PROJECT ANALYSIS OF ALGORITHM

NAME:M.HANZALAH JAMAL

SAP ID:55257

SUBMITTED TO: SIR USMAN SHARIF

4TH SEMESTER

SPRING 2025

Table Of Contents

FLOYD WARSHALL ALGORITHM

1. OVERVIEW

The Floyd-Warshall Algorithm is an all-pairs shortest path algorithm for a weighted graph with positive or negative edge weights (but no negative cycles). It finds the shortest paths between all pairs of nodes in a graph. The algorithm systematically updates the distances between pairs of nodes based on intermediary nodes. This makes it particularly useful for computing transitive closures and other graph-based computations.

Floyd-Warshall is an iterative dynamic programming algorithm that computes the shortest paths between every pair of nodes in a graph. It works by incrementally improving the solution by considering whether each node can be an intermediate node in a shorter path.

2. ALGORITHM EXPLAINATION

The core idea of the Floyd-Warshall algorithm is to consider all possible paths between every pair of vertices and iteratively improve the solution. Here's how the algorithm works:

1. Initialization: Start by creating a distance matrix where the direct edge between two nodes is represented by the weight of the edge. If there is no edge, the distance is set to infinity. The diagonal is initialized to zero, representing the distance from a node to itself.
2. Iterative Update: For each pair of nodes **i** and **j**, the algorithm checks if the path through an intermediate node **k** (where **k** is any node in the graph) provides a shorter path. This is done for all nodes **i, j, k,** updating the distance matrix as follows:

   dist[i][j]=min(dist[i][j],dist[i][k]+dist[k][j])dist[i][j] = \min(dist[i][j], dist[i][k] + dist[k][j])dist[i][j]=min(dist[i][j],dist[i][k]+dist[k][j])
3. Termination: After iterating through all the nodes, the distance matrix will contain the shortest distances between all pairs of nodes.

3. FOCUS

The Floyd-Warshall Algorithm is designed primarily for finding the shortest paths between all pairs of vertices in a graph. It is most beneficial when dealing with dense graphs or graphs where you need to compute the shortest paths between every possible pair of nodes, rather than just a single source to destination. Its main advantage is its ability to handle negative edge weights and still produce correct results, as long as the graph does not contain negative weight cycles.

4. KEY-FEATURES

- All-Pairs Shortest Path: It computes shortest paths for all pairs of nodes.
- Handles Negative Weights: It can handle graphs with negative edge weights (provided no negative weight cycles are present).
- Works on Dense Graphs: Ideal for dense graphs, where you want to find distances between many nodes at once.
- Dynamic Programming Approach: It uses a dynamic programming technique, considering each node as an intermediate point to improve path lengths.

5. APPLICATION

The Floyd-Warshall Algorithm can be applied in various areas such as:

- **Network Routing:** To find the shortest path for routing packets between multiple nodes.
- Web Crawling: To find the shortest links between pages.
- **Social Network Analysis:** To determine the shortest connections or degrees of separation between people.
- **Geographical Information Systems (GIS):** To compute shortest travel paths between locations.
- **Graph Analysis:** In cases where a transitive closure (reachability between nodes) is needed.

6. PERFORMANCE ANALYSIS

TIME COMPLEXITY

The time complexity of the Floyd-Warshall algorithm is **O(n^3),** where n is the number of vertices in the graph. This is due to the triple nested loops that iterate over all pairs of nodes **i**, **j** and intermediary nodes **k**.

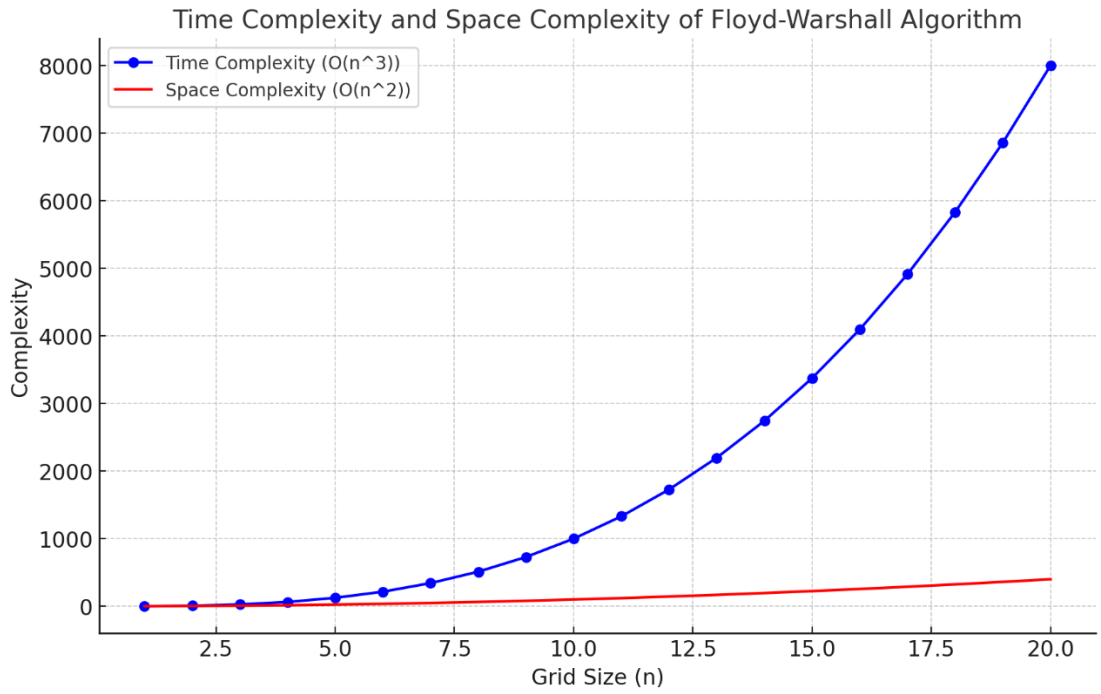
- Outer loop: Iterates over all vertices as the potential intermediate nodes.
- Inner loops: Iterate over all pairs of nodes **i** and **j** to update the distance between them.

Thus, the time complexity is dominated by the **O(n^3)** complexity.

SPACE COMPLEXITY

The space complexity of Floyd-Warshall is **O(n^2).** This is because the algorithm requires a matrix of size n×n \times n×n to store the distances between each pair of nodes. Even though the algorithm doesn't use any extra storage beyond this matrix, the space required is quadratic in terms of the number of vertices.

GRAPH



7. OVERALL PERFORMANCE

Floyd-Warshall performs efficiently for graphs that are dense, or when computing all-pairs shortest paths is required, and n is relatively small or moderate. However, for large sparse graphs or graphs with a large number of vertices, it becomes less efficient

compared to other algorithms like Dijkstra's (for single-pair shortest paths) or the Bellman-Ford algorithm (which handles negative weights).

The algorithm's key advantage is its ability to handle negative weights and compute all shortest paths in a straightforward manner using a dynamic programming approach.

8. WHY THIS ALGORITHM IS EFFICIENT?

The Floyd-Warshall algorithm is considered efficient in situations where you need the shortest paths for every pair of nodes and the graph is relatively small or dense. Its main benefit lies in its simplicity and correctness even with negative weights, unlike algorithms like Dijkstra's, which cannot handle negative edge weights directly.

Despite its cubic time complexity, it's a powerful solution for problems where all-pairs shortest path computation is necessary, and it works well when the graph size is moderate and the edge weights are not particularly large.

9. LIMITATION OF THE ALGORITHM

- **High Time Complexity**: With a time complexity of **$O(n^3)$**, it is not scalable for very large graphs with thousands or millions of vertices.
- **Space Complexity**: The algorithm requires **$O(n^2)$** space to store the distance matrix, which may not be feasible for large graphs.
- **Negative Cycles**: If a graph contains a negative weight cycle, the algorithm will not work correctly, as the shortest path would not be well-defined (distance would decrease indefinitely).
- **Not Optimal for Sparse Graphs**: For graphs with relatively few edges, the Floyd-Warshall algorithm is not the most efficient option since it checks all pairs of nodes regardless of whether they are connected.

10. CONCLUSION

In conclusion, the Floyd-Warshall Algorithm is an elegant and effective solution for calculating the shortest paths between all pairs of nodes in a weighted graph. It is highly reliable for graphs with negative weights and provides an efficient means of obtaining these shortest paths through a dynamic programming approach. However, for very large graphs, its **$O(n^3)$** time complexity can become a bottleneck, and other algorithms may be more appropriate in such cases.

11. CODE STRUCTURE

I have implemented this code in Python language.

I have used Pycharm Tool for the compilation and execution of the program as it is very efficient tool for the execution.

The code is organized into various components, with the main program logic contained in the following sections:

- Maze Generation (generate_maze function)
- Pathfinding using Floyd-Warshall Algorithm (floyd_warshall function)
- Visualization (visualize_maze function and MazeApp class)
- User Interface (UI) with Tkinter (MazeApp class, GUI elements, and event handling)

12. MAIN DATA STRUCTURE USED

1. 2D Arrays (Matrices)

- **Description:** The grid (maze) is represented as a 2D array (matrix), where each element in the matrix corresponds to a cell in the maze. The maze itself is represented by a 2D numpy array (grid).
- **Use Case**: The grid stores the maze layout, and the floyd_warshall algorithm operates on this 2D array to compute the shortest path distances.
- **EXAMPLE**:

```
grid = np.zeros( shape: (size, size), dtype=int) # Create a grid with all paths (0)
```

2. Numpy Arrays

- **Description**: numpy arrays are used to store both the maze grid and the distance matrices for the **Floyd-Warshall algorithm**.

Use Case:

- **dist** stores the shortest distance between each pair of nodes.
- **prev** stores the previous node in the shortest path for each cell, allowing for path reconstruction after running Floyd-Warshall.

- **EXAMPLE:**

```
dist = np.full((size, size), np.inf)  # Distance matrix initialized to infinity
prev = np.full((size, size, 2), -1, dtype=int)  # Previous node matrix to track the path
```

3. Tuples

- **Description**: Tuples are used to represent positions (coordinates) in the maze, such as the start and end positions or during the reconstruction of the shortest path.
- **Use Case**: Tuples are used to represent the **(row, column)** positions of the start and end points, as well as during the path backtracking (to reconstruct the path in the maze).
- **EXAMPLE:**

```
start = (0, 0)
end = (size - 1, size - 1)
```

4. Lists

- **Description**: Lists (in Python) are used to store the shortest path points (coordinates) in a list that will later be drawn as a red line in the maze.
- **Use Case**: The path list stores the coordinates of the shortest path from the start to the end, which will be highlighted in the visualization.
- **EXAMPLE**:

```
path = []
```

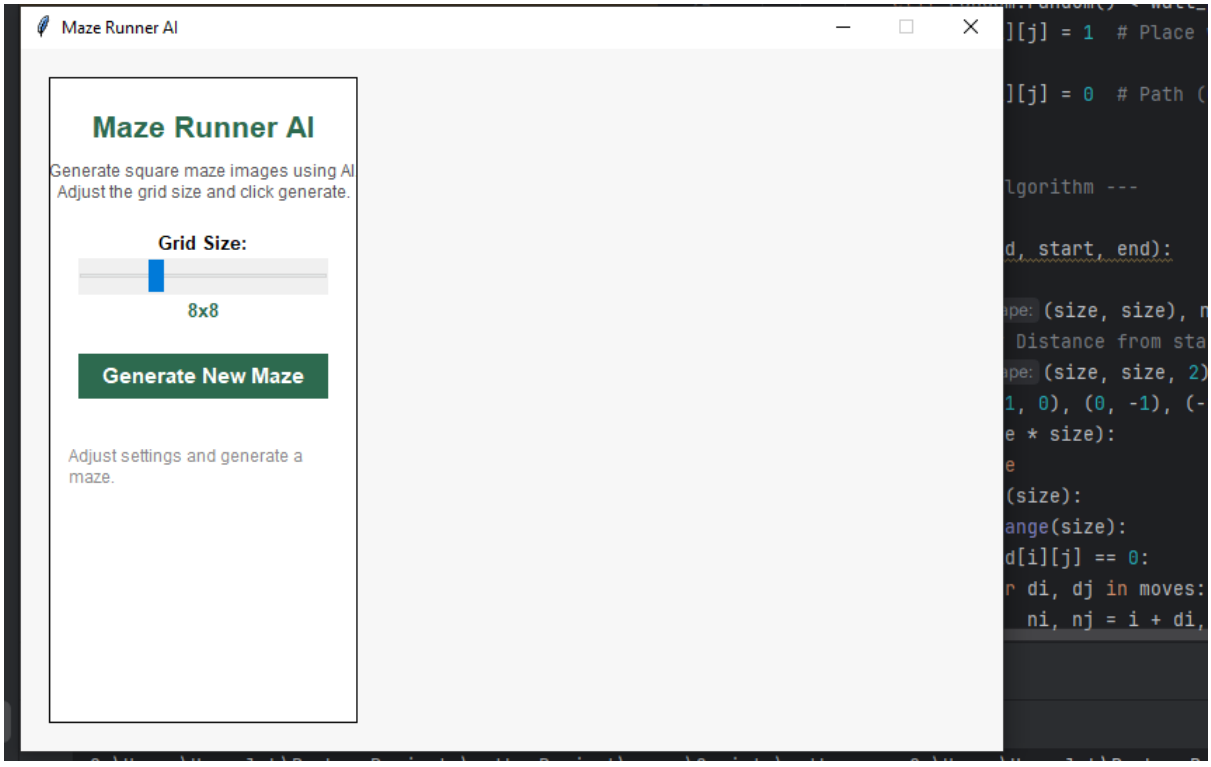
5. Stack (Implicit in Path Reconstruction)

- **Description**: The process of path reconstruction in the Floyd-Warshall algorithm involves traversing backwards from the end point to the start point. This is effectively **a stack** operation, where you keep pushing coordinates into a list (which behaves like a stack) and then pop them off as you traverse back to the start.
- **Use Case**: The shortest path is reconstructed by "pushing" the coordinates onto the list (path) during backtracking, then reversing the list to get the correct order.
- **EXAMPLE**:

```
path.append(tuple(current))
current = tuple(prev[tuple(current)])
```

OUTPUT:

MENU

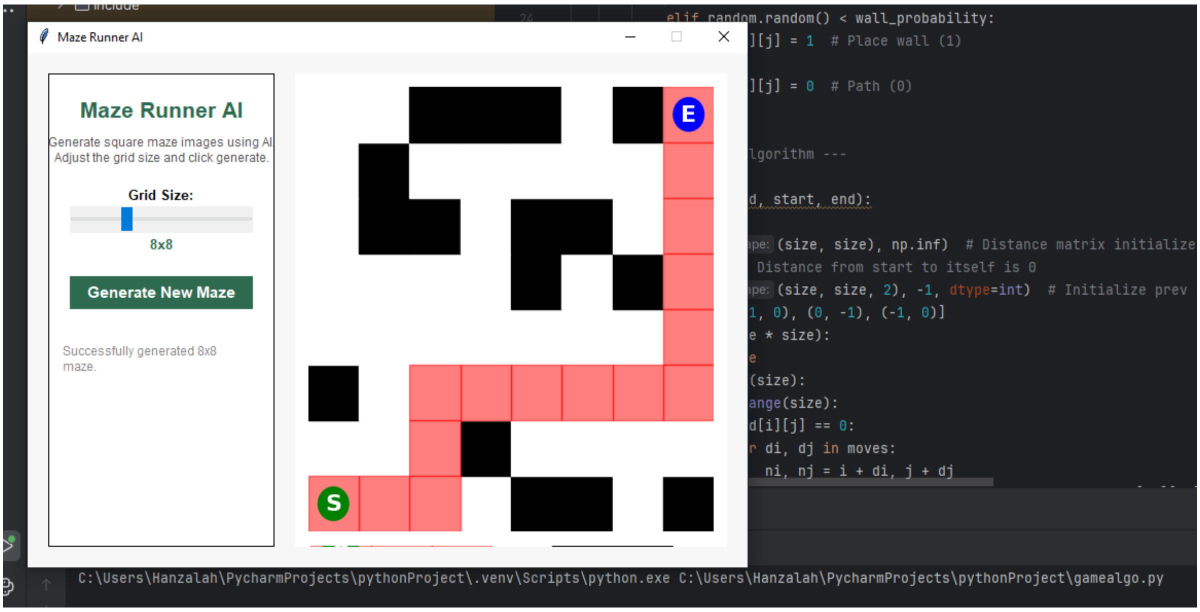


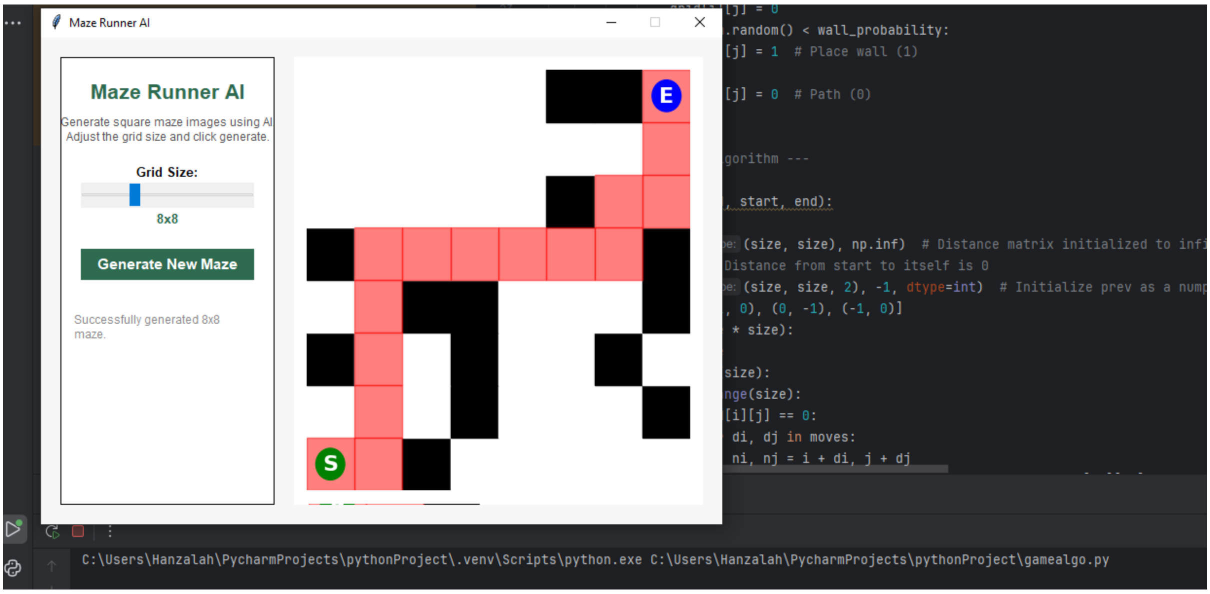
You can select the size of the maze that you want to generate.

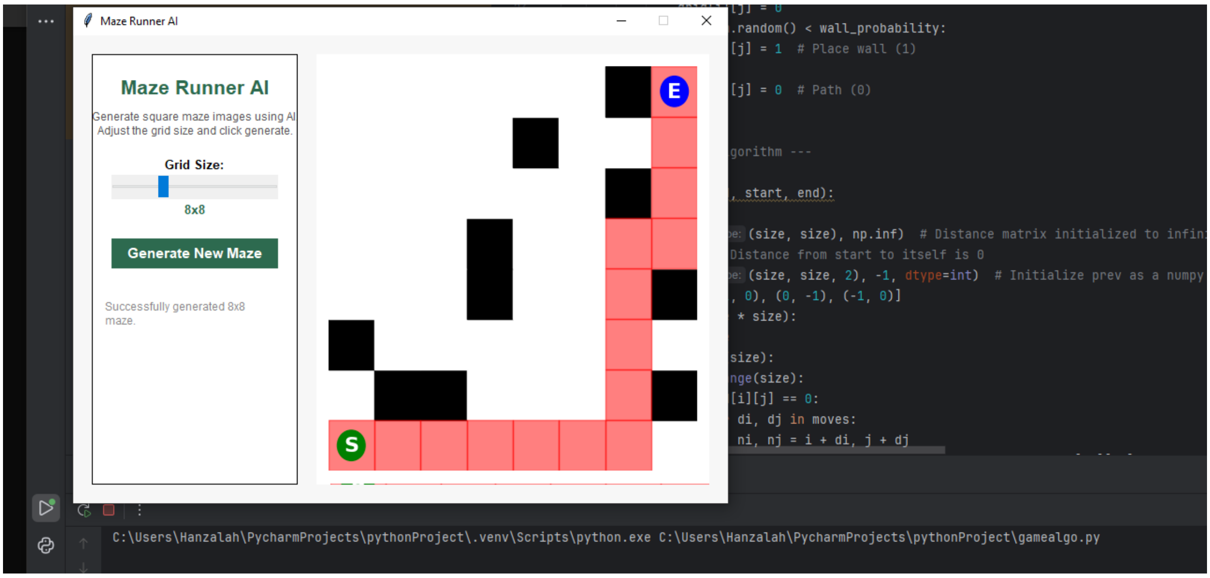
After selecting the maze size click on the generate new maze button so after clicking you'll get the maze that shows the shortest possible path that can be exist in the maze, it generate randomly maze.

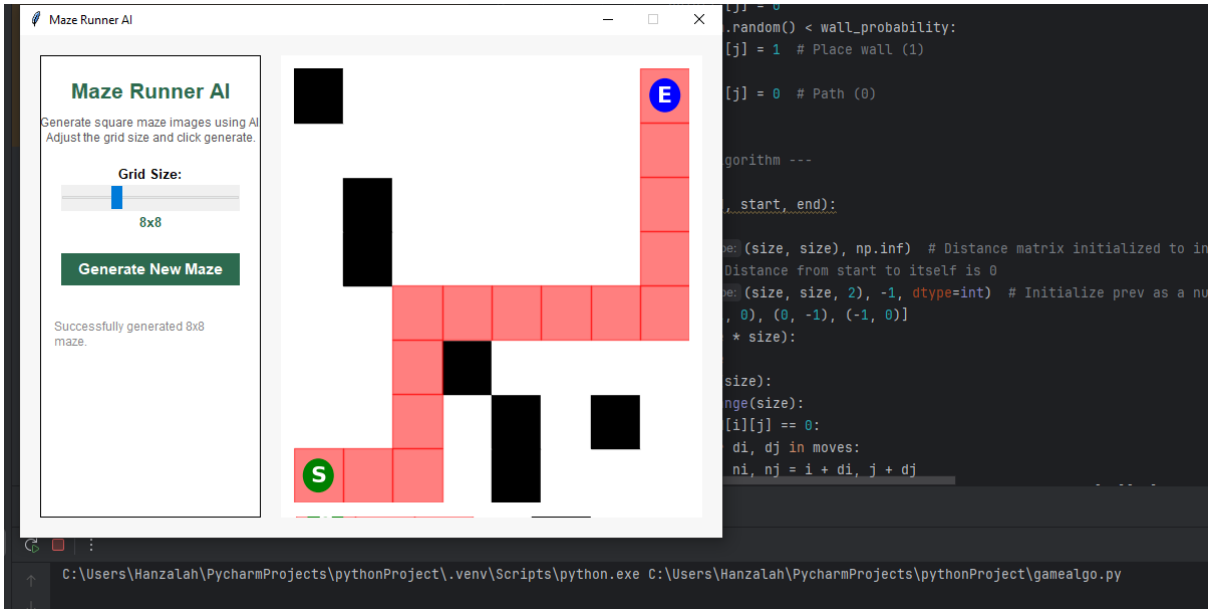
I have attached 2-3 outputs of the randomly maze that generates shortest possible path according to the algorithm.

The output is given below:


Maze Runner AI
Maze Runner AI
Generate square maze images using AI
Adjust the grid size and click generate.
Grid Size:
8x8
Generate New Maze
Successfully generated 8x8 maze.
E
S
elif random.random() < wall_probability:
][j] = 1 # Place wall (1)
][j] = 0 # Path (0)
lgorithm ---
d, start, end):
(size, size), np.inf) # Distance matrix initialize
Distance from start to itself is 0
(size, size, 2), -1, dtype=int) # Initialize prev
1, 0), (0, -1), (-1, 0)]
e * size):
(size):
ange(size):
d[i][j] == 0:
r di, dj in moves:
ni, nj = i + di, j + dj
C:\Users\Hanzalah\PycharmProjects\pythonProject\.venv\Scripts\python.exe C:\Users\Hanzalah\PycharmProjects\pythonProject\gamealgo.py


Maze Runner AI
Maze Runner AI
Generate square maze images using AI
Adjust the grid size and click generate.
Grid Size:
8x8
Generate New Maze
Successfully generated 8x8 maze.
E
S
.random() < wall_probability:
[j] = 1 # Place wall (1)
[j] = 0 # Path (0)
gorithm ---
, start, end):
(size, size), np.inf) # Distance matrix initialized to inf
Distance from start to itself is 0
(size, size, 2), -1, dtype=int) # Initialize prev as a nump
, 0), (0, -1), (-1, 0)]
* size):
size):
nge(size):
[i][j] == 0:
di, dj in moves:
ni, nj = i + di, j + dj
C:\Users\Hanzalah\PycharmProjects\pythonProject\.venv\Scripts\python.exe C:\Users\Hanzalah\PycharmProjects\pythonProject\gamealgo.py


Maze Runner AI
Maze Runner AI
Generate square maze images using AI
Adjust the grid size and click generate.
Grid Size:
8x8
Generate New Maze
Successfully generated 8x8 maze.
E
S
.random() < wall_probability:
[j] = 1 # Place wall (1)
[j] = 0 # Path (0)
gorithm ---
, start, end):
(size, size), np.inf) # Distance matrix initialized to infin
Distance from start to itself is 0
(size, size, 2), -1, dtype=int) # Initialize prev as a numpy
, 0), (0, -1), (-1, 0)]
* size):
size):
nge(size):
[i][j] == 0:
di, dj in moves:
ni, nj = i + di, j + dj
C:\Users\Hanzalah\PycharmProjects\pythonProject\.venv\Scripts\python.exe C:\Users\Hanzalah\PycharmProjects\pythonProject\gamealgo.py



Github Link

https://github.com/iamhanzalah86/ANALYSIS-OF-ALGORITHM/tree/main/analysis%20of%20algorith%20project