UNIVERSITY OF WESTMINSTER

INFORMATICS INSTITUTE OF TECHNOLOGY

BSc(Hons) Computer Science

Algorithms Theory, Design, and Implementation

5SENG003C.2

Module Leader : Mr. Ragu Sivaraman

INDIVIDUAL COURSEWORK

Student ID **-:** 20221671

Student UoW ID **-:** w1999483

Student Name **-:** Hansaja H. Hapuarachchi

Tutorial Group **-:** CS Group G

Brief explanation of the data structure used in implementation

The maze-solving algorithm of my application is built on the foundation of using a few intertwining data structures. Firstly, a 2D array serves as a comprehensive representation of the maze layout in both dimensions. Each cell of the array corresponds to the current position in the maze. This structure enables easy access to the maze cell contents, underpinning the algorithm's operations and structuring the maze-solving process.

At the same time, Queue using another LinkedList is used as a blueprint for organizing the breadth-first search algorithm . Thus, this queue represents a physical manifestation of the BFS algorithm as running adjacent until the whole layer of cells is iterated over. Since the shortest path from the start to the end point when implemented in BFS is discovered layer by layer, the queue ensures this blueprint is followed, maximizing the efficiency of the algorithm and solution optimality.

Aside from the two foundational structures, the algorithm uses a HashMap to store the paths traversed in the most effective way possible. The use of a HashMap, associating each location with its respective path, allows the algorithm to explore and discard paths faster relative to a more basic storage method. Similarly, a LinkedList is used to store the steps taken from finish to start. When the program has reached the end, the LinkedList is used to put out from the start to end, helping us print the answer more easily and effectively. All three data storage methods thus cooperate to create a working, effective maze-solving algorithm.

Brief explanation of algorithm used in implementation

The major algorithm that forms the foundation of our maze-solving algorithm is the (BFS) Breadth-First Search. BFS was chosen because this algorithm looks for the shortest way from the start to the finish in an unweighted graph, which is the ultimate objective of our program. Namely, BFS's operational approach implies exploring adjacent cells in all four directions, up, down, left, and right from the current cell unless the finish cell is reached. It implies traveling the entire maze "level by level" therefore, bfs is designed to automatically yield the optimal solution.

To conclude, the selected data structures and the BFS algorithm have proven to be most effective ways of solving mazes rapidly. Arrays, queues, HashMap's, and linked lists have been selected based on their suitability for supporting effective research and visualization of the types of maze paths and the resolution of the maze. The analysis presented in this report confers the significance of selecting proper data structures and algorithms on the maze-solving process and their impact on performance and effectiveness.

Experimenting with the Algorithm on Some Simple Test Cases

Example 1 :

```
File    Edit    View

.....0...S
....0.....
0.....0..0
...0....0.
.F......0.
.0........
.......0..
.0.0..0..0
0.........
.00.....0.
```

INPUT

```
Run    MazeRunner

"C:\Program Files\Java\jdk-20\bin\java

Start at: (1,10)
Path down to: (2,10)
Path left to: (2,6)
Path down to: (10,6)
Path right to: (10,8)
Path up to: (8,8)
Path right to: (8,9)
Path up to: (6,9)
Path left to: (6,3)
Path up to: (1,3)
Path left to: (1,1)
Path down to: (2,1)
Path right to: (2,4)
Path down to: (3,4)
Path left to: (3,2)
Path down to: (5,2)
Completed!
Execution time: 28 ms

Process finished with exit code 0
```

OUTPUT

Example 2:

```
File    Edit    View

...........0.......
0...0.....0.........
..........0000......
.....0......0.....0.
0.......0..0..0..F0.
..0..0.00......0.0..
0....0......0.......
......0.............
....0.....00........
0..........0....0...
....................
...0.........0....0.
....0...0.....00..0.
.000.0..0..0........
0.....00...0.....0.0
..0.................
.............0.....
....................
..00..0.......0.....
0...0.....0...S0..0.
```

INPUT

```
Run    MazeRunner

"C:\Program Files\Java\jdk-20\bin\java.exe"

Start at: (20,15)
Path left to: (20,12)
Path up to: (16,12)
Path left to: (16,4)
Path up to: (15,4)
Path right to: (15,6)
Path down to: (20,6)
Path right to: (20,10)
Path up to: (1,10)
Path right to: (1,12)
Path down to: (2,12)
Path right to: (2,20)
Path down to: (14,20)
Path left to: (14,13)
Path up to: (8,13)
Path left to: (8,8)
Path up to: (7,8)
Path right to: (7,12)
Path up to: (6,12)
Path right to: (6,15)
Path down to: (12,15)
Path right to: (12,18)
Path up to: (7,18)
Path up to: (4,14)
Path right to: (4,18)
Path down to: (5,18)
Completed!
Execution time: 30 ms

Process finished with exit code 0
```

OUTPUT

c) A performance analysis of your algorithmic design and implementation.

To conduct an empirical study, we run the algorithm on multiple mazes of various sizes and complexities. Each maze's solution time is recorded, which enables us to examine how the algorithm's execution time increases with the size of input transect. For example, we can test the doubling hypothesis; the maze's size is doubled, and we measure how the algorithm's runtime more than doubles. This experiment provides critical information about the running time of the algorithm in practice and how well it could handle the input size.

From a theoretical standpoint, we can analyze the algorithm's time complexity using Big-O notation. Since the algorithm employs breadth-first search (BFS), its time complexity is typically O(V+E) where V is the number of vertices (cells) in the maze and E is the number of edges (connections) between vertices. In the context of maze-solving, V represents the total number of cells in the maze, while E corresponds to the total number of possible connections between neighboring cells.

Based on the analysis, we can suggest an order-of-growth classification for the algorithm's time complexity. In the case of BFS-based maze-solving algorithms, the time complexity is often classified as O (V+E) or **O(n)** , where n represents the total number of cells in the maze. This classification indicates that the algorithm's runtime scales **linearly** with the size of the maze. Specifically, as the number of cells in the maze increases, the algorithm's execution time increases proportionally.

In conclusion, the empirical study, and the investigation of the algorithm from the theoretical perspective help us form a general understanding of its performance characteristics. The empirical study reveals how well the algorithm functions given the scale of real input and verifies the hypotheses, whereas the theoretical overview provides insights into a formalized model of its time complexity. When used alongside one another, the two methods reveal the algorithm's performance stipulations and limitations and provide grounds for its further evolution and improvements.