University of Rhode Island

Department of Computer Science and Statistics

Data Structure Course Project

K-D Tree

Group Members:

Sathiarith Chau

Max Faramarzi

Jarell Rosa

Advisor:

Christian Esteves

Fall 2021

Table of Contents

ABSTRACT

The k-d tree is a multi-dimensional version of the binary search tree where each level of the tree represents the axis in which the data point partitions the space. This type of tree is thereful useful to store geospatial data (the X, Y, and Z axes). The search time is efficient when retrieving particular points, nearest, or ranges of points is due to the ability of the tree to exclude particular search spaces (i.e., particular nodes). The objective of the group was to implement our own k-d database and have it be dynamic to the needs of the user. Upon the end of our project timeline, we were able to successfully create the algorithm to build a k-d tree database and perform search and insert functionality on that. Also visual illustration of tree structure using Graphviz validated the outcome of the program. K-Nearest Neighbor algorithm search would be our next goal in testing the spatial partitioning aspect of the database.

Keywords**:** K-D Tree; Search; Insert, Graphviz

I. INTRODUCTION

A k-dimensional (k-d) tree is a space-partitioning data structure which affords quick search times under ideal circumstances. The k is the number of dimensions the data will occupy. The classic example of two particular dimensions would be X and Y coordinates. In terms of time complexity, its average case for searches is log n, due to the ability to exclude certain subtrees. For application, K-d trees are typically used for nearest neighbor searches, range searches, and creating point clouds (i.e., data representing points in space.

The aforementioned space-partition descriptor highlights how each node (data) is placed in the tree. Like the binary search tree, data is allocated to the left or right side of the tree (and sequent subtrees) through comparison of it to the parent nodes. Additionally, each level of the tree (depth) designates a dimension of the data which is used to discriminate how each data traverses down the tree [1]. In figure 1, the data is said to be 2-dimensional in nature due to containing an X value and a Y value. Based on the dimension of the example data, the less-than or greater-than comparisons determine whether the first or second data is used [2]. For example, at level 0, only the first number are compared to traverse down the tree.

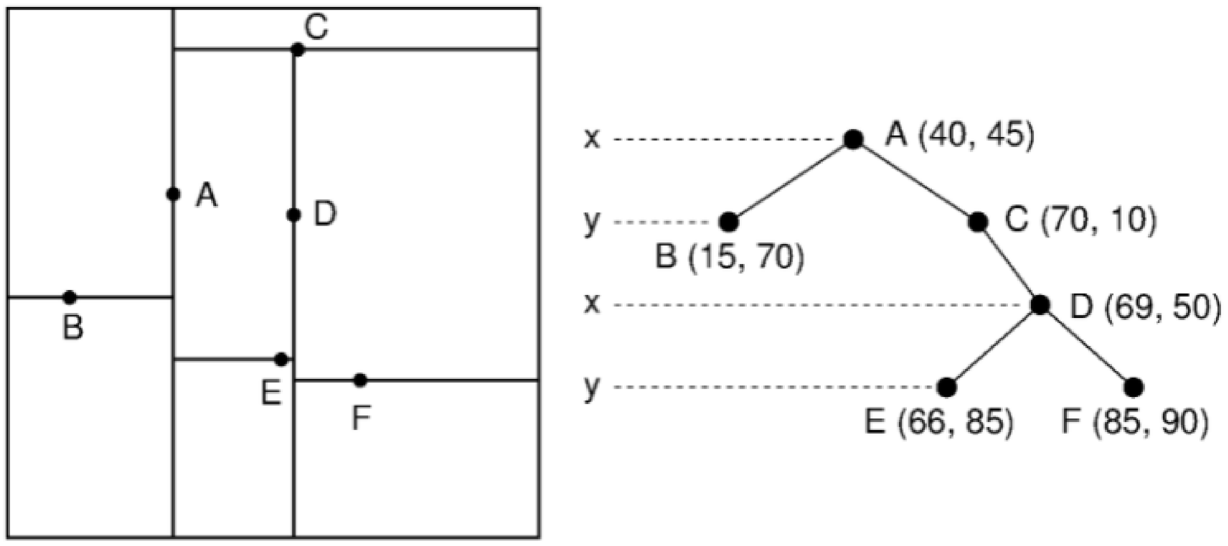


Figure 1. Organization of a 2-dimensional dataset [3]

A. Limitations

In order to perform in O(logn) search time, there are certain requirements that need to be met. First, the tree should be constructed with a median-selection algorithm with respect to the dimension of each level in order to be balanced. That way, the tree should have roughly the same number of nodes on both sides.

The number of dimensions also needs to be considered as it dictates the minimum amount of data points needed in the tree when applied to various search algorithms such as the Nearest Neighbor Search [4]. Any less would result in having not enough data to represent spatial data. Essentially, the end result would be that search time would be the same as an exhaustive search.

B. Project Roadmap

The objective of this project are as below:

1. Construct a K-D tree class using C++ programming language, with some functionality such as searching and inserting.
2. Populating the K-D Tree with randomly generated K-D data.
3. A visual representation of the K-D tree through making a DOT file of connections and feeding that to the Graphviz online [5].

The number of dimensions, number of points and maximum range of points are arbitrary and input by the user in the main.cpp file.

The next step would be to develop a K-Nearest Neighbor (KNN) algorithm to test a given point and find the closest point. K-NN would highlight the ability of our n-dimensional tree to retrieve the closest point quicker when compared to another database (exhaustive search through vector storage).

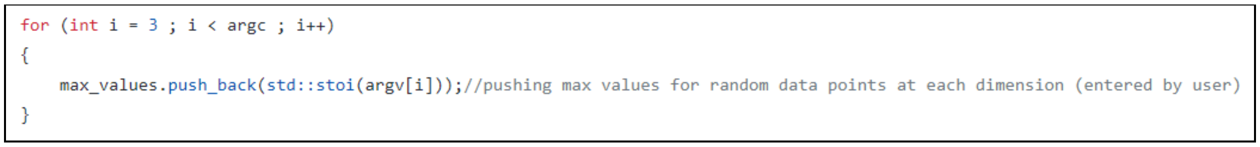
II. IMPLEMENTATION

C. Graphviz

Graphviz is open source graph visualization software. Graph visualization is a way of representing structural information as diagrams of abstract graphs and networks. In data science, one use of Graphviz is to visualize decision trees [5].

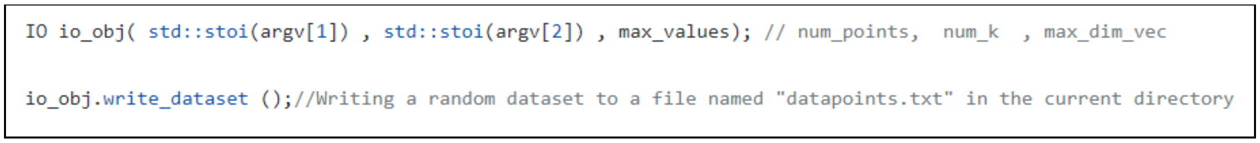
D. Input

1) Within the main function, there was a class *write_dataset()* method which takes in command line arguments (*argv[1]* and *argv[2]*) along with a vector of k integers to designate maximum int values for our randomly generated points (Figure 3). Since the fourth argument (*argv[3]*) determines the maximum values but is cstring data type, it is passed to a for-loop to set the number of elements in the vector to k and assigning the value to the maximum number specified (Figure 2).



```
for (int i = 3 ; i < argc ; i++)
{
    max_values.push_back(std::stoi(argv[i]));//pushing max values for random data points at each dimension (entered by user)
}
```

Figure 2. Passing *argv[i]* to set number of elements and max values each.



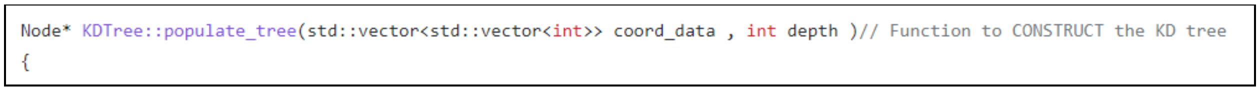
```
IO io_obj( std::stoi(argv[1]) , std::stoi(argv[2]) , max_values); // num_points,  num_k  , max_dim_vec

io_obj.write_dataset ();//Writing a random dataset to a file named "datapoints.txt" in the current directory
```

Figure 3. Passing command line arguments and vector to generate dataset.

2) The last step of *write_dataset()* creates a text file called “"datapoints.txt" to be populated into the K-D tree.

III. Populating Tree

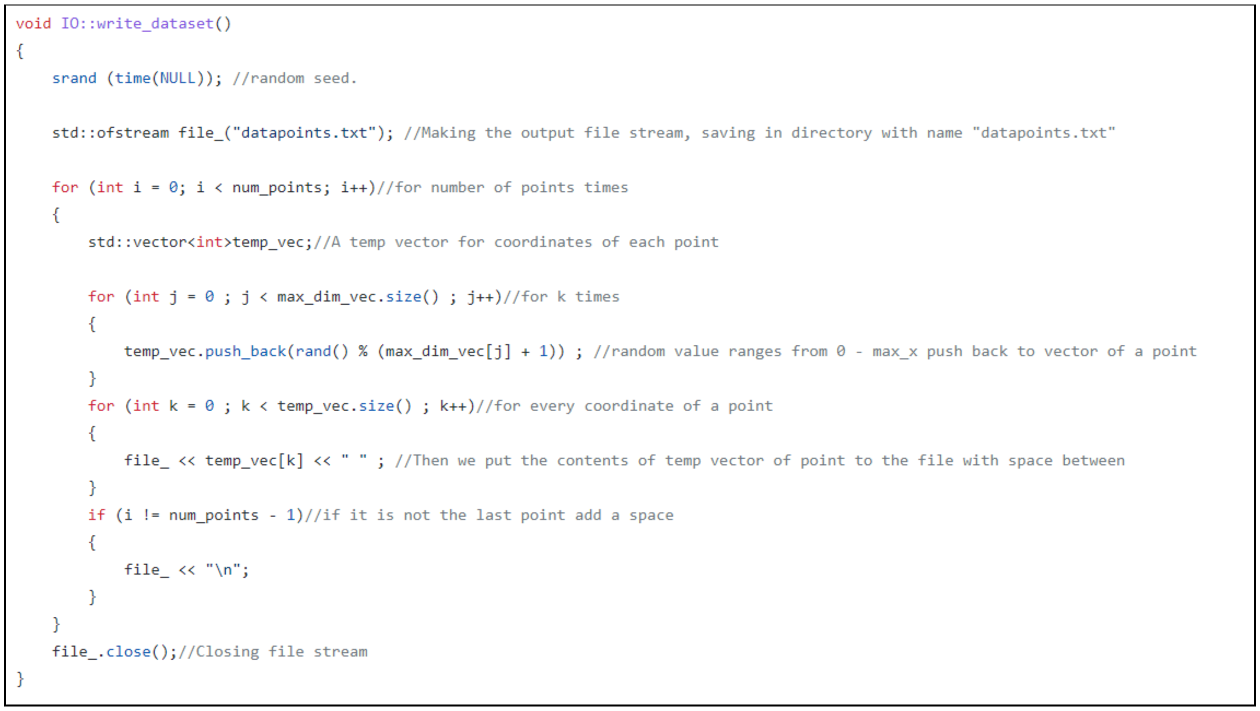
The main function that handles the population of the K-D Tree and therefore its creation is the *populate_tree()* method. This method takes in two parameters, the randomly created coordinate data points as a 2 dimensional vector and the current depth of the tree (zero) as shown in figure 4.



```
Node* KDTree::populate_tree(std::vector<std::vector<int>> coord_data , int depth )// Function to CONSTRUCT the KD tree
{
```

Figure 4. The parameters for the *populate_tree()* method

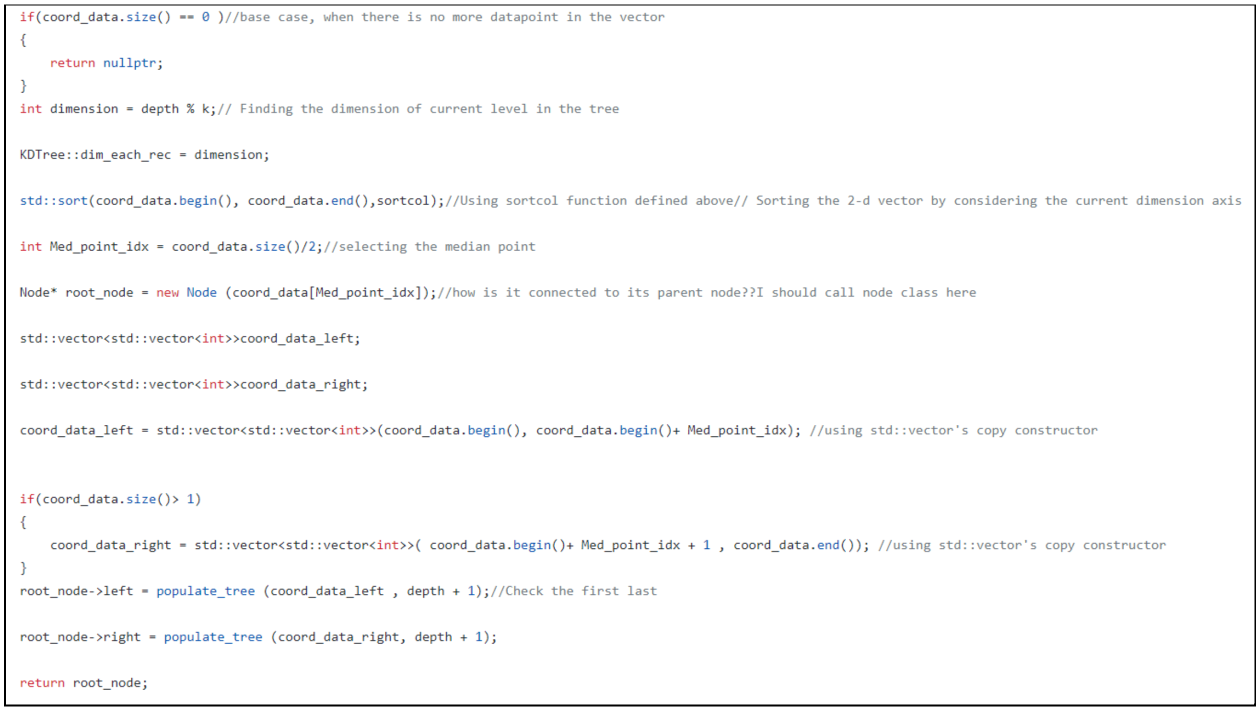
It then uses the depth and the dimension of the tree (k) to compute which dimension the sorting algorithm



```
void IO::write_dataset()
{
    srand (time(NULL)); //random seed.

    std::ofstream file_("datapoints.txt"); //Making the output file stream, saving in directory with name "datapoints.txt"

    for (int i = 0; i < num_points; i++)//for number of points times
    {
        std::vector<int>temp_vec;//A temp vector for coordinates of each point

        for (int j = 0 ; j < max_dim_vec.size() ; j++)//for k times
        {
            temp_vec.push_back(rand() % (max_dim_vec[j] + 1)) ; //random value ranges from 0 - max_x push back to vector of a point
        }
        for (int k = 0 ; k < temp_vec.size() ; k++)//for every coordinate of a point
        {
            file_ << temp_vec[k] << " " ; //Then we put the contents of temp vector of point to the file with space between
        }
        if (i != num_points - 1)//if it is not the last point add a space
        {
            file_ << "\n";
        }
    }
    file_.close();//Closing file stream
}
```

Figure 5. Body for the data points method of *IO*

should run on. Using this current dimension the method sorts the dataset based on that dimension, and then finds the midpoint of the dataset. With this a node is created with that datapoint, two new 2 dimensional vectors are created (for the vectors greater than and less than the medium found) and the method is recursively called on each side until the tree is fully populated using the coordinate data points as shown in figure 6.

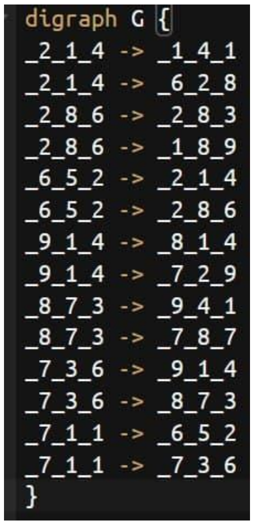


```
if(coord_data.size() == 0 )//base case, when there is no more datapoint in the vector
{
    return nullptr;
}
int dimension = depth % k;// Finding the dimension of current level in the tree

KDTree::dim_each_rec = dimension;

std::sort(coord_data.begin(), coord_data.end(),sortcol);//Using sortcol function defined above// Sorting the 2-d vector by considering the current dimension axis

int Med_point_idx = coord_data.size()/2;//selecting the median point

Node* root_node = new Node (coord_data[Med_point_idx]);//how is it connected to its parent node??I should call node class here

std::vector<std::vector<int>>coord_data_left;

std::vector<std::vector<int>>coord_data_right;

coord_data_left = std::vector<std::vector<int>>(coord_data.begin(), coord_data.begin()+ Med_point_idx); //using std::vector's copy constructor


if(coord_data.size()> 1)
{
    coord_data_right = std::vector<std::vector<int>>( coord_data.begin()+ Med_point_idx + 1 , coord_data.end()); //using std::vector's copy constructor
}
root_node->left = populate_tree (coord_data_left , depth + 1);//Check the first last

root_node->right = populate_tree (coord_data_right, depth + 1);

return root_node;
```

FIgure 6. The body of the *populate_tree()* method

IV. RESULTS (OUTPUT)

The main output function that was implemented was the *write_dot_file()* method and the *inOrder()* method that aided that. The *inOrder()* method is an in order traversal of the tree which by definition recursively traverses the left branches, does an operation (in this case pushing back data into the *dot_fil_conn* vector of pairs of vectors), and then traverses the right branches. The pairs of vectors are the source and destination coordinates respectively of the tree. The *write_dot_file()* method then uses this vector to create a dot file with the format indicated in the black picture to the left to create the graph visualized on the right.



```
digraph G {
_2_1_4 -> _1_4_1
_2_1_4 -> _6_2_8
_2_8_6 -> _2_8_3
_2_8_6 -> _1_8_9
_6_5_2 -> _2_1_4
_6_5_2 -> _2_8_6
_9_1_4 -> _8_1_4
_9_1_4 -> _7_2_9
_8_7_3 -> _9_4_1
_8_7_3 -> _7_8_7
_7_3_6 -> _9_1_4
_7_3_6 -> _8_7_3
_7_1_1 -> _6_5_2
_7_1_1 -> _7_3_6
}
```

Figure 7. Dot file code of a K-D Tree connections

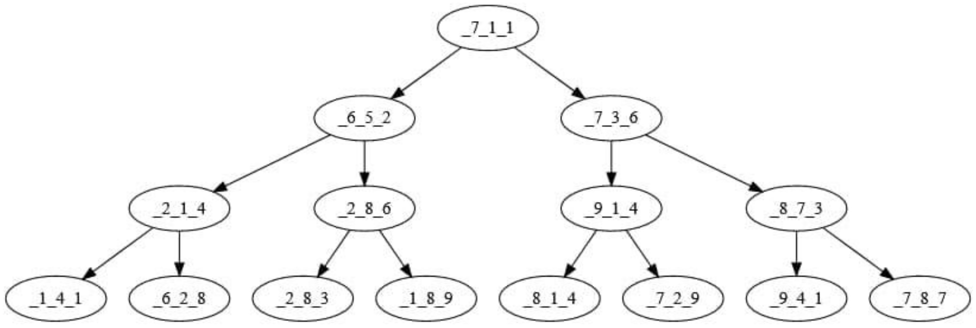


Figure 8. Three-dimensional structure of sample dataset (k = 3, n = 15)

VI. DISCUSSIONS

Since the kd tree is a modification to the BST we were initially inspired by that concept to kick start this project. Then the differences were considered, which was that the kd tree makes branching decisions based on a particular search key associated with that level, called the discriminator. We also considered balancing techniques in K-D Tree implementation to achieve a more efficient algorithm. Implementing the actual "*populating_tree"* and also developing the code for the correct output of data points and connections was more complex than initially thought. There were multiple versions of data structures, and algorithms including recursion, vectors, pairs, and iterative solutions that we tested.

VII. REFERENCES

[1] Bentley, J. L. (1975). "Multidimensional binary search trees used for associative searching". *Communications of the ACM*. **18** (9): 509–517. doi:10.1145/361002.361007. S2CID 13091446.

[2] Berg, Mark de; Cheong, Otfried; Kreveld, Marc van; Overmars, Mark (2008). "Orthogonal Range Searching". *Computational Geometry*. pp. 95–120. doi:10.1007/978-3-540-77974-2_5. ISBN 978-3-540-77973-5.

[3] KD Trees, retrieved Dec 8 2021 from: https://opendsa-server.cs.vt.edu/ODSA/Books/CS3/html/KDtree.html

[4] Radovanović, M.; Nanopoulos, A.; Ivanović, M. (2010). *On the existence of obstinate results in vector space models*. 33rd international ACM SIGIR conference on Research and development in information retrieval - SIGIR '10. p. 186. doi:10.1145/1835449.1835482. ISBN 9781450301534.

[5] GraphvizOnline, retrieved Dec 8 2021 from: https://dreampuf.github.io/GraphvizOnline/#digraph%20G%20%7B%0A%0A%20%20l1%20-%3E%20l2%3B%0A%20%20l2%20-%3E%20l3%3B%0A%0A%7D

[6] Visualizing Decision Trees,retrieved Dec 8 2021 from: https://towardsdatascience.com/visualizing-decision-trees-with-python-scikit-learn-graphviz-matplotlib-1c50b4aa68dc