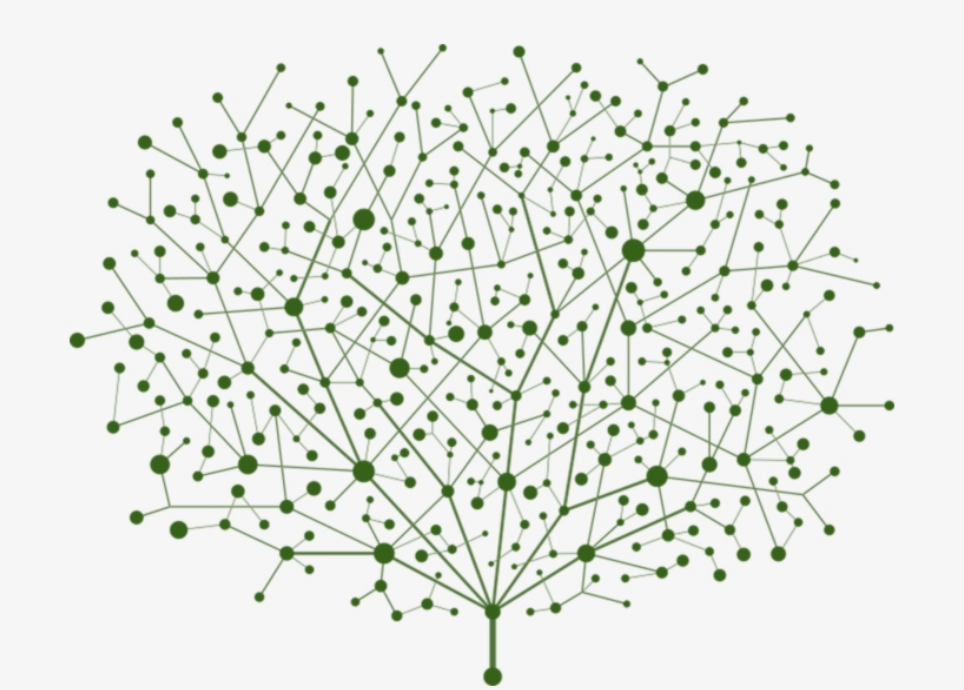
CSC 212: Data Structures and Abstractions

Templated Implementation of a K-D Tree Data Structure with Balancing

GitHub Repository: *github.com/raymondturrisi/CSC212-Final-Project*



Team Members:

Raymond Turrisi	Mechanical Egr.ing & Computer Science
Alfred Timperley	Computer & Data Science
Brennan Richards	Business & Computer Science

University of Rhode Island

Department of Computer Science and Statistics

Final Project Report

Fall 2020

Contents

List of Figures

List of Tables

1 Introduction

This team was tasked with developing an implementation of the K-D Tree data structure within C++. Other objectives for this project were to: design the code efficiently which takes advantage of the many tools available within C++, code cleanly so it can be easily communicated and worked on by other users, and apply theorems covered in this CSC 212 Data Structures and Algorithms course.

Theorems and principles covered in this class which were applied directly to this project and discussed in this paper are:

- Empirical analysis of algorithm performance: measuring run time and stressing the algorithm for increasing complexities and edge cases.
- Theoretical Asymptotic analysis of algorithm performance under several assumptions.
- Result visualization which took advantage of MATLAB ® and the *.dot* file format.
- Using Git and GitHub in order to collaboratively develop a large project.

With this, the team designed a fully templated K-D Tree class and sought to create relevant applications that take advantage of the key features that this data structure offers, answering questions to real world problems. The scope of these applications is limited to 2D geographical data, however the class was defined to work with K dimensions for enhanced functionality to be well suited for other applications. Furthermore, this paper will provide a broad overview of K-D Trees from their history and common applications today, theoretical asymptotic and empirical analysis of K-D Trees and the fundamental methods. From this, the methods implemented into this class will be discussed in *Methods*, where the algorithms will be explained in-depth using the C++ syntax. Next the authors will discuss the implementation of this K-D Tree class and how it used in this project: from obtaining real data with Python and the Yelp API, and the various techniques for visualizing and communicating the team's findings. The paper will be wrapped up with additional considerations for the teams project, a conclusion, followed by the contributions of each member.

1.1 On K-D Trees

The K-D tree is a multidimensional BST which was first created in 1975 by Dr. Jon Louis Bentley, a current professor at Carnegie Mellon University's Computer Science and Mathematics Department [1]. Per Dr. Bentley's thesis, the K-D Tree was developed during an investigation of a multidimensional divide-and-conquer technique which decomposes a problem with n points within K dimensions [2]. A key feature of this data structure is that it allows the efficient tree traversal for an algorithm which finds an accurate approximated nearest neighbor to a node or a comparable location in $\Theta(log_2 n)$.

K-D trees are simple in nature, and are highly efficient for processing multi-dimensional keys. While they are very similar to a Binary Search Tree, there are several distinguishing features in their construction, insertion, and traversal. First, K-D trees make branching decisions based on a condition and the level of the node with respect to the root node, and this is done by tracking its depth, d, in a tree, and a *discriminator*, i [3]. Recursive functions identify the *type* of level they are on with this discriminator, where:

$$i = d\%K, \quad i \in [0, K] \tag{1}$$

For example, take two dimensional data represented by traditional x and y, with discriminators 0 and 1 respectively, across three levels with data:

$$\{(293, 267), (271, 98), (372, 260), (337, 156)\}$$

The steps for insertion would proceed as follows:

- Insert node (293,267) on depth $d = 0$ with iterator $i = 0$ as *root*.
- Insert node (271,98) by checking the dimension belonging to the current iterator, such as on $i = 0$, compare $K = x$, where this node will go to the left of the tested node.
- Insert node (372,260) by checking the dimension belonging to the current iterator, such as on $i = 0$, compare $K = y$, where this node will go to the right of the tested node (271,98).
- Insert (337,156) by checking the dimension belonging to the current iterator, such as on $i = 0$, compare $K = x$, where this node will go to the right of the tested node (271,98) being unto testing against node (372,260). Now, $i = 1$ which belongs to the second dimension y, where $156 < 260$, where now node (337,156) will be the left child of (372,260).

With this, a 2-D tree with four nodes has been created as seen in Fig. 4 below. Even with so few points, it provides important insight on how the K-D tree achieves such high performance, with the partitioning from this example demonstrated in Fig. 2. Such that as the first node inserted has nodes to both the left and right, however when looking for points to the right of the first node, everything to the left is neglected. On the next discriminator where the Y-plane gets partitioned at $y = 260$, all the nodes that would be above it are no longer considered in the search or query for inserts, searches, and the approximated method for nearest neighbors if looking for a point which is less than this nodes discriminated value. This partitioning in space is where the average run times for these fundamental methods are $\Theta(log_2 n)$, and how later when balancing the K-D Tree is discussed, it would be best to find the median for each partition and insert to effectively achieve ideal half plane partitioning.

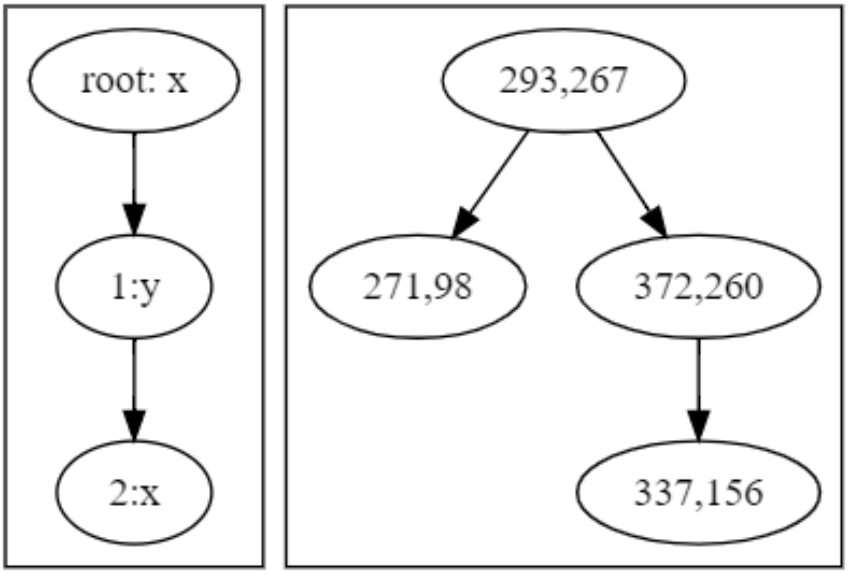


Figure 1: 2 dimensional K-D Tree from Example 1.

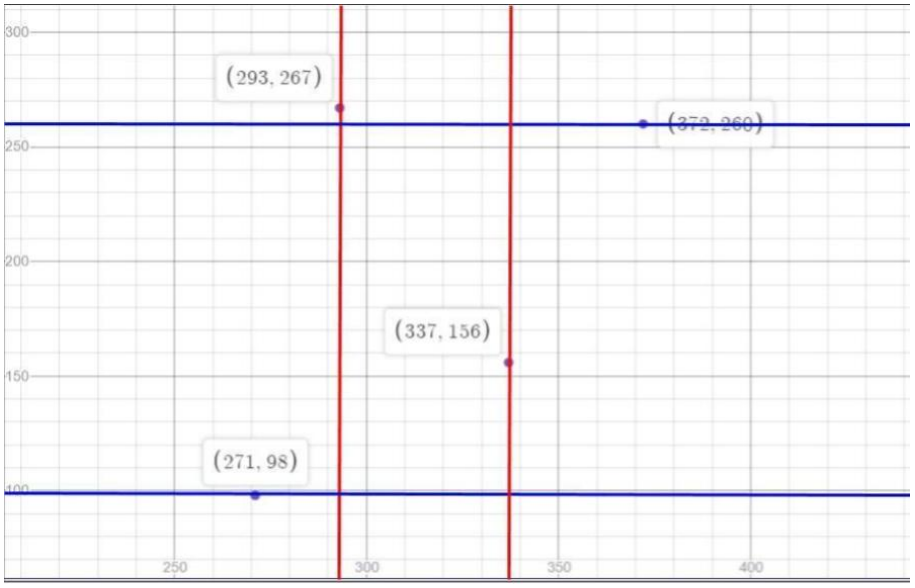


Figure 2: Coordinates on the Cartesian plane with partitioning at each insert in the graphical form.

1.2 General Applications of K-D Trees

The most common use for K-D trees are for solving nearest-neighbor searches with respect to neighboring nodes, relative coordinates, or nearest-neighbor queries returning all nodes within a 'radius' of an existing node/coordinates of k-dimensions or all nodes within a region bounded by existing nodes. However, there are many creative deployments of K-D Trees which have been compiled from several sources and briefly described below:

- The N-Body Problem: By partitioning space in a 3-D space with N-Bodies one can more easily simulate the effects of gravity on many bodies acting on each other [4].
- Color Reduction: To more efficiently determine the best color out of 256 bit range to reduce the available colors in an image [4].
- Searching for all elements within a radius, or returning n number of nearest elements, which can be modified by requesting other information at the node, such as in the implementation described later, nodes with data of a type *"type"*.
- And many more, with several examples local to this teams implementation described later.

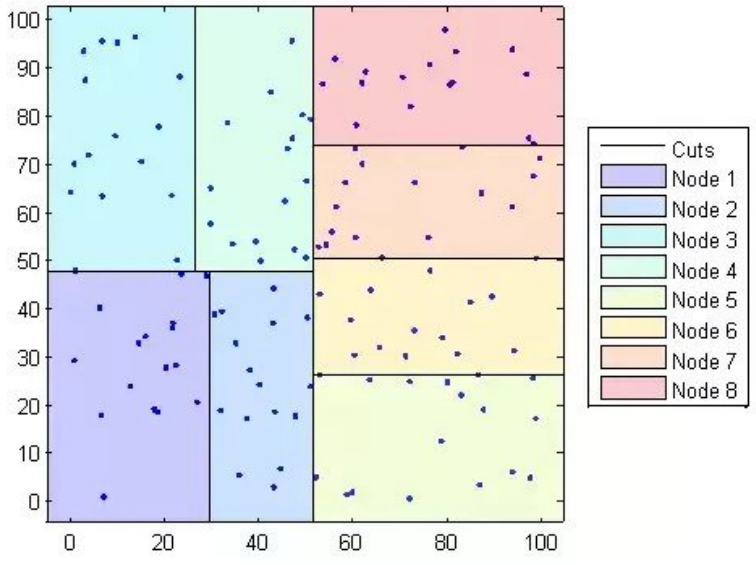


Figure 3: 2-D Tree query request for returning information with in a region.

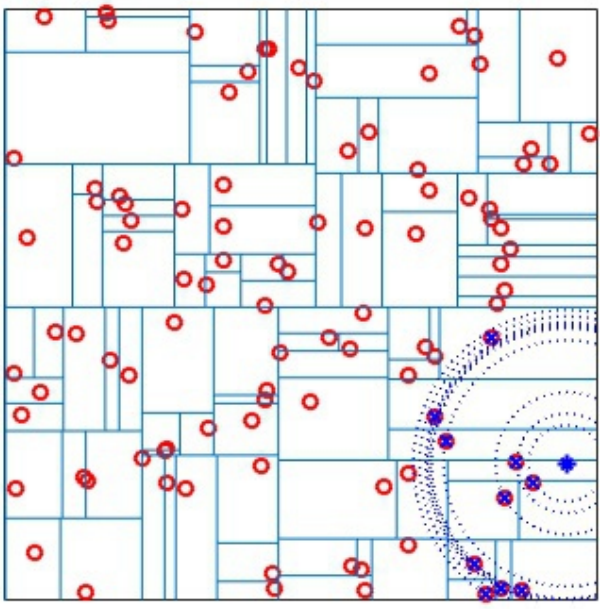


Figure 4: Depiction of points in a 2-D space querying for N number of nearest neighbors with respect to a base coordinate, which here can be seen at the bottom of right.

1.3 Relations to this Project's Implementation

The culmination of this project manifests itself with our real world application description at the end of the paper. However in brief, the final product was a fully templated K-Dimensional Tree class making as few assumptions about the input data as possible. Several features included

in this project are a balancing tree constructor, inserts, searches, and nearest neighbors with overload function calls providing more options for the user. There was an analysis of this project's implementation performance, in addition to various ways to verify the validity of the construction and visualize the data. All of which, are explained further into this paper.

2 Theory

This section will detail the asymptotic complexity of the functions used in the *k-d* tree. The following table showcases the time and space complexity of the Standard functions of the *k-d* Tree.

Table 1: Time and Space Complexity for Standard Functions

Algorithm	**Average**	**Worst Case**
Space	$\Theta(n)$	$\Theta(n)$
Search	$\Theta(log_2 n)$	$\Theta(n)$
Insert	$\Theta(log_2 n)$	$\Theta(n)$
Delete	$\Theta(log_2 n)$	$\Theta(n)$

There are three designs for building a static *k-d* tree from n points with the following complexities.

- $\Theta(n\, log_2^2 n)$ if a $\Theta(n\, log_2 n)$ sort is used to find the median at each level of the tree, such as Heapsort or Mergesort.
- $\Theta(n\, log_2 n)$ if a $\Theta(n)$ sort is used to find the median at each level of the tree, such as the medians of medians algorithm
- $\Theta(k\, n\, log_2 n)$ if n points are presorted by each of the k-dimensions using a $\Theta(n\, log_2 n)$ sort, such as Heapsort of Mergesort, prior to building the *k-d* tree

This team decided to build their *k-d* tree with the medians of medians algorithm design as this method offers the best complexity. The medians of medians algorithm is detailed in section 3.1.5.The recursive relationship for the Median of Medians Algorithm is $T(n) = cn/5 + T(n/5) + cn + T(7n/10)$, which simplifies down to $T(n) = n$.

Using the median of medians algorithm and building the tree from a list of locations allows the tree to be balanced. A balanced trees time complexity for the Standard Function and other functions will be better than an unbalanced tree's. This can bee seen in Table 1, as a tree gets more unbalanced it's time complexity gets closer to the worst case. There is an extra cost in time complexity to building a balanced tree but if the tree is intended to be used many times the gained efficiency in the functions will quickly outweigh the initial cost.

3 Methods

This implementation of a K-D Tree class came with many useful methods, providing full functionality of a K-D Tree class which has dimensional data in an member called *coords* that inherits the *K-D Tree Key* sub class which provides the source of the dimensional data. In addition to the fundamental methods of a K-D Tree, there were several additional methods included which were used in the *DefaultLocation* class, which can still be easily modified to provide the same benefits for other applications. See below for the full list of public methods

which have been implemented into this class, and can be accessed as a member of the KDTree object.

KD Tree:
template <class t>

- Constructors:
  - Empty:
    KDTree< *t* >(unsigned int _dimensions)
  - Single Location:
    KDTree< *t* >(t data)
  - Balancing Constructor:
    KDTree< *t* >(std::vector<t*> vector_of_structs, unsigned int _dimensions)
- Destructors
  - Recursive tree destruction:
    void **destroy**()
- Insertion:
  - Single Location:
    void **insert**(t data)
- Search:
  - bool **search_for_element**(t data)
- Nearest Neighbor, with several overloaded calls, accepting arguments as:
  - With respect to another object of type t:
    t **nearest_neighbor**(t data)
  - With respect to another object of type t, returning by reference an optional distance argument:
    t **nearest_neighbor**(t data, double &best_distance)
  - With respect to another object of type t equal to a particular *type**:
    t **nearest_neighbor_oftype**(t data, std::string wanted_type)
  - With respect to another object of type t equal to a particular *type**, returning by reference an optional distance argument:
    t **nearest_neighbor_oftype(** t data, std::string wanted_type, double &best_distance)
  - With respect to a set of coordinates of with K dimensions:
    t **nearest_neighbor**(std::vector< double> local_coords)
  - With respect to a set of coordinates of with K dimensions, returning by reference an optional distance argument:
    t **nearest_neighbor**(std::vector< double> local_coords, double &best_distance)
  - With respect to a set of coordinates of with K dimensions equal to a particular *type**:
    t **nearest_neighbor_oftype**(std::vector< double> local_coords, std::string wanted_type)
  - With respect to a set of coordinates of with K dimensions equal to a particular *type**, returning by reference an optional distance argument:

t **nearest_neighbor_oftype**(std::vector< double> local_coords, std::string wanted_type, double &best_distance)

- Printing / Output:
  - Post Order Printing:
    void **postOrder**()
  - In Order Printing:
    void **inOrder**()
  - Pre-Order Printing:
    void **preOrder**()
  - File output in Dot File Format:
    void **dotOut**()
- Further, there are in general a private recursive function paired with these public top level function calls. This was done in order to preserve user friendliness of the class.
- *type**: This type is with respect to the member *type* within the Default Location class. This search method can be modified for other uses as it is only one boolean in the Nearest Neighbors algorithm, and the member passed can remained a string called *wanted_type*, or match a member of a any other class.

To allow for the construction of a balanced tree the K-D Tree class also makes use of the Medians of Medians algorithm which is conveniently packaged into another templated class. See below for the full list of methods which have been implemented into the Median of Medians class, however they are abbreviated as they are private.

Median of Medians:

- Constructors
  - Empty constructor
  - With list and dimension to operate on (primary constructor described in previous list)
- Select index
- Pivot about index
- Sort a partition of the vector
- Reset the current partition

3.1 Algorithms and performance

Herein this section we provide the fundamental algorithms implemented into this teams project. We will first discuss the fundamental algorithms to a K-D Tree, and then we will go into the technique for developing a balanced K-D Tree with the Median of Medians algorithm, which as explained before, is regarded as one of the most efficient ways for building a balanced K-D Tree. Both of which are explained in pseudocode, where with more explicit definitions commented within the submitted source code.

3.1.1 Insertion

There are two methods of insertion included in this *K-D* Tree and they are as follows.

- $\Theta(log_2 n)$ if a unbalanced tree is constructed from inserting a single location at a time.
- $\Theta(nlog_2 n)$ if a balanced tree is constructed from inserting a list of locations.

The unbalanced tree is constructed in traditional binary tree fashion. The value to be inserted is compared against the current root of the tree. If it is less than it enters into the left child node and if it is greater it enters into the right child of the node. This process continues recursively until the node it is comparing against does not exist and in that case a new node is created with the value to be inserted.

The balanced tree can only be constructed from a list of locations, and cannot guarantee the tree will stay balanced if further insertions commence. To create a balanced tree each node has to be assigned to the median value of the list of locations that have not been inserted yet. Since this is a K-D tree, the current depth has to also be known to ensure that the algorithm is comparing based on the correct dimension. To find the median of the list of locations the K-D tree uses the Median of Medians algorithm (explained in Section 3.1.5) to find the median in $\Theta(log_2 n)$ time. The Median of Medians algorithm also ensures that the list of locations has been re-ordered so that all values to the left of the median are less than the median and all values to the right of the median are greater than the median. This can be used to split the list into a left half and right half and then recursively pass each through the function again. Thus the tree is created with each node being the median of the proper dimension for all of the children below it. Below is the pseudo code for the basic insert, with an in depth description for the Median of Medians algorithm and its respective functions described below.

Algorithm 1 Insertions into a K-D Tree

```
Insert_r(class data, unsigned int depth_r, tree_node node)

  if node == NULL then
    node = new Node(data)
  end if
  i = depth_r%K
  if data.coords[i] < node.coords[i] then
    Insert_r(data,depth_r+1,node.left_child)
  else
    Insert_r(data,depth_r+1,node.right_child)
  end if
```

3.1.2 Search

Searching follows the same procedure as insertions, guaranteeing thatK once a tree is constructed, the element would have traversed the tree in the same order if it exists, which eliminates the need to check all the trees in a space of K dimensions. Where these functions differ is it returns a boolean if it reaches a leaf node or if it locates an equal member in the tree.

Algorithm 2 Searching a K-D Tree

Search(class data, unsigned int depth_r, tree_node node)

```
if node == NULL then
  return false
end if
if node.data == data then
  return true
end if
i = depth_r%K
if data.coords[i] < node.coords[i] then
  Search(data,depth_r+1,node.left_child)
else
  Search(data,depth_r+1,node.right_child)
end if
```

3.1.3 Nearest Neighbors

While looking slightly longer, this algorithm does the same thing as insertion and search, while recording the best distance between all recursive calls and the associated member as *result*. The pseudocode below can be simplified for making sense on paper, however it was transcribed from the provided source code to be more cohesive - where it was implemented as follows to minimize errors which were found in the original implementation.

Algorithm 3 Nearest neighbors, approximated

```
Nearest_neighbor(tree_node node, class data, class &result, unsigned int depth_r, double
&best_dist)

i = depth_r%K
distance = distance_between(node.data, data)
if node.data.coords < data.coords then
  if node.right_child exists then
    if best_dist > dist and node.data != data then
      best_dist d̄istance
      result n̄ode.data
      return
      Nearest_neighbor(node.left_child, data, result, depth_r+1, best_dist)
    end if
  end if
else
  if node.right_child exists then
    if best_dist > dist and node.data != data then
      best_dist = distance
      result = node.data
      return
      Nearest_neighbor(node.left_child, data, result, depth_r+1, best_dist)
    end if
  end if
end if
if best_dist > dist and node.data != data then
  best_dist = dist
  result = node.data
end if
```

3.1.4 Destroy Tree

To destroy the tree, it simply recursively traverses to all the leaf nodes, and upon returning from the base case proceeds to delete both of it's children nodes (truly tragic), and repeating until it reaches the root of the tree.

Algorithm 4 Destroy K-D tree

```
Destroy(tree_node node)

if node == NULL then
  return
end if
Destroy(node.left_child)
Destroy(node.right_child)
delete node.left_child
delete node.left_child
if node == tree.head then
  delete tree.head
end if
```

3.1.5 Medians of Medians Class

The following is the pseudo-code of the Median of Medians class with an in depth explanation of the algorithms mechanics following.

Algorithm 5 Select Potential Median

```
select(left, right, target)
  if right - left <= 5 then
    sort(left,right)
    medianIdx = target
    return list[target]
  end if
  swapIdx = left
  for each 5 elements from left to right do
    sort(left,right)
    swap the now 3rd element to the swapIdx-th position
    swapIdx++
  end for
  newTarget = left + floor((swapIdx-left)/2)
  median = select(left, swapIdx, newTarget)
  location = pivot(median, left, right)
  if location == target then
    return list[target]
  end if
  if target > newTarget then
    return select(newTarget+1, right, target)
  end if
  return select(left, newTarget, target)
```

Algorithm 6 Pivot List Around Potential Median

```
pivot(val, left, right)
  swap(location of median with right most index)
  j = left-1
  for each element from left to right do
    if list[i] < val then
      j++
      swap(list[i], list[j])
    end if
  end for
  j++;
  swap(list[j], list[right])
  return j
```

Algorithm 7 Insertion Sort

```
insertion sort(left, right)
  i = left + 1
  while  i <= right do
    j = 1
    while list[j-1] > list[j] do
      swap(list[j],list[j-1])
    end while
    i++
  end while
```

The Constructor of the Median of Medians Class takes a list and the dimensions it is first operating on as an input. The Class also has a method called reset which resets the local values of the Median of Medians Class with a new list and new dimensions to be operated on. This essentially reconstructs the class to be used for other lists and other dimensions.

The Medians of Medians algorithm first takes the left bound (in the beginning it is most commonly zero), the right bound (in the beginning it is most commonly size of list minus one), and a target index input into the select function. The algorithm will then split the list into sub-lists of five elements each. If there are five or fewer elements in the list the algorithm will preform an insertion sort and return out the target index. However if there are more than five elements to be operated on the algorithm will continue. Each sub-list will be sorted via the insertion sort and the median of each sub-list is swapped to the front of the list, starting with index zero and will be incremented by one for every sub-list. The algorithm then recursively calls itself to find the median of the compiled medians collected in the front of the list, hence the name median of medians. This is done without copying lists as the bounds of the list (the left bond input and right bound input) are simply changed when calling the algorithm recursively to ensure it is only operating on the desired values. The target index is also recomputed to be the median of the values to be operated on. This is computed with the floor() method of c++ so the median will be the right of center value in an even list. This is due to the fact that we cannot average the two center values as then the value returned would not actually be contained in the list and thus cannot be used to build the tree. So, a choice to select the value to the left of center or right of center must be made for lists of an even length and in this algorithm we decided to choose a right of center variation.

{10, 11, 12, 13}

As an example, the above list would return 12 as the median value. This process will continue until the bounds of the list recursively being operated on is five elements or less in size, and as mentioned earlier, when there are five or fewer elements being operated on the algorithm will preform an insertion sort and return out the target index. Now that the value of the trial median is selected, the algorithm must ensure that the trial median is indeed the median of the original inputted list. To do this the algorithm preforms a pivot on the list by placing the trial median in it's correct index relative to the other values in the list. If the index of the trial median is equal to the target index desired the method will return that value. If the trial median is not correct then the function will recursively call itself again, however this time the left bound and right bound will be changed. A feature of the pivot algorithm is that is has ensured that every value less than the trial median is to the left of the trial median and that every value to the right of the trial median is to the right of the trial median as it has placed the trial median in it's correct spot. With this behavior you can test whether or not trial medians index is less than or greater than the target index and then only continue to operate on that section of the list, as you know the median can not be in the other section. This process continues until the trial median's index is equal to the target index or the 'splitting' of the list yields a section where there are 5 or less elements to be operated on. Then, as mentioned before, the elements would be sorted and median returned.

3.2 Comments On, and Analysis of Methods

3.2.1 Insertions & Searches

Proper empirical analysis of algorithms must be stringent - where the attributes device must be well defined and documented in order to correctly correlate the time domain and the operative domain which is directly synonymous to the qualities of algorithm rather than limitations of the hardware. Here, a simple comparative analysis was performed in order to determine the *general* performance of the algorithm. These next two figures were ran in the CS50 IDE running on an EC2 instance which has poor performance compared to what is available on most modern computers.

First presented is the unbalanced insertions seen in Fig. 5. Within this plot, the timed performance of the insertions algorithm is plotted with respect to the number of elements inserted, however this was then performed 100 times, and at the end of each round of insertions, the tree was destroyed, the input vector was shuffled in a random order, and then the process repeats for a total of 100 cycles, where this scatter plot actually contains roughly 1,100,000 insertions. This was done in order to try to obtain the regions for which the algorithm is both upper and lower bound, $\Theta(n)$ and $\Theta(log_2 n)$, and produce a reasonable scale for the the team can confidently assume that both the upper and lower bound have been encapsulated within the data and thus the timed performance of the computer can be neglected and the operations can be more strongly correlated to the number of insertions. This similar procedure was followed for searching in the K-D Tree, seen in Fig. 6.

Here, one takes an existing tree and and searches for a random element in both an unbalanced tree and a balanced tree in order to compare the performance. The unbalanced tree was not repeatedly shuffled and reconstructed as a random element in the initial vector was being searched for and the K-D Tree within our data after processing stochastic data had more often than not remain relatively balanced and never truly reached the worst case scenario.

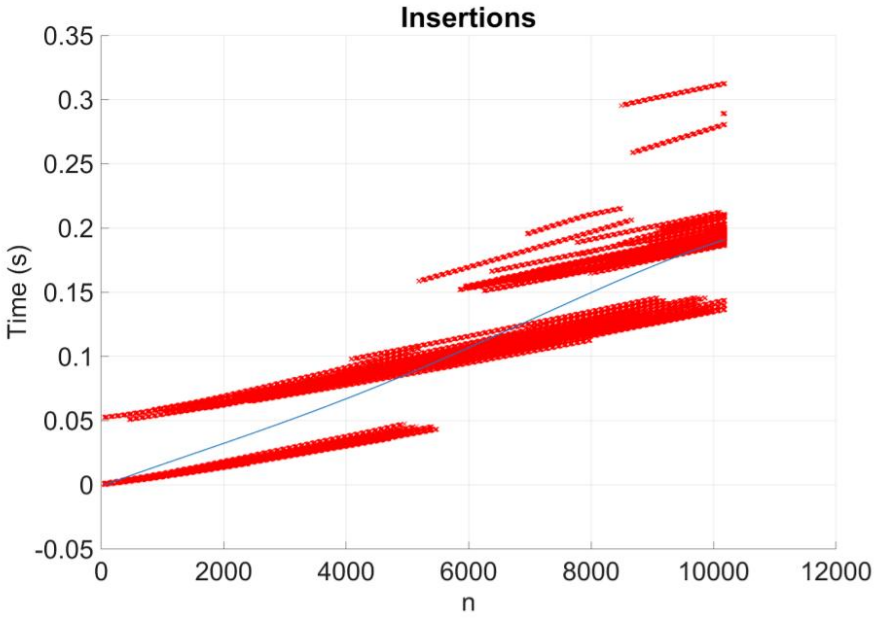


Figure 5: 100 insertions of 10,180 elements into a 2-D K-D Tree.

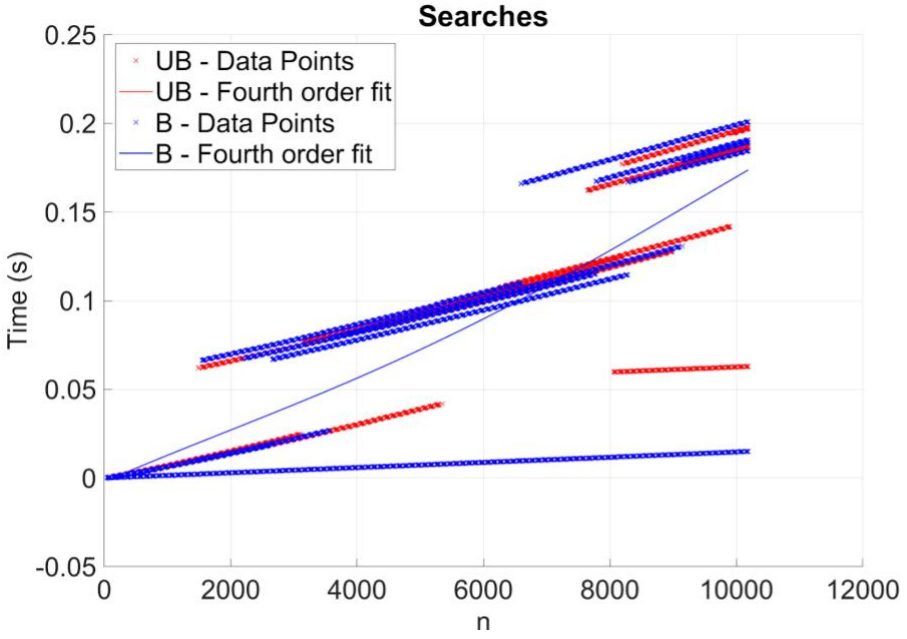


Figure 6: 10 trials of searches over a tree of 10,180 elements on both a balanced and unbalanced tree. Seen in this figure, the balanced K-D Tree set the record low for all the trials with incredibly efficient run times.

3.2.2 Nearest Neighbors

For the nearest neighbor algorithm, it became less interesting to measure the timed-domain performance of searches due to the uncertainties previously described, however with this algorithm there were other creative implementations which allowed the team to 1) verify the algorithm was working properly, and 2) depict unique qualities of this algorithm and the cost

of performing in $\Theta(log_2 n)$ time rather than inspecting all the elements in the tree. Nearest Neighbor divides the dimensional planes spatially in order to achieve a highly efficient run time, however this comes at the cost of it occasionally not returning the true nearest neighbor, as seen in Fig. 21. Here, there are several instances when the algorithm returns different outputs for how the dimensions are split at each level of the tree. This leads to an approximation of nearest neighbors, with subtle differences between both the left and right plot seen in Fig. 21, where depending on the application this approximation may be in general sufficient due to the drastic differences in run times between this technique, with a run time of $\Theta(log_2 n)$, and checking every element of a tree, $\Theta(n)$. For communicative purposes and the quality of the development of the project, both methods were accounted for and added, with the true nearest neighbor algorithm being implemented in the *nearest_neighbors_best()*. This method follows a post order convention (arbitrary) and checks every node in the tree, with the results of the algorithm seen in Fig. 8.

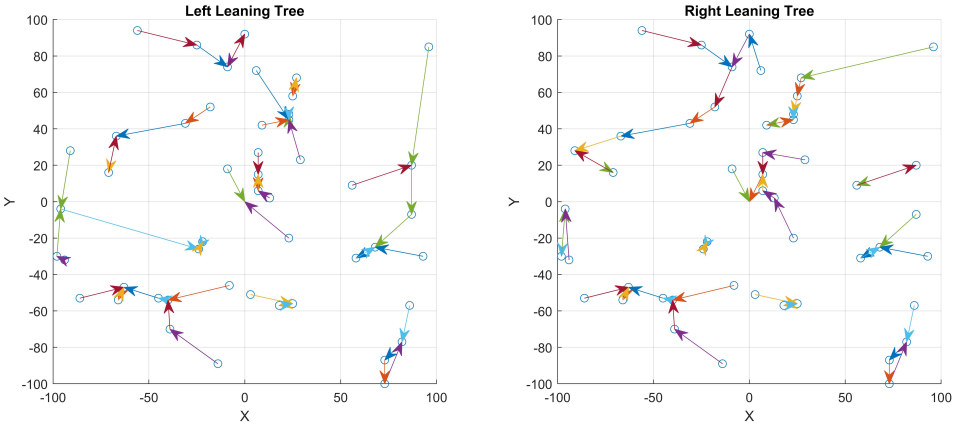


Figure 7: Left and right leaning implementation of the K-D Tree using the same nearest neighbors function.

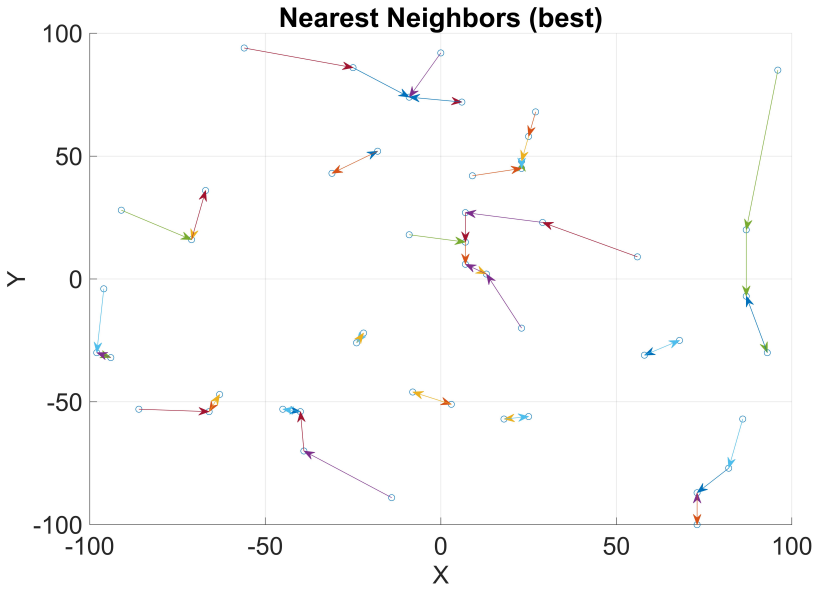


Figure 8: Results from nearest_neighbors_best, which produces a result indifferent of left and right leaning biases.

Other creative tests for nearest neighbors will be discussed in the presentation and available on the GitHub, however not explored further in this paper. The team did explore visualization of the nearest neighbor algorithm within three dimensional spaces, in addition to moving bodies in three dimensional spaces to explore repeatability within across small changes, in addition to qualitatively ensure the algorithm was working properly.

4 Implementation

In the provided code and as hinted at throughout this paper, the class was fully templated in order to be readily adopted by and expanded into other applications. This was done because in the research process, there were the case studies on K-D Trees and their theory, yet there were no libraries or implementation which was readily available which we could compare our results to. The team figured that it could be beneficial to someone who had similar questions to us, and would like to use our code easily in their application without many modifications. Likewise, the only assumptions that this code uses for the fundamental methods, is that it has a standard template vector containing K dimensions with respective coordinates as type *double*, where within the class this is referred to as "coords", which can be added to the original class, or inherited from the KDTree_Key class and added with one additional function call. Throughout the research process, the more flexible K-D Trees had taken advantage of storing dimensional data in vectors due to their ability to store all K dimensions, work nicely with determining the discriminator and accessing the i^{th} element. With this it was taken as a sufficient standard and the assertion was made that this is how this class will be constructed in order to meet all the objectives set forth by the team from the onset of the project. The diagram below in Fig. 9 depicts how the teams code was implemented in order to be as flexible as possible.

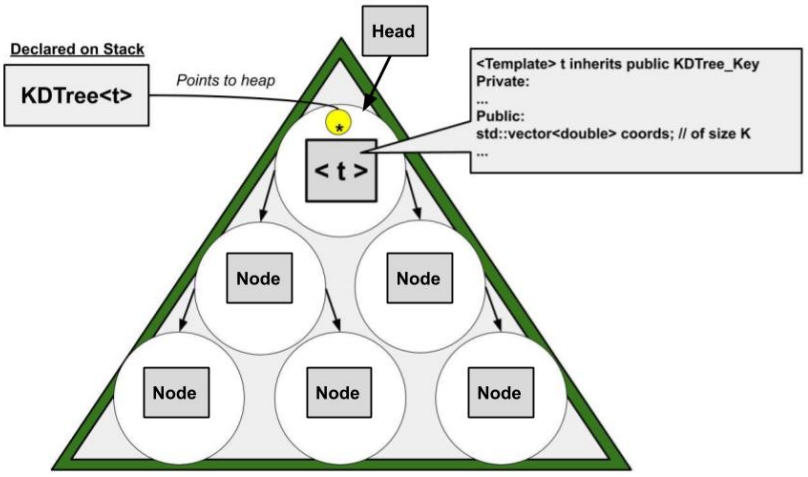


Figure 9: Diagram for how the tree is constructed between the stack and heap.

5 Application

The team decided to build this k-dimensional tree implementation with a specific application in mind. Because a properly implemented K-Dimensional tree is extremely well-suited to search for data points in physical space, with a balanced tree like our own being capable of finding the nearest neighbor in $\Theta(log_2 n)$ time, so the team solely focused on building an application of this kind. After discussion, it was concluded that the k-dimensional tree implementation would be used to store location data on real businesses, and to provide an interface for a human user to find the nearest location of some type to their current location.

The team decided to gather and use actual location data for this application. For a data source, this team turned to *Yelp*, a privately owned company which provides an online interface mostly for writing and reading reviews about local businesses and organizations. Yelp provides a suite of application programming interfaces (APIs) for extracting business/location data from the Yelp system using HTTP on the web. The Yelp API is provided open-source to anyone with a Yelp account and the ability and desire to use the service.

To extract the data, the team used Python and specifically libraries such as: *requests* for making HTTP requests, *pandas* for reading, outputting and cleaning data and *numpy* for a handful of statistical calculations. The particular Yelp API that the team used takes in search parameters and returns a list of data on the locations that fall within the search parameters passed. Using this method the team was able to collect rich data on approximately 10,000 businesses and organizations across Rhode Island, Connecticut, and Massachusetts. After retrieving the data, the team parsed the JSON objects which were received into comma separated strings of the important attributes before storing those lines all together in a *.csv* file, so that it would be possible to easily parse and use the data in C++ code and with various other tools. Figure 6 below shows the geographical spread of the locations in the data set as well as the density, which is displayed using the scale to the right, where the density increases as we climb up the color scale. One easily recognizable and interesting feature of the data set collected is that it is highly biased to be concentrated around the cities of Providence, Boston, and Hartford.

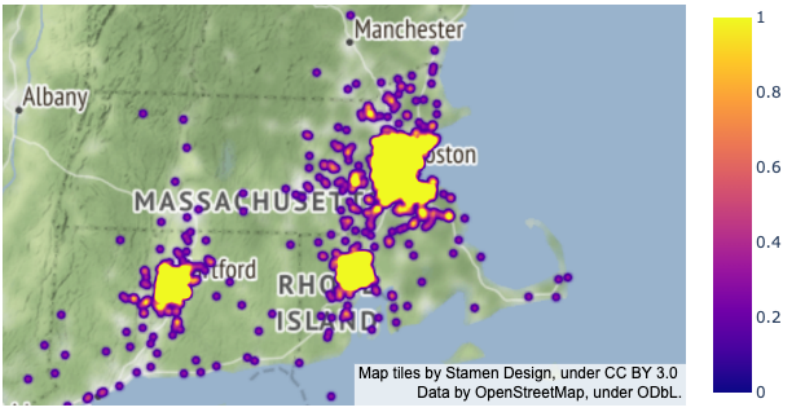


Figure 10: Geographical spread and density of the location data we retrieved from Yelp.

The retrieval of longitude and latitude attributes for each location object is essential to the application mentioned. The longitude attribute serves as the first dimension of our tree, and the latitude attribute serves as the second dimension for the 2-dimensional tree data structure, which for the data set of 10,180 entries, can be prepared in memory in a fraction of a second (less than half a 0.33 seconds, worst case).

During the location data retrieval process, it was necessary to store a qualifier which was commonly referred to among group members as the "type" of each location. For example some of the possible location types are: 'coffee', 'bank', or 'towing'. These location types have an English meaning of *places that sell coffee*, *banks*, and *businesses that provide a vehicle towing service* respectively. This type attribute is important to our application because although the user is able to find the nearest location based on some longitude and latitude input, which is possible due to the principles of k-dimensional trees, the nearest location is not usually a useful thing to know, for example, if the user's car has run out of gas for which reason they desire to be towed, and the application returns to this person the nearest coffee shop. Instead, users of our application pass a longitude, a latitude and a choice for the type of location that they want to search for as input, and are returned the nearest location of their desired type.

Figure 11 shows a view of the command line application after running the executable file. A menu can be seen displaying an English interpretation of the "type" qualifier attribute mentioned in the previous paragraph. The user should enter the location type they wish to search for by entering one of the numbers to the left of each choice. This number is then transformed into the actual string used throughout our code.

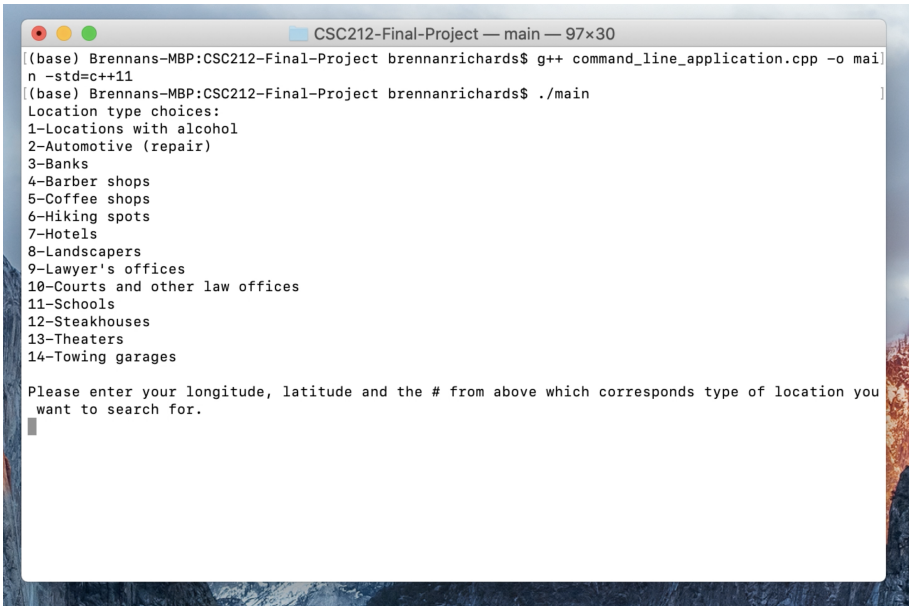


Figure 11: View of our command line application before entering the required input.

After the user enters input according to their use case, the nearest neighbor algorithm is used to find the closest location to the input longitude and latitude which is of the desired type, and displays all of the information available for that location. Fig. 12 shows the resulting call to the nearest neighbor function using the coordinates of an arbitrary point in Providence, Rhode Island, and the number 5 is entered in order to search for the nearest coffee shop. At this point a user can visit the location (after all in the ideal situation it is very close in distance), use the phone number provided to contact the location, or find other means of contact by visiting the Yelp URL.

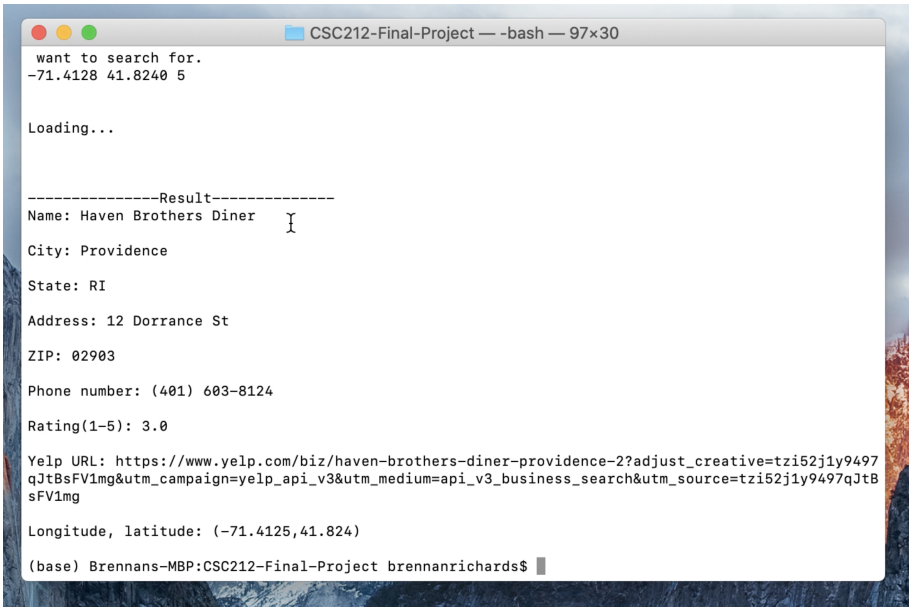


Figure 12: Command line application after input is entered.

A demonstration which the team found useful throughout the process of developing this application was to map both the input coordinates and the coordinates of the resulting nearest neighbor on Google Maps. The closer the input coordinates are to the nearest neighbor, the more favorably the application can be deemed to have performed on the input case. Fig. 13 shows the mapped results of the demonstration in Fig. 11 and Fig. 12 above.

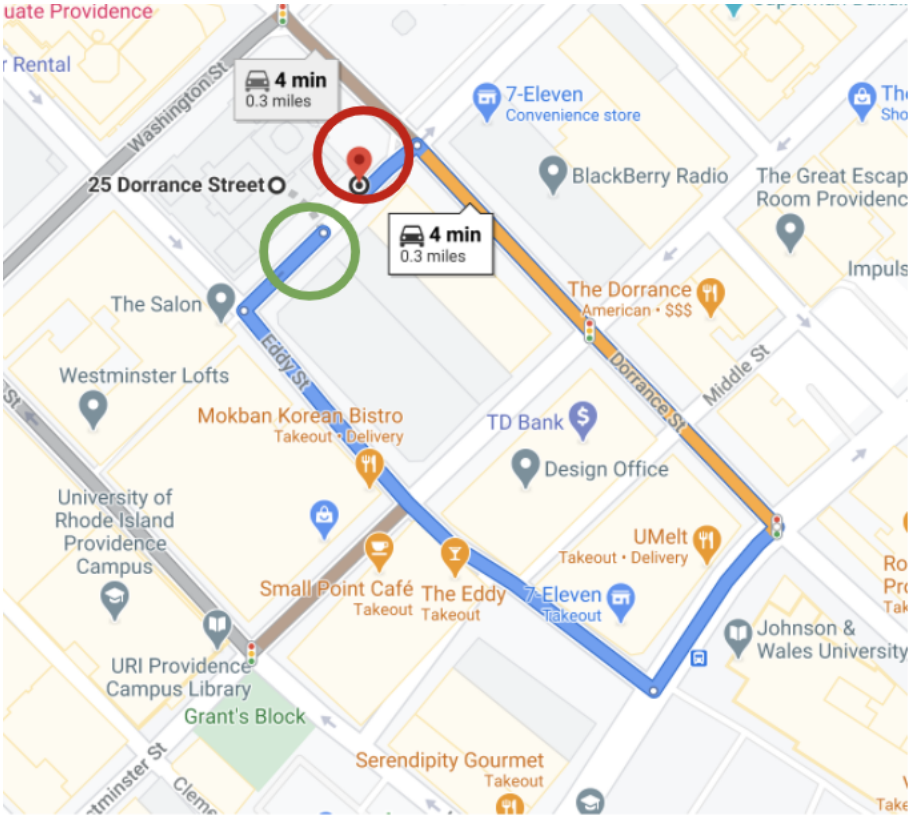


Figure 13: Mapped results of the command line application with input from above. In green is the input point, and in red is the location returned by the nearest neighbor algorithm.

The green circle encloses the input coordinates, and the red circle encloses the result – the closest coffee shop. Because the result is extremely close, a walking distance of less than 100 feet, the team considers the application to have performed extremely favorably in this case.

6 Additional Considerations

Throughout the development of this project the team had to make several hard choices in order to determine what should reasonably be taken on within the time frame of the project without compromising the final quality. Herein the team discusses their thoughts on what else could have been added to the project with additional time.

To start, the team would have liked to go back and further optimize the code - not just for the sake of the project, but for the expectation that the code will be found one day by others and improving the performance of the algorithms would provide more value to the community. Another feature would have been the ability to search for and delete an element. The problem that had arisen with this feature is that once a node is removed, if it is not from the leaf,

everything at that node and below must be destroyed and reinserted to the tree. In terms of additional features, the team strongly considered and was close to implementing various methods for returning all elements within a range, or requesting a query of all elements bound within a region, for these would be interesting to visualize, useful for many applications, and would continue to take advantage of the efficient run times of the K-D Tree data structure.

The other considerations lie within the teams application, such that in its current state it runs from command line within a terminal. These are major limiting factors when considering the usability of the application. Given more time, the team would have migrated the K-D Tree library and database of locations onto a web server to build a web-based user interface. Another limiting factor is the coverage of data featured within this prototype. Further, Yelp limits the amount of data which can be queried from it's API, rightfully so. While this prevents the team from drawing all their data from Yelp, the team would need to source data from several sources in order to increase the number of locations and the geographical area covered.

7 Contributions of Team Members

Below are the following contributions of the team members, along with the overall dynamics for how the team worked together in order to produce this project.

All

Over the past several weeks, the team members individually researched K-D Trees in order to gain a common understanding of the data structure in addition to branching out and investigating different outstanding challenges the members had foreseen needing to overcome. Some of these challenges range from the real world applications of K-D Trees, designing the code intelligently in C++, the use of class templates, and sophisticated algorithms such as the balancing algorithm Median of Medians. Further, there was the additional work put into the project such as making the presentation, writing the paper, and actually writing the code.

Raymond Turrisi

- Fronted the development of the K-D Tree templated class structure in C++.
- Produced the MATLAB® 2D static graphics, 3D animated graphics, and analysis of insertion sort and searches.
- Introduced and created/formatted the LaTeXtemplate.
- Worked on report and presentation [Introduction, Methods, Implementation, Additional Considerations].

Alfred Timperley

- Abstracted an untemplated version of the K-D Tree class working on shared pointers.
- Developed the Median of Medians implementation.
- Worked on report and presentation [Theory, Methods].

Brennan Richards

- Compiled location data set using Python and Yelp's "Fusion" API.

- Developed the command line application for a user to interface with the k-dimensional tree class.
- Worked on report and presentation [Application, Additional Considerations].

8 Acknowledgements

This team would like to thank Professor Marco Alvarez and the Teaching Assistants; Christian Esteves, John Bertsch, and Johann Muller, for putting together the resources and facilitating an environment conducive to building high levels of proficiency in this introduction to data structures and algorithms.

Bibliography

[1] *k-d tree - Wikipedia.* https://en.wikipedia.org/wiki/K-d_tree. (Accessed on 12/03/2020).

[2] J. L. Bentley and M. I. Shamos, *Divide-And-Conquer In Multidimensional Space.* Proceedings of the Eighth Annual ACM Symposium on Automata and Theory of Computing.

[3] *20.3. KD Trees — OpenDSA Data Structures and Algorithms Modules Collection.* https://opendsa-server.cs.vt.edu/ODSA/Books/Everything/html/KDtree.html. (Accessed on 12/04/2020).

[4] *kD Trees.* http://groups.csail.mit.edu/graphics/classes/6.838/S98/meetings/m13/kd.html. (Accessed on 12/04/2020).

9 Appendices

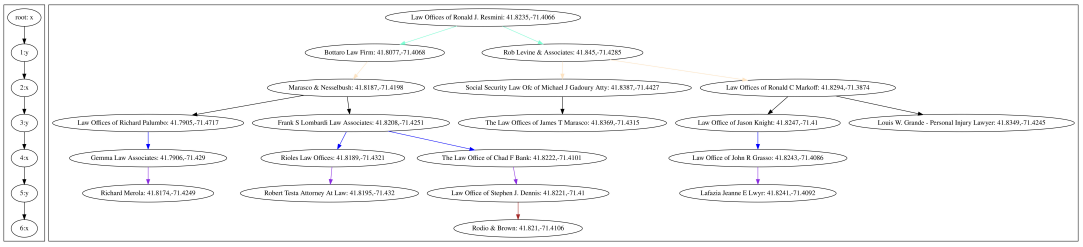


Figure 14: Graph of a tree holding lawyer locations in RI limited to 20 insertions.

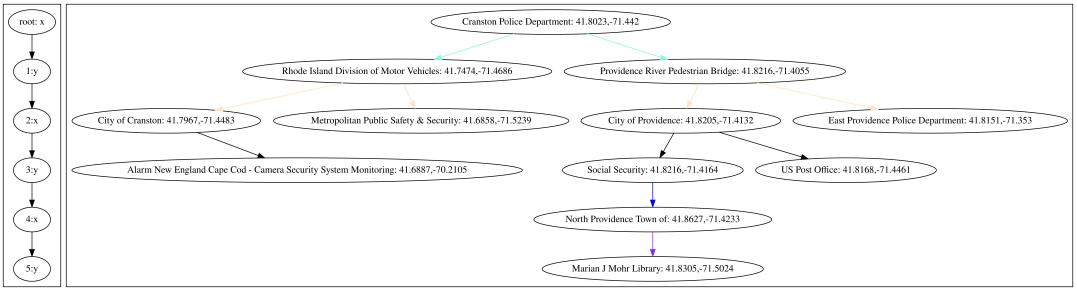


Figure 15: Graph of a tree holding police station locations in RI.

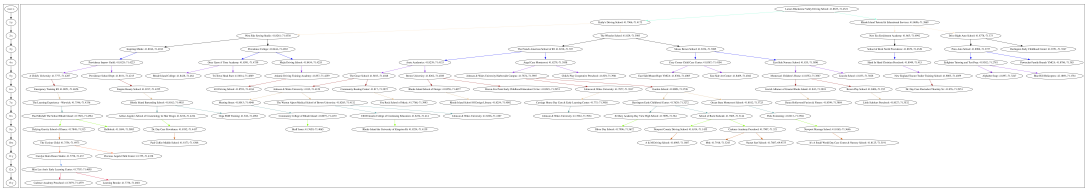


Figure 16: Graph of a tree holding school locations in RI.

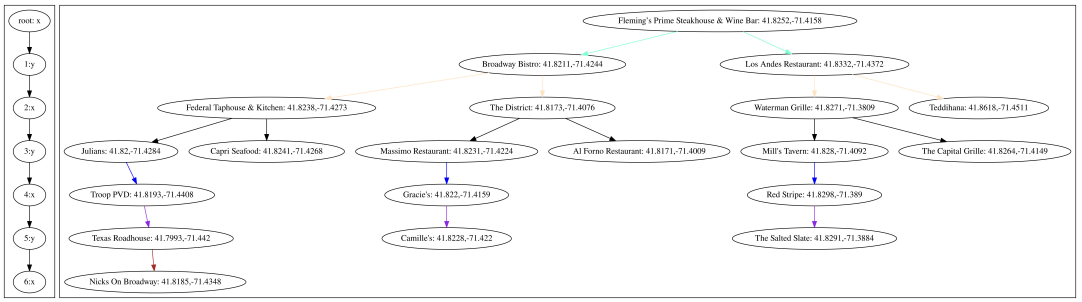


Figure 17: Graph of a tree holding steak house locations in RI, limited to 20 insertions.

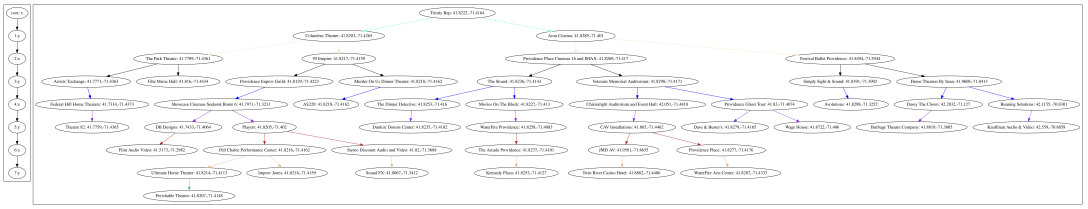


Figure 18: Graph of a tree holding theater locations in RI.

Figure 19: Graph of a tree holding towing locations in RI.

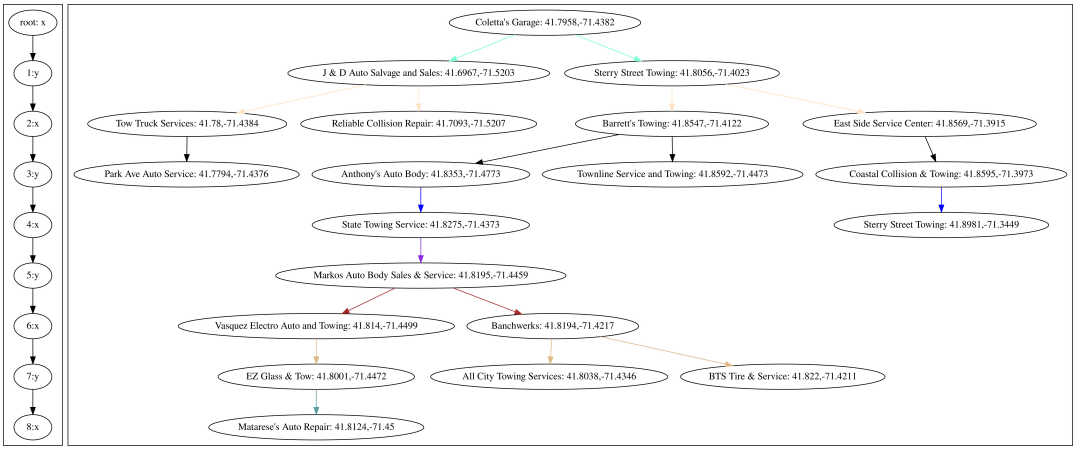


Figure 20: Graph of a tree holding towing locations in RI limited to 20 insertions.

Figure 21: Whilst not terribly practical, the **dotOut**() methods output for a tree with 10,180 elements (< 1 min).