Nearest Neighbor using KDTree

Design of KDTree to represent spatial points and its subsequent use to query nearest neighbor amongst them to an external point .

Shashank Hegde

Table of contents

Overview

This document will cover the salient design decision/feature in the design of KDTree and its use to calculate nearest neighbor. It also notes other features that could have been added to the design, and what problems they could further solve, under which conditions.

Design Overview and structure

The KDTree library consists of 2 main classes, **KDTree** and **KDNode**. The KDNode contains the core data-structure used to represent each node in the KDTree. The KDTree provides support for creating it using array of spatial-points, and then use the structure to query the nearest neighbor to an external point. The main.cpp provides **build_kdTree** and **query_kdTree** that provides means to orchestrate the building of the KDTree by reading a csv file of spatial point and then using query_kdTree, which uses a csv file of spatial query points, to query the nearest neighbor of the query point from the KDTree.

The next sections would detail the design details of:

- KDNode
- KDTree

KDNode

The KDNode is expected to collectively represent 2 different data-points:

- A node which has details on what axis and value on that axis to split on so as to represent the median of the range of values on the remaining spatial point and hence also reference to the resultant left and right child.
- A node representing the actual spatial point itself. (in which case there is no remaining spatial point to further split on)

In lieu of that following struct was designed

```
int indexLabel;
int splitAxisIndex;  // hyperAxis/splittingPlane is perpendicular to the axis pointed by this index.
valType splitValue;
KDNode* left;
KDNode* right;
point<valType, dim>* nodePoint;    // set only when left and right are NULL.
```

OTHER NOTES

- std::array was used to represent spatial N dimensional points.
- findDistance finds Euclidean distance between 2 spatial K dimensional points.

KDTree

The KDTree maintains the reference to root of the KDTree and provides support to construct the trees using array of spatial points and then finding nearest neighbor amongst them for an external spatial point.
The key methods/features of KDTree and some design commentary are as below:

- **Constructing KDNode using vector of spatial points:** The central design is to divide the spatial point on the dimension that has max range of values and to follow this division until we reach individual spatial points. The theme is recursion and a helper *createKDNode* was constructed to support the constructor run this strategy.

- **Saving the KDTree to file for enabling reloading KDTree when required**: The problem here is to save KDNode that have all the leftNode and rightNode already saved to avoid repetitive update to the structure that will be saved to file. **PostOrder traversal** using *saveKDTreeToFileHelper* where the child nodes are the first one to be saved and then the parent aligns itself to the concept. This design also made **loading KDTree from file** seamless. The time complexity of this is O(n).

- **Nearest neighbor:** For this we first calculate the "best guess" by recursively searching to find the external point, using *doesPointExistHelper,* which also populates a stack of KDNode's it traversed to find the point. If we indeed found the point, then the topmost on the stack is the nearest(0 distance), otherwise our best guess is the leaf KDNode on top of the stack.

  We use euclidean distance calculated from the "best guess" to external point as radius, with the external point as centre to form hyperSphere. We do stack unwinding, to check if there exists point on other side of splitting-plane that could better our "guess nearest neighbor". We explore the other side, if the splitting plane(hyperAxis) intersect the hyperSphere, using *doesPointHelpe*r to help guide us to the better neighbor.
- The worst case time complexity of this approach is O(log(n)^2), where n is the number of points in the KDTree. This is because we get "best guess nearest neighbor" in log(n) and for each of the log(n) KDNode we may need to recurse to find if "better neighbor exists"

Additional features:

These are features that could have also been added, given more time.

- Parallelizing the construction and search : For a very large data-set, a significant effort will be needed to construct the KDTree. Parallelizing the effort could significantly reduce the time required for this. Also once the "best guess nearest neighbor" is found, we can also parallelize the effort to better the nearest neighbor. Again the complexity lies in thread-safe updates like most multithreaded application.

- Ability to select different strategy other than just max range media for splitting should be built. This is especially necessary for sparsely distributed data points. In case of sparsely distributed spatial points, splitting point that provide equal number of points on both side may be more favored, to create balanced tree.
- Storing at each KDNode that is closest with respect to splitting axis on either side would be good to have a more decisive and efficient way to explore when bettering the nearest neighbor. However this would also increase the space complexity.