**Table of Contents**

1. Abstract
2. Introduction
3. Methodology
4. Implementation Details
5. Experimental Results
6. Discussion and Future Work  
   6.1 Future Work  
   6.2 Discussion
7. Conclusion
8. References

## 1. Abstract

To explore, understand and compare the pathfinding algorithms is the main aim of this project. Four algorithms Were tested here**, the Breadth-First Search (BFS), the Depth-First Search (DFS), the A\*, and the Greedy BFS**, these algorithms are applied to a maze-solving problem with varying obstacle densities configurations **(0%, 10%, 30%, and 50% obstacles)** to assess their efficiency. Performance metrics including **path length and exploration length** Were measured for each algorithm in all different obstacle densities. The results show that while A\* consistently provides the shortest path, its exploration length increases significantly with higher obstacle densities. In contrast, BFS and Greedy BFS exhibit more balanced results, with Greedy BFS sometimes outperforming others in terms of exploration efficiency. DFS, while showing a high exploration length, struggles in higher obstacle environments, often requiring more exploration steps to find a path. This study highlights the importance of algorithm selection based on the maze complexity and obstacle distribution.

## 2. Introduction

The maze solving problems are a common challenge in computer science world, these problems are used to test how Ill the search algorithms work. It's important in fields like robotics, AI navigation, and video game design. The goal is to find the best path from the start to the goal in a maze, while avoiding obstacles along the way. As the percentage of obstacles in the maze increases, solving the maze becomes harder, so choosing the right algorithm is essential for efficiency and accuracy.

In this project, I focus on four popular algorithms: BFS (Breadth-First Search), DFS (Depth-First Search), A\*, and Greedy BFS. Each algorithm has a different way of exploring the maze and performs differently depending on how complex the maze is. I tested these algorithms on mazes with different obstacle densities to see how they handle varying levels of difficulty. The main factors I looked at to compare their performance Ire:

1. **Path length**: the shortest path from start to goal.
2. **Exploration length**: how many cells the algorithm explores before finding the solution.

Thes lengths are calculated for all 4 algorithms in all different density of obstacles in the maze. This project not only just show the efficiency of the maze, but also gives me a clear picture of the change in behaviour of the maze when the density of obstacles increases.

## 3. Methodology

The maze solving problem was implemented using the Python programming language, employing the Pyamaze library for maze generation and visualization. The four algorithms under study—BFS, DFS, A\*, and Greedy BFS—Were implemented as separate modules and imported dynamically based on user input.

### 3.1 Maze Generation

The mazes were generated with varying obstacle densities, represented by CSV files with 0%, 10%, 30%, and 50% obstacles. These obstacles were randomly distributed in the maze grid, and the algorithms were tasked with finding the shortest path from the start point to the goal while avoiding the obstacles. The grid sizes were set to 50x100, providing a balance between computational complexity and performance evaluation. The main maze is **Maze\_1\_90\_loopPercent.csv**, this maze has 90% loop percentage which means there are more chances to find multiple paths to reach goal within this maze.

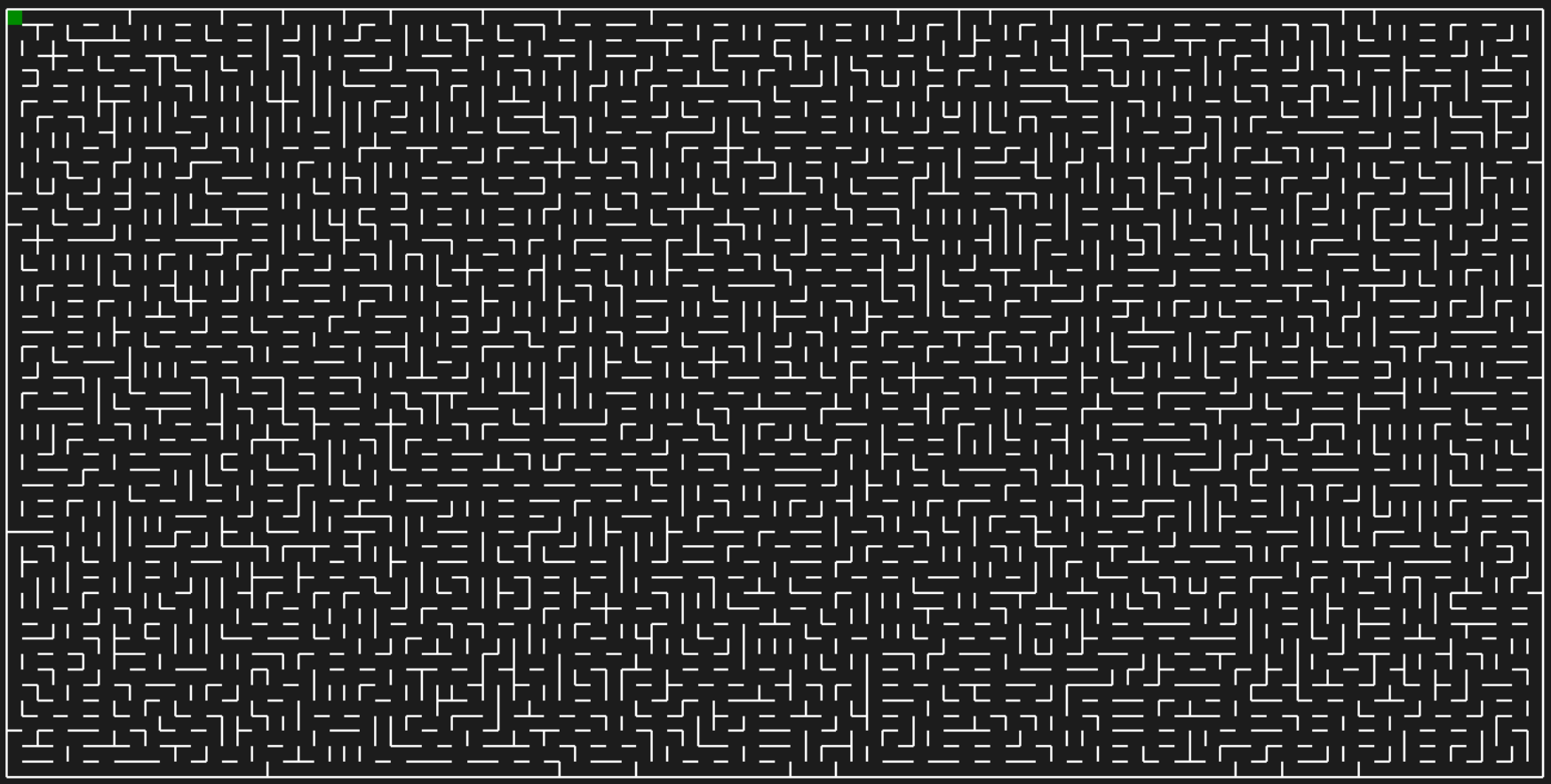
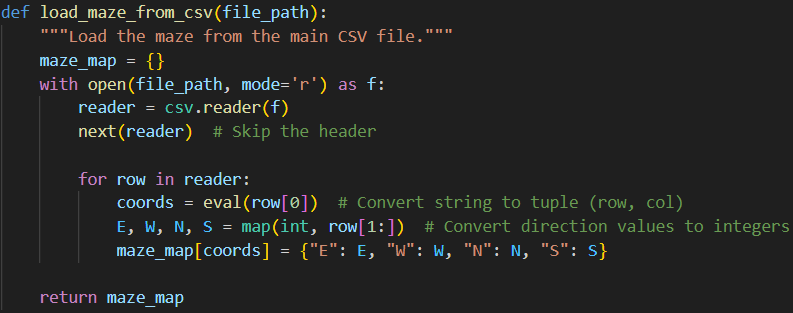


Fig 1: Maze with 90% loop in it. (The main Maze with no obstacles)

Now to introduce the obstacles, so this same maze environment, I had used 0 Create\_\_Obstacles.py file.

This process has 3 functions:

1. Load the csv to get maze environment, (takes the Maze\_1\_90\_loopPercent.csv as input)



Code: 2 Function to Load maze from .csv

1. Add random obstacles within it with given obstacles percentage. (here 50%)

A computer screen shot of text

Description automatically generated

Code 3: Function to add random Obstacles in loaded maze.

1. Save this new maze (old maze + obstacles) in .csv format (here maze\_with\_Obstacles.csv)

A screen shot of a computer code

Description automatically generated

Code 5: Function to same the new maze with obstacles in .csv format

All these functions are called in main function and run with required input.

A computer screen with text

Description automatically generated

Code 6: Obstacles creation with given percentage.

By running the above code multiple times, I created these .csv (s) with different obstacles percentage.

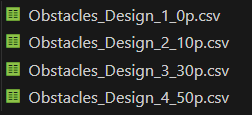


Fig 2: Obstacles .csv files

### 3.2 Algorithm Selection

The four algorithms that are chosen based on their popularity and varying approaches to pathfinding. Let’s discuss then one by one:

1. **BFS (Breadth-First Search)**:

**Approach**: BFS is an uninformed search algorithm that explores the maze level by level, starting from the initial position and expanding outward. It guarantees the shortest path in an unweighted grid like the one used in this project.

**Use Case**: Ideal for finding the shortest path in a maze with a relatively small number of obstacles.

1. **DFS (Depth-First Search)**:

**Approach**: DFS explores as far down a path as possible before backtracking. It doesn't guarantee the shortest path, and in dense mazes, it may explore large portions of the maze unnecessarily which I will show you in result section.

**Use Case**: Often used in environments where a solution is needed quickly, but not necessarily the optimal path.

1. **A\***:

**Approach**: A\* is a heuristic-based search algorithm that uses both the cost to reach a node and an estimate of the remaining cost to the goal (calculated via a heuristic). A\* generally provides the optimal path if the heuristic is well-designed.

**Use Case**: Great for finding the shortest path in complex mazes, especially when there are obstacles scattered in the maze.

1. **Greedy BFS**:

**Approach**: Greedy BFS uses a heuristic that evaluates the cost to the goal. Unlike A\*, it doesn't account for the cost of getting to the current node but focuses purely on reaching the goal as quickly as possible.

**Use Case**: Often faster than A\* but can lead to suboptimal paths because it doesn't consider the entire problem space, like in video games for NPC

### 3.3 Evaluation Metrics

The two key performance metrics were:

* **Path Length**: The number of steps taken from the starting point to the goal. A lower path length indicates that the algorithm found the shortest path.
* **Exploration Length**: The number of cells the algorithm visited (including both explored and unexplored cells). A lower exploration length indicates that the algorithm was more efficient in finding the solution.

Both metrics were collected for each algorithm at four different obstacle densities (0%, 10%, 30%, 50%).

### 3.4 Environment and Tools

* **Programming Language**: Python 3.9
* **Libraries Used**:
  + **Pyamaze**: For maze generation, visualization, and handling maze objects.
  + **Tkinter**: For building a simple graphical user interface (GUI) to select the algorithm and obstacle density.
  + **Importlib**: To dynamically load and execute the algorithm modules.
* **Maze Files:** Each obstacle density was stored in a CSV file containing maze configurations.

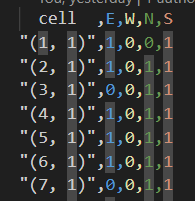
## 4. Implementation Details

### 4.1 Maze Visualization and Agent Movement

A main aspect of this project is the ability to track the agent's movement through the maze during the execution of the various algorithms by visualizing it. This visualization allowed us to observe not just the final solution, but also the exploration behaviour and the efficiency of each algorithm in real-time.

The **Pyamaze** library was utilized to create a graphical representation of the maze and monitor the agent’s pathfinding behavior. Each maze is depicted as a grid of cells, where each cell represents either an obstacle or an open path. The agent’s movement from the start point to the goal is highlighted by tracing the steps the algorithm takes.

Key elements of the maze visualization include:

1. **Maze Representation**: The maze is displayed as a grid with 50 rows and 100 columns. Open cells are represented in white, and walls are shown in black. When these walls form a box, it makes an obstacle. The agent’s progress is tracked visually as it moves through the grid.

On database, .csv is visualized like this, here cell is representing the cell number (1, 1), (2, 1) and E, W, N S are representing the direction of movement within the cell where 1 means wall and 0 means no wall.

Fig : Maze.csv structure

1. **Agent Movement**: The agent is visualized as a colored square. Different algorithms use different colors for the agent’s path:
   * **BFS** and **Greedy BFS**: Red
   * **A\***: Yellow
   * **DFS**: Blue The agent’s position is updated dynamically, providing a visual representation of the agent’s movement as it explores the maze and reaches the goal.

A computer screen shot of text

Description automatically generated

Fig : Work on agent Moment and visualization (this is for A\*)

1. **Path Tracing**: As the algorithm progresses, the agent traces the path it takes to the goal. This allows us to understand not only the solution (the path to the goal) but also the exploration process itself. The traced path is shown step-by-step as the agent moves, (just for fats moment I used delay=1) and we can observe how efficiently the agent navigates the maze.

A black background with white text

Description automatically generated

Fig : Path tracing of agents. (this is for A\* only)

1. **Goal and Exploration**: The goal position is marked with a green square. The exploration process is shown by the agent’s footprints or the cells that were visited during the search. This helps us visualize the total number of cells explored by the algorithm and compare it across different algorithms.

A black background with white text

Description automatically generated

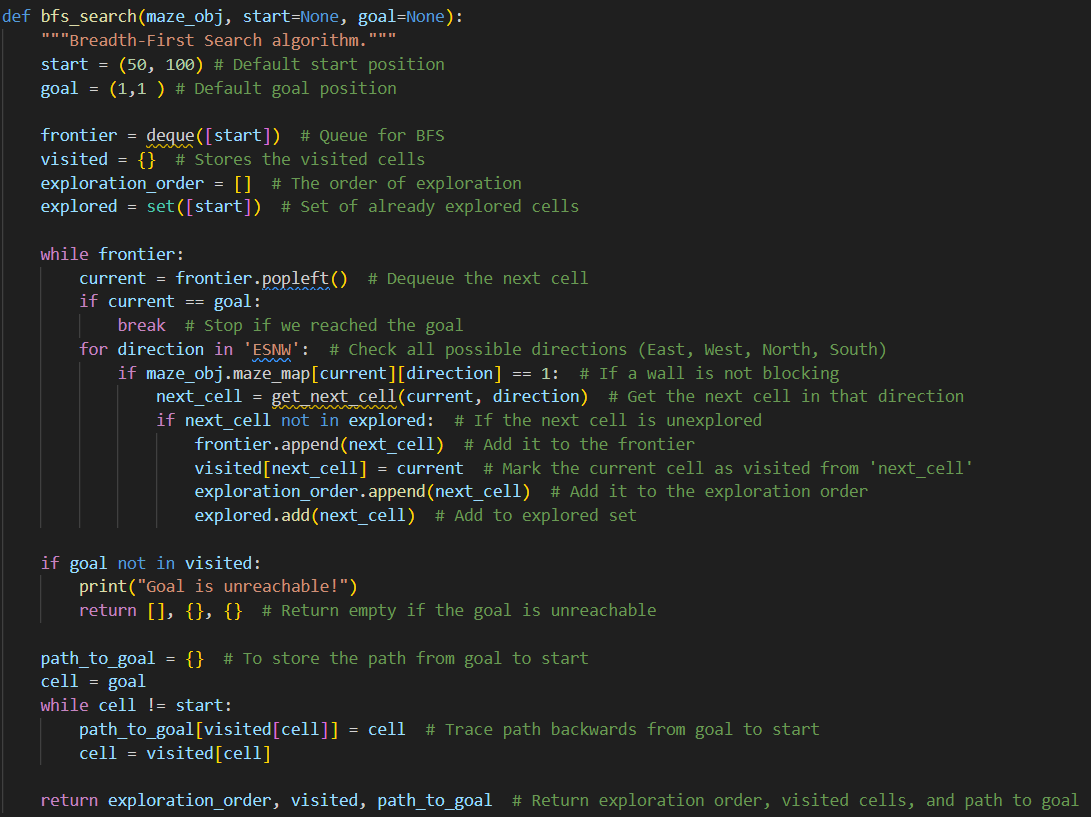
Fig : Code behind text Labelling.

The maze visualization provides a clear and engaging way to observe the behavior of each algorithm as it tackles different maze configurations and obstacle densities. It also highlights the differences in how the algorithms explore the maze and find the shortest path.

**Specific Algorithm Behavior and Insights**

This section provides a deeper dive into how each algorithm behaves across different maze configurations, with a particular focus on how they handle varying obstacle densities.

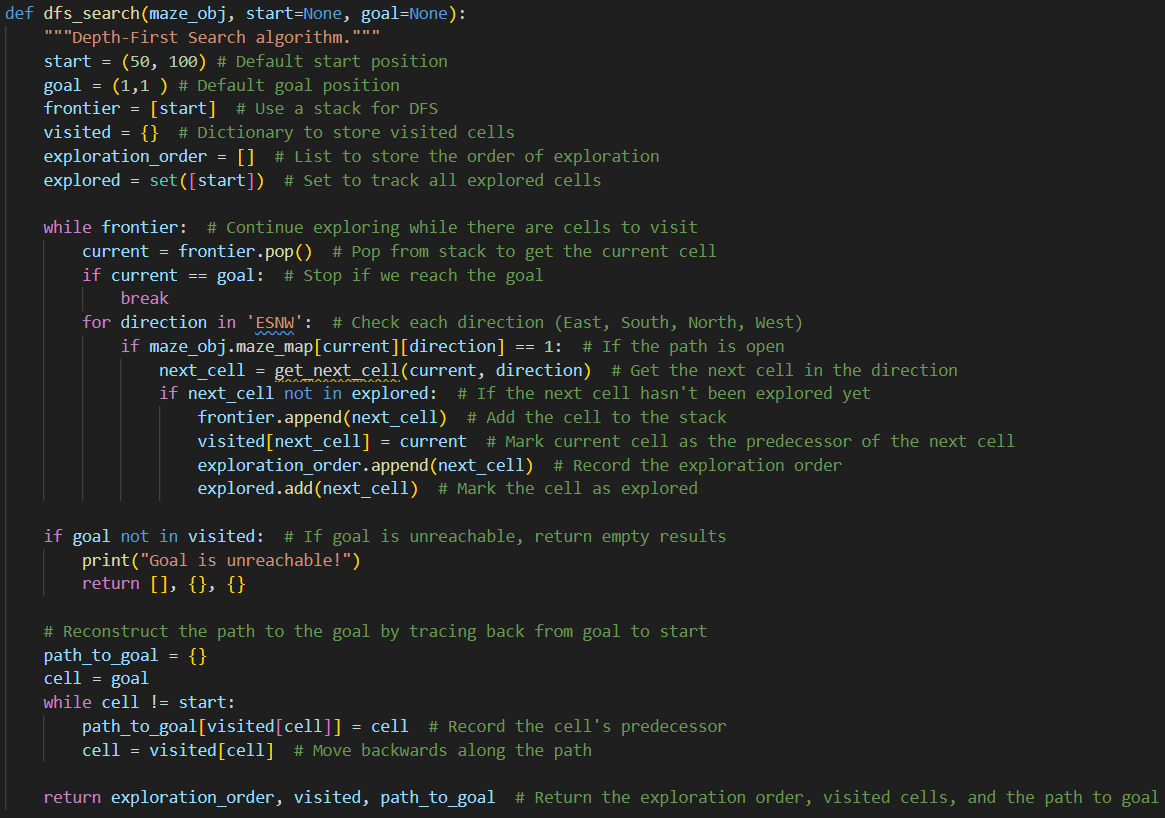
1. **Breadth-First Search (BFS)**
   * **Behavior**: BFS explores the maze level by level, systematically visiting all possible paths before moving deeper. This guarantees that the shortest path to the goal will be found, but it often results in high exploration lengths, especially in larger mazes.
   * **Edge Case - High Obstacle Density**: As the obstacle density increases (30% or 50%), BFS starts to explore larger sections of the maze before finding a viable path. The agent’s movement, as visualized, shows a large amount of redundant exploration, especially in the 50% obstacle maze. The path is found, but BFS takes a long time to reach it, as the exploration length grows disproportionately.



Code: BFS Algorithm logic.

* + The code implements of Breadth-First Search (BFS) is to **find** the shortest path in a maze, loaded from a CSV file. It starts from a given starting point and explores all neighboring cells in all directions (East, South, West, North), marking visited cells and avoiding revisits. Once the goal is reached, it reconstructs the path from the goal to the start using the visited cells. The maze and exploration are visualized with agents tracing the BFS path and the final path to the goal.

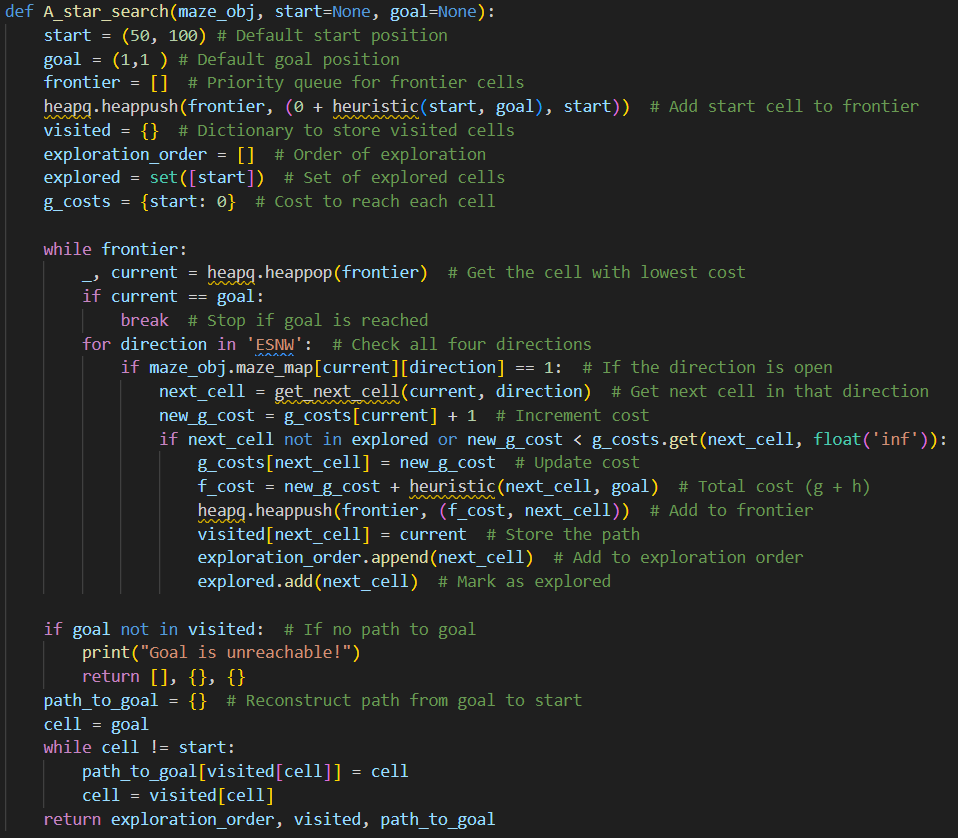
1. **Depth-First Search (DFS)**
   * **Behavior**: DFS explores one path as far as possible before backtracking. While it does not guarantee the shortest path, it will eventually find a path to the goal if one exists. However, it can take a considerable amount of time to find the solution, especially in mazes with many dead-ends.
   * **Edge Case - Sparse Obstacles**: In mazes with 0% or 10% obstacles, DFS can explore efficiently since fewer dead-ends or obstacles force the algorithm to backtrack. The agent’s path in the visualization appears somewhat erratic, with frequent backtracking, but the agent reaches the goal in relatively fewer steps than in denser mazes.
   * **Edge Case - Dense Obstacles**: In denser mazes (30% and 50% obstacles), DFS struggles significantly. The algorithm often explores long paths that lead to dead-ends, forcing it to backtrack frequently. The maze visualization shows the agent wandering aimlessly, revisiting areas already explored, which leads to inefficient exploration and longer computation times.



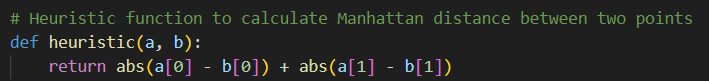
Code : DFS Algorithm Logic

* + This script uses **Depth-First Search (DFS)** to find a path through a maze, loaded from a CSV file. It explores all possible directions from each cell, marking visited cells, and backtracks when necessary. The process is visualized with agents showing the exploration and the path to the goal.

1. **A\* (A-Star)**
   * **Behavior**: A\* is a heuristic-driven search algorithm. It evaluates paths based on both the actual distance traveled so far and an estimate of the remaining distance to the goal. This makes A\* more efficient in terms of exploration compared to BFS, as it prioritizes the most promising paths based on the heuristic.
   * **Edge Case - Sparse Obstacles**: In mazes with 0% or 10% obstacles, A\* operates efficiently, exploring fewer cells while still guaranteeing the shortest path to the goal. The agent moves strategically, avoiding unnecessary paths, and the visualization shows the agent following a more direct route.
   * **Edge Case - Dense Obstacles**: As obstacle density increases, A\* still performs well due to its heuristic-based approach, but the exploration length increases. In a 50% obstacle maze, the agent might explore a larger portion of the maze, but the path length remains relatively short. The agent’s movement is more calculated and direct compared to BFS and DFS, although it requires more exploration than in sparser mazes.



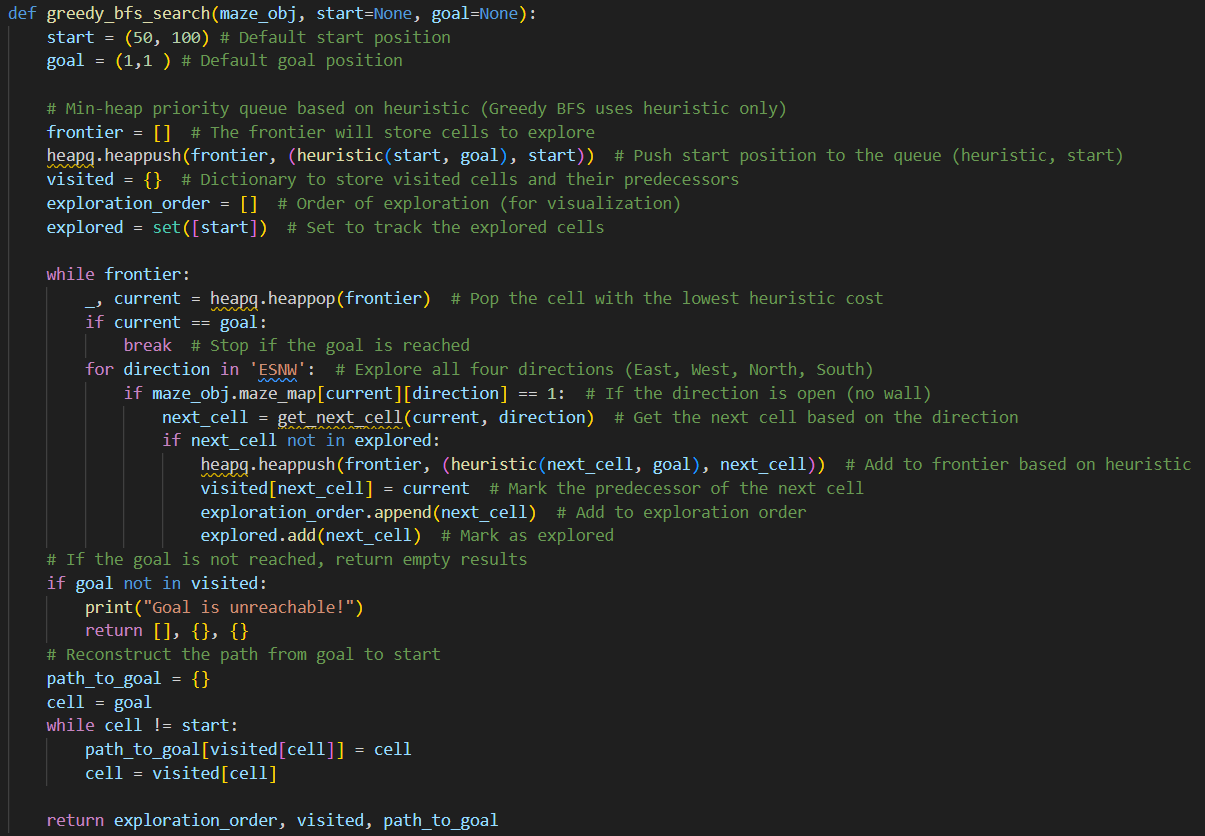
Code : A\* Algorithm Logic



Code : Heuristic function for A\* (Manhattan Distance.

* + This code simulates an A\* pathfinding algorithm on a maze. It loads the maze from a CSV file, adds obstacles, and then uses A\* to find the shortest path from a starting point to a goal. The maze is visualized with agents tracing both the search process and the final path. The code includes functions for handling the maze data, adding obstacles, and calculating the optimal path, making it a useful tool for exploring algorithms in AI and robotics.

1. **Greedy BFS**
   * **Behavior**: Greedy BFS focuses solely on the heuristic distance to the goal, making it faster in terms of exploration compared to both BFS and DFS. However, it does not guarantee the shortest path, as it may take detours in search of a closer goal.
   * **Edge Case - Sparse Obstacles**: Greedy BFS performs relatively well in mazes with low obstacle density (0% and 10%), as the heuristic efficiently guides the agent toward the goal. The agent’s movement in the visualization shows the agent heading toward the goal with fewer unnecessary steps.
   * **Edge Case - Dense Obstacles**: In mazes with higher obstacle densities (30% and 50%), Greedy BFS shows a tendency to take detours as it prioritizes cells that seem closer to the goal. This leads to suboptimal paths, and the exploration length increases as the agent explores unnecessary areas. The agent's movement appears more erratic and less efficient in denser mazes.

****

**Code : Greedy\_BFS Algorithm Logic**

* + This code implements a **Greedy BFS** search algorithm to find a path through a maze. It first loads the maze from a CSV file, then it uses the Greedy BFS algorithm, which explores the maze using a heuristic (Manhattan distance) to prioritize cells closer to the goal. Random obstacles can be added to the maze. The maze is visualized using the pyamaze library, where agents trace both the search process (exploration order) and the final path to the goal. If no path is found, the program will output that the goal is unreachable.

## 5. Experimental Results

In this section, we provide a detailed analysis of the performance of the four algorithms (BFS, DFS, A\*, and Greedy BFS) across different maze configurations with varying obstacle densities (0%, 10%, 30%, and 50%). We evaluate each algorithm based on two key metrics: **path length** and **exploration length**.

**Performance Analysis**

1. **Path Length**
   * **A\***: The A\* algorithm consistently produced the shortest path to the goal. In mazes with 0% and 10% obstacles, the path length remained constant at 149 steps. As the obstacle density increased (30% and 50%), the path length increased slightly to 161 and 231 steps, respectively. Despite this increase, A\* consistently maintained a shorter path than the other algorithms.
   * **BFS**: BFS, like A\*, guarantees the shortest path to the goal. The path length remained the same as A\* in the 0% and 10% obstacle mazes (149 steps). However, as the obstacle density increased, the path length grew to 161 steps at 30% and 231 steps at 50%, similar to A\* but with greater exploration costs.
   * **DFS**: DFS did not consistently find the shortest path and had longer path lengths, especially in denser mazes. In sparse mazes (0% and 10% obstacles), the path length was 167 and 185, respectively. As the obstacle density increased, DFS’s path length grew significantly to 327 steps in the 30% maze and 259 steps in the 50% maze.
   * **Greedy BFS**: Greedy BFS also found relatively short paths, but not as efficiently as A\* or BFS. The path length at 0% and 10% obstacles was 177 and 209, respectively. With 30% and 50% obstacles, Greedy BFS's path length increased to 211 and 271 steps, showing its tendency to take detours in search of the goal.
2. **Exploration Length**
   * **A\***: A\* showed efficient exploration, exploring only as many cells as necessary to find the goal. At 0% and 10% obstacles, the exploration length was 2492 and 2618, respectively. In the 30% obstacle maze, it increased to 3321, and at 50%, it reached 2198. Despite these increases, A\* still maintained an efficient search compared to BFS and DFS.
   * **BFS**: BFS's exploration length was notably high, especially in denser mazes. At 0% and 10%, BFS explored 4998 and 4984 cells, respectively. As the obstacle density increased, the exploration length remained disproportionately large, peaking at 4889 cells at 30% and then dropping to 2389 cells in the 50% obstacle maze.
   * **DFS**: DFS had the shortest exploration length in sparse mazes, with only 301 and 327 cells explored at 0% and 10% obstacles. However, as the obstacle density increased, DFS’s exploration length grew rapidly, reaching 700 cells at 30% and 832 cells at 50%.
   * **Greedy BFS**: Greedy BFS showed intermediate performance, with exploration lengths of 2481 cells at 0% obstacles and 2850 at 10%. The exploration length grew to 4100 and 4465 cells at 30% and 50% obstacles, respectively, as the algorithm took longer detours.

**Insights from Maze Visualizations**

The maze visualizations made it clear that **A\*** was the most efficient in terms of exploration and pathfinding. The agent’s path was more direct, and the exploration length was relatively low. In contrast, **BFS** showed excessive exploration, particularly in denser mazes, where it visited a larger portion of the maze before finding the goal. **DFS** performed well in sparse mazes but struggled significantly in dense mazes due to excessive backtracking. **Greedy BFS** was faster than BFS and DFS but did not always find the optimal path, especially in complex mazes with higher obstacle densities.

## 6. Discussion and Future Work

**6.1 Future Work**

Several improvements can be made to the project in future iterations:

* *Optimization of A and BFS*\*: Both A\* and BFS can be optimized using more advanced data structures (e.g., priority queues for A\*).
* **Additional Algorithms**: Algorithms like Dijkstra’s and Bidirectional BFS could be included for further comparison.
* **Dynamic Obstacles**: Implementing dynamic obstacles that change during the maze-solving process could add an interesting layer of complexity.
* **Visual Enhancements**: The GUI could be enhanced with visual features such as real-time exploration tracking and interactive maze design.

**6.2 Discussion**

The results highlight the trade-offs betIen algorithm performance and maze complexity. A\* provides the shortest path but at the cost of increased exploration time, particularly in obstacle-dense environments. BFS, while consistent, is inefficient in terms of exploration. DFS, while quick to find solutions in sparse mazes, becomes inefficient as the number of obstacles grows. Greedy BFS shows potential for a balanced approach, though it still requires significant exploration in complex mazes.

## 7. Conclusion

This study compares four pathfinding algorithms—A\*, BFS, DFS, and Greedy BFS—across varying obstacle densities in maze environments. The experimental results demonstrate that while A\* consistently finds the shortest path, it is not always the most efficient in terms of exploration length. BFS and Greedy BFS provide a more balanced trade-off, while DFS struggles with higher obstacle densities. These findings emphasize the importance of choosing the right algorithm based on the maze's complexity and obstacle distribution.

## 8. References

1. Ahuja, R. K., Magnanti, T. L., & Orlin, J. B. (1993). *Network Flows: Theory, Algorithms, and Applications*. Prentice Hall.
2. Russell, S., & Norvig, P. (2020). *Artificial Intelligence: A Modern Approach* (4th ed.). Pearson.
3. Hart, P. E., Nilsson, N. J., & Raphael, B. (1968). A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2), 100-107.
4. Mitchell, M. (2009). *An Introduction to Genetic Algorithms* (2nd ed.). MIT Press