

k -dimensional trees (k -d trees)

In this tutorial, we will study the k -d tree data structure and its application to the Nearest Neighbours (NN) problem. The problem is to find the nearest neighbour for a query point from a given data set. It is broadly used in data analysis and machine learning.

As usual, the file `KDTree_test.py` contains several test functions to help you with debugging.

There are some optional complexity questions in this assignment. If you have the answers, you can put them in the comments alongside the corresponding functions as you have done for the midterm exam. *However, you are not required to do so.* Make sure you do *read* these questions.

1 Linear Scan: A straightforward Nearest Neighbour search

We start by implementing the Linear Scan algorithm for nearest neighbour queries in a database of k -dimensional points, each represented as an array of `float` values. We shall complete the function `linear_scan(query, P)` in `KDTree.py`, which computes the nearest neighbour, with respect to the Euclidean distance, of the query point `query` in the list of points `P`. To compute the distance between two points `p` and `q` you can use the function `dist(p,q)`, which is imported from the Python module `math`.

Question 1. Complete the function `linear_scan(query, P)`. This function takes as parameters: the query point `query` and the list of points `P` that is the database that we will scan to find the nearest neighbour to our query point. It proceeds by comparing the query point to every point in the database 1-by-1 (hence the name “linear scan”) and returns a point in the array `P` that is closest to the query point `query`. If there are multiple points in `P` at the same minimum distance to `query`, it returns any of them.

Clearly, the worst case complexity of Linear Scan is $O(n \cdot k)$, where n is the size of the point cloud and k is the dimension of the underlying space. Notice that, for query points that are *not in the database*, this is also the best case complexity since we have to check every point. In the subsequent sections, we will prepare and use k -d trees to improve on that best case complexity.

2 Building the k -d tree

As illustrated in Figure 1, a k -d tree is a binary tree wherein every node is a k -dimensional point. Every non-leaf node can be thought of as implicitly generating a splitting hyperplane that divides the space into two parts, known as *half-spaces*. The sub-trees of each node represent, respectively, the two sets of points on both sides of that hyperplane.

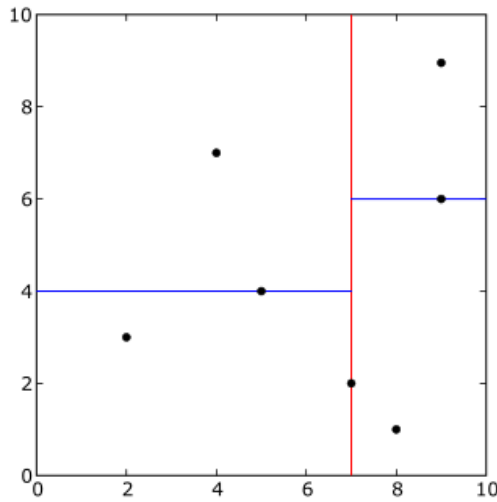
There are different ways of constructing a k -d tree given a cloud P of points of dimension k . We will proceed roughly as follows, using $P = [[2, 3], [5, 4], [9, 6], [4, 7], [8, 1], [7, 2], [9, 9]]$ with $k = 2$ (as in Figure 1) for illustration.

Start with the coordinate $c = 0$ and repeat the following steps:

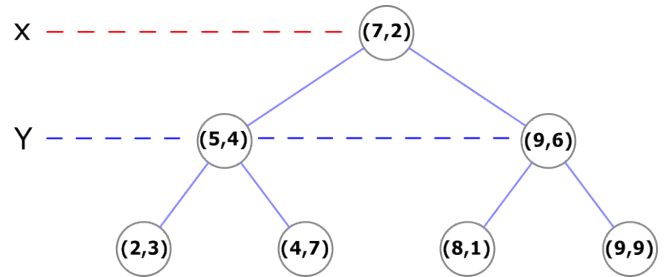
1. Sort the points of P along the coordinate c . In the example, for $c = 0$, we obtain:

$$P = [[2, 3], [4, 7], [5, 4], [7, 2], [8, 1], [9, 6], [9, 9]].$$

2. Find the median point m . In the example, for $c = 0$, we take $m = (7, 2)$.
3. Create a node holding that point.
4. Split the remaining points into two sub-clouds. In Figure 1a, this is represented by a vertical red line (orthogonal to the x axis) going through point $(7, 2)$.



(a)



(b)

Figure 1: A k -d tree decomposition for the point set $(2,3)$, $(5,4)$, $(9,6)$, $(4,7)$, $(8,1)$, $(7,2)$, $(9,9)$ (a)¹ and the resulting k -d tree (b)²

5. If no points are left in the cloud, stop.
6. Otherwise, use the coordinate $(c + 1 \bmod k)$ to recursively build
 - (a) the left sub-tree of the node from step 3 using the points before m in P (points $[[2, 3], [4, 7], [5, 4]]$ in the example), and
 - (b) the right sub-tree using the points after m in P (points $[[8, 1], [9, 6], [9, 9]]$ in the example).

IMPORTANT: To simplify your task, from now on we assume that every coordinate of every point in the database is different.

Question 2. Complete the function `partition(P, query, coord)`, which, given a point `query` in the database `P`, splits the points of `P` according to the `coord`-th coordinate of `query`. It returns a tuple of two lists `(L,R)`, where `L` contains the elements of `P` with `coord`-th coordinate strictly smaller than the one of `query`, and `R` contains the elements with `coord`-th coordinate strictly bigger than the one of `query`. Remember we assume that every coordinate of the points in `P` is different.

Question 3. Complete the function `MomSelect(P, coord, k)` which computes the k -th bigger element of `P` with respect to its `coord`-th coordinate. You have to implement the quick selecting strategy using *median of medians* that you learnt in the 3rd week of the course (Fig. 2). Remember that this algorithm is linear in the size of `P`.

We will use the class `KDTree`, which represents a binary tree with the following attributes:

- `k`, the dimension of the points
- `rootPoint`, the point in the root of the k -d tree.
- `coord`, the coordinate we use to split the reminding points.
- `left`, the k -d tree given by points with smaller `coord`-th coordinate than the one of `rootPoint`. This attribute is `None`, if there is no left child.³

¹Based on a diagram by KiwiSunset at the English-language Wikipedia, CC BY-SA 3.0, here

²Based on a diagram by MYguel – Own work, Public Domain, here

³The clause `Child is None` checks if the element `Child` is `None`.

Median-of-medians selection algorithm

MOM-Select(A,k):

```
n ← |A|
if ( n small enough ) :
    return k-th smallest element of A via mergesort
Group A into n/5 groups of 5 elements each (ignore leftovers)
B ← median of each group of 5
p ← MOM-Select(B, n/10) ← median of medians
(L, R) ← Partition(A,p)
if ( k < |L| ) return MOM-Select(L,k)
else if ( k > |L| ) return MOM-Select(R, k - |L| -1)
else return p
```

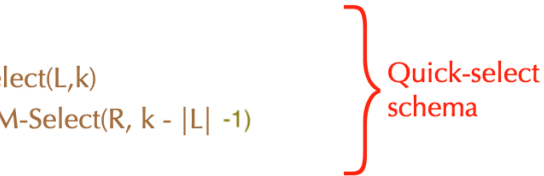


Figure 2: Quick select using Median of Medians, as seen in class

- **right**, the k -d tree given by points with bigger **coord**-th coordinate than the one of **rootPoint**. This attribute is **None**, if there is no right child.

Question 4. Implement the initialisation procedure `__init__ (self,P,coord = 0)` of the class, which takes a set of points and recursively constructs the k -d tree associated to it. As above, we assume that every coordinate of every point in the set is different.

This data structure will be used for the subsequent Nearest Neighbour queries.

3 Nearest Neighbour search via k -dimensional trees

Question 5. In the procedure `NN_exhaustive_search(self, query)` of the class `KDTree`, implement a recursive algorithm computing the Nearest Neighbour in the k -d tree of a given **query** point. For this, compare the distance of **query** to the root of the tree, with the distance of the Nearest Neighbour of **query** and the points in each sub-tree of the k -d tree.

Clearly, there is no reason to use k -d trees to compute the Nearest Neighbour using the exhaustive search algorithm as we could have done the same operation using the function `linear_scan` that we implemented at the beginning.

Now, we will consider the so-called “defeatist” approach for finding a candidate for the nearest neighbour of a query point **query** using a k -d tree. This consists in applying the same strategy as for the usual Binary Search Trees.

1. Let **c** be the coordinate associated to the root of the tree, let **p** be the point of its root, and let **med** be the **c**-th coordinate of **p**.
 - (a) if the **c**-th coordinate of **query** is less than or equal to **med**, proceed recursively on the left sub-tree. Let **cand** be the point computed by the “defeatist” approach in the left sub-tree. If the left sub-tree is empty, then **cand** is **None**.
 - (b) otherwise, proceed recursively on the right sub-tree and define **cand** accordingly.
2. If **cand** is not **None** and the distance between this point and **query** is smaller than the distance of **p** and **query**, return **cand**. Otherwise, return **p**.

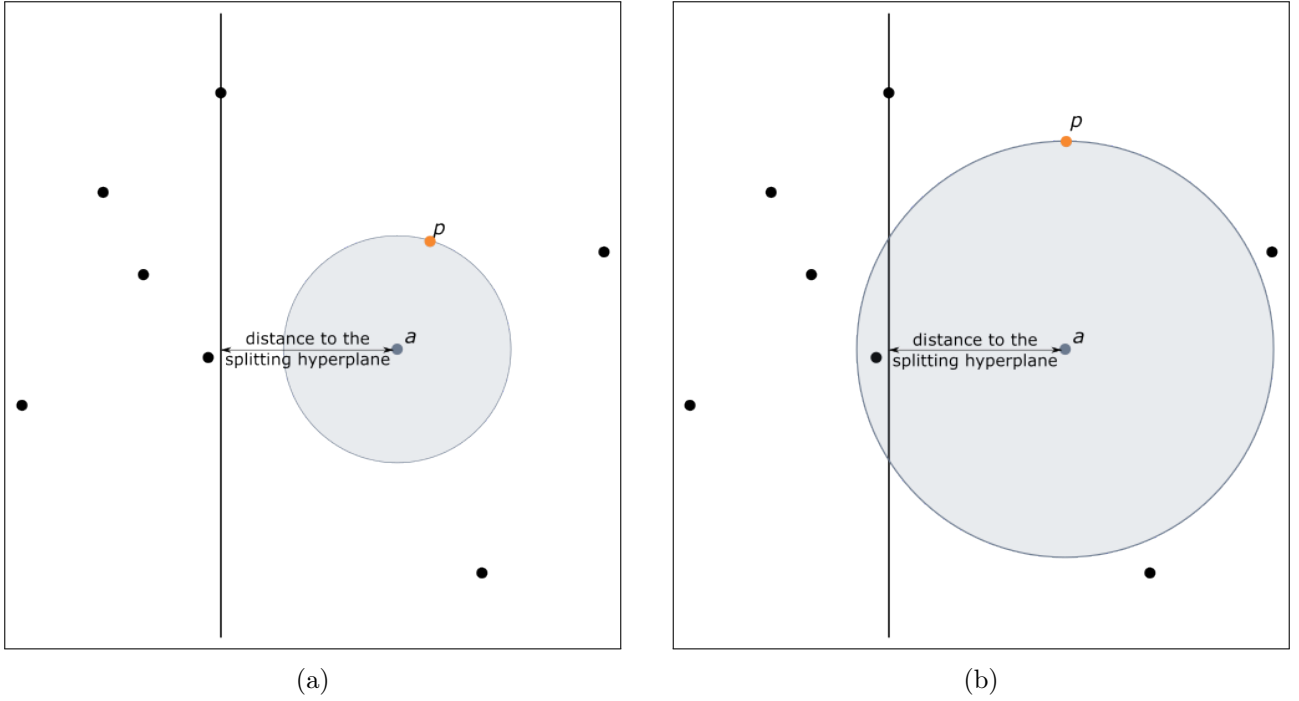


Figure 3: Two situations illustrating the necessity (or not) of searching for the nearest neighbour in the half-space other than the query point (adapted from diagrams by Pierre Lairez)

This approach is called defeatist because—in order to answer the query quickly—it does not try “hard enough” to find the true nearest neighbour, returning a candidate in the hope that it is “near enough”.

Question 6. Implement the recursive algorithm above in the procedure `NN_defeatist_search(self, query)` of the class `KDTree`.

Consider the two situations in Figure 3, where a is the query point and p is the current candidate for the nearest neighbour. If $\text{dist}(a, p)$ is smaller than the distance from a to the current splitting hyperplane (see Figure 3a), then indeed it is sufficient to search for the nearest neighbour in the same half-space as a . However, if $\text{dist}(a, p)$ is greater than the distance from a to the current splitting hyperplane (see Figure 3b), the actual nearest neighbour might turn out to be in the other half-space, which `NN_defeatist_search` does not visit.

Our last task is to integrate `NN_defeatist_search` and `NN_exhaustive_search` to get an algorithm which always finds the nearest neighbour, using the structure of the tree to avoid visiting every element. The idea is to use a backtracking approach starting at the candidate point found by `NN_defeatist_search`. For the ease of the exposition, we will present this algorithm slightly differently. We proceed recursively, assuming that at each step we have a candidate nearest neighbour `cand`.

1. Compare `query` to the point `p` associated to the current root of the tree: if $\text{dist}(\text{query}, p) < \text{dist}(\text{query}, \text{cand})$, the point `p` becomes the new candidate (update `cand`).
2. If the current node is a leaf, return the current `cand`.
3. Otherwise, let `c` be the coordinate associated to the root of the tree, and let `med` be the `c`-th coordinate of `p`.
 - (a) if the `c`-th coordinate of `cand` is less than or equal to `med`
 - i. Proceed recursively on the left sub-tree, updating `cand` as needed.
 - ii. If $\text{dist}(\text{query}, \text{cand}) > |\text{query}[c] - \text{med}|$, proceed also recursively on the right sub-tree.

(b) otherwise, proceed symmetrically starting with the right sub-tree.

Question 7. Implement the procedure `NN_backtracking_search_aux(self, query, cand = None)` in the class `KDTree`.

Optional: What are the best and worst case complexities of `backtracking_search` for queries that are not in the database?

As mentioned at the beginning of the TD, these trees are quite used in practice as they behave good for many databases, even though some bad example are also known. In what follows, we will consider two extreme examples.

Question 8. Use the procedures `testTimeRandom()` and `testTimeCircle()` to compare the performance of backtracking algorithm and the exhaustive search. These tests use random data to measure the performance of the algorithms. The first procedure, `testTimeRandom()`, constructs a database by picking uniformly random points in a 2-dimensional square. The second procedure `testTimeCircle()` constructs its database by picking uniformly random points in a 3-dimensional sphere. In both cases, the queries are points picked uniformly at random from a square/cube. Guess the asymptotical complexity of the backtracking approach in these examples.