COSC2123 Assignment 1

Thanawat Neamboonnum (3958798)

Theoretical Analysis for "Adjacency List Graph"

Operations	Best Case	Worst Case
updateWall()	O(1) In a maze with few walls, the best case occurs when the edge to update is directly accessible at the start of the vertex's adjacency list. Accessing the head of a list is a constant-time operation.	O(n) In a densely connected maze, updating wall at the end of a long adjacency list can necessitate traversing the entire list. The computational complexity of the operation increase proportionally with the quantity of edges, denoted as E, that are connected to the vertex.
neighbours()	O(1) Accessing the adjacent cells of a conner cell within a sparse maze characterized by a minimal adjacency list. In addition, direct access to a small list is a constant-time operation.	O(n) Obtaining the neighbours for a highly connected cell in the centre of a dense maze requires traversing numerous edges. The operation scales with the degree of the vertex, which can be large in complex maze.

Theoretical Analysis for "Adjacency Matrix Graph"

The adjMatGraph utilizes a matrix to represent the maze, where each cell (matrix[i][j]) indicates whether there is a wall (1) or no wall (0) between vertex i and vertex j.

Operations	Best Case	Worst Case
updateWall()	O(1) In a maze with few walls, the best case occurs when the edge to update is directly accessible at the start of the vertex's adjacency list. Accessing the head of a list is a constant-time operation.	O(n) In a densely connected maze, updating wall at the end of a long adjacency list can necessitate traversing the entire list. The computational complexity of the operation increase proportionally with the quantity of edges, denoted as E, that are connected to the vertex.
neighbours()	O(n) Finding neighbours in a row for a cell located at one end of the maze, where 'n' is the total number of vertices (assuming `n = h * w` for a maze). Even if the first element checked form an edge (neighbour), it must still check every other element (vertex) to determine the presence of additional edges.	O(n) It is also O(n), for the same reason as the base case. Regardless of the edge distribution, every other vertex must be checked to accurately determine all neighbours.

Algorithms and Analysis: Assessment Task 1:

Thanawat Neamboonnum (s3958798)

Data Generation

To ensure thorough evaluation of the adjListGraph and adjMatGraph data structures, I adopted a strategic approach to test them across various maze configurations. Instead of limiting my tests to a handful of static mazes, which might not provide comprehensive insights, I developed a Python script to generate maze configuration files on demand. This script not only standardizes my test but also scales the process efficiently.

With the script, I generated mazes of varying sizes:

- Small Mazes (5x5): These are crucial for testing basic functionalities and the data structures' ability to handle straightforward operations swiftly.
- Medium Maze (10x10): Representative of typical use cases, allowing me to gauge performance under common conditions.
- Large Maze (50x50) and Extra-Large Mazes (70x100): I also tested and designed to test the limits of the data structures; these mazes help me assess their ultimate capabilities.

I do not just test each configuration once. Instead, I generate several mazes with the same parameters and average the results to minimize outliers and provide a clear, consistent evaluation of performance. Moreover, this methodical approach the data generation is pivotal for deriving meaningful, actionable insight form my tests, offering the flexibility to rigorously evaluate the adjListGraph and adjMatGraph data structures.

Experimental Setup

To create a diverse set of test scenarios, I developed a Python script that generates maze configuration files in JSON format. These files define the structure and complexity of the mazes, with the following parameters:

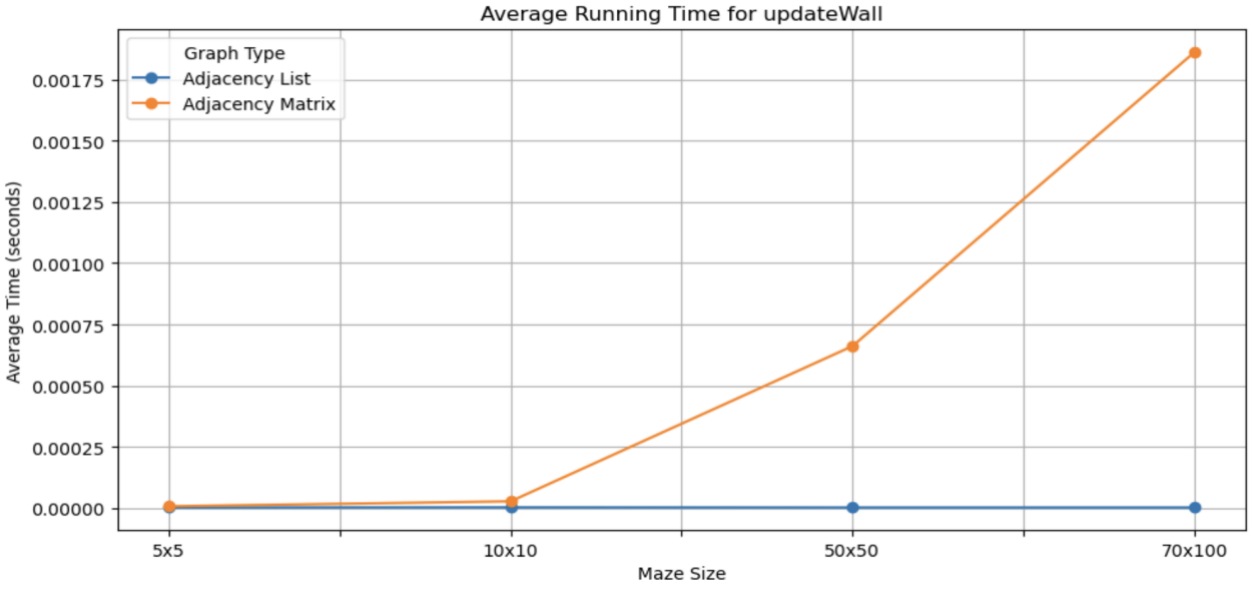
- Complexity Variables: This script allows to control the wall density and the number of entrances and exits in the mazes.
- Format: Storing the configurations in JSON format makes it easy to use and modify them as needed.

For each graph type, I executed a series of operations that are central to maze functionality—namely `updateWall()` and `neighbours()`. These functions were run within a complete maze generation cycle to closely mirror their actual usage. The operations were selected for their critical role in both constructing the maze and enabling navigation within it.

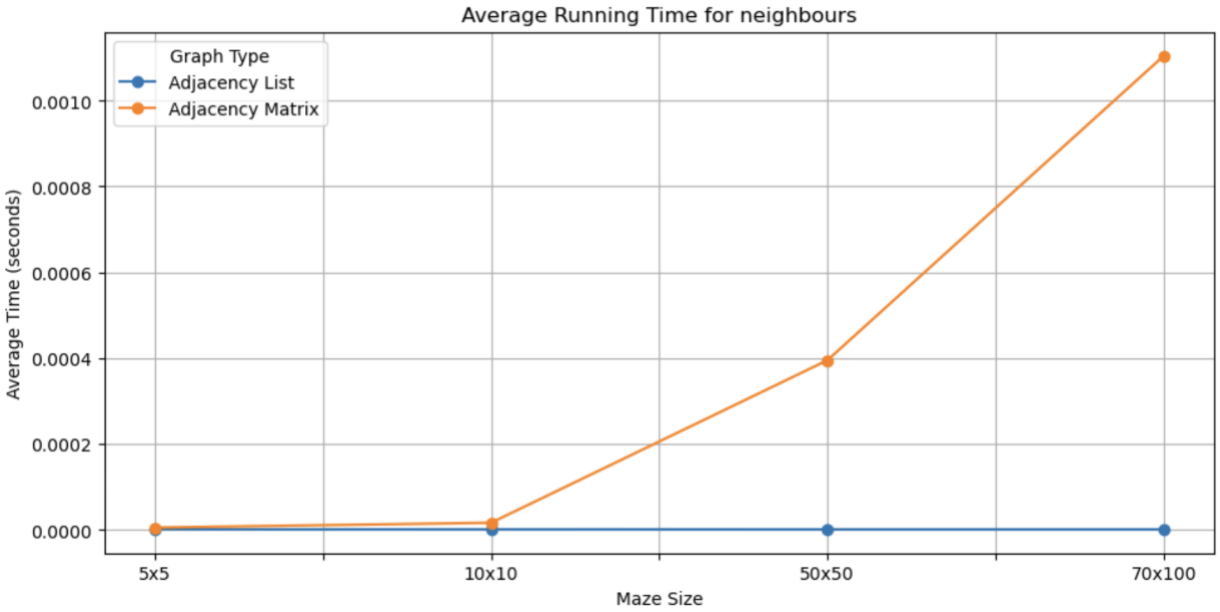
The execution times for these operations were recorded during the maze generation process. The timings were not merely snapshots but reflective of consistent performance across repeated runs, giving a reliable average that speaks to each structure's efficiency and effectiveness in handling maze-related tasks.

Empirical Analysis

The outcomes of different dimensions.



Graph 1.1 Average Running Time for updatewall()



Graph 1.2 Average Running Time for neighbours()

Analysis of Varying Size

In examining the average running time data for the updateWall() operation, it is evident that the Adjacency List data structure maintains consistent efficiency across all maze sizes. This consistency suggest that the list structure is not overly affected by the increasing complexity of the maze, which could be attributed to its direct access to vertices and edges without the need to traverse unnecessary elements.

Conversely, the Adjacency Matrix shows a substantial increase in operation time as the maze size expands, with the most pronounced jump occurring as I transition from medium-sized to large maze. This trend is likely because, in a matrix, the operation must handle a large set of data – most of which are irrelevant to any given update operation—as the matrix grow with the maze size.

The contrasting behaviours of these two data structures imply that the Adjacency List is better suited for mazes of varying size, particularly as they get larger. Its performance stability makes it a reliable create choice for applications where maze dimensions are dynamic or largely unpredictable. The Adjacency Matrix, while straightforward in implementation, might only be preferred for smaller maze where its time complexity does not exponentially degrade performance.