Assignment 1

Benjamin Hatfield

S3980858

Algorithms and Analysis

RMIT University

2024

Theoretical Analysis:

The following table outlines the theoretical analysis of the two methods updateWall() and neighbours() for both the Adjacency matrix and adjacency list implementations. Variables h and w are used to represent height and width of the maze being generated (border cells included).

Operations	Best Case	Worst Case
Adjacency Matrix updateWall()	**Asymptotic Complexity:** O(1) **Explanation:** Update wall takes in two coordinate values and a Boolean value. The function first checks whether an edge is present between the two coordinates by using three dictionary lookups. Dictionary lookups have an average time of O(1). The best case scenario would be when there is no edge present between the specified coordinates so no other computations would be completed other than returning false which is again a constant time function. Thus the best case asymptotic complexity would be O(1).	**Asymptotic Complexity:** O($h^4 + w^4$) **Explanation:** The initial procedure for the update wall function is the same for the worst case, however hash collisions must be considered for a poorly design hash function. If we consider that our graph is a 2D dictionary of h+w rows and h+w columns and the vertices dictionary is also h+w long. When hash collisions are considered (which are very rare) a worst-case time can be O(n). When translated to our function this will be O((h+w)(h+w)(h+w)(h+w)) for the four dictionary lookups being performed. When an edge has been found between the two coordinates, an assignment of the given boolean will be made to the respective coordinates. This will then add the same time complexity to the equation two more times. Resulting in O($3h^4 + 3w^4$) which can be simplified as O($h^4 + w^4$)
Adjacency Matrix neighbours()	**Asymptotic Complexity:** O(h + w) **Explanation:** The Neighbours function must iterate through the vertices dictionary that is h+w long. For each of the values in the dictionary the same lookup that hasEdge uses is called again. As mentioned above the average case complexity for this lookup will be O(1) when the hash function is well designed. The result of the append function will always be constant time, thus giving the result O(h+w).	**Asymptotic Complexity:** O($h^5 + w^5$) **Explanation:** Similar to the best case, the function will always iterate through the entire vertices dictionary h+w long. However when we take into consideration a poorly designed hash function, the dictionary lookup performed by hasEdge will have O((h+w)(h+w)(h+w)(h+w)). Due to this action being performed on every coordinate in the vertices dictionary we end up with O(((h+w)(h+w)(h+w)(h+w)(h+w)) resulting in O($h^5 + w^5$)

Adjacency List updateWall()	**Asymptotic Complexity:** O(1) **Explanation:** This function calls the hasEdge() function which looks for both of the keys within each other's respective dictionaries, dictionary lookups have a best case constant time. We then modify the value of two dictionary values at the specified key, which also uses constant time for best case. Therefore, the overall best case for the function is constant time.	**Asymptotic Complexity:** O(h+w) **Explanation:** This function uses 2 dictionary lookups of a dictionary h+w long. Furthermore two assignments are completed that move into the 2D nature of the graph dictionary. Each of these dictionaries have h+w rows and 4 columns at the longest due to each vertex having a max of 4 neighbours. Therefore the worst case complexity of the function is O(4(h + w)) simplified to O(h+w)
Adjacency List neighbours()	**Asymptotic Complexity:** O(1) **Explanation:** This function iterates through the specified vertices edges. Due to the nature of each vertex having a max of 4 neighbours, the function will execute in constant time O(1).	**Asymptotic Complexity:** O(1) **Explanation:** Worst case for this method is would be O(n) but is still constant time as the edge count (n) cannot exceed 4 for each vertex.

Experimental/Empirical analysis

The approach taken for this experiment was to test each of the maze dimensions mentioned below 10 times, averaging the results across these individual tests. This accounted for any variance across each algorithm. The dimensions that were used for testing were 1x1, 5x5, 10x10, 20x20, 30x30, 40x40, 50x50, 75x75 and 100x100 for square mazes. The data generator then checked cases where absolute cell count was static, but shape of the maze was altered so that it could be determined whether shape of the maze would impact upon execution time of each build. The cell count for each of these tests was 100 cells.

The rationale for selecting these sizes was to ensure that a thorough analysis of each datatype could be performed, and behavior observed across small, medium and large datasets. Irregular shapes were also included to discover whether shape would alter each datatype's behavior. This would allow for accurate evaluation of results to be performed enabling and confident recommendations to be given for the best use case of each datatype.

Data was generated using a modified version of the mazeTester.py file that would iterate through the specified datatype and requested dimensions ten times and return the average result of the time taken to execute each task. The program would then do this for each datatype before editing the dimension values so that the rows and columns incremented to the next required size mentioned above.

The approach taken to measure timing results was measuring the time for each data type to be fully built (all vertices added, all edges added) for each of the dimensions listed above. This approach was then repeated ten times for each set of dimensions with the times being averaged across these results to account for variances.

Evaluation of experimental results

	1x1	5x5	10x10	20x20	30x30	40x40	50x50	75x75	100x100
2DArr	0.0001	0.0024	0.0085	0.0308	0.0634	0.1084	0.1734	0.3840	0.2067
AdjList	0.0002	0.0036	0.0131	0.0461	0.0975	0.1792	0.2768	0.6323	0.3562
AdjMat	0.0002	0.0061	0.0418	0.4243	1.9009	5.7121	13.358	51.972	198.89

Figure 1 *Provided to observe smaller variances in adjlist and 2darr types*

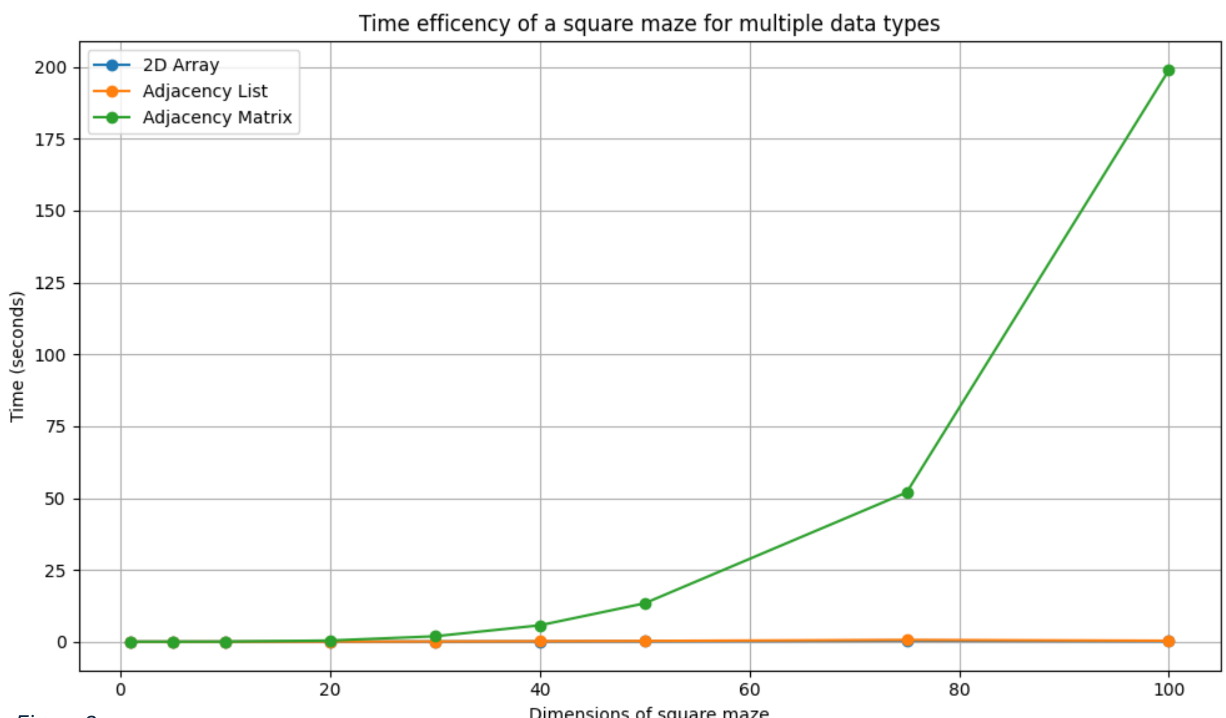

Figure 2

From the experimental results shown in figures 1, 2 and 3, it can be established that, as maze size increases, all maze construction time increases. As expected, the rate of this increase is vastly larger for the adjacency matrix datatype. This divergence can be observed as early as the results for the 5x5 where the adjacency matrix has taken twice the execution time than the 2D array and adjacency list. This trend continues to grow exponentially as the maze size increases, eventually leading to the adjacency matrix taking almost 200 times the execution time of the other datatypes.

The reasoning behind this large difference in time is largely due to the organization of the datatypes. The adjacency list will only create dictionary values for existing edges (a maximum of 4 per vertex), rather than every possible edge like in the adjacency matrix (resulting in $(h + w)^2$ edges in every instance). This means that the storage requirement for the adjacency matrix is much higher than that of the adjacency list and that if the matrix is ever iterated through, it will have $(h + w)^2$ edges to iterate through rather than 4(h+w) in the adjacency list.

This can be further examined with assistance from the theoretical analysis of the neighbours function within both the adjacency matrix and adjacency list. In a best case scenario when hash collisions are completely avoided, the matrix must iterate through h+w vertices to find the initial vertices neighbours. This is then compared to the adjacency list implementation only being required to iterate through a max of 4 values, regardless of how large the maze dimensions are. The exponential growth of the matrix datatype can be associated with the initiation of the 2D dictionary (graph) within the addEdge function. This is the only function within the adjacency matrix datatype implementation that requires $O((h + w)^2)$ best case time complexity on its first call, which is the largest within all the datatypes. The function requires this amount of time due the process of instantiating a (h+w) row and (h+w) column graph and setting each value to a placeholder prior to edges being added.

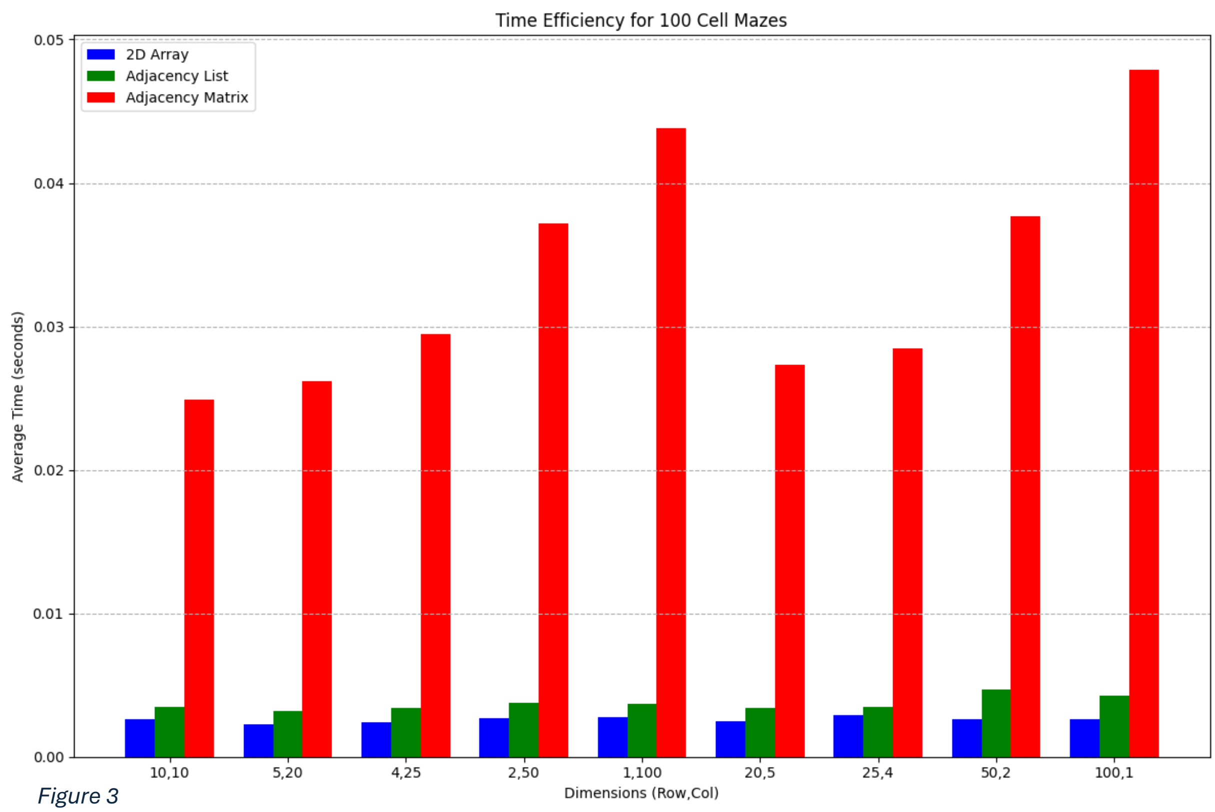

Figure 3

Furthering this we can see in figure 3, that altering the shape of the maze structure has very little effect on the execution time of the 2D array and adjacency list. However, as shown in figure 3, the larger difference in row to column count creates a noticeable increase in the execution time of the adjacency matrix. This is likely due to the nature of each maze having a one cell boarder surrounding the entire maze. As the columns and rows differ to a larger extent the number of cells required to create this boarder also increases with regard to the cell count inside the maze structure staying static. As we know from the above analysis, when a single vertex is added to the matrix, all possible vertices are added as edges to the graph dictionary. This means that the matrix requires much more space for the same internal cell count maze as the shape becomes more rectangular. 10x10 maze would require (12x12-4) or 140 cells/vertices with borders included where as a 1x100 maze would require (3x102-4) cells/vertices or 302 cells which is drastically more storage space for the matrix that will need to create a 302x302 sized matrix to store each possible edge.

Recommendations for future use

As mentioned above, when the maze dimensions are smaller than 10x10 the difference of each approach is quite small and therefore any of the three data types will work effectively to implement a maze of this size. However, as the maze dimensions grow, due to each vertex's nature only having a max of four edges, the adjacency list becomes the more efficient datatype to build and implement. Therefore, it would be my recommendation that the adjacency list be used for any maze above 10x10 or 100 cells.

However, there were a case in which each vertex could have up to n edges (not applicable in this scenario), then the adjacency matrix would definitely become the more efficient data type, due to its faster ability to look up elements from the matrix, as the build time for the adjacency list would become a similar size to the matrix.

It is also worth noting that, as mentioned within the analysis, the most time hungry algorithm of the matrix datatype is when the first edge is added to the maze structure, as this creates the (h+w) x (h+w) graph. So, if there were a case in which the maze structure was only to be built once, (i.e. the dimensions never changed) and this structure was reused for multiple different mazes. Then the datatype and it’s functions would be much faster in their execution times with O(h+w) being the worst of the best case time complexities within the functions of the matrix.

References:

Levitin, A. (2011). *Introduction to the design & analysis of algorithms*. Pearson Education.

MIT. (2023). *OpenDSA Data Structures and Algorithms Modules Collection*. Opendsa-Server.cs.vt.edu. https://opendsa-server.cs.vt.edu/OpenDSA/Books/Everything/html/index.html