The University of Melbourne
School of Computing and Information Systems
COMP90077 Advanced Algorithms and Data Structures
Semester 1, 2020

Assignment 2

James Barnes, 820946

Experimental Environment

All experiments were performed on a Dell XPS 13 (9360), sporting an Intel Core i7-8550 CPU @ 1.80 GHz with 8GB of onboard memory, running Windows 10 Home (build 19041.264). The C code, conforming to the 1999 C standard, was compiled using GCC (Ubuntu 7.4.0-1ubuntu1~18.04.1) 7.4.0, inside the Windows Subsystem for Linux, running Ubuntu 18.04.

Data Point and Query Generation

All necessary data for any given run of an experiment was generated before any timing takes place. This allows the runtime of the algorithm to be the only event measured, providing clean timing with no interference from any data generation.

Random numbers generated are generated such that they have as little skew as possible on top of the skew from the C function `rand`.

The generation of a point generates two random numbers $(x, y) \in [1, M] \times [1, M]$ forming the pair that is the point.

To generate a set of points, P, the code generates a series of points stored in an array with increasing `id`. There is no check to make sure that all points are pair-wise distinct, so in the case where two points are tied in some ordering, the point with the smaller `id` is deemed smaller.

To generate a query, Q, of size s, the code first generates a point p,
where $p = (x, y) \in [1, M - s] \times [1, M - s]$, and builds a square range with sidelengths s and origin p, i.e, $Q = [x, x + s] \times [y, y + s]$.

Experiments on Construction Efficiency

This experiment is designed to test the efficiecy of the two different construction methods for the range-tree, namely the *naīve* (unsorted) method and the *sorted* method.

These experiments follow as described below:

- For $i \in [1, 10]$
  - Generate 100 by invoking `generate_point_set`(n_i), where $n_i = 2^i \cdot 1000$
  - Construct 100 range-trees using the *naīve* and *sorted* construction methods on the same 100 sets of points
  - Take the average time over the 100 constructions for each construction method

The choice to vary n, the size of the input point set, was due to the number of input points being the only variable in the asymptotic runtime of the construction methods. The values of n were chosen to provide a large range of values for n. The choice to take the average time over 100 trials is to account for any time discrepancies caused by instabilities in the testing environment.

It is expected that, according to theory, the runtime of the *naīve* construction method should follow $O\left(n \log^2 n\right)$ growth rate and the *sorted* method should follow $O\left(n \log n\right)$ growth rate. This means also that the growth of the runtime for the *naīve* method should dominate that of the *sorted* method.

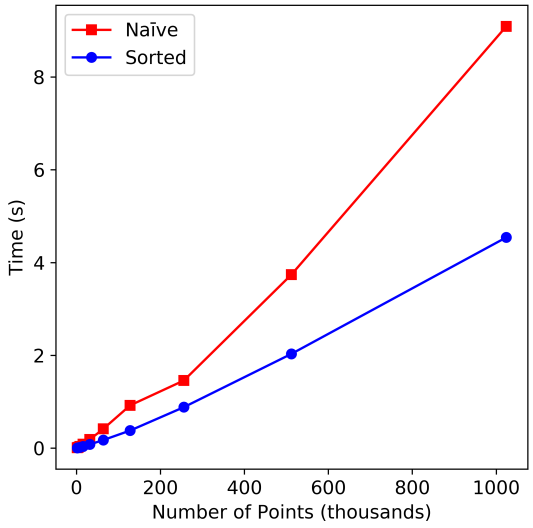

Figure 1: Experimental results for the runtime of the construction methods when varying n.

As shown in figure 1, the runtime of the *sorted* method is dominated by that of the *naīve* method, as expected. Both runtimes show superlinear behaviour. It follows that both runtimes confirm the theory, with the *naīve* and *sorted* methods plausibly exhibiting $O\left(n \log^2 n\right)$ and $O\left(n \log n\right)$ growth rates, respectively, as both runtimes are superlinear and subquadratic.

Experiments on Query Efficiency

For the purposes of these experiments, the two variants of the range-tree are referred to as *original* for the range-tree which uses range-trees as its secondary structure, and *fractional cascading* for the range-tree which utilises fractional cascading as its secondary structure.

Accoridng to theory, the query runtime of the *original* range-tree should follow $O\left(\log^2 n + k\right)$ growth rate and the *fractional cascading* range-tree should follow $O\left(\log n + k\right)$ growth rate, where n is the size of the range-tree and k is the number of reported points in the query.

The $O\left(\log^2 n\right)$ and $O\left(\log n\right)$ components of the aforementioned runtimes are referred to as the *logarithmic* components of the runtimes.

Experiment 1 - Fixed n, Varying s

This experiment is designed to test the efficiency of the two range-tree variants where the number of points, n, is fixed, and the size of the query, s, is varied.

As the size, n, of the point set, P, is constant, the logarithmic components can be considered constant. Further, as the query, Q, forms a square with sidelength s, and the points and query location is chosen uniformly at random, the expected value of $k = |P \cap Q| = O\left(|Q|\right) = O\left(s^2\right)$. This would imply the runtimes of the queries should follow quadratic growth rates of $O\left(s^2\right)$.

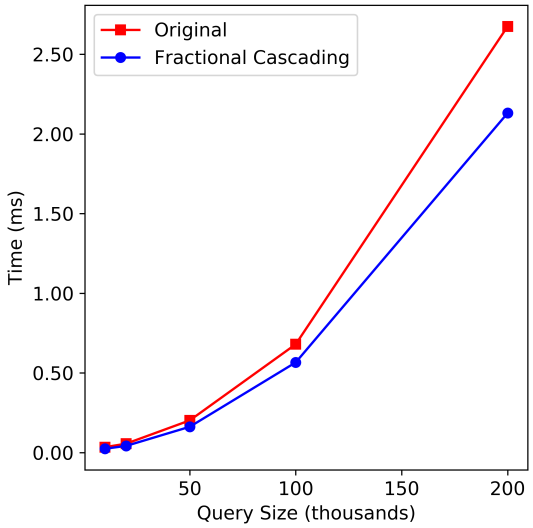

Figure 2: Experimental results for the runtime of queries when varying s.

As shown in figure 2, the runtime of both range-tree variants follow quadratic growth in their runtime, confirming the theory. Also, the runtime of the *original* range-tree is greater than that of the *fractional cascading* range-tree by some near-constant factor of approximately 1.25. This is likely due to the overhead of recursively reporting all points in the secondary range-trees in comparison to the scan of the fractional cascade.

Experiment 2 - Fixed s, Varying n

This experiment is designed to test the efficiency of the two range-tree variants where the number of points, n, is fixed, and the size of the query, s, is varied.

The size of the point set has two effects. On one hand, it increases the size of the range-tree causing an increase in the logaritmic components of the runtimes as this is affected by the size of the structure. The other effect is to increase the number of reported points, as $k = |P \cap Q| = O(|P|) = O(n)$.

As both logatithmic components are sub-linear, it is expected that for small terms there may be some appearance of logarithmic growth in the runtime, however for larger values of n the growth should exhibit linear growth, as $O(\log n + n) = O\left(\log^2 n + n\right) = O(n)$.

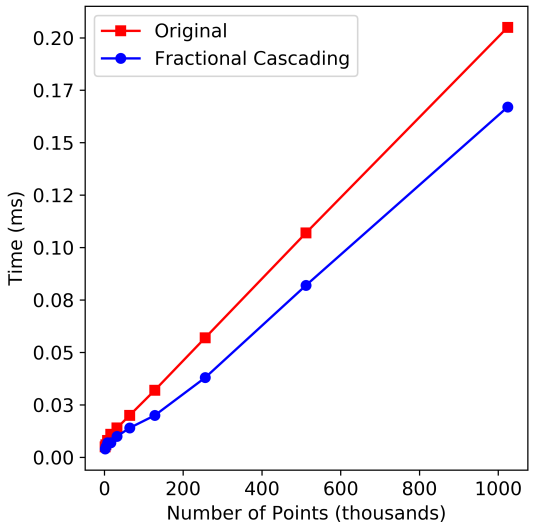

Figure 3: Experimental results for the runtime of queries when varying n.

As shown in figure 3, the runtime for larger values of n (> 200) appear to follow linear growth rates. This confirms the theory that the growth should follow $O(n)$. For the smaller values of n (< 100), the runtimes do not appear to follow a linear growth rate, and appear to follow some sort of logarithmic growth. This is probably a result of the logarithmic components of the runtimes.

Conclusion

Overall, I have found that the range-tree's performance is optimal when constructed utilising the *sorted* method and utilises fractional cascading as the secondary structue. In tandem, a range-tree constructed using the *sorted* method and using fractional cascading forms the ultimate range-tree structure. When built using a sorted-method, there is some constant factor of overhead when building the range-tree using fractional cascading. However, if the expected workload of the range-tree involves considerably more queries than constructions, the *fractional cascading* range-tree should hold superior performance in terms of runtime.