École Polytechnique — CSE202 Design and Analysis of Algorithms

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 `testing.py` contains several test functions to help you with debugging. Notice the variable `longTests` at the bottom of that file. When it is set to `True`, a different set of tests is used. It *might* catch more errors but the tests will take longer time. One possibility is to debug the requested functions using `longTests = False` until they pass the shorter tests, then try with the longer ones.

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 following functions in `retrieval.py`: `dist(p, q)`, which computes the Euclidean distance between two points `p` and `q`, and `linear_scan(query, P)`, which computes the nearest neighbour of the query point `query` to points in the database `P`.

Question 1. The function `dist(p, q)` in `retrieval.py` computes the Euclidean distance between two points `p` and `q`. All points in the database are assumed to have the same dimension.

Question 2. Now that you have implemented `dist`, you can complete the function `linear_scan(query, P)`. This function takes as parameters: the query point `query` and the array 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 pair consisting of 1) the position of the point in the array `P` that is closest to the query point `query` and 2) the distance from that point to `query`. If there are multiple points in `P` at the same minimum distance to `query`, it returns the position of the first one.

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.

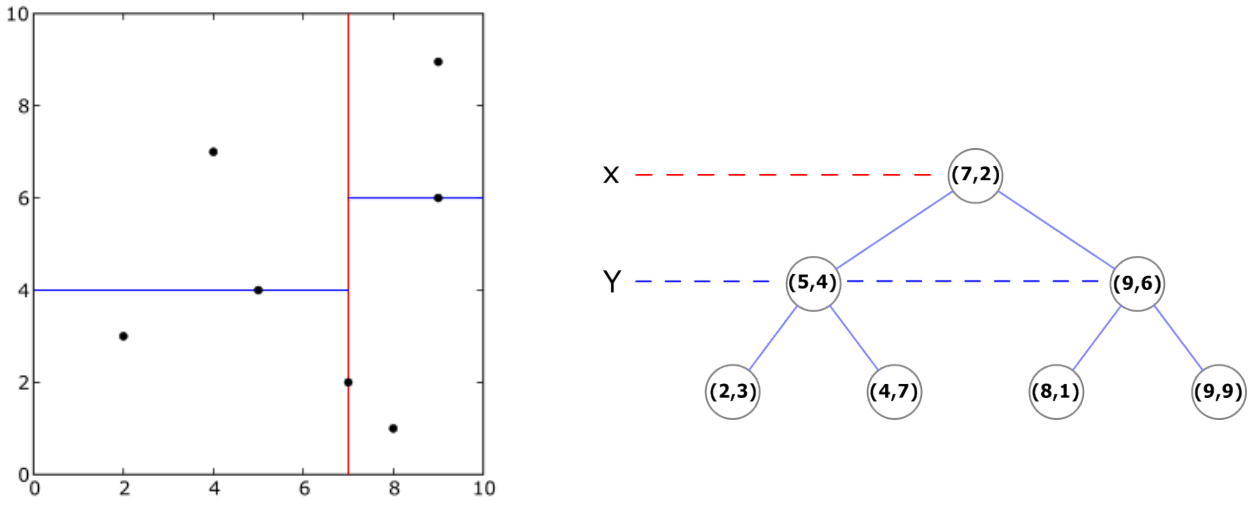
2 Building the k-d tree

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*. Points to the left of this hyperplane are represented by the left subtree of that node and points to the right of the hyperplane are represented by the right subtree.[1]

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] `https://en.wikipedia.org/wiki/K-d_tree`



(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)[2] and the resulting k-d tree (b)[3]

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 (ortogonal to the x axis) going through point $(7,2)$.
5. If no points are left in the cloud, stop.
6. Otherwise, use the coordinate $(c+1 \mod k)$ 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).

Question 3. Complete the function `compute_median(P, start, end, coord)`. In addition to the point cloud `P`, it takes as parameters: `start` and `end`—the indices of the range to process (`start` included, `end` excluded)—and the coordinate `coord` to use for splitting the points. The function returns the median of the collection of `coord`-th coordinates of the points in the sub-array.

NB: We are not asking you to use a DaC algorithm here—a simple sort will do.

Question 4. Complete the function `partition(P, start, end, coord)`, which arranges the points around the median. It has the same parameters as `compute_median` from Question 3 and returns the index idx of the point m in `P` whose `coord`-th coordinate is used as the median. The function rearranges the points in the array `P` in such a way that every point within the subrange [`start` : idx] has the `coord`-th coordinate less than or equal to that of m (with `P`[idx][`coord`] $= m$), while every point within the subrange [$idx + 1$: `end`] has a `coord`-th coordinate strictly greater than the median.

NB: Here, we do expect a linear complexity algorithm.

[2] Based on a diagram by KiwiSunset at the English-language Wikipedia, CC BY-SA 3.0, here
[3] Based on a diagram by MYguel – Own work, Public Domain, here

Question 5. Using the provided class `Node`, complete the function `build(P, start, end, coord)`, which builds the k-d tree for points in `P`. It has the same parameters as the functions in Questions 3 & 4 and returns the k-d tree constructed according to the algorithm above (`None` if the range is empty).

NB: To maintain the link between the tree nodes and cloud points, we store in each node the *index* of the corresponding point. For internal nodes of the tree, we will additionally store 1) the (index of the) coordinate used to partition the range during the construction of the node, and 2) the median value used to split the range along that coordinate. This information will be used for the subsequent Nearest Neighbour queries.

3 Nearest Neighbour search via k-dimensional trees

A first, so-called "defeatist" approach to finding the nearest neighbour of a query point q using a k-d tree consists in applying the same strategy as for the usual Binary Search Trees. We proceed recursively, assuming that at each step we have a candidate nearest neighbour $\hat{q}$.

1. Compare q to the point p indexed by the current node: if $dist(q, p) < dist(q, \hat{q})$, the point p becomes the new candidate (update $\hat{q}$).
2. If the current node is a leaf, return the current candidate.
3. Otherwise, let c and m be, respectively, the coordinate and the median stored in the current node:
   (a) if the c-th coordinate of q is less than or equal to m, proceed recursively on the left sub-tree,
   (b) otherwise, proceed recursively on the right sub-tree.

Question 6. Implement the recursive algorithm above in the function `defeatist_search_help(query, P, node, index, dmin)`. Parameters `index` and `dmin` are used to pass, respectively, the index of the current candidate point and the distance between `query` and that point. Further, implement the function `defeatist_search(query, P)`, which builds the k-d tree and initiates the recursive computation by `defeatist_search_help`.

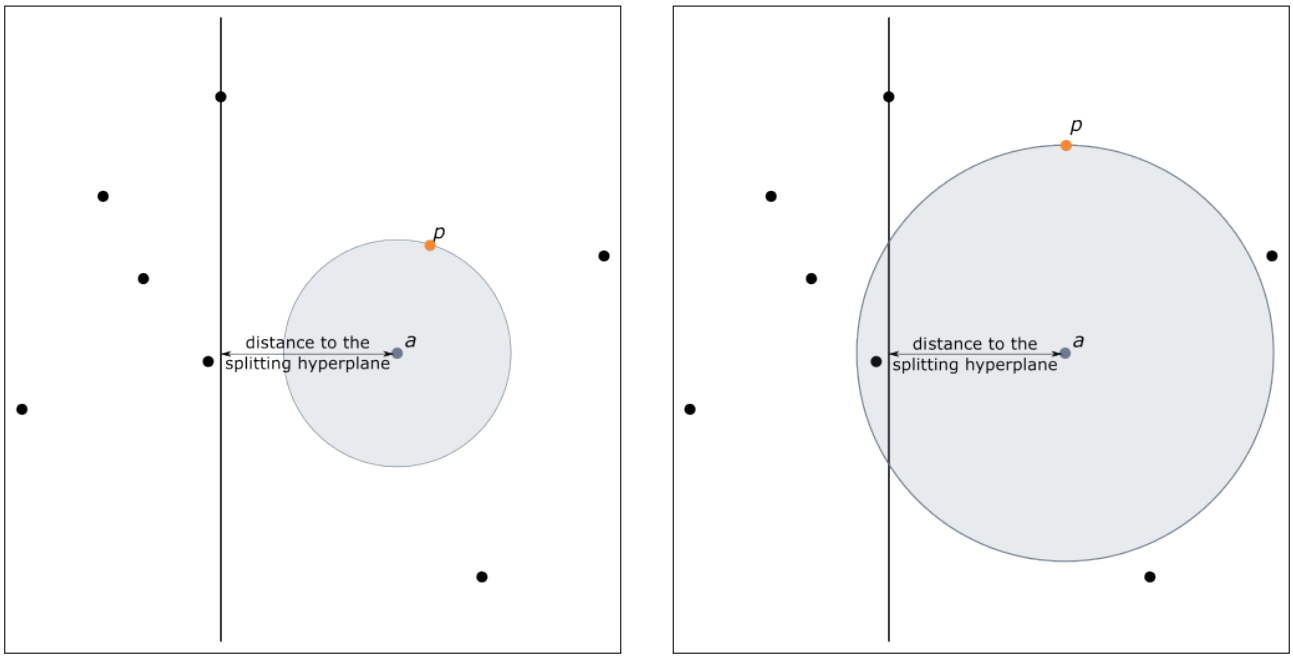
Optional: What are the best and worst case complexities of `defeatist_search` for queries that are not in the database?

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".

Consider the two situations in Figure 2, where a is the query point and p is the current candidate for the nearest neighbour. If $dist(a, p)$ is smaller than the distance from a to the current splitting hyperplane (see Figure 2a), then indeed it is sufficient to search for the nearest neighbour in the same half-space as a. However, if $dist(a, p)$ is greater than the distance from a to the current splitting hyperplane (see Figure 2b), the actual nearest neighbour might turn out to be in the other half-space, which `defeatist_search` does not visit.

Question 7. Implement the function `backtracking_search_help(query, P, node, index, dmin)` modifying the previous algorithm, so as to check the other half-space, when necessary. Further, implement the function `backtracking_search(query, P)`, which builds the k-d tree and initiates the recursive computation by `backtracking_search_help`.

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



(a) (b)

Figure 2: 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)