KD-TREE FOR KNN SEARCH ADVANCED ALGORITHMS UE20CS311

NAME : S M SUTHARSAN RAJ
SRN : PES1UG20CS362

- S M SUTHARSAN RAJ

IEEE PAPER

KD TREE FOR K NEAREST NEIGHBOUR SEARCH

S M SUTHARSAN RAJ
PES UNIVERSITY-RR CAMPUS,
Dept of CSE ,Bangalore-76
Email :
sutharsanraj2001@gmail.com

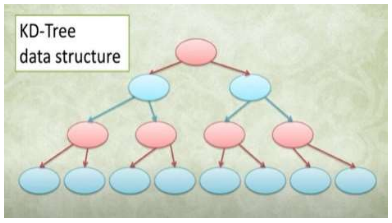


Fig 1 : A simple KD tree

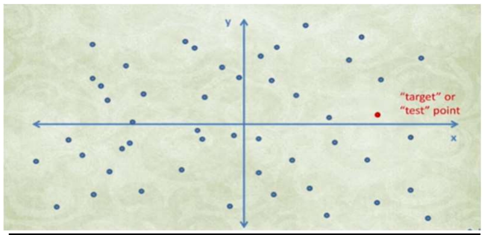


Fig 2 : The target point and training data

Abstract—

For practical applications, any neighborhood concept imposed on a finite point set P is not of any use if it cannot be computed efficiently. Thus, in this paper, we give an introduction to the data structure of **k-d trees**. Apart from that, we use this tree for finding the **k nearest neighbour** search in a **logarithmic time** and also prove that how it is better than the normal binary search tree or any other naïve algorithm for that matter.

I. INTRODUCTION

An efficient algorithm to find the nearest neighbour in logarithmic time compared to the polynomial time taken by the basic binary search tree. Also, a lot of industry applications like, for Eg: Connecting our call to nearest police station for reporting crime, in hospitals, keeping track of patients with similar medical condition, etc…
Other applications are : Efficient storage of spatial data, range search etc… We will try to incorporate these as well. Our aim is to Find Nearest Neighbour of a point from a given set of points with the help of a K-D Tree being constructed.
We also implement KNN for a given set of query points to show the multiple results. Lastly, we compare our kd tree with naïve algorithm for KNN and hence find out the efficiency factor and time complexity analysis.

1) MOTIVATION

The problem of naïve/ brute force algorithm:
Though the Brute Force Approach also finds the KNN for a given query point,
By using an array or a naïve BST, we still have complexity issue.
Brute Force/Naïve kNN algorithm calculates the squared distances from each query feature vector to each reference feature vector in the training data set. Then, for each query feature vector it selects k objects from the training set that are closest to that query feature vector. For example, in the above image if the query point is red and the other training points are blue, then to find nearest neighbour , we find each distance from red point to every other blue point and take the shortest distance. But imagine if there are k-such red points, then complexity increases drastically. So time complexity becomes TIME COMPLEXITY : O(k*n), where k is query points(red points) and n is the total blue points(training points). Definitely, this would be the slowest if there are million points in the data.

The problem of array or bst datastructure: When the points are stored in an array or bst, we tend to store either x or y coordinate, but both the points are not taken into consideration. For example, when we take an array we compare only the x values. Now let us take a case

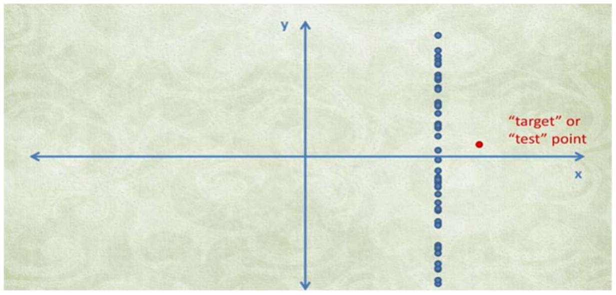


Fig 3 : The target point and skewed training data

In the above case, we have all the x values same. So, it becomes an ordeal task to find the distance from the query point to all other points as the x values are skewed, which again becomes a hefty problem. Therefore, to avoid this, KD tree comes to the rescue.

2) *KD-TREE*

A KD-Tree is necessarily a Binary Search Tree, with each node having 2 values representing the x and y coordinates. The K in KD tree refers to the dimension of the spatial extent and D refers to the Dimension. It splits points between every alternating axes. Every leaf node is a k-dimensional point. By separating space by splitting regions, nearest neighbor search can be made much faster when using an algorithm like euclidean clustering. This improved idea is to associate groups of points by how close together they are. By grouping points into regions in a KD-Tree, we can avoid calculating distance for possibly thousands of points just because we know they are not even considered in a close enough region.

II. RELATED WORKS

- ALGLIB has C# and C++ implementation s of *k*-d tree based nearest neighbor and approximate nearest neighbor algorithms
- CGAL the Computational Algorithms Library, has an implementations of *k*-d tree based nearest neighbor, approximate nearest neighbor as well as range search algorithms.
- SciPy, a Python library for scientific computing, contains implementations of *k*-d tree based nearest neighbor lookup algorithms.
- scikit-learn, a Python library for machine learning, contains implementations of *k*-d trees to back nearest neighbor and radius neighbours searches.
- *k*-d trees are a useful data structure for several applications, such as searches involving a multidimensional search key (e.g. range searches and nearest neighbor searches) and creating point clouds. *k*-d trees are a special case of binary space partitioning trees.

III. METHODOLOGY

1) *CREATING K-D Tree*

- First inserted point becomes root of the tree.
- Select axis based on depth so that axis cycles through all valid values. In our example: axis = depth % 2 , where depth of Root is 0, and axis = 0/1 means to choose x/y axis accordingly.
- Sort point list by axis and choose median as pivot element. If less branch left, if greater branch right.
- Traverse tree until node is empty, then assign point to node.
- Repeat step 2-4 recursively until all of the points processed.

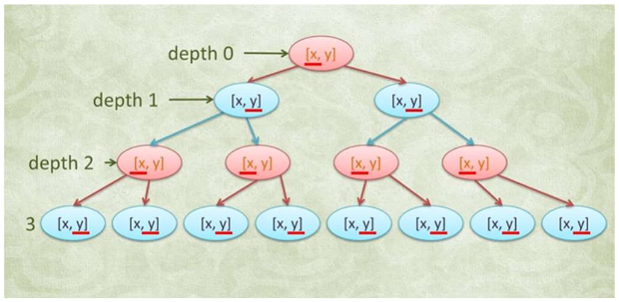


Fig 4 : Representation of KD tree and depicts the construction of a kd tree.

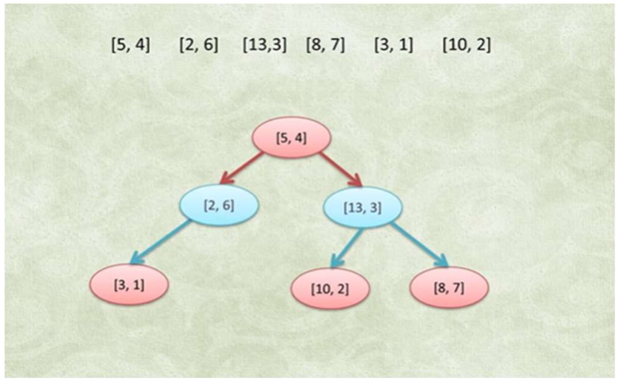


Fig 5 : Constructing KD Tree with given set of points

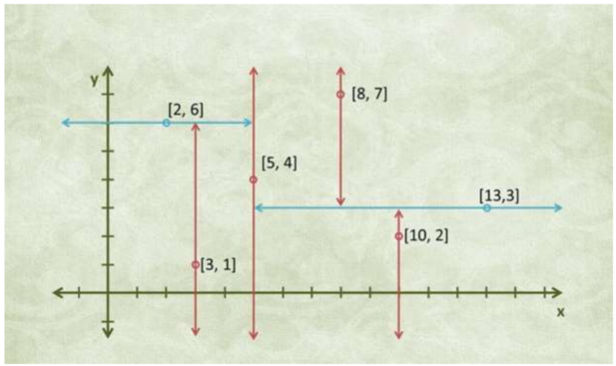


Fig 6 : Each point's coordinate split sections by its respective x and y-axis. All the points split the spatial extent as seen in the above manner.

2) *PERFORMING THE KNN SEARCH*

- Starting with the root node, the algorithm moves down the tree recursively, in the same way that it would if the search point were being inserted (i.e. it goes left or right depending on whether the point is lesser than or greater than the current node in the split dimension).
- Once the algorithm reaches a leaf node, that node point is saved as the "current best".
- The algorithm unwinds the recursion of the tree, if the current node is closer than the current best, then it becomes the current best.

In the below example our query point is (9,4)

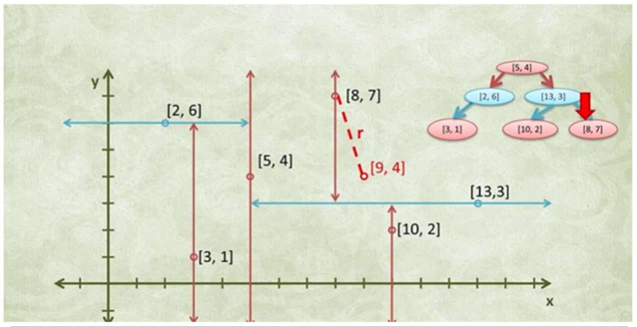


Fig 7 : Recursively we reach to the target point as (8,7) but still necessarily not the best.

Just by normal traversal we find our KNN for [9,4] to be [8,7] but still is not the best solution. There is still possibility of a much closer point. Think 'r' to be the shortest distance.
The distance r can be at any angle. We may not know.
But this is not a problem!!!

- Solving the above issue,

the algorithm also checks whether there could be any points on the other side of the splitting plane that are closer to the search point than the current best. In concept, this is done by intersecting the splitting hyperplane with a hypersphere around the search point that has a radius equal to the current nearest distance.
If the hypersphere crosses the plane, there could be nearer points on the other side of the plane, so the algorithm must move down the other branch of the tree from the current node looking for closer points, following the same recursive process as the entire search.
If the hypersphere doesn't intersect the splitting plane, then the algorithm continues walking up the tree, and the entire branch on the other side of that node is eliminated, meaning to say, let us see the following continuing example

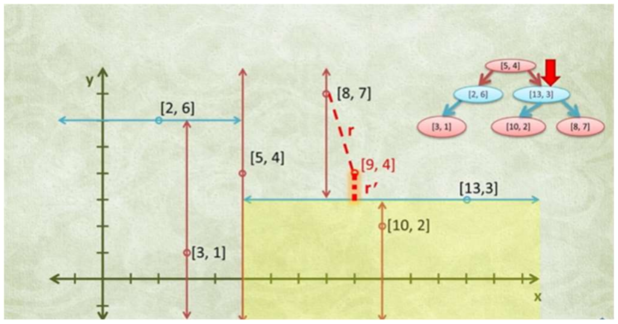


Fig 8 : Checking for a perpendicular line from the query point to the nearest spatial extent.

If we have a perpendicular line directly down to the section not visiting while recursive call is shorter then there is higher possibility that section may have the closer point, that is the KNN of our query point [9,4]
Hence, when we recurse back, we get the sectional distance r' and if r' < r, then r' is the best distance.
Hence KNN is [10,2] !!!

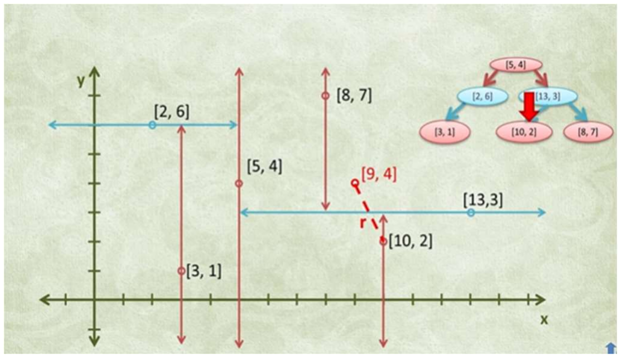


Fig 9 : The distance is now updated to the best distance between (9,4) and (10,2)

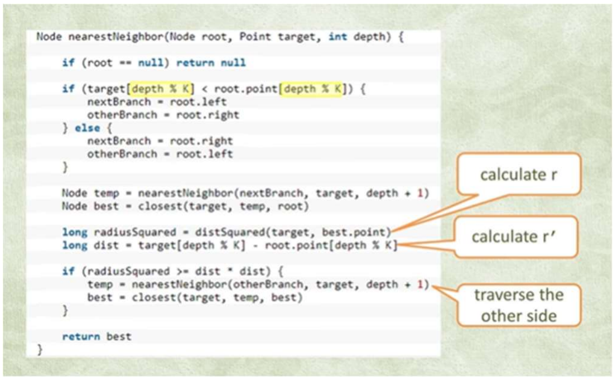
Hence the biggest challenge of doing KNN is completely solved by the help of the KD tree.
Also, the mini hurdles of finding the best solution is also overcome by the structure and functioning if the KD tree as we had seen above.

Remember :: We don't visit all the nodes of tree. If height of tree is h, then we at most visit 2h nodes, which means, it is logarithmic. When the algorithm reaches the root node, then the search is complete.

3) *ALGORITHM FOR KNN SEARCH*

```
1: procedure NNk-dTree(point p, k-d tree T, distance ε, number k)
2: L ←empty list
3: return NNk-dTreeRec(p,root,ε, k,L)
4: end procedure
5: procedure NNk-dTreeRec(point p, k-d tree T,          distance ε, number k, list L)
6: if T = ∅ then
7: return L
8: end if
9: Extract point pj from T.root and store it in L if kpj − pk ≤ ε.
10: if L is larger than k then
11: Delete the point with largest distance to p from L.
12: end if
13: if T is just a leaf then
14: return L
15: end if
16: if T.root.leftSubtree contains p then
17: T1 = T.root.leftSubtree, T2 = T.root.rightSubtree
18: else
19: T2 = T.root.leftSubtree, T1 = T.root.rightSubtree
20: end if
21: NNk-dTreeRec(p, T1, ε, k,L)
22: if /L/ < k and kp − T.root.hyperplanek < ε then
23: NNk-dTreeRec(p, T2, ε, k,L)
24: else if kL.farthest − pk > kp − T.root.hyperplanek and kp − T.root.hyperplanek ≤ ε then
25: NNk-dTreeRec(p, T2, ε, k,L)
26: end if
27: return L
28: end procedure
```

This makes up the algorithm. Now let us see some of the analysis



```
Node nearestNeighbor(Node root, Point target, int depth) {

    if (root == null) return null

    if (target[depth % K] < root.point[depth % K]) {
        nextBranch = root.left
        otherBranch = root.right
    } else {
        nextBranch = root.right
        otherBranch = root.left
    }

    Node temp = nearestNeighbor(nextBranch, target, depth + 1)
    Node best = closest(target, temp, root)

    long radiusSquared = distSquared(target, best.point)
    long dist = target[depth % K] - root.point[depth % K]

    if (radiusSquared >= dist * dist) {
        temp = nearestNeighbor(otherBranch, target, depth + 1)
        best = closest(target, temp, best)
    }

    return best
}
```

Fig 10 : Depicting the parameters r and r' in the algorithm.

IV. ANALYSIS AND COMPLEXITY

1) *COMPLEXITY ANALYSIS ON BRUTE FORCE TECHNIQUE*

Doing a KNN search by a brute force technique takes $O(k * n)$ time, where k is the number of different query points

2) *COMPLEXITY ANALYSIS ON KD TREE*

- Building a static KD-Tree from n points takes $O(n\log\ n)$ on average
- Insert a new point into a balanced k-d tree takes $O(\log\ n)$ time.
- Find nearest neighbor in a balanced KD-Tree takes $O(\log\ n)$ time on average.

As for the prediction phase, the k-d tree structure naturally supports "k nearest point neighbours query" operation, which is exactly what we need for kNN. The simple approach is to just query k times, removing the point found each time — since query takes O(log(n)), it is O(k * log(n)) in total. But since the k-d tree already cuts space during construction, after a single query we approximately know where to look — we can just search the "surroundings" around that point. Therefore, practical implementations of k-d tree support querying for whole k neighbours at one time and with complexity O(sqrt(n) + k), which is much better for larger dimensionalities, which are very common in machine learning.

V. RESULTS

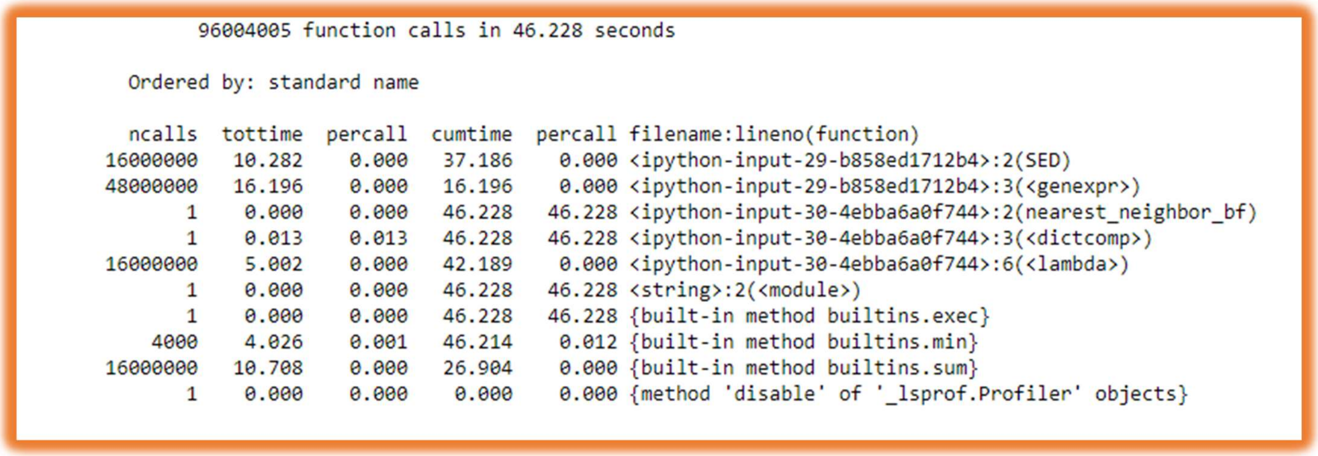
Both the Brute Force Approach and the KD tree approach for KNN search is implemented in the jupyter notebook with respect to the algorithms as discussed above and the following results were shown :

TABLE I. THIS IS THE HEADING FOR A TABLE

METHODS	Number of Function Calls	Wall Clock Time in (sec)
NAÏVE / BRUTE FORCE	96004005	46.228
KD TREE	425748	0.406

[a] Summary of Result analysis

TABLE II : BRUTE FORCE METHOD/NAÏVE KNN for 4000 random points and selected random query points

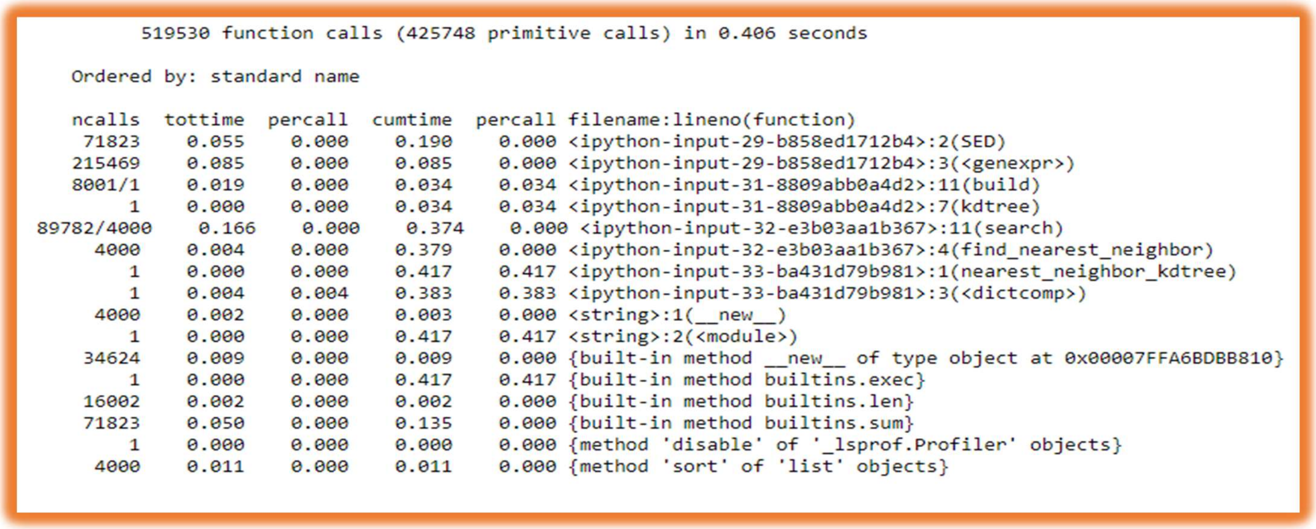


```
         96004005 function calls in 46.228 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
 16000000   10.282    0.000   37.186    0.000 <ipython-input-29-b858ed1712b4>:2(SED)
 48000000   16.196    0.000   16.196    0.000 <ipython-input-29-b858ed1712b4>:3(<genexpr>)
        1    0.000    0.000   46.228   46.228 <ipython-input-30-4ebba6a0f744>:2(nearest_neighbor_bf)
        1    0.013    0.013   46.228   46.228 <ipython-input-30-4ebba6a0f744>:3(<dictcomp>)
 16000000    5.002    0.000   42.189    0.000 <ipython-input-30-4ebba6a0f744>:6(<lambda>)
        1    0.000    0.000   46.228   46.228 <string>:2(<module>)
        1    0.000    0.000   46.228   46.228 {built-in method builtins.exec}
     4000    4.026    0.001   46.214    0.012 {built-in method builtins.min}
 16000000   10.708    0.000   26.904    0.000 {built-in method builtins.sum}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

TABLE III : KD TREE METHOD KNN for 4000 random points and selected random query points



```
         519530 function calls (425748 primitive calls) in 0.406 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
    71823    0.055    0.000    0.190    0.000 <ipython-input-29-b858ed1712b4>:2(SED)
   215469    0.085    0.000    0.085    0.000 <ipython-input-29-b858ed1712b4>:3(<genexpr>)
   8001/1    0.019    0.000    0.034    0.034 <ipython-input-31-8809abb0a4d2>:11(build)
        1    0.000    0.000    0.034    0.034 <ipython-input-31-8809abb0a4d2>:7(kdtree)
89782/4000    0.166    0.000    0.374    0.000 <ipython-input-32-e3b03aa1b367>:11(search)
     4000    0.004    0.000    0.379    0.000 <ipython-input-32-e3b03aa1b367>:4(find_nearest_neighbor)
        1    0.000    0.000    0.417    0.417 <ipython-input-33-ba431d79b981>:1(nearest_neighbor_kdtree)
        1    0.004    0.004    0.383    0.383 <ipython-input-33-ba431d79b981>:3(<dictcomp>)
     4000    0.002    0.000    0.003    0.000 <string>:1(__new__)
        1    0.000    0.000    0.417    0.417 <string>:2(<module>)
    34624    0.009    0.000    0.009    0.000 {built-in method __new__ of type object at 0x00007FFA6BDBB810}
        1    0.000    0.000    0.417    0.417 {built-in method builtins.exec}
    16002    0.002    0.000    0.002    0.000 {built-in method builtins.len}
    71823    0.050    0.000    0.135    0.000 {built-in method builtins.sum}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
     4000    0.011    0.000    0.011    0.000 {method 'sort' of 'list' objects}
```

VI. NOTATIONS

- **Query Points :** The points , whose KNN is to be found for
- **Target Points :** The point, other than the training points
- **KNN : ** K Nearest Neighbours
- **KD Tree :** K dimensional Tree

VII. CONCLUSIONS

Return of the nearest neighbour and comparison with the naïve algorithm by showing the wall clock time too. Successful insertion and search are also displayed.

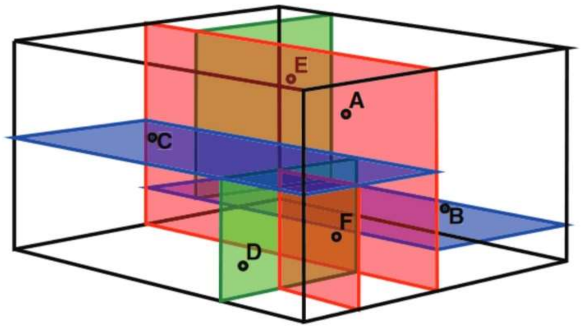
We also see a drastic improvements in performance of KD tree when compared to Naïve method.

Hence, KD is optimal for KNN by this approach.

VIII. FUTURE WORKS

The use of KD tree in

1. Efficient storage of spatial data, and
2. Database queries involving a multidimensional search key
3. Working with 3D or Higher Dimension search will be worked upon.



IX. ACKNOWLEDGEMENT

I thank my teacher and the teaching assistants for assisting me in accomplishing the working on this paper. I also thanks the authors of the reference research papers for helping me with the contents and design of this paper.

X. REFERENCES

[1] The k-d tree data structure and a proof for neighborhood computation in expected logarithmic time," *Martin Skrodzki.*, vol. 24, no. 2, pp. 379–393, March 2019 doi: 10.1080/10618600.2014.901225.

[2] The k-d tree data structure – Yasen Hu – 2021

[3] ACM Transactions on GraphicsVolume 27Issue 5December 2008 Article No.: 126pp 1–11https://doi.org/10.1145/1409060.1409079

[4] An Improved Algorithm Finding Nearest Neighbor Using Kd-trees Rina Panigrahy Microsoft Research, Mountain View CA, USA rina@microsoft.com