KD-Tree Coding Challenge

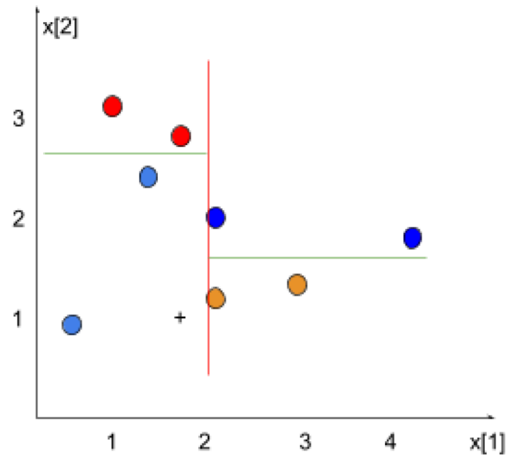
Background

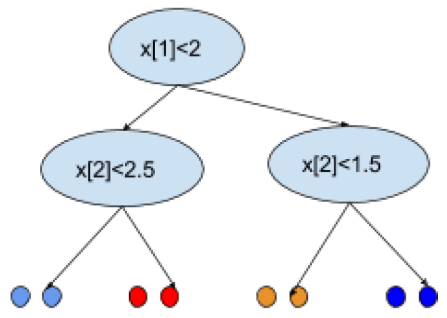
A KD-tree (see https://en.wikipedia.org/wiki/K-d_tree for more info) defines a recursive binary space partitioning, designed for performing efficient spatial queries such as nearest neighbor search. Given a set of K-dimensional points {x1, x2,...,xn} each internal node in the binary tree defines a partitioning of the space – an axis-aligned hyper plane – that "splits" the points associated with that node, such that points in the left sub tree are less than the splitting plane, and points in the right sub tree are greater than the splitting plane. This splitting procedure continues recursively, until some minimal number of points remain (possibly one), which are stored explicitly, by reference, or by index in the leaves of the tree.

A key consideration in producing a well- balanced tree is the choice of splitting axis and position at each node. One heuristic for selecting the splitting axis is to simply cycle through the axes in order. Another is to select the axis that has the greatest range or variance. Using either heuristic, the split position can then be chosen as the median (or median-of-medians) of the point values along that axis.

Given a query point, Y, finding the nearest neighbor in the tree requires a depth first search. At each node in the traversal, we test whether the value of the query Y in the node's splitting axis (e.g. Y[j]) is less or greater than the split position, and descend into the left or right sub tree respectively until a leaf node is reached.

A greedy search algorithm would terminate at this point, returning the X stored in the leaf node. This is suboptimal, however, since a closer point may in fact exist on the other side of the splitting hyper planes. Consider the query in the figure below, shown as the `+` point. Greedily descending through the tree would yield the light blue points, where in fact the orange point is closer.





Therefore, during traversal it is necessary to maintain a bound on how far the nearest neighbor could be from the query point. At internal nodes, if the bound is larger than the distance from the query point to the splitting point (indicating that points on the other side of the hyper plane may present viable candidates), it is necessary to search both sub trees. By keeping track of the best leaf node found to date, branches of the tree that are too far from the query point (i.e., the distance to the hyper plane is larger than the best found node to date) can be pruned/ignored entirely.

The Task

With these considerations in mind, your task is to develop a KD-tree library that provides at least the following interface/capabilities:

❑ Create a tree from a set of K-dimensional points. The tree should be created by recursively choosing a good axis (dimension) and position to split on. Choose an appropriate measure of "goodness" (e.g. largest range, variance, cycle) and justify your decision. Likewise for the split position (e.g. median, median-of-medians).

❑ Support efficient exact query for the nearest stored point to a query point. That is, search through the tree efficiently and return an iterator or reference to the nearest point.

❑ Support I/O to save/load the tree from disk.

❑ Your implementation should be templated on the scalar type of the point such that it supports at least \`float\` and \`double\` precision types.

Bonus Points:
❑ Allow the split axis and split position selection algorithms to be chosen at compile time or runtime.

Using your library, you should develop two applications:

❑ build_kdtree: An application to build the KD-tree from a sample dataset (a simple CSV file –details below). It should build the tree and save it to a user specifiable location.

❑ query_kdtree: An application to load a KD-tree generated by build_kdtree and a CSV file containing query points, and compute the nearest neighbors of each of the points. The application should output for each query point the exact nearest index (0based) in the sample dataset, and the corresponding Euclidean distance from the query point to the nearest neighbor.

File Formats

We provide a set of test points and query points attached. The file format is CSV with a comma delimiter:
X00, X01, X02, … X0N
X10, X11, X12, … X1N
…

Each line contains data for a single point. You can safely assume the file is formatted correctly and do not need to develop an extensive parsing system. The point ID is the 0based row number.

Your application should take in a set of query points and produce a file output that consists for each query point, the corresponding closest index of the sample data and the Euclidean distance from the query point to the nearest neighbor:

query0_closest_index, query0_minimum_distance
query1_closest_index, query1_minimum_distance
…