SSN COLLEGE OF ENGINEERING, KALAVAKKAM

Department of Computer Science and Engineering

UCS2504 – Foundations of Artificial Intelligence
3 Year CSE (Semester 5)

Academic Year 2022-23
Batch: 2021- 2025

A-Mazing Adventures

Mini Project

Faculty Incharge : Dr. Saritha M

Project Students:

Pooja Premnath (3122 21 5001 066)
V S Pranav (3122 21 5001 069)
K G Sanjai Balajee (3122 21 5001 092)
Sanjjit S (3122 21 5001 094)

Problem Statement:

This project sets out to construct a Pygame application that serves as a sophisticated visual representation of two pivotal search algorithms: A* (A-star) and BFS (Breadth-First Search). The crux of this application lies in the dynamic generation of mazes, achieved through the precise implementation of Prim's algorithm. Unlike conventional maze generators, the mazes produced herein are not arbitrary; they are meticulously engineered to adhere to a strict criterion of solvability, ensuring the existence of a viable path from a user-defined starting point to an endpoint within the labyrinth.
The user interface gives users the capability to actively shape the maze-solving process by specifying the starting and ending points within the maze. As users establish the maze's initial conditions, the application orchestrates a meticulous, step-by-step visualization of the maze-solving process. The chosen algorithm, whether A* with its heuristic prowess or BFS with its systematic exploration strategy, is brought to life on the screen. Each iteration represents a computational step, revealing the algorithm's decision-making process and unfolding the pathfinding progress.

Objectives:

The primary objectives of this project are to create an interactive Pygame application that visualizes two fundamental artificial intelligence searching algorithms: A* (A-star) and BFS (Breadth-First Search). The project seeks to implement a dynamic maze generation using Prim's algorithm, ensuring the solvability of the mazes generated. The graphical representation of the mazes will be structured as a graph, with cells as nodes and edges as possible connections. User interaction is a key objective, allowing users to define starting and ending points within the maze through an intuitive interface. Additionally, the application aims to provide users with the option to select between A* and BFS algorithms for real-time visualization, highlighting the step-by-step process of solving the maze.

Solution:

The proposed solution involves the development of a Pygame application that seamlessly integrates maze generation, user interaction, algorithmic visualization, and performance metrics. Prim's algorithm will be employed to generate solvable mazes, which will then be represented as graphs. The user interface, designed for ease of interaction, enables users to set start and end points within the maze. A* and BFS algorithms will be implemented, encapsulating their respective logic within modular functions or classes. Real-time visualization will showcase the evolving path and explored cells during the maze-solving process.

Motivation:

Navigating through mazes is a classic problem that transcends various domains, from robotics to gaming. This Pygame project seeks to offer a captivating exploration into the inner workings of two fundamental search algorithms, A* and BFS, that lie at the heart of maze-solving strategies. The dynamic maze generation, powered by Prim's algorithm, ensures that the mazes created are not just random puzzles but intricately designed challenges with a guaranteed solution path. By allowing users to actively shape the maze-solving process through defining start and end points, the project provides an interactive and educational experience. As the chosen algorithm unfolds on the screen, each step becomes a visual testament to the algorithm's decision-making prowess, offering a unique insight into the world of computational problem-solving. This project not only serves as a practical application of algorithms but also aims to kindle a curiosity for the art and science of maze navigation, making it an engaging journey for both enthusiasts and learners alike.

Requirements:

Software:

The provided code is written in Python and uses the Pygame library for creating a graphical user interface. To run this code, you will need the following software requirements:
1. Python: Ensure that Python is installed on your system. You can download and install Python from the official website: https://www.python.org/downloads/
2. Pygame: Install the Pygame library using pip, which is the package installer for Python.

Functional Requirements for the Maze Game:
1. Grid Generation: The game should generate a grid-based maze with customizable dimensions.
2. Node Types: The maze should support different node types such as blank, start, end, wall, mud, and dormant.
3. Pathfinding Algorithms: The game should implement pathfinding algorithms such as Breadth-First Search (BFS) and A Search for finding paths from the start to the end node.
4. User Interaction: Users should be able to interact with the maze by setting start and end points, drawing walls, and initiating pathfinding algorithms.
5. Visualization: The game should visually display the maze, pathfinding process, and the final path from start to end.

Non-Functional Requirements for the Maze Game:
1. Performance: The game should provide smooth and responsive interaction even with large maze sizes and complex pathfinding algorithms.

2. Usability: The user interface should be intuitive and easy to use, allowing users to interact with the maze and algorithms seamlessly.
3. Scalability: The game should be able to handle mazes of varying sizes and complexities without significant performance degradation.
4. Reliability: The pathfinding algorithms should consistently find valid paths in different maze configurations.
5. Compatibility: The game should be compatible with different operating systems and screen resolutions.

These requirements outline the essential functionalities and qualities that the maze game should possess to provide an engaging and user-friendly experience.

Design Alternatives:

- In lieu of Pygame, alternative graphics libraries such as Tkinter, Pyglet, or Kivy may be considered by the developer for crafting the graphical user interface.
- The exploration of various pathfinding algorithms, including Depth-First Search, Dijkstra's algorithm, or other heuristic search algorithms, is suggested, in addition to the initial consideration of Breadth-First Search and A* Search.
- Rather than utilizing a class-based representation for nodes, the developer might contemplate adopting a matrix-based representation for the maze grid.
- Different methods of user interaction could be explored, such as enabling users to draw the maze themselves, importing maze layouts from external files, or incorporating a maze editor tool.
- The enhancement of visual effects is proposed through the addition of animations, sound effects, or the incorporation of diverse themes for both the maze and nodes.
- The developer is advised to consider optimizing the performance of pathfinding algorithms and grid rendering, particularly for larger maze sizes. This could involve implementing parallel processing or incorporating algorithmic optimizations to enhance overall efficiency.
- These design alternatives offer the developer opportunities to explore diverse approaches in implementing the maze game, potentially augmenting its functionality, user experience, and performance.

Usage of Algorithms for Solving Mazes

The A* (A-star) and BFS (Breadth-First Search) algorithms stand as fundamental tools in solving maze navigation challenges, each with distinctive approaches and applications. A* is renowned for its efficiency in finding the shortest path by employing a heuristic evaluation that combines the cost incurred and an optimistic estimate of the remaining distance. This heuristic-driven approach enhances its ability to prioritize promising paths, making it particularly effective in scenarios where optimizing for both time and space is crucial. On the other hand, BFS explores the maze systematically, layer by layer, examining all possible paths from the start point outward. Although BFS ensures the discovery of the shortest path, its computational complexity may render it less efficient for larger mazes. Both algorithms contribute significantly to maze-solving strategies, with A* excelling in scenarios that demand optimality, and BFS providing reliability in scenarios where exhaustively exploring all possibilities is acceptable.

Methodology:

Maze Generation:

- Implementation of Prim's algorithm for random maze generation.
- Ensuring the generated maze is solvable, guaranteeing the existence of a path from the start to the end point.

Graph Representation:

- Representing the maze as a graph, where individual cells constitute nodes, and edges represent feasible connections between neighboring cells.

User Interaction:

- User-friendly functionality allowing users to set starting and ending points through mouse clicks.
- Provision for resetting the maze and selecting new start and end points.

Algorithm Selection:

- Implementation of A* and BFS algorithms for pathfinding.
- User interface to choose between A* and BFS for solving the maze.

Pathfinding Visualization:

- Real-time visualization of the algorithm solving the maze.
- Highlighting explored cells, cells in the final path, and other relevant information during the search.

Algorithm:

A* Search Algorithm

- **Initialize data structures:**
    - Create a priority queue to manage open nodes during the search.
    - Initialize a set to keep track of visited nodes.
    - Create a dictionary to store the best-known path cost (g_values) to each node.
- **Add the start node to the priority queue with an initial priority of the heuristic value.**
    - Use the A* heuristic function to estimate the cost from the start node to the goal.
    - While the priority queue is not empty:
        - a. Pop the node with the lowest priority from the priority queue.
        - b. If the popped node is the goal, reconstruct and return the path.
        - c. Mark the popped node as visited.
        - d. For each neighbor of the current node:
            - i. If the neighbor is already visited or is a wall, skip it.
            - ii. Calculate the tentative path cost from the start to the neighbor.
            - iii. If the neighbor is not in the priority queue or the new path cost is lower:
    - Update the path cost in the g_values dictionary.
    - Push the neighbor to the priority queue with a priority value based on the sum of the path cost and heuristic.
- **If the priority queue is empty and the goal has not been reached, there is no path.**

BFS Algorithm:

```
1. Create a queue and enqueue the starting node.
2. Create a set to track visited nodes.
3. While the queue is not empty:
  a. Dequeue a node from the queue.
  b. If the dequeued node is the goal, reconstruct and return the path.
  c. Mark the dequeued node as visited.
  d. For each valid neighbor of the dequeued node:
    i. If the neighbor is not visited and is not a wall:
      - Enqueue the neighbor.
      - Mark the neighbor as visited.
      - Set the parent of the neighbor to the dequeued node.
4. If the queue is empty and the goal has not been reached, there is no path.
```

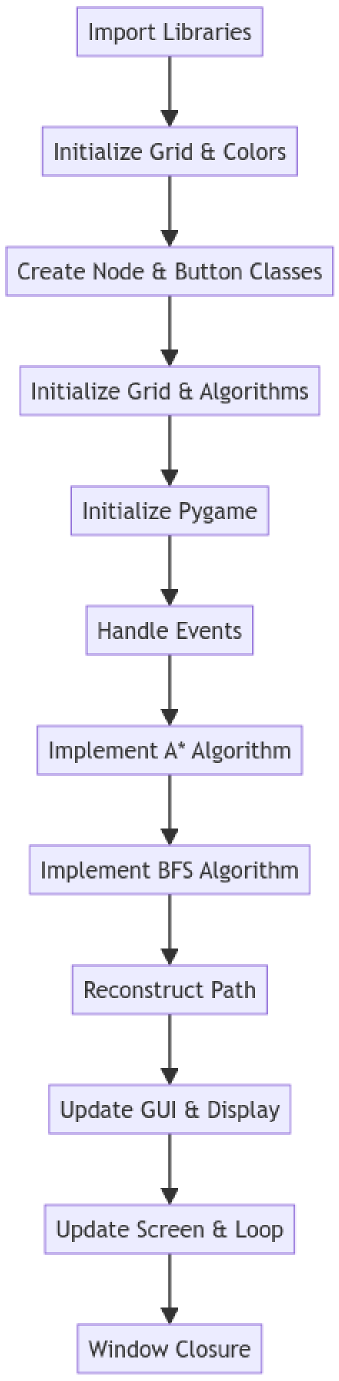
Path Reconstruction:
- After reaching the goal, reconstruct the path by backtracking from the goal to the start using the parent pointers set during the search.

<u>Main Program:</u>

1. Import the necessary libraries, such as pygame.
2. Define the colors to be used in the grid.
3. Create a Button class to handle button functionality.
4. Create a Node class to represent each cell in the grid.
5. Set the width, height, and margin of each grid location.
6. Create a 2-dimensional array to represent the grid.
7. Set the start and end points for the pathfinder.
8. Initialize the pygame module and set the screen size.
9. Create buttons for the A and BFS algorithms.
10. Set up the main program loop.
11. Handle events, such as mouse clicks and key presses.
12. Update the grid based on user interactions.
13. Implement the A algorithm for pathfinding:
    a. Create a priority queue to store nodes to be explored.
    b. Initialize the start node with a distance of 0 and add it to the priority queue.
    c. While the priority queue is not empty:
        - Dequeue the node with the lowest priority (based on the sum of the distance from the start node and the heuristic estimate to the end node).
        - If the dequeued node is the end node, stop the algorithm and reconstruct the path.
        - Otherwise, mark the node as visited and explore its neighbors.
        - Update the distance and priority of each neighbor node.
        - Add the neighbor nodes to the priority queue.
14. Implement the BFS algorithm for pathfinding:
    a. Create a queue to store nodes to be explored.
    b. Initialize the start node and add it to the queue.
    c. While the queue is not empty:
        - Dequeue the next node.
        - If the dequeued node is the end node, stop the algorithm and reconstruct the path.
        - Otherwise, mark the node as visited and explore its neighbors.
        - Add the unvisited neighbor nodes to the queue.
15. Create a function to reconstruct the path from the end node to the start node.

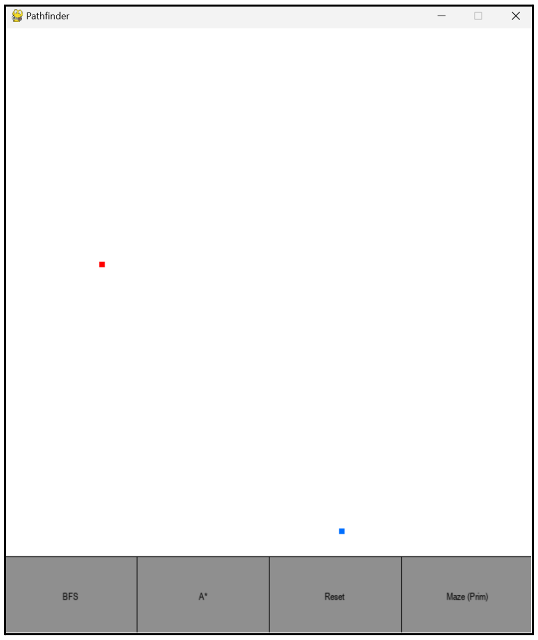
16. Update the GUI to reflect changes in the grid.
17. Display the grid and buttons on the screen.
18. Update the screen and handle the clock tick.
19. Repeat the main program loop until the user closes the window.

Flowchart:

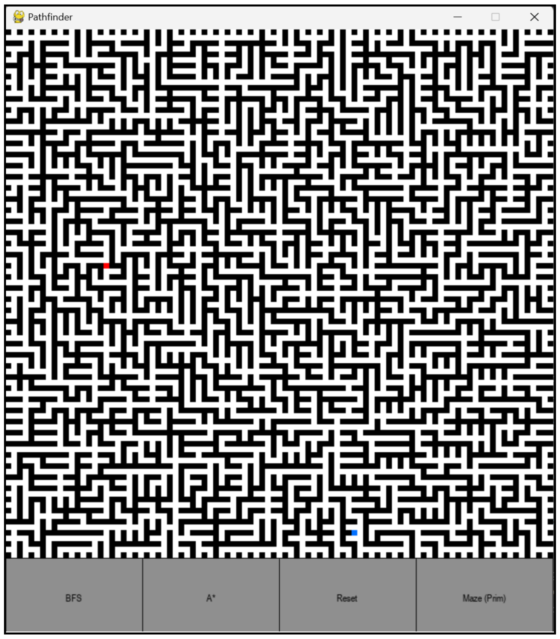


Output Screenshots:

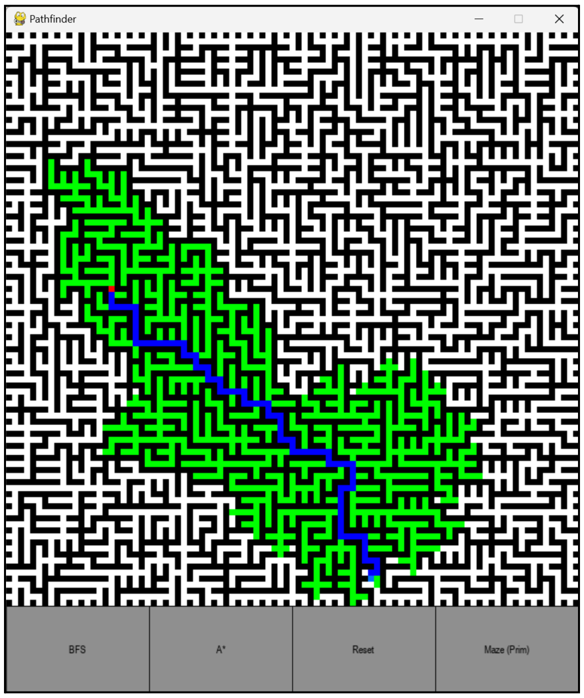
1. Initial UI with the starting and ending points, and the option to generate a maze.



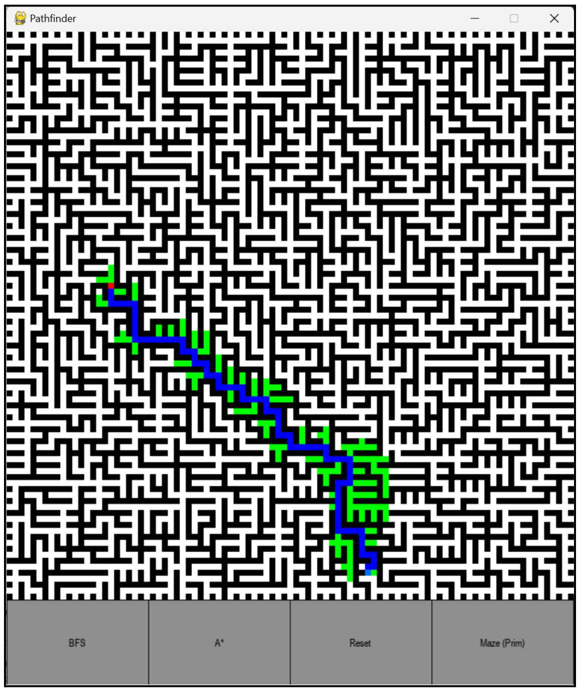
2. Generation of a random maze using Prim’s algorithm



3. Solving the maze with BFS



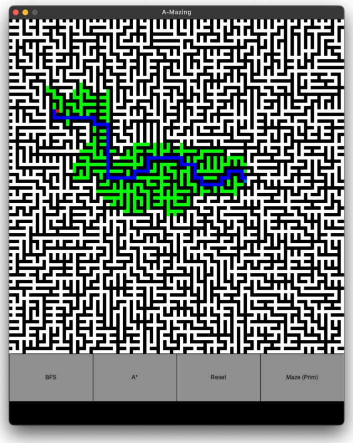
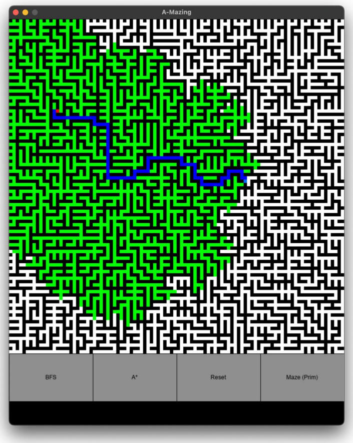
4. Solving the maze with A*



Performance Analysis

The performance of the A* and BFS algorithms can be analyzed by studying the number of nodes visited in the case of each algorithm. It can also be studied by looking at the general spread of the search area for both algorithms.

Grid Size: 100 x 100

Algorithm	A*	BFS
Visual Representation		
Number of visited nodes	1092	2509

Clearly, the number of nodes visited in the case is much larger. The A* algorithm tends to make more informed decisions because of the usage of heuristics to compute the next node to move to. BFS takes a more explorative strategy, looking at a much larger number of nodes in the vicinity.

Technological Improvement (Weak AI/Strong AI):

The technological improvement for the maze problem would fall under the category of Weak AI. The maze problem involves implementing pathfinding algorithms, user interaction, and visualization, which are well-defined and specific tasks. The AI in this context is focused on solving a particular problem within a constrained environment (the maze) rather than exhibiting general intelligence.

Reasoning:
1. **Narrow Focus**: The AI's task is narrowly focused on solving the maze and providing a visual representation of the pathfinding process. It does not exhibit general problem-solving capabilities beyond this specific domain.
2. **Rule-Based**: The AI's behavior is rule-based, following predefined algorithms for pathfinding and user interaction. It does not possess learning capabilities or adaptability beyond the programmed rules.
3. **Limited Context**: The AI's understanding is limited to the maze environment and the defined rules for pathfinding and visualization. It does not demonstrate understanding or reasoning beyond this context.

In summary, the technological improvement for the maze problem represents Weak AI, as it addresses specific tasks within a constrained domain without exhibiting general intelligence or learning capabilities.

Learning Outcomes:

- In conclusion, the implementation of pathfinding algorithms, A* and BFS, within the Pygame framework has showcased their efficiency in finding optimal routes.
- The visual representation and interactivity provided by Pygame offer a user-friendly experience, aiding in understanding the algorithms' decision-making process.
- Both A* and BFS algorithms demonstrate their strengths in different scenarios, where A* excels in finding the shortest path efficiently, while BFS guarantees completeness in exploring all possible routes.
- This project underscores the versatility and effectiveness of these algorithms in navigating complex mazes or maps, serving as a valuable tool for various applications in fields such as gaming, robotics, and navigation systems.