DATA STRUCTURES & OBJECT ORIENTED PROGRAMMING FINAL PROJECT REPORT: MAZE SOLVER

Written by:
Bagaskara Leo - 2802553165
Catherine Isabelle Ong - 2802501035
Raihan Hakim Anwar - 2802323130

I : TABLE OF CONTENTS

1. BACKGROUND

1.1 SIGNIFICANCE OF MAZE SOLVING IN THE MODERN ERA

One important field in robotics, artificial intelligence (AI), and computer problem solving is the study of maze solving solutions. Mazes are used as standards to assess the effectiveness of algorithms, decision-making procedures, and self navigating systems. AI has advanced as a result of the creation of numerous maze solving algorithms, especially in the areas of pathfinding and optimisation (Cully, et al., 2015). Maze solving techniques are essential in a variety of fields, such as network optimisation, robotics, and game creation. Robots with maze solving skills can help in search and rescue missions, navigate uncharted territory, and streamline logistical routes. Moreover, maze solving algorithms are essential for supporting decision making processes in AI, genetic algorithms, and machine learning.

Machine autonomy, logical reasoning and decision making processes have all progressed due to a large part of research into maze solving algorithms. It delivers a monitored, well organised and adaptable environment for evaluating, assessing and improving algorithms. Maze solving has developed over time from a problem in entertainment and games to a fundamental standard for algorithm development in robotics and AI (Potanin, et al., 2011). A number of well known algorithms have been created to effectively solve mazes. Among the most popular are Reinforcement Learning-Based Approaches, new developments use reinforcement learning to navigate mazes, increasing efficiency and adaptability and graph based search techniques in pathfinding techniques like Dijkstra's algorithm and A* are still being improved for real time robotics and artificial intelligence applications (Majumdar, et al., 2023).

Even if these algorithms are effective, problems still arise in practical applications. Optimised solutions are necessary for complex mazes with dynamic impediments, time limits, and significant computing costs. Furthermore, adaptive algorithms that can adapt in response to environmental feedback are necessary for real time maze solutions in robots. The purpose of this study is to investigate and assess several maze solving algorithms in order to ascertain their effectiveness, suitability, and potential for practical use. Gaining knowledge of these algorithms can help develop robotics, artificial intelligence, and computational problem solving techniques, which will result in more intelligent and self-sufficient systems. In summary, a fundamental component of artificial intelligence and the computer sciences is the study of maze solving solutions. Researchers can create more effective and efficient algorithms for navigation, problem solving, and decision-making by examining various strategies and refining them for practical uses.

1.2 EVOLUTION AND HISTORICAL CONTEXT

For thousands of years, individuals have been intrigued by the concept of solving mazes. Ancient mazes and labyrinths are used in myths to represent difficulties, adventures and journeys. But the idea took on an entirely new concept in the current digital era. Early computer scientists began testing algorithms that resembled human problem solving skills in mazes in the 1950s and 1960s (Nilsson, 2009). Simple electromechanical devices or early programmable robots, such as Grey Walter's tortoise robots, were utilized in one of the earliest observed experiments (Jun Tani, 2016). Maze solving has been an essential challenge for evaluating search algorithms like Depth-First Search (DFS) and Breadth-First Search (BFS) throughout the time of the development of modern computing. More complicated methods like Dijkstra's algorithm, A* and genetic algorithms were developed as AI advanced, offering not just precise but also the ideal answers (Marquis et al., 2020). At the moment, maze solving is an essential element of robotics. Robots must increasingly function in dynamic and irregular contexts, whether they are navigating damaged structures during search and rescue procedures, navigating through overcrowded metropolitan areas or maximizing movement in crowded warehouses. Many of these tasks are conceptually based on maze solving algorithms, particularly in realms like robot localization, avoiding obstacles and pathway routing (Steven Michael Lavalle, 2014).

1.3 THE ROLE OF MAZE SOLVING IN LABYRINTH CONTEXT

For thousands of years, people have been intrigued with labyrinths. Despite being frequently used concurrently with the word "maze", the labyrinth has a unique cultural and analogous meaning. Contrary to mazes, which offer options, branches and end points. labyrinths are typically unicursal with only one path leading to the centre. However, the journey along complicated paths serves as a metaphor for self-discovery, contemplation and change in both cases. Human cognition, myth, ritual and the pursuit of meaning are all deeply linked to the process of solving such systems. Mazes and labyrinths can be found in a variety of entertainment and educational settings. Every year, hundreds of people visit garden mazes, hedge mazes and cornfield mazes because they offer excitement as well as challenges. Human maze solvers actively relate to these areas, frequently going backwards and use reasoning, pattern recognition and memory in order to identify the exit. Such activities foster logical thinking and cognitive growth in addition to providing entertainment (Carter, 2015). Mazes are also included in children's puzzle books to help youngsters develop their patience, problem solving and motor skills.

It is impractical to exaggerate the educational benefit of maze solving. Mazes are used in classrooms to teach endurance and organization. They can be used to teach older pupils the basic concepts of algorithmic problem solving, the study of graphs and narrative organization. For instance, mazes can be employed in literature lessons to illustrate the format of quests or journeys, assisting students in absorbing how each individual encounters hurdles and develop over their journey. Labyrinths are also applied in clinical and social-emotional learning contexts. It is frequently advised to walk a labyrinth as a form of mindfulness that promotes tranquility and self-reflection especially for people who are struggling with stress, trauma or anxiety (Artress, 2006).

Additionally, labyrinths serve as essential to experience design and art. The maze has intrigued artists and constructors as a symbol of psychological complexity and alteration. Mirror labyrinths designed by contemporary artist Olafur Eliasson puzzle and intrigue visitors, encouraging them to investigate illusion and space in different manners (Eliasson, 2012). Through physical movement, the spectators piece together narrative components as they go through a maze-like setting at their own pace in interactive shows like Sleep No More (Machon, 2017). These instances emphasize unique interpretation over problem solving, turning the maze solver into a researcher and a co-creator of meaning.

In conclusion, all of these interpretations are connected by the understanding that managing a labyrinth is a trip that frequently reflects deeper societal, spiritual or interpersonal implications rather than just being a test of memory or reasoning. Maze solvers have long represented a basic human desire to search for meaning in complexity, persist through challenges and to emerge with increased understanding. It is important to keep in mind the human aspects of maze solving in a period where artificial intelligence and computational logic rule the day. Robots and algorithms have the skills of figuring out the best pathways through a grid, but they are unable to understand the enigma, admiration or shift that labyrinths have provided throughout history. In its fundamental and everlasting form, maze solving is still a specifically human activity that merges play and philosophy, logic and emotion, struggle and thinking.

2. PROBLEM DESCRIPTION

2.1 CHALLENGES OF TRADITIONAL PATHFINDING

The maze solving problem involves identifying the most efficient path from the starting point to a goal which is the ending point within an environment consisting of pathways and obstacles (walls) (Geeksforgeeks, 2023). These environments can be structured (grid-based mazes) or unstructured, with different levels of difficulty. The difficulty of pathfinding increases significantly with maze size, the density of obstacles, and any limitations on movement. In many cases, solving mazes like these requires careful algorithmic planning to minimize both time and memory taken.

Traditional pathfinding algorithms such as Depth-First Search (DFS) and Breadth-First Search (BFS) are commonly used to explore and solve mazes. While they guarantee finding a solution in connected graphs, they usually don't find the best possible path, especially in large or complex mazes. BFS usually ends up using more memory as it explores all possible paths evenly, while DFS can become inefficient if it explores long, unnecessary paths (Geeksforgeeks, 2025).

A major challenge in maze pathfinding is in computational efficiency. As the maze complexity increases, the number of possible paths grows exponentially. This leads to increased processing time and memory requirements. In real world applications such as robotics, game AI, and autonomous navigation systems, these computational limitations can affect responsiveness and performance. For instance, in autonomous vehicles, slow pathfinding computations could result in navigation delays or failure to react to sudden changes in the environment (Kleinberg & Tardos, 2005).

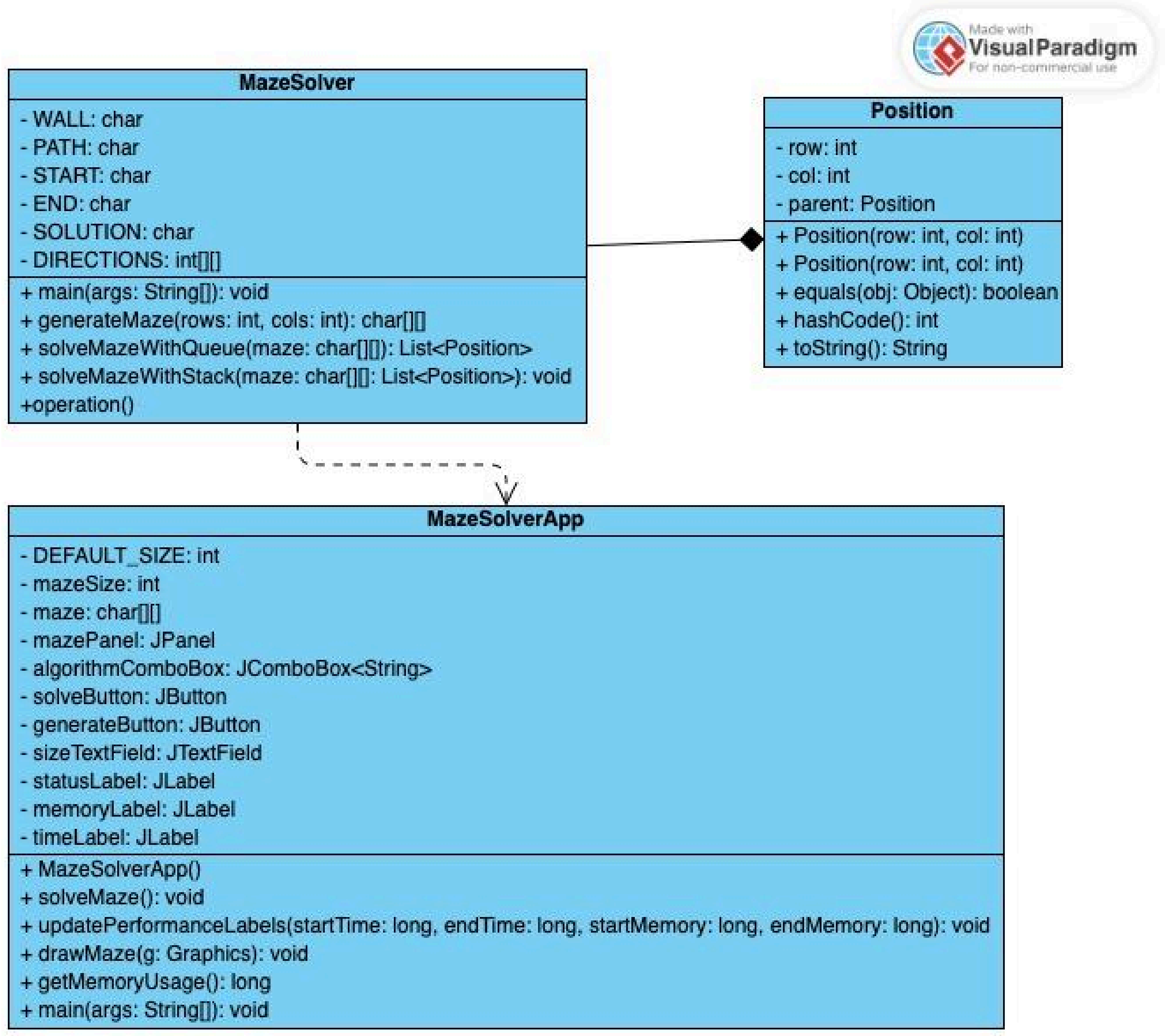
2.2 THE NEED FOR MORE EFFICIENT AND SCALABLE SOLUTIONS

Another difficulty is balancing between finding the shortest path and thoroughly exploring all the alternative routes. Some algorithms prioritize optimality but may become trapped in local minima or ignore potentially better paths due to early pruning or heuristics (Liu et al., 2025). More recent approaches, such as reinforcement learning, attempt to mitigate this by enabling agents to learn from experience and explore multiple possible routes before committing to a final path.

Given these limitations, there is a clear need for more advanced and adaptive pathfinding algorithms. These should be capable of operating efficiently in complex, dynamic, and large-scale environments. The aim of this project is to explore and evaluate pathfinding methods that improve on traditional approaches by offering better scalability, reduced memory consumption, and faster execution times, ultimately leading to more practical solutions in real-world applications.

3. SOLUTION

3.1 CLASS DIAGRAM


Made with Visual Paradigm For non-commercial use
MazeSolver
- WALL: char
- PATH: char
- START: char
- END: char
- SOLUTION: char
- DIRECTIONS: int[][]
+ main(args: String[]): void
+ generateMaze(rows: int, cols: int): char[][]
+ solveMazeWithQueue(maze: char[][]): List<Position>
+ solveMazeWithStack(maze: char[][]: List<Position>): void
+operation()
Position
- row: int
- col: int
- parent: Position
+ Position(row: int, col: int)
+ Position(row: int, col: int)
+ equals(obj: Object): boolean
+ hashCode(): int
+ toString(): String
MazeSolverApp
- DEFAULT_SIZE: int
- mazeSize: int
- maze: char[][]
- mazePanel: JPanel
- algorithmComboBox: JComboBox<String>
- solveButton: JButton
- generateButton: JButton
- sizeTextField: JTextField
- statusLabel: JLabel
- memoryLabel: JLabel
- timeLabel: JLabel
+ MazeSolverApp()
+ solveMaze(): void
+ updatePerformanceLabels(startTime: long, endTime: long, startMemory: long, endMemory: long): void
+ drawMaze(g: Graphics): void
+ getMemoryUsage(): long
+ main(args: String[]): void

3.2 ALTERNATIVE DATA STRUCTURES

To navigate the maze and determine the shortest path from the starting point to the end point destination (or whether or not the maze is solvable at all in the first place), two data structures will be used and compared against each other to see which one is more efficient in solving the maze. The criteria to determine how efficient the data structure is includes measuring the time taken in milliseconds to execute the program and the amount of memory taken in kilobytes to execute the program. The average of this data will then be taken to find which data structure is the most efficient.

These two data structures are a Queue (FIFO) for breadth-first search (BFS) and Stack (LIFO) for depth-first search (DFS). Each data structure influences how the algorithm explores the maze and affects the efficiency and memory usage. We chose to implement the stack data structure and the queue data structure because when designing a maze solver, choosing the data structure directly impacts the strategy used to explore the maze.

QUEUE DATA STRUCTURE (BFS)

A queue is a first in, first out (FIFO) data structure which means that the first element added is the first to be processed and taken out. This algorithm ensures that all possible moves are explored level by level when applied to BFS. This makes BFS a good option for finding the shortest path in an unweighted maze. The algorithm starts from the start position and systematically explores all the adjacent cells before moving deeper into the maze.

QUEUE DATA STRUCTURE ADVANTAGES AND DISADVANTAGES

The main advantage of using a queue is that it guarantees the shortest path. This is because BFS explores all paths at the same depth first, the first time the destination is reached is through the shortest possible path. This makes using a queue useful for mazes with multiple possible paths. Additionally, it works well in wide-open mazes where there are many possible directions to go at each step (Russell & Norvig, 2021). When solving a maze there is a chance that the maze might be unsolvable. Using BFS makes sure that if there is a solution, it will definitely be found. This is because it explores all of the possible paths until the first (and therefore the shortest) solution is found (GeeksforGeeks, 2025). Overall, using a queue data structure to implement this maze solver allows for a systematic exploration which means that it would be good for all maze shapes.

However, the disadvantage of using BFS is that it uses high amounts of memory (especially in mazes that are large or wide in size). Since BFS explores all paths at once, it stores a large number of nodes in the memory which makes it inefficient for larger mazes. This memory used increases proportionally with the number of cells explored which can become a problem when dealing with large mazes. The space complexity of BFS is O(b^d) where b is the branching factor (number of children at each node) and d is the depth. Because of this, BFS is very space-bound when implemented in a maze solver so the memory taken up in the process of running the program will be a large amount (GeeksforGeeks, 2025).

STACK DATA STRUCTURE (DFS)

A stack is a last in, first out (LIFO) data structure which means that the last element added is the first to be processed and taken out. This algorithm follows one path as deeply as possible before backtracking if a dead end is reached. This approach prioritizes depth over breadth which makes it suitable for mazes with long corridors or single pathways (Cormen et al., 2009).

STACK DATA STRUCTURE ADVANTAGES AND DISADVANTAGES

The main advantage of using a stack is that it requires less memory than using a queue. This is because it does not store all possible paths simultaneously, instead it only keeps track of the current track and backtracks if necessary. This makes it more efficient for larger mazes, especially when the correct path is likely to be found earlier. This can also be faster in some cases such as in a maze with few branching paths since it does not waste time exploring unnecessary options (Sedgewick & Wayne, 2011). The time complexity of DFS is O(bd) where b is the branching factor (the number of children at each node) and d is the depth. This is because it generates the same set of nodes as BFS but in a different order so instead of being limited to space, it is limited to time instead (GeeksforGeeks, 2024).

However, the main disadvantage of using a stack is that it does not guarantee the shortest path to solving the maze if there is more than one possible path to solve the maze. Since DFS follows one path before backtracking if it reaches a dead end, it may explore inefficient routes first which can lead to a longer solution. In the worst case, it may explore the entire maze before reaching the correct path which makes it much slower than using a queue. There is also a chance that the DFS algorithm can get lost in deep dead ends. This is because it might keep going down the left-most path endlessly. Even in a maze with a solution, DFS could end up wasting time exploring too deeply. The best way to avoid this is by setting a limit on how deep the search can go but since we don't know exactly how deep it needs to be, the value can be made too small which can cause the solution to be missed or the value can be too big which can cause the search to still waste time and not find the shortest solution (GeeksforGeeks, 2024).

3.3 HOW IT WORKS

DATA STRUCTURE FOR STORING THE MAZE: 2D ARRAY

A two-dimensional (2D) array is an array that can have both rows and columns. A row stores the horizontal elements while the column stores the vertical elements. Using a 2D array in this case is useful since the data in a maze is already naturally arranged in rows and columns (path, wall, start point, end point) (Runestone Academy, n.d.). Using this structure is also ideal for representing other problems that involve the same natural layout such as matrices and game boards.

In this project, we will be using a 2D array to store the maze data. A 2D array of characters is used to represent the maze. Each cell in the array holds a character that defines its role in the maze: a hash '#' represents a wall, a dot '.' represents an open path, an 'S' indicates the starting point, an 'E' indicates the end point the maze will have to reach, a 'V' marks a visited cell during backtracking, and an asterisk '*' indicates a part of the correct solution path. The maze solver program uses this array to navigate through the maze recursively, updating the values as it explores through and solves the randomly generated maze.

ADVANTAGES AND DISADVANTAGES OF USING A 2D ARRAY

One of the main advantages of using a 2D array is how simple it is to implement. It allows for fast and constant access to any cell using its row and column indices. The layout of a 2D array also naturally mirrors the structure of a real life maze, which makes it a good choice to use in this context. Another advantage is that using a 2D array is memory efficient for small to medium sized mazes where the dimensions are known in advance. In addition, the regular structure of a 2D array makes it easier to visualize, print, and debug during development. In terms of memory access time, the use of a fixed-size array can also lead to a more optimized performance compared to dynamic structures.

On the other hand, using a 2D array also has its disadvantages. These include the fact that its size is fixed which means that the dimensions must be specified at the time of creation and cannot be changed later on. This gives a limit to the flexibility for the application if dynamic resizing is needed. Since in this case it is not needed and the dimensions are already given at the start of the creation of the maze, this can be ignored. Additionally, in cases where the maze contains many walls and only few paths, the 2D array may allocate memory inefficiently by storing unnecessary cells. Another disadvantage is that a 2D array only holds the current state of the maze and it doesn’t already support tracking the path taken through the maze unless additional data structures are introduced.

HOW THE MAZESOLVERAPP & MAZESOLVER FILES WORK

We decided to make two separate files called MazeSolverApp and MazeSolver. Each of these files has a different purpose that will be discussed further below. However, these two files when run together create a console-based maze solving application.

1. **MAZESOLVERAPP FILE**
   This is the application class that contains the main method and runs the program. This file includes the main method, the code to create a maze, an instance of MazeSolver, and it calls the solver method and prints the result (either the path or the solved maze). This file works by creating a maze as a 2D array and calls a function (MazeSolver.solveMazeWithQueue(maze) or MazeSolver.solveMazeWithStack(maze)) to solve the maze. Then the result is printed (the time taken, the path taken, the memory taken, success/failure to solve the maze, etc.).

2. **MAZESOLVER FILE**
   This is the file that defines the logic for solving the maze. It contains the class responsible for actually performing the maze solving algorithm in the form of depth-first search (using stack data structure) or breadth-first search (using queue data structure). This file includes a class that takes a maze (2D array) as an input and tries to find a path from the start to the end. It implements stack and queue data structures to move through the maze. This file works by creating a 2D array where a hash ‘#’ represents a wall, a dot

‘.‘ represents an open path, an ‘S’ indicates the starting point, an ‘E’ indicates the end point the maze will have to reach, a ‘V’ marks a visited cell during backtracking, and an asterisk ‘*’ indicates a part of the correct solution path. The maze solver then checks each possible direction (up, down, left, right), marking visited cells and continuing until the solution is found.

STACK IMPLEMENTATION

The solveMazeWithStack method implements the Depth-First Search (DFS) algorithm which uses a stack data structure to explore the maze. It begins by searching for the start and end positions within the maze. If any of them is not present, it returns null, indicating there is no solution possible. A copy of the maze is created to avoid modifying the original, and a stack is initialized to hold positions to explore, along with a boolean matrix to track visited locations. The starting position is pushed onto the stack and marked as visited. The algorithm then enters a loop that continues as long as the stack is not empty. In each iteration, a position is popped from the top of the stack. If this position matches the end position, the algorithm reconstructs the path by tracing back through the parent positions and returns the path as a list. Otherwise, the algorithm examines the four other positions (up, down, left, right) of the current position. For each of the four positions, it checks if the move is valid. This means that the neighboring position is still in the maze boundaries, is either a path or the end position, and has not been visited before. If a neighboring position meets these criterias, it is marked as visited, pushed onto the stack, and its parent is set to the current position. This process continues, diving deeper along one path until it either finds the end or reaches a dead end. If the stack becomes empty before the end position is found, it tells us that all the possible paths have been explored and there has been no success. This leads to the algorithm returning null which indicates that there is no solution that exists. The maze copy is used to mark visited locations with 'V', providing a visual trace of the algorithm's exploration without altering the original maze structure.

QUEUE IMPLEMENTATION

The solveMazeWithQueue method implements the Breadth-First Search (BFS) algorithm to find a path from the start to the end position in a maze. The algorithm begins by initializing a queue to store positions to explore and a boolean matrix to keep track of visited positions. The starting position is added to the queue and marked as visited. The algorithm then enters a loop that continues as long as the queue is not empty. In each iteration, the algorithm retrieves and removes a position from the front of the queue. If this position is the end position, the algorithm reconstructs the path from the end to the start and returns it. Otherwise, the algorithm explores the neighboring positions (up, right, down, left) of the current position. For each neighboring position, it checks if the move is valid by checking if: the neighbor is within the maze bounds, is a path or the end position, and has not been visited yet. If a neighbor is valid, it is marked as visited, added to the queue, and its parent is set to the current position. This makes sure that the path can be reconstructed later. If the queue becomes empty before the end position is found, the algorithm decides that there is no path and returns null. If this were to happen, this would mean that there is no existing solution to be found. A copy of the maze is used to mark visited cells with 'V' to visualize the search process without making changes to the original maze.

3.4 RESULTS

Data structure: Stack

Solve maze								
Input maze size	Test 1		Test 2		Test 3		**Average (Test 1, 2 & 3)**	
	Memory (KB)	Time (ms)	Memory (KB)	Time (ms)	Memory (KB)	Time (ms)	Memory (KB)	Time (ms)
10	491	2.16	375	0.65	81	0.32	315.67	1.043
20	491	0.44	491	0.79	491	0.59	491.00	0.607
30	491	2.74	81	2.98	491	0.45	354.33	2.057
40	491	0.43	81	0.74	81	0.71	217.67	0.627
50	491	1.99	81	0.53	81	1.79	217.67	1.437
60	491	1.26	245	0.45	245	0.85	327.00	0.853
70	491	1.86	245	1.53	245	0.71	327.00	1.367
80	163	2.91	245	1.09	245	1.17	217.67	1.723
90	163	4.55	259	1.28	259	0.42	227.00	2.083
100	163	0.95	259	2.38	259	0.53	227.00	1.287
Avg:	392.6	1.929	236.2	1.242	247.8	0.754	**292.201**	**1.3084**

Data structure: Queue

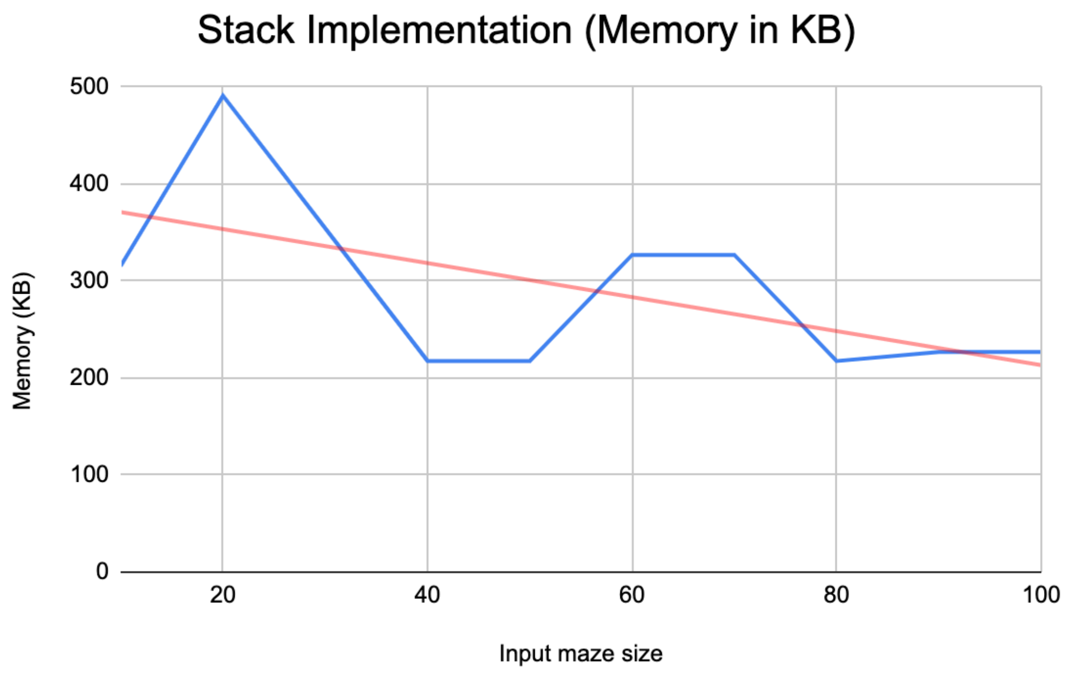
Solve maze								
Input maze size	Test 1		Test 2		Test 3		**Average (Test 1, 2 & 3)**	
	Memory (KB)	Time (ms)	Memory (KB)	Time (ms)	Memory (KB)	Time (ms)	Memory (KB)	Time (ms)
10	81	0.09	259	0.15	259	0.11	199.67	0.117
20	81	0.49	259	0.40	259	0.36	199.67	0.417
30	81	0.64	81	0.27	259	0.56	140.33	0.490
40	81	0.85	259	0.41	259	1.02	199.67	0.760
50	163	0.96	163	0.46	81	1.59	135.67	1.003
60	163	2.24	81	1.82	163	1.52	135.67	1.860
70	163	1.97	81	0.65	163	1.57	135.67	1.397
80	163	2.38	245	3.69	491	1.03	299.67	2.367
90	464	3.90	347	0.54	245	0.52	352.00	1.653
100	464	3.09	491	1.02	491	0.62	482.00	1.577
Avg:	190.4	1.661	226.6	0.941	267	0.89	**228.002**	**1.1641**

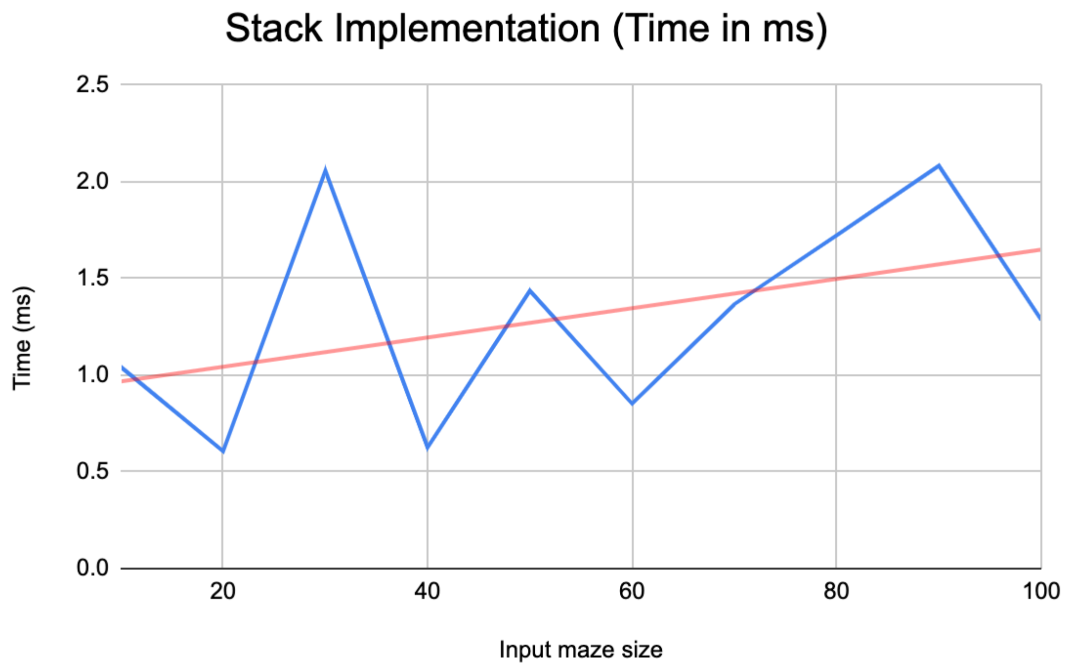
GROWTH GRAPHS: STACK IMPLEMENTATION

Legend:

- Blue line: data plotted in a line graph
- Red line: line of best fit

AVERAGE OF TESTS 1, 2 & 3:



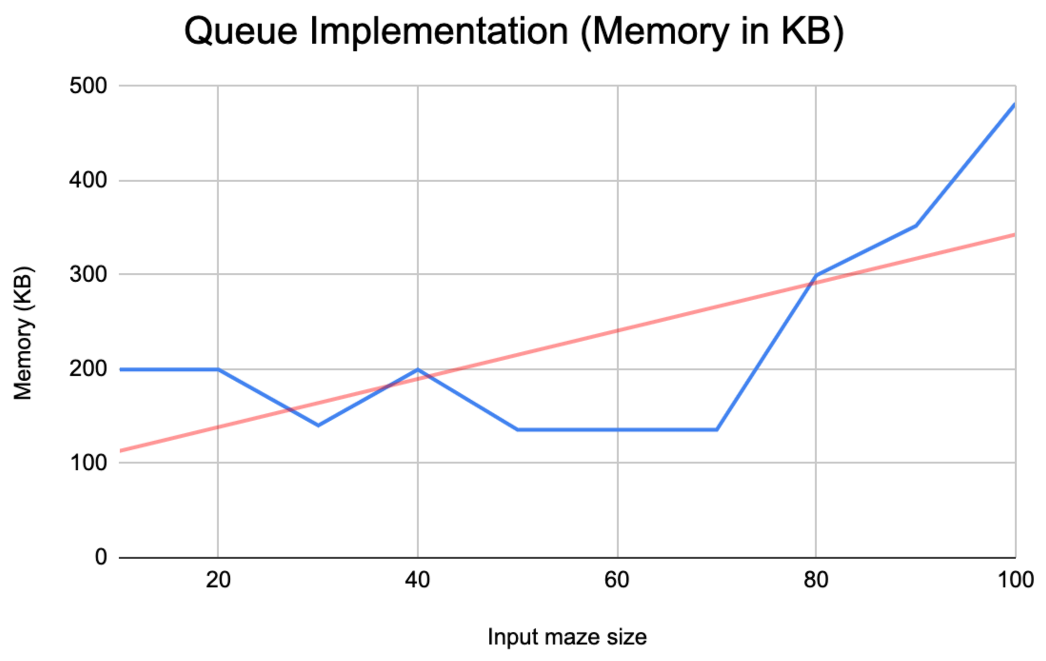


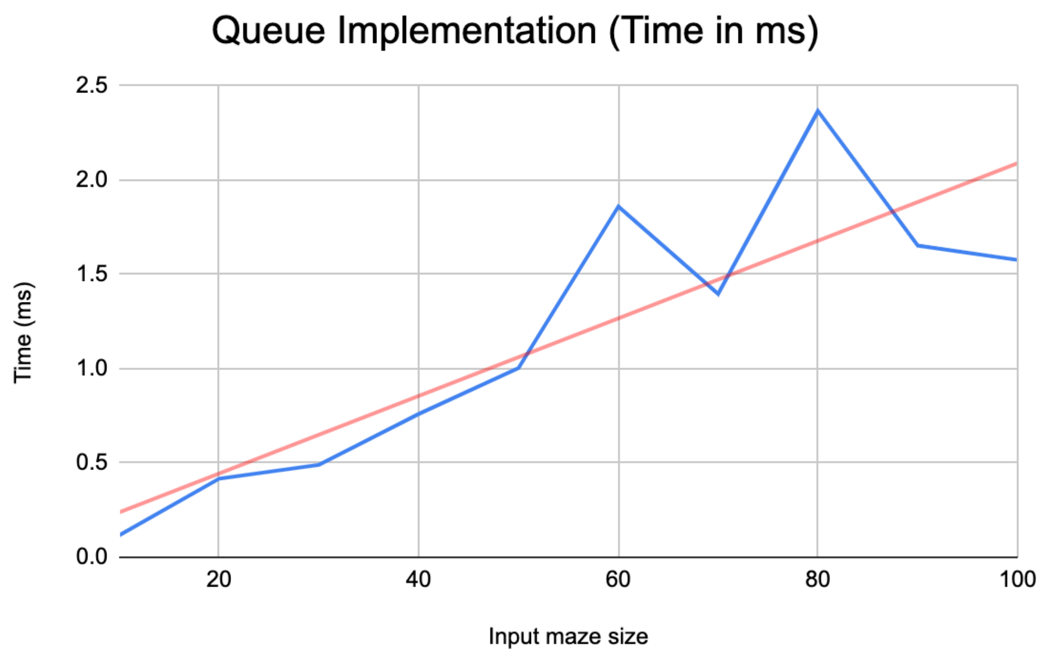
GROWTH GRAPHS: QUEUE IMPLEMENTATION

Legend:

- Blue line: data plotted in a line graph
- Red line: line of best fit

AVERAGE OF TESTS 1, 2 & 3:





FINAL AVERAGE TESTING RESULTS:

Stack data structure:

- Memory: 292.201 KB
- Time: 1.3084 ms

Queue data structure:

- Memory: 228.002 KB
- Time: 1.1641 ms

CONCLUSION

We ran the program using the stack data structure and the queue data structure to solve different sizes of mazes. We decided to do the testing at intervals of 10 starting from maze size 10 all the way up to size 100. As a result of our testing, we came up with the data above. Generally, as the size of the maze increases for both stack and queue data structures, the time taken to run the program increases. However, this is not always the case as sometimes the random generation of the maze path values can cause the program to store a larger size/smaller size maze. In all three of our stack data structure testing, the memory decreased. However, for the rest of the tests the data all seemed to increase. This is because DFS explores deep paths first and can exit early once it finds the goal, especially in sparse or lucky configurations. If the maze has fewer branches or a straightforward path, DFS may quickly reach the goal without visiting most of the grid. If there are less visited nodes, the visited map footprint becomes less and the stack becomes shallower leading to less memory usage. This could also be by random chance since each test had different wall placements and maze density that would affect the number of nodes the DFS explores and change how much of the maze is stored in the memory.

From our testing results, we can conclude that using the queue data structure is more efficient than using the stack data structure. On average, using the stack data structure takes up 292.201 KB of memory and 1.3084 ms of time. On the other hand, using the queue data structure takes up an average of 228.002 KB of memory and 1.1641 ms of time. Taking the average value allows us to best evaluate which data structure is the best and most efficient in solving the randomly generated maze.

3.5 COMPLEXITY ANALYSIS

QUEUE DATA STRUCTURE

TIME COMPLEXITY

Every cell in the maze could potentially be visited once. BFS uses a queue, and for each dequeued cell, it examines up to 4 neighbors. So in the worst case that the entire maze is open:

- Visits each cell once: O(R x C)
- Constant work per cell (checking on neighbors and marking visited cells): O(1)
- R x C = n
- In conclusion, the time complexity is O(n)

SPACE COMPLEXITY

- The visited array stores one boolean per cell: O(R x C)
- The queue can hold up to all the cells in the maze in the worst case: O(R x C)
- The path list might store nearly all the cells: O(R x C)
- R x C = n
- In conclusion, the space complexity is O(n)

STACK DATA STRUCTURE

TIME COMPLEXITY

Like BFS, DFS explores each cell at most once. Even with recursion or explicit stack, it processes each accessible cell once: O(R x C)

- R x C = n
- In conclusion, the time complexity is O(n)

SPACE COMPLEXITY

- The visited array: O(R x C)
- The recursion stack can go as deep as the number of open cells: O(R x C)
- The path list stores the final path: up to O(R x C)
- R x C = n
- In conclusion, the space complexity is O(n)

5. TEAM WORKLOAD

CATHERINE ISABELLE ONG

- Performance Analysis & Testing: Measures runtime & memory efficiency, and compares results
- Report Writing: Solution, problem description & appendix
- Poster
- Final presentation slides

BAGASKARA LEO

- Algorithm Implementation: DFS & BFS
- UI & Visualization: Maze display design and user interaction
- Report Writing: Background & appendix

RAIHAN HAKIM ANWAR

- Algorithm Implementation: DFS & BFS
- UI & Visualization: Maze display design and user interaction

5. REFERENCES

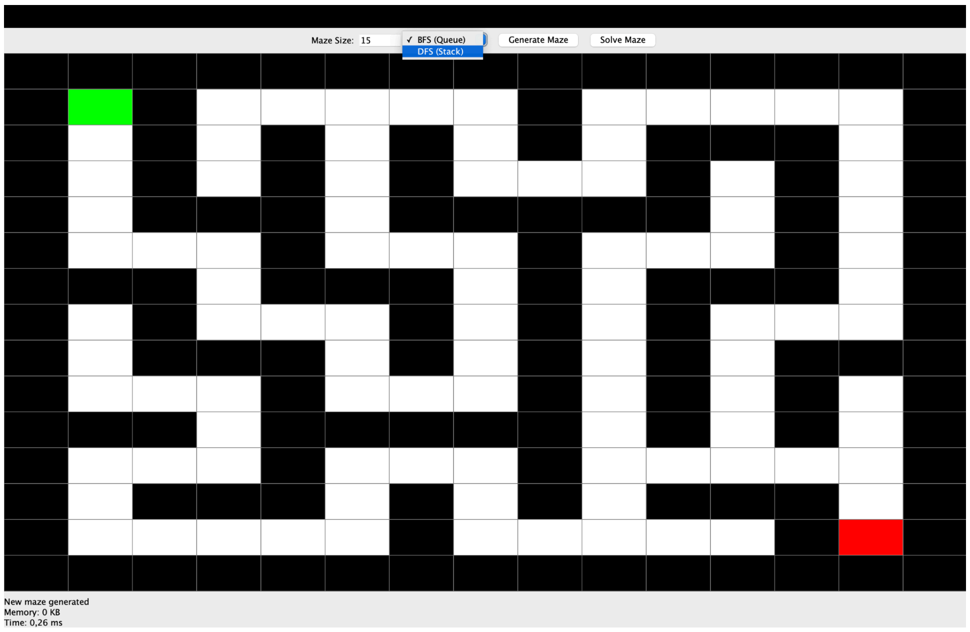
In text references:

1. Cully, A., Clune, J., Tarapore, D., & Mouret, J. B. (2015). Robots that can adapt like animals. *Nature, 521*(7553), 503-507. https://doi.org/10.1038/nature14422
2. Majumdar, A., Mei, Z., & Pacelli, V. (2023). Fundamental limits for sensor-based robot control. *The International Journal of Robotics Research, 42*(12), 1051-1069. https://doi.org/10.1177/02783649231189107
3. Potanin, A., & University, V. (2011). *Algorithms and data structures*. Pearson Education New Zealand.
4. Nilsson, N. J. (2009). *Artificial intelligence a new synthesis*. San Francisco, California Morgan Kaufmann Publishers, Inc.
5. Jun Tani. (2016). *Exploring Robotic Minds : Actions, Symbols, and Consciousness as Self-Organizing Dynamic Phenomena*. New York Oxford University Press -12-23.
6. Marquis, P., Papini, O., & Prade, H. (2020). *A Guided Tour of Artificial Intelligence Research*. Springer Nature.
7. Steven Michael Lavalle. (2014). *Planning algorithms*. Cambridge University Press, , Cop.
8. Carter, R. (2015). *Language and Creativity*. Routledge.
9. Artress, L. (2006). *Walking a sacred path : rediscovering the labyrinth as a spiritual practice.* Riverhead.
10. Ólafur Elíasson, Engberg-Pedersen, A., & Ursprung, P. (2008). *Studio Eliasson : an encyclopedia*. Taschen.
11. Machon, J. (2017). *Immersive Theatres*. Bloomsbury Publishing.
12. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (3rd ed.). MIT Press.
13. Russell, S. J., & Norvig, P. (2021). *Artificial Intelligence: A Modern Approach* (4th ed.). Pearson.
14. GeeksforGeeks. (2025, April 21). *Applications, advantages and disadvantages of breadth first search (BFS)*. GeeksforGeeks. https://www.geeksforgeeks.org/applications-of-breadth-first-traversal/
15. Sedgewick, R., & Wayne, K. (2011). *Algorithms* (4th ed.). Addison-Wesley Professional.
16. GeeksforGeeks. (2024, May 15). *Applications, advantages and disadvantages of depth first search (DFS)*. https://www.geeksforgeeks.org/applications-of-depth-first-search/
17. Runestone Academy. (n.d.). Introduction to 2D arrays. https://runestone.academy/runestone/static/JavaReview/Array2dBasics/a2dBasics.html
18. GeeksforGeeks. (2023, November 28). Tutorial on path problems in a grid, maze, or matrix. https://www.geeksforgeeks.org/dsa/tutorial-on-path-problems-in-a-grid-maze-or-matrix/
19. GeeksforGeeks. (2025, June 9). Difference between BFD and DFS. https://www.geeksforgeeks.org/difference-between-bfs-and-dfs/
20. Kleinberg, J., & Tardos, É. (2005). *Algorithm Design*. Pearson.
21. Liu, H., Cao, J. & Wang, Z. Research on path planning of mobile robot in complex environment. *Discov Appl Sci* 7, 330 (2025). https://doi.org/10.1007/s42452-025-06713-y

Code references:

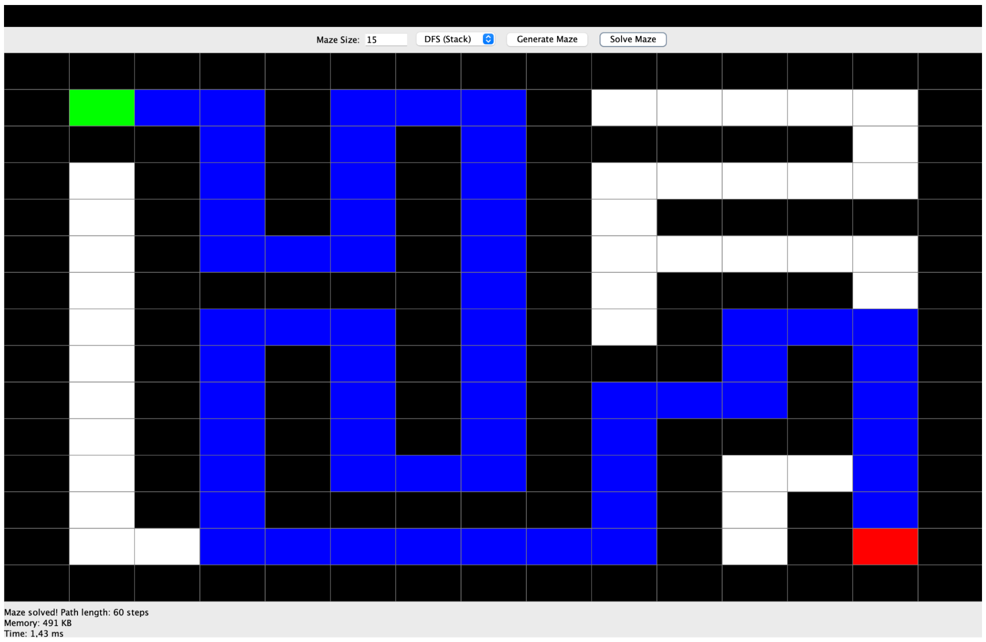
1. https://youtu.be/qyFbX5xoF4Y?si=a8PsqGgi6MzSRrO7
2. https://youtu.be/hC8yRec0EJ0?si=X1UYL6yKdg2ZBM_V

6. APPENDIX



New maze generated
Memory: 0 KB
Time: 0,26 ms

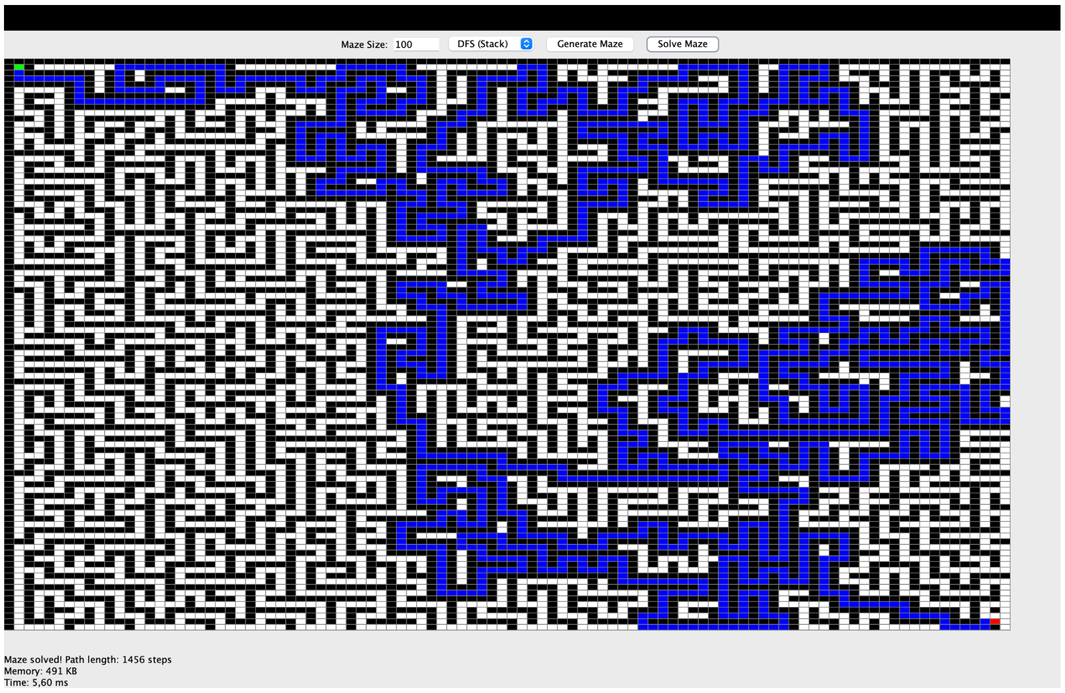
: : THERE IS A BUTTON AT THE TOP MIDDLE OF THE SCREEN THAT GIVES YOU THE OPTION TO CHOOSE WHICH ALGORITHM YOU WANT TO USE. YOU CAN USE EITHER BFS (QUEUE) OR DFS (STACK). BEFORE YOU GENERATE OR SOLVE A MAZE, YOU MUST PICK AN ALGORITHM TO USE AND YOUR DESIRED MAZE SIZE.



Maze solved! Path length: 60 steps
Memory: 491 KB
Time: 1,43 ms

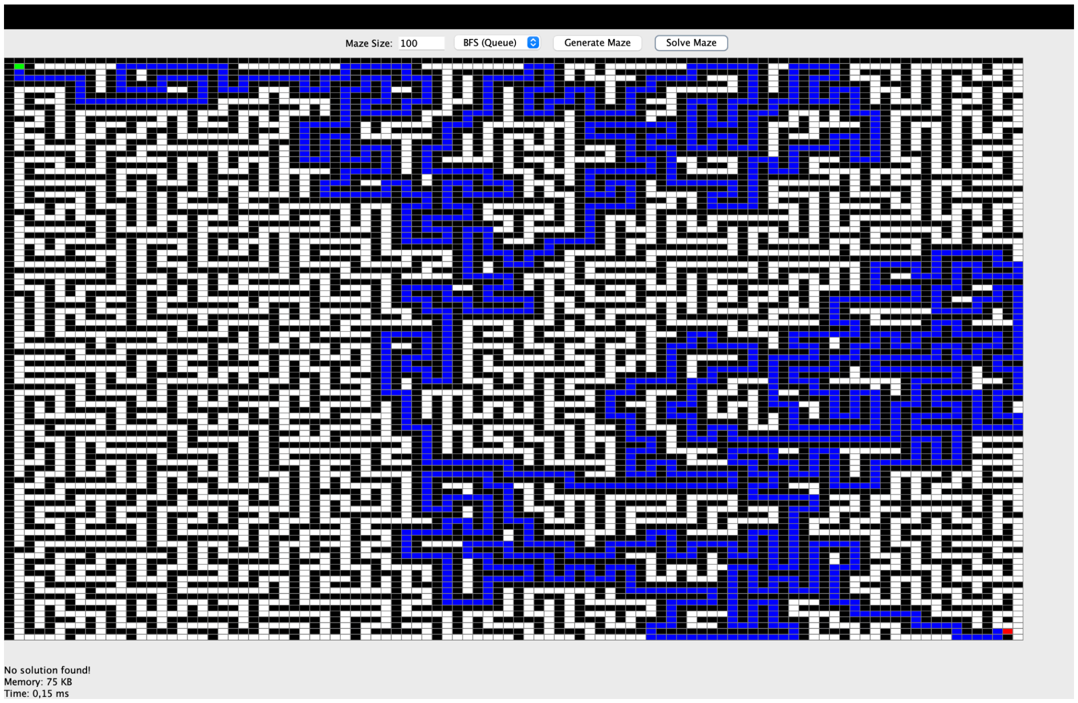
: IN THIS PICTURE, IT IS SHOWING A SMALL MAZE OF SIZE ‘15’ GENERATED AND SOLVED USING DFS (STACK). TO GENERATE THE MAZE, INPUT YOUR DESIRED MAZE SIZE (IN THIS CASE 15) AND CLICK ‘GENERATE MAZE’. TO SOLVE THE ALREADY

GENERATED MAZE, CLICK ‘SOLVE MAZE’ AND IT WILL SHOW YOU THE BEST POSSIBLE PATH TO SOLVE THE MAZE. THE PROGRAM SHOWS THE AMOUNT OF STEPS TAKEN TO SOLVE THE MAZE, THE MEMORY TAKEN, AND THE TIME TAKEN AT THE BOTTOM LEFT CORNER OF THE SCREEN.



```
Maze solved! Path length: 1456 steps
Memory: 491 KB
Time: 5,60 ms
```

: IN THIS PICTURE WE ARE SHOWING A BIGGER SIZE OF THE MAZE SOLVER ‘100’, AND USING THE DFS (STACK) ALGORITHM. TO GENERATE THE MAZE, INPUT YOUR DESIRED MAZE SIZE (IN THIS CASE 100) AND CLICK ‘GENERATE MAZE’. TO SOLVE THE ALREADY GENERATED MAZE, CLICK ‘SOLVE MAZE’ AND IT WILL SHOW YOU THE BEST POSSIBLE PATH TO SOLVE THE MAZE. THE PROGRAM SHOWS THE AMOUNT OF STEPS TAKEN TO SOLVE THE MAZE, THE MEMORY TAKEN, AND THE TIME TAKEN AT THE BOTTOM LEFT CORNER OF THE SCREEN.



```
No solution found!
Memory: 75 KB
Time: 0,15 ms
```

:IN THIS PICTURE WE ARE SHOWING THE BIGGER SIZE OF THE MAZE SOLVER “100” AND WITH USING THE BFS (QUEUE) ALGORITHM. TO GENERATE THE MAZE, INPUT YOUR DESIRED MAZE SIZE (IN THIS CASE 100) AND CLICK ‘GENERATE MAZE’. TO SOLVE THE ALREADY GENERATED MAZE, CLICK ‘SOLVE MAZE’ AND IT WILL SHOW YOU THE BEST POSSIBLE PATH TO SOLVE THE MAZE. THE PROGRAM SHOWS THE AMOUNT OF STEPS TAKEN TO SOLVE THE MAZE, THE MEMORY TAKEN, AND THE TIME TAKEN AT THE BOTTOM LEFT CORNER OF THE SCREEN.

GITHUB LINK

https://github.com/bagaskara-hub/MazeSolver

PRESENTATION LINK

https://www.canva.com/design/DAGqKtbrpY8/cML2zyEIHGUPm4Y4mvrawA/edit?utm_content=DAGqKtbrpY8&utm_campaign=designshare&utm_medium=link2&utm_source=sharebutton

POSTER LINK

https://www.canva.com/design/DAGp4Wk9_zA/xAtw-FmhiFN8sypTC57xjw/edit?utm_content=DAGp4Wk9_zA&utm_campaign=designshare&utm_medium=link2&utm_source=sharebutton