The University of Melbourne
School of Computing and Information Systems
COMP90077 Advanced Algorithms and Data Structures

An Experimental Study on Range Trees

Marco Marasco 834482

Introduction

The following report explores the nature of different methods of construction and query operations for the 2-Dimensional Range Tree data structure. Several experiments are conducted with varying parameters to assess the performance of the construction and query operations for multiple implementations of the 2-Dimensional Range Tree. The runtimes of each of these experiments with the different Range Tree variations are recorded and plotted for analysis.

The experimentation will be conducted by running several experiments with varied size of operation sequences and varied probabilities of distinct operations occurring in the operation. The runtimes of each of these experiments with the different variations will recorded and plotted for analysis.

This report will outline the experimentation environment, explain how the data for the construction and query operations was constructed, conduct several experiments for construction and query over 2-Dimensional Range Trees, discuss the results, and conclude with a summary of the findings. Please note all references to Range Trees throughout this report are directly referring to 2-Dimensional Range Trees.

Experimentation Environment

For full transparency of the experimentation and in order to assist for reproduction of the results, an outline of the experimentation environment and tools are presented below:

- **Machine:** Dell Optiplex 990
- **Operating System:** Ubuntu 19.10
- **CPU Model:** Intel(R) Core(TM) i7-2600 CPU
- **CPU Frequency:** 3.40GHz
- **Machine RAM Capacity:** 16GB
- **Compiler:** GNU C++ Compiler v9.2.1
- **Programming Language:** C++

Data Generation

To ensure the quality of the experimentation, coordinate values are generated pseudo-randomly during runtime. At the beginning of the program that conducts the experiments, the program creates a pseudo-random number generator using the current epoch time value as an initialisation seed. As machines inherently can't generate truly random numbers, this seed allows for a pseudo-random sequence to be generated. By selecting the current time as the seed, it is highly probable that a distinct sequence of coordinates and query ranges were generated for each experiment execution.

Points for this experiment were generated using a function that generated pseudo-random integer points in 2-dimensional space within a pre-defined range. Sets of points were generated by repeatedly pseudo-randomly generating points, assigning a distinct identification value to each point, and adding them to a set. Each set was asserted to be a proper set, in that no duplicate points were contained. Query ranges were generated by generating a point (q, p) using the aforementioned method, and a defined range of $[1, 10^6 - s]$, where s is an input range length such that $1 < s < 10^6$. A query range Q was generated such that $Q = [p, p + s] \times [q, q + s]$.

The results that are analysed for each experimentation in this report is the average data for 5 separate and distinct executions of the same experiment. This has been done so to mitigate the impact experimental outliers may have on the analysis thus strengthening the accuracy of analysis on the different algorithms tested.

Construction Efficiency

Experimentation Design

The original design for the Range Tree consisted of a WB-BST where each internal node u in the base tree (on the x-coordinates) is associated with a secondary WB-BST on the y-coordinates of all the points in u's sub-tree.

Two distinct algorithms exist for constructing the data structure:

- **Contr-Naive:** The naive construction algorithm $O(n \log^2 n)$ time.
- **Contr-Sorted:** An improved construction algorithm that utilises pre-sorting for construction $O(n \log n)$.

To compare the two implementation of the two construction algorithms, an experiment was conducted to construct and record the time taken to construct two separate Range Trees that each implemented a different construction algorithm.

Each Range Tree was constructed using the same set P of pseudo-randomly points in $[1, 10^6] \times [1, 10^6]$, and the time to construct each structure was recorded. As the runtime complexities of each of the construction algorithm were dependent on the size of the set of points each of the Range Trees were constructed over, in order to analyse the performance of each of the construction algorithms, the size of the set P was varied over several execution rounds.

By varying the size of P, this allowed the runtimes to be plotted and an overall trendline to be derived across different sizes of P. The trendlines generated would be able to provide insight into the nature of each of the construction algorithms, and how their performance varied over different sizes of P. The overall focus of this experiment is to see how the runtime of each algorithm changes as the size of P changes.

Expected Theoretical Results

Both algorithms initially sort P by the x-coordinates, allowing for $O(n \log n)$ construction of the initial WB-BST of x-coordinates. For construction of the secondary WB-BST on the y-coordinates of all the points in each node u's sub-tree, the main algorithmic difference between the two construction algorithms is how **Contr-Sorted** utilises sorting P by y-coordinate prior to construction.

In the **Contr-Naive** algorithm, during the construction of each node the algorithm has access to an array of the points in u's sub-tree sorted by x-coordinates. Each node has to sort this array by y-coordinates, and then constructs the WB-BST on the y-coordinates. To construct the secondary y-coordinate WB-BST at node u, where where U is the set of points in a node u's sub-tree, takes $O(|U| \log |U|)$ time. Summing over all nodes in the primary WB-BST, the overall time complexity is bounded by $O(n \log^2 n)$.

Compared to **Contr-Sorted**, this construction algorithm is inefficient in that every node has to sort on the y-coordinates at each node. By pre-sorting the coordinates in P on both the x and y coordinates, when constructing the x-coordinate WB-BST of P, each constructed node has access to a sorted list of coordinates in its sub-tree, sorted by the y coordinate. This allows the **Contr-Sorted** to construct the secondary y-coordinate WB-BST in $O(|U|)$, where U is the set of points in a node u's sub-tree.

From a theoretical standpoint, the time complexity bounds indicate that the **Contr-Sorted** Range Tree will not only be constructed faster than the **Contr-Naive** Range Tree for each set P. Furthermore, as the time complexity of **Contr-Sorted** is of an order less than the time complexity of **Contr-Naive**, the theoretical expectation is that the gap in construction time between the two Range Trees will increase as the number points increases.

Experimental Results

Below shows the experimental results generated from conducting the Construction efficiency experiments, using point sets of size $0.1 \cdot 10^6$, $0.2 \cdot 10^6$, $0.5 \cdot 10^6$, $0.8 \cdot 10^6$, and $1 \cdot 10^6$.

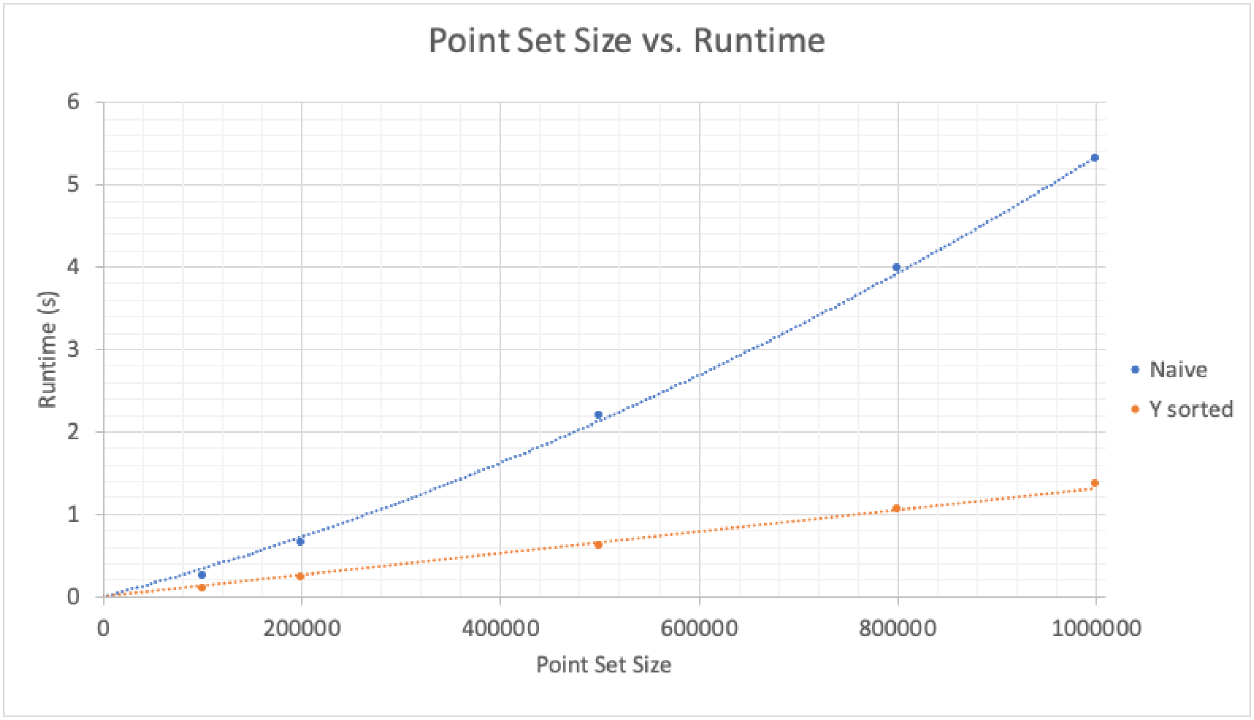

Figure 1: Construction Runtimes

An initial inspection of the results show the construction time for **Contr-Sorted** outperforms the **Contr-Naive** for every point set size. It is also plotted that as the point set size increases, so does the difference in runtime between the two construction methods.Before relating the comparative performance with respect to the expected theoretical results, it is worth assessing how each construction algorithm has performed with respect to its own theoretical properties.

n	$0.1M \to 0.2M$	$0.2M \to 0.5M$	$0.5M \to 0.8M$	$0.8M \to 1M$
Theoretical	2.248073521	2.889430333	1.716666957	1.29137896
Actual	2.608132861	3.323667156	1.817751222	1.334356736

Figure 2: Expected $O(n \log^2 n)$ Growth

Looking at the trendline for **Contr-Naive**, from a high-level it is evident the graph follows some non-linear, non-logarithmic trend. Whilst it could almost arguably be said to be linear over this range, there is a distinct upward curve to the trendline. Pleasingly, this overall upwards trend is expected for the **Contr-Naive**, with its runtime bound of $O(n \log^2 n)$. Figure 2 contains the expected and actual runtime growth between point set sizes tested for this experiment. As can be seen in the table, the **Contr-Naive** algorithm implemented closely follows the expected rate of runtime growth, and can be reasoned to be an acceptable implementation for this algorithm, that follows the correct runtime bounds.

n	$0.1M \to 0.2M$	$0.2M \to 0.5M$	$0.5M \to 0.8M$	$0.8M \to 1M$
Theoretical	2.120411998	2.687671079	1.657307193	1.270521035
Actual	2.338569044	2.695769591	1.724805138	1.296963936

Figure 3: Expected $O(n \log n)$ Growth

Upon inspection of the **Contr-Sorted** trendline, the line looks relatively linear, however has a slight curve in its shape, akin to an expected $O(n \log n)$ trendline. Looking at the expected and actual runtime growth between point set sizes in figure 3, it is clear that the runtime growth for this implementation closely follows the expected growth.

As both construction algorithms have been verified to follow their expected behaviour, the comparative results can be conducted without consideration of error in the implementation. As evident from the graph, the **Contr-Naive** performs slower than the **Contr-Sorted** in all cases, and the runtime growth is of an order higher, hence the widening gap between runtimes. These results follow the expected theoretical behaviours.

Query Efficiency

The following experiment is concerned with comparing the query efficiency of two Range Trees that are constructed differently and utilise different query algorithms.

The two types of Range Trees concerned are defined as **RangeTree-Org** and **RangeTree-FC**. The structure **Range-TreeOrg** was defined in the *Construction Efficiency* section of this report. The structure **RangeTree-FC** differs from **RangeTree-Org** in that it does not utilise a secondary WB-BST on the y-coordinates for each node in the tree, but instead each node u maintains an array of all the points in u's subtree, sorted by their y coordinate.

For a given query $Q = [a_1, b_1] \times [a_2, b_2]$, the query algorithm for **RangeTree-Org** constructed on a set P consists of following the path from the lowest common ancestor (split) of $succ(a_1)$ and $pred(b_1)$ in the x-coordinate WB-BST, and for each node on the path from the split to the aforementioned successor and predecessor, evaluating the y-coordinates of the candidate tree at that node. Overall, this achieves an runtime complexity bound of $O(k + \log^2 n)$, where k is the number of points in $P \cap Q$.

In the case of the **RangeTree-FC** for the same query Q and set of points P, the different structure requires a different query algorithm. Utilising a method known as fractional cascading, the **RangeTree-FC** query algorithm has a runtime bound of $O(k + n)$.

Query Efficiency with Fixed n and Varying s

Expected Theoretical Results

From the theoretical runtime complexities, it would be expected that for any given query range Q in a set of points P, the **RangeTree-FC** would achieve faster runtime than the **RangeTree-Org**. However, this experiment maintains a fixed size of points and varies the query range size. As a result, the variations in runtimes are expected to be caused by the varying query range size.

When comparing the two algorithms steps, they both maintain (relatively) the same steps for the WB-BST on the x-coordinates, find the LCA of $succ(a_1)$ and $pred(b_1)$, traverse down the tree to these points and assess each candidate tree found along the path, with both query algorithms having $O(\log n)$ candidate trees. The key difference is how they search of the y-coordinates in the candidate tree. For each candidate tree C rooted at a node v in the **RangeTree-Org**, the cost of range reporting the y-coordinates in the secondary WB-BST of v is $O(k_v + \log |P(v)|) = O(k_v + \log n)$, where $P(v)$ is the number of points is the number of points in the subtree rooted at v, and k_v is the number of points in $P(v) \cap Q$. By summing over all $O(\log n)$ candidate trees, the aforementioned query runtime bound of $O(k + \log^2 n)$ is reached. This however, is the runtime bound, and not necessarily representative of the time expected for each of the executions in this experiment. This bounded runtime would be expected if a query Q' was to search over the entire range of a set P'.

When analysing this query algorithm, the *expensive* portion is searching through the secondary WB-BSTs. What this indicates is that the number of candidate trees that are checked increases, the runtime of the algorithm would be expected to increase. In the context of this experiment, this increase in candidate trees would be caused by increasing the value of s, thereby increasing the query range, the length of the path from the split to the successor/predecessor, and increasing the number of candidate trees that need to be checked. As s increases, from a theoretical perspective, the runtime is expected to increase, and will tend towards the $O(k_v + n \log^2 n)$ bound as the value of s increases.

For querying in the **RangeTree-FC**, the computational expense incurred when checking y-coordinates is far less than the **RangeTree-Org**. As previously stated, each node u in the tree contains a sorted array of all the points in its subtree, sorted by the y-coordinate. Furthermore, each element in the array also maintains a two pointers to the location of its successors in the arrays stored in the left and right children of u. For the query algorithm, all that is required of the algorithm is to still find the lowest common ancestor (split) of $succ(a_1)$ and $pred(b_1)$ in the x-coordinate WB-BST, utilise a binary search ($O(\log n)$) in the split to find the successor of a_2, and on the paths from the split to the x-coordinate successor/predecessor, use the "cascading" points in the y-coordinate array to find the element in each traversed node u's y-coordinate array. By having these pointers in the y-coordinate array, the y-coordinate array in each candidate tree can be evaluated in $O(1 + k_v)$, where k_v is the number of points reported in the candidate tree's y-coordinate array.

Across the different values of s there will be several subtle differences in the different queries executed for the **RangeTree-FC**. As in the **RangeTree-Org**, increasing s will increase the query range, and thus is expected to increase the length of the path from the split to the successor/predecessor, and likely to increase the number of points reported. However, the initial binary search cost is unlikely to change, as the set size remains constant.

Comparing the two algorithms, their different query algorithms are expected to possess different behaviour for the varying values of s. As stated, the size of the paths from the split node to the the successor/predecessor is likely to increase with s. However, in the scale of 10^6 points, and each range tree height bounded by $\theta(\log n)$, the extra traversals are relatively inexpensive, and unlikely to cause any noticeable change in runtime.

Thus, it is the range reporting on the y-coordinates that is likely to cause differences in behaviour. As the **RangeTree-Org** has a bound of $O(k_v + \log |P(v)|)$ for each candidate tree, and the **RangeTree-FC** has the bound of $(1 + k_v)$, it is expected that not only will the **RangeTree-FC** perform faster in each case, as the value of s increases, and thus the number of candidate trees and reported points, the **RangeTree-Org** will be more heavily impacted than the **RangeTree-FC**, and the runtime difference between them will increase. It should be noted that due the execution environment capabilities and size of the sets queried, the runtime variations for **RangeTree-FC** will be expected to be quite small.

Experimental Results

Below shows the experimental results generated from conducting the experiments, using a point sets of size $n = 10^6$, and taking the average of 100 range queries over 5 different range query sizes s.

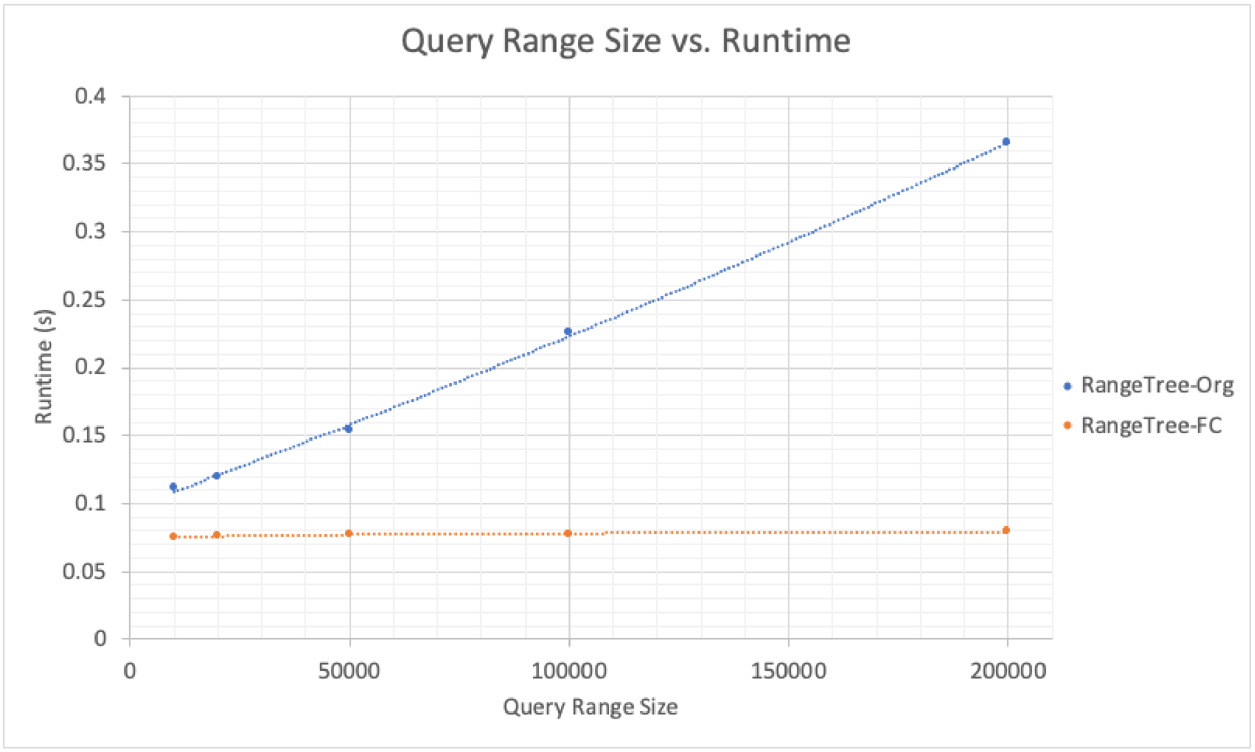

Figure 4: Query Runtimes with Fixed n and Varying s

At an initial glance of the data, it is evident that the **RangeTree-Org** has a slower runtime than the **RangeTree-FC** for every value s.

Relating back to the theory, this case-by-case behaviour was expected. The **RangeTree-Org** requires the query algorithm to report over a secondary WB-BST ($O(k_v + \log |P(v)|)$ as stated in the theoretical section), compared to the $O(1 + k_v)$ query bound for the **RangeTree-FC**. Hence as the value of s increased, and therefore the number of candidate trees that would need to be examined in each Range Tree, the theory explains why the **RangeTree-FC** would perform faster than the **RangeTree-Org**.

Whilst it is established that the **RangeTree-FC** performs faster than the **RangeTree-Org**, discussion into their respective behaviour across the varying ranges of s provides the greatest insight from the experimental results. As supported by the theory, the difference in run times increases as the value of s increases. This behaviour effectively shows the impact of larger query ranges across the two Range Trees. For the **RangeTree-Org**, it was predicted by its theoretical properties that larger query ranges would have a more significant impact on the runtime, compared to the impact on the **RangeTree-FC**'s runtime. This behaviour essentially indicates that the **RangeTree-Org** is less suited to querying in scenarios where there is likely to be a larger number of points reported. As can be seen from the graph, the varying size of s had little impact on the runtime of the **RangeTree-FC**. From a computational and theoretical perspective, the reporting of candidate trees is relatively inexpensive, hence the little change in runtime. The slight runtime changes could be exacerbated in future experimentation, using larger sets of points for constructing the **RangeTree-FC**, but maintaining all other experimentation parameters.

Query Efficiency with Fixed s and Varying n

Expected Theoretical Results

Unlike the previously conducted query-related experiment, the size of the point set P the Range Trees are constructed on is varied across this execution. In this experiment, the query range size is maintained across each execution, however, the range of the 2-Dimensional space ($[1, 10^6] \times [1, 10^6]$) from which points in P are extracted does not change, and neither does any points pseudo-random probability of being chosen change. Therefore, as the size of P decreases, a plane plotting the points of P becomes sparser. Accordingly, this experiment has the capacity to provide insight into how each Range Tree performs over querying sparser planes of points. For the given parameters of this experiment, any point p in the 2-Dimensional space has probability of being in the generated query range equal to 0.0025.

Firstly concerning the **RangeTree-Org**, with query runtime bound $O(k + \log^2 n)$, it is evident that as n ($|P|$) grows, so will the runtime. As discussed in the previous experiment, there is considerable computational expense in how the **RangeTree-Org** queries the secondary WB-BSTs. For this particular experiment, as n increases, and therefore the probability of the **RangeTree-Org** containing points in a given query range **Q**, it is expected that the runtime will increase according to the $O(k + \log^2 n)$ bound as more and more secondary WB-BSTs are searched.

For the **RangeTree-FC**, with query runtime bound $O(k + \log n)$, it is similarly expected that as n grows, so to will the runtime. For **RangeTree-FC**, searching through its array of sorted y-coordinates is relatively inexpensive, whilst not totally negligible, it does not provide the same impact onto runtime as searching y-coordinates in the **RangeTree-Org**.

For this experiment, 100 different range queries are generated with query range $[p, p + s] \times [q, q + s]$, where $p, q \in [1, 10^6] \times [1, 10^6]$ and $s = 5\% \cdot 10^6$.

By varying the size of the set of points the Range Trees are constructed over, the probability a point in each Range Tree will be reported for a given query range is reduced. For smaller values of n, there are going to be a lot of "better-case" query runtimes. As it is less likely that in each primary x-coordinate WB-BST there will be a candidate tree for the query range, there will be less occurrences of querying through the secondary y-coordinate related structures. Put simply, there will be a lot of query executions that will return very few values. As the size of n increases, and as such, the expected number of points reported for each tree increases, it is expected that the runtime will increase for both Range Trees.

By their respective complexities, it is firstly expected that the **RangeTree-Org** will perform slower than the **RangeTree-FC**. As the value of k in each Range Tree will be equal, and their runtimes for traversing the x-coordinate WB-BST will be essentially the same, it is once again the process for searching the secondary y-coordiante structure that will cause the divergence in runtime.

As n increases, the **RangeTree-Org** will runtime will tend towards an $O(k+\log^2 n)$ runtime, and the **RangeTree-FC** to approach a $O(k + \log n)$. Furthermore, this difference in order of runtime bounds should cause significant divergence between the two Range Trees, with the runtime gap increasing with n.

Experimental Results

Below shows the experimental results generated from conducting the experiments, using a constant range query size $s = 5\% \cdot 10^6$, and taking the average of 100 range queries over 10 different sets of points P_i, such that $P_i = 2 \cdot 10^3$, for $i = 1, 2, ..., 10$. To assist in visualisation of the results of this experiment, the x-axis units have been transformed into $\log n$. Due the large scaling of the sizes of n tested, the lower values of n ($< 2^4 \cdot 10^3$ clustered too tightly together on the graph.

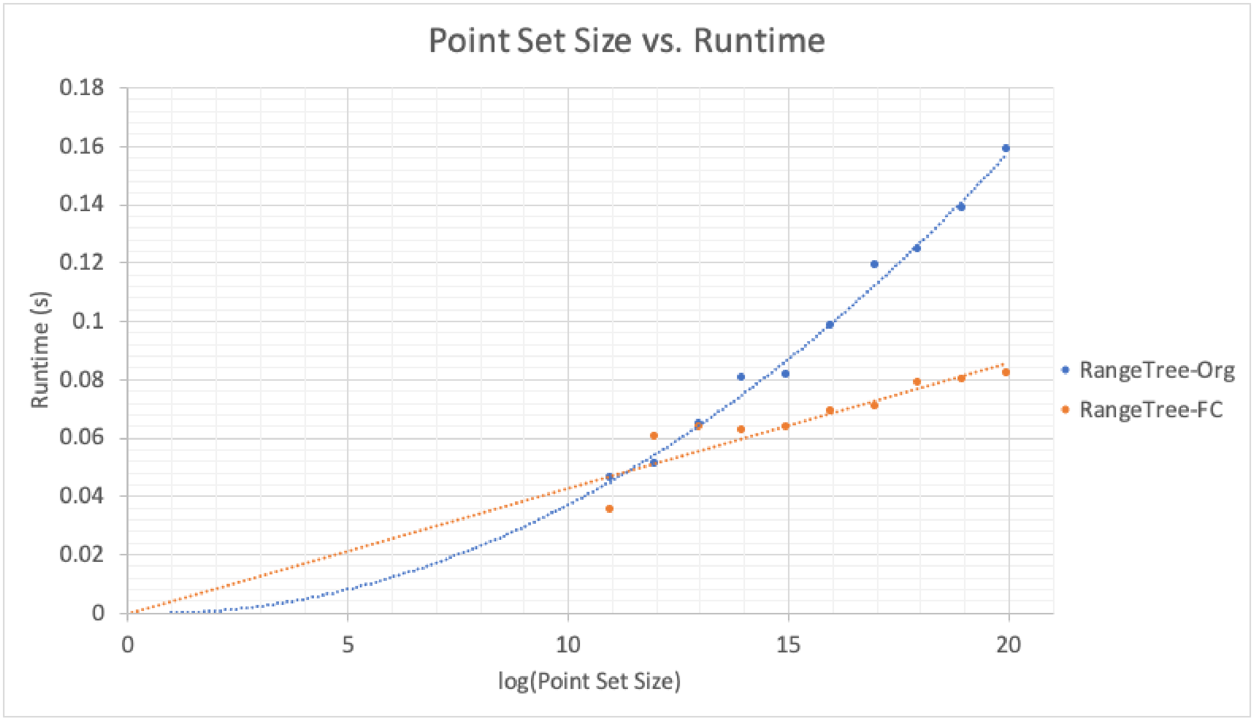

Figure 5: Query Runtimes with Fixed s and Varying n

From a first overview of the plotted results, it appears that for the first time in this report, the **RangeTree-FC** does not always perform faster than the **RangeTree-Org**, with **RangeTree-Org** having a faster query time for $n = 2^2 \cdot 10^3$. This is a deviation from the theoretical expected results, as the different runtime complexity bounds would indicate that the **RangeTree-FC** would out perform the **RangeTree-Org** for all sizes of n. Possible reasoning into this could be due to the memory access/management within the execution environment. It may be the case that for these smaller values of n, the theoretical algorithmic advantages for the **RangeTree-FC** do not outweigh the computational costs incurred for this structure in the execution environment. This anomaly aside, at the very least it appears that the repsective runtime growths of each Range Tree follow a rough expected behaviour. For the transformed graph, it would be expected the **RangeTree-Org** to follow a roughly linear line, and the **RangeTree-FC** to follow a quadratic line.

From the results of this graph, it is evident that the difference in runtimes increases as n increases, a pleasing result supported by the theoretical expectations. As was hypothesised, the **RangeTree-Org's** runtime is more significantly impacted than the **RangeTree-FC's** as the query ranges become less sparse, and more candidate trees are queried.

Conclusion

This report has discussed, implemented, and tested multiple variations of 2-Dimensional Range Tree construction and query algorithms. The first experiment conducted was concerned with the construction time between the **Contr-Naive** and **Contr-Sorted** construction algorithms. By varying the size of the set of points for construction, the analysis of the results of this experiment indicated that the **Contr-Sorted** is a superior algorithm for overall runtime, and should be the preferred construction algorithm when constructing the **RangeTree-Org**. The second experiment conducted involved critiqing the query performance for the **RangeTree-Org** and the **RangeTree-FC** over a constant set P, but varied sizes of query ranges. The performance observed for this experiment proved the **RangeTree-FC** to be a superior query algorithm. The final experiment completed was conversely concerned with a fixed query size, but varied sizes for the set of points for construction. The results from this experiment indicated that in most cases, the **RangeTree-FC** possessed a better query algorithm. From the findings of the second and third experiments, it is advisable to utilise the **RangeTree-FC** for all implementations of the Range Tree. Future work should be conducted into analysing the query performance of each Range Tree under a dynamic set of points (insertions and deletions), as the reconstruction cost of the Range Trees may present different recommendations for these more practical settings.