INTERNATIONAL INSTITUTE OF INFORMATION TECHNOLOGY HYDERABAD

APS Course Project

Implementation of K-D tree and its applications

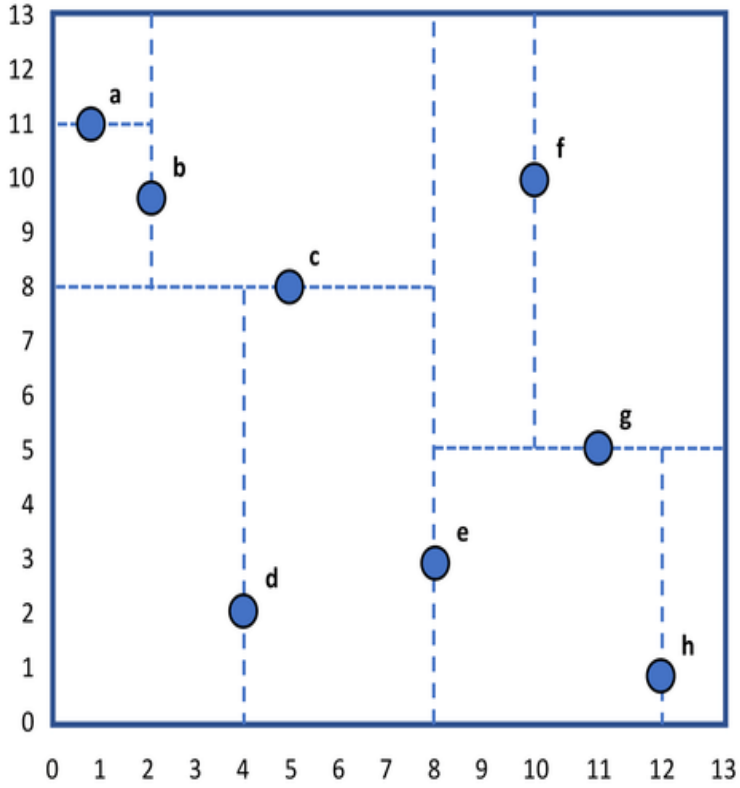
Shanu Shrivastava (2019202005)
Aditya Mohan Gupta (2019201047)

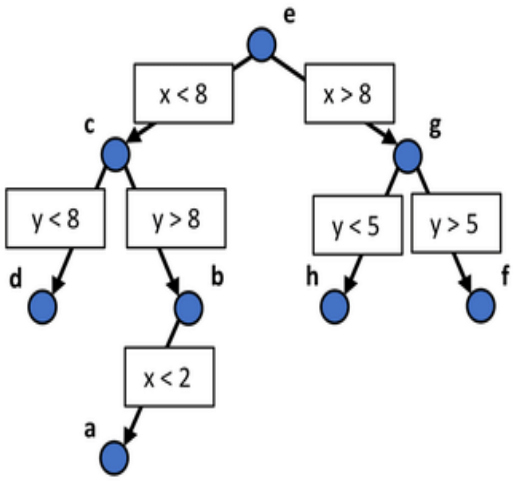
1. Introduction

1.1 K-Dimensional tree

K-D trees were invented in 1970s by Jon Bentley. The name originally meant “3d-trees, 4d-trees, etc” where k was the number of dimensions. A k-d tree, or k-dimensional tree, is a data structure used in computer science for organizing some number of points in a space with k dimensions. It is a binary search tree with other constraints imposed on it. Every node in the tree represents a point in the space. The k-d tree is a modification to the BST that allows for efficient processing of multi-dimensional search keys.

A sample K-D Tree with K=2 and space partitioning





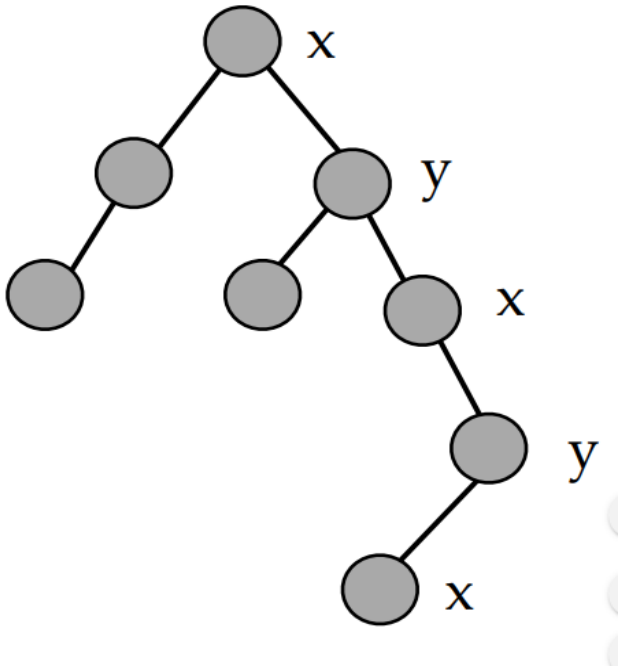
1.2 Design of a K-D tree

- A K-D tree consists of cutting dimensions along each level, based upon the number of dimensions i.e., value of k.
- Each node contains a point like point, P =(x, y, z…). To, traverse the tree; we need to compare the node value with the cutting dimensions, and accordingly make a move.

- Each level of a k-d tree splits all children along a specific dimension, using a plane that is perpendicular to the corresponding axis.

- At the root of the tree all children will be split based on the first dimension (i.e. If value of that dimension is less than the root's dimension, then child will be on the left side or else on the right side, when greater).

- The cutting dimension, along each level varies in the form of dimension % K.



- Real-world data is often represented in tabular form, which makes it well-suited to being stored in an Excel spreadsheet or a database table.

- In such a table with *k* columns, each column can be viewed as a dimension and each row represents a *k*-dimensional point.

- Efficient processing of multidimensional data is challenging.

- Binary Search Trees (when balanced) can store *n* elements effectively to guarantee O(logn) performance in searching. Similarly, a K-D tree stores n elements with k dimensions.

Structure of a K-Dimensional Node:

```
struct kdtree
{
        int *data;
        struct kdtree *left,*right;
        int dim;
        int cdim;
        int id;
};
```

The node structure defines the data part, which is a K (dim) sized array to store the data, left and right pointers respectively and the dimension of the node, and the cutting dimension of the node. The structure also contains an id field which gives a unique id to a node of the K-D tree.

2. Operations on K-D Tree

The following operations are performed:

2.1 Static K dimensional Data collection :

- Static data is collected for construction of a K-D tree. The k-d-tree is not designed for changes, and will quickly lose efficiency.
- It relies on the median, and thus any change to the tree would worst-case propagate through the entire tree.
- Thus, we need to collect all the Data beforehand. Hence, all the data which is to be classified, is collected and further processing is carried out to build the K-D tree.
- This collected data is then used for the construction of the K-D tree.

2.2 Insertion:

- Adding elements in a Binary Search tree fashion will lead to an unbalanced KD tree.
- Thus we need to perform balancing in the tree to ensure that the data points are correctly placed in the KD tree.

- For this, we need to find the median of the data and at each iteration and insert in the tree.

- This will give us a balanced tree for K-dimensional data.

- Every insert operation divides (partitions) the space.

Pseudo code for Insertion in K-D tree:

```
insert(Point x, KDNode t, int cd)
 {
     if t == null
              t = new KDNode(x)
     else if (x == t.data)
         // error! Duplicate
     else if (x[cd] < t.data[cd])
          t.left = insert(x, t.left, (cd+1) % DIM)
     else
          t.right = insert(x, t.right, (cd+1) % DIM)
     return t
}
```

Initially, we have training data i.e., a set of points. We sort the data and find the median according to the starting dimension. After finding the median, the points are divided into two parts, and now, we apply the same procedure of sorting the data on (start++ % K) dimension, on the individual parts, recursively to obtain the median points and hence insert the obtained points in the tree.

Time Complexity of K-D tree construction: $O(nlogn^2)$

2.3 FindMin:

- This is a pre - requisite for deleting elements in the K-D tree.

- In this operation, we find the minimum data point in a given dimension**.**

Pseudo code for FindMin operation:

```
Point findmin(Node T, int dim, int cd):
{
```

```
    if T == NULL: return NULL
        if cd == dim:
                if t.left == NULL: return t.data
                else return findmin(T.left, dim, (cd+1)%DIM)
        else: return minimum(
                findmin(T.left, dim, (cd+1)%DIM),
                findmin(T.right, dim, (cd+1)%DIM)
               T.data )
}
```

Time Complexity for FindMin Operation: O(logn)

2.4 Deletion :

- If node to be deleted is a leaf node in the K-D tree then we simply delete it.
- If the node to be deleted is not a leaf node then we need to find the replacement node for that particular node.
- A replacement node is the minimum node in the same dimension as that of the node to be deleted.
- Searching for replacement node is done by traversing the suitable part of the K-D tree, depending upon its structure.

Pseudo code for Deletion Operation:

```
    Point delete(Point x, Node T, int cd):
    {
            if T == NULL: error point not found!
            next_cd = (cd+1)%DIM
            if x = T.data:
                    if t.right != NULL:
                            t.data = findmin(T.right, cd, next_cd)
                           t.right = delete(t.data, t.right, next_cd)
                    else if T.left != NULL:
                            t.data = findmin(T.left, cd, next_cd)
                           t.right = delete(t.data, t.left, next_cd)
```

```
            else
                t = null
        else if x[cd] < t.data[cd]:
            t.left = delete(x, t.left, next_cd)
        else
            t.right = delete(x, t.right, next_cd)
        return t
}
```

Time Complexity for Deletion: O(logn)

3. Applications of K-D Tree

3.1 Nearest Neighbor:

- The Nearest Neighbor algorithm returns the nearest point of a given point in a given set of points in space.

Pseudo code:

```
Nearest(Node n,Point goal, Node best):
If n is null, return best
If n.distance(goal) <best.distance(goal),
        best=n
        If goal < n (based on cutting dimension)
            goodside = n.left
            badside=n.right
        Else
            goodside=n.right
            badside=n.left
best=nearest(goodside,goal,best)
//If bad side can have something useful
        best=nearest(badside,goal,best)
        return best
```

Time complexity to find the nearest neighbor in average case: θ(logn)
Time complexity to find the nearest neighbor in worst case O(n)

3.2 K- Nearest neighbor:

- It returns the k nearest points to a point in space.

- The biggest use case of K-NN search might be Recommender Systems. If you know a user likes a particular item, then you can recommend similar items for them. To find similar items, you compare the set of users who like each item—if a similar set of users like two different items, then the items themselves are probably similar!

  Pseudo code:

  Maintain a min heap(Q) that contains all the visited elements, while traversing the tree to find the nearest neighbours.

```
while K is not zero
        e= extract min(Q)
        print  e
        If e.left not visited
                Q.insert(e.left)
                For each children t in e.left which is not visited:
                        Q.insert(t)
        if e.right not visited
                Q.insert(e.right)
                For each children t in e.right which is not visited:
                        Q.insert(t)
        Decrease k
```

  Time Complexity: $O(n \log K)$

4. Code

The project is done in C++ language. Complete source code of this project can be downloaded from the following repository :
https://github.com/shanu-sh/aps_project_kdtree

5. References

- https://en.wikipedia.org/wiki/K-d_tree
- http://jcgt.org/published/0004/01/03/paper.pdf#page=17&zoom=100,0,660
- http://pointclouds.org/documentation/tutorials/kdtree_search.php
- https://www.cs.cmu.edu/~ckingsf/bioinfo-lectures/kdtrees.pdf