Comparison of Different Search Algorithms in Maze Solving(January 2023)

Alperen Özkaya, MEF University, Istanbul, Turkey, ozkayaal@mef.edu.tr

***Abstract* - This report is about the comparison of different shortest path algorithms such as BFS, DFS, Dijkstra, and Random Walk in terms of path length, accuracy of the shortest path, and execution time ısing the module 'pyamaze' to generate required mazes. The implementation of the algorithms, and the structure of maze are explained in detail. The comparison of the algorithms are made, and represented with the related visuals. Lastly, it was concluded that the different algorithms may give different results depending on the different cases.**

Keywords: AI, shortest path, BFS, DFS, Dijkstra, Random Walk, maze solving.

I. Introduction

This paper describes the project that investigates the performance of several search algorithms for maze solutions. The project is built with Python and the 'pyamaze' module, which enables the creation of randomly generated mazes as well as the display of search pathways. Breadth-First-Search (BFS), Depth-First-Search (DFS), Dijkstra's Algorithm, and Random Walk are the four search algorithms implemented in the project. The goal of this project is to compare the performance of various algorithms in terms of search path length, accuracy of shortest path, and time required to reach to destination.

II. Algorithms

A. Breadth First Search:

The algorithm begins at the bottom right corner of the maze and utilizes a deque (double-ended queue) as the frontier queue to choose which cells to investigate next. The method investigates each cell by considering all potential motions (East, West, North, and South) and adds valid child cells to the frontier queue. The method additionally remembers the search path, the bfs path (the shortest path from start to destination), and the shortest path by saving each child cell in the bfs path dictionary with the current cell as the key. When the goal is reached, the algorithm returns to the beginning to find the shortest path.

B. Depth First Search:

Similarly, the dfs() function solves the maze using the Depth-First-Search method, with the same starting point and aim as the bfs() function. Instead of a deque, a stack is used as the frontier queue to keep track of which cells to investigate next. The method investigates each cell by considering all potential motions (East, West, North, and South) and adds valid child cells to the frontier stack. The method additionally remembers the search path, the dfs path (the shortest path from start to destination), and the shortest path by saving each child cell in the dfs path dictionary with the current cell as the key. When the target is achieved, the algorithm traces backward to show the shortest path.

C. Dijkstra:

Dijkstra's algorithm is a graph search algorithm that generates the shortest path by solving the single-source shortest path problem with non-negative edge weight. It is similar to Breadth-First Search in that it explores vertices in the order of their distance from the beginning point using a priority queue. Dijkstra's algorithm calculates the distance between the starting point and each vertex and utilizes this information to select which vertex to investigate next.

D. random walk:

This algorithm simply lets agent to choose a direction to move randomly. It does not ensure

that the path it finds is the shortest path, since it explores the maze by choosing one of the directions(N, E, S, W) randomly in each turn.

III. Maze

The 'pyamaze' module is used to generate random mazes. The maze is of fixed size and my_maze.rows and my_maze.cols represent the number of rows and columns respectively. The starting point of the maze is in the bottom right corner of the maze and is represented by the tuple (my_maze.rows, my_maze.cols) and the goal point is represented by my_maze._goal.

The maze is represented by a 2D array called maze_map and each cell in the array represents a cell in the maze. The keys of the maze_map dictionary represent the cells and the values are the walls surrounding each cell, with 'N' for North, 'E' for East, 'S' for South, and 'W' for West. If there is no wall in the corresponding direction, the value will be set to True, otherwise it will be set to False.

To track paths, algorithms use various data structures such as deques, lists, dictionaries (traversed lists, frontier deques, bfs_path dictionaries, search_path lists, etc.). These data structures are used to store the cell being inspected, the cell not yet inspected, the shortest path from the destination to the starting point, and the search path, respectively.

IV. Uses

There are two options for the user:

The first option is that the user can create a new random maze by specifying the size of it.

The second option lets users load the mazes that were created before in CSV format. In the project file there are 4 mazes that are ready to use with the sizes: 5x5, 10x10, 20x,20 and 30x30.

```
# choose to create maze from scratch or use the existing ones
create_maze = input('Enter 1 if you want to create the maze yourself:\n')
```

Fig. 1. Code line that asks the user to choose to create maze or not.

The user is asked to enter '1' if the desire is to create a new random maze, otherwise the user can use the pre-created mazes.

```
paths = find_paths(test_maze, str(input("Choose an algorithm: bfs-dfs-dijkstra-random:\n")))
```

```
algorithm = str(input("Choose an algorithm: bfs-dfs-dijkstra-random:\n"))
```

Fig. 2. Code line that asks the user to choose an algorithm.

For both options user is allowed to choose which algorithm to use to solve the maze among: Breadth First Search, Depth First Search, Dijkstra, and Random Walk.

V. Comparison of Algorithms

A. 5x5 MAZE

```
Choose an algorithm: bfs-dfs-dijkstra-random:
bfs
Choose the size of maze --> 0:5x5, 1:10x10, 2:20x20, 3:30x30
```

Fig. 3. An example of an input.

BREADTH-FIRST SEARCH

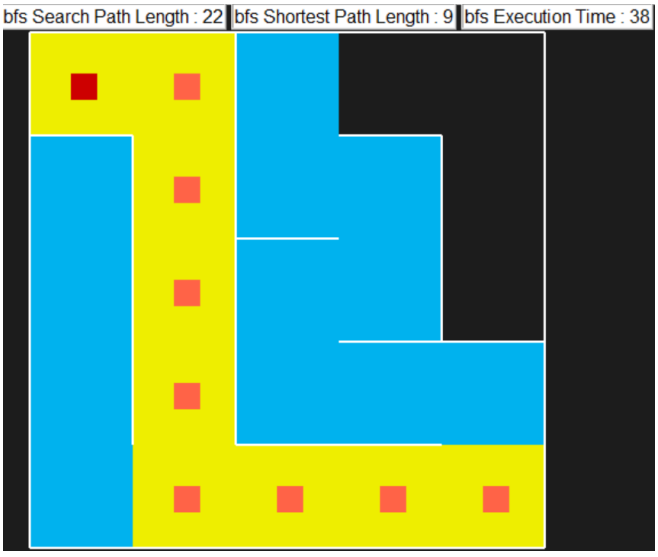

Fig. 4. A visual representation of the BFS algorithm in a 5x5 maze using the 'pyamaze' module.

DEPTH FIRST SEARCH

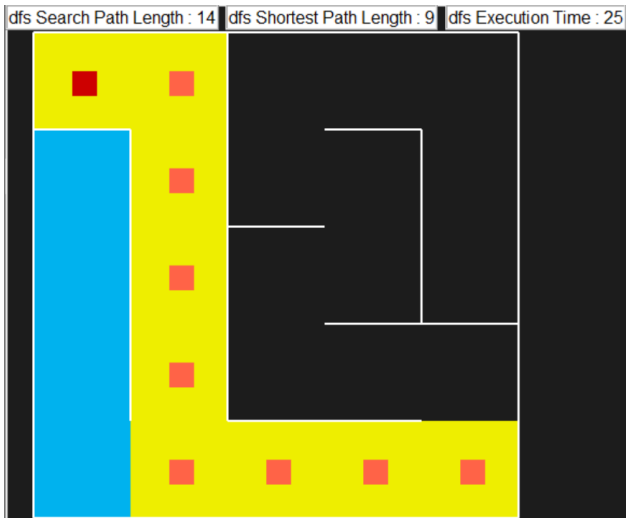

Fig. 5. A visual representation of DFS algorithm in a 5x5 maze using the 'pyamaze' module.

DIJKSTRA

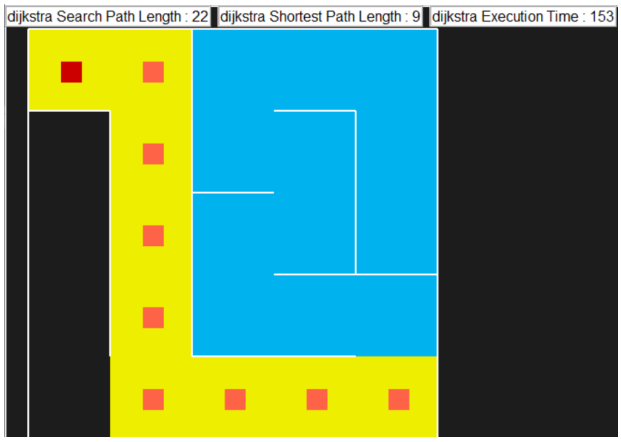

Fig. 6. A visual representation of the Dijkstra algorithm in a 5x5 maze using the 'pyamaze' module.

RANDOM WALK

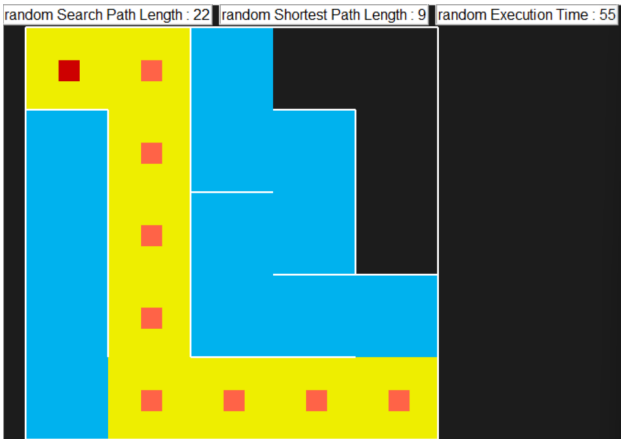

Fig. 7. A visual representation of the Random Walk algorithm in a 5x5 maze using the 'pyamaze' module.

Explanation: It can be seen that all the algorithms have found the shortest path with the value '9'. DFS has the best search path length '14', while the other algorithms iterated over 22 cells to find the shortest path. Observing the execution times, it can be seen that dijkstra is the slowest algorithm for this 5x5 sized maze case.

B. 10x10 MAZE

BREADTH-FIRST SEARCH

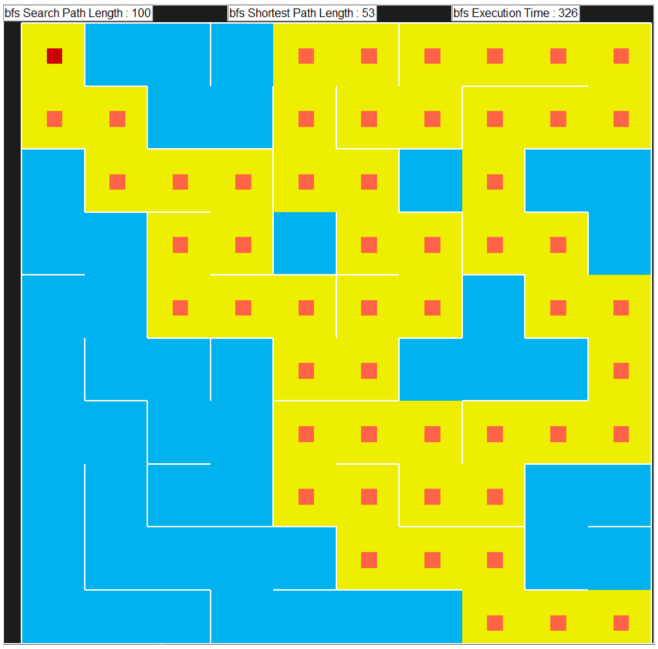

Fig. 8. A visual representation of the BFS algorithm in a 10x10 maze using the 'pyamaze' module.

DEPTH FIRST SEARCH

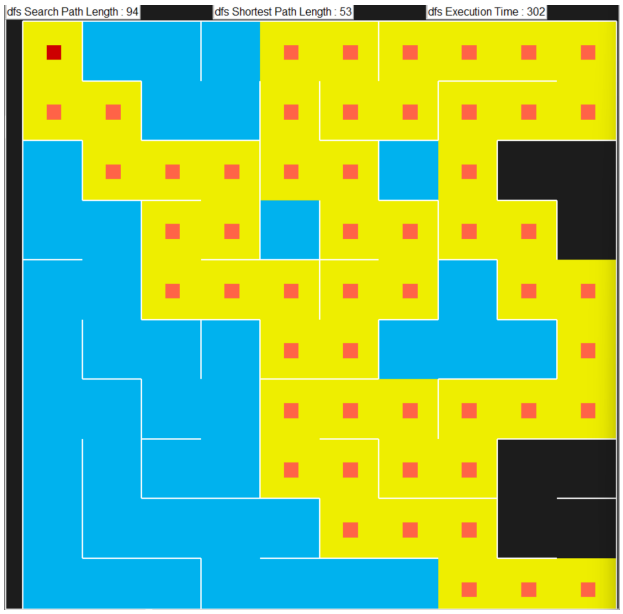

Fig. 8. A visual representation of the DFS algorithm in a 10x10 maze using the 'pyamaze' module.

DIJKSTRA

Fig. 8. A visual representation of the Dijkstra algorithm in a 10x10 maze using the 'pyamaze' module.

RANDOM WALK

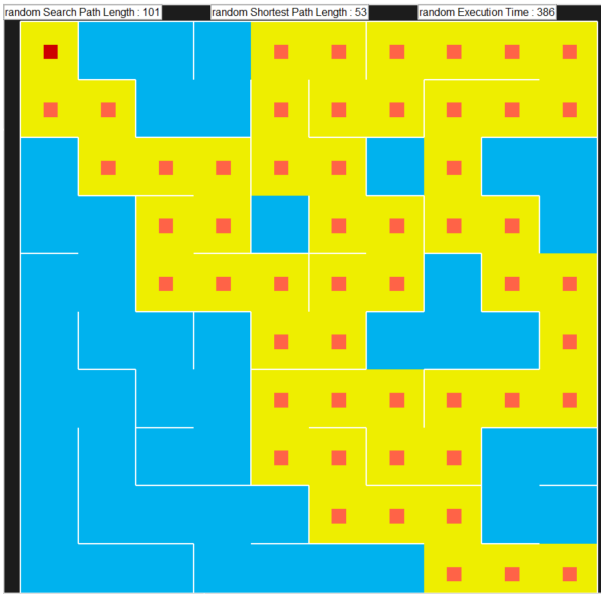

Fig. 8. A visual representation of the Random Walk algorithm in a 10x10 maze using the 'pyamaze' module.

Explanation: It can be seen that all the algorithms have found the shortest path with the value '53'. Dijkstra has the best search path length '72', while the other algorithms iterated about 100 cells to find the shortest path. When the execution times are examined, it can be seen that, like in the 5x5 case, dijkstra is the slowest algorithm for the 10x10 sized maze case.

C. 20x20 MAZE

BREADTH-FIRST SEARCH

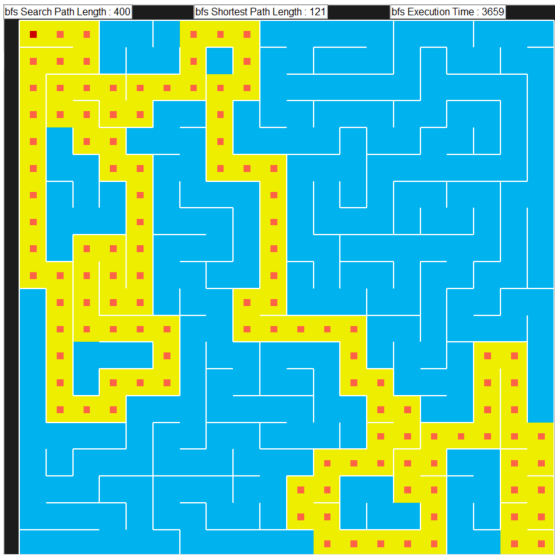

Fig. 8. A visual representation of the BFS algorithm in a 20x20 maze using the 'pyamaze' module.

DEPTH FIRST SEARCH

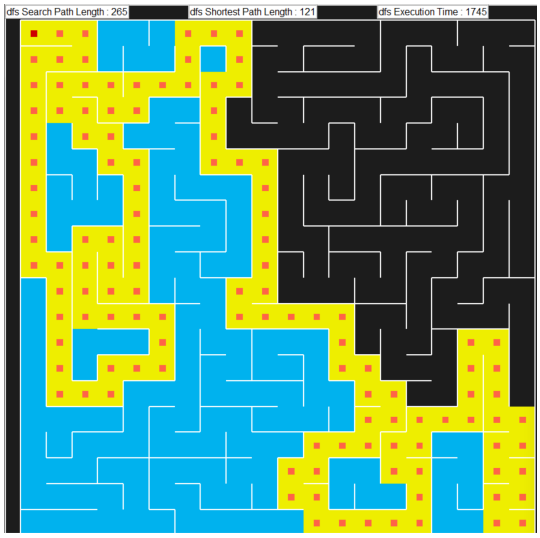

Fig. 8. A visual representation of the DFS algorithm in a 20x20 maze using the 'pyamaze' module.

DIJKSTRA

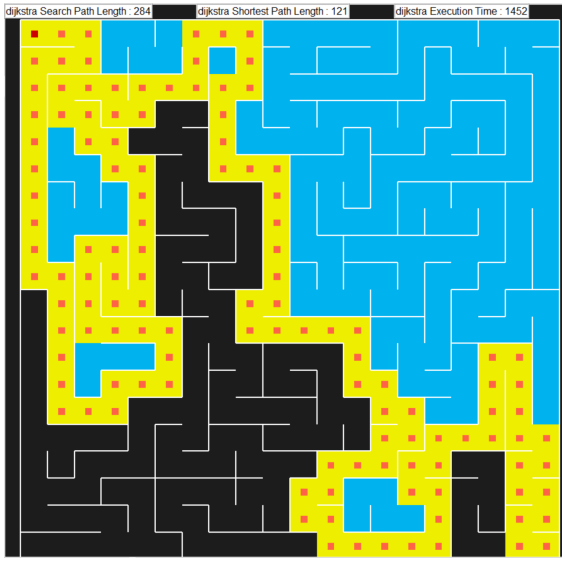

Fig. 8. A visual representation of the Dijkstra algorithm in a 20x20 maze using the 'pyamaze' module.

RANDOM WALK

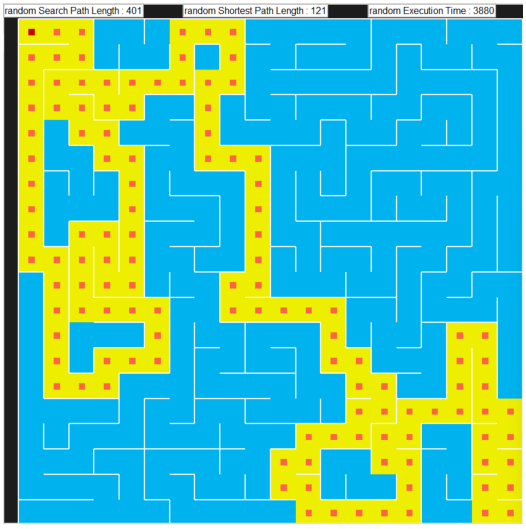

Fig. 8. A visual representation of the Random Walk algorithm in a 20x20 maze using the 'pyamaze' module.s

Explanation: It can be seen that all the algorithms have found the shortest path with the value '121'. Depth First Search has the best search path length '265', Dijkstra also has a similar search path length with the value '284'. Breadth First Search and Random Walk algorithms have given the results 400, 401 respectively, they both can be considered bad in terms of memory consumption since they need to store more values compared to Dijkstra and DFS When the execution times are examined, it can be seen that, BFS and Random Walk algorithms are slower than Dijkstra and DFS.

D. 30x30 MAZE

BREADTH-FIRST SEARCH

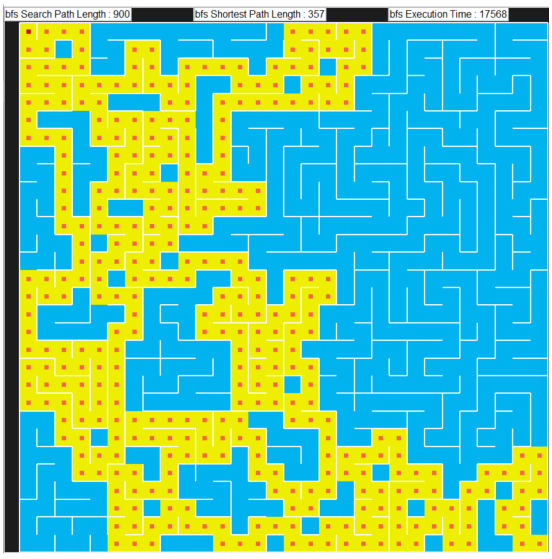

Fig. 8. A visual representation of the BFS algorithm in a 30x30 maze using the 'pyamaze' module.

DEPTH FIRST SEARCH

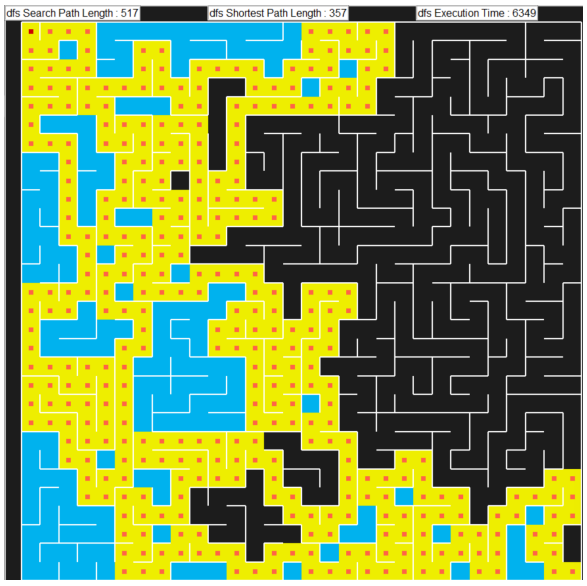

Fig. 8. A visual representation of the DFS algorithm in a 30x30 maze using the 'pyamaze' module.

DIJKSTRA

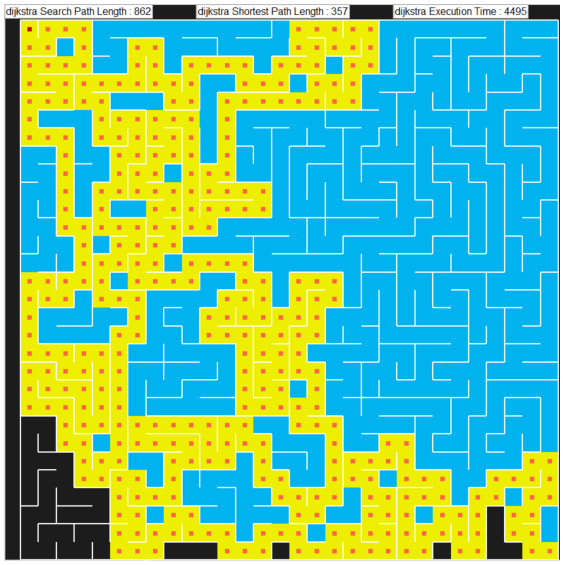

Fig. 8. A visual representation of the Dijkstra algorithm in a 30x30 maze using the 'pyamaze' module.s

RANDOM WALK

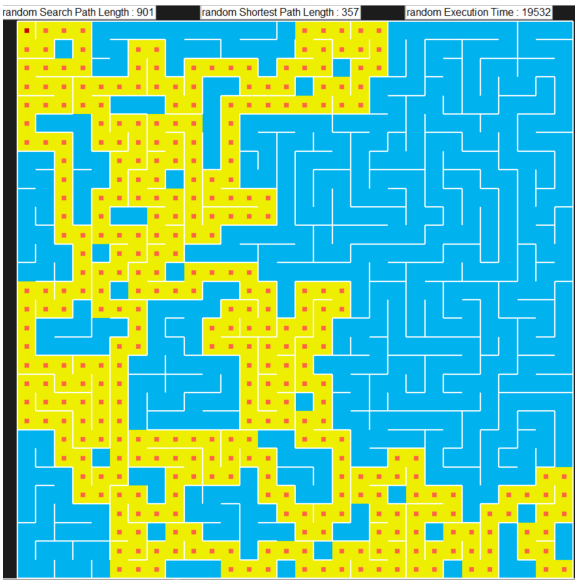

Fig. 8. A visual representation of the Random Walk algorithm in a 30x30 maze using the 'pyamaze' module.

Explanation: It can be seen that all the algorithms have found the shortest path with the value '357'. Depth First Search has the best search path length '517', while all the other algorithms have similar and worse results compared to DFS . When the execution times are examined, Dijkstra is the fastest algorithm while Random Walk is the slowest algorithm for this case.

VI. General Discussion

Observing the results for each different sized maze, it can clearly be seen that algorithms can give different results for different cases. It should be noted that Dijkstra and DFS in most cases, give the best results in terms of memory consumption, while BFS mostly gives the worst results. Random walk generally gives inconsistent results in terms of search path length, because of the randomness of the choices. However its execution time is the worst one among these algorithms except the first case with 5x5 sized maze.

VII. References

1] M. A. Naeem, “A Python Module for Maze Search Algorithms,” *Medium*, Sep. 24, 2021. https://towardsdatascience.com/a-python-module-for-maze-search-algorithms-64e7d1297c96

[2] “Shortest path in a Binary Maze - GeeksforGeeks,” *GeeksforGeeks*, May 16, 2016. https://www.geeksforgeeks.org/shortest-path-in-a-binary-maze/

[3] “Pathfinding,” *Wikipedia*, Dec. 15, 2022. https://en.wikipedia.org/wiki/Pathfinding#Algorithm