Theoretical and Empirical Comparative Analysis on Implementing Different Data Structures to Create the Spreadsheet

Background

We created the spreadsheets using array (list in Python), double-linked list, and Compressed Sparse Row (CSR) to determine the most suitable data structure for each given scenario. In this comparative analysis, we evaluated and compared the performance of all three data structures under different input data scenarios. In addition, we aimed to compare and contrast the actual performance against the theories.

Methods

- **Data generation design**
  We have decided to have three sizes of datasets with different total number of cells: small (100, 400, 900) medium (1600, 2500, 4900), and large (10,000, 22500, 50625). We believe that these numbers are adequate to observe differences across all data structures. Due to our device limitations, these numbers are an appropriate range for us to run without crashing. Along with the different sizes, we also have 3 different density levels (20%, 50%, 100%) to see whether the density also affects the running time. In total, there are 9 sets of data. These datasets are in the form of text files (txt). This is to adhere to the same file data as the sampleData.txt file that was provided for us in the assignment specifications.

Dataset Specifications

File Size (S/M/L)	Total Cells	Total Cells with Value	Number of rows	Number of columns	Density
S	100	20	10	10	20%
S	400	200	20	20	50%
S	900	900	30	30	100%
M	1,600	320	40	40	20%
M	2,500	1,250	50	50	50%
M	4,900	4,900	70	70	100%
L	10,000	2,000	100	100	20%
L	22,500	11,250	150	150	50%
L	50,625	50,625	225	225	100%

- **Data generation tool**
  The data generation method (see Appendix B) was tested and experimented on Jupyter Notebook. The advantage of utilising Jupyter Notebook is because it has a user-friendly interface. The interface allows quick trial and error where the user can make changes to the code simultaneously. Jupyter Notebook also allows external Python Libraries such as Pandas, which we used during our data generation.

- **Experimental Setup**
  To set up our project, we connected to Filezilla with our RMIT student account through **<studentnumber>@titan.csit.rmit.edu.au** and placed our python project inside the Assignment1 folder after installing miniconda. To run the commands, we created a working environment called “myenv”. To activate the newly created environment, we run the command “**conda activate myenv**” after changing our directories to the project folder.

- **Method used for running time**
  To measure the running time of each operation, we made an adjustment to the spreadsheetFilebased.py python file. We imported the time module in order to utilise **the time() function**. The function was implemented at the beginning of where the code parses the command, and also towards the end of the method. Then we used the print function to print out “Running time: {duration} seconds”.

- **Data collection**
  In order to run the tests, we need to generate an input file and run it on the **titan.csit.rmit.edu.au** server. In each input file, there are 150 commands. We believe this number is sufficient in generating consistent results without crashing our operating system. An example of a command that was used to collect data is: **python3 spreadsheetFilebased.py csr largedata20.txt insertCol.in resultCsr.out.** The insertCol.in is the input file. The result of this command will return the run time for a CSR data structure to complete the insert column function when traversing through a large dataset. The result will be saved into the resultCsr.out output, while also printing the duration it took to complete the function. To ensure the robustness of our data collection, we ran the command 5 times for each test to prevent outliers. Then we used the average of the 5 results to plot on a graph in order to observe the trend when adjusting parameters (size and density).

Evaluation of the outcome using the generated data

As a result of analysing the graphs, we have made several observations regarding the different functions.

Insert Row Function (see Appendix C)

- Data structure CSR is the most efficient in running this function
- The run time of CSR remained somewhat the same when the size and the density increased

Insert Column Function (see Appendix D)

- Similar trend as the insert row function
- CSR remained the most efficient data structure
- Increased in run time for CSR compared to insert row function, but still remained most efficient

Update Function (see Appendix E)

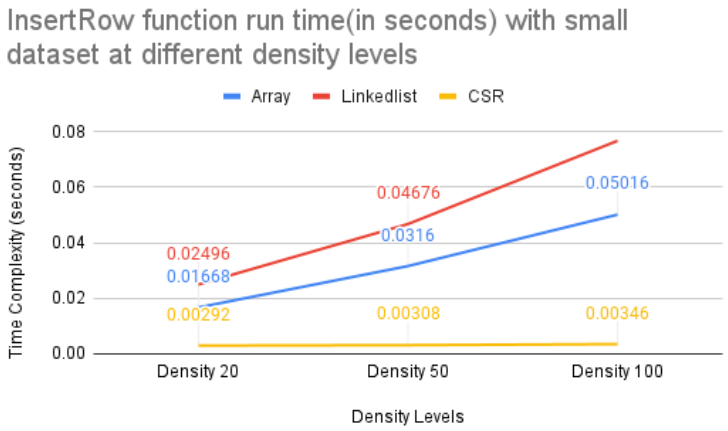
- Both Array and Double Linkedlist are more efficient than CSR, as they follow a similar pattern
- It is observed that as density increases to 100%, both Array and Double Linkedlist reach a plateau
- Whereas data structure CSR requires significantly more time to complete (see Appendix E)

Find Function (see Appendix F)

- Doubly-Linked List is observed to be more efficient as the data size and density increases
- Array is roughly 50% slower than Doubly-Linked List
- CSR is observed to be the least efficient, it has seen a significant increase in run time as the data size and density increases

In the insert row and insert column graphs, there is a clear trend given that as the dataset increases in size and density, the longer it takes for the function to initiate.

As an example to illustrate this trend, the graph below shows that CSR is the most efficient data structure when performing the insert row function because it remains a constant speed (an average of 0.00315 seconds) when the size and the density increases.



On the contrary, the data structure array and linked list are observed to have an increased run time as the density increases.
In the find function graphs (see Appendix F), the trend is similar to the insert row and insert column graphs where the time to complete function increases when the dataset size and density increases.
Whereas in the update function (see Appendix E), the trend is less apparent in array and linked list operations. This is potentially due to errors that were made during the designing process.

Comparative Analysis

- **Theoretical expected time complexity**

Time Complexity	Array	Linked List	CSR
Insert row	$O(n^2)$	$O(n^2)$	O(1)
Insert column	$O(n^2)$	$O(n^2)$	O(n)
Find	$O(n^2)$	$O(n^2)$	$O(n^2)$
Update	O(1)	O(n)	O(n)

We collected the above Theoretical time complexity by checking our Python codes, e.g., arraySpreadsheet.py, linkedlistSpreadsheet.py, and csrSpreadsheet.py.

- **Empirical time complexity**

From the time complexity graphs in Appendices (see Appendix C to Appendix F), the time complexity is as follows:

Time Complexity	Array	Linked List	CSR
Insert row	$O(n^2)$	$O(n^2)$	O(1)
Insert column	$O(n^2)$	$O(n^2)$	O(n)
Find	$O(n^2)$	$O(n^2)$	$O(n^2)$
Update	O(1)	O(1)	$O(n^2)$

The majority of the results are as observed. The running times are supported by the theoretical time complexities of the operations of each approach, e.g., array, linked list, and CSR. To clarify, when comparing the theoretical and empirical time complexity tables, the performances of all the data structures align with their theoretical time complexities, except for the update function, where the time complexity of update function for the linked list should be O(n) instead of O(1), as well as the time complexity of update function for the CSR should be O(n) instead of $O(n^2)$. The difference may be a consequence of an inefficient way of designing Linked Lists and CSR spreadsheets.

Recommendation

It is important to keep in mind that there isn't one single data structure that is the best choice. According to the results of our tests, the efficiencies of the data structures vary depending on the size of the spreadsheet, the density of the dataset, and the functions that are required. As a result, we have concluded the best data structure to use under certain conditions:

Spreadsheets that require Insert row and Insert column:

- We recommend **CSR** since it has the lowest run time compared to array and double linked list

Spreadsheets that require Update function:

- We recommend **Array** and **Double Linked List** since it has the lowest run time

Spreadsheets that require Find function:

- We recommend **Double Linked List** since it has the lowest run time, and can handle large data files quicker than array and CSR.

Appendices

Appendix A - Test Data

Appendix A.1 - Insert Row Function Test Data in Seconds

*Bolded numbers represent averages of the five tests

IR	sdata20	sdata50	sdata100	mdata20	mdata50	mdata100	ldata20	ldata50	ldata100
Array	0.0171 0.0161 0.0166 0.0166 0.0170 **0.01668**	0.0314 0.0317 0.0329 0.0314 0.0306 **0.0316**	0.0495 0.0511 0.0499 0.0491 0.0512 **0.05016**	0.0707 0.0691 0.0694 0.0695 0.0694 **0.06962**	0.0928 0.0936 0.0929 0.0929 0.0953 **0.0935**	0.1514 0.1507 0.1511 0.1509 0.1492 **0.15066**	0.2622 0.2624 0.2733 0.2627 0.2662 **0.26536**	0.5162 0.5322 0.5211 0.5311 0.5128 **0.52268**	1.0654 1.0274 1.2234 1.0663 1.0453 **1.08556**
DLL	0.0249 0.0248 0.0248 0.0249 0.0254 **0.02496**	0.0463 0.0464 0.0462 0.0469 0.0480 **0.04676**	0.0826 0.0737 0.0770 0.0767 0.0739 **0.07678**	0.1032 0.1041 0.1010 0.1030 0.1062 **0.1035**	0.1410 0.1387 0.1392 0.1411 0.1387 **0.13974**	0.2130 0.2311 0.2317 0.2315 0.2260 **0.22666**	0.3938 0.3964 0.3907 0.3989 0.3883 **0.39362**	0.7650 0.7792 0.7849 0.7752 0.7900 **0.77886**	1.7961 1.7992 1.8011 1.8121 1.7900 **1.7997**
CSR	0.0030 0.0028 0.0029 0.0029 0.0030 **0.00292**	0.0030 0.0031 0.0031 0.0031 0.0031 **0.00308**	0.0036 0.0036 0.0033 0.0036 0.0032 **0.00346**	0.0042 0.0041 0.0033 0.0032 0.0031 **0.00358**	0.0033 0.0041 0.0033 0.0032 0.0030 **0.00338**	0.0036 0.0033 0.0036 0.0032 0.0036 **0.00346**	0.0032 0.0033 0.0032 0.0033 0.0031 **0.00322**	0.0035 0.0035 0.0035 0.0030 0.0033 **0.00336**	0.0039 0.0041 0.0036 0.0039 0.0039 **0.00388**

Appendix A.2 - Insert Column Function Test Data in Seconds

*Bolded numbers represent averages of the five tests

IC	sdata20	sdata50	sdata100	mdata20	mdata50	mdata100	ldata20	ldata50	ldata100
Array	0.0255 0.0275 0.0276 0.0276 0.0277 **0.02718**	0.0548 0.0549 0.0549 0.0546 0.0545 **0.05474**	0.0877 0.0816 0.0877 0.0875 0.0884 **0.08658**	0.1245 0.1184 0.1242 0.1261 0.1211 **0.12286**	0.1690 0.1680 0.1697 0.1700 0.1678 **0.1689**	0.2529 0.2707 0.2666 0.2702 0.2603 **0.26414**	0.4581 0.4647 0.4544 0.4554 0.4695 **0.46042**	0.8747 0.8840 0.8980 0.8788 0.8740 **0.8819**	1.7588 1.7890 1.8120 1.7579 1.7681 **1.77716**
DLL	0.0365 0.0358 0.0402 0.0402 0.0365 **0.03784**	0.0723 0.0721 0.0700 0.0721 0.0771 **0.07272**	0.1169 0.1169 0.1711 0.1701 0.1165 **0.1383**	0.1654 0.1666 0.1596 0.1680 0.1580 **0.16352**	0.2162 0.2201 0.2131 0.2129 0.2158 **0.21562**	0.3541 0.3566 0.3466 0.3445 0.3539 **0.35114**	0.5535 0.5832 0.5819 0.5566 0.5821 **0.57146**	1.1299 1.1321 1.1353 1.1245 1.1335 **1.13106**	3.0773 3.1281 3.3110 3.0821 3.2100 **3.1617**
CSR	0.0033 0.0034 0.0031 0.0033 0.0035 **0.00332**	0.0068 0.0067 0.0069 0.0067 0.0067 **0.00676**	0.0221 0.0219 0.0208 0.0215 0.0217 **0.0216**	0.0101 0.0093 0.0095 0.0092 0.0094 **0.0095**	0.0280 0.0279 0.0284 0.0290 0.0289 **0.02844**	0.1095 0.1013 0.1014 0.1014 0.1026 **0.10324**	0.0448 0.0435 0.0434 0.0462 0.0435 **0.04428**	0.2306 0.2321 0.2301 0.2306 0.2311 **0.2309**	1.0713 1.0611 1.0702 1.0719 1.0713 **1.06916**

Appendix A.3 - Update Function Test Data in Seconds

*Bolded numbers represent averages of the five tests

U	sdata20	sdata50	sdata100	mdata20	mdata50	mdata100	ldata20	ldata50	ldata100
Array	0.0034 0.0032 0.0031 0.0032 0.0029 **0.00316**	0.0036 0.0033 0.0038 0.0038 0.0035 **0.0036**	0.0039 0.0039 0.0032 0.0034 0.0039 **0.00366**	0.0038 0.0040 0.0039 0.0039 0.0035 **0.00382**	0.0040 0.0035 0.0041 0.0038 0.0032 **0.00372**	0.0038 0.0038 0.0035 0.0041 0.0033 **0.0037**	0.0036 0.0034 0.0041 0.0035 0.0035 **0.00362**	0.0037 0.0037 0.0038 0.0039 0.0043 **0.00388**	0.0045 0.0043 0.0045 0.0045 0.0053 **0.00462**
DLL	0.0033 0.0033 0.0033 0.0038 0.0033 **0.0034**	0.0034 0.0035 0.0039 0.0039 0.0039 **0.00372**	0.0039 0.0029 0.0038 0.0036 0.0039 **0.00362**	0.0068 0.0047 0.0048 0.0044 0.0044 **0.00502**	0.0035 0.0040 0.0050 0.0043 0.0051 **0.00438**	0.0055 0.0046 0.0037 0.0038 0.0048 **0.00448**	0.0061 0.0055 0.0056 0.0054 0.0054 **0.0056**	0.0057 0.0051 0.0051 0.0047 0.0051 **0.00514**	0.0061 0.0061 0.0057 0.0057 0.0061 **0.00594**
CSR	0.0039 0.0034 0.0036 0.0039 0.0058 **0.00412**	0.0044 0.0037 0.0034 0.0041 0.0047 **0.00406**	0.0066 0.0075 0.0078 0.0080 0.0076 **0.0075**	0.0053 0.0072 0.0063 0.0066 0.0064 **0.00636**	0.0191 0.0195 0.0169 0.0167 0.0171 **0.01786**	0.1154 0.1159 0.1155 0.1160 0.1161 **0.11578**	0.0938 0.0904 0.0889 0.0900 0.0947 **0.09156**	0.3169 0.3241 0.3226 0.3167 0.3623 **0.32852**	0.9174 0.9146 0.9141 0.9225 0.9110 **0.91592**

Appendix A.4 - Find Function Test Data in Seconds

*Bolded numbers represent averages of the five tests

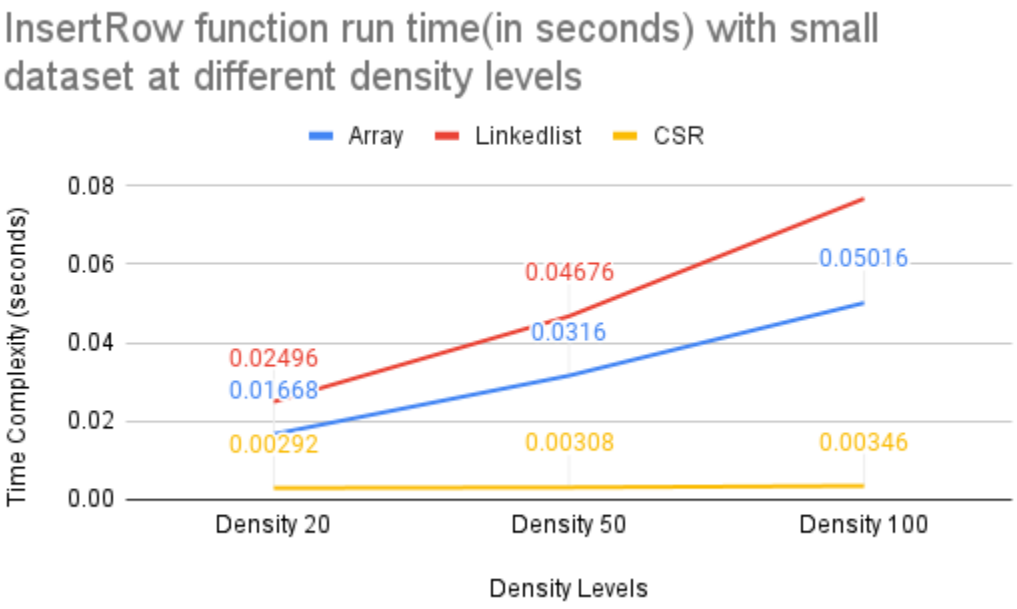
F	sdata20	sdata50	sdata100	mdata20	mdata50	mdata100	ldata20	ldata50	ldata100
Array	0.0144 0.0062 0.0064 0.0068 0.0068 **0.00812**	0.0157 0.0149 0.0161 0.0144 0.0157 **0.01536**	0.0235 0.0309 0.0330 0.0324 0.0325 **0.03046**	0.0379 0.0352 0.0362 0.0347 0.0385 **0.0365**	0.0755 0.0725 0.0689 0.0698 0.0729 **0.07192**	0.1821 0.1744 0.1771 0.1743 0.1775 **0.17708**	0.2008 0.1990 0.1824 0.1884 0.1812 **0.19036**	0.5834 0.5838 0.5732 0.5873 0.5754 **0.58062**	1.8576 1.8373 1.8752 1.7577 1.8173 **1.82902**
DLL	0.0187 0.0108 0.0098 0.0092 0.0089 **0.01148**	0.0242 0.0226 0.0215 0.0237 0.0219 **0.02278**	0.0475 0.0466 0.0458 0.0472 0.0510 **0.04762**	0.0703 0.0664 0.0643 0.0640 0.0658 **0.06616**	0.1217 0.1159 0.1168 0.1146 0.1185 **0.1175**	0.2739 0.2566 0.2758 0.2601 0.2535 **0.26398**	0.3701 0.3670 0.3713 0.3734 0.3707 **0.3705**	1.0237 0.9698 0.9838 1.0020 0.9732 **0.9905**	2.7960 2.6204 2.7412 2.6591 2.6712 **2.69758**
CSR	0.0101 0.0066 0.0060 0.0091 0.0069 **0.00774**	0.0239 0.0230 0.0232 0.0238 0.0231 **0.0234**	0.0895 0.0903 0.0873 0.0893 0.1118 **0.09364**	0.0371 0.0427 0.0359 0.0407 0.0382 **0.03892**	0.1327 0.1326 0.1311 0.1289 0.1317 **0.1314**	0.4939 0.4879 0.4882 0.4878 0.4947 **0.4905**	0.2078 0.2037 0.2044 0.2070 0.2045 **0.20548**	1.1201 1.1348 1.1350 1.1190 1.1415 **1.13008**	5.1358 4.9561 5.0379 5.0627 4.9500 **5.0285**

Appendix B - Data generation code

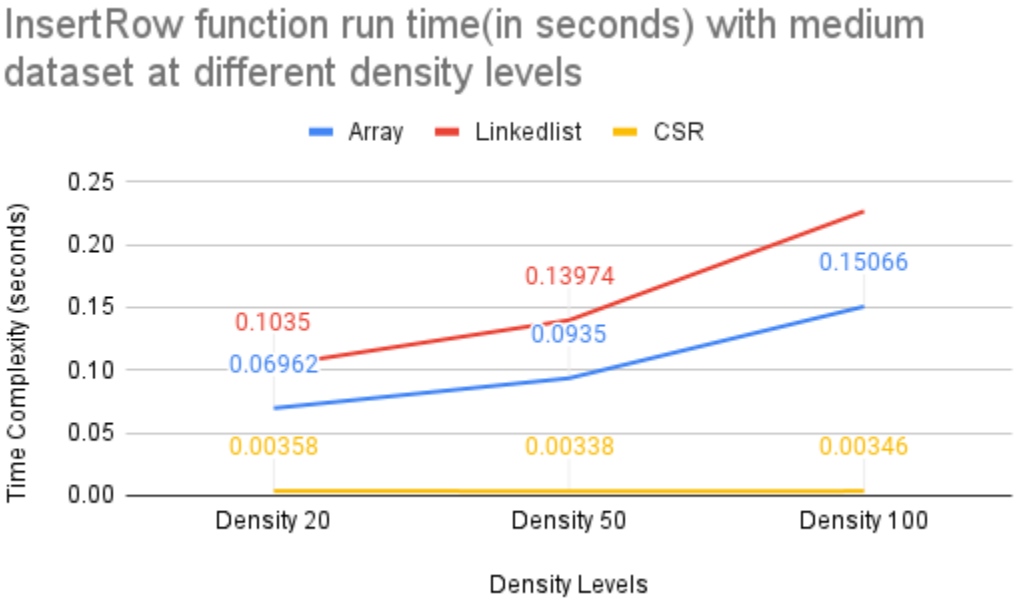
```
# File creation
with open('smalldata20.txt', 'w') as f:

    # Generate the data
    rows = set()
    while len(rows) < 20:
        row_number = random.randint(1, 10)
        column_number = random.randint(1, 10)
        while (row_number, column_number) in rows:
            # Avoid duplicates
            row_number = random.randint(1, 10)
            column_number = random.randint(1, 10)
        cell_value = round(random.uniform(-10, 10), 2)

        # Write the data to the file
        f.write('{} {} {}\n'.format(row_number, column_number, cell_value))
        rows.add((row_number, column_number))
```

Appendix C - Insert Row Graphs

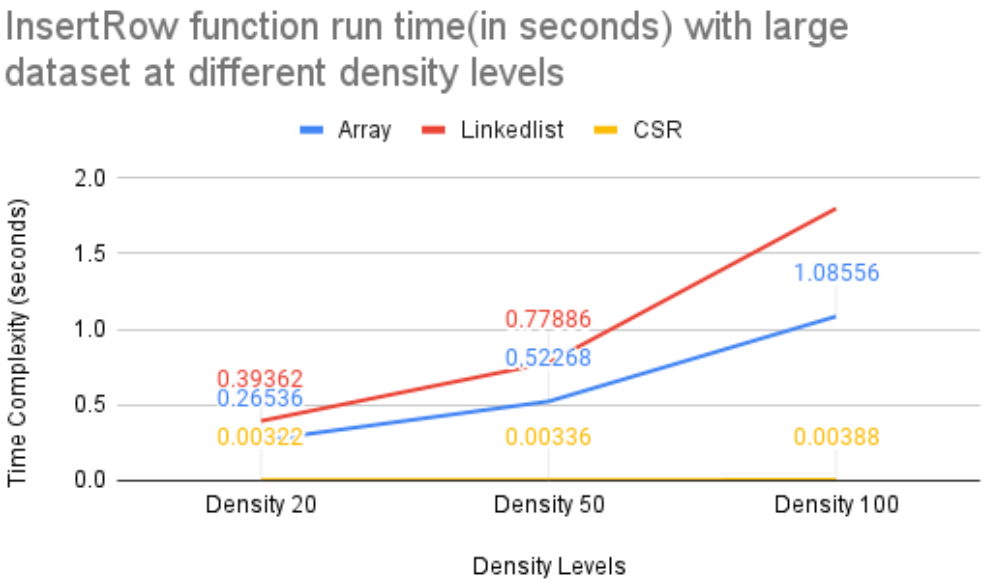
C.1 Small Dataset



C.2 Medium Dataset

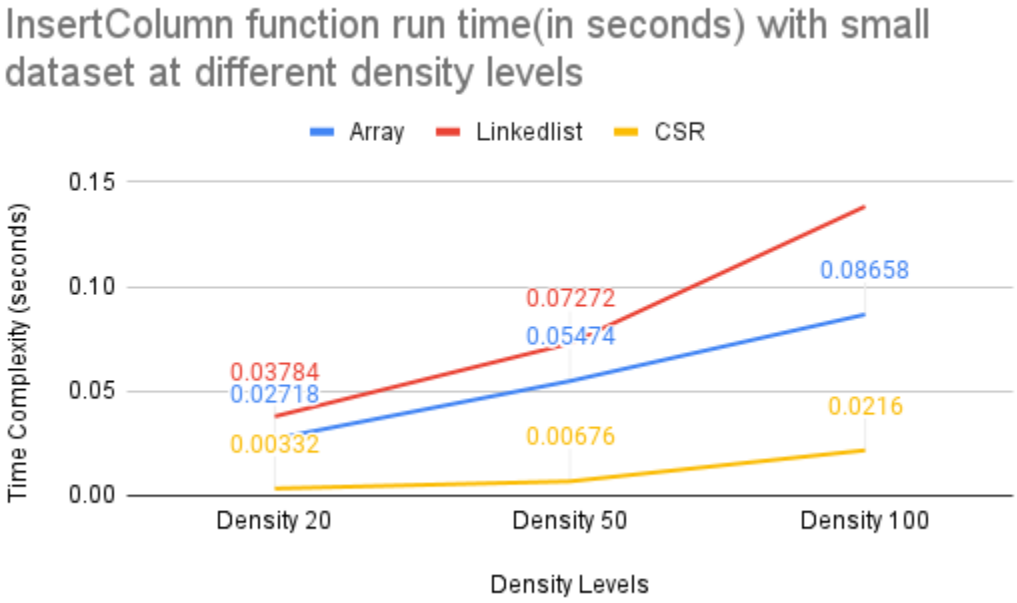


C.3 Large dataset

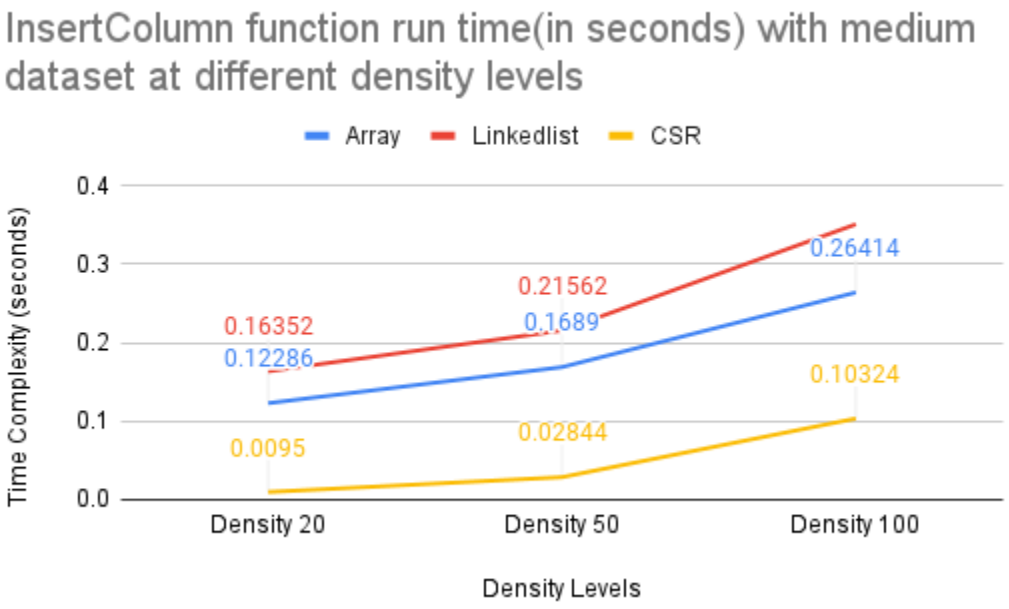


Appendix D - Insert Column Graphs

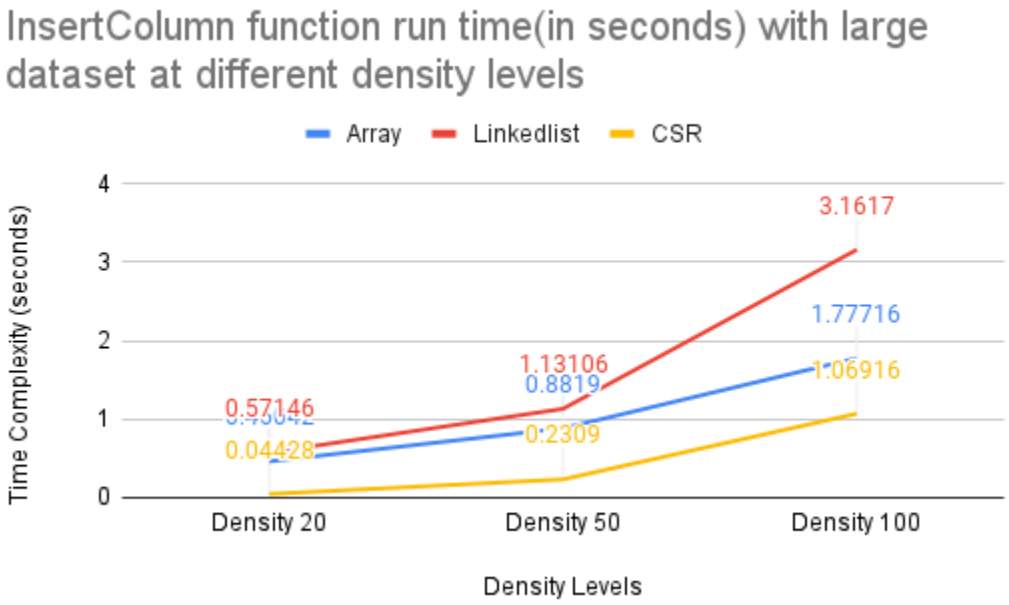
D.1 Small dataset



D.2 Medium dataset

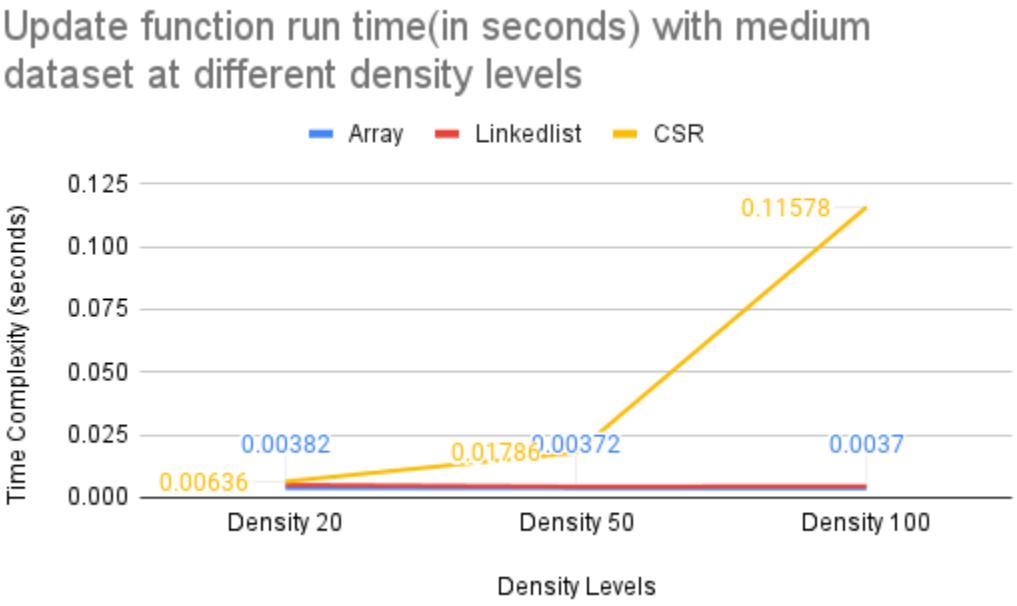


D.3 Large dataset

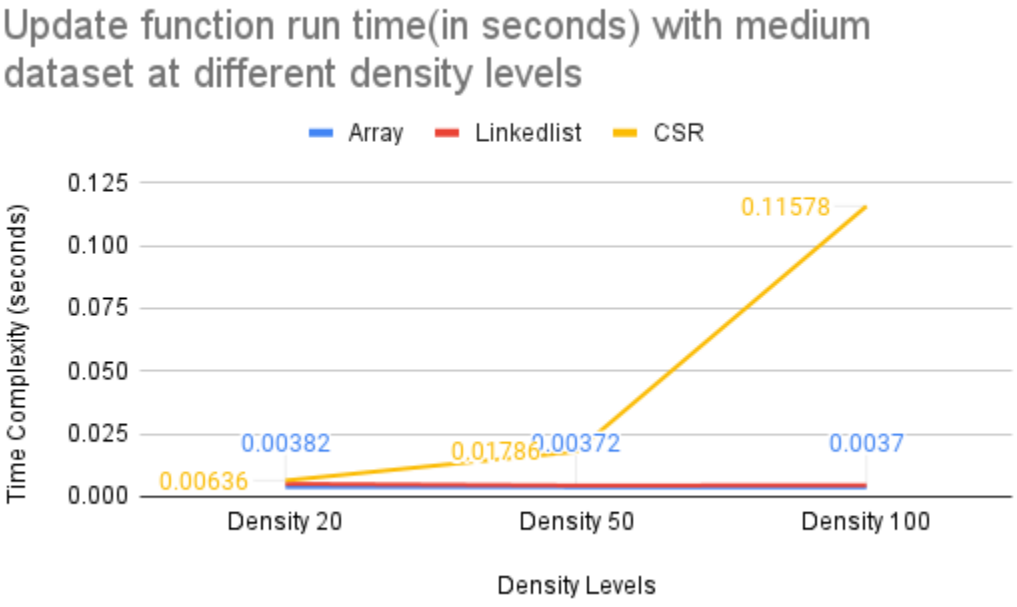


Appendix E - Update Graphs

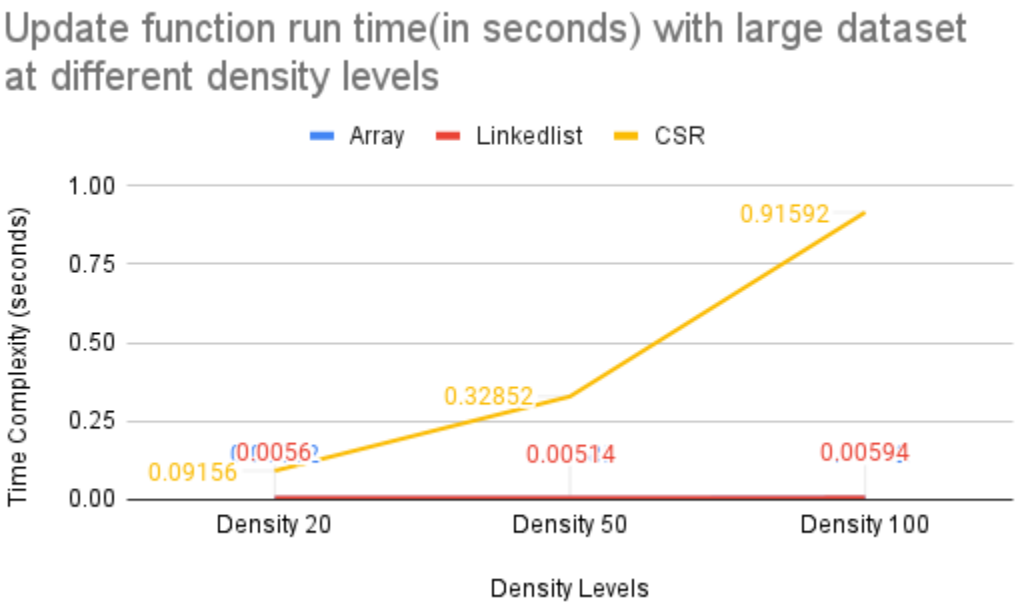
E.1 Small dataset



E.2 Medium dataset

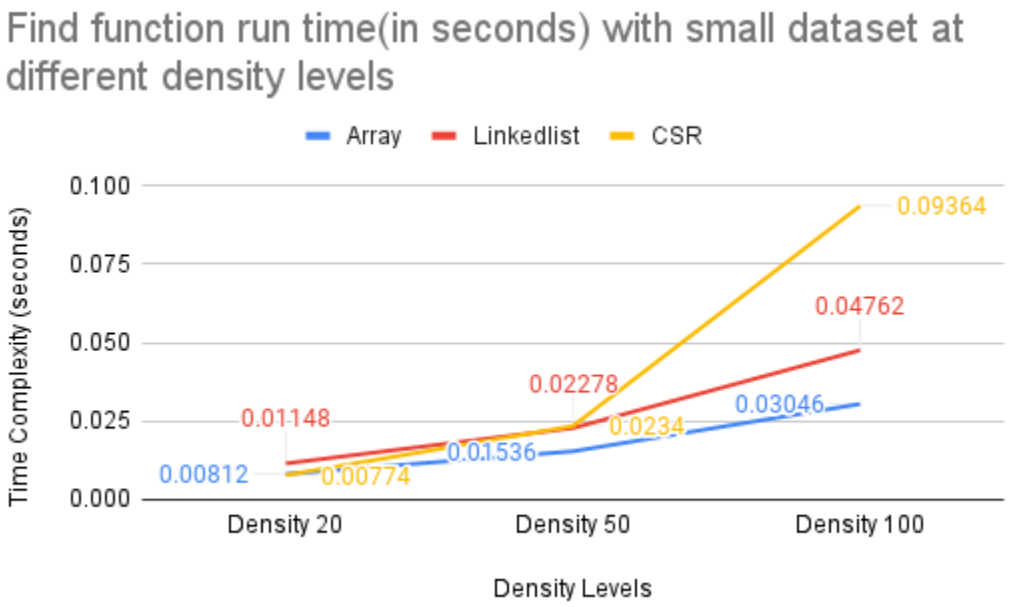


E.3 Large dataset

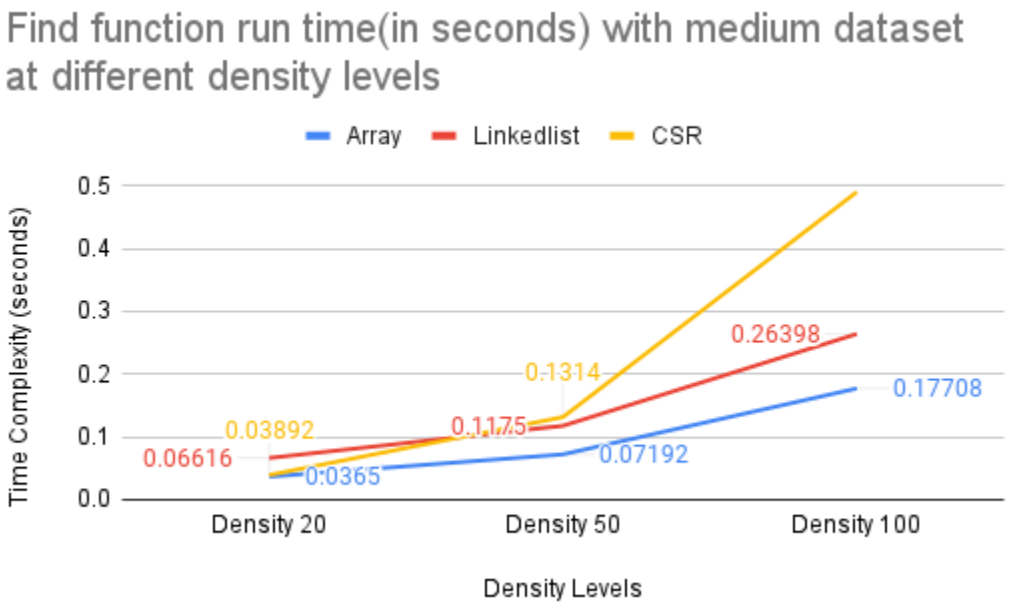


Appendix F Find Graphs

F.1 Small dataset



F.2 Medium dataset



F.3 Large dataset