Similarity Search

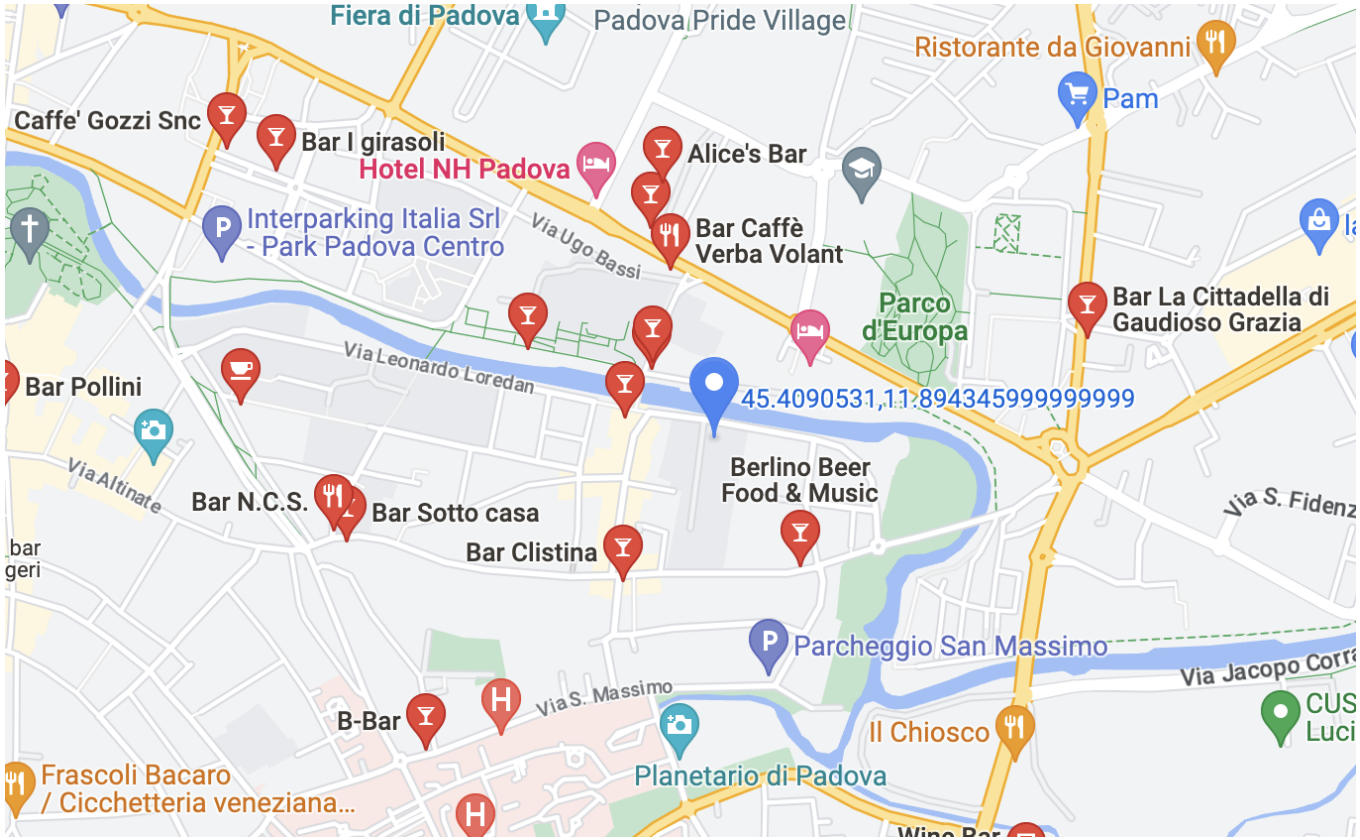
(Part 1)

OUTLINE

1. Introduction to similarity search
2. Similarity search in low dimensions
   - kd-trees
   - Curse of dimensionality
3. Similarity search in high dimensions (Part 2)
   - Approximate Near Neighbor (ANN) search
   - Locality Sensitive Hashing (LSH)

Introduction to similarity search

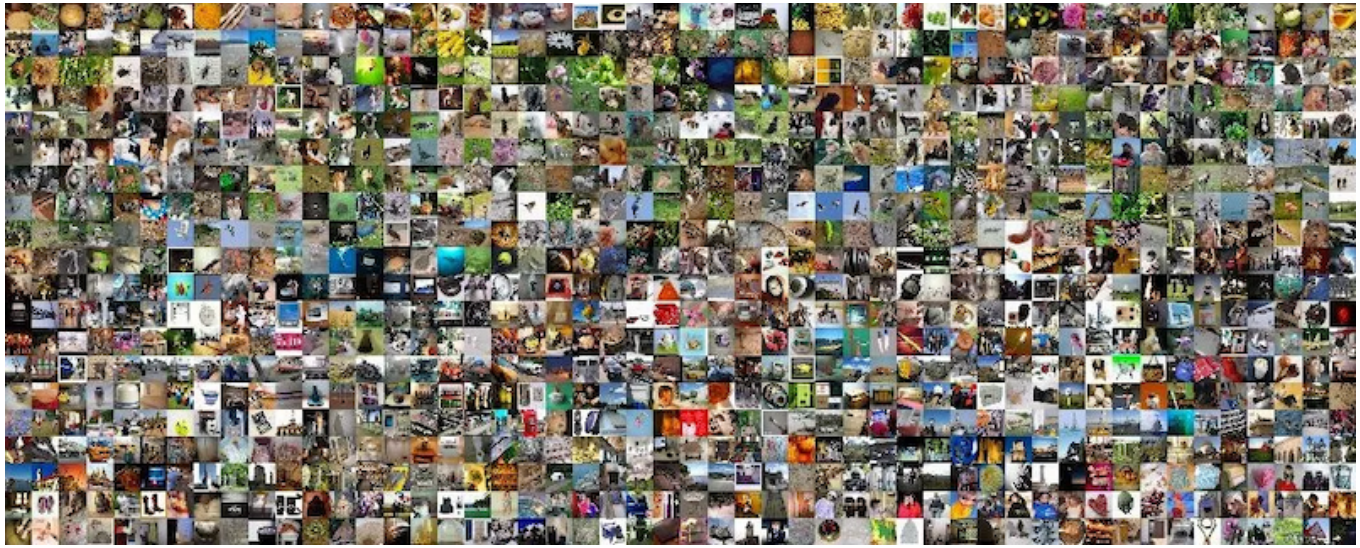
Searching near POIs



DATA: Points of interest (POIs) in a map, a distance function, a threshold r

GOAL: Find POIs near my current position (e.g., distance $\leq r$)

Searching similar images



DATA: A set of images, a distance function, a threshold r

GOAL: Find images similar to a given image (i.e., within distance $\leq r$)

- Similarity can be comptued by embedding images into high-dimensional vectors with a machine learning model, and then compute the distance between the respective vectors (e.g., Euclidean distance, angular distance, …)
- Similar problem for songs (e.g., Shazam)

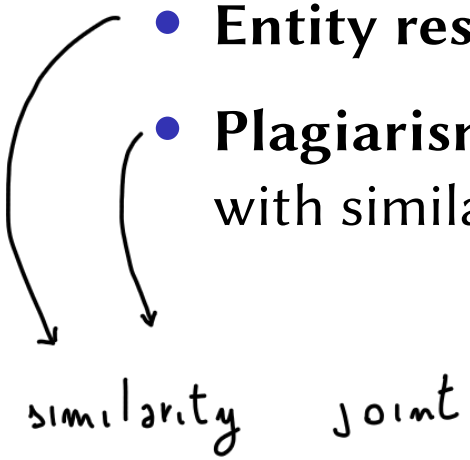
Similarity Search

- The two problems are two examples of similarity search: given a set of objects, find items that are similar (or near).
- A distance function is required to evaluate the similarity of two objects: two objects are similar (or near) if their distance is below a given threshold.
- Main difference between the two examples: points in the map have low dimensions (points belong to $\mathbb{R}^3$), images are represented by vectors with high dimensions (e.g. in $\mathbb{R}^c$ with $c >> 100$).

Similarity Search: applications

Some applications:

- **Information retrieval**: search similar documents, images, ...
- **Recommendation systems**: provide suggestions based on behaviors of similar users.
- **Entity resolution**: match equal or similar records.
- **Plagiarism detection**: match documents (e.g., papers, exams) with similar structure.



r-Near neighbor search

For a given metric space (M, d), point $q \in M$, and distance threshold $r > 0$ define the ball of radius r around q as

$P \subseteq M$

$$B_r(q) = \{p \in M \ : \ d(p, q) \leq r\}$$

Definition: r-Near neighbor search (r-NNS) problem

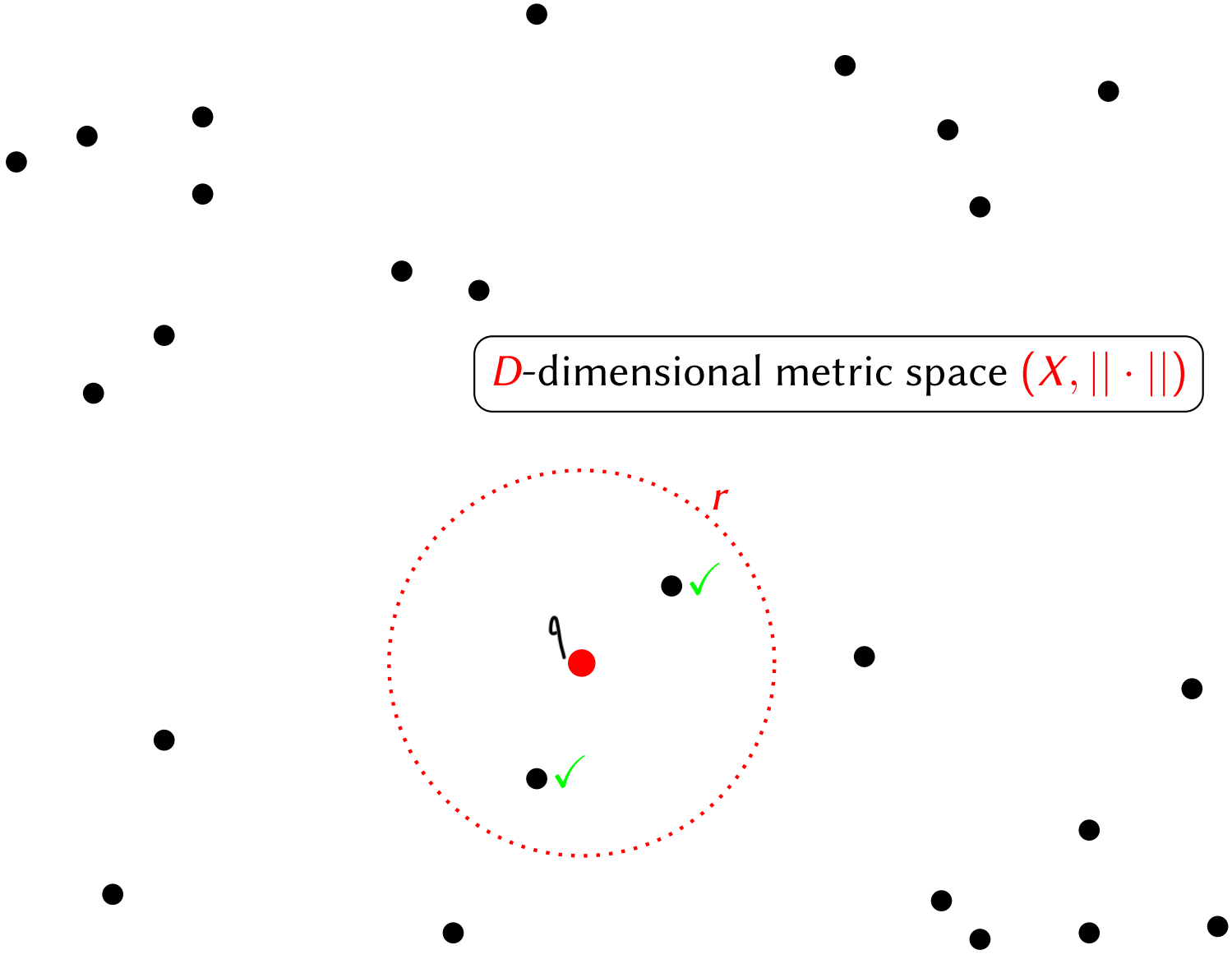
Given a set P of n points from the metric space (M, d), construct a data structure that, given a query point $q \in M$ and a distance threshold $r > 0$, returns:

- a point $p \in B_r(q) \cap P$, if any such point exists;
- `null` if $B_r(q) \cap P = \emptyset$.

Observations:

- The query is unknown at the construction of the data structure
- If there are more points in $B_r(q) \cap P$, an arbitrary one is returned.
- The distance function and the threshold r depend on the application.

Example of r-NNS



Other similarity search problems

There are several other variants of similarity search problems e.g.,:

- r-near neighbor reporting: given the set P, a query point q, and a threshold r, return *all* points in $B_r(q) \cap P$.
- Nearest neighbor search: given the set P and query point q, find the *closest point* to q in P. That is, return $\text{argmin}_{p \in P} d(p, q)$.
- k-nearest neighbor search: given the set P, a query point q,and an integer $k \geq 1$, find the top-k closest points to q in P.
- Similarity join: given two sets P and Q and a threshold r, find all similar pairs between the two sets. That is, find all $p \in P, q \in Q$ with $d(p, q) \leq r$.

Problems searching for the nearest point do not require a threshold r; however, they are usually solved by reducing to a r-near problems using a suitable sequence of r values.

Computational setting

We will consider the sequential setting and will use the following performance indicators:

- Construction time: the time to construct the initial data structure;
- Space: the space (in memory words) to store the data structure;
- Query time: the time to execute one query. ⟶ what we want to optimize

For points in $\mathbb{R}^D$ we assume that $d(x, y)$ can be computed in $O(D)$ time and that a point is stored in $O(D)$ space.

Remark: Several works have addressed similarity search in the MapReduce and streaming models.

Brute force solution

Construction

- Store points of P in a list.

Query q

- Compute $d(q, p)$ for all $p \in P$ returning the first point $p' \in P$ with $d(q, p') \leq r$, if one exists, and returning `null`, otherwise.

The brute force approach requires:

- $O(Dn)$ construction time;
- $O(Dn)$ space;
- $O(Dn)$ query time.

IMPORTANT: **The query time is too high!**

Similarity search in low dimensions

r-NNS and Range Reporting in $\mathbb{R}^D$

To solve more efficiently the r-NNS problem in $\mathbb{R}^D$ under Euclidean distance, we use the kd-tree

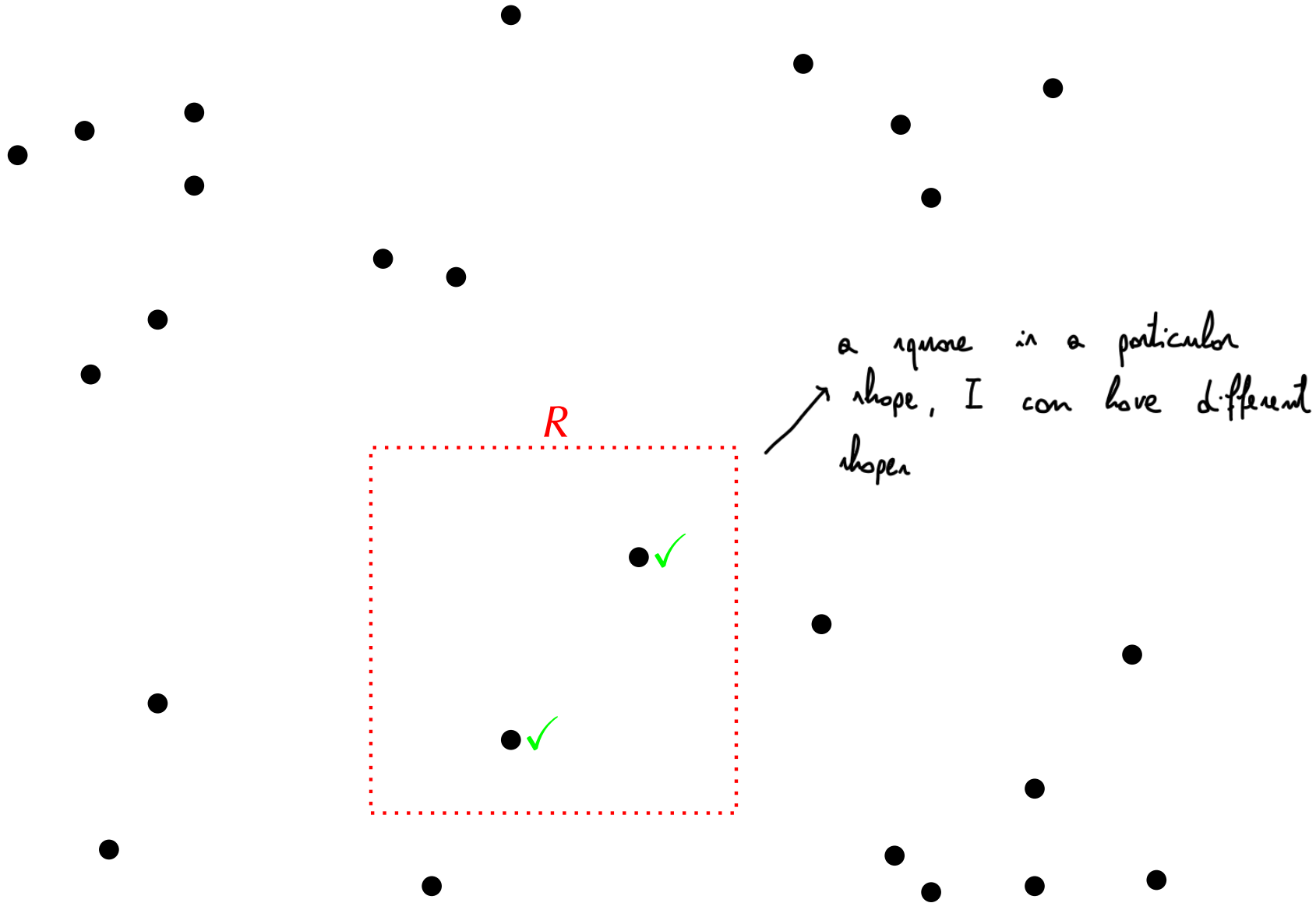
- A kd-tree is a binary tree that recursively partition the space into rectangular regions.
- kd-tree solves a slightly different problem named Range Reporting.

Definition: Range Reporting (RR) problem

Given a set $P \subset \mathbb{R}^D$ of n points, construct a data structure that, given a rectangular region R, returns all points of P contained in R.

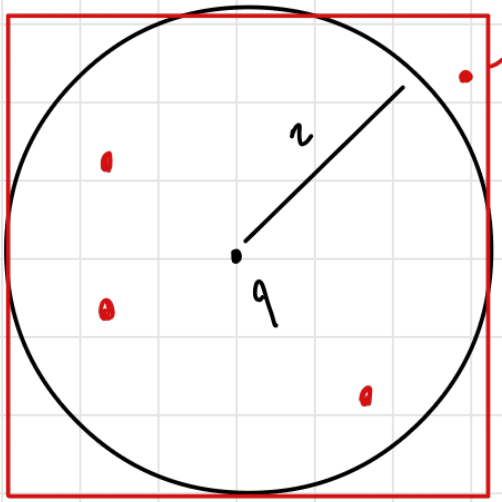
Note: a rectangular region $R = [x_{1,1} : x_{1,2}] \times \ldots \times [x_{D,1} : x_{D,2}] \subset \mathbb{R}^D$ is the region including all points $p = (p_1, \ldots, p_D)$ with $p_i \in [x_{i,1} : x_{i,2}]$ for all $i \in \{1, \ldots D\}$.

Range Reporting in $\mathbb{R}^2$



Range Reporting vs *r*-NNS

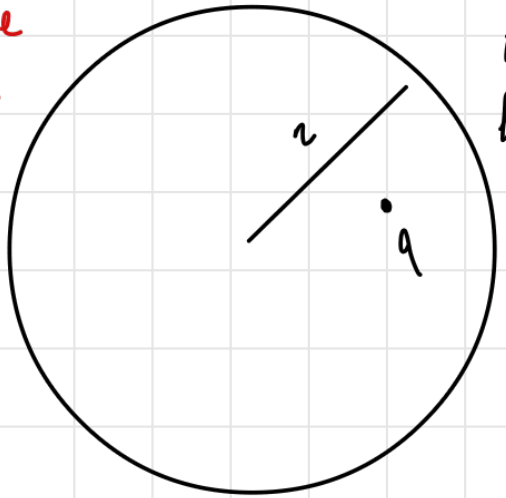
r-NNS



false positive

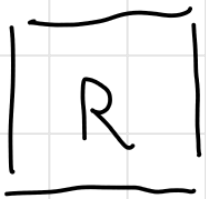
r-NNS must one

side 2r

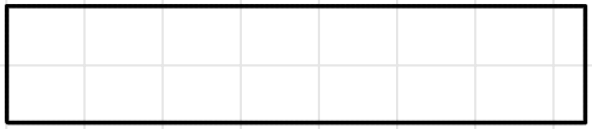


since we have a range reporting ⇐ we report ALL the one inside hence it's easy then to detect the false positive

Range reporting q



R'



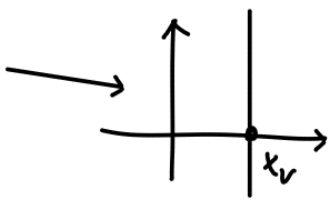
R''

kd-trees

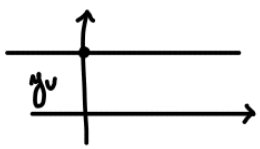
For simplicity we define the kd-tree for the case $D = 2$. The definition can be extended to $D > 2$.

A kd-tree for a set P of n points in $\mathbb{R}^2$ is defined as follows:

- Each node v represents a subset $P_v \subseteq P$. In particular, the root represents P and the leaves represent distinct individual points ($\rightarrow n$ leaves).
- Each internal node v represents the set of points $P_v \subseteq P$ at the leaves of its subtree (hence, the root represents P) and is associated with a line ℓ (vertical or horizontal depending whether v is at even or odd depth) which splits P_v into 2 subsets $P_{v,1}$ and $P_{v,2}$ of size $\lceil |P_v|/2 \rceil$ and $\lfloor |P_v|/2 \rfloor$, respectively. The left child of v represents $P_{v,1}$ and the right child represents $P_{v,2}$. If ℓ is a vertical line at abscissa x_v, then
  - $P_{v,1}$ contains all points of P_v with abscissa $\leq x_v$;
  - $P_{v,2}$ contains all points of P_v with abscissa $> x_v$

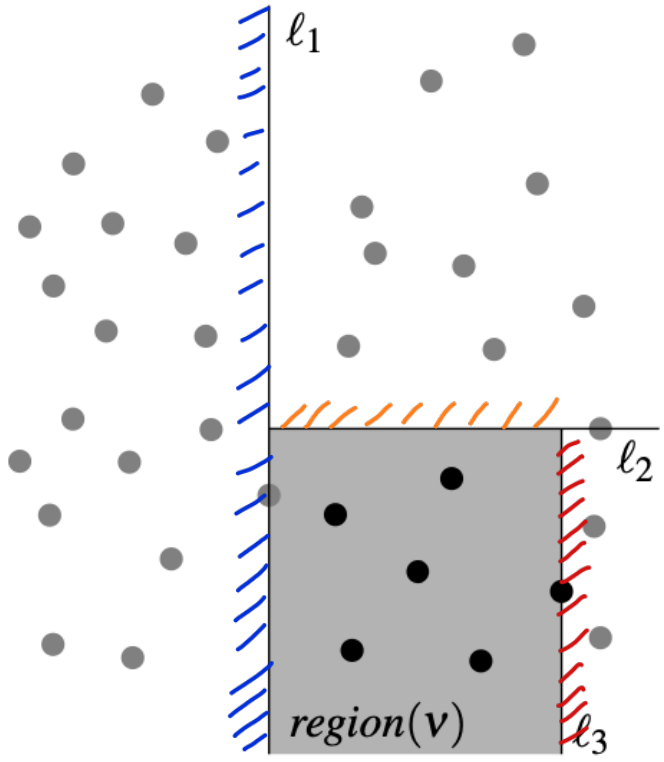


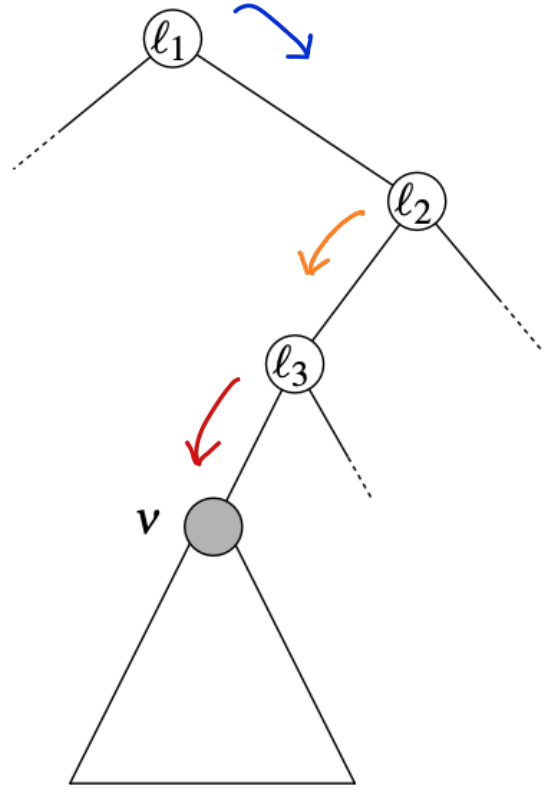
The case when ℓ is horizontal is analogous.



kd-trees

Each node v represents a rectangular region defined by the vertical/horizontal lines associated to path from the root to v.

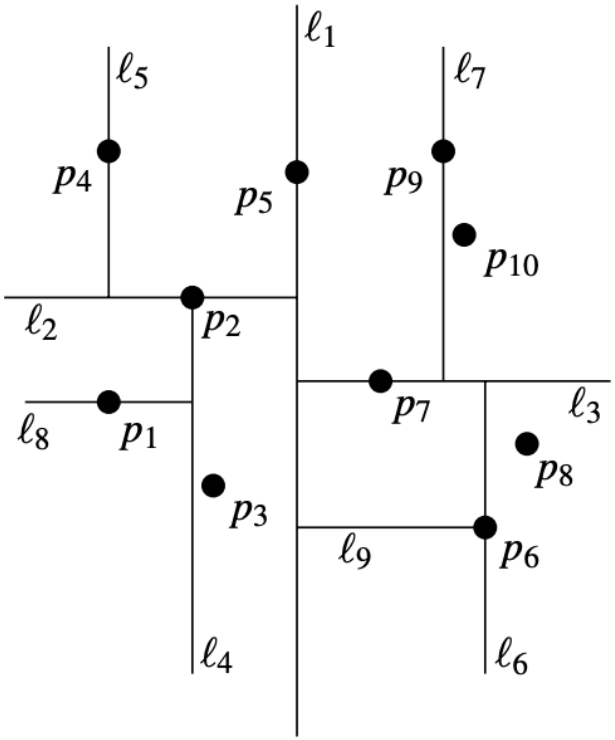


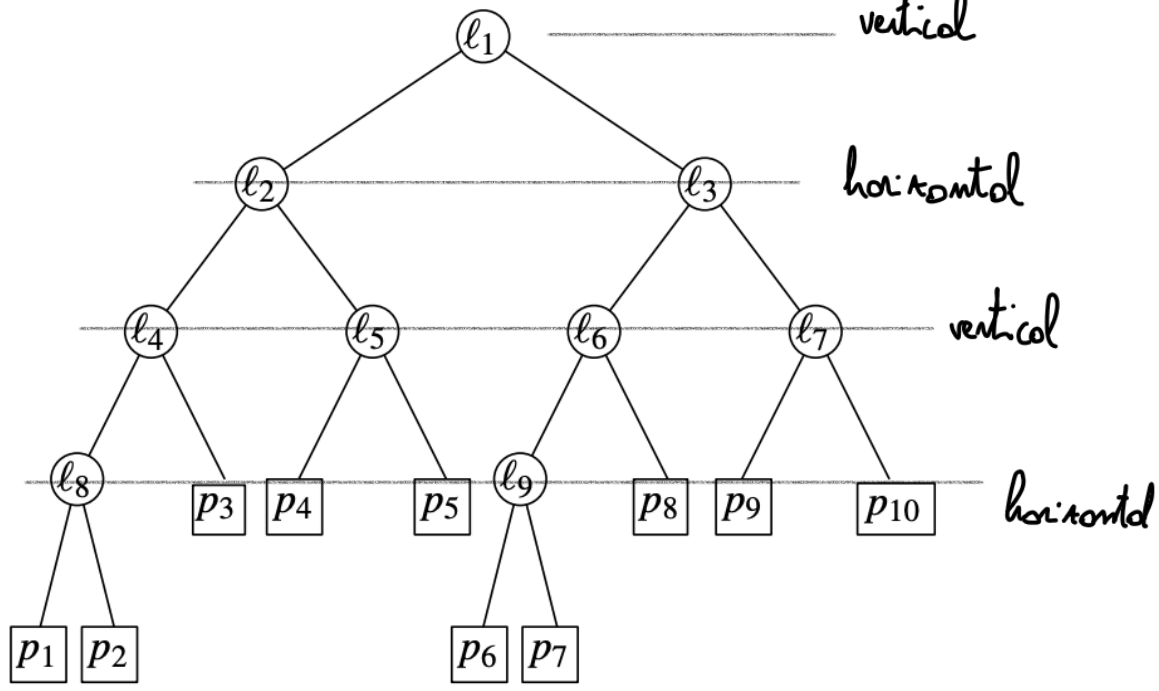


kd-tree: example

WE CAN CHANGE THE ORDER

just stay consistent





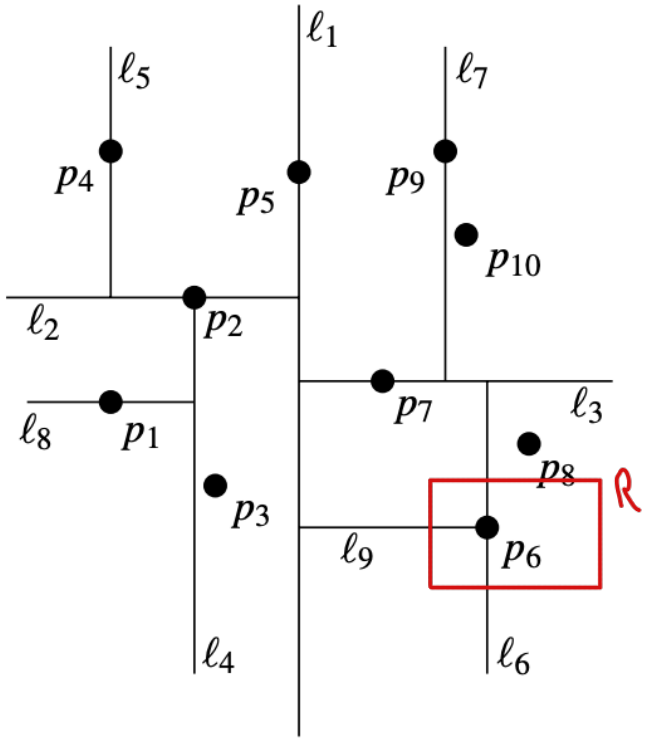
kd-tree: query algogrithm

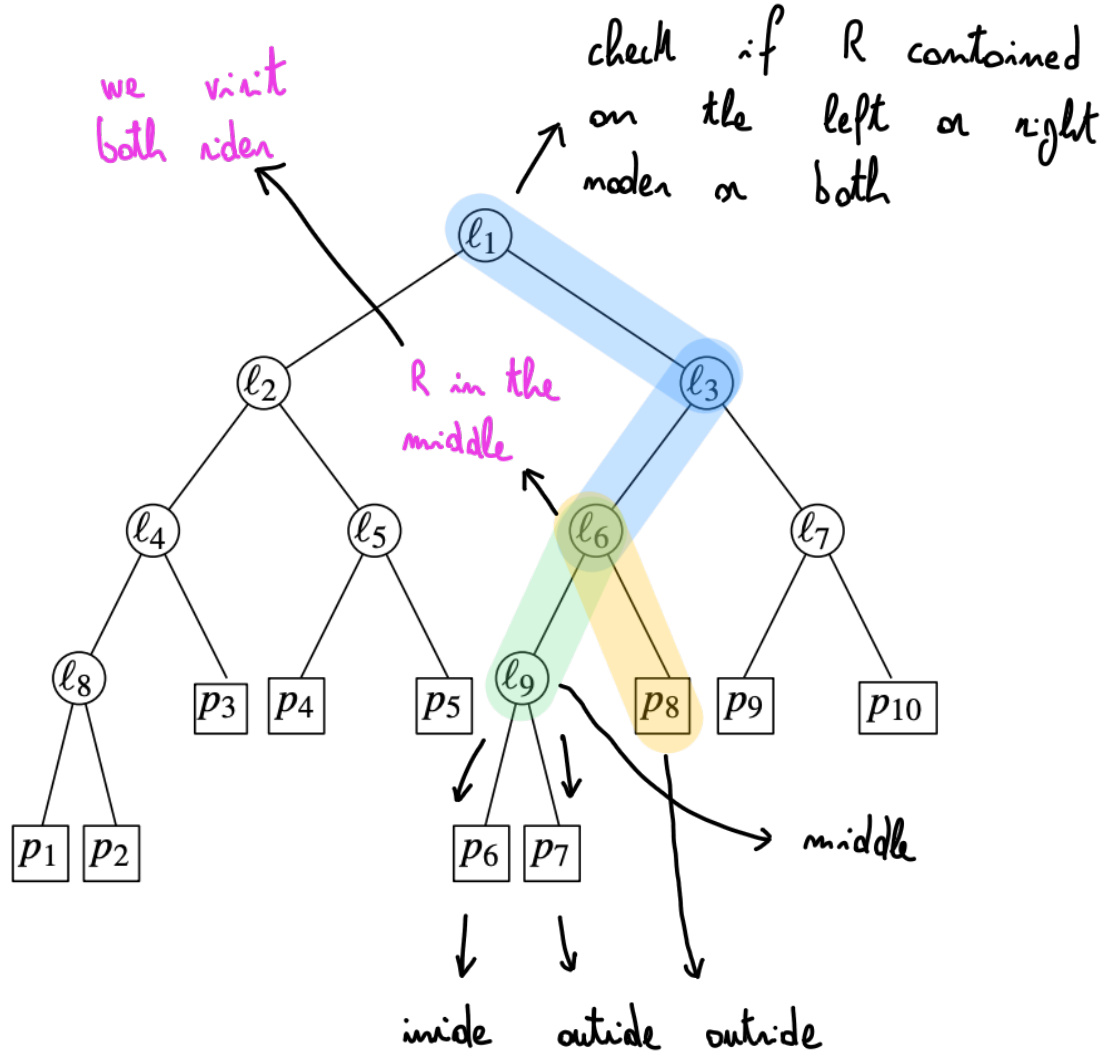
- Given a query region $R = [x_1, x_2] \times [y_1, y_2]$, the query algorithm visits all nodes whose regions intersect R.
- Given an internal node v associated with a vertical line at abscissa x_v (similarly for a horizontal line):
  - If $x_1 \leq x_v$, we recursively search v's left subtree;
  - If $x_2 > x_v$, we recursively search v's right subtree;

  Note that *recursion might proceed on both subtrees*.
- When the recursive algorithm reaches a region entirely contained in R, it returns all points in it.

kd-tree: example of query

$R = [x_1, x_2] \times [y_1, y_2]$





kd-tree: query algorithm (details)

Consider a kd-tree T for a set of n points P.

Let `ReportAll`(v) be a primitive that given a node $v \in T$ returns all points associated with the leaves of the subtree of v.

- It is not difficult to show that `ReportAll`(v) can be implemented in time proportional to the number of returned points (which is also proportional to the number of nodes of the subtree of v).
- We will use this primitives, in the query algorithm

kd-tree: query algorithm (details)

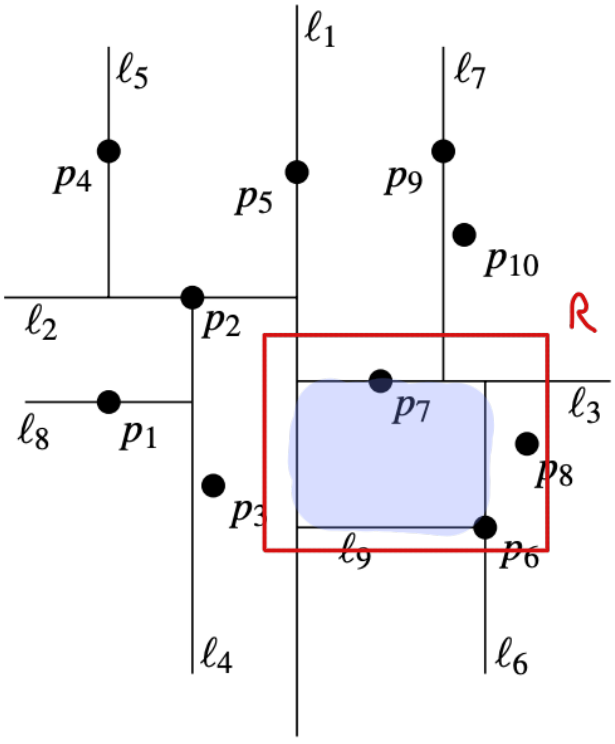
`SearchKdTree`(v, R) // *first invocation: v = root*

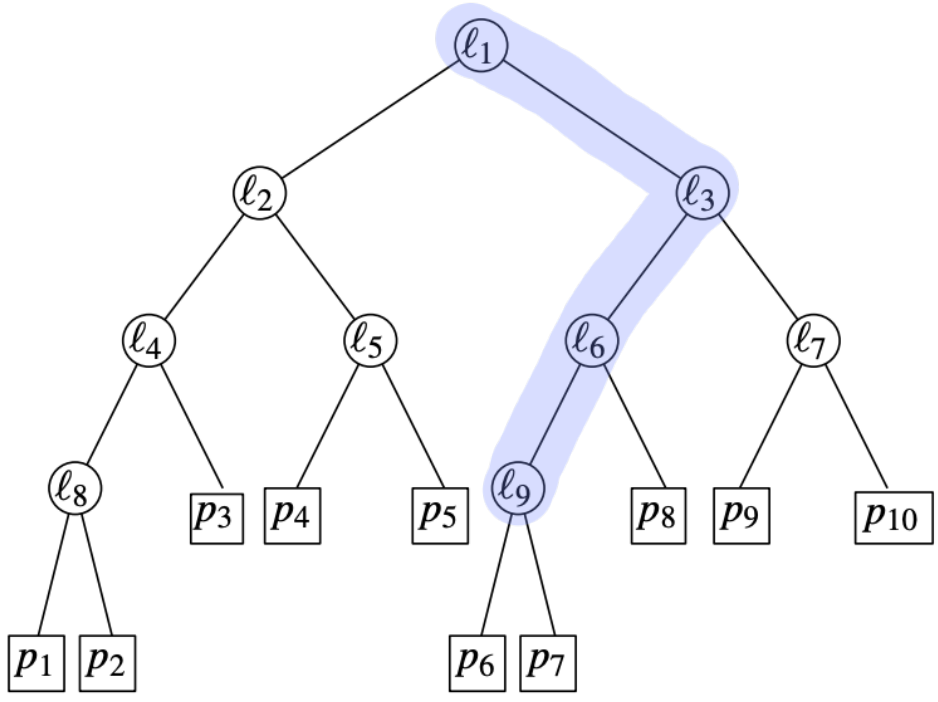
Input: A node v in the kd-tree, and a range R

Output: The set of points $p \in R$ from the subtree of v

```
out ← ∅;
if (v is a leaf associated to point p) then
    if (p ∈ R) then out ← {p};
else
    Let u, w be the left and right child of v, respectively;
    Let R_u, R_w be regions represented by u and w, respectively;
    if (R_u intersects R) then
        if (R_u ⊆ R) then out ← out ∪ ReportAll(u);
        else out ← out ∪ SearchkdTree(u,R);
    if (R_w intersects R) then
        if (R_w ⊆ R) then out ← out ∪ ReportAll(w);
        else out ← out ∪ SearchkdTree(w,R);
return out
```

kd-tree: example of query





kd-tree: performance in $\mathbb{R}^2$

Theorem

Let P be a set of n points in $\mathbb{R}^2$. Then, the Range Reporting problem can be solved using the kd-tree data structure with the following performance

- **Construction time:** $O(n \log n)$
- **Space:** $O(n)$
- **Query time:** $O(\sqrt{n} + k)$, *where k is the number of reported points.*

Intuition on the query time

Consider the execution of `SearchKdTree` on T storing a set P of n points for a query region R.

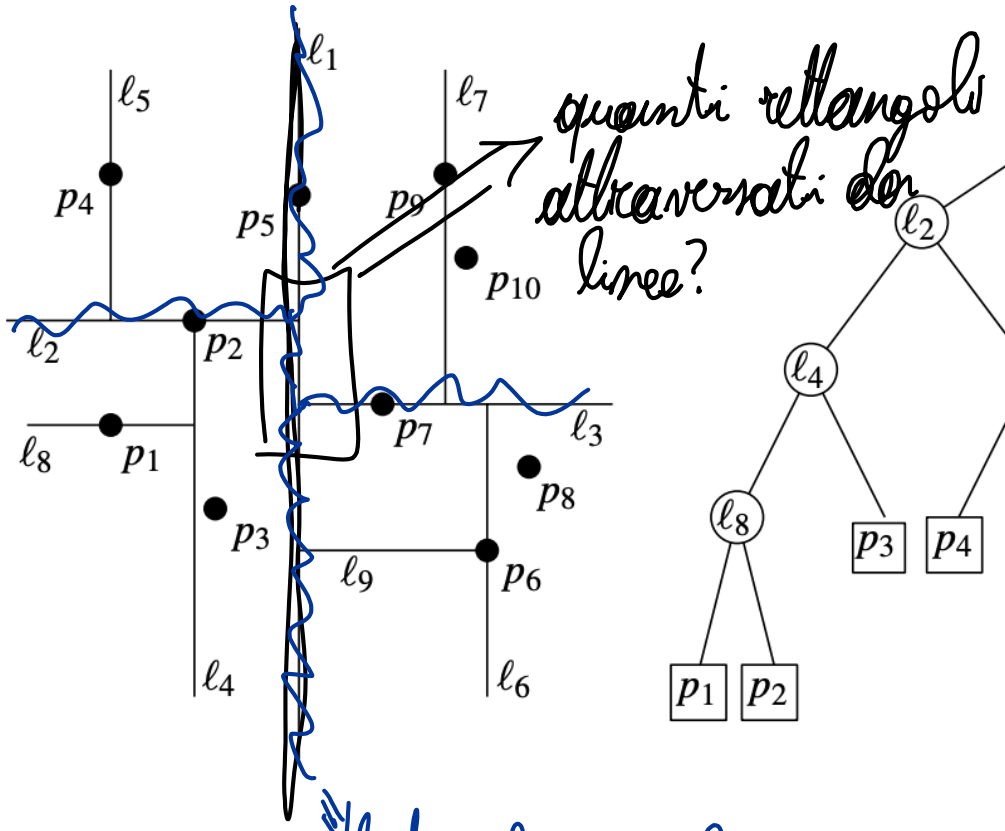
Let $Q(R)$ be the set of nodes of T on which `SearchKdTree` or `ReportAll` is called.

We have that $Q(R) = Q_1(R) \cup Q_2(R)$, where

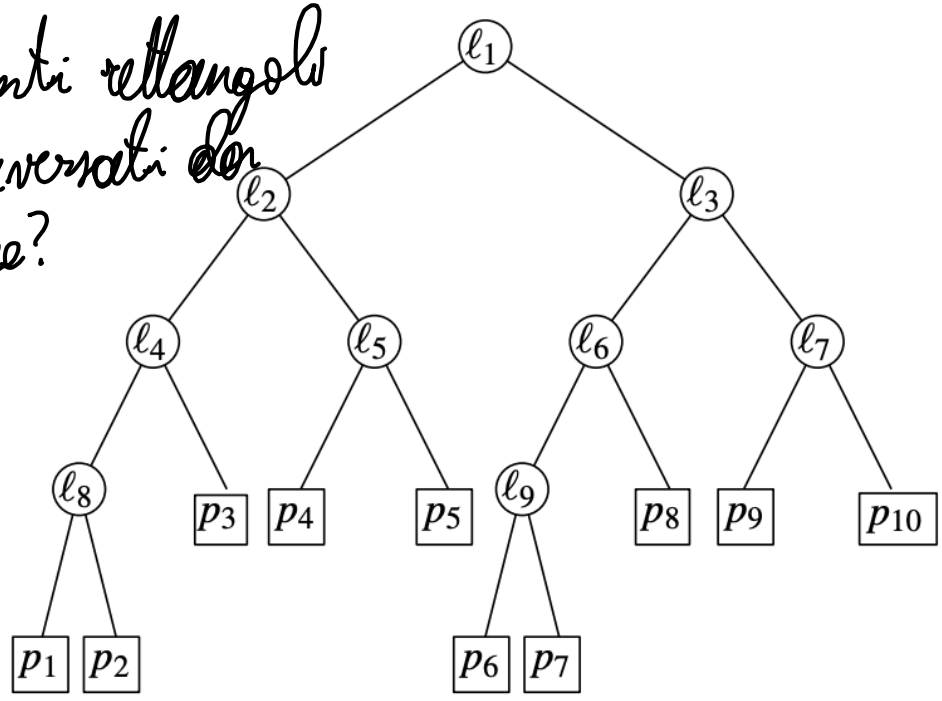
- $Q_1(R) =$ nodes whose regions intersect R but are not entirely contained in R;
- $Q_2(R) =$ nodes whose regions are entirely contained in R (hence `ReportAll` is run on each of them);

Intuition on the query time

nodi visitati senza nodi "buoni": $P(n) = 2P\left(\frac{n}{4}\right) + 2 = O(\sqrt{n})$



quanti rettangoli attraversati da linee?



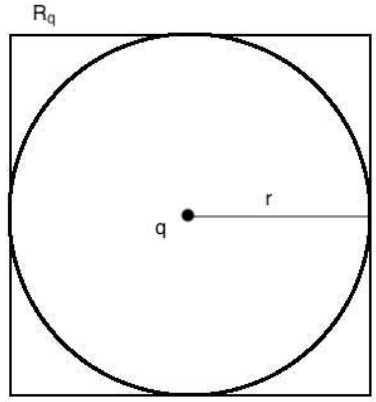
dati da ℓ_2 e ℓ_3

Intuition on the query time

It can be argued that

- $|Q_1(R)| = O(\sqrt{n})$;
- The call on each node of $Q_1(R)$ contributes $O(1)$ time to the overall query time;
- The calls on all nodes of $Q_2(R) =$ contribute, aggregately, $O(k)$ time to the overall query time.

kd-tree for r-NNS



soluzione esatta in buon tempo

To solve an r-NNS query with center $q \in \mathbb{R}^2$ on the input set P:

- Compute the smallest square R_q enclosing the ball of radius r and center q.
- Let S be the output of an RR query with range R_q on a kd-tree of P.
- Remove from S all points in R_q but not in the ball (corner cases).

kd-tree for r-NNS

- The solution works efficiently from a practical point of view.
- Let k_R be the number of points returned by the R_q query, and let k_q be the actual number of points in the ball with center q and radius r.
- The above solution requires $O\left(\sqrt{n} + k_R\right)$ not $O\left(\sqrt{n} + k_q\right)$; in some pathological cases, $k_R >> k_q$.
- Not clear how to close the gap from a theoretical point of view.

Exercise

Exercise

Let P be a set of n points in $\mathbb{R}^2$ and let T be a kd-tree for P. Suppose that in each node v of T an arbitrary point p associated with one of the leaves of v's subtree is stored. Show how to adapt `SearchKdTree` so that, given a query region R, returns an arbitrary point $p \in P$ belonging to R, if one exists, or `null` otherwise, in time $O(\sqrt{n})$.

Hint: use the properties stated before when arguing about the query time of `SearchKdTree`.

kd-tree: performance in $\mathbb{R}^D$

The above approach can be generalized to D dimensions.

- The kd-tree is defined in a similar fashion, but each level takes care of a distinct coordinate, in cycling order (i.e., $1, 2, \ldots, D, 1, 2 \ldots$).
- `SearchKdTree` remains unchanged

The following theorem generalizes the one for $\mathbb{R}^2$.

Theorem

Let P be a set of n points in $\mathbb{R}^D$. Then, the Range Reporting problem can be solved using the kd-tree data structure with the following performance

- **Construction time:** $O(Dn \log n)$
- **Space:** $O(Dn)$
- **Query time:** $O\left(Dn^{1-1/D} + k\right)$, *where k is the number of reported points.*

Curse of dimensionality

- The query cost of kd-tree converges to linear scanning time when D increases.
- There is a strong conjecture that the exact r-NNS problem cannot be solved in (truly) sublinear query time when D is large, i.e., a query time $O(n^{\epsilon})$ for a constant $\epsilon < 1$.