COSC2123 Algorithms & Analysis
Assignment 1: Spreadsheets
Task B: Empirical Analysis on Data Structures

By

Faith Ha, s3890479
Elissa Van, s3935201

RMIT University
College of Technology

April 19, 2023

0.1 Data Generation

The data used to analyze and compare different data structures was produced from randomly selected numbers set from -500.5 – 500.5, creating a realistic approximation of the average case for each data structure. The parameters decided on, were datasets of the size 10,000, 50,000, 100,000 and 500,000, with densities of 50%. In addition, to test if density affected the run times of each function, the densities of 50%, 75% and 100% were generated for the 100,000 dataset. Each of the datasets were stored in txt files, which were then read into the program.

Furthermore, the parameters decided on to analyse the insert and update functions were chosen from the start, middle, and end of each dataset, ensuring a fairer, less biased result. In addition, the find function was examined by averaging the times taken to search for values that existed (found) and values that do not exist (empty) in the datasets. Thus, ensuring that a robust examination was used to test the efficiency of different data structures.

0.2 Running Time Method

To evaluate the execution times of each algorithm performance, the approach taken utilized the time module for its standard and easy implementation. Albeit, it is worth acknowledging that while this method is precise it is not entirely reliable as this method of timing refers to the system clock. In which, the system clock is liable to time shifts for several reasons, some of which are due to leap days, leap seconds or the time may also synchronise with a remote clock and move backwards or forwards. The running times of each data structures and parameters were collected in seconds using time.time() function. These experiments were ran three times each, averaged and rounded to the third decimal place in order to achieve more accurate and consistent results. The average time was then used to analyze, compare, and evaluate each data structure.

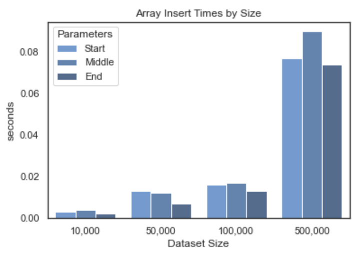
0.3 Evaluation/Analysis

0.3.1 2D Array

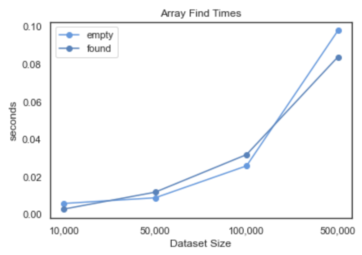
The findings seen in Figure 1, show the times for different functions according to different dataset sizes. All three functions performed exceptionally, as the time was under a second for all functions. However, a steep increase was observed when the dataset size reached 500,000, following the expectation that runtime increases proportionally to dataset size.

As seen in Figure 1a, The insert function supports the theoretical time complexity of $O(n)$ time, as there is a rough linear relationship between the dataset size and runtime. Unlike the find function (Figure 1b), which does not support the theoretical time complexity of O(n^2), as the runtimes are significantly faster than expected. This could be because the element was found quickly with an O(1) time complexity, or the element was not found at all in the array making for a $o(m * n)$ time complexity. For the find function, the observed run-times do support the theortical time complexity of $O(n^2)$, as runtimes increase proportionally to the size of the dataset.

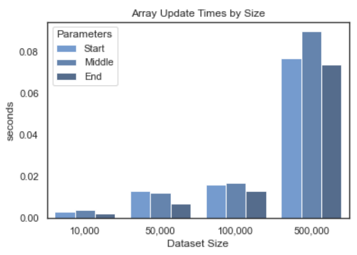
For the insert function, the start and middle parameters for both dataset size and density, performed the worst as each of the following elements in the array had to be updated, whilst the end parameter was simply appended to the end (Figure 1a & Figure 2a). As seen in Figure 1b & Figure 2b, while both the empty and found value parameters had similar times, the empty parameter performed slightly worse. Surprisingly, in the update function, the end parameters performed more quickly than the start and middle parameters (Figure 1c). This may be explained by the processor taking advantage of the cache locality to ensure faster updates.



(a)



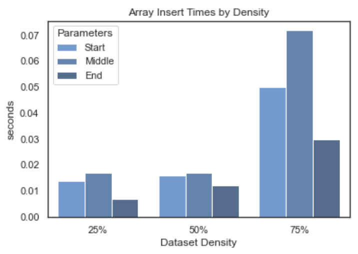
(b)



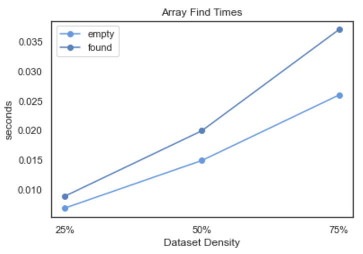
(c)

Figure 1: Array Average Execution Time Vs Dataset Size

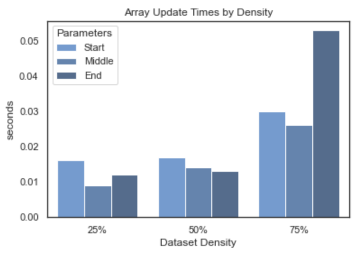
In comparison, similar times were also observed when testing the functions according to density. The 75% dataset had the highest times out of the dataset densities, as the functions had more elements to iterate over (Figure 3). Accordingly, the higher the density the longer the run time for each function.



(a)



(b)

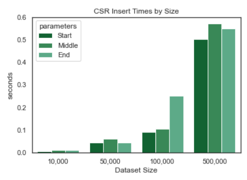


(c)

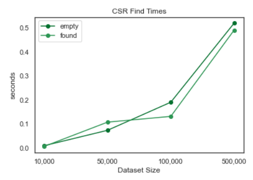
Figure 2: Array Average Execution Time Vs Dataset Density

0.3.2 CSR (Compressed Sparse Row) Matrix

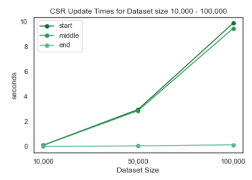
The CSR matrix data structure insert, find, and update functions all showed sudden increases when data size reached 500,000 (Figure 3). However, the insert and find times were overall quite efficient with time under a second no matter the size of the dataset. On the other hand, the update function had the slowest performance among the functions being analyzed. This is evident from the two graphs shown in Figure 3c & d, which show the steep increase of over 500 seconds for the 500,000 dataset. This can be explained as the CSR matrix requires scanning through multiple zero elements to find an element's row and column indexes. Moreover, when updating a value, the CSR data structure is not efficient, as it requires all three of its arrays to be updated.



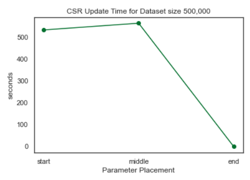
(a)



(b)



(c)



(d)

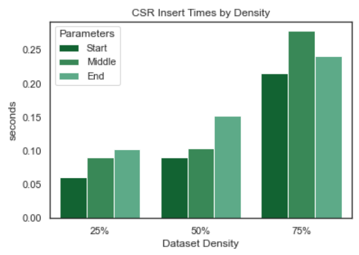
Figure 3: CSR Average Execution Time Vs Dataset Size

The insert function supports the theoretical time complexity of $O(nnz + n)$, as the relative runtimes observed are quite low, indicating that the CSR matrix is efficient with its memory usage (Figure 3a). Furthermore, the find function supports the theoretical time complexity of $O(logn)$, as runtime increases with dataset size, but not at a rate that indicates a higher time complexity (Figure 3b). However, the observed runtimes for the update function (Figure 3: c & d) suggest that it has a time complexity much higher than the theoretical time complexity of $O(k + log(n))$. This may be due to coding implementation and hardware limitations affecting the run time.

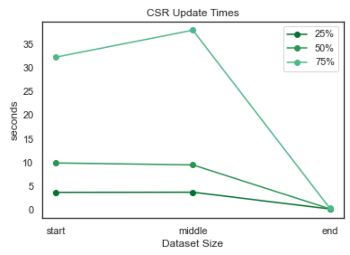
When analyzing the results of the start, middle, and end parameters in Figure 3, it is interesting to observe that the end parameters recorded a significant jump in time for the insert function. The find function also mirrored the 2D array results, with the empty value parameter having a slightly worse time, as the program iterated over all values. The update function seen in Figure 3: c & d, show how the update function's start and middle parameters have the longest run times, with a runtime of over 500 seconds for the 500,000 dataset. However, it is interesting to observe how the end parameter times for each dataset are dramatically reduced, as the sumA list does not need to be recalculated.

The density of the CSR matrix also plays an important role in the run time of each function, since the CSR matrix only stores non-zero values, it requires much less memory than a dense matrix. This is observed from Figure 4, which shows the run time for each function according to the density, with the less dense dataset having faster run times. The update function performed the most dramatically when compared over different densities, with a huge drop observed.

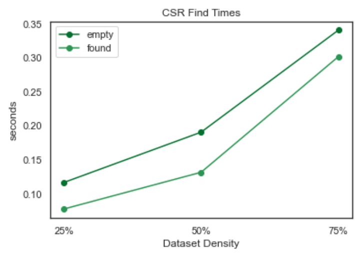
The start, middle, and end parameters seen in Figure 4, also closely mirror the results seen from Figure 3. The effect of the density of the dataset can be clearly seen from each of the graphs in Figure 4b, where observed times for the 75% dataset are clearly higher than the 25% and 50% dataset.



(a)



(b)



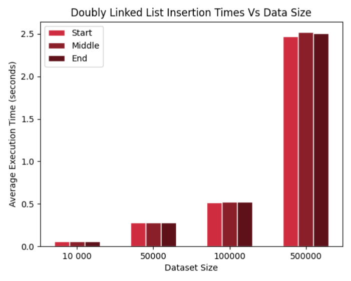
(c)

Figure 4: CSR Average Execution Time Vs Dataset Density

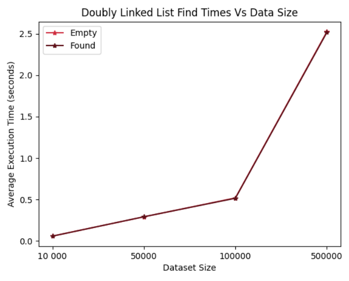
0.3.3 Doubly Linked Lists

The data structure of a linked list is linear and can be defined as a collection of nodes where each node consists of a two parts; a data part to store data and an address part in order to reference the next node. Thus, traversal of the list is only possible in the one forward direction. A doubly linked list is essentially the same, granted it has a third part, that is a second address location, allowing it to refer to previous nodes. Hence, a doubly linked list differs in the aspect that it is capable of traversing backwards and forwards as each of its nodes holds two references.

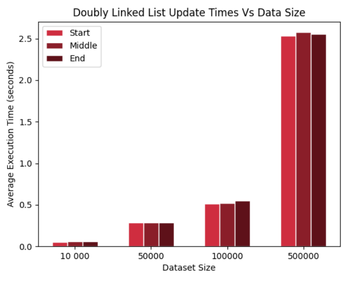
After experimenting, it is clear that in any situation whether it be inserting a value, finding the indexes of a value or updating a value at certain indexes, the notion that the greater the size of the data, the longer the running time of the data algorithm remains consistent throughout this entire experiment. It is supported not only by 2D arrays, csr matrices but also doubly linked lists as seen in figure 5. Whereby at a data set of 10 thousand, the doubly linked list algorithm took an overall average of 0.059 seconds for inserting data at the start, middle and end, compared to a larger data set of 500 thousand, the overall average time take was 2.498 seconds. The same pattern applies for updating values as the overall averaged times take for the smaller (10 thousand) compared to the larger (500 thousand) data set size was 0.057 and 2.556, respectively. When finding values that exist (found) and those that do not (empty) within the data set, analysing the data size of 10 thousand against 500 thousand, the overall averaged times taken for values that existed took 0.060 and 2.523 seconds and for those that do not exist, the time taken was 0.059 and 2.515 seconds.



(a)



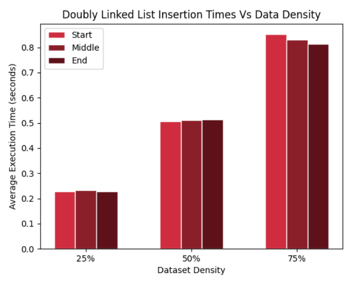
(b)



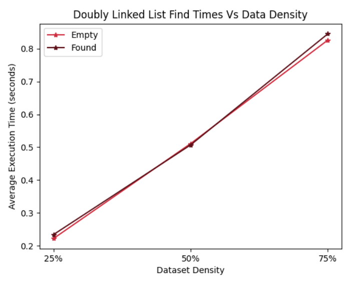
(c)

Figure 5: Doubly Linked List Average Execution Time Vs Dataset Size

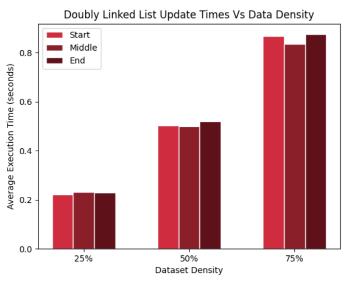
Scrutinising the running time with data density once again yields a similar concept of increased time taken for greater (75%) data densities than lower (25%) data densities. This is proven by the graphs in figure 6 in which at a density of 25% the overall averaged times taken for insertion, finding (empty, found) and updating methods are 0.230, 0.222, 0.234, 0.227 seconds, respectively. In contrast, the time taken for the same methods but at a larger density of 75% took 0.832, 0.826, 0.845, 0.859 seconds. Additionally, these times aligns with doubly linked lists time complexity of $O(n)$ for updating or searching and $O(1)$ for insertion and deletion. Since we are capable of traversing the list backwards and forwards, starting at the head or at the tail due its properties. When it comes to inserting, the time taken might be slightly longer than 2D arrays and CSR as a result of both address parts of the nodes needing to be updated accordingly.



(a)



(b)



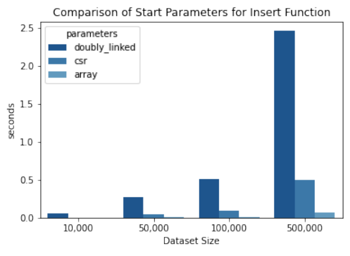
(c)

Figure 6: Doubly Linked List Average Execution Time Vs Dataset Density

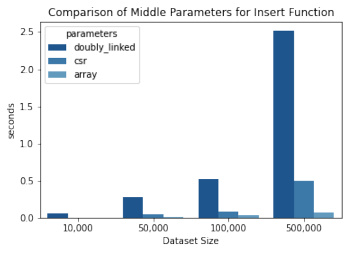
0.4 Conclusion

Ultimately, the study has several limitations such as use of only one hardware and software configuration to test the algorithms, since the efficiency of the software used might influence the resultant times. The chosen method for measuring time, that is, the time module was also a limitation due to its nature of being platform-dependent, and might vary depending on the operating system capabilities. Moreover, the study only considered a limited number of dataset sizes and densities. In future research, exploring a wider range of dataset sizes and densities would provide a more thorough understanding of the time complexity of the different data structures. Overall, in any scenario, be it inserting, searching or updating values, the belief that execution time increases as data size and density increases holds true for 2D arrays, CSR and doubly linked lists. In which, 2D arrays performed the best out of the three with the fastest run-times in all given circumstances. However, it is worth noting that the efficiency of the way the code was implemented can play a role in the speed of the algorithm's run times, as was the case for the CSR matrix. Whilst, doubly linked lists may provide a flexible and efficient way to represent and manipulate spreadsheets of arbitrary size. It may not be as effective as 2D arrays when it comes to dealing with very large sizes of data, as each cell is represented by a singular node, which consequently can result in high memory usage. Hence, 2D arrays are most suitable and recommended in this case of building a spreadsheet, that is not to say it is more superior than CSR matrix and doubly linked lists. In fact, CSR matrix or doubly linked lists can potentially be more advantageous and appropriate than 2D arrays depending on the type of data, its size, density and the scenario one is dealing with.

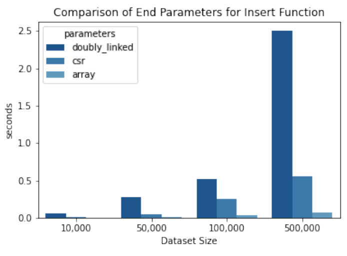
0.5 Appendix



(a)

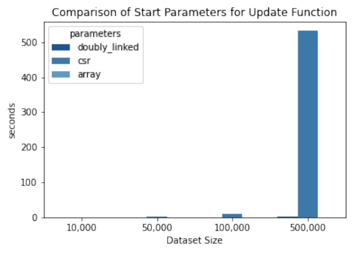


(b)

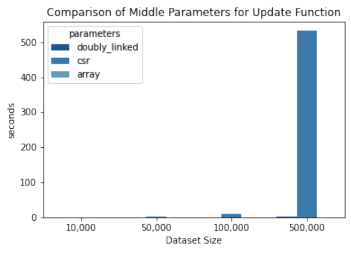


(c)

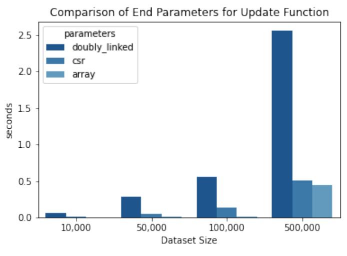
Figure 7: Comparison of each data structure's parameter insert times Vs Dataset Size



(a)

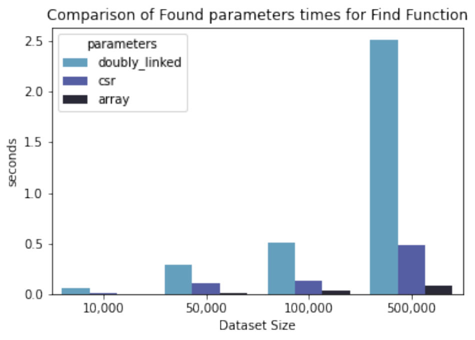


(b)

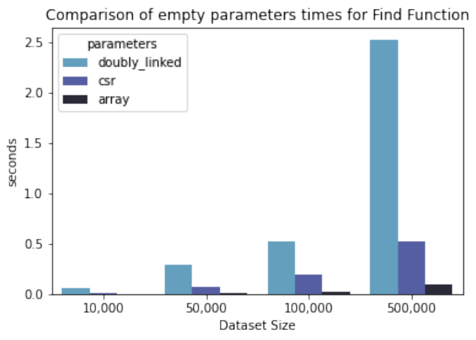


(c)

Figure 8: Comparison of each data structure's parameter update times Vs Dataset Size



(a)



(b)

Figure 9: Comparison of each data structure's parameter Find times Vs Dataset Size