Nearest Neighbour Search By K-Dimension Trees

November 3, 2023

Priyanshi Singh(2022MCB1276) ,
Sneha Sahu(2022CSB1129) ,
Hartik Arora(2022CSB1314)

Instructor:
Dr. Anil Shukla

Teaching Assistant:
Manpreet Kaur

Summary: A space-partitioning data structure called a "K-D tree" is used to arrange points in a k-dimensional space. We wish to divide the area into sections and look for the closest neighbours in one of the advantageous within the space.

1. Introduction

A K-D Tree, or K-Dimensional Tree, is a data structure for organizing points in a K-dimensional space. It is a binary search tree, where each node represents a K-dimensional point. Every non-leaf node in the tree acts as a hyperplane, dividing the space into two partitions. This hyperplane is perpendicular to the chosen axis, which is associated with one of the K dimensions. A normal Binary Search Tree is also a K-D Tree with K=1. Thus we can say, that K-D Trees are a direct extension of the binary search tree.

2. Problems

In recent decades, the availability of cheap and effective storage devices and information systems has led to a rapid increase in the size and complexity of graphical and textual databases. This has made it easier to collect and store information, but also more difficult to retrieve relevant information, especially from large-scale databases.
Information retrieval (IR) is the process of representing, storing, and searching collections of data to extract relevant knowledge or documents in response to a user query. There are a number of algorithms and data structures used for IR, one of the most common being K-nearest neighbors (KNN).
KNN is a simple algorithm that calculates the distance between a query observation and each data point in a training dataset and then finds the K closest observations. Over the years through great research, huge improvements have been made in the KNN algorithm, leading to the development of K-D trees. In this project, we are going to implement K-D trees to find the nearest neighbour.
K-D trees can be used to perform nearest neighbor search in O(log N) time complexity, where N is the number of points in the dataset. This is a significant improvement over the O(N) time complexity of the brute-force approach to nearest neighbor search.

3. Objective

•To implement the K-D tree data structure.
•To perform nearest neighbour search in O(logn) time.
•To demonstrate this nearest neighbor search's practical application.

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-D tree data structure. Since it is a 2-D tree every node has two components (x,y). Initially, 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.
Consider an Example of insert in the 2-D plane:
(20, 28), (10, 25), (15, 20), (16, 23), (5, 60), (30, 40), (40, 50), (25, 50), (45, 55).

1. Insert (20, 28): Since the tree is empty, make it the root node.

Figure 1: Root node with (20, 28) inserted

2. Insert (10, 25): Compare it with the root node point. Since the 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.

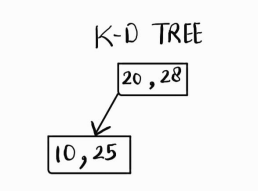


Figure 2: Inserted (10,25) in K-D tree

3. Insert (15, 20): The x-value of this point is less than the X-value of the point in the root node. So, this will lie in the left subtree of (20, 28). Again Compare the Y-value of this point with the Y-value of point (10, 25). Since the Y-value of this point is less than (10, 25). So this point lies in the left subtree of (10, 25). This point will be X-aligned.

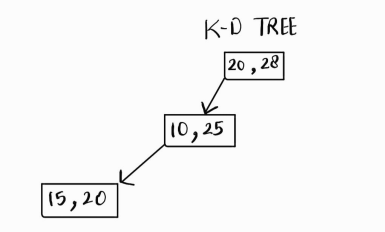


Figure 3: Inserted (15, 20) in K-D tree

4. Insert (16, 23): The x-value of this point is less than the X-value of the point in the root node. So, this will lie in the left subtree of (20, 28). Again Compare the Y-value of this point with the Y-value of point (10,

25) (Why?). Since, 20 < 25, this point will lie in the left subtree of (10, 25). This point will be X-aligned. So compare the X-value of this point with the X-value of point (15, 20). Since, 16 > 15, this point will lie in the right subtree of (15, 20.)

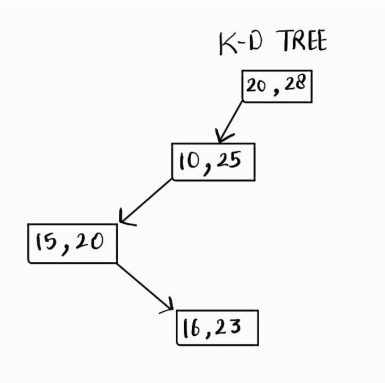


Figure 4: Inserted (16,23) in K-D tree

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

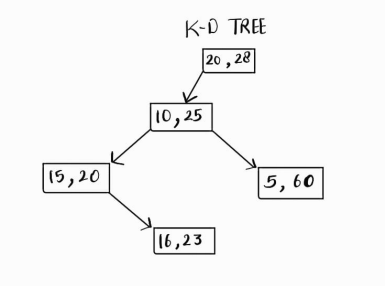


Figure 5: Inserted (5, 60) in K-D tree

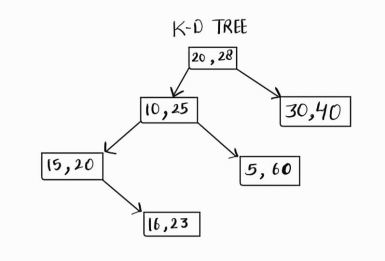


Figure 6: Inserted (30, 40) in K-D tree

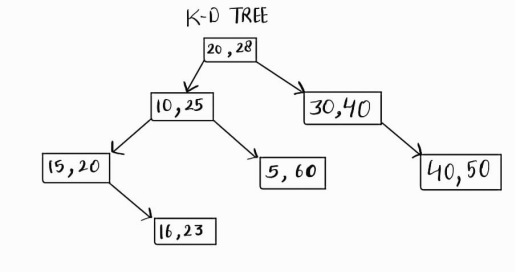


Figure 7: Inserted (40, 50) in K-D tree

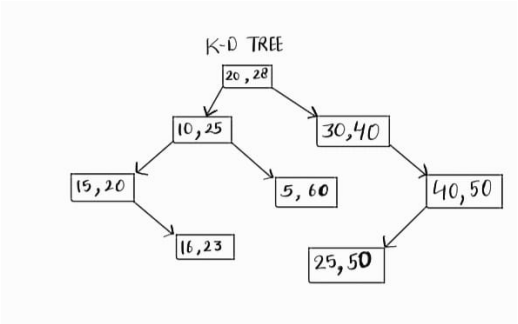


Figure 8: Inserted (25, 50) in K-D tree

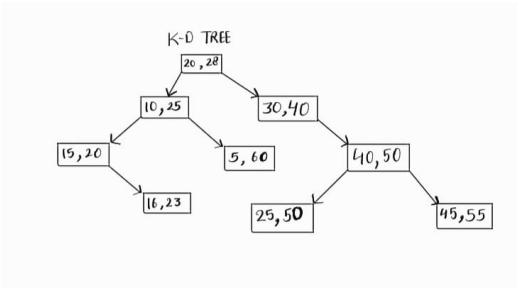


Figure 9: Inserted (45, 55) in K-D tree

4.1.1 Algorithms

Pseudo-algorithms of insertion in K-D trees:

Algorithm 1 Insertion in K-D trees

```
node* Insert(node* x , int* point, int depth)
if x is NULL then
  Temp = AllocateMemory()
  for i = 0 to k − 1 do
Temp.point(i) = point(i)
  end for
  x = Temp
return Temp
end if
if point(depth mod k) > x.point(depth mod k) then
  Insert at right
else
  Insert at left
return x
```

Building a K-D Tree has an average O(logn) time complexity and O(n) space complexity.

4.2. Search

Search in a K-D tree works in somewhat the same fashion 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 (20, 28) will divide the space into two parts. So we draw a line x=20.

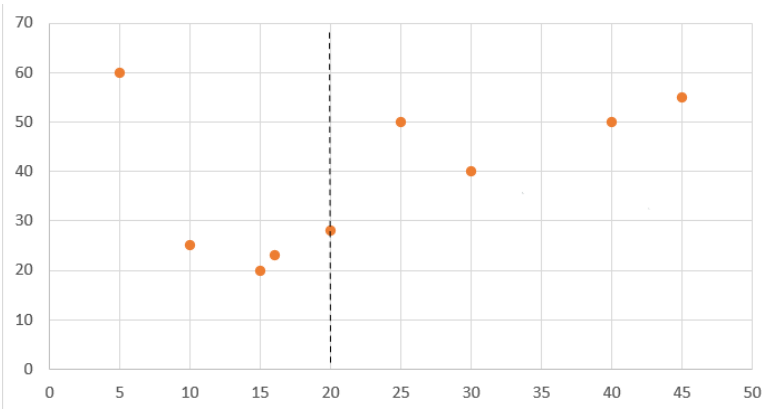


Figure 10: Draw line x = 20

2. Point (10, 25) will divide the space to the left of line X = 20 into two parts horizontally. So we draw a line y = 25 to the left of line x = 20.

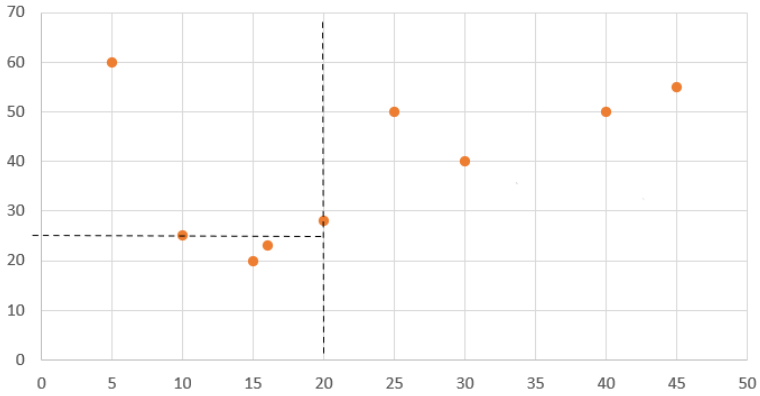


Figure 11: Draw line y = 25 to the left of x = 20

3. Point (15, 20) will divide the space to the left of line X = 20 and below the line y = 25 into two parts horizontally. So we draw a line x = 15 to the left of line x = 20.

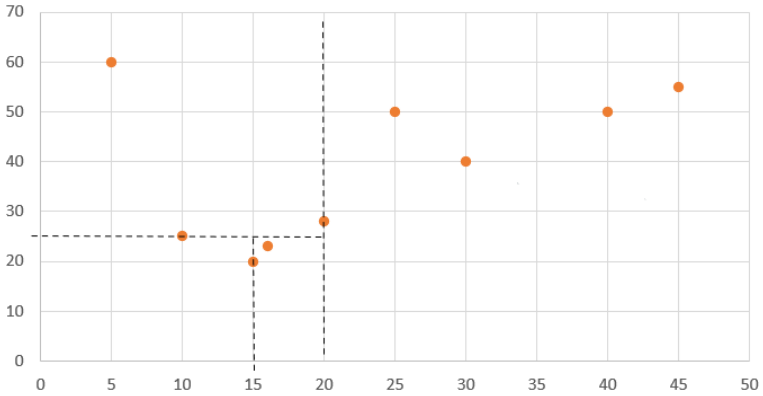


Figure 12: Draw line x = 15 below the line y = 25

4. Point (16, 23) will divide the space below line Y = 25 and to the right of line X = 15 into two parts. So we draw a line y = 23 to the right of line x = 15 and below line y = 25.

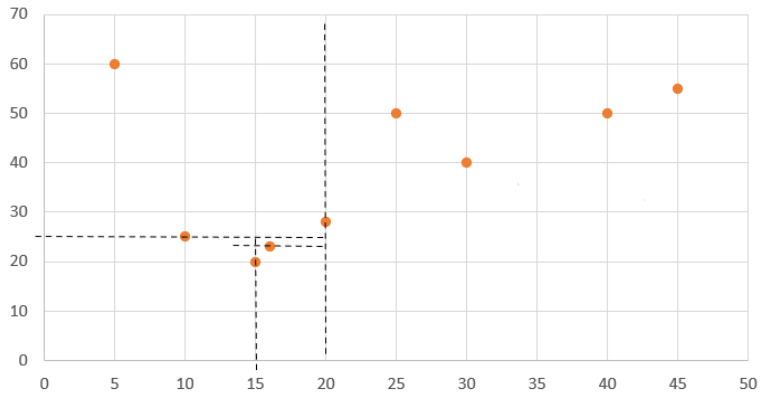


Figure 13: Draw line y = 23 to the right of x = 15

5. Point (5, 60) will divide the space above line Y = 25 and to the left of line X = 20 into two parts. So we draw a line x = 5 to the left of line x = 20 and above line y = 25.

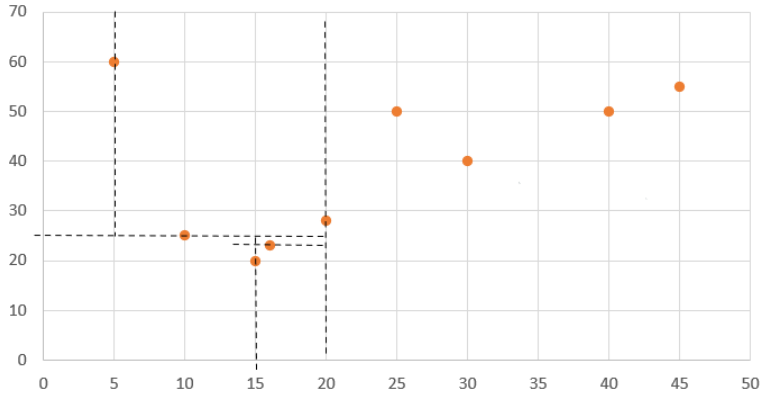


Figure 14: Draw line x = 5 to the left of x = 20 and above y = 25

6. Point (30, 40) will divide the space to the right of the line X = 20 into two parts. Hence, we draw a line y = 40 to the right of the line x = 20

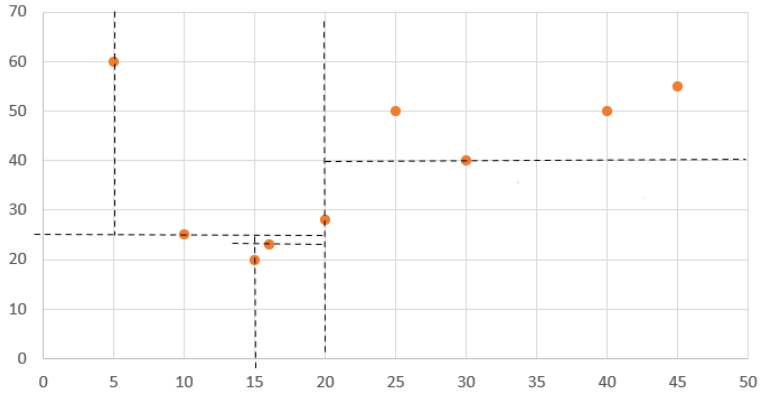


Figure 15: Draw line y = 40 to the right of x = 20

7. Point (40, 50) will divide the space to the right of line X = 20 and above line Y = 40 into two parts. So we draw a line x = 40 to the right of line x = 20 and above line y = 40.

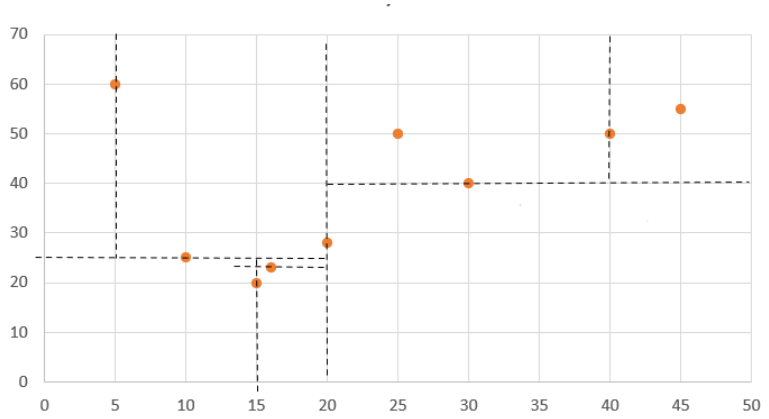


Figure 16: Draw line x = 40 to the right of x = 20 and above y = 40

8. Point (25, 50) will divide the space between lines X = 20, X = 40 and Y = 40 into two parts. Hence, we draw a line y = 50 between lines x = 20 and x = 40.

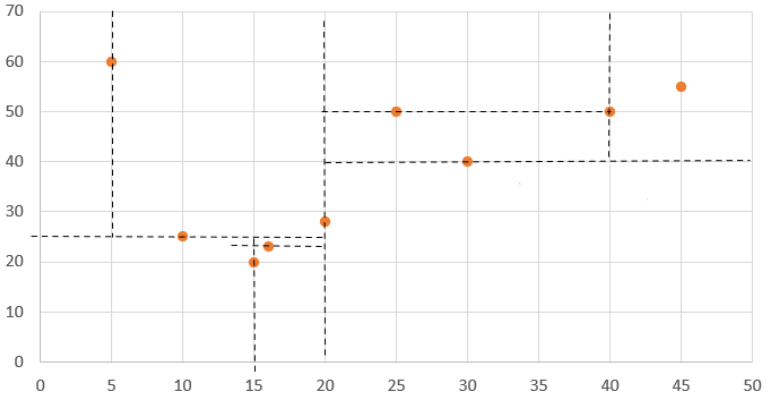


Figure 17: Drawn line y = 50 between lines x = 20 and x = 40

9. Point (45, 55) will divide the space to the right of line X = 40 and above line Y = 40 into two parts. So we draw a line y = 55 to the right of line x = 40 and above line y = 40.

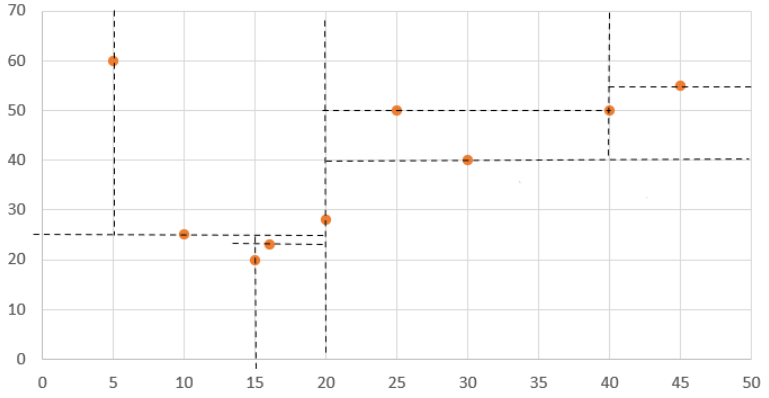


Figure 18: Draw line y = 55 to the right of x = 40 and above y = 40

As we have seen, as more and more points are getting added into our K-D Tree, we are doing more and more partitions of the space, as a result, the nearest neighbor of any member can be searched in O(log N) time.

4.2.1 Algorithms

Pseudo code of searching in K-D trees:

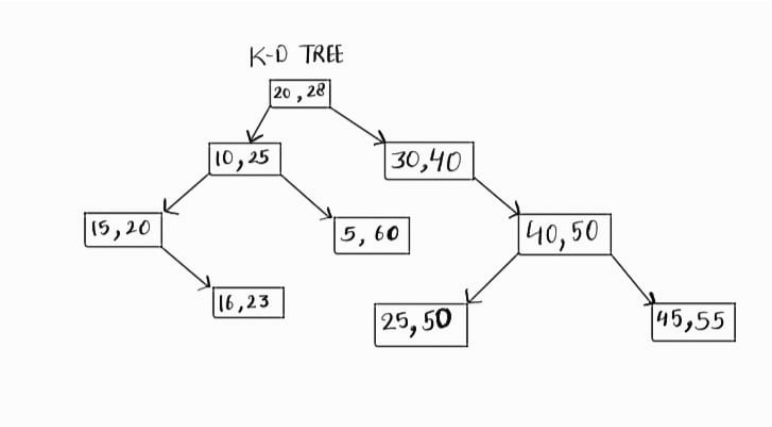
Algorithm 2 Searching in K-d tree

```
node* Search(node* x , int* point, int depth)
if x is NULL then
  return NULL
end if
if point(depth mod k) == x.point(depth mod k) then
  if the point is equal to x.point then
    return x
  else return Search(x.left, arr, depth + 1)
else if point(depth mod k) < x.point(depth mod k)
    return Search(x.left, point, depth + 1)
else
    return Search(x.right, point, depth + 1)
```

Searching works in O(logn) time complexity in the average case

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.



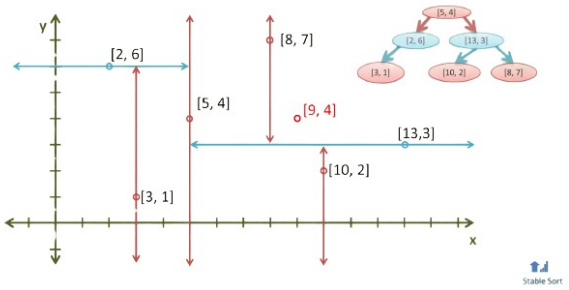

```
Inorder: (15, 20), (16, 23), (10, 25), (5, 60), (20, 28), (30, 40), (25, 50), (40, 50), (45, 55)
```

Figure 19: insertion in K-D tree

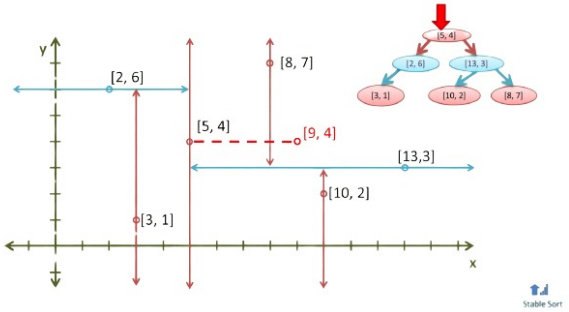
Here, as we are visiting each node only one time, hence the time complexity is O(n)

4.4. Nearest Neighbour

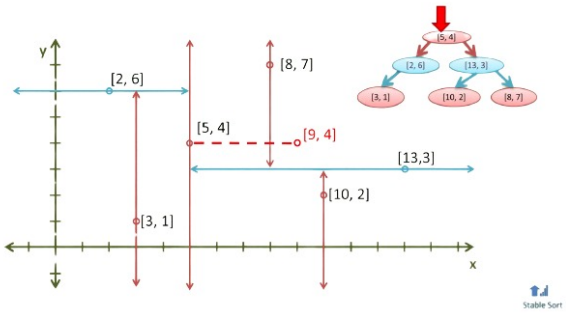
In this section, we have described our nearest neighbor algorithm by the example with query point (9, 4).
1.
We traverse the tree looking for the nearest neighbor of the query point.



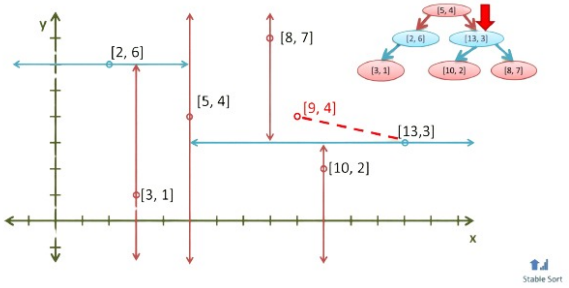
2. Compare the query point with the root node, since the x-coordinate of the query point is larger than the root node so it lies in the right of the line x = 5.



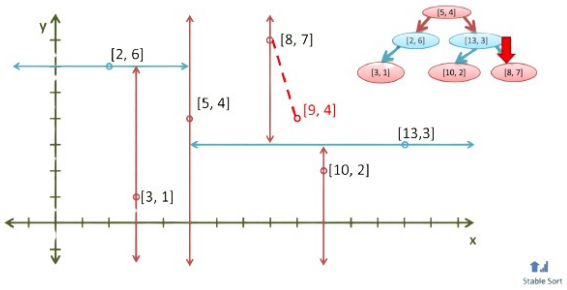
3. Explore the branch of the tree that is closest to the query point first.



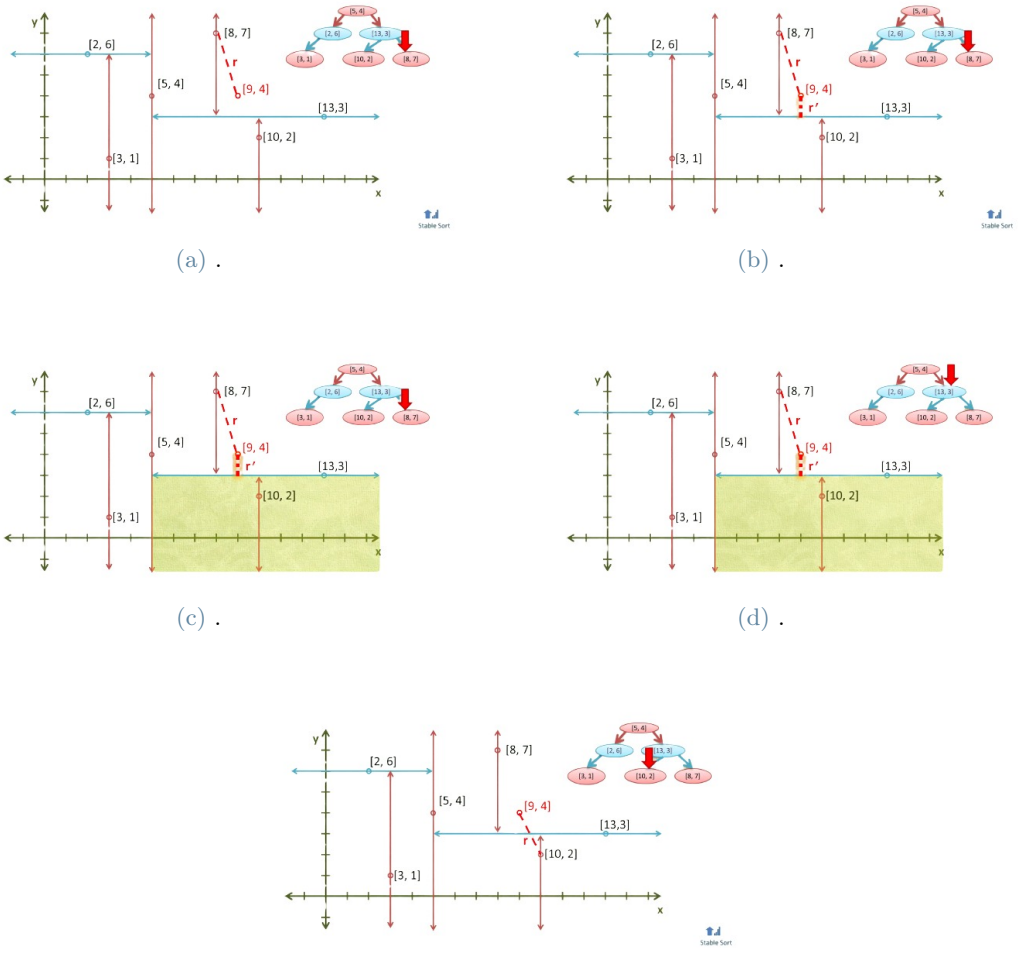
4. compute the distance of each node with the query point and mark the shortest distance.



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



6. we need to backtrack because the distance of the leaf node is larger than the distance of the line y = 3 passing through the node (13, 3) so there is the possibility of finding the nearest neighbor in the subtree of the node.



(a) . (b) .

(c) . (d) .

(e) .

Figure 20: Finding Nearest Neighbor of query point

Complete analysis of nearest neighbor search is described below.

5. Analysis of Nearest Neighbor Algorithms

When performing a query in a tree structure, the process involves traversing from the root down to a leaf node. This journey to the leaf node might require visiting all N nodes in the worst-case scenario, especially when no pruning is possible due to the problem's structure.

The time complexity varies significantly depending on the level of pruning achievable. If extensive pruning occurs, reducing the number of nodes to navigate, the complexity approaches O(log N). Conversely, if no pruning is feasible, the complexity reaches O(N) due to having to traverse through all nodes.
Therefore, the complexity of a 1-nearest neighbor (1-NN) query fluctuates within the range of O(log N) to O(N). Traversing to the starting point typically takes O(log N), while the maximum backtrack and traversal may peak at O(N) in the worst-case scenario.

6. Application

. K-D trees have a wide range of applications. As already mentioned, one of the most common uses is nearest neighbor search, where they're used to find the closest point to a given point quickly. This makes them a useful component of many machine learning approaches like k-nearest neighbors(KNN) and a wide range of distance-based clustering algorithms. The nearest neighbor search problem arises in numerous fields of application, including:
1. Machine Learning and Pattern Recognition: KD-trees are extensively used in machine learning for k-nearest neighbor (KNN) classification algorithms. They help in identifying the closest neighbors for a given point, which aids in classification tasks.
2. Multi-attribute search on online shopping sites: When browsing online shopping platforms, the presence of a filter button streamlines our search by allowing us to set preferences for the products we're interested in. This functionality aligns with the underlying concept of the K-D Tree's nearest neighbor search. The filter options, such as brand name, price, color, and various product features, are essentially mapped to distinct coordinates within a node, representing an extended version of the K-D Tree. When a user selects specific filters, the algorithm identifies the nearest data points related to the input query, presenting them in ascending order. This ensures that the most relevant or high-priority elements appear at the top of the list, while those less aligned with the user's preferences are placed towards the bottom.
3. Robotics and Autonomous Navigation: KD-trees assist robots and autonomous systems in spatial mapping, path planning, and obstacle avoidance by efficiently finding the nearest objects or obstacles in their vicinity.
4. Database queries involving a multidimensional search key: Suppose we have a query asking for all the employees are in the age-group of (40, 50) and earn 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

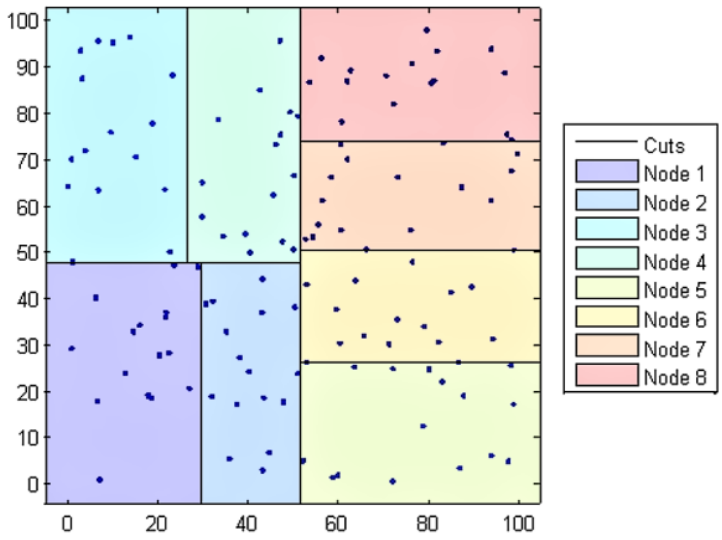


Figure 21: Multidimensional search key

A 2-dimensional k-d tree on the composite index of (age, and 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. Conclusion

The study of K-D trees has uncovered their creation, node insertion, and search processes. Their value in multidimensional spaces for nearest-neighbor search has been highlighted, demonstrating significant performance enhancements. Through the presented algorithms, K-D trees efficiently streamline complex searches, proving beneficial in various applications such as image processing, robotics, and machine learning. Ultimately, the insights emphasize the practical significance of K-D trees, showcasing their prowess in optimizing search operations and enhancing computational efficiency in diverse domains. Usually, when we traverse the K-D Tree, to search for a neighbor, we prune almost half of the nodes, and finally, on average maintain a time complexity of O(h) which is evidently equal to O(logn).

8. Biblography and citations

While doing this project, we have taken references from many different sources, both online and offline. We have used the following sources.
BAELDUNG
K-D trees wikipedia

9. Reference

[1] Bentley, Jon Louis. "Multidimensional binary search trees used for associative searching."

[2] Brown, Russell A. "Building k-d Tree in O (kn log n) Time." Journal of Computer Graphics Techniques Vol 4, no. 1 (2015).

[3] Thomas H. Cormen. Introduction to algorithms. 1990.