UNIVERSITY OF WESTMINSTER

INFORMATICS INSTITUTE OF TECHNOLOGY

Informatics Institute of Technology

BEng (Hons) in Software Engineering

Algorithms: Theory, Design and Implementation

5SENG003C.2

Task 05: Brief Report

Aaysha Fazal Mohamed

Uow Number – w1956175

IIT Number – 20221493

Group – C

This document highlights the essential data structures that underpin an efficient solution to the maze navigation challenge.

a) A short explanation of your choice of data structure and algorithm.

This section outlines how a strategic combination of a 2D array, Priority Queue, and Boolean 2D array forms an effective maze-solving algorithm.

1. **2D Array (maze)**:

A 2D array is used to represent the maze structure. This data structure is suitable for storing the layout of the maze, where each cell can be accessed using row and column indices.

This approach allows for the efficient storage and retrieval of maze information, making it easy to access and maintain.

2. **Boolean 2D Array (visitedCells)**:

A 2D boolean array is used to track visited cells in the maze. This helps prevent revisiting cells during the search process.

- Each cell is marked as visited once it has been explored during the search.

3. **PriorityQueue (openCells)**:

A PriorityQueue is utilized to store cells in the order of their total cost. This facilitates the efficient selection of cells with the lowest cost during the A* search algorithm.

- The priority queue is ordered based on the total cost of each cell, which is calculated as the sum of the heuristic cost and the cost to reach the cell from the starting point.

Collectively, these data structures underscore the critical role of thoughtful algorithmic strategies and robust storage mechanisms in tackling complex challenges such as maze navigation.

The A* search algorithm is a powerful tool for finding the shortest path from a starting point to a destination within a maze, leveraging the strengths of both Dijkstra's algorithm and greedy best-first search to efficiently guide its process with a heuristic. This heuristic is specifically the calculation of the Manhattan distance between the current cell and the destination, aiding in the prediction of the most promising path.

The implementation involves representing the maze as a grid of cells that are either traversable or blocked, which makes it easier to navigate and assess adjacent cells. A priority queue, organized by the total cost of each cell which includes both the heuristic and the actual costs from the start, facilitates the selection of the next cell to explore. This cost estimation is rooted in the Manhattan distance to the goal, serving as a forecast of the expense required to reach the endpoint from any given cell. The algorithm's operation persists until the priority queue is empty or the goal is achieved, focusing on cells with the minimal total cost and assessing neighbors in every cardinal direction to update their final costs accordingly. If the goal is reached or deemed unreachable, it traces back the shortest path by following parent pointers in each cell that were recorded during the search. The path is then visually represented on the grid with '@' symbols, effectively illustrating the shortest route discovered.

The chosen algorithms and data structures efficiently solve maze problems, with 2D arrays representing mazes and priority queues for effective cell selection. The A* algorithm ensures solution completeness and path optimality. Familiar techniques like 2D arrays and A* search improve code readability and maintenance.

Overall, the combination of these data structures and algorithm provides a robust solution for solving the maze problem efficiently and effectively.

b) A run of your algorithm on a small benchmark example. This should include the supporting information as described in Task 4.

The program has a user-friendly interface allows users to choose between running examples or benchmark files. Users can select the maze size they want to run and test the program with different types of input mazes of various dimensions. The program displays the solution by marking the path on the maze grid, allowing users to see the optimal route from the start to the endpoint.

Figure 1: maze10_1.txt Output and output visualization.

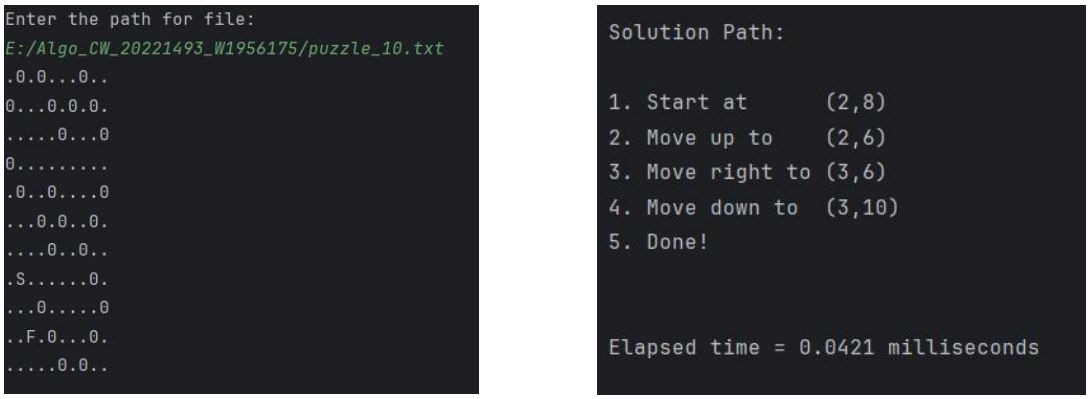

Figure 2: puzzle_10.txt Output and output visualization.

Additionally, the program provides information about the execution time, giving users insights into the program's performance.

c) A performance analysis of your algorithmic design and implementation. This can be based either on an empirical study, e.g., doubling hypothesis, or purely theoretical considerations, as discussed in the lectures and tutorials. It should include a suggested order-of-growth classification (Big-O notation).

File Name	_1	_2	_3	_4	_5	Average Time (ms)
maze10	0.0576 ms	0.0643 ms	0.1488 ms	0.0816 ms	0.2044 ms	0.11134 ms
maze15	0.0755 ms	0.0715 ms	0.0994 ms	0.0882 ms	0.0986 ms	0.08664 ms
maze20	0.1337 ms	0.1105 ms	0.1602 ms	0.1171 ms	0.2621 ms	0.15672 ms
maze25	0.2166 ms	0.1004 ms	0.3401 ms	0.1493 ms	0.2236 ms	0.2060 ms
maze30	0.6829 ms	0.4075 ms	0.333 ms	0.2376 ms	0.4241 ms	0.41742 ms

Table 1: Performance analysis Empirical approach Example files.

File Name	Average Time (ms)
puzzle_10	0.1637 ms
puzzle_20	0.0908 ms
puzzle_40	0.23 ms
puzzle_80	1.1647 ms
puzzle_160	1.4249 ms
puzzle_320	3.4264 ms
puzzle_640	8.2035 ms
puzzle_1280	21.5252 ms
puzzle_2560	44.2184 ms

Table 2: Performance analysis Empirical approach Benchmark files.

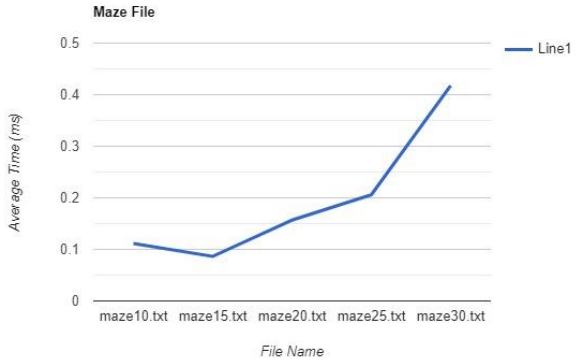

Figure 3: Performance analysis on Example files.

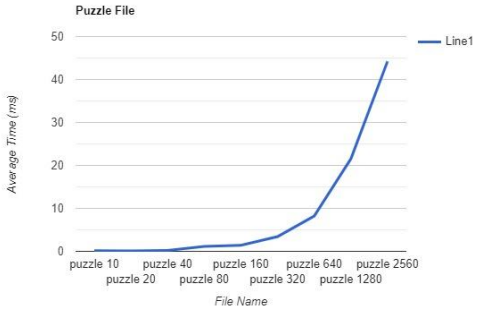

Figure 4: Performance analysis on Benchmark files.

After analyzing the provided data and observations, it's evident that the A* algorithm exhibits distinct performance characteristics when applied to maze solving compared to puzzle solving.

For maze solving, the average time tends to increase as the size of the maze grows, though not necessarily in a linear fashion. This increase becomes more pronounced as one progresses from smaller mazes to much larger ones. The time complexity for solving mazes with the A* algorithm seems to fall between O(n) and O(n^2), indicating a growth rate that is more than linear but less than quadratic.

Several factors, including the maze's structure and the choice of heuristic function, influence the actual performance of the A* algorithm in this context. Based on the data, it's reasonable to classify the A* algorithm's time complexity for maze solving as somewhere between linear and quadratic, likely leaning towards linear for smaller mazes and moving closer to quadratic for larger ones. In contrast, puzzle-solving presents a significantly different picture.

The average time required for solving puzzles escalates sharply with an increase in problem size, suggesting a time complexity that is likely exponential, perhaps even O(2^n) or higher. This stark difference can be attributed to the more complex search and optimization challenges encountered in puzzle solving compared to maze solving. Consequently, the time complexity for the puzzle-solving aspect of the A* algorithm can be categorized as exponential or potentially even higher, based on the observed escalation in average time across various problem sizes.

This analysis reveals that while the A* algorithm for maze solving shows a time complexity ranging between linear and quadratic, the algorithm’s application to puzzle solving indicates a significantly higher time complexity, likely exponential or beyond, reflecting the distinct challenges and computational demands of each problem type.