entire grid is a 2/3D array. Can work if the points are roughly uniformly spaced.



- If points are sampled uniformly in a unit cube, use grid of size: $\sqrt{n/C} \times \sqrt{n/C}$. Expected C points per cell.
- Theorem: For uniform distribution, spiral search finishes in $O(1)$ time. [Bentley, Weide, Yao '80]
- Bad if non-uniform. Many Empty cells!

Nearest Neighbor in 3D.

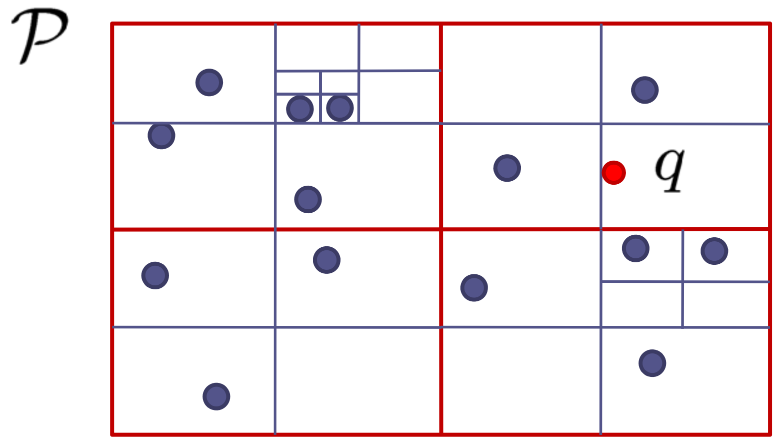
- Simple, yet practical methods, perhaps at the expense of worst case running times.

- Second Simple Idea:

  Recursively partition the space into 4 cells (2D)/8 cells (3D). Quad/Oct-trees.

Quad trees in 2D/3D

- Recursively partition the space into 4 (2D)/8 (3D) cells until each cell has a constant number of points.

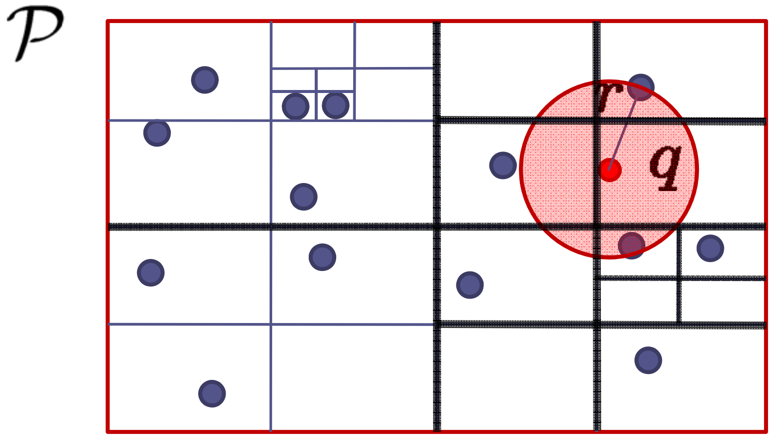


- Answering a query:

  Cells do not have direct access to neighbors. Need a different method.

Quad trees in 2D/3D

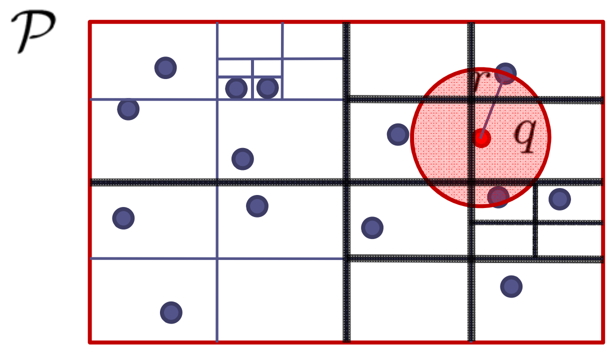
- Recursively partition the space into 4 (2D)/8 (3D) cells until each cell has a constant number of points.



- Do a depth first search to find the cell where q is located. If find a leaf, update r.
- Descend into cells that are less than r away from q.

Nearest neighbor search is Quad trees

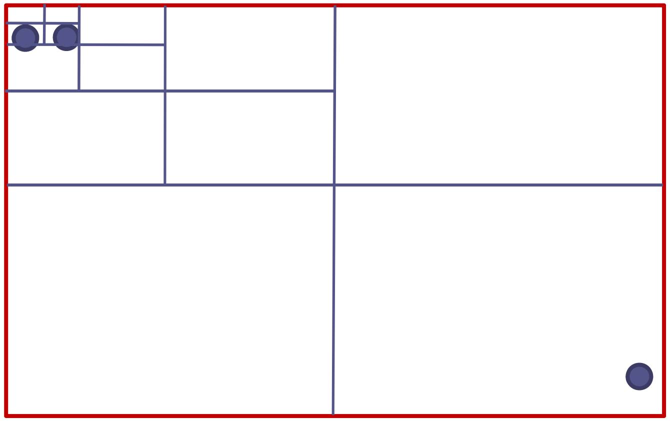
- Complete Algorithm:



  - put the root on the stack, $r = \infty$
  - repeat
    - pop the next node T from the stack
    - for each child C of T :
      - if C is a leaf, examine point(s) in C, update r
      - if C intersects with the ball of radius r around q, add C to the stack ordered by distance from q

Nearest neighbor search is Quad trees

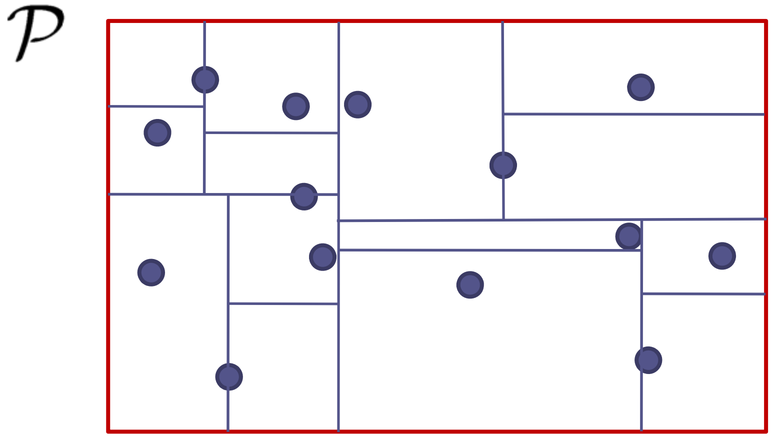
- Main problem with Quad Trees: many empty cells.
- If the data is unbalanced, can take a very long time to subdivide



- Need more intelligent splitting rules.

kD-trees

- Two main differences from Quad trees:
  1. Split dimension by dimension (not together)
  2. During each split, try to make the tree as balanced as possible



Many Strategies. Common: cycle through dimensions and each time split along the median.

kD-trees

- Guaranteed not to have empty cells.

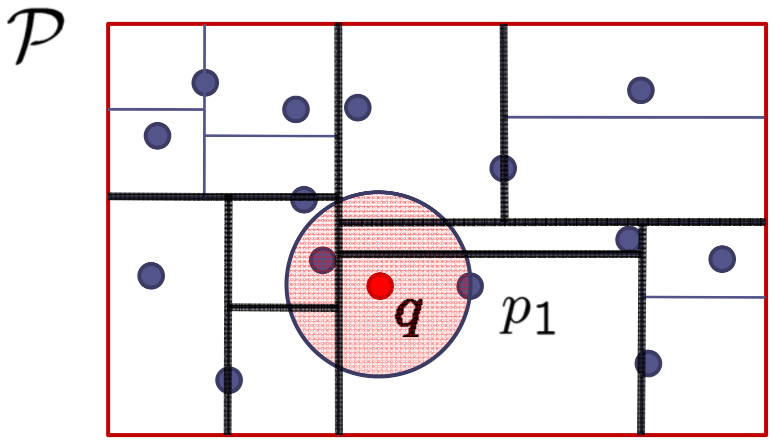
- Median of n points can be found in $O(n)$ time.
- Construction is $O(n \log n)$.

- Query can be done in the same way as in quad-trees.
- Query time depends on the distribution of points.

- In the worst case, have to examine all cells: $O(n)$

Nearest neighbor query with a kD-tree

- Depth first search to find the cell containing q.
- Set $r = d(q, p_1)$.
- Go up the tree inspecting cells closer than r from q, where r is the distance to the closest point so far.



kD-trees

- Query example 1:

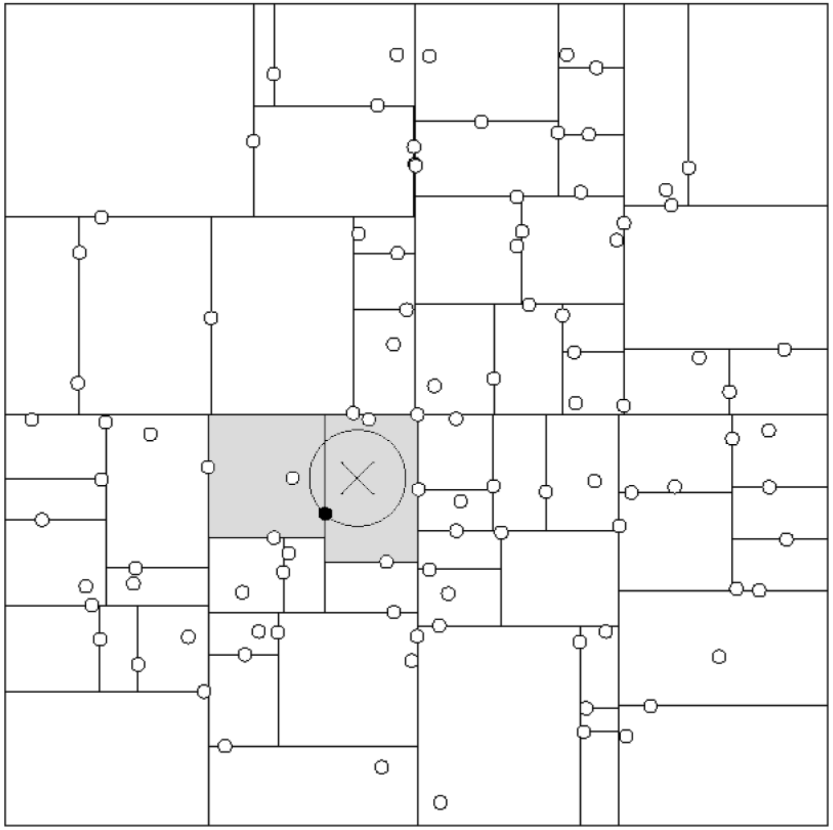


Image by S. Renals

kD-trees

- Query example 2:

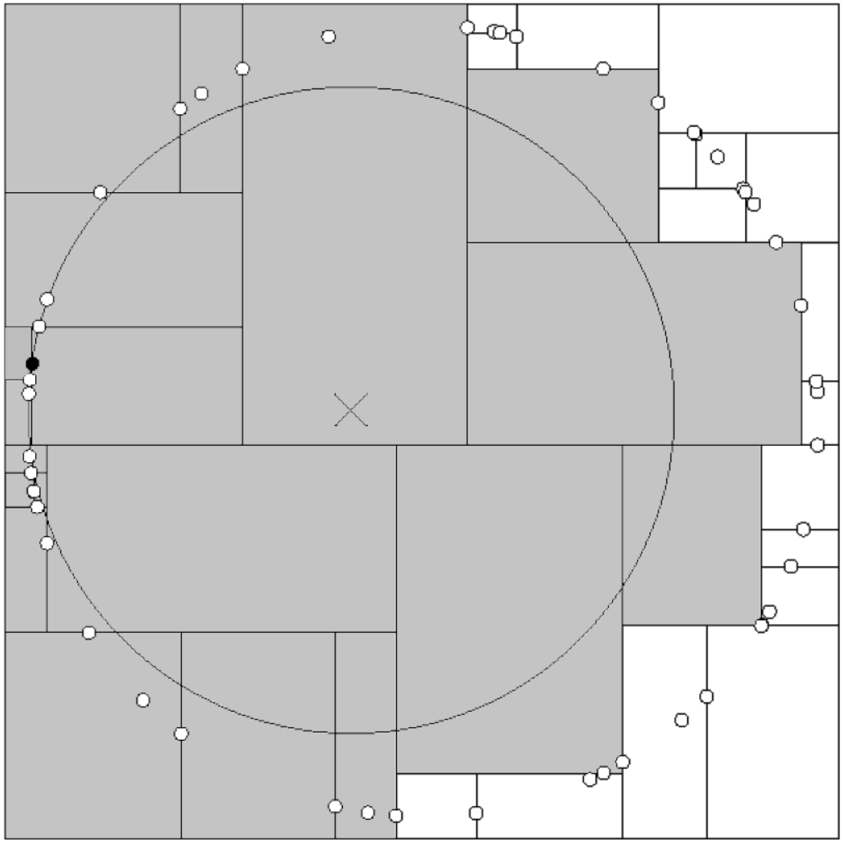
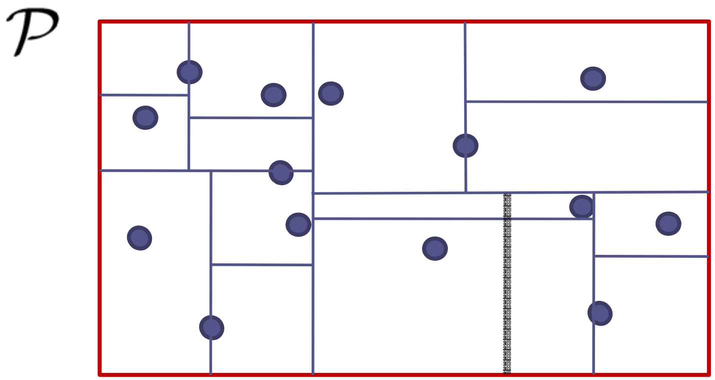


Image by S. Renals

kD-trees

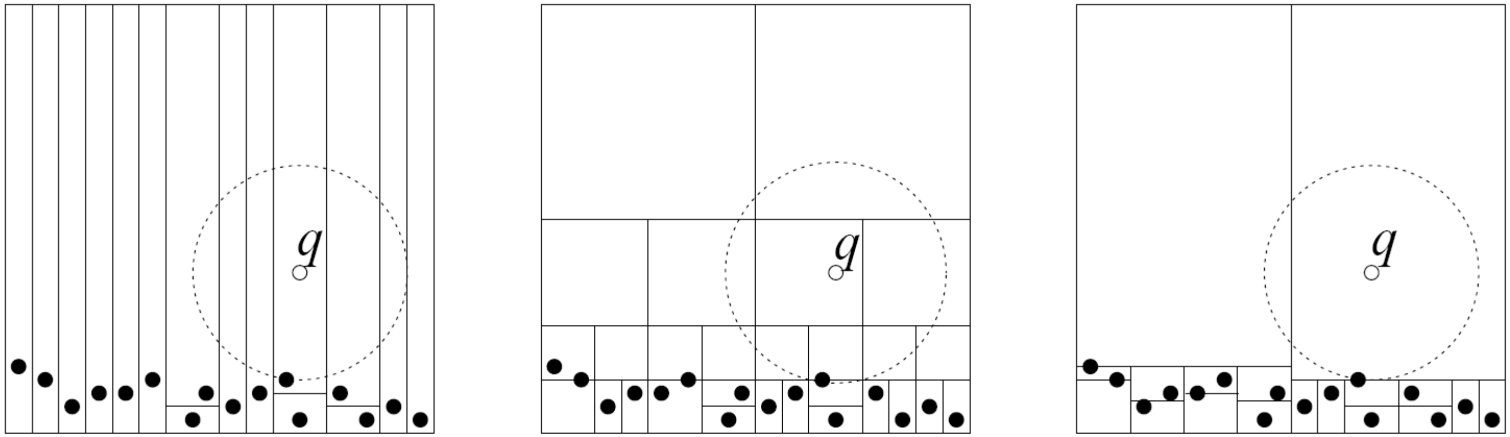
- Main goals:
  - Keep the tree balanced (no empty cells).
  - Avoid making skinny rectangles (many neighbors).
- These are conflicting goals. Quad-tree have good aspect ratios, with many empty cells.

- Common modification: instead of cycling through dimensions, pick the one along which points are most spread.



kD-trees

- Not guaranteed to work. Many interesting splitting schemes have been proposed.



Friedman et al. Quad Tree Arya & Fu

Songrit Maneewongvatana and David M. Mount *It's okay to be skinny, if your friends are fat*, 1999

kD-trees

- In practice, kD-trees work remarkably well.
- Can be extended to higher dimensions, but other problems arise: exponential dependence of the query time on the dimension.
- Can be extended for approximate nearest neighbor queries: we're happy with a point that's close to the nearest neighbor. Much more efficient in high D.
- No need to implement the kD-tree from scratch. A very robust implementation: ANN by Arya and Mount:

  http://www.cs.umd.edu/~mount/ANN/