Advanced Algorithms and Data Structures
Assignment 2

Student ID: 760625

Name: Tom Harris

May 2020

1 Experimental Environment

These experiments were conducted on a MacBook Pro with a processor speed of 2.2 GHz and a 16GB RAM. The operating system used was macOS Mojave 10.14.6. The programming language used was Java (Java 8 - 1.8.0_202).

2 Data Generation

Each experiment involved testing the runtime of various constructions and querying algorithms of range trees and comparing their performance.

The data point generation creates a new coordinate (x and y) within the 2-dimensional integer space of two positive integer inputs. The point is generated uniformly at random. A set generation function was also used that generated a number of points using the data point generator with the inputs of $(1, 10^6)$.

The query generator creates a 2-dimensional integer space that is square shaped with an integer input side length. This is done through data point generation with the given side length being added to the point's x and y dimensions to form the query space.

3 Experiments

3.1 Exp 1: Number of Points vs. Time

3.1.1 Description

This experiment varies the size of the point set P and observes the effect on the construction runtime for the 'Naive' and 'Sorted' construction algorithm. The key difference between the two methods of construction is the alternate methods of internal node secondary tree generation. Both use a secondary WB-BST created on the y coordinate of the subtree of the internal nodes, however, differ on the method of generation.

The naive approach sorts the nodes contained within the internal node's subtree on y and then uses the standard WB-BST generation algorithm for creation of the secondary tree. WB-BST construction takes $O(n \log n)$ time, where n is the number of nodes to be generated. This gives the naive approach a $O(n \log^2 n)$ time complexity, as the secondary WB-BST generation of a single level of the primary WB-BST ($O(\log n)$ levels) can be completed in $O(n \log n)$.

The sorted method reduces this time complexity of construction by utilising an original sorting of all n coordinates on y. Secondary tree generation iteratively splits the sorted sequence into nodes present in the sub-tree of each internal node whilst retaining the sorting on y. This allows for a secondary tree to be constructed in $O(n)$ time rather than $O(n \log n)$ in the naive approach. This gives the algorithm an overall time complexity of $O(n \log n)$.

This experiment aims to demonstrate this difference in time complexity of the two construction methods. Various sized point sets were generated, ranging from $2 \cdot 10^5$ to 10^6, increasing in increments of $2 \cdot 10^5$. These point sets were then used to generate two range trees using the different construction algorithms with the runtime of each process measured. Through variation of the size of the point set, the alternate construction time complexities should be demonstrated. The experiment was run 10 times with the average run time of each construction in each instance being taken. This was done to reduce the chance of an anomalous result.

This experiment would be expected to show a general increasing trend in both algorithms as n increases. It would be expected the sorted approach will consistently perform better due to its smaller time complexity. The rate of change in the naive trend should be slightly higher as $\frac{d}{dn}(n \log n) = O(\log n) < \frac{d}{dn}(n \log^2 n) = O(\log^2 n)$, meaning each relationship should diverge.

3.1.2 Result Demonstration

See Figure 1.

3.1.3 Analysis

The results show the general trend that should be expected. Both constructions show an increasing relationship with increased n and the sorted approach does consistently outperform the naive. The naive approach does appear to be increasing at a higher rate which would be expected from the above discussion. The sorted curve appears almost linear which would be expected over the range

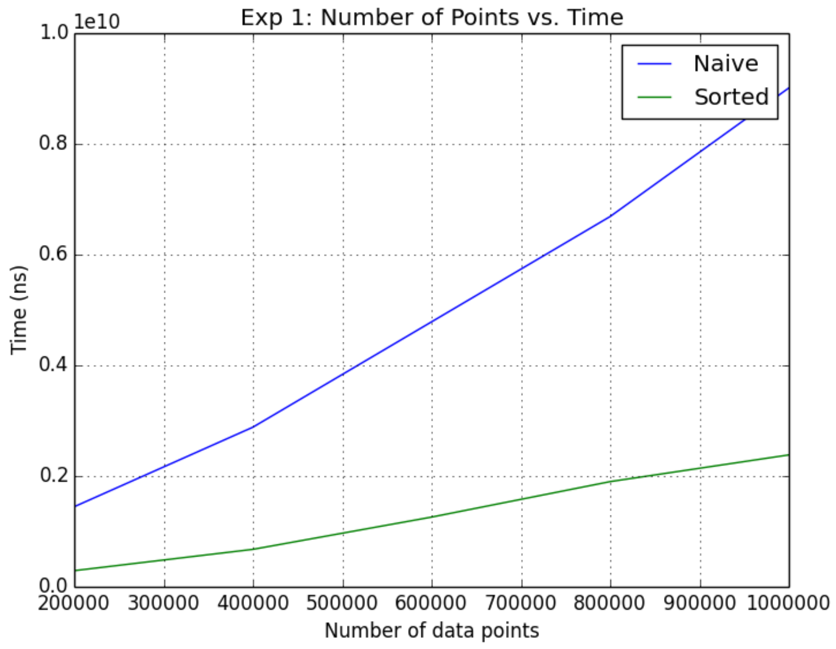

Figure 1: Exp 1 - Number of Points vs. Time

tested; assessing the rate of change at each end of the measured range, it can be observed $\log(200000) \approx 5.3$ which is fairly similar to $\log(100000) \approx 6$. Comparing this to the naive curve which has a slight increase in rate of change over the range tested, which is expected as $\log^2(200000) \approx 28.1$ which is significantly smaller than $\log^2(100000) \approx 36$.

3.2 Exp 2: Query Efficiency, with Fixed n and Varying s

3.2.1 Expectation

This experiment varies the size of the query space and measures the resulting effect on the runtime of both the original range tree querying algorithm and the variation that utilises Fractional Cascading. The original querying algorithm uses 1-dimensional range reporting to determine valid nodes within an internal node's candidate trees which has a time complexity of $O(k_v + \log n)$, where v is the root of a candidate tree and k_v is the number of nodes to be reported in the candidate tree rooted at v. As the number of candidate trees is bounded by $O(\log n)$ and the cost of reporting nodes on both paths to successor and predecessor relevant to query is bounded by $O(\log n)$, an overall run time complexity of $O(k + \log^2 n)$ is found, where k is the total number of nodes to be reported.

The Fractional Cascading variant utilises auxiliary pointers to lower sub trees to reduce the cost of querying. By linking each node's successor determinations of each y in its subtree to its left and right subtrees successor determinations, querying can be changed to avoid a second successor determination which reduces the query time complexity to $O(k + \log n$.

Given these observations, the effect of increasing the query space size should result in a equally proportional increase to both algorithms runtime. Result size (k) should be the same for both algorithms, meaning as the query space increases (and k increases), both algorithms should increase proportionally. As n remains fixed, the disparity between both algorithms should remain constant with the Fractional Cascading variant being the lower of the two.

3.2.2 Result Demonstration

See Figure 2. Note the log scale used.

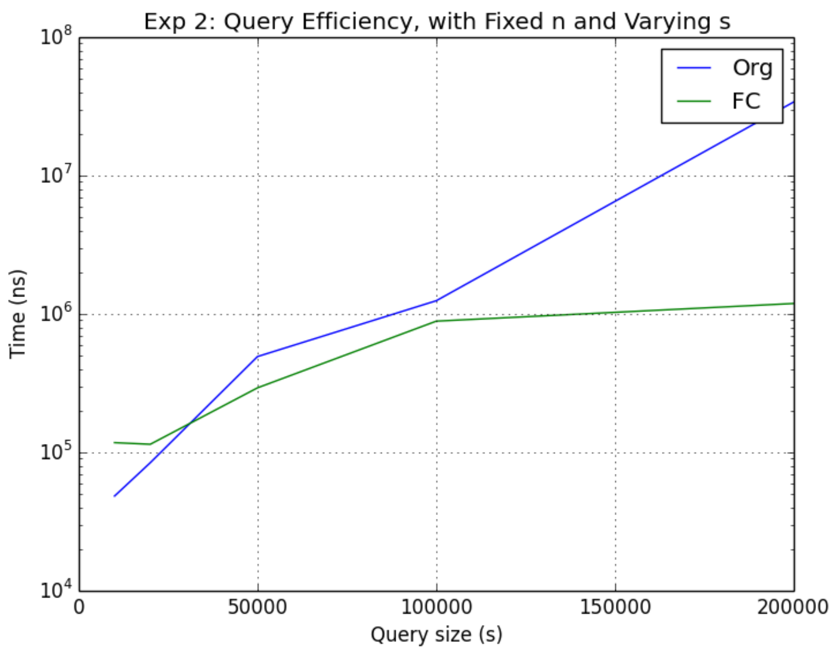

Figure 2: Exp 2 - Query Efficiency, with Fixed n and Varying s

3.2.3 Analysis

The results from this experiment were somewhat inconsistent with the predicted trend. The algorithm that incorporated Fractional Cascading largely performed better than the original algorithm which was expected; this algorithm produced a rough linear trend which taking some degree of randomness into account is the expected result. The original algorithm roughly follows this trend also, until $s > 100000$ at which point it greatly diverges. This was not expected as n in this case is fixed, meaning runtime should grow roughly linearly with query size.

There are various possible reasons for the above result. The most likely is that the implementation of the original querying algorithm may be inefficient in some areas, giving the algorithm a response that suffers in a non-linear fashion to changes in k. The code set in general is not an optimised implementation of either of the algorithms and could likely be a reason for any unexpected results.

3.3 Exp 3: Query Efficiency, with Fixed s and Varying n

3.3.1 Expectation

See 3.2.1 for discussion of difference between original and Fractional Cascading variation.

This experiment examines another aspect of both querying algorithms, varying the size of the point set. This impacts both the size of the range tree that is constructed and the resultant size of the query results (more points to fit in query range). This should give the experimentation runtimes of both algorithms an increase from increasing k (as explored in Experiment 2), but an additional increasing trend from the increasing n. This should mean the disparity between both algorithms is no longer fixed, as the $O(\log n)$ part of the Fractional Cascading method compared to the $O(\log^2 n)$ part of the original method, producing different rates of change as n increases.

3.3.2 Result Demonstration

See Figure 3

3.3.3 Analysis

These results roughly match the predicted outcome with some caveats that further demonstrate the limitations of the current implementation. Both algorithms show an increasing trend with the Fractional Cascading algorithm largely outperforming the original algorithm which was expected. Runtimes recorded for $n \leq 256000$ were fairly similar which is likely indicative of the expected trend not being significant in lower values for n. This may be due to the variability of each iteration and lower sized data sets not being large enough to clearly show the disparity. The difference between the curves in recordings at $n = 512000$ and $n = 1024000$ is more indicative of the expected result, specifically the increasing gap between the two algorithms which would be expected from the $O(\log n)$ versus $O(\log^2 n)$ parts of the time complexity bounds.

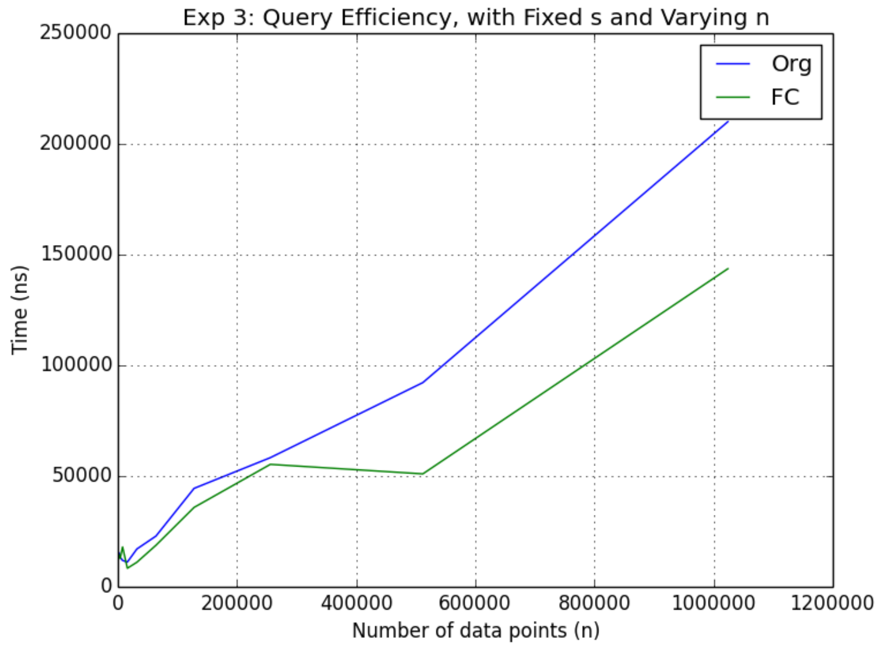

Figure 3: Exp 3 - Query Efficiency, with Fixed s and Varying n

Variability of run times for this experiment is a likely further indicator of a poor implementation. Results significantly varied between each run of the experiment with the general trends shown in the results persisting, but often being skewed by a few anomalous results. Whilst a degree of randomness in each run definitely exists (eg. data point/query generation), the difference in results between each run was significantly large. This may be indicative of the implementation of each algorithm dealing inefficiently with specific circumstances that skew the data. The slight decrease in FC runtime in the captured results between $n = 256000$ and $n = 512000$ is an example of this variability.

4 Conclusion

While a number of the trends expected in the analysis of each construction and querying algorithm were not observed clearly, the outlined experiments demonstrate various performance characteristics of the different range tree implementations. The naive and sorted approaches to construction were explored with the runtime results of both generally showing the expected trend. This experiment clearly showed the sorted algorithm is a more appropriate construction method, becoming more dominate as the number of data points in the range tree increases. The two querying algorithms explored produced mixed results, but generally demonstrated the expected trends. Whilst these experiments do not allow for a conclusive statement to be made about when each querying algorithm is more appropriate, particularly with their limitations as implemented, it is generally shown the algorithm that utilises Fractional Cascading is a better approach to querying. This is specifically relevant when the number of data points in a tree is large as demonstrated in the analysis.