Algorithms and Analysis
Assignment 1
James Plecher

Theoretical Analysis

Adjacency List

Function	Worst Case Time Complexity and Example	Best Case Time Complexity and Example
updateWall()	O(h+w) (linear) Case: Vertices are on opposite ends of the list	O(1) (constant) Case: When the nodes are next to each other in the list
neighbours()	O(height + width) (linear) Case: The node has maximum possible neighbours.	O(1) (constant) Case: When there are no neighbours so returns an empty list

Adjacency Matrix

Function	Worst Case Time Complexity and Example	Best Case Time Complexity and Example
updateWall()	O(1) (constant) Case: Wall is not adjacent	O(1) (constant) Case: Wall is adjacent
neighbours()	O(h * w)(linear) Case: Has maximum possible neighbours	O(1) (constant) Case: Has no neighbours

Data Generation:

Some data was generated using a random range number generator in python and updating a config.json file to place the exits and entrances.

The other changing parameters are the height and width of the maze. For the main tests, the mazes were kept square. Later testing implemented a random number generator to change the height and width randomly.

The following square maze size ranges were implemented for square mazes:

Small: 10x10. Medium: 50x50. Large: 100x100.

The following square maze size ranges were implemented for non-square mazes:

1x100. 100x1. 2x50. 50x2. 4x25. 25x4

I used three ranges for the square mazes to be able to adequately show the relationship of each functions time complexity on different data sizes, however after review I should have made more smaller size ranges, such as five in total as opposed to three to better display the information. This mistake was later learnt from and implemented for non-square mazes.

100x100 range was chosen as the maximum as with the high linear factor of the matrix neighbours() function, it would take too long to computer larger numbers. (over 24 seconds sometimes)

10x10 was chosen as the smallest as the minimum size must be at least 3, but for some leeway and diversity in the graph I opted for 10x10 being the smallest.

For the sake of formula simplicity, since it is relatively unimportant, the entrances can only appear on the bottom of the maze and exits on the right.

Experiment Set Up:

To set up the experiment I needed to test the run time for the neighbours() and updateWalls() functions, for both the adjacency list and matrix implementations.

This was done by getting the start and end time and subtracting the difference.

```
startGenTime : float = time.perf_counter()
##FUNCTION RUNS HERE###
endGenTime: float = time.perf_counter()
print(f'{endGenTime - startGenTime:0.10f}')
```

I would also count the amount of times both functions (neighbours(), updateWalls() would run throughout the maze generation process.

For extra data, I will loop the amount of times the maze was generated, to get 1016 records of each time recorded for each maze size range (small, medium, large), to later be able to get the average of these for a more accurate analysis.

To get the overall time for the maze generations, I will set up a for loop and run the maze generation on each size range one-hundred times and get the average.

The loops will be used in combination with the visualisations being turned off. This allows for mazes to be generated back to back, extremely efficiently, so I can take all the results from one execution.

The results will be logged into an excel file for further review as they are gathered.

Empirical Results

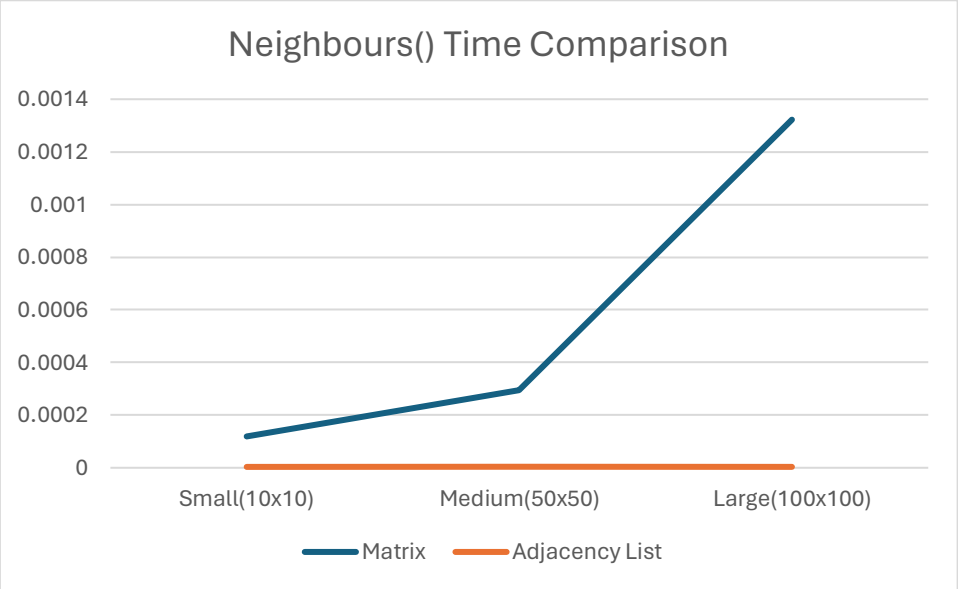

Figure 1.1* Above is results for neighbours() function times. Each range (small, medium, large) is an average of 1016 results. Y axis in units of seconds.

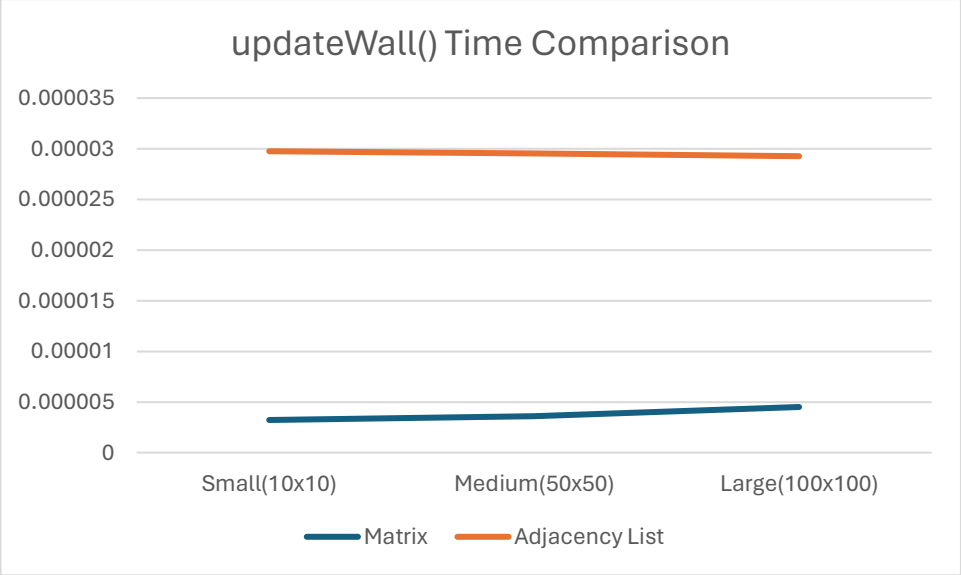

Figure 1.2* Above is results for updateWall() function times. Each range (small, medium, large) is an average of 1016 results. Y axis in units of seconds.

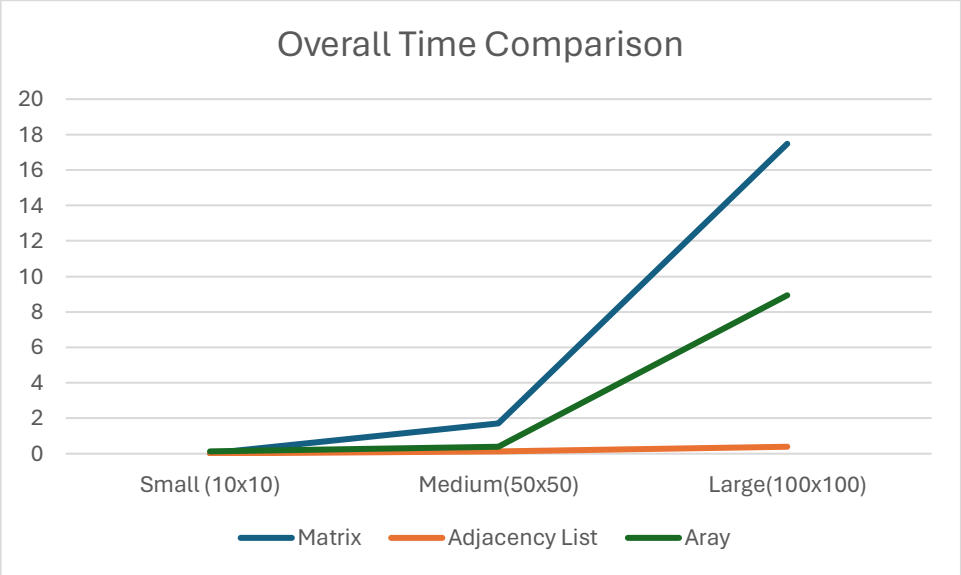

Figure 1.3* Above is results for overall maze generation times for each implementation. Each range (small, medium, large) is an average of 100 results. Y axis in units of seconds.

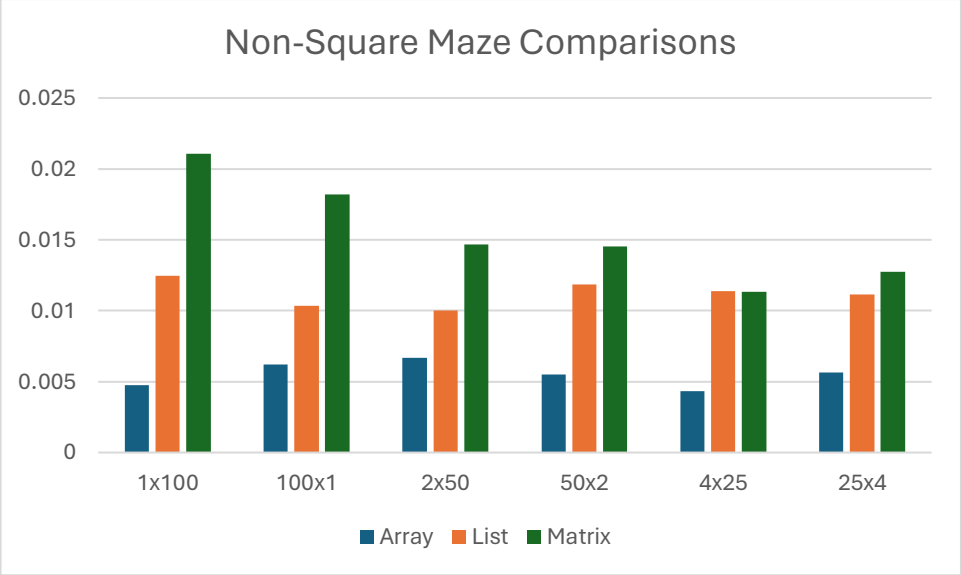

Figure 1.4* Results for non-square mazes. Each bar is an average for 100 runs each. Y axis in units of seconds.

Insights

Referencing figure 1.1*

For the **neighbours**() functions, it would appear the **adjacency list** is in constant time which is different to my theoretical analysis of linear time. My four theories on this difference is either the scale is too zoomed out with the inclusion of that it seems constant but could actually be linear. The issue is that the exact numbers of the three range averages, being 3.28378E-06 seconds (small), 3.89452E-06 seconds (medium), 3.45978E-06 seconds (large) are extremely close, practically identical in terms of time and there is no clear increasing linear pattern, making it seem unlikely. The second theory is I simply did not go to a high enough range to see the pattern. Another case could be that this is average case running time, which doesn't reflect many worst case run times to skew the data. Or lastly, my theoretical analysis prior to the real testing was incorrect. As theorised however, the **matrix** implementation seems to be in exponential time, tripling in time from small to large, and more than quadrupling from medium to large in time taken. This is one of the two instances where I wish I tested five ranges instead of three, to enhance the analysis and better show the pattern. This is because although the worst case is linear time, nowhere near every case will be worst case, so the average time will be different and more easily viewed on a different scale.

Referencing figure 1.2*

For the **updateWall**() function, both **matrix** and **adjacency list** seem to have a constant time complexity. Interestingly however, the matrix implementation for all ranges executes in roughly 5E-6 seconds, where as the adjacency list seem to execute in 3E-5 seconds. That is six fold slower than the matrix function. Whilst for a few executions that is negligent, the updateWall() function executes 10003 times on a 100x100 matrix, leading to an extra 0.30009 seconds execution time overall. Not significant on an arbitrarily small scale but as the scale exponentially increases this could be a significant factor. So whilst both implementations theoretically have a best case time

complexity of constant time as well as the worst case of adjacency list, the theorised worst case of linear time complexity seems to be balanced out by the best case, leaving to a seemingly constant time adjacency list.

Referencing figure 1.3*

For the overall time for the mazes to generate it appears that the **matrix** implementation is vastly inferior to the **adjacency list** in terms of time complexity. Whilst both started at almost the same time for the small range, being 0.014 seconds for the adjacency list and 0.017 seconds for matrix, adjacency linearly increases as the sizes increase, going to 0.132 seconds at medium and 0.392 at large ranges. As seen however, the matrix exponentially increases to a 17.480 second average execution time at the large ranges, revealing either a major inefficiency in the code I produced, or more likely, mirroring the "updateWall()" function growth time above and actually increasing at a higer degree linearly. The **array** approach seems to sit in the middle of these two methods for square arrays. This possibly hints at the adjacency list having the best neighbour retrieval method, with arrays not as efficient, and matrix's being diminished by their excessive storage demands.

Referencing figure 1.4*

For non-square mazes its slightly different to square mazes. Arrays are by far the best approach, being roughly two times faster than adjacency lists in every tested circumstance. This could be because in arrays, every cell accessed is using constant time, regardless of its position in the array. However, the adjacency list has to traverse multiple neighbours in its list to find the correct vertices leading to less efficiency when the maze is not a square and can have long lists.

In theory, the **matrix** implementation has potential to be more efficient than adjacency list, with some faster operations and capability of accessing edges in constant time leaving it with an overall time complexity of O(h * w). However the **adjacency** list simply outperforms the matrix in every regard empirically in reference to both square and non square mazes. Both expand in a linear time complexity, just with anything that's larger than a very small range, adjacency list will outperform the matrix maze implementations. Arguably though, the **array** implementation is more versatile, as it is still somewhat efficient at compared to the adjacency list method for square arrays, but non-square arrays is a different story, being drastically more efficient.

Overall depending on the use case, either can be utilised. If a maze is non-square, an array should be implemented, but if it is known in advance it will be close to square, it should probably be an adjacency list used.