Algorithms
CS2223

K-D Tree Spatial Structure
Xeropsamobeus Team

Submission by:
William Burke
Josue Contreras
Ian Coolidge
Mayank Govilla

Abstract

The goal of the project was to implement a 2-dimensional tree data structure (k-d tree implementation) and evaluate its performance of datasets of increasing size N. The nearest-neighbor query search was tested, and its evaluated performance outperformed the brute force approach on random data sets of size N. Additionally, an interactive GUI was implemented to visualize the k-d trees spatial structure in a 2-d space.

Introduction

K-D trees are a simple and efficient space partitioning data structure. This tree structure is widely used in numerous applications like computer graphics, neural networks, data mining, and data analysis. The two main queries that take advantage of this data structure are nearest-neighbor search and range search. Nearest neighbor search finds the closest point to a queried point. Range search finds all points contained by the given query region. One interesting application that our team came across when researching k-d trees was the implementation of a 2-d tree to create a flocking boids simulator. This simulator uses this spatial structure to direct a flock of birds to fly together while avoiding the predator hawk. The animations provided by this article demonstrate the efficiency of this structure as they can manipulate the boids without any observable straggle.[1]

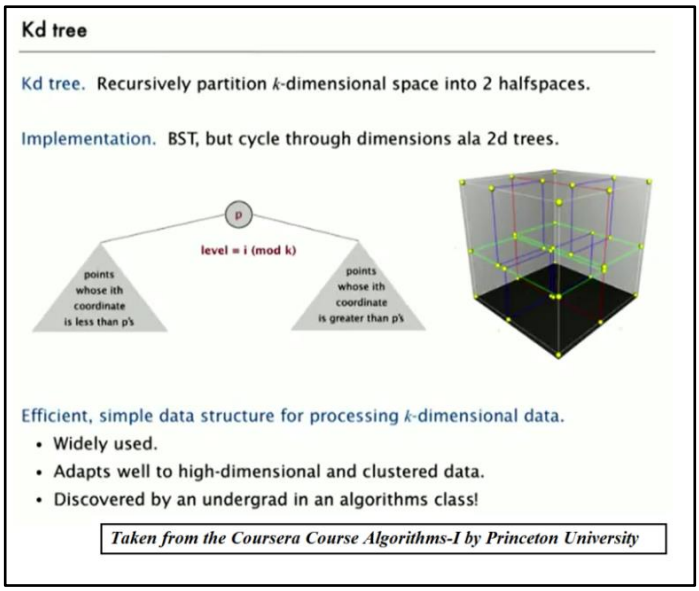

Figure 1: KD-Tree Overview by Princeton University[1]

A K-th dimension search tree, or k-d tree is an implementation of a Binary Search Tree (BST) that is primarily used to store two dimensional points on a plane. The nodes that make up the BST have a high and low child, set of points (many times x,y) and discriminator. The

[1] Sadipan Dey, *Implementing a KD-tree*, (sadnipanweb, 2017)

discriminator is a key value calculated as depth modulus k (where k is the dimension) that signifies how the node is aligned. If the k value of a tree is 2, the root node would have a discriminator value of 0, signifying that it is x-aligned. Its children would have a discriminator value of 1, signifying they are y-aligned. When adding or searching nodes, the parent's discriminator must be considered so that the correct point values are being compared. For example, when adding to an **x-aligned** root node, the **x** values of the new node and root node would be compared, traversing down the left (low) or right (high) side of the tree depending if the new x value is lower or higher. Like any BST, this would be done recursively until the node reaches a null child node where it can be inserted. The following is an example of adding points (in order) to a KD tree where k=2. [2, 3]

Example (inserting points to an empty k-d tree):

- *Points to be added:* (10, 12), (14, 5), (6, 18), (16, 7), (19, 1)
- Insert: *Point* (10, 12)
  - x-aligned
- Insert: *Point* (14, 5)
  - Since 14 > 10, it is stored as the high (right) child
- Insert: *Point* (6, 18)
  - Since 6 < 10, it is stored at the low (right) child
- Insert: *Point* (16, 7)
  - Since 16 > 10, so it will traverse down the right side.
  - The next node is y-aligned, so the y values of parent node (14, 5) and new node (16, 7) will be compared, and (16, 7) is added as the right child of (14, 5) because 7 > 5
- Insert: *Point* (19, 1)
  - This point traverses down right side and is added as left child of (14, 5) because 1 < 5

[2] Andrew W. Mooore, *An introductory tutorial on kd-trees*, (Carnegie Mellon University, 1991)
[3] Jon Louis Bently, *Multidimensional Binary Search Trees Used for Associative Searching*, (Stanford University, 1975)

K-D Tree Design

Our team used the original KD tree paper to develop an implementation using recursion.[2] Much like similar data structures such as a binary search tree, we used a KDNode class to store the data for a single node, including pointers to its children, and an overall KDTree class to store the root of a tree and several methods global to the tree. Using our understanding from the original paper, recursive methods are used to traverse the tree similarly to a normal binary search tree in order to check if a node is contained in the tree or get or insert nodes. In addition to this, our team also chose to implement the Iterable interface for our KD tree class. The iterator for this class implements an in-order traversal of the tree, allowing the user of the class to utilize the set of individual points in the tree.

One of the important algorithms that makes k-d trees useful is an implementation of nearest neighbor that can run in log n time (crucially only for a balanced tree). The algorithm first finds the leaf node where point P would have been inserted. As the recursion unwinds, the algorithm searches for nodes that could possibly be closer on the other side of the tree. Figure 2 demonstrates an example of a closer node existing on the other side of the tree. To find these nodes, the perpendicular distance from P to the current node's discriminator is calculated. If the perpendicular distance is less than the current closest distance, then the other side of the tree must be searched for any closer nodes. If a node is found, the distance is compared with the original closest node, and the closer one is returned. This algorithm can find the nearest neighbor in at best **Ω**(log n) time, but imbalances in the tree can cause a worst case of O(n) time.

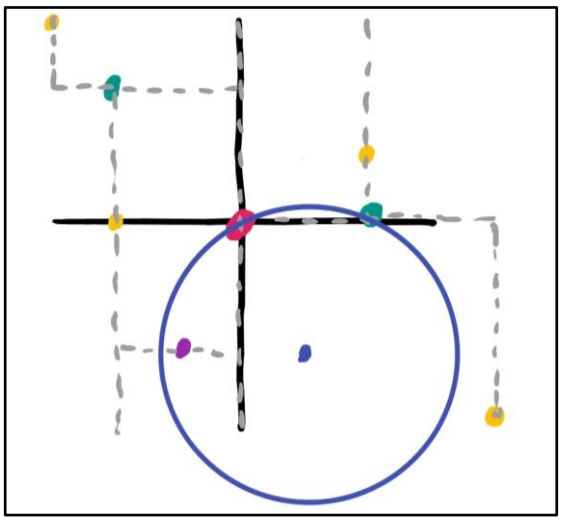

Figure 2: Nearest Neighbor is on the other side of the tree

Along with the nearest neighbor method our team implemented the range search method. This method takes advantage of the KD trees' recursion to find the points within the given region. Our team used a Linked List data structure to retrieve the points within the region. This allows for N number of arbitrary points to be added without the need for resizing the structure.

The implementation of this query is simple since it recurses through the tree until it finds the node with a region that contains the specified region. Then a helper function “drain” was implemented to traverse the left and right subtrees of this node and add each point to the linked list. In theory the average performance of the range query is $O(n^{1-1/d} + r)$, *d* is the dimensions of the data (2 in this case) and *r* is the number of points returned by the query. The worst case performance is O(n) that happens at higher dimensions.[4] For this project the performance of the range method was not tested, however it was implemented as it is well known to take advantage of the KD tree implementation and can further be developed by the team members.

Another operation that our team implemented was the ability to create a balanced KD tree from a given list of points. Implementing this for a KD tree has the added benefit of being able to balance an unbalanced tree, by reading out the points of the tree into a list using the iterator and reconstructing a balanced version of the tree from this list. This is implemented via the generate() method in the KDFactory class in the repository which takes an array of points as input, and an alternative implementation is used in the balance() method of KDTree which instead uses an iterable list of points.

Our balancing algorithm works by recursively sorting a list along the alignment coordinate, putting the median as the root of the associated KD tree, the smaller points in the below subtree and the larger points in the above subtree. We used a convention that if a root had a child with the same coordinate, we would place that child in the above subtree, so that the recursive functions in the KD tree would know which direction to traverse. The MedianPolicy enum and findMedian() function in KDFactory allow this convention to be changed or reversed if necessary.

GUI Design

Our team decided to build a custom KD tree visualizer that enables users to easily interact and visualize with this spatial structure in a 2-d space. Our team decided to use Java's built in Swing GUI toolkit library since this enables our end user, the professor and TA's, to easily run our program without the need for additional external libraries. The GUI created by our team contains a user-friendly interface which allows the user to interact with the spatial data structure more easily. The visual composition was inspired by the classic retro neon theme.

[4] George T. Heineman, *Algorithms in a Nutshell*, (O’Reily Media, 2009), 292

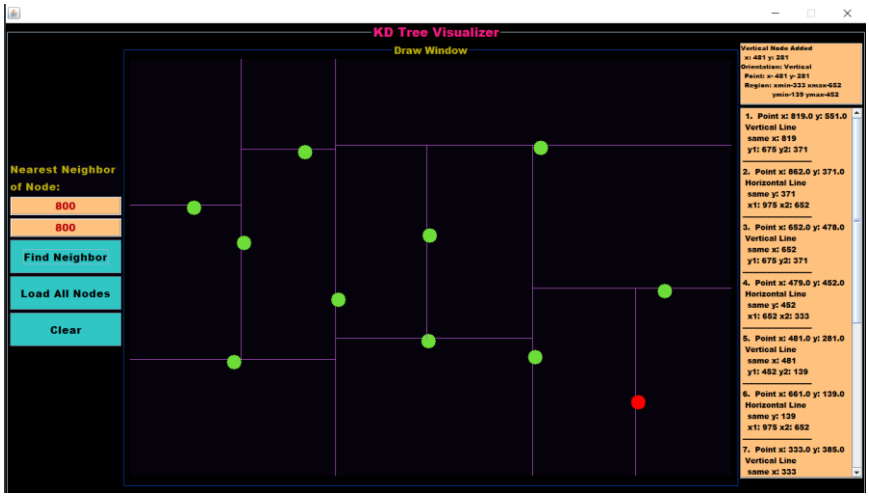

Figure 3: KD-Tree Visualization GUI

The KD-Tree GUI is structured by three panels, the control panel, interactive panel and information panel. The interactive panel label as "Draw Window" allows for the user to click on the 2D space to insert a KD node to the KD-Tree. This update the information displayed on the top right text box with the point and region added. The control panel on the left allows the user to input a point in 2-space to find the nearest neighbor in the KD-tree. When a point is added and the "Find Neighbor" button is clicked the nearest neighbor will be displayed by a red dot on the interactive panel. Furthermore, the control Panel also provides "Load All Nodes" and "Clear" buttons. The "Load All Nodes" button loads information pertaining to each KD node in the KD-tree. This information is displayed on the right most information panel. In the information panel each KD node is displayed with point and region information. The top text box displays the most currently added KD Node to the structure. Then when hit refresh the bottom text box displays ALL the nodes in the KD structure. It is important to note that the order of the KD nodes displayed is from left to right on the KD-tree and is not displayed by the order inserted into the KD-tree. The "clear" Button clears the graphics, text displays, and re-initializes the KD-tree.

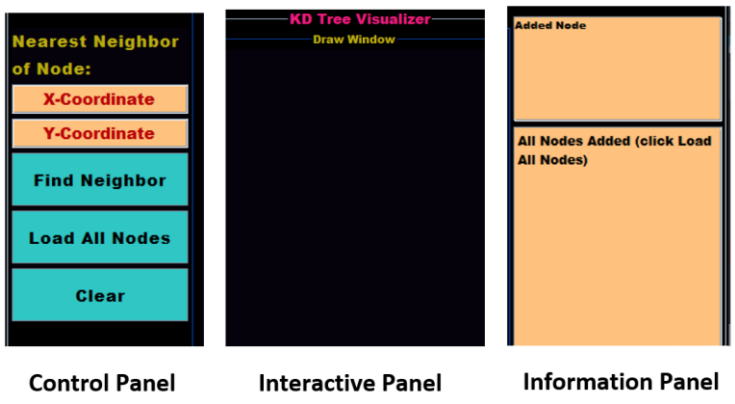

Figure 3: KD-Tree GUI Panels

Evaluation and Results

For the nearest neighbor query, the worst-case scenario would be where the radius of possible closer nodes intersects with all regions of the tree. An obvious arrangement for this is for the points in the k-d tree to outline a circle and have the point in question near the center. This will cause the algorithm to search every part of the tree to ensure that there are no points inside that circle but will never find one. Therefore, it ends up searching every node in the tree.

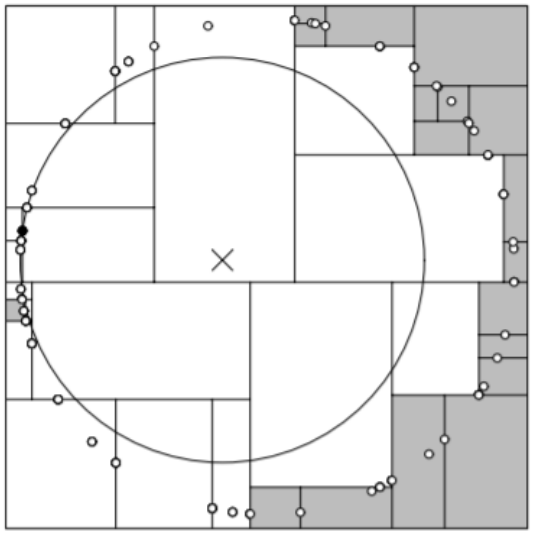

Figure 4: Nearest neighbor search worst case diagram

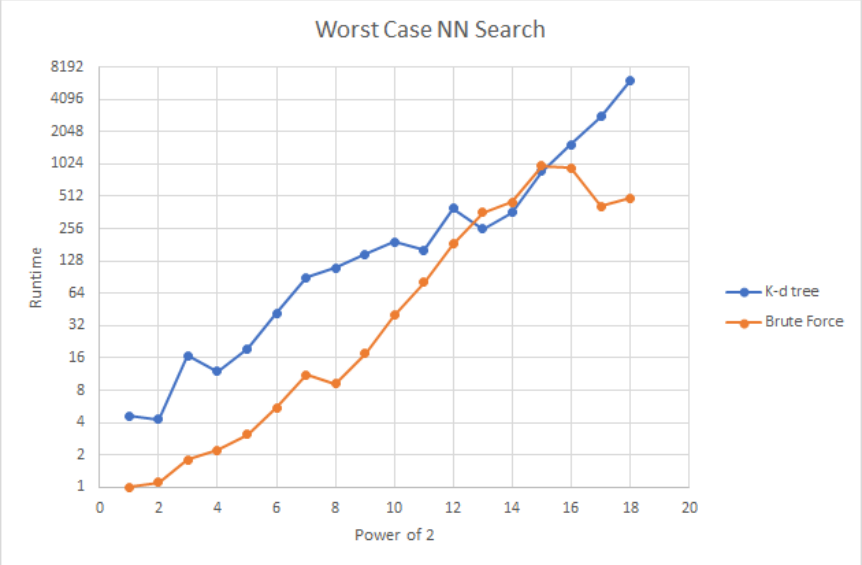

Figure 5: Nearest neighbor search of (0, 0) where the points were in a circle

We constructed a k-d tree for points in a circle (number of points was 2^x in the graph) and compared the nearest neighbor search to the brute force implementation. Although the data is a bit noisy, it shows that the implementations both have a linear runtime with the problem size.

When executing time evaluation tests, we tested the k-d tree nearest Neighbor method and the brute force algorithm to analyze the average, minimum and maximum times for multiple data set sizes. We tested each algorithm by running multiple trials, each trial comparing a randomized point against a data structure containing 2^n random points, where n was an integer value from 2 to 18. For every n value the same randomized points were added to a k-d tree and Point[], both of size 2^n, and each of these data structures were newly generated for every n value, not for every trial.

N	Avg	Min	Max	BruteAvg	BruteMin	BruteMax
2	0.358300	0.100000	21.400000	0.214000	0.100000	32.200000
3	0.172970	0.000000	15.400000	0.358000	0.100000	5.400000
4	0.193310	0.000000	4.600000	0.265000	0.100000	9.700000
5	0.207910	0.000000	6.400000	0.292000	0.000000	43.000000
6	0.365050	0.000000	6.300000	1.945000	0.700000	31.700000
7	0.455070	0.100000	9.700000	2.265000	1.400000	36.800000
8	0.508940	0.100000	11.900000	1.096000	0.400000	204.400000
9	0.555900	0.100000	4.300000	0.995000	0.800000	32.400000
10	0.648730	0.100000	3.600000	1.887000	1.700000	19.800000
11	0.740610	0.100000	32.500000	3.889000	3.600000	61.200000
12	0.962870	0.200000	67.000000	10.548000	8.400000	1016.000000
13	1.185830	0.300000	20.500000	22.594000	16.800000	104.800000
14	1.207200	0.300000	107.200000	43.049000	33.600000	114.100000
15	1.398800	0.300000	39.500000	82.263000	68.100000	272.500000
16	2.808820	0.600000	162.800000	334.268000	274.700000	1227.500000
17	4.980380	1.000000	25.800000	1013.800000	847.000000	1923.000000
18	4.575740	1.300000	28.800000	2506.507000	1925.700000	4625.400000

Figure 6: Time performance results of nearest-neighbor vs brute algorithms

As expected, the nearest neighbor query worked very efficiently, outperforming the brute force algorithm for most data sets. From the n value of 5 to the value 11, the time cost was only increasing by about 0.1, increasing progressively more after that value. Even though the delta was increasing, the time cost of the algorithm managed to stay very low, reaching a maximum of only about 5 microseconds. The efficiency of this algorithm is O(log(n)), which can be seen in the data as the number of elements gets exponentially bigger while the time cost increases on a much smaller scale.

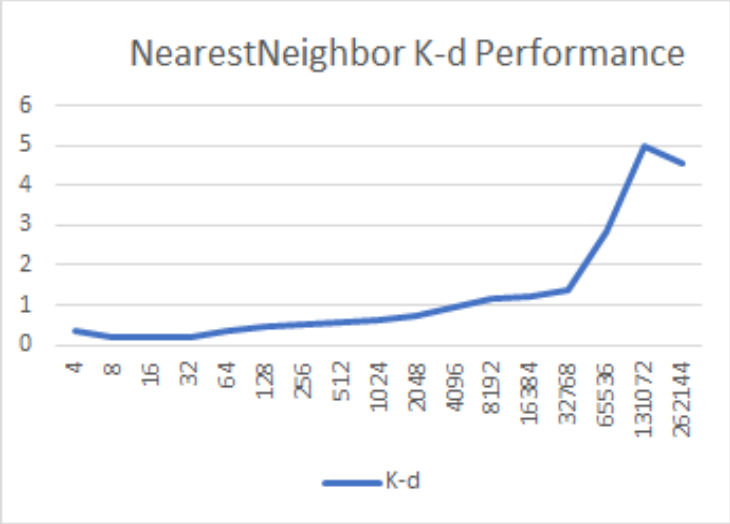

Figure 7: Nearest-neighbor KD-Tree performance

The brute force algorithm was obviously much slower, as it operates in O(n) time, hindering its performance almost immediately. What the algorithm essentially does is iterated through an array of 2^nelements, comparing the distance from the trial point to the point in the current cell, replacing it if it was shorter than the minimum.

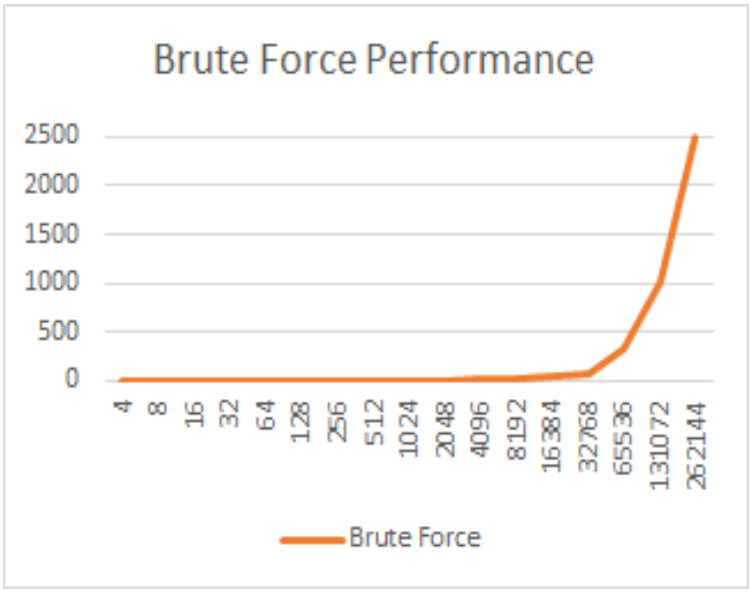

Figure 8: Brute force KD-Tree performance

Throughout every significant n value, the average time of the nearest neighbor query outperforms the average time of the brute force algorithm. The only value of n, where brute force has a superior time is 2, which is outside of the range$2^5$to 2^{18}. Although it is slower for almost all values of n, the brute force approach still competes with the nearest neighbor query for some of the n values. Until the n value 11, the time averages of the two algorithms are within 3

microseconds, and very similar until the time cost for brute force begins to spike and increase to a maximum average of about 2507 microseconds.

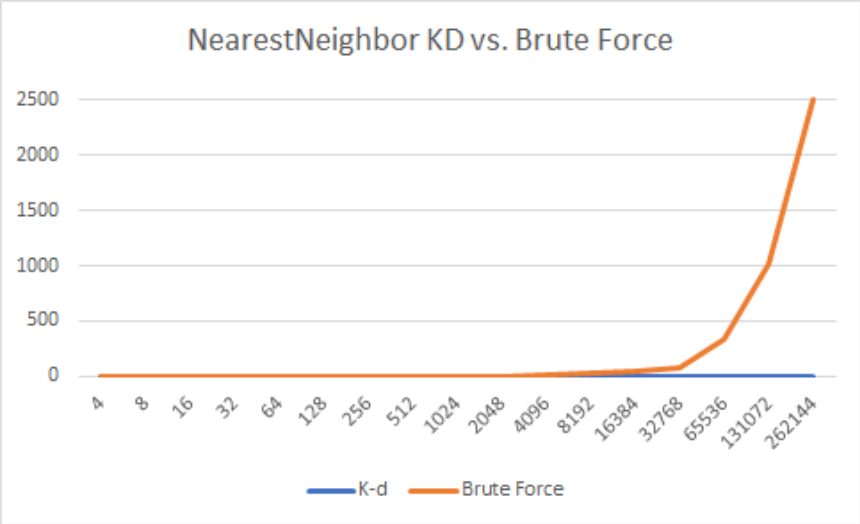

Figure 9: Nearest-neighbor vs. brute force KD-Tree performance