Spatial/Vector Indexes

CSE 4/562: Database Systems | Lecture 15

Multidimensional Data

Multidimensional Data

- Restaurant Locations (Lat + Lon)
- Flight Recorder Data (X, Y, Z + Time)
- Weather Data (Volumetric X, Y, Z + Time)
- Simulation State (e.g., Fly eBrain)
- Games!
- Embeddings

What's Special about Space?

Spatial data lives in a “Metric Space”

Distance Function

$d(p_1, p_2) \to \mathbb{R}$

- $d(p, p) = 0$
- $d(p_1, p_2) > 0$
- $d(p_1, p_2) = d(p_2, p_1)$
- $d(p_1, p_2) + d(p_2, p_3) \geq d(p_1, p_3)$

Distance Function

$d(p_1, p_2) \to \mathbb{R}$

- $d(p, p) = 0$
- $d(p_1, p_2) > 0$
- $d(p_1, p_2) = d(p_2, p_1)$
- $d(p_1, p_2) + d(p_2, p_3) \geq d(p_1, p_3)$

Metric spaces define a notion of “similarity” over their points.

Typical Queries over Point Data

Typical Queries over Point Data

- **Bounding Rectangle**: What records fall within a rectangular query region?
  - Find me all bike repair shops in Buffalo?

Typical Queries over Point Data

- **Bounding Rectangle**: What records fall within a rectangular query region?
  - Find me all bike repair shops in Buffalo?
- **Bounding Sphere**: What records are at most ε away from a query point?
  - What restaurants are within a 5 minute walk?

Typical Queries over Point Data

- **Bounding Rectangle**: What records fall within a rectangular query region?
  - Find me all bike repair shops in Buffalo?
- **Bounding Sphere**: What records are at most ε away from a query point?
  - What restaurants are within a 5 minute walk?
- **K Nearest Neighbors**: What are the k closest records to a query point?
  - Where is the nearest bus stop?

Typical Queries over Point Data

- **Bounding Rectangle**: What records fall within a rectangular query region?
  - Find me all bike repair shops in Buffalo?
- **Bounding Sphere**: What records are at most ε away from a query point?
  - What restaurants are within a 5 minute walk?
- **K Nearest Neighbors**: What are the k closest records to a query point?
  - Where is the nearest bus stop?

Other Types of Data

- **Intersection**: What polygons overlap with my query region?
  - What neurons will be activated when a given neuron fires?

Typical Queries over Point Data

- **Bounding Rectangle**: What records fall within a rectangular query region?
  - Find me all bike repair shops in Buffalo?
- **Bounding Sphere**: What records are at most ε away from a query point?
  - What restaurants are within a 5 minute walk?
- **K Nearest Neighbors**: What are the k closest records to a query point?
  - Where is the nearest bus stop?

Other Types of Data

- **Intersection**: What polygons overlap with my query region?
  - What neurons will be activated when a given neuron fires?

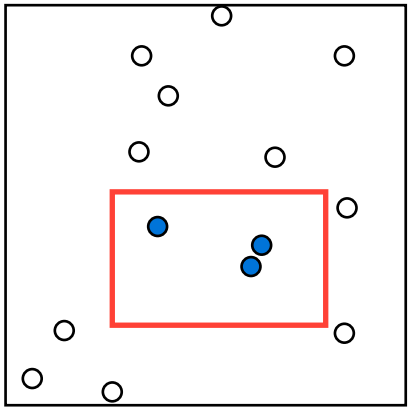
Data

- A collection of points. Each point may have a payload.

Query

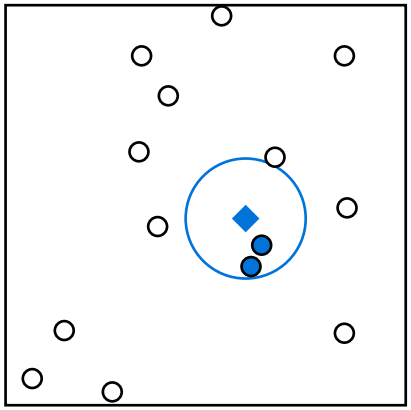
Find the points (and their payloads) that meet the given criteria.

Bounding Box



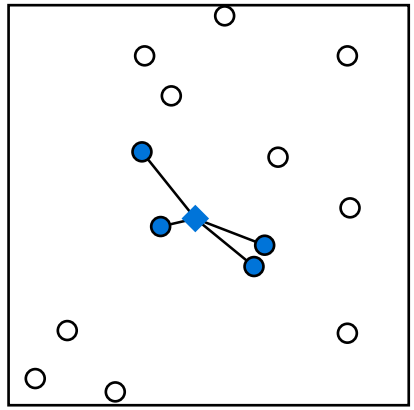
$q_{\text{low}}.x < p_i.x \leq q_{\text{high}}.x \wedge$
$q_{\text{low}}.y < p_i.y \leq q_{\text{high}}.y$

Bounding Sphere



$d(p_i, q) < \varepsilon$

K Nearest Neighbors



$\{p_i\}$ s.t.
$|\{p_j \mid d(p_j, q) < d(p_i, q))\}| < K$

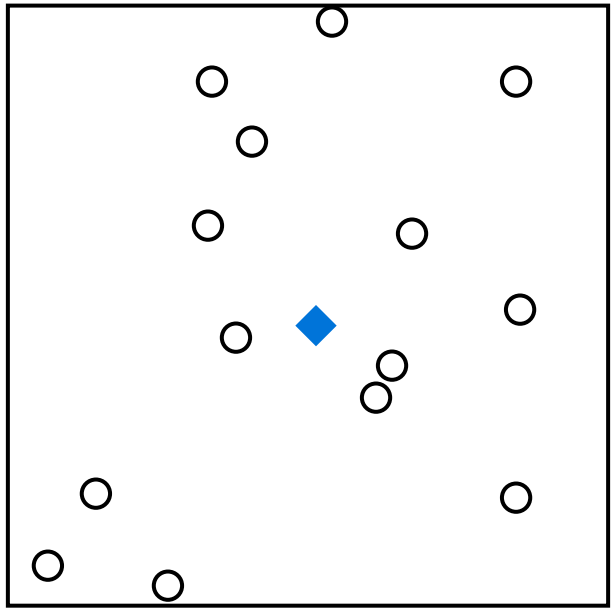
One more thing...

In many situations, an approximation is good enough.

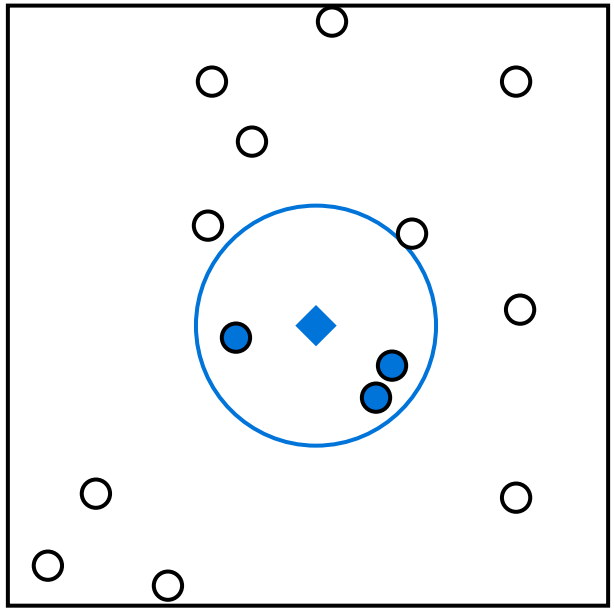
- Where's the nearest grocery store?
- What are the nearest 5 bus stops?
- Which taxi is closest?

These problems are related

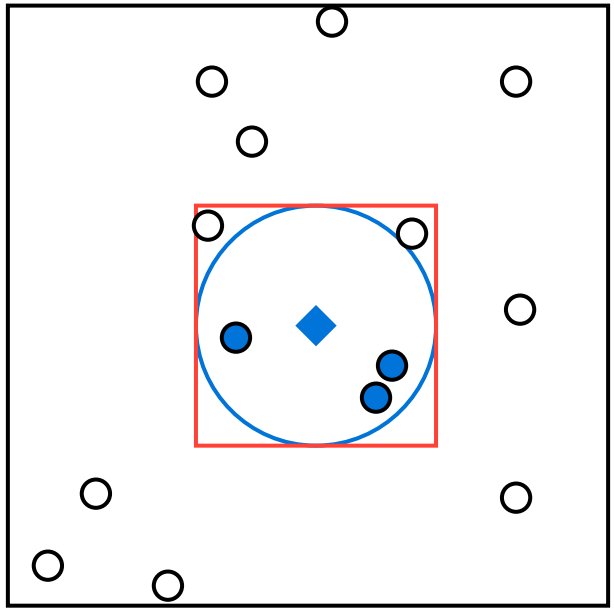
Bounding Sphere via Bounding Rectangle



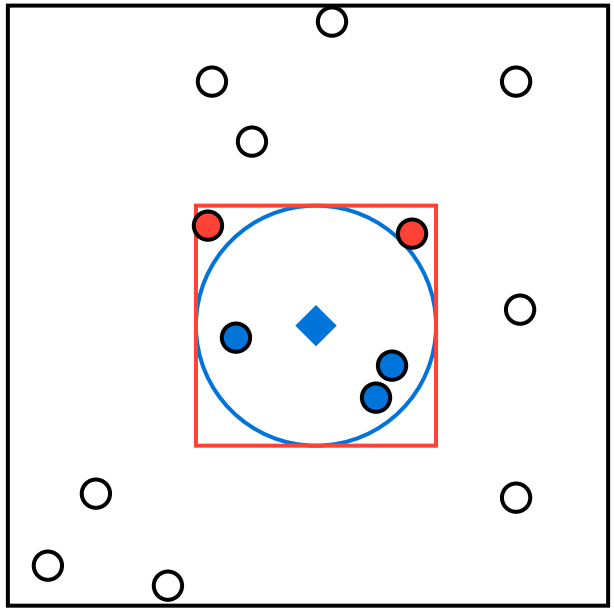
Bounding Sphere via Bounding Rectangle



Bounding Sphere via Bounding Rectangle



Bounding Sphere via Bounding Rectangle



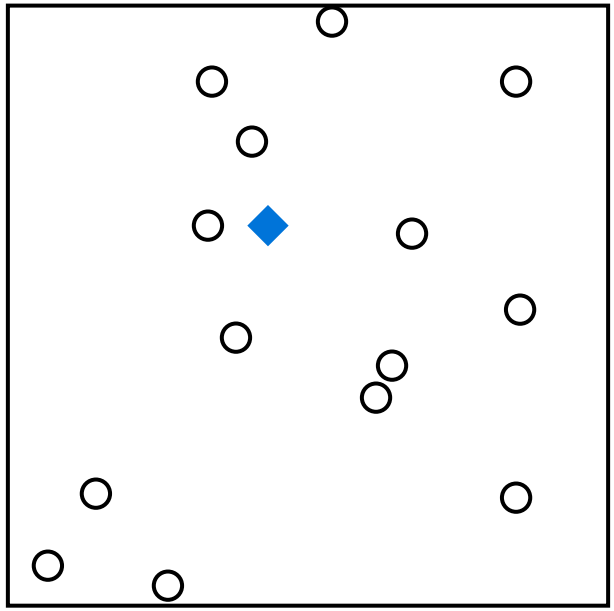
Bounding Sphere via Bounding Rectangle


```
def bounding_sphere(self, center, e):
    box = compute_rectangle_enclosing_sphere(center, e);

    for pt in self.bounding_rectangle(box):
        if distance(pt, center) < e:
            yield pt
```

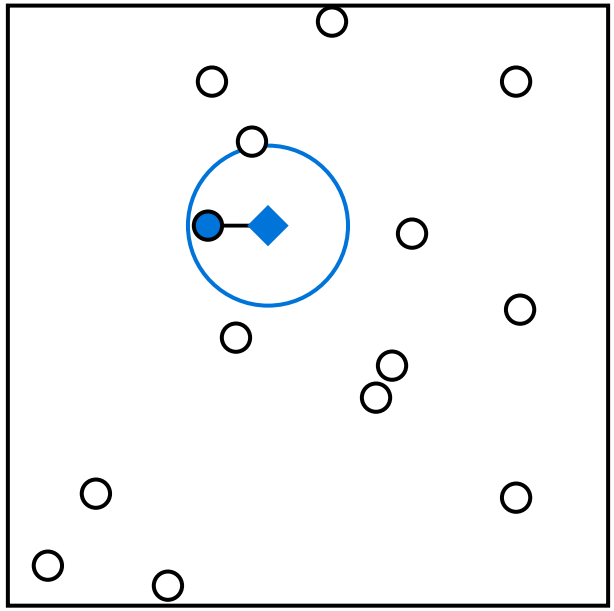
vs

```
def bounding_sphere(self, center, e):
    for pt in self.all_points:
        if distance(pt, center) < e:
            yield pt
```

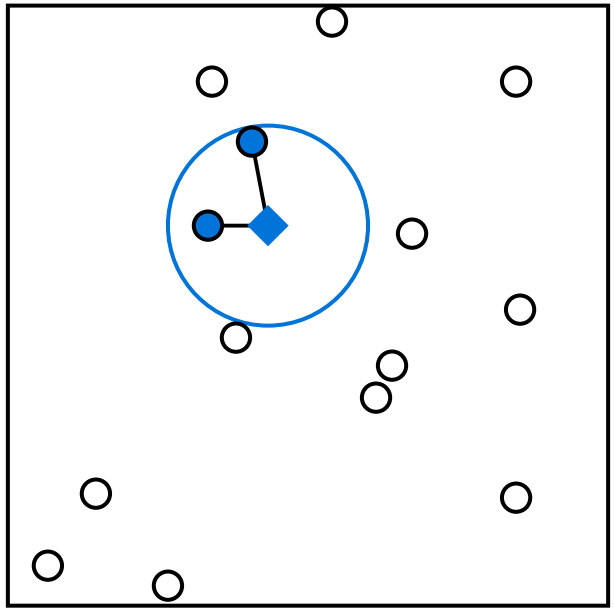
KNN via Bounding Sphere



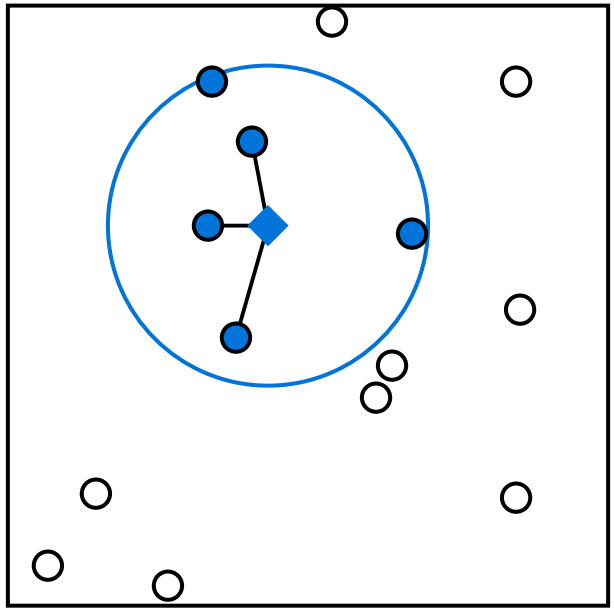
KNN via Bounding Sphere



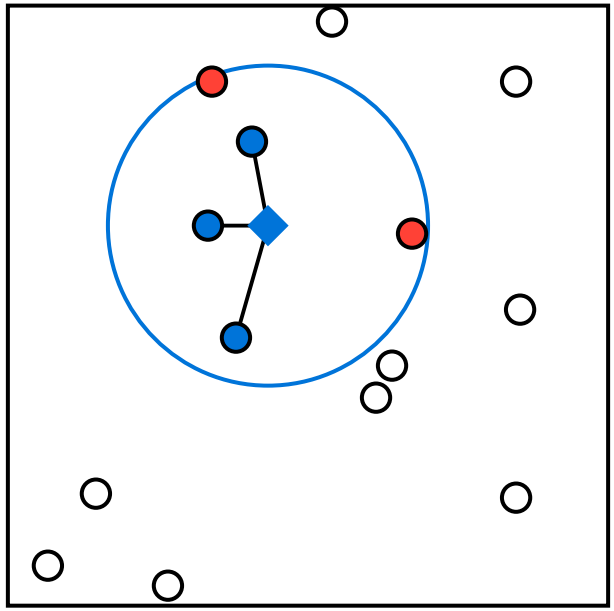
KNN via Bounding Sphere



KNN via Bounding Sphere



KNN via Bounding Sphere



KNN via Bounding Sphere


```
def k_nearest_neighbors(self, query, k):
    epsilon = SOME_DEFAULT
    while len(result := self.bounding_sphere(query, epsilon)) < k:
        epsilon *= 2

    result = result.sorted(key = lambda p: distance(query, p))
    for pt in result[0:k]:
        yield pt
```

Point Search Problems are Related

If you can solve one problem, you (probably) can solve the others.

The algorithms I gave are very naive. You can almost always do better.

1: The Naive Solution

Exploring the Design Space

1. We want to do range scans.
   - Hash-based indexing is out
2. We have 2+ dimensions
   - We need to linearize the data to use a tree-based index
3. Space is continuous
   - Data points are sparse

Binary Trees

- Each node has 2 children.
- Each node partitions the space of its dependencies 2 ways.

B+ Trees

- Each node has c children.
- Each node partitions the space of its dependencies c ways.

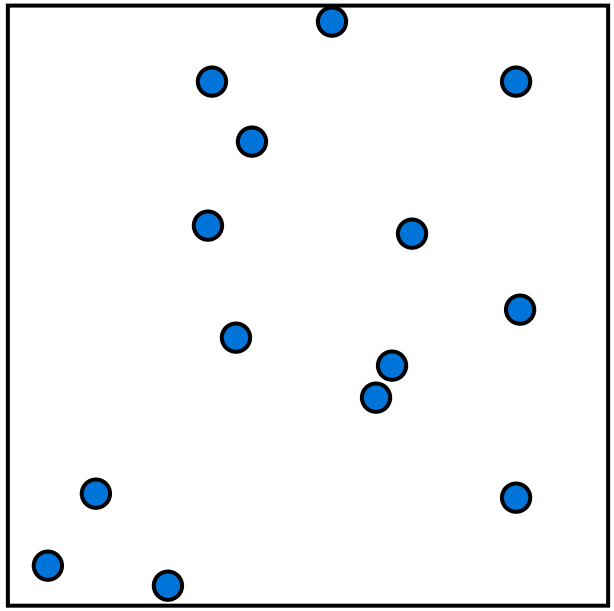
> We'll be using binary trees today, but you can increase fanout.

Lexical Ordering

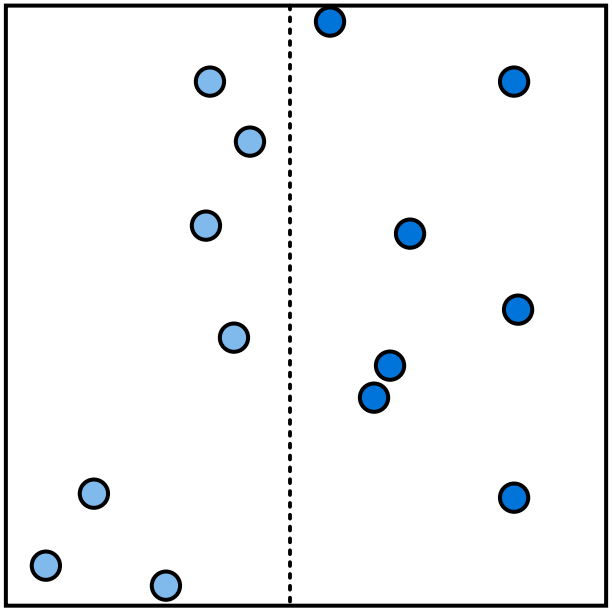
Given attributes $(a_1, ..., a_d)$:

- Sort on attribute a_1
- Break ties on (a_1) using a_2
- Break ties on (a_1, a_2) using a_3
- ...
- Break ties on $(a_1, ..., a_{d-1})$ using a_d

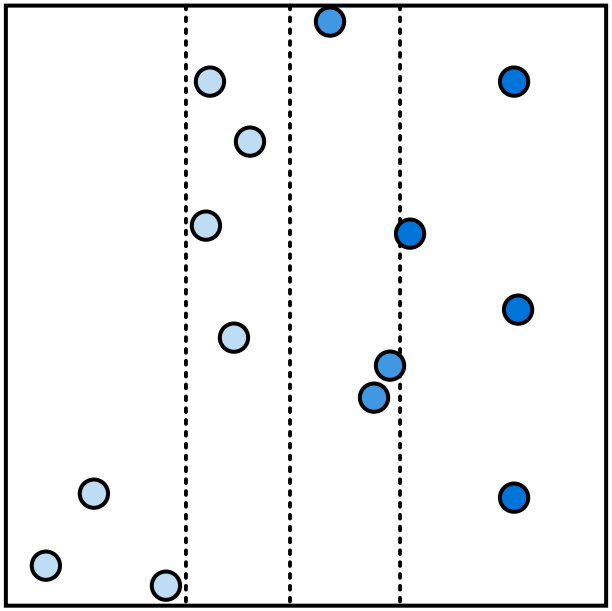
Using Lexical Order



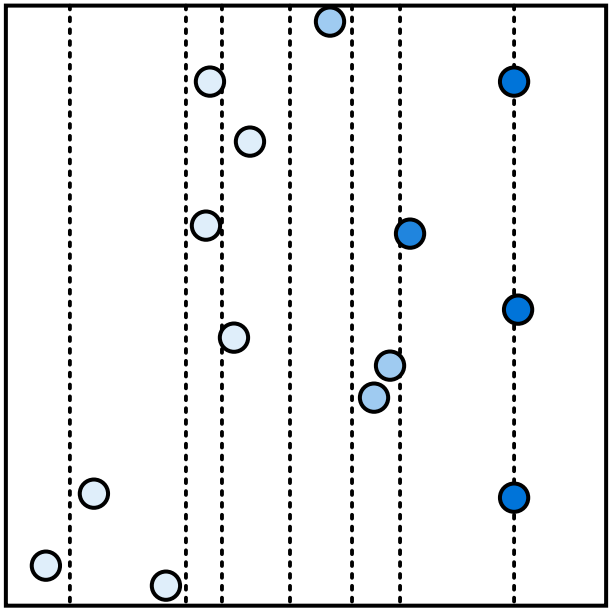
Using Lexical Order



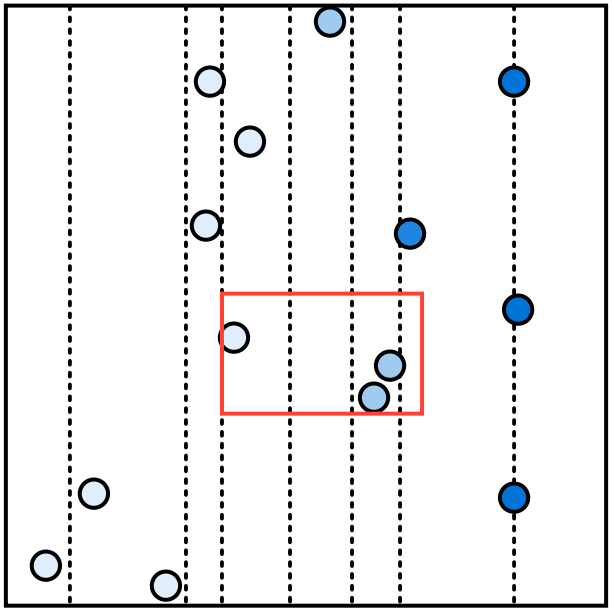
Using Lexical Order



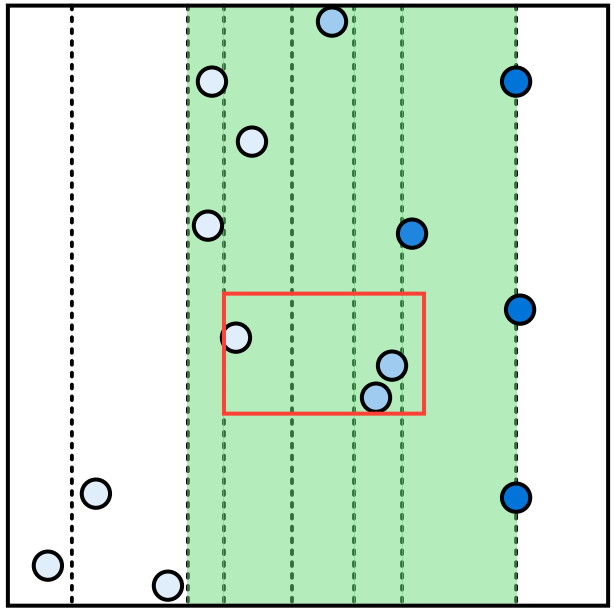
Using Lexical Order

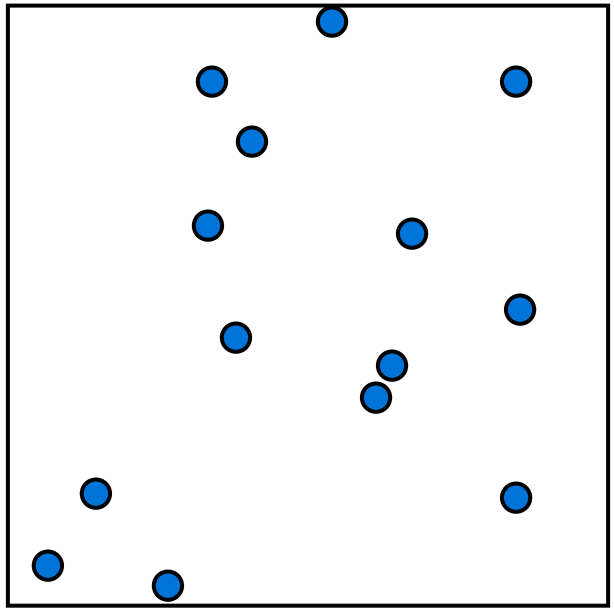


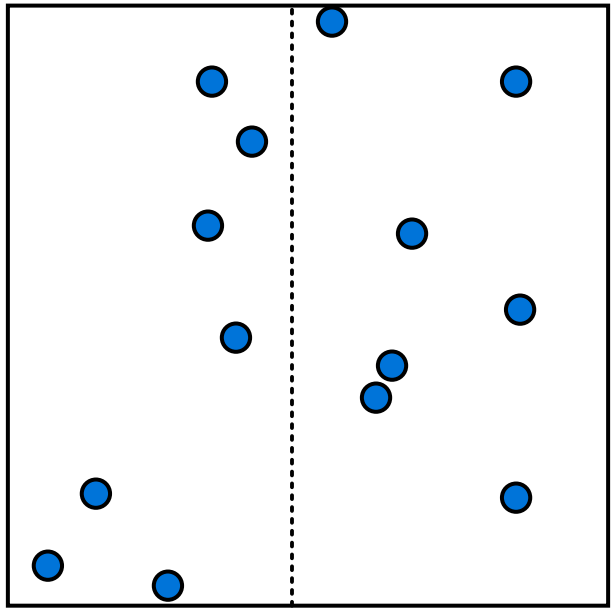
Using Lexical Order

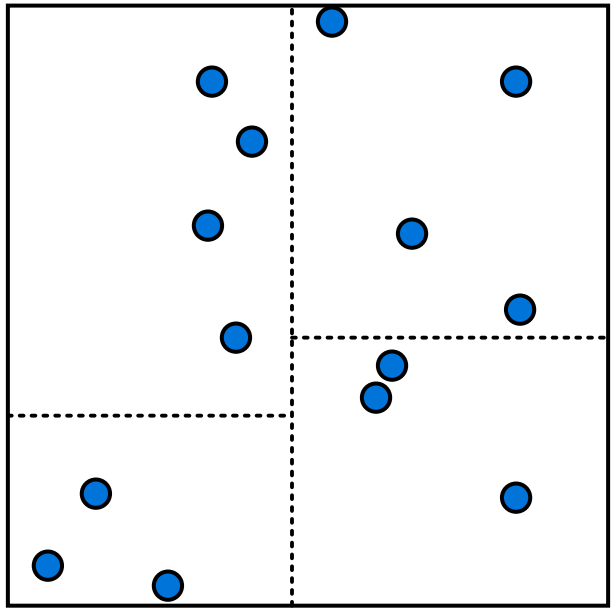


Using Lexical Order

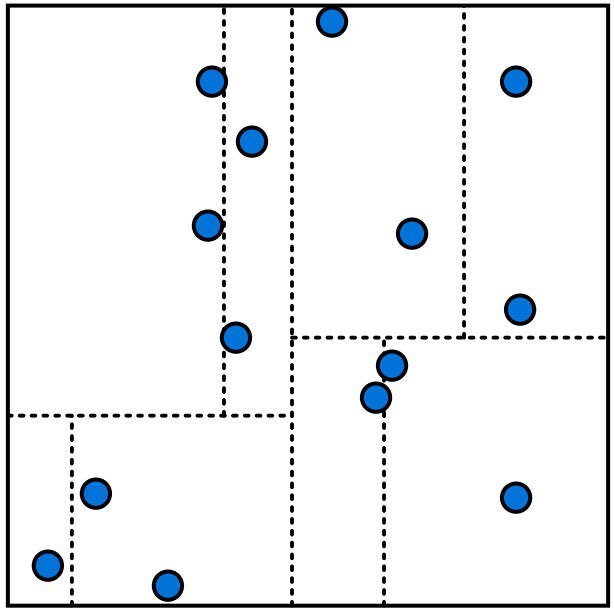


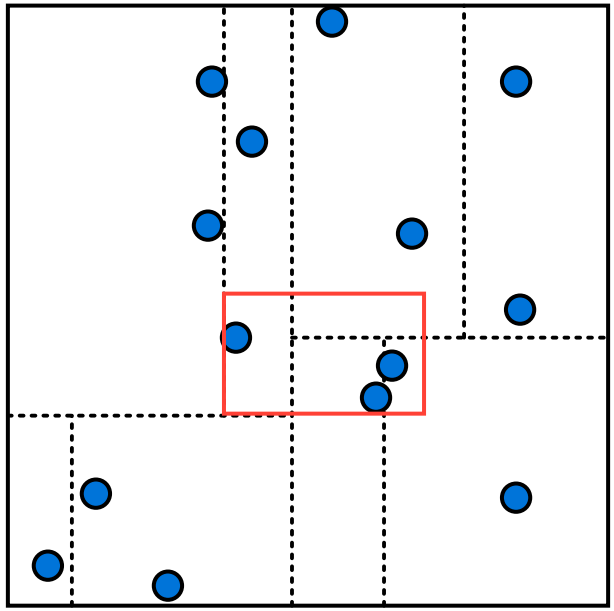






K-D Trees





Building a K-D Tree


```
def build_kd(data, level = 0):
    if len(data) < 2:
        return LeafNode(data)

    dimension = level % NUM_DIMENSIONS
    data.sort(key = lambda pt: pt[dimension])

    low = data[:len(data)/2]
    high = data[len(data)/2:]
    sep = high[0][dimension]

    return DirectoryNode(
      build_kd(low, level + 1),
      sep,
      build_kd(high, level + 1))
```

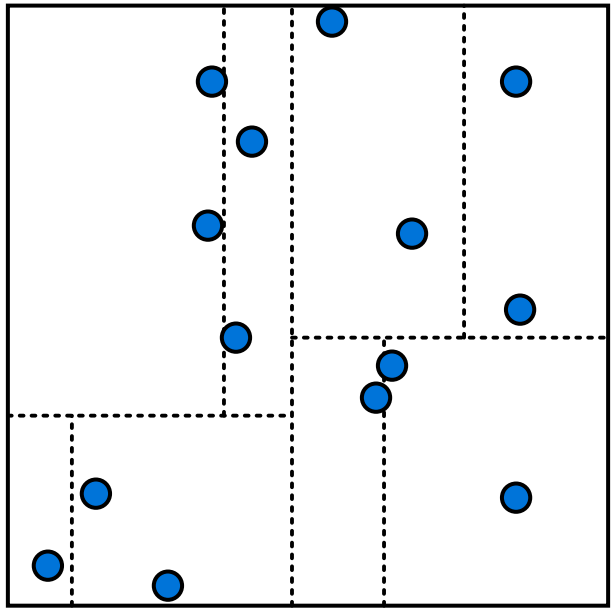
Point Lookups in a K-D Tree


```
class DirectoryNode:
  def find(self, query_point, level = 0):
      dimension = level % NUM_DIMENSIONS

      if query_point[dimension] < self.sep:
          self.low.find(query_point, level+1)
      else:
          self.high.find(query_point, level+1)

class LeafNode:
  def find(self, query_point, level = 0):
      if self.point == query_point:
          yield self.point
```

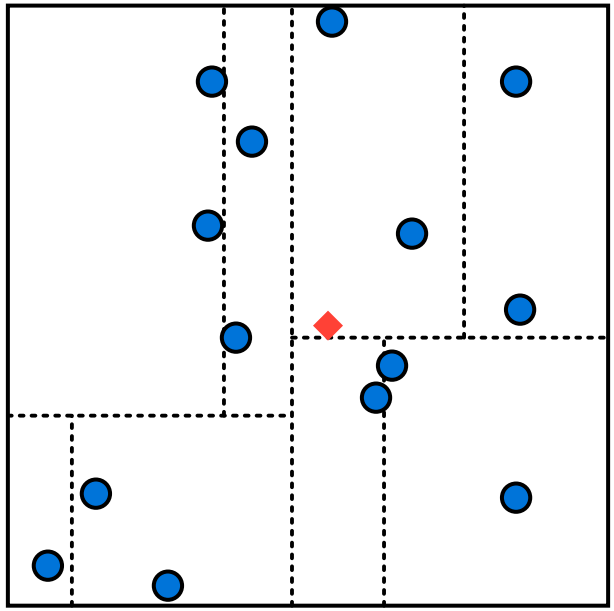
Range Lookups in a K-D Tree


```
class DirectoryNode:
  def find_range(self, query_box):
      if intersects(self.low.bounds, query_box):
          self.low.find_range(query_point, level+1)
      if intersects(self.high.bounds, query_box):
          self.high.find_range(query_point, level+1)
```

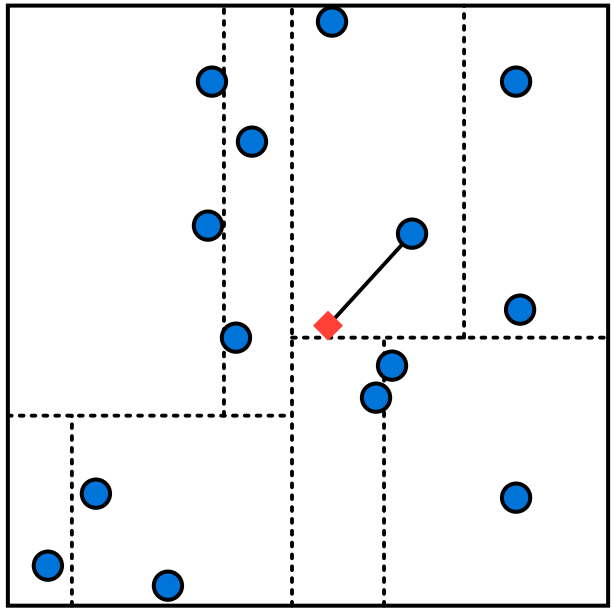
KNN in K-D Trees



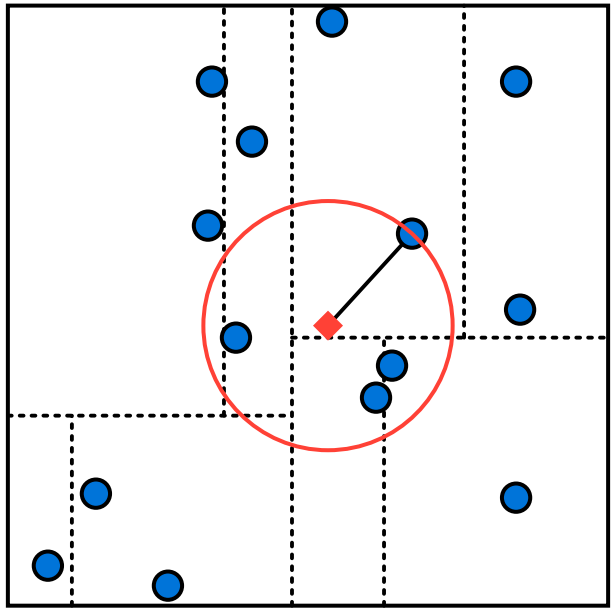
KNN in K-D Trees



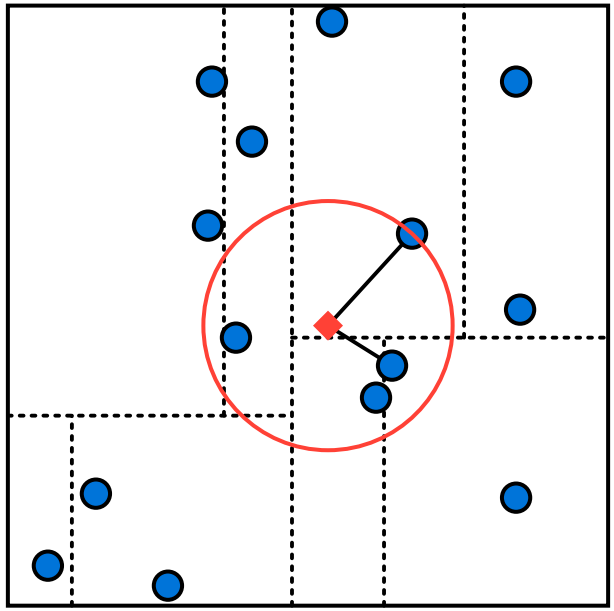
KNN in K-D Trees



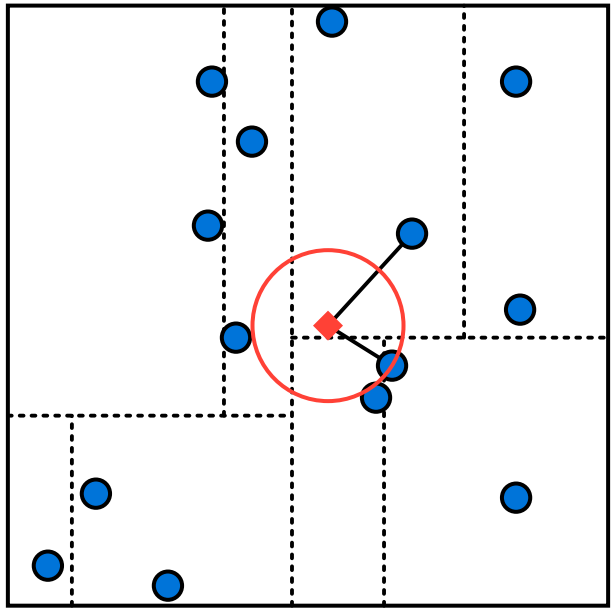
KNN in K-D Trees



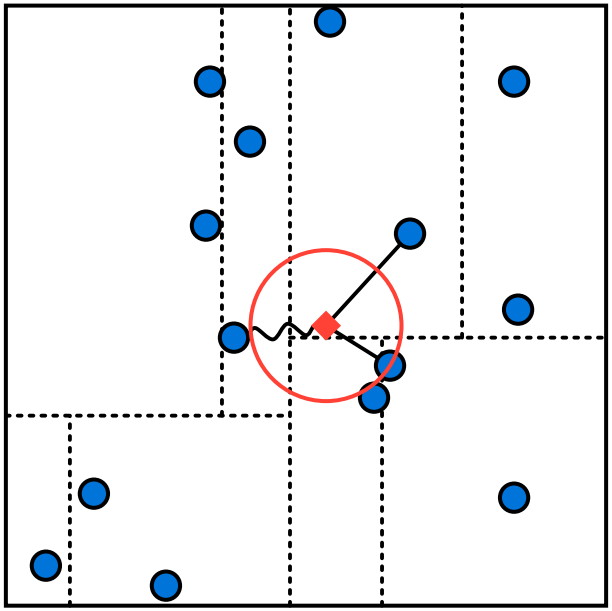
KNN in K-D Trees



KNN in K-D Trees



KNN in K-D Trees



1. Find the leaf containing the query point.
2. Find the nearest point in the leaf. Call this distance D_{min}.
3. Find an (i) unexplored leaf (ii) within D_{min} of the query point.
4. If the leaf contains a point closer than D_{min}, update D_{min}.
5. Repeat from 3 until all leaves within D_{min} explored.

KD Trees

Like a Binary (or B+ Tree), except directory nodes at level i partition on dimension $i \bmod \text{NUM\_DIMENSIONS}$

Scaling to more than 3 dimensions

The curse of dimensionality

1 dimension

- Length(Line) = Length(Line)

2 dimensions

- Area(Circle) = $\frac{4}{\pi}$ Area(Square)

3 dimensions

- Area(Sphere) = $\frac{1}{6\pi}$ Area(Cube)

The volume of a (hyper-)sphere becomes massively smaller than the area of its bounding (hyper-)cube at high dimensions

Embeddings

Suppose you have some complex, expensive measure of “similarity”

- Nurse, Doctor, Medtech
- CSE 462, CSE 562
- Dog, Cow

$$\text{Dog} \rightarrow f(\cdot) \rightarrow [0, 1, 1, 2, 1, 0]$$

$$\text{Cow} \rightarrow f(\cdot) \rightarrow [0, 1, 2, 2, 1, 0]$$

$$d(f(A), f(B)) \approx \text{how similar A is to B}$$

The mechanics of f are out-of-scope for today's lecture, but $f(\cdot)$ is a very high-dimensional vector (1000s of dimensions).

At 1000s of dimensions, (hyper)cubic regions are ineffective as filters.

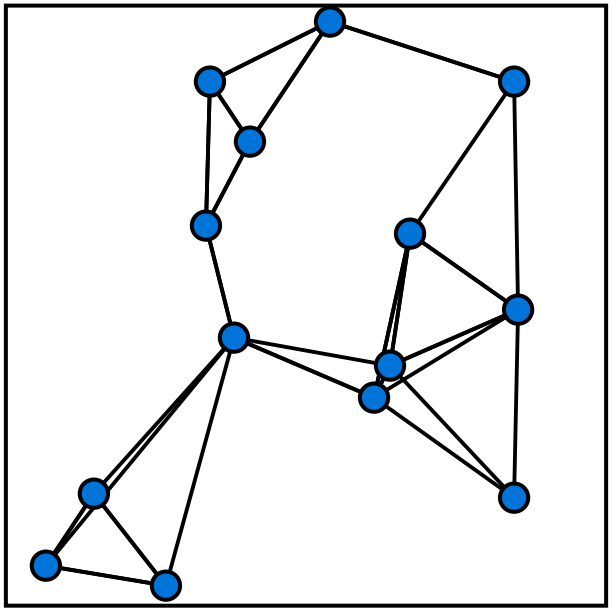
Graph Vector Indexes

Represent the vectors as a graph

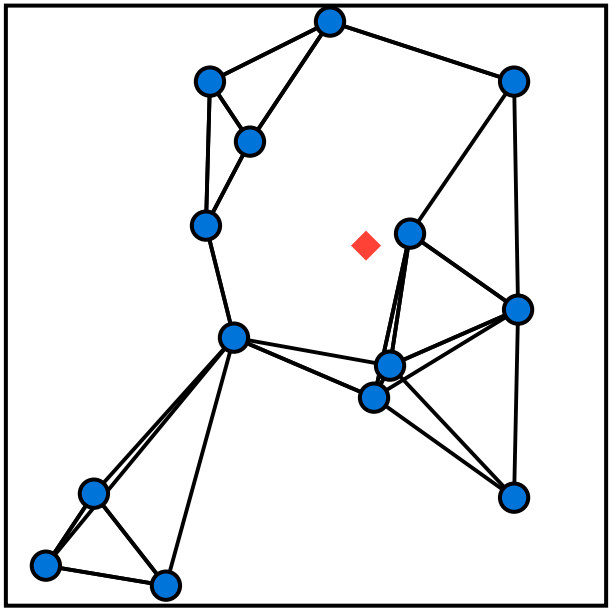
- Each vector to be stored is one node
- Each node has edges to the ℓ closest nodes
  - Compute this once by brute force.

This structure can be encoded as a Node and an Edge table with appropriate indexes.

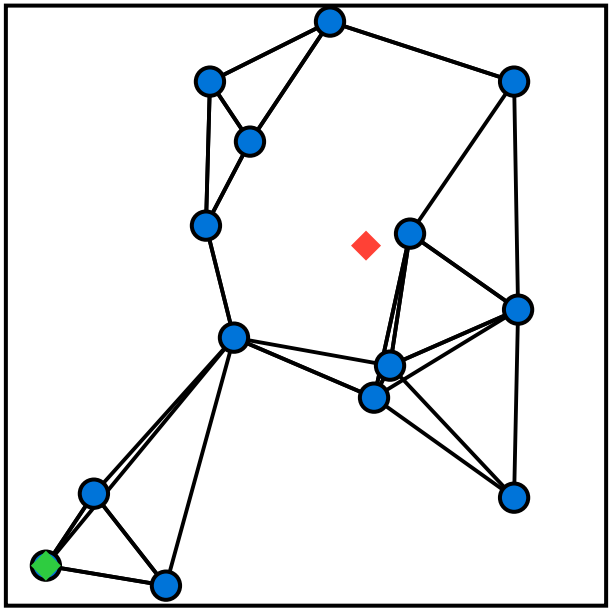
Graph Vector Indexes



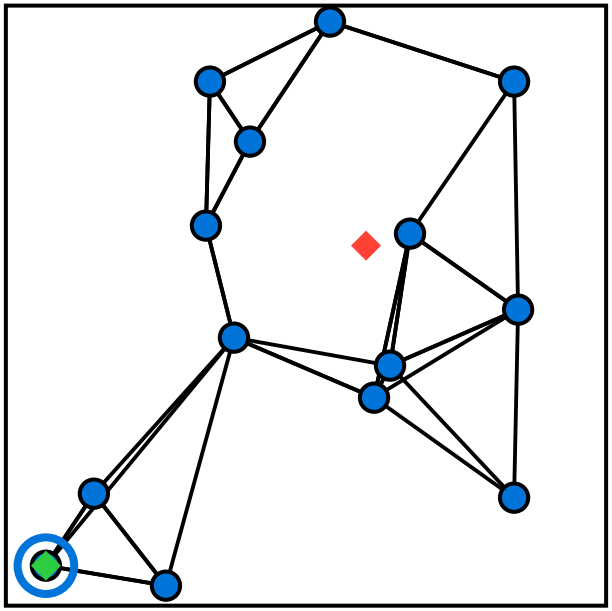
Graph Vector Indexes



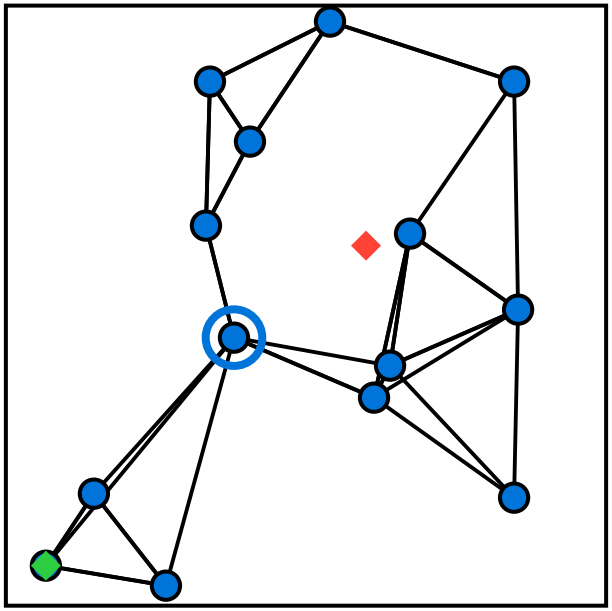
Graph Vector Indexes



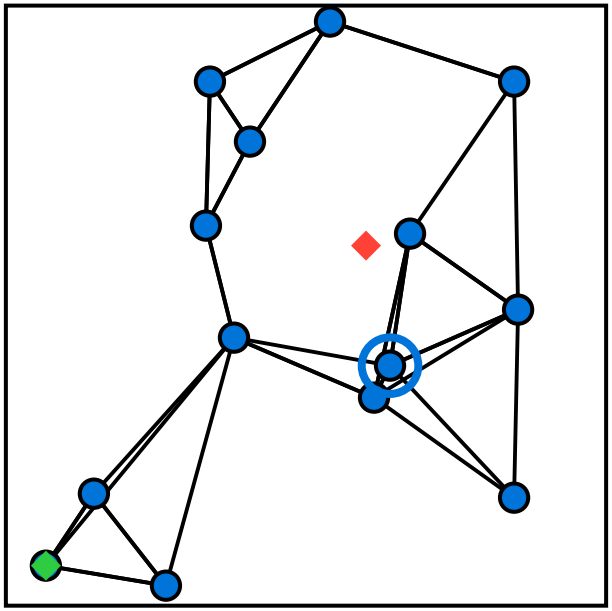
Graph Vector Indexes



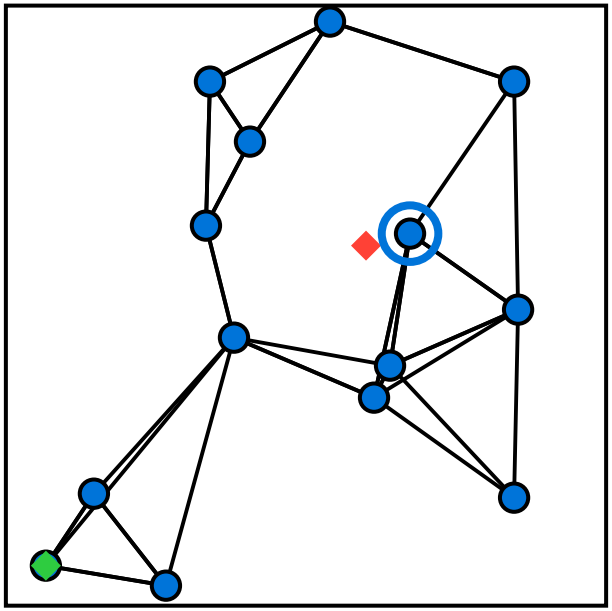
Graph Vector Indexes



Graph Vector Indexes



Graph Vector Indexes



Nearest Neighbor Graphs


```
def nearest_neighbor(self, query, epsilon):
  start = self.pick_any_node()

  best = start
  d_best = distance(start, query)
  todos = PriorityQueue( (d_best, start) )

  while (d_next, next) := todos.dequeue() is not None
          and d_next < d_best + epsilon:
    if d_next < d_best:
      best = next
      d_best = d_next

    for neighbor in self.neighbors(next):
      todos.enqueue( (distance(neighbor, query), neighbor) )
```

Nearest Neighbor Graphs

K-Nearest Neighbor is the same, but with a reservoir.

- Keep K closest points
- End when the worst point (plus epsilon) is closer than the next closest todo.

Problems

1. Isolated clusters can form.
2. The path to the closest node can be long.

NSW and NSG Indexes

Idea: Bias edges towards nodes for points that are more distant from each other.

Given a point p_1 with edges to nodes in W.

Include an edge to p_2 if

1. $|W|$ is below a desired threshold, and
2. $\text{dist}(p_1, p_2) < \min(\text{dist}(p_2, p_3) \mid p_3 \in W)$

Reducing Distances

Create a second graph by sampling from the first with probability p

1. Find the closest node in “layer” 2
2. Use this node to start your search in layer 1

Reducing Distances

Create a third graph by sampling from the second with probability p, ... and a fourth ...

- Layer 1 has 100% of nodes
- Layer 2 has 100% $\cdot\, p$ of nodes
- Layer 3 has 100% $\cdot\, p^2$ of nodes
- Layer i has 100% $\cdot\, p^{i-1}$ of nodes

Assuming a sufficiently uniform sample, each layer gets you a factor of p closer.

Each node can be sampled individually:

- All layers up to $M_{\max}$: $p^{M_{\max}-1}$
- All layers up to i: $p^{i-1} \cdot (1-p)$
- Only layer 1: $1-p$

NSWN Indexes

- Encode vertices as nodes in a graph; Edges connect nearby neighbors
- Layered graph structure skips to nearby nodes
- Useful for KNN-style queries over very high dimensional data