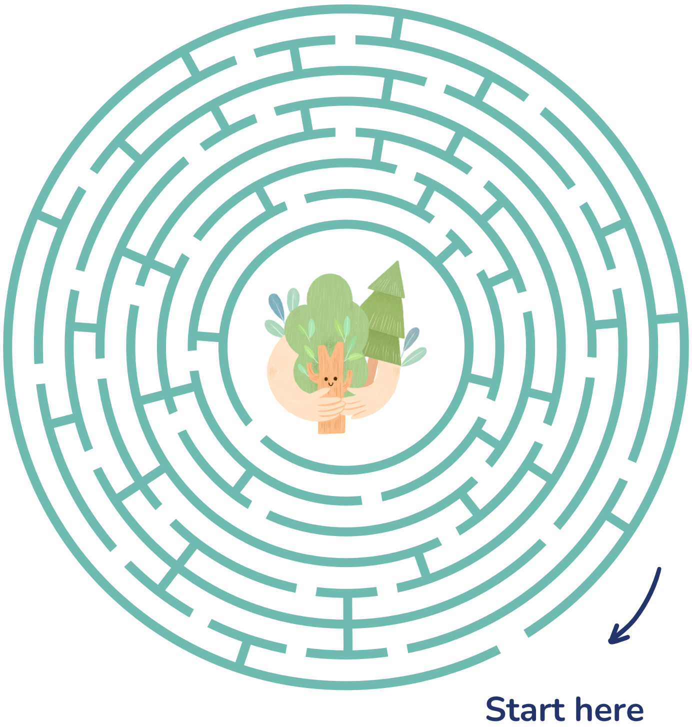
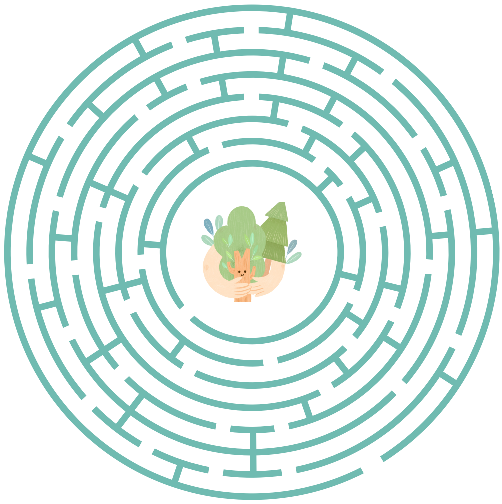
REPORT
ON

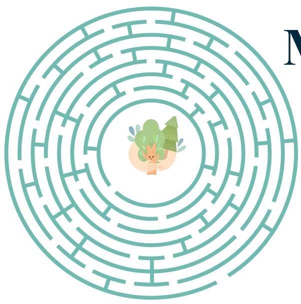
PATH FINDING VISUALIZER

Course Title: Artificial Intelligence Lab
Course Code: CSE-3636





Submitted to:
Ms. Bibi Sara Karimullah
Adjunct Lecturer, CSE



Submitted by:
1.Sakaratul Ara Tasmia
ID: C223298
2. Nowshin Islam Mim
ID: C223303
3. Tahsin Islam Nafisa
ID: C223311

TABLE OF CONTENTS

Project Name: Pathfinding Visualizer

1. Introduction

Pathfinding is a critical area in computer science with applications ranging from robotics and AI to game development and network routing. The **Pathfinding Visualizer** project is a desktop-based educational tool designed to visually demonstrate how various pathfinding and maze generation algorithms operate on a grid. This tool allows users to set start and end points, create obstacles, and watch how each algorithm finds a path, thus making algorithm behavior easy to understand and compare.

2. Objectives

- To provide an interactive and visual representation of pathfinding algorithms.
- To help students and learners understand algorithm efficiency and behavior.
- To enable real-time experimentation with different algorithms and maze structures.
- To offer an intuitive interface for comparing performance and path quality.

3. Technology Stack

Components	Technology
Language	Python 3.10+
GUI Library	Pygame
IDE	Any Python IDE or code editor (e.g., VSCode)
OS Compatibility	Cross-platform (Windows/Linux/macOS)

4. Frontend

The frontend is created using **Pygame**, which handles:

- Drawing the grid layout.
- Detecting user interactions (mouse clicks to place walls, select start/end).
- Animating the algorithm's execution and maze generation.

- Displaying real-time visual feedback such as visited nodes, path found, and unvisited nodes.

The UI includes:

- Grid display.
- Algorithm selection and maze generation buttons.
- Keyboard and mouse event handling for interaction.

5. Backend

Though the project is not server-based, the backend here refers to the core logic implementing:

- **Pathfinding algorithms**:
  - Breadth-First Search (BFS)
  - Depth-First Search (DFS)
  - Dijkstra's Algorithm
  - Greedy Best-First Search
  - A* Search
- **Maze generation algorithms**:
  - Recursive Division
  - Prim's Algorithm

These algorithms are written in Python and operate directly on grid data structures using queue/stack-based approaches and heuristics.

6. Methodology

Algorithm Execution Steps:

1. Initialize a grid with empty cells.
2. User selects start and end nodes.
3. Walls or obstacles can be drawn manually or auto-generated.
4. Selected algorithm runs step-by-step:
   a. Explores neighboring nodes.
   b. Updates node statuses (visited, in path, etc.).
   c. Stops when the goal is reached or path is not found.

The methodology used emphasizes **real-time animation** and **heuristic-driven decisions** (in the case of A*, Greedy Best-First).

Performance Formula (Used Internally)

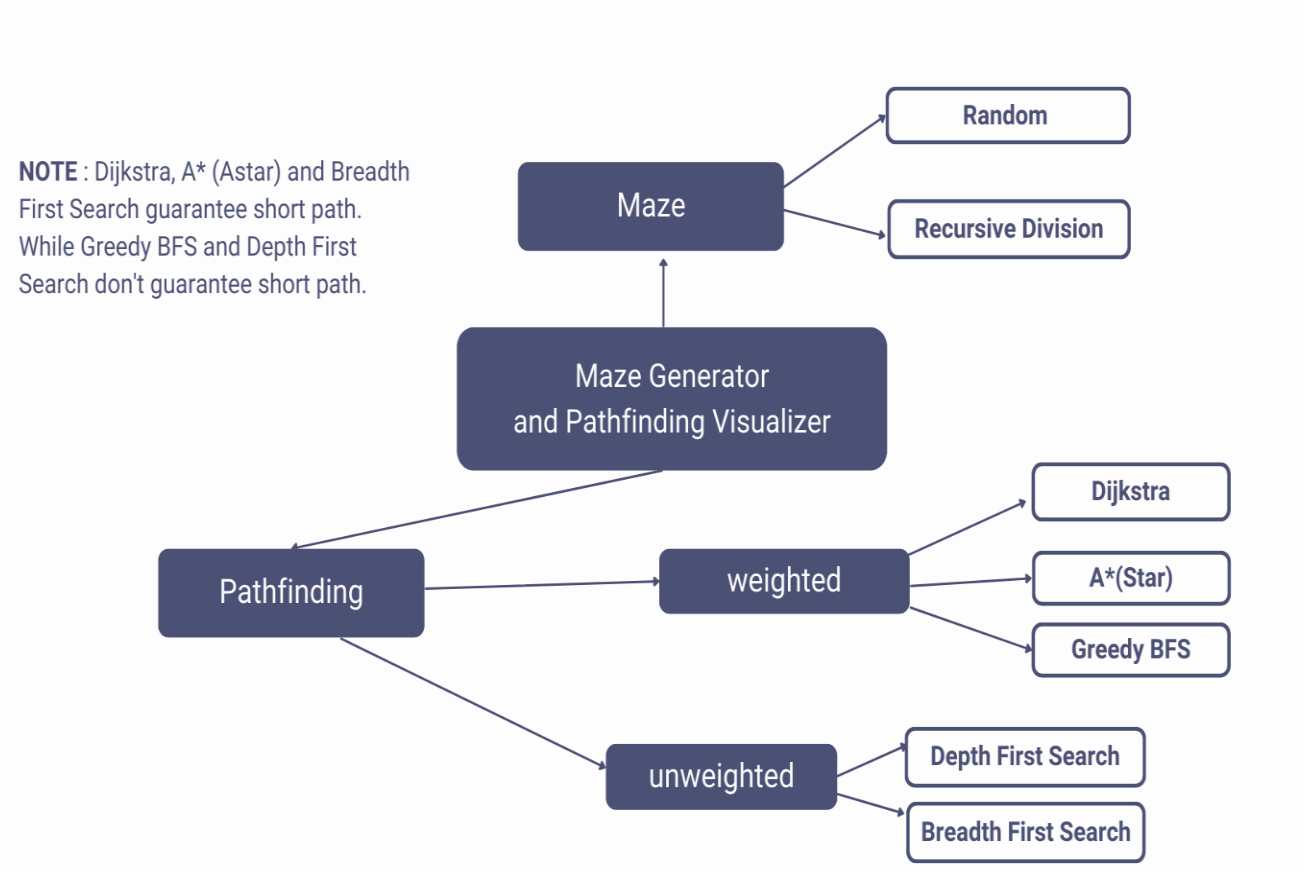
While no explicit performance formula is implemented, heuristics like:

$f(n) = g(n) + h(n)$

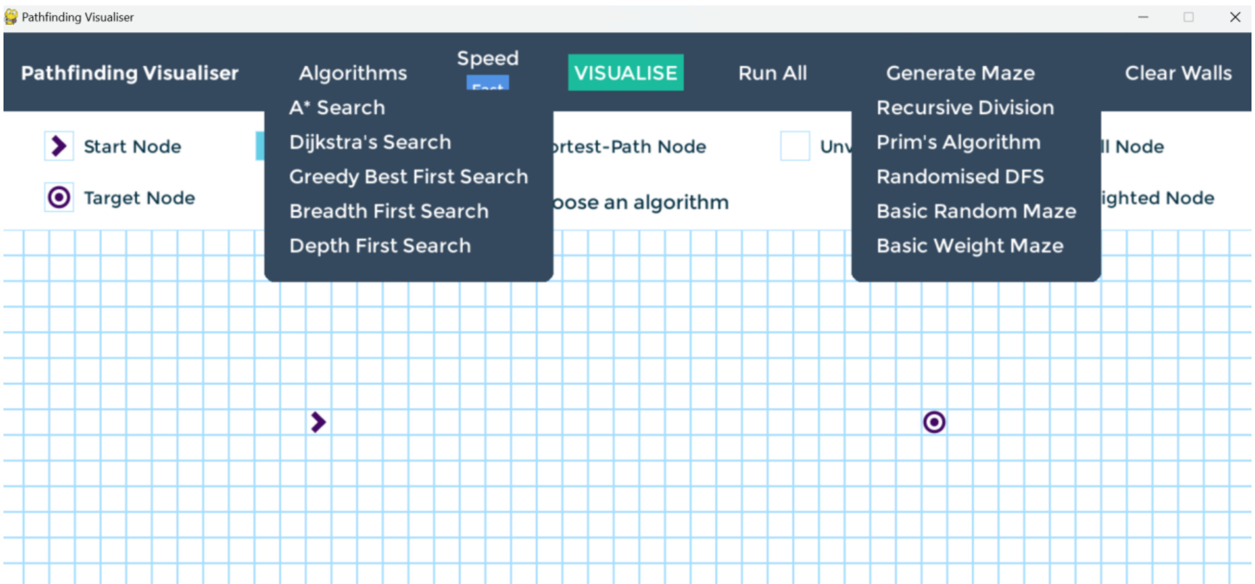
are used in A* where:

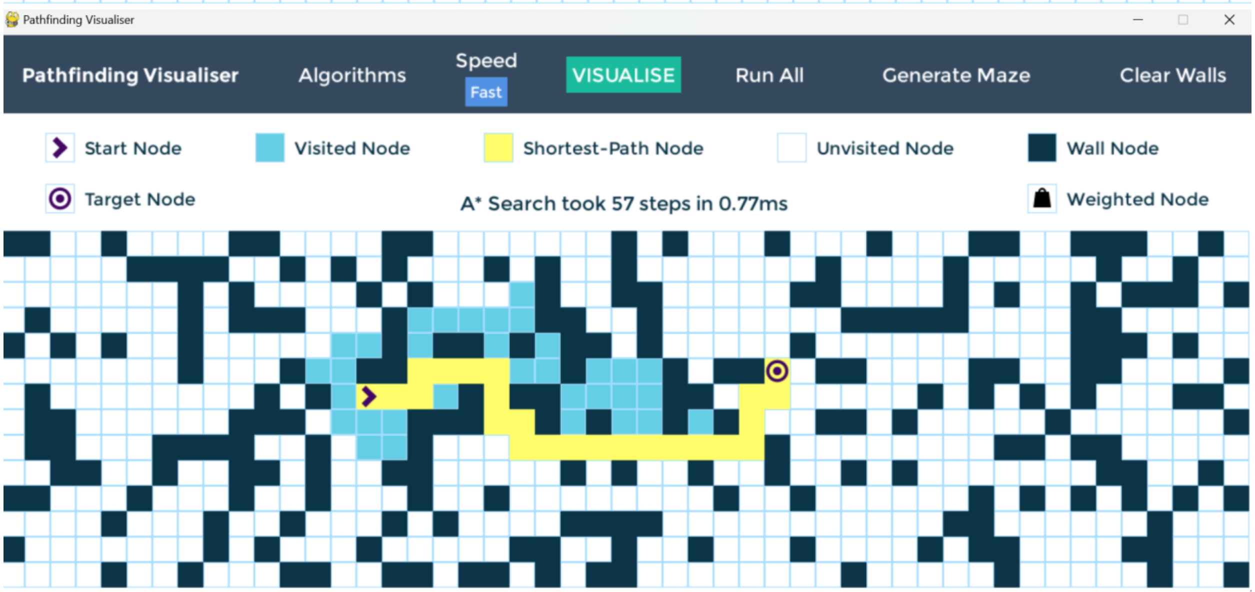
- $g(n)$ = cost to reach current node
- $h(n)$ = estimated cost to goal (usually Manhattan Distance)

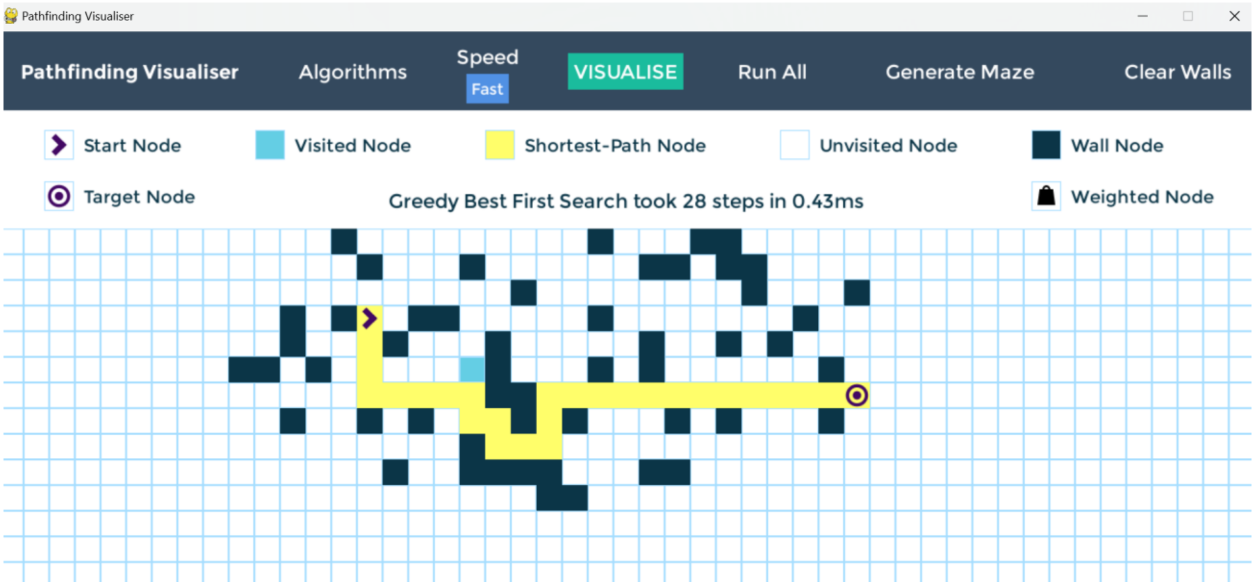
7. Flowchart

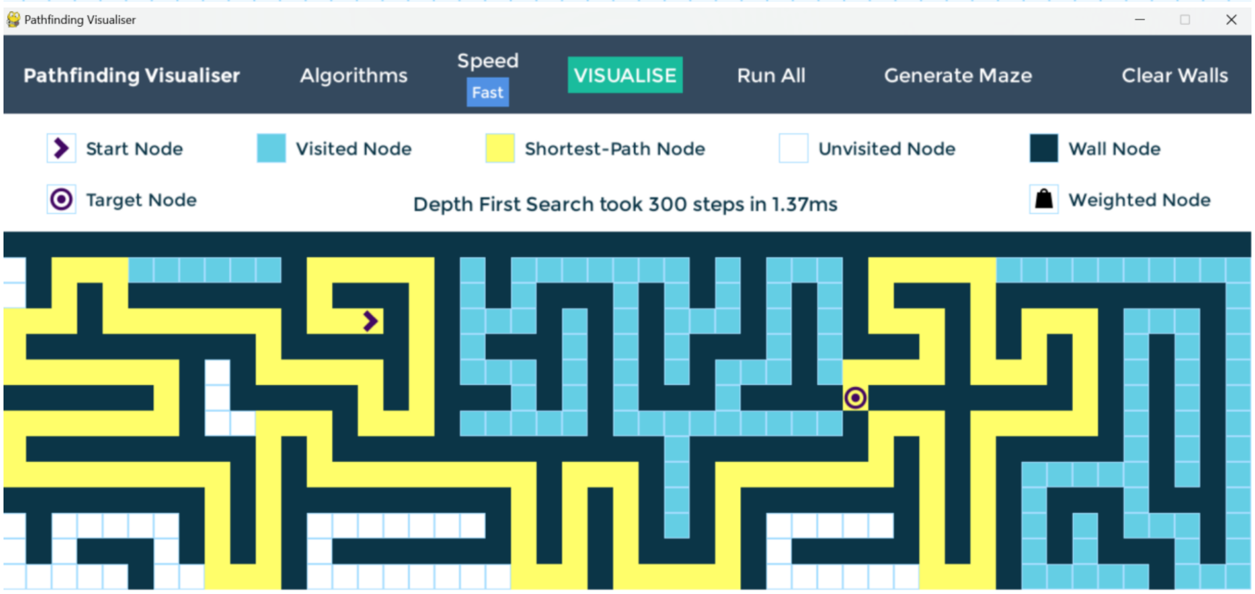

NOTE : Dijkstra, A* (Astar) and Breadth First Search guarantee short path. While Greedy BFS and Depth First Search don't guarantee short path.
Maze
Random
Recursive Division
Maze Generator and Pathfinding Visualizer
Pathfinding
weighted
Dijkstra
A*(Star)
Greedy BFS
unweighted
Depth First Search
Breadth First Search

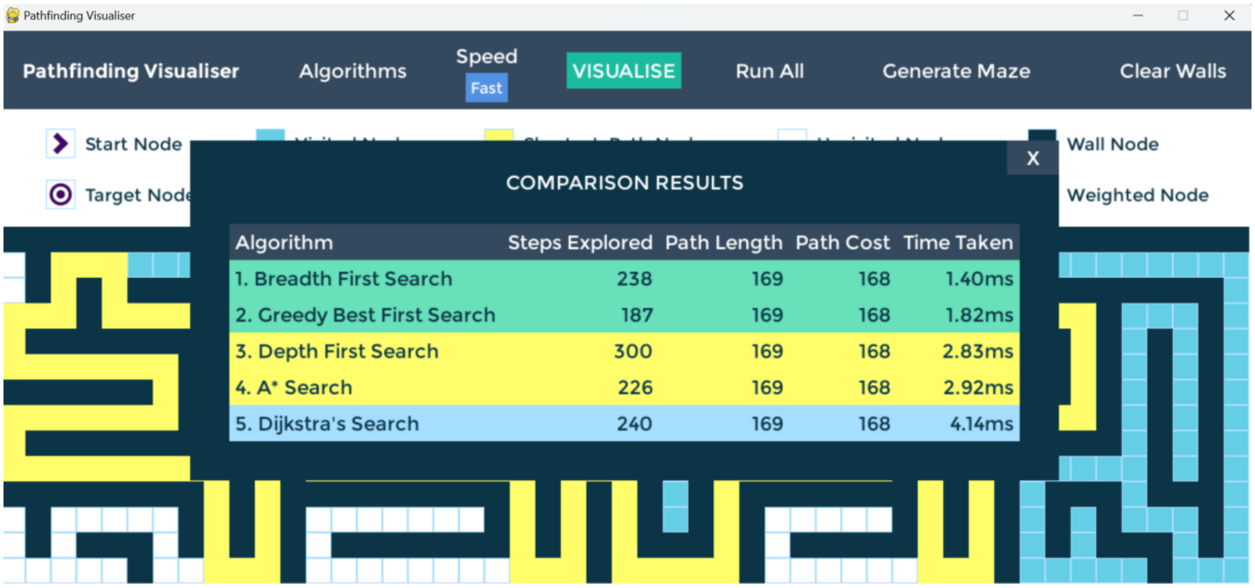
7. Screenshots






Pathfinding Visualiser
Pathfinding Visualiser
Algorithms
Speed
Fast
VISUALISE
Run All
Generate Maze
Clear Walls
Start Node
Visited Node
Shortest-Path Node
Unvisited Node
Wall Node
Target Node
Greedy Best First Search took 28 steps in 0.43ms
Weighted Node


Pathfinding Visualiser
Pathfinding Visualiser
Algorithms
Speed
Fast
VISUALISE
Run All
Generate Maze
Clear Walls
Start Node
Visited Node
Shortest-Path Node
Unvisited Node
Wall Node
Target Node
Depth First Search took 300 steps in 1.37ms
Weighted Node



8. Implementation Highlights

- Modular and object-oriented code structure.
- run.pyw serves as the entry point for the application.
- Nodes are represented as square grid cells, with properties like visited, start, end, wall, etc.
- Efficient use of data structures: queues for BFS, priority queues for A*, etc.
- Color-coded animation:
  - Blue: explored
  - Yellow: in queue
  - Green: path
  - Black: wall
- Interactive UI with real-time feedback during mouse clicks and keyboard events.

9. Result

- All implemented algorithms were tested and correctly visualized.
- The tool provides an excellent visual aid to understand how each algorithm behaves in various scenarios.

- Comparison between algorithms becomes intuitive based on their speed, path quality, and area explored.
- Maze generation helps test the robustness of the algorithms.

10. Limitations

1. Simplified Environment: Limited to 2D grid simulations, it may not capture real-world complexities.
2. Algorithmic Focus: Primarily explores pathfinding algorithms, lacking broader AI context.
3. Metrics Scope: Provides basic metrics, but not exhaustive performance analysis.
4. UI Complexity: Adding features can complicate usability for non-technical users.
5. Educational Depth: Offers visualization but may not substitute for comprehensive learning resources.
6. Hardware Constraints: Performance may vary on different devices, affecting scalability.

11. Conclusion

The **Pathfinding Visualizer** successfully demonstrates classic pathfinding algorithms through interactive and educational animations. It is a valuable resource for computer science students, educators, and enthusiasts who wish to understand search techniques in a practical manner. The integration of real-time animation, multiple algorithms, and user controls makes it both functional and engaging.

12. Future Enhancements

- **Weighted grids**: Allow users to add weighted cells with varied traversal costs.
- **Diagonal movement**: Enable more realistic pathfinding across grids.
- **Performance metrics**: Show number of nodes visited, time taken, and path length.
- **Save/load grid state**: Let users save custom grid layouts and re-test them.
- **Web version**: Convert the app into a web-based tool using JavaScript and canvas or react.
- **More algorithms**: Add Johnson's, Bellman-Ford, Jump Point Search, etc.
- **Dark mode & UI themes** for better visual appeal.