KNN Search Using K-d Trees

November 7, 2022

Dhananjay Goel (2021CSB1165) ,
Dhruv Singh Negi (2021CSB1167) ,
Virat Jain (2021CSB1220)

Instructor:
Dr. Anil Shukla

Teaching Assistant:
Anurag Jaiswal

Summary: In the project, we implemented the Search of K Nearest points from a point using d dimension trees. The K-D tree is a binary tree in which every node is a d dimension point. We did the full implementation of the K-D tree and used its function in our KNN algorithm to search for the k nearest points for a given point in the dataset

1. Introduction

K-D Tree is a data structure useful when organising data by several criteria. K-D Tree data structure is much like the regular binary search tree you are familiar with. Except each node would typically hold not just one value but a list of values. When traversing a K-D Tree, we cycle through the index to the values list based on the depth of a particular node. When inserting a new node, just like in the typical binary search tree, we compare the node's value and go left if it's smaller and right if it's bigger. It's just that, as we go deeper and deeper down the tree, we keep alternating between using coordinates values in the comparison.

d Dimensional data set on which the tree is created represents a partition of the d-dimensional space formed by the d-dimensional data set. Every node in the tree corresponds to a d-dimensional hyper-rectangle area.

`Space Partitioning Method:`

A K-d tree, which iteratively bisects the search space into two regions containing half of the points of the parent region. Queries are performed via traversal of the tree from the root to a leaf by evaluating the query point at each split. Depending on the distance specified in the query, neighbouring branches containing hits may also need to be evaluated. For constant dimension query time.

2. Time Complexity Analysis

To find the K nearest neighbours using brute force, the distance between every point in the data set is calculated using the Euclidean equation:
The Euclidean distance between a and b (both are K dimensional points) is defined as follows:

$$d(a,b) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2 + (z_1 - z_2)^2 + ... + (r_1 - r_2)^2}$$

`Brute Force Method:`

For every point, distance between it and all the other points can be calculated in O(n) time. The sorting can be done on the basis of the distances obtained in worst case O(n*n) time and for each point we have K dimensions so their swapping takes O(K) time.
So, for k points, the time complexity would be O(K*n^2)

The worst case when k=n will be of time complexity O(n^3).
Therefore, the time complexity to find k Nearest Neighbours in n points using brute force method is: O(k*n^2)

`Optimised Method:`

Insertion of an element in K-D tree is O(d*logn), so inserting n elements takes O(n*d*logn) time, where d is the dimension of points and n is the total number of points in the K-D tree.
Searching an element in the tree takes O(d*logn) time.

Nearest neighbour algorithm upon inspection had a time complexity of O(d*logn) in average and O(d*n) in worst case scenario.
The K-Nearest-Neighbour algorithm similarly was found to have an average time complexity of O(d*n*logn) and O(d*n*n) in worst case.

3. Figures and Algorithms

This following gives an idea about how the K-D Tree data structures stores data in a spatial arrangement in the K-dimensional space. (Here K=2)

3.1. Figures

The image given below gives us a clear visualisation of how K-d trees stores data efficiently for the searching k closest neighbours

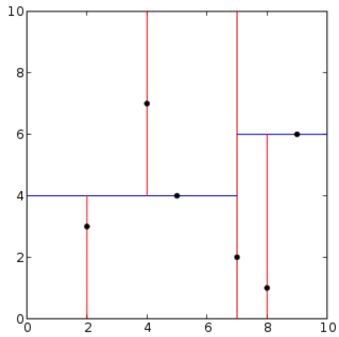

Figure 1: K-D tree decomposition for the point set (2,3), (5,4), (9,6), (4,7), (8,1), (7,2)

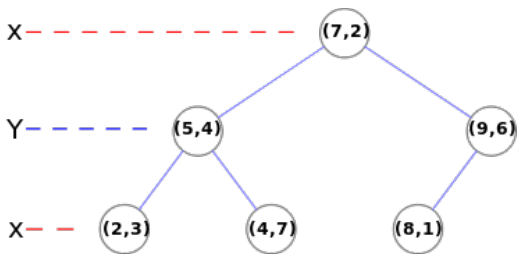

Figure 2: The resulting K-D Tree

3.2. Algorithms

Algorithm 1 InsertNode in K-D Tree (node* root, point[], depth, dimension)

```
if root = NULL then
    root = createNode
end if
CurrDepth= Depth%dimension
if point[CurrDepth] < root->point[CurrDepth] then
    root->left= InsertNode(root->left, point[],depth+1,dimension)
else
    root->right= InsertNode(root->right,point[],depth+1,dimension)
end if
return root
```

Algorithm 2 Search (root,point[],depth,dimension)

```
if root = NULL then
    return 0
end if
flag=1
for i=0 till dimension do
    if root->array[i] not equal point[i] then
        flag=0
        break
    end if
end for
if flag=1 then
    return 1
end if
CurrDepth= Depth%dimension
if root->array[CurrDepth] < point[CurrDepth] then
    return Search(root->right, point[],depth+1,dimension)
else
    return Search(root->left, point[],depth+1,dimension)
end if
```

Algorithm 3 NearestNeighbour (root,point[],depth,dimension)

```
bestBranch
otherBranch
if root = NULL then
    return NULL
end if
if point[depth %dimension]< root->array[depth] then
    bestBranch= root->left
    otherBranch= root->right
else
    bestBranch= root->right
    otherBranch= root->left
end if
temp = NearestNeighbour (bestBranch,point[],depth+1,dimension)
bestTillNow = Closest (temp,root,point[],dimension)
distanceSquared = Distance (point[],bestTillNow,dimension)
distanc = point[depth %dimension] - root->array[depth]
if distanceSquared >= distanc*distanc then
    temp = NearestNeighbour (otherBranch,point[],depth+1,dimension)
    bestTillNow = Closest (temp,bestTillNow,point[],dimension)
end if
return bestTillNow
```

Algorithm 4 enqueue (array[],dist,dimension)

```
size++
if size >0 then
    reallocate (size+1) to queue
end if
queue [size].arr = alloc (dimension)
for i=0 till dimension do
    queue [size].arr[i]= array[i]
end for
queue [size].priority= dist
```

Algorithm 5 Sort (K,dimension)

```
if K = 0 then
    return
end if
index= max(K, dimension
swap (queue[index], queue[K],dimension)
K–
Sort (K, dimension)
```

Algorithm 6 KNearestNeighbour (root,point[],depth,dimension,K)

```
bestBranch
otherBranch
if root = NULL then
    return
end if
if point[depth %dimension]< root->array[depth] then
    bestBranch= root->left
    otherBranch= root->right
else
    bestBranch= root->right
    otherBranch= root->left
end if
KNearestNeighbour (bestBranch,point[],depth+1,dimension,K)
distanc = Distance (point,root->array[],dimension)
if size < K-1 then
    enqueue (root->array[],distanc,dimension)
else
    index=max (K, dimension)
    maxDist= queue [index.priority]
    if distanc < maxDist then
        for i=0 till dimension do
            queue[index].arr[i]= root->array[i]
            queue[index].priority]= distanc
        end for
    end if
end if
if size < K-1 then
    KNearestNeighbour otherBranch,point[],depth+1,dimension,K)
else
    index=max (K, dimension)
    maxDist= queue [index.priority]
    dist = point[depth %dimension] - root->array[depth]
    if maxDist >= dist*dist then
        KNearestNeighbour (otherBranch,point[],depth+1,dimension,K)
    end if
end if
```

4. Applications of K-D Tree

Various application of K-D Tree and K-Nearest-Neighbour are:

1. Efficient storage of spatial data;
2. Range queries
3. Used extensively in 3D computer graphics, especially game design
4. Solve a near neighbour for cross identification of huge catalog and realise the classification of astronomical objects
5. Used for Classification in various Machine Learning Models

5. Conclusions

In this project,we implemented basic K-D Tree and it's basic functions. We optimised the K nearest neighbour algorithm using K-D Trees. The proposed algorithm showed an improvement in time complexity in searching K- Nearest points. We also implemented a custom-made priority queue consisting array as it's element to store

K nearest points.

6. Bibliography and citations

Here is a list of all the sources consulted while developing the work:

https://en.wikipedia.org/wiki/K-d_tree
https://en.wikipedia.org/wiki/K-nearest_neighbors_algorithm
https://opendsa-server.cs.vt.edu/ODSA/Books/CS3/html/KDtree.html
https://en.wikipedia.org/wiki/Priority_queue
https://towardsdatascience.com/space-partitioning-and-kd-trees-7b0e12b368d0

Acknowledgements

I would like to express my sincere gratitude to Prof. Anil Shukla for giving me the opportunity to work on a subject that interests me. This opportunity allowed me to conduct extensive study and I was able to learn a great deal of fresh information. Additionally, I want to thank Mr. Anurag Jaiswal for helping me with any issues I encountered and assisting me at every stage of the project. Without their amazing direction and support, I would not have been able to finish my project.
[2] [1]

References

[1] Aymane Hachcham. The knn algorithm- explanation, opportunities, limitations. *neptune.ai*, 10:1–10, 2020.

[2] Marcello la Rocca. *Advanced Algorithms and Data Structures*. Manning Publications Co., 2021.