Algorithms & Analysis
COSC2123

ASSIGNMENT 1:

IMPLEMENTING & EVALUATING DATA STRUCTURES FOR MAZE GENERATION

HARRIET MATHEW
s3933558

Theoretical Analysis

Adjacency List		
Operations	Best Case	Worst Case
updateWall()	- The best running time in terms of exact complexity of the updateWall() function is **Θ(1)** [constant time]. - The best case scenario will be when the vertices present in each other's list and the function updates the wall status. - This can be considered as the best case scenario as the vertices are already present, and we are just updating the wallStatus of the edge, which is an operation of constant time complexity. Since we are checking for a vertex present in the adjacency list of the other vertex, which is a dictionary, the lookup time will be of constant time complexity. This is because the average lookup time for a dictionary is of **Θ(1)** especially when the hash function is well designed. - As a result of all these operations, best case running time will result in constant time complexity **Θ(1)**.	- The worst running time in terms of exact complexity of the updateWall() function is also **Θ(1)** [constant time]. - The worst case scenario will be when the vertices are not present in the lists. - The worst case can also be thought as **Θ(n)** with the assumption that the function will iterate through all the values to find the particular vertex, and the worst case lookup time for a dictionary if it's hash function is poorly designed is also **Θ(n).** - But in a 2-D maze generated by an adjacency list, each vertex will have only a maximum of 4 neighbours (above,below,right,left) as we are checking the adjacency list of the particular vertex. Since there is a limit on the size as well, which is 4, we can say that this is of constant time as there will not be a scenario where size is greater than 4. - Therefore the worst case scenario of the updateWall() will also be of constant time complexity **Θ(1)**.
neighbours()	- The best running time in terms of exact complexity of the neighbours() function is **Θ(1)** [constant time]. - The reason why the best case scenario will be constant time complexity is because the neighbours function of adjacency list just returns a list of neighbours from the particular label and this function is independent of the input size. - Even if the label has a small number of neighbours, it would still not have much effect, and return the list in constant time. - Therefore, this also will result in constant time complexity **Θ(1)**	- The worst running time in terms of exact complexity of the updateWall() function is also **Θ(1)** [constant time]. - The reason why the worst case scenario is also constant time is because the function simply returns the list of neighbours from the particular label and takes the same time to execute. It is not dependent on the size or the number of neighbours of the label. Even if the label has a large number of neighbours, the neighbours() function would still return the list of neighbours in constant time. - Hence, this will result in constant time complexity **Θ(1)**.

Adjacency Matrix		
Operations	Best Case	Worst Case
updateWall()	- The best running time in terms of exact complexity of the updateWall() function is **Θ(1)** [constant time]. - The best case scenario will be when the vertices are present first and adjacent, then the status of the wall is updated. - This can be considered as the best case scenario as the function only needs to perform the lookups at the first element and update the wall status, both of which are constant time. The **isAdjacent()** function present in updateWall() is also of constant time as it just has a conditional check for adjacency which is independent of the input size. The **getRow()** and **getCol()** present in isAdjacent() function just return the row/column and hence are also of constant time complexity. Since we are checking for a vertex present in the list of the other vertex, which is a dictionary, the lookup time will be of constant time complexity taking a scenario where the hash function is well designed. This is because the average lookup time for a dictionary is of **Θ(1)** as the chances of encountering a poorly designed hash function is low. - As all these operations are of constant time complexity, this will also result in constant time complexity **Θ(1)**.	- The worst running time in terms of exact complexity of the updateWall() function is **Θ(n)** [linear time]. - The worst case scenario can be the case when the vertices are not present. - The worst case can be assumed as **Θ(1)** because we are checking if a vertex is present list of the other vertex, which are represented as dictionaries in the adjacency matrix. The average lookup time for a dictionary is **Θ(1),** however, if the hash function is poorly designed, the lookup time will be of linear time complexity **Θ(n).** Furthermore, we are checking through the entire list where vertices that are not neighbours can also exist as this is the case of adjacency matrix. This also results in **Θ(n)** complexity. - So the worst case scenario will be **Θ(n)** as it is dependant on the way the hash function is designed, and also on the number of vertices that are present in the list.
neighbours()	- The best running time in terms of exact complexity of the neighbours() function is **Θ(n)** [linear]. - The best case scenario is when the function doesn't iterate or only iterates once to find the neighbours of the particular label. - This can be considered as the best case scenario as it does not need to go through a lot of elements. We can	- The worst running time in terms of exact complexity of the neighbours() function is also **Θ(n)** [linear]. - The worst case scenario is when the function iterates through all the possible vertices and has to append all the neighbours of the particular label. - The reason why this will be considered as the worst case

	assume that it may be **Θ(1)** if it has no neighbours or has to only check and append only one element. However, it has to iterate through the list which is dependent on the size that is not static as there can exist elements which are not neighbours too. The **append()** function is of constant time complexity, so even if the existing element has to be appended, it would not have an effect on the complexity. - This results in linear time complexity **Θ(n)**, where n is the number of elements in self.matrix[label], because the function iterates depending on the size of the data structure, and the input is not static.	scenario is because it has to go through all the vertices. Suppose 'n' is the amount of elements in the particular label, it has to go through all 'n' elements in the label to find out the neighbours of the label. The **append()** function which adds the neighbour to the neighbour list will not have an effect as it is of constant time complexity. - Since this function is dependent on the size of the data structure and since linear time is more dominant than constant time complexity, we can conclude that this will result in linear time complexity **Θ(n)**

Experimental Setup

In the experimental setup for my assignment, I measured the time taken for each data structure for a range of maze dimensions (n*n and n*m) and for mazes with different number of entrances and exits to ensure that there is a fair comparison. In order to measure its performance accurately, I generated the maze manually ten times for each maze dimension and then averaged the time it took for each dimension.
This method gives me a proper understanding of how long the program and operation takes to generate mazes of different dimensions and shapes. By doing the process of averaging the timings over multiple runs, I can get a more accurate estimate of the data structure's efficiency. To obtain the running times, I just had to analyse the running time generated by the maze generation, which had been already given in the starter code.

Data Generation

The data used for the experiment included mazes of various dimensions, starting from small ones like of 5*5 up to larger ones like 150*150 grids. The idea behind choosing a range of sizes is to test how well each data structure works for different dimensions and the scalability of the data structures. The various dimensions that were included are:

- 5*5
- 10*10
- 20*20
- 30*30
- 50*50
- 60*60
- 75*75
- 90*90
- 100*100
- 150*150

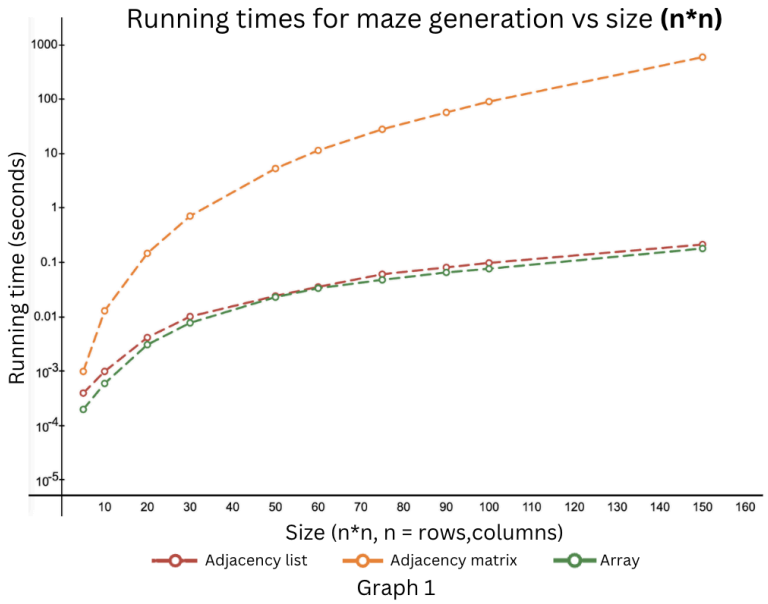
In addition to having square mazes (n*n, rows and columns are equal), I had also tested cases of rectangular mazes (n*m, rows and columns were different) with varying maze shapes, based on a fixed size (250 cells). Analysing such rectangular mazes showed the robustness of the data structures and to what range can the data structures perform really well. These were the following maze rectangular maze shapes:

- 1*250
- 2*125

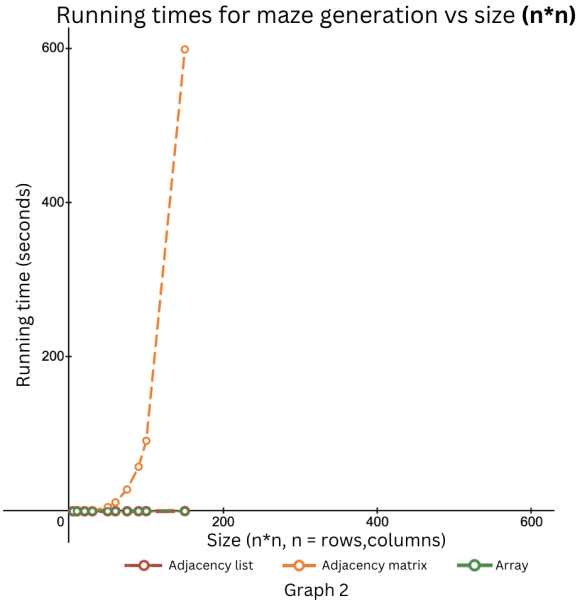
- 4*50
- 25*10
- 125*2
- 10*25
- 50*4
- 250*1

Apart from these, I had also tested cases of a square maze (50*50) with different number of entrances and exits from just 1 entrance and exit upto 50 entrances and exits which was also done manually.
By testing out various shapes of mazes, I could see how efficient the data structures work for different maze layouts. The running times that I had obtained were put in the form of .csv files, where each row specifies the running time taken by each data structure. I had created multiple config files for each data structure with different maze dimensions where each config file's name specifies the data structure and the maze dimension (<datastructure> <dimension>.json). For mazes with different number of entrances and exits, I created config files with the data structures, maze dimensions and entrances, exits present in the form of <datastructure><dimension>[<entrances,exits>].json

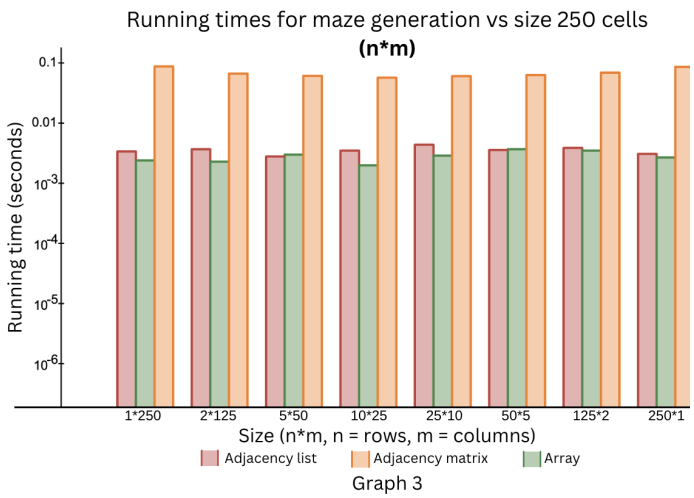
Empirical Analysis



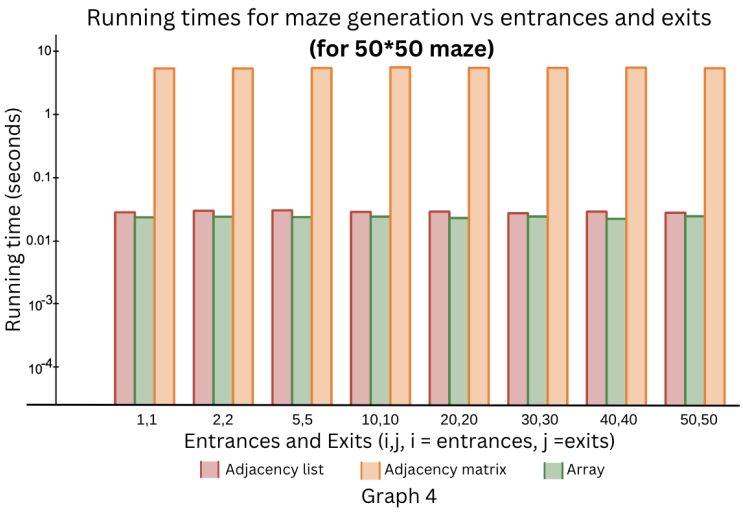
Graph 1



Graph 2



Graph 3



Graph 4

Insights

From graph 1, we see the results from the empirical analysis over a range of dimensions for square mazes for three different data structures. The results showed that as the size of the maze increases, the time required for maze generation also grew proportionally for all three data structures which was predictable. However, the Array data structure consistently performed faster with Adjacency List closely following it, while the Adjacency Matrix was consistently performing slower as the size was increasing. The slow performance of Adjacency Matrix can be explained by its complexity class which is quadratic $\Theta(n^2)$, which was proved by the empirical analysis (also in graph 2 [linear scale]) and also the theoretical analysis of the functions such as neighbours() which has a complexity class of Θ(n) and other functions as well that contribute to the total complexity class (included in appendix). Meanwhile, the fast performance of Array and Adjacency Matrix can be proved because of their complexity class being constant Θ(1), which was again proved by the empirical analysis from graph 1 and graph 2. This was also evident with the theoretical analysis of functions neighbours(), updateWall() and other functions which were of constant time complexity.

From graph 3, I measured the running time over the same maze dimension (250 cells) but generating different types of rectangular mazes. The graph shows us that changes in the dimensions with the same size did not have much effect on the performance.
From graph 4, we can see even the number of entrances or exits do not play a huge part in the maze generation performance. This analysis again proves that as maze size increases, this will play a huge part in the performance of the maze generation for all three data structures, no matter what the number of entrances,exits or the ratio of rows and columns are.

Conclusions/Recommendations

From the above empirical and theoretical analysis, we can clearly see that both array and adjacency list consistently outperformed adjacency matrix as the size of the maze increased. These results show the importance of taking into account the scalability of data structures when working with larger maze sizes. The analysis of rectangular mazes with the same number of cells with different dimensions and mazes with different number of entrances and exits showed minimal variation in performance. This shows that the data structures are able to adapt to a range of maze configurations.

It is always important to consider the complexity classes especially when selecting data structures as the observed performance aligns with the theoretical and empirical analysis. I would recommend using an array or adjacency list for maze generation projects, especially in situations when we're dealing with large maze sizes. These data structures have better adaptability and efficiency compared to adjacency matrix. However, we should also consider other factors like the requirements and the operations that should be performed. In situations where we have a static graph, we can consider cases of adjacency matrix but in cases of dynamic graph, adjacency list or array is the best data structure to consider as they are more efficient with larger maze dimensions than adjacency matrix.

APPENDIX

Adjacency List		
Operations	Best Case	Worst Case
addVertices()	- The best running time in terms of exact complexity of the addVertices() function is **Θ(n)** [linear time]. - The best case scenario will be when there is only one vertex present in the list. - This can be considered as the best case because the for loop will only be executed once as there is only one vertex present in the list and this will result in the least running time as it does not have to add a lot of vertices. The addVertex() function of constant time complexity will not have an impact on the total time complexity. - As a result, the best case running time complexity for the addVertices() function will be **Θ(n)** - This however will not make the total complexity class of **Adjacency List** as **Θ(n)** as the class is more dominated by functions which are of constant time complexity.	- The worst running time in terms of exact complexity of the addVertices() function is also **Θ(n)** [linear time]. - The worst case scenario can be the case when a lot of vertices are present. - This can be considered as the worst case because with having a number of vertices, the for loop will have a complexity of linear time **Θ(n)** as it will have to add each vertex into the list. The addVertex() function that is present in the for loop is only of constant time complexity and will not have an effect in the total complexity class as the for loop is of **Θ(n)**. - As a result of all these operations, we can conclude that the worst case running time of linear time complexity **Θ(n).** - This however will not make the total complexity class of **Adjacency List** as **Θ(n)** as the class is more dominated by functions which are of constant time complexity.

Adjacency Matrix		
Operations	Best Case	Worst Case
addVertices()	- The best running time in terms of exact complexity of the addVertices() function is **Θ(n^2)** [quadratic]. - The best case scenario will be when there is only one vertex present in the list. - This can be considered as the best case because the for loops	- The worst running time in terms of exact complexity of the addVertices() function is also **Θ(n^2)** [quadratic]. - The worst case scenario can be the case when a lot of vertices are present. - This can be considered as the worst case because with having a

	will only be executed once as there is only one vertex present in the list. The first for loop will have a complexity of linear time **Θ(n)** as it has to iterate through the list of vertLabels**.** The addVertex() function which is present in the first function of constant time complexity and will not have an effect on the total time complexity. - The nested for loop will also have a time complexity of linear time **Θ(n)** as this loop also has to iterate through the list of vertLabels again. - When combining the complexities of both for loops, we get the best case running time complexity of the addVertices() function as quadratic time complexity **Θ(**n^2**).**	number of vertices, the first for loop will have a complexity of linear time **Θ(n)**. The addVertex() function that is present in the first for loop is only of constant time complexity and will not have an effect in the total complexity class. As there is a nested for loop, that will also have a complexity of linear time **Θ(n).** - When combining both the complexities of the for loops, we get the total running time complexity for the worst case scenario as **Θ(**n^2**)**