Implementation of KD Trees and Nearest Neighbor Search

November 8, 2024

Atharv Srivastava (2022MEB1301) ,
Arjun Rana (2022EEB1370) ,
Atharva Sharma (2021MEB1302)

Instructor:
Dr. Swapnil Dhamal

Teaching Assistant:
Miss Shradha Sharma

Summary: K-D tree (short for K-dimensional tree) is a space-partitioning data structure for organizing points in a k-dimensional space. We will be partitioning a set of datapoints using K-D tree and search for the nearest neighbours for a given point.

1. Introduction

A K-dimensional tree, often abbreviated as a "K-D tree" is a data structure used for organizing and searching points in a multi-dimensional space. It is a specialized binary tree designed to efficiently partition and index data in k-dimensional space. In a K-D tree, a non-leaf node divides the space into two half-spaces. The left subtree of this node contains points to the left of this partition, and the right subtree contains points to the right of this partition. It has various useful applications in computer graphics, computer vision, data mining, and computational geometry.

2. Problem

In recent years, growth of digital storage and data systems has made easier to collect and store vast amount of information. However, shifting this large amount of data to find specific, relevant information has become challenging, particularly in large-scale databases. To help users in accessing these databases and extract the relevant information, Information Retrieval (IR) is used. Information Retrieval is a process that organizes, stores and searches data in response to user queries or requests, making information more accessible.
One common choice for IR is K nearest neighbor search. KNN is one of the simplest algorithms that calculates the distance between the query observation and each data point in the training dataset and finds the K closest observations. Over the years through great research, huge improvements have been done in the KNN algorithm, leading to the development of K-D trees.
In this project, we are going to implement a K-D tree data structure and we want to use it to perform the nearest neighbor search in O(log N) time complexity.

3. Objectives

•To implement the K-D tree data structure
•To perform nearest neighbour search

4. Data Structure Details

Functions we are implementing:

4.1. Insert

This function will help to insert coordinates into the K-D Tree data structure.
Let's discuss how the insert function works.

Let's take K=2, i.e. it is a 2-dimensional K-D Tree or simply a 2D Tree. The root would have an x-aligned plane, the root's children would both have y-aligned planes, the root's grandchildren would all have x-aligned planes, and the root's great-grandchildren would all have y-aligned planes, and so on.
In this way, we are partitioning the space into two planes. If the root node is aligned in plane A, then the left sub-tree will contain all points whose coordinates in that plane are smaller than that of the root node. Similarly, the right sub-tree will contain all points whose coordinates in that plane are greater or equal to that of the root node.
Consider an example of insert in the 2-D plane:
(3, 6), (17, 15), (13, 15), (6, 12), (9, 1), (2, 7), (10, 19)
1. Insert (3, 6): Since the tree is empty, make it the root node.

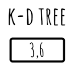


Figure 1: Root Node with (3,6) inserted

2. Insert (17, 15): Compare it with root node point. Since root node is X-aligned, the X-coordinate value will be compared to determine if it lies in the right subtree or in the left subtree. This point will be Y-aligned.

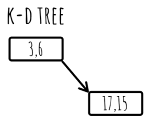


Figure 2: Inserted (17,15) into the K-D tree

3.. Insert (13, 15): The x-value of this point is greater than the X-value of the point in the root node. So, this will lie in the right subtree of (3, 6). Again Compare the Y-value of this point with the Y-value of point (17, 15). Since they are equal, this point will lie in the right subtree of (17, 15). This point will be X-aligned.

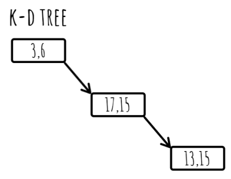


Figure 3: Inserted (3,15) into the K-D tree

4. Insert (6, 12): The x-value of this point is greater than the X-value of the point in the root node. So, this will lie in the right subtree of (3, 6). Again Compare the Y-value of this point with the Y-value of point (17, 15) (Why?). Since, 12 $<$ 15, this point will lie in the left subtree of (17, 15). This point will be X-aligned.

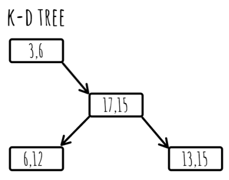


Figure 4: Inserted (6,12) into the K-D tree

5. Similarly rest points will be inserted by alternatively comparing the X and Y coordinates of the child with that of the parent.

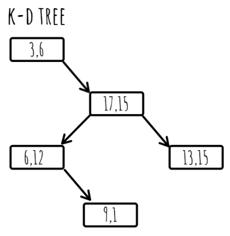


Figure 5: Inserted (9,1) into the K-D tree

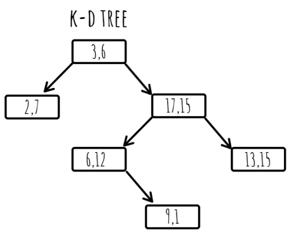


Figure 6: Inserted (2,7) into the K-D tree

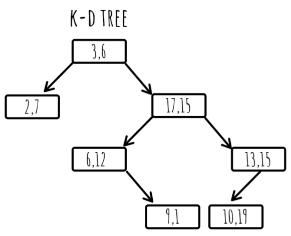


Figure 7: Inserted (10,19) into the K-D tree

4.1.1 Pseudo-Code

Algorithm 1 Insertion

```
node* Insert(node* x,int point[ ],int depth)
if $x$ is $NULL$ then
    temp=Allocate memory()
    for $i = 0$ upto $k - 1$ do
        $temp.point_i = point_i$
    end for
    x=temp
    return temp
end if
if $point_{depth\%k} < x.point_{depth\%k}$ then
    return Insert(x.left,point,depth)
else
    return Insert(x.right,point,depth)
end if
return x
```

It has an average of O(log n) time complexity and O(n) space complexity.

4.2. Search

Search in a K-D tree works in somewhat the same as that in a binary tree, however, the difference is that every time we go to a new level, we change our pivot(called discriminator in this case). Basically, we change the element with which we are comparing our element, to the next consecutive coordinate present in the node. All nodes have the same discriminator at any level. So the root node will have discriminator 0 and it's two child nodes will have discriminator 1 and so on.
Searching is very efficient as we are partitioning the space with each comparison. The actual space partitioning is described below.
1.Point (3, 6) will divide the space into two parts. So we draw a line x=3.

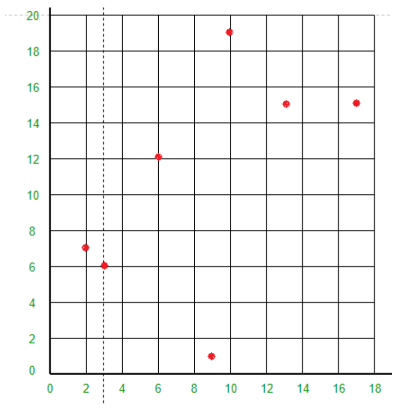


Figure 8: Drawn line x=3

2.Point (2, 7) will divide the space to the left of line X = 3 into two parts horizontally. So we draw a line y=7 to the left of line x=3.

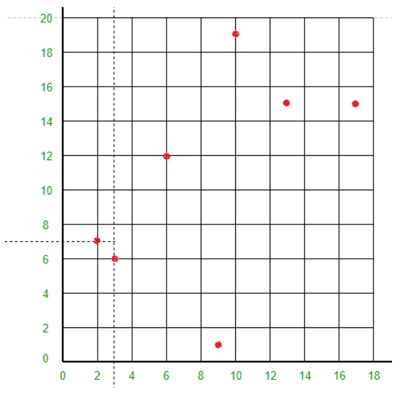


Figure 9: Drawn line y=7 to the left of x=3

3.Point (17, 15) will divide the space to the right of line X = 3 into two parts horizontally. So we draw a line y=15 to the right of line x=3.

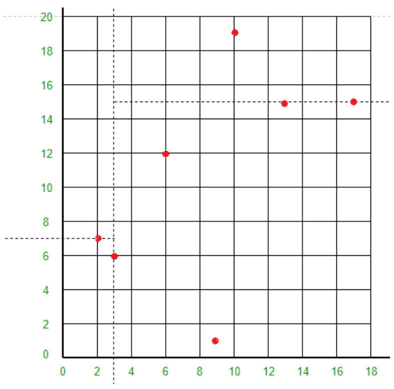


Figure 10: Drawn line y=15 to the right of x=3

4.Point (6, 12) will divide the space below line Y = 15 and to the right of line X = 3 into two parts. So we draw a line x=6 to the right of line x=3 and below line y=15.

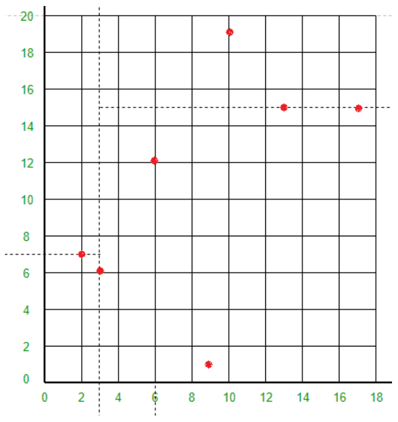


Figure 11: Drawn line x=6 to the right of x=3 and below y=15

5.Point (13, 15) will divide the space below line Y = 15 and to the right of line X = 6 into two parts. So we draw a line x=13 to the right of line x=6 and below line y=15.

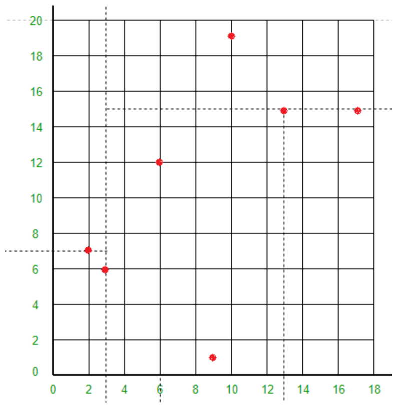


Figure 12: Drawn line x=13 to the right of line x=6 and below line y=15

6.Point (9, 1) will divide the space between lines X = 3, X = 6 and Y = 15 into two parts. Hence, we draw a line y=1 between lines x=3 and x=13.

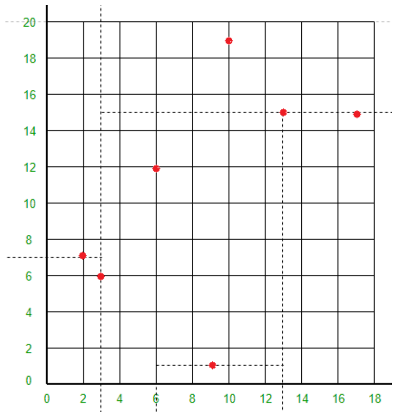


Figure 13: Drawn line y=1 between lines x=3 and x=13

7.Point (10, 19) will divide the space to the right of line X = 3 and above line Y = 15 into two parts. So we draw a line y=19 to the right of line x=3 and above line y=15.

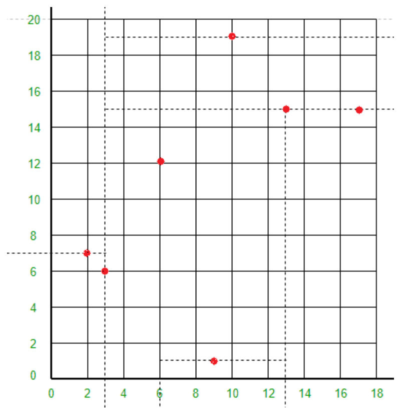


Figure 14: Drawn line y=19 to the right of line x=3 above line y=15

As we have seen, as more and more points are getting entered into our K-D Tree, we are doing more and more partitions of the space.

4.2.1 Pseudocode

Algorithm 2 Search

```
1: node* Search(node* x,int point[ ],int depth)
2: if $x$ is $NULL$ then
3:     return NULL
4: end if
5: if $point_{depth\%k}=x.point_{depth\%k}$ then
6:     if $point=x.point$ then
7:         return x
8:     else if $point_{(depth+1)\%k}<x.point_{(depth+1)\%k}$ then
9:         return Search(x.left,point,depth+1)
10:    else
11:        return Search(x.right,point,depth+1
12:    end if
13: else if $point_{depth\%k}<x.point_{depth\%k}$ then
14:    return Search(x.left,point,depth)
15: else
16:    return Search(x.right,point,depth)
17: end if
```

It has an average of O(log n) time complexity.

4.3. Traversal

The K-D tree can be traversed in the same way as a typical Binary Tree. In this, we employ inorder traversal. In order to perform an inorder traversal, we must first visit all the nodes in the left subtree while simultaneously printing all the points that are stored in each node, followed by visiting the root node and printing all the points that are stored there, and then visiting all the nodes in the right subtree.

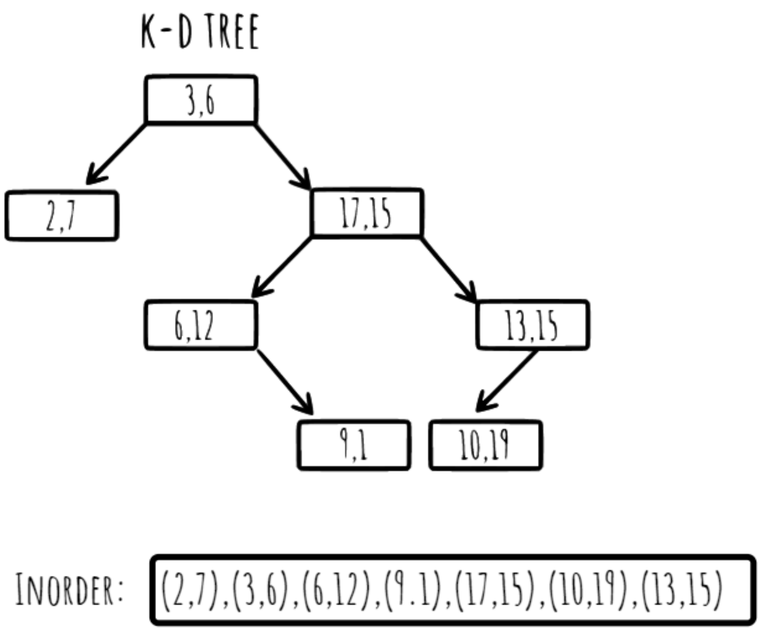


Figure 15: Inorder Traversal

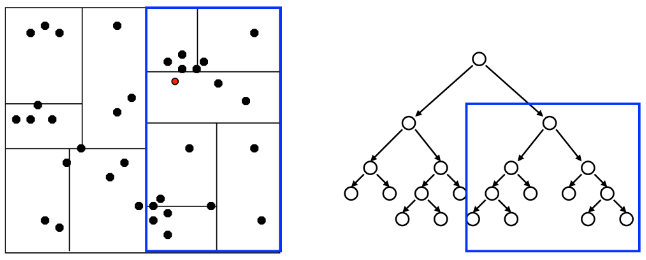
4.3.1 Pseudocode

Algorithm 3 Inorder Traversal

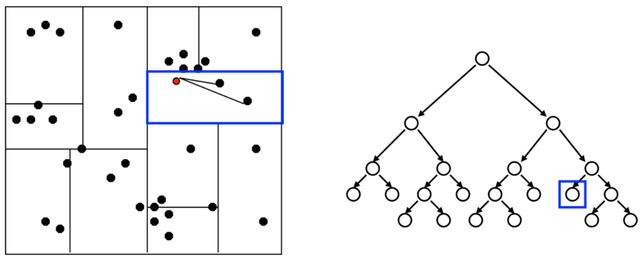
```
1: void Inorder(node* x)
2: if $x$ is $NULL$ then
3:     return NULL
4: end if
5: Inorder(x.left)
6: print x.point
7: Inorder(x.right)
```

4.4. Nearest Neighbour

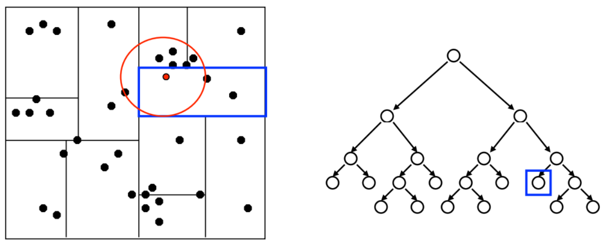
In this section, we have described the nearest neighbor algorithm.
1.We traverse the tree looking for the nearest neighbor of the query point.



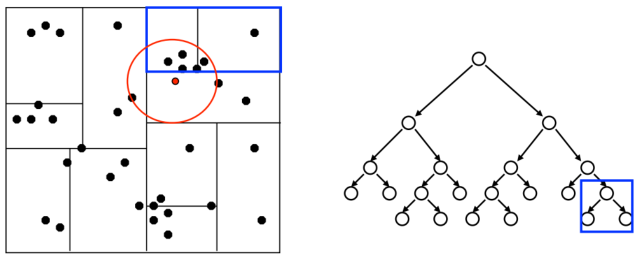
2.Explore the branch of the tree that is closest to the query point first.



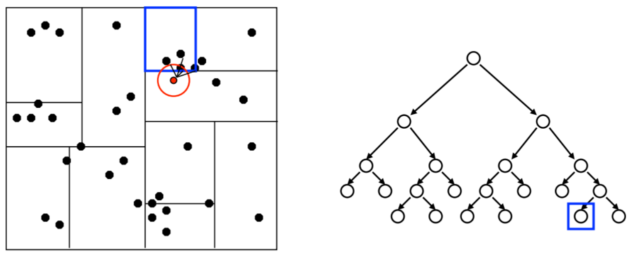
3.When we reach a leaf node, compute the distance to each point in the node.



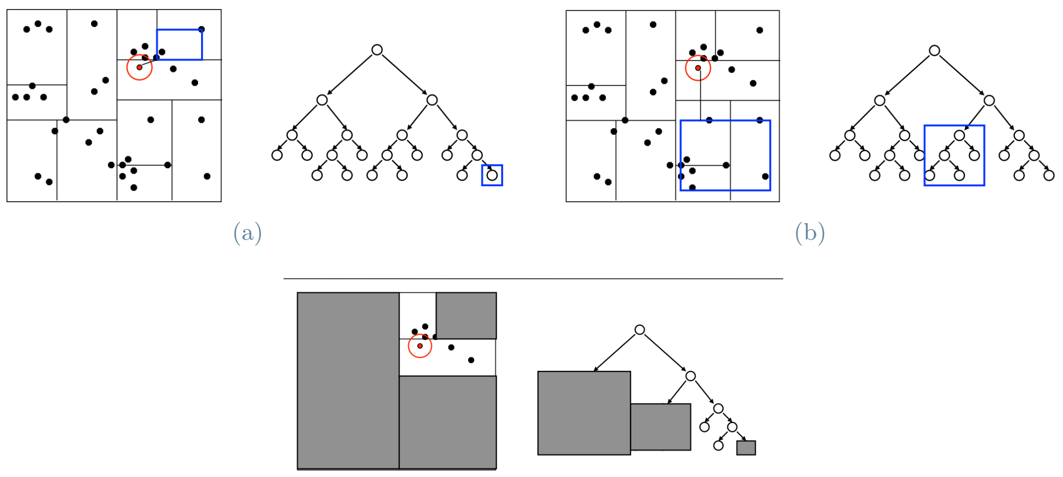
4.Then we can backtrack and try the other branch at each node visited.



5.Each time a new closest node is found, we can update the distance bounds.



6.Using the distance bounds and the bounds of the data below each node, we can prune parts of the tree that could NOT include the nearest neighbour.



(a) (b) (c)

Figure 16: Pruning the unfavourable regions

From figure 15, it is quite clear that, as we keep on traversing to and fro the K-D tree, we are reducing the number of spacial points available in the database.
As a result we can perform the nearest neighbour searching in an efficient manner.

5. Time Complexity Analysis

5.1. Insertion

• Insertion takes place similar to a Binary Tree. After comparison with the root node, insertion takes place in either the left subtree or the right subtree of the K dimensional tree. In average case, the insertion takes place O(h) time where h is the height of the tree.
So, average time complexity of insertion is O(log n).

5.2. Search

Searching for a point in a K dimensional tree involves traversing the tree in a manner similar to insertion.If the point we are looking for exists in the tree, we will eventually find it.
The time complexity for search is also O(log n) on average.

5.3. Traversal

The time complexity of inorder traversal of a k-d tree is O(n), where ’n’ is the number of points in the tree. This is because, in inorder traversal, all nodes of the tree are visited exactly once.

5.4. Nearest Neighbor

• When we have a query point, we traverse all the way down to a leaf node. So we’re going to go all the way down to the depth of the tree. In the worst case, we would have to traverse N nodes, the worst case being when, because of the structure of the problem we’re not able to prune anything at all.
• The complexity range is anywhere from on the order of log N, if everything gets pruned, all the way to on the order of N, if we can’t do any pruning.
One Nearest Neighbor query results in:
1. Traversing down the tree from the starting point takes O(logN)
2. Maximum backtrack and traversal takes O(N) in worst case.
Hence the complexity ranges from O(logN) to O(N).

6. Application

The major application of KNN using K-D Trees is finding nearest neighbour in a suitably large database in a very short time. This sole application is utilised in various fields of data science and to solve some real world problems like:
1. Classification
KNN is often used for classification tasks, where it predicts the label of an unseen data point based on the labels of its k-nearest neighbors. This application is widely used in fields such as image recognition, text classification, and spam detection.
2. Recommendation Systems:
KNN-based collaborative filtering is used in recommendation systems to provide personalized recommendations for users. It identifies users who have similar preferences to recommend products, movies, or content that a given user might like based on the preferences of their neighbors.
3. Spatial Data Analysis:
KNN is frequently used in geographic information systems (GIS) and spatial data analysis to find, for example, the nearest points of interest, the closest healthcare facility, or the most similar regions based on various features.
4. Database queries involving a multidimensional search key:
Suppose we have a query asking for all the employees in the age-group of (40, 50) and earning a salary in the range of (15000, 20000) A geometrical problem can be created by plotting the age along the x-axis and the income along the y-axis in order to find all of the employees who are between the ages of (40) and (50) and make between (15000 and 20000) each month.

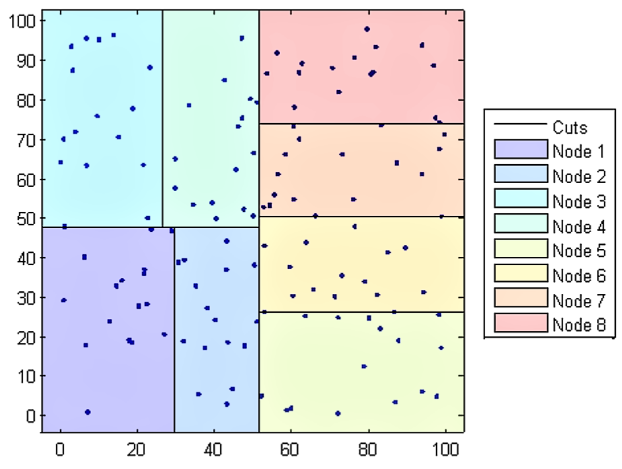


Figure 17: Multidimensional search key

A 2-dimensional k-d tree on the composite index of (age, salary) could help us efficiently search for all the employees that fall in the rectangular region of space defined by the query described above.

7. Conclusions

In a nutshell, we can say that K-D trees are an efficient way of searching for the nearest neighbor when there are millions of data points available. However, as the input size grows, it takes a huge time to search for the nearest neighbours. When we have a query point, we will descend all the way to the bottom of the tree. If the problem's structure prevented us from performing any pruning at all, the worst-case scenario would require us to explore N nodes, but this is generally not the case. Usually, when we traverse the K-D Tree, to search for a neighbour, we prune almost half of the nodes, and finally, on average maintain a time complexity of O(h) which is equal to O(logn).

8. Bibliography and citations

ALGLIB
CMSC Lecture Notes
K-D Tree Wikipedia

[1–3]

References

[1] Giuseppe Bonaccorso. *Mastering Machine Learning Algorithms*. May,2018.

[2] Peter Brass. *Advanced Data Structures*. 2008.

[3] Thomas H. Cormen. *Introduction to Algorithms*. The MIT Press, New York, NY, USA, 1990.