Data structures and algorithms for proximity search

A: Tree-based proximity search in Euclidean space

The complexity of proximity search

Given n data points in $\mathbb{R}^d$, how long does it take to find the nearest neighbor of a query point q?

The complexity of proximity search

What if $d = 1$? Is there a clever data structure that can be used? What is the preprocessing time and query time?

Data structures for proximity search

Given a data set of n points in $\mathbb{R}^d$ (or in a more general distance space), build a data structure for efficiently answering subsequent nearest neighbor queries q.

- Data structure should take space $O(n)$
- Query time should be $o(n)$

Many data structures have been designed for this purpose, e.g.,

1. K-d trees
2. Ball trees
3. Locality-sensitive hashing

These are part of standard Python libraries for NN, and help a lot.

Spatial data structure: k-d tree

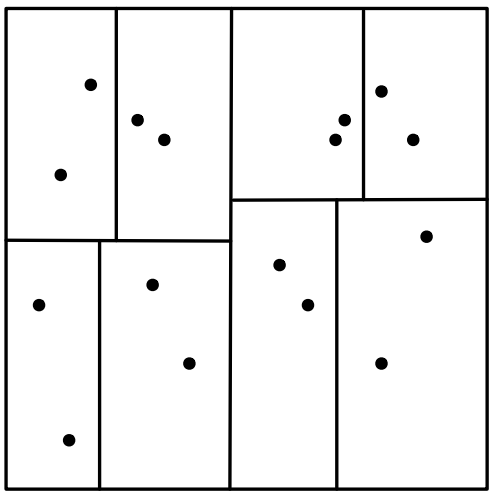
A hierarchical, rectilinear spatial partition.

For data set $S \subset \mathbb{R}^d$:

- If $|S| \leq n_o$: return leaf cell containing S
- Pick a coordinate $1 \leq i \leq d$
- $v = \text{median}(\{x_i : x \in S\})$
- Split S into two halves:

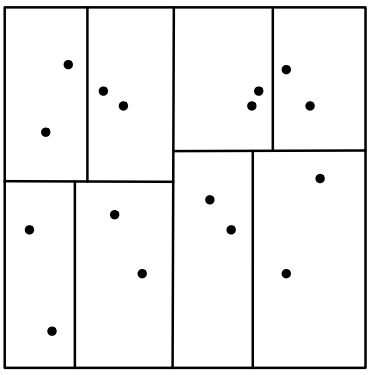
$$S_L = \{x \in S : x_i < v\}$$
$$S_R = \{x \in S : x_i \geq v\}$$

- Recurse on S_L, S_R



What is the height of the tree? And how long does it take to build?

Proximity search with a k-d tree

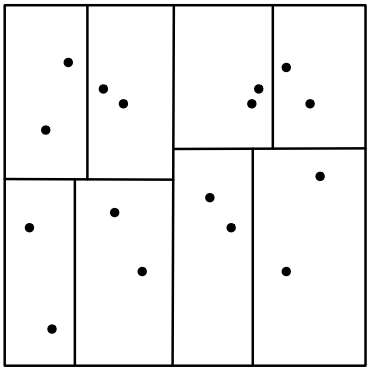


Defeatist search for query q:

- Move q down to its leaf-cell
- Return closest point in that cell
- Time: $O(\log(n/n_o)) + O(d \cdot n_o)$

Problem: Might not return the actual nearest neighbor.

Proximity search with a k-d tree



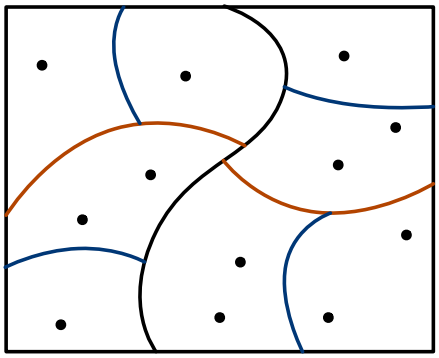
Comprehensive search for query q:

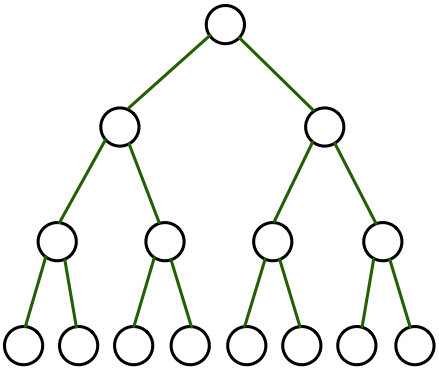
- Go to q's leaf cell, find closest point in that cell
- Backtrack up the tree, looking for closer points and using geometric reasoning (triangle inequality) to decide if a subtree can be ignored
- Always returns the nearest neigbor
- Worst case: look at all points, $O(dn)$

B: Tree-based search in metric spaces

Data structure: Ball tree

Instance space $\mathcal{X}$ with metric d.





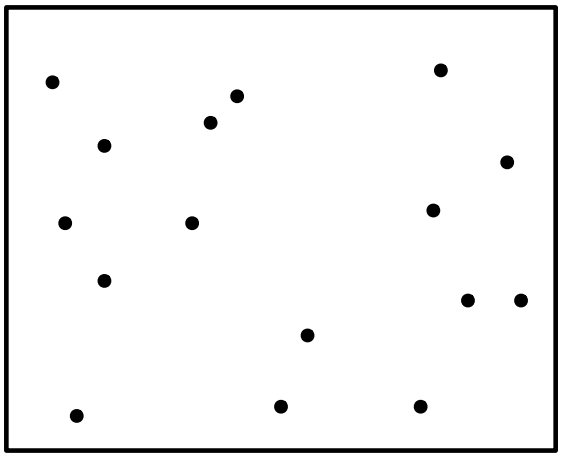
Ball tree for a set of points $S \subset \mathcal{X}$:

- Hierarchical partition of S with cells organized in a tree
- Each node of the tree has an associated ball

$$B(z, r) = \{x \in \mathcal{X} : d(x, z) \leq r\}$$

that contains all points in that node

Building a ball tree

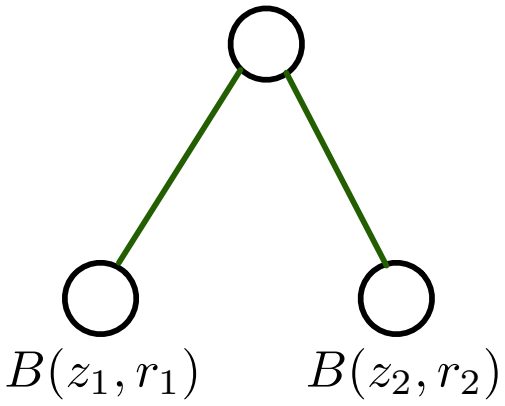


Lots of flexibility in how to split a cell, e.g.

- Pick two points in the cell
- Partition the cell according to which of those two points is closer

Defeatist search with a ball tree

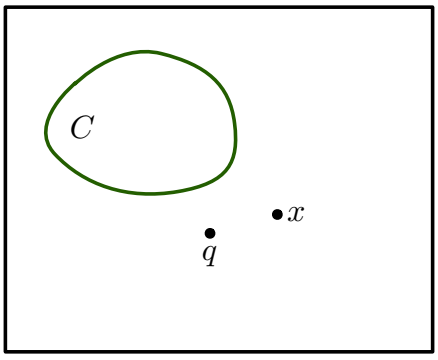
Given a query q, how do we move down the tree?



When you get to a leaf: return the nearest neighbor in that leaf.

Comprehensive search with a ball tree

Suppose the closest point we've found so far (to query q) is x.
How to decide whether we need to explore a cell C?



Let the bounding ball of C be $B(z, r)$. Then we need to go into C if:

C: The curse of dimension in proximity search

The curse of dimension in nearest neighbor search

Situation: n data points in $\mathbb{R}^d$.

1. Not good: Storage is $O(nd)$
2. Not good: Time to compute distance is $O(d)$ for ℓ_p norms
3. But worst of all: Geometry
   It is possible to have $2^{O(d)}$ points that are roughly equidistant from each other.

The best current methods for exact nearest neighbor search have *worst-case* query time proportional to 2^d and $\log n$.

The nightmare scenario in proximity search

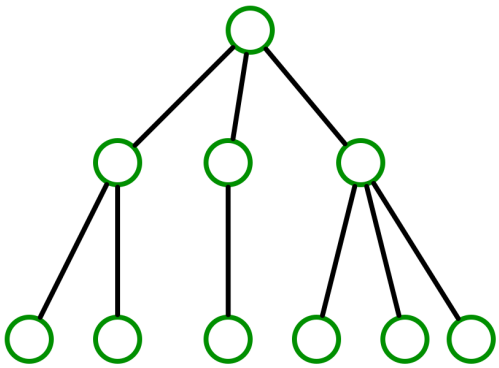
For any $0 < \epsilon < 1$,

- Pick $2^{O(\epsilon^2 d)}$ points uniformly from the unit sphere in $\mathbb{R}^d$
- With high probability, all interpoint distances are $(1 \pm \epsilon)\sqrt{2}$

How can this bad case be defeated?

1. Most real data doesn't look like a uniform distribution on a sphere.
   Data is often "intrinsically" lower-dimensional that it appears (e.g. lies on a d_o-dimensional manifold in $\mathbb{R}^d$, for $d_o \ll d$). Many methods for exact search can replace dependence on d by d_o.
2. Approximate nearest neighbor search

Cover trees for metric spaces



- Hierarchical cover of an arbitrary metric space
- Space $O(n)$, permits dynamic insertion and deletion of data points
- Query time $O(\text{poly}(c) \log n)$, for a dimension-related constant c

A finite set X in a metric space has **expansion rate** c if for any point x and any radius $r > 0$,

$$|B(x, 2r) \cap X| \leq c \cdot |B(x, r) \cap X|.$$

D: Approximate nearest neighbor search

Approximate nearest neighbor

For data set $S \subset \mathbb{R}^d$ and query q, a c-approximate nearest neighbor is any $x \in S$ such that

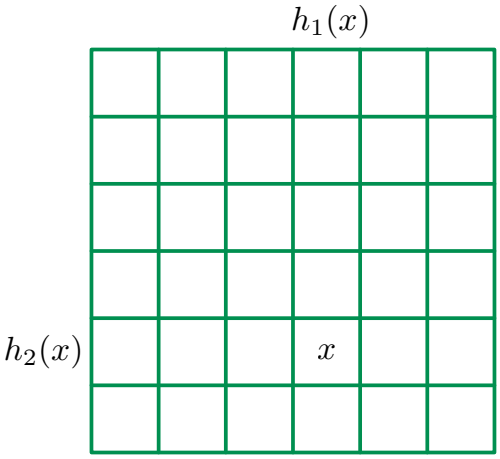
$$\|x - q\| \leq c \cdot \min_{z \in S} \|z - q\|.$$

Complexity of approximate NN search in Euclidean space:

- Data structure size n^{1+1/c^2}
- Query time n^{1/c^2}

This is based on **locality-sensitive hashing**.

Locality-sensitive hashing



Typical hash function h_i:
random projection + binning

$$h_i(x) = \left\lfloor \frac{r_i \cdot x + b}{w} \right\rfloor$$

- r_i is a random unit vector
- b is a random offset
- w is the bin width

- For any data set $x_1, \ldots, x_n$, query q: probability < 1 of failing to return an **approximate** NN.
- To reduce this probability, make t tables. Space: $O(nt)$.

Proximity data structures: an impressionistic history

- 1975: The k-d tree (Bentley and Friedman).
  Widely used, but algorithmic guarantees on weak footing.
- 1980s-1990s: More tree structures (e.g. Clarkson, Mount).
  Could accommodate general metric spaces.
- 1990s-: It's okay to fail sometimes (e.g. Clarkson, Kleinberg).
- Late 1990s-: Locality-sensitive hashing (Indyk, Motwani, Andoni).
  Hashing scheme with some failure probability, widely used.
- Recently: many variants of hashing; resurgence of trees.