Theoretical Analysis

Adjacency List

Operations	Best Case	Worst Case
updateWall()	Asymptotic complexity: O(1) When the vertices are adjacent but there is no change in status of wall. The method will only need to perform a constant number of operations, which will take a constant amount of time, therefore the asymptotic complexity is O(1)	Asymptotic complexity: O(n) where n is the number of vertices adjacent to the given vertices vert1 and vert 2 When removing a wall between two adjacent vertices, we need to search through the adjacency list of each vertex to find and remove the other vertex. If the vertex we want to remove is at the end of the list, we have to go through all the vertices in the list one by one until we find it. This could take a long time if the list is large, so the worst-case time it could take is O(n), where n is the number of vertices in the list. Since we perform this search operation twice (once for each vertex), the total worst-case time complexity becomes O(n).
neighbours()	Asymptotic complexity: O(1). This is because if we have a small number of neighbors relative to the total number of vertices, we can quickly check each neighbor's adjacency status without iterating over a large list. In the best case, we might find the neighboring vertices directly adjacent to the given vertex without needing to search through the entire adjacency list. Thus, the time complexity would be constant, regardless of the size of the graph.	Asymptotic complexity: O(n). This will happen if every vertex in the graph is connected to every other vertex. When checking for neighbours in a particular vertex, the code will have to check all vertices in the graph to see if they are connected to given vertex. Which means the code might potentially have to iterate through all the vertices in the graph.

Adjacency Matrix

Operations	Best Case	Worst Case
updateWall()	Asymptotic complexity: O(1) When the vertices are adjacent but the wall status does not need changes. The method will simply check if the corresponding entry in the matrix indicates an edge between the vertice	Asymptotic complexity: O(n) The method will need to go through the entire row and column corresponding to the vertices being updated to toggle the wall status. This results in the time complexity being linearly proportional to the number of vertices in the graph.
neighbours()	Asymptotic complexity: O(1) When the vertex has no neighbours, its corresponding row or column in the matrix contains all zeroes, which indicates no edges incident to that vertex. In this case, the method can quickly determine that there are no neighbours without having to iterate over any elements in the matrix, which results in a constant time complexity.	Asymptotic complexity: O(n) The method needs to iterate over the entire column or row to find the neighbours of a vertex. Which makes the time complexity linearly proportional to the number of vertices.

Empirical Analysis

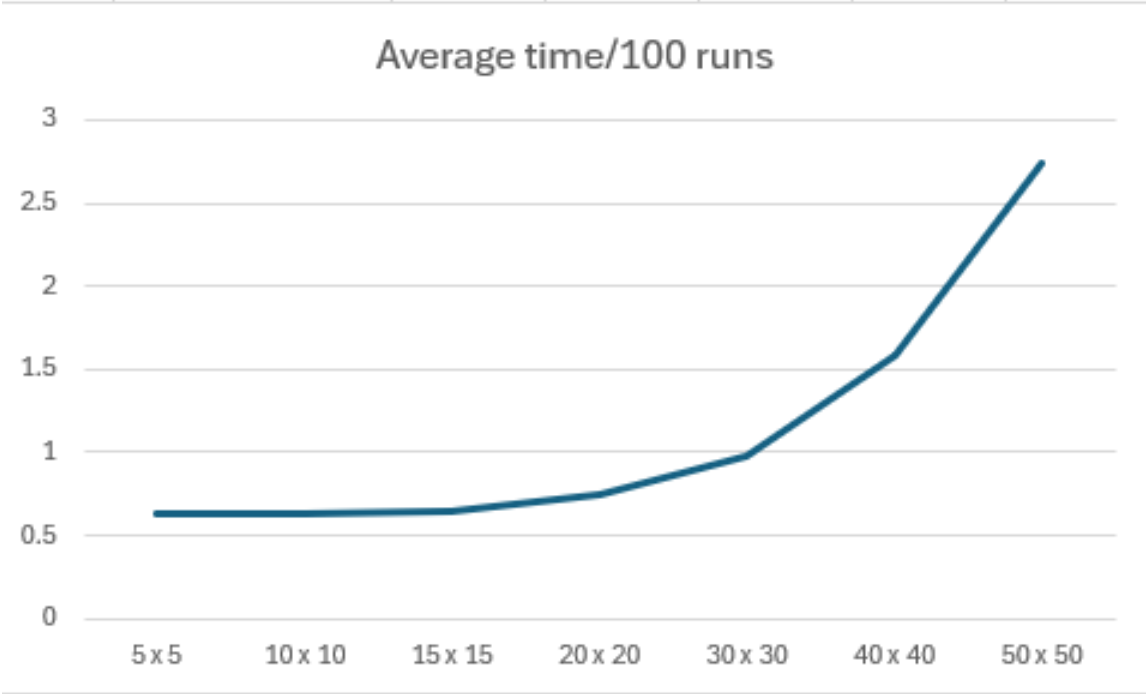
Data Generation

The data to be used for the experiment will increase values in the columns and rows from 5 to 50. This range of rows and columns was selected because I wanted to test the maze with small/minimal input, and it goes up to 50 x 50 because I wanted to assess the scalability and performance difference when generating more extensive mazes. The entrances and exits coordinates will be kept constant throughout the testing. The data generated will be in the form of a txt file named “data”.

Experimental Setup

In this experiment, we measure the time taken for each maze to be generated 100 times for each of the 3 different maze sizes. To obtain these run times, we will be using the python time.time() function. We then take the average of these run times and use them to compare the times it takes to generate different sized mazes.

Adjacency List



Analysis

The data collected shows that as the size of the maze increases, the average time taken to generate the maze also increases. This indicates that there is a positive correlation between the maze sizes and the time complexity of maze generation.

When analysing the varying sizes of mazes, we see the following trends:

1. Linear Trend: As the size of the mazes increases from 5x5 to 50x50, there is a consistent linear trend in the average time required for maze generation. This suggests that the time complexity of maze generation algorithms may exhibit linearity or close to linearity concerning maze size.

2. Acceleration in Growth Rate: With the increase in maze size, the rate of growth in the average generation time also accelerates. For instance, generating a 40x40 maze takes significantly more time than a 30x30 maze, and a 50x50 maze takes even longer.

3. Concerns about Scalability: The data underscores potential scalability issues when generating large mazes. The average generation time for a 50x50 maze is notably higher than for smaller mazes, implying that the algorithm's performance might deteriorate rapidly with larger maze sizes.

Considering these results, we might need to enhance the maze generation algorithm or investigate alternative methods to enhance scalability and lower the time complexity of creating larger mazes.