Name: Sahan Rudrigo
Date: 10th April 2025

Intelligent Pathfinding System
BFS and A* Algorithm Comparison
- A Python Implementation with Visualization -

1. Introduction

This project implements and compares two fundamental pathfinding algorithms—Breadth-First Search (BFS) and A* within a 2D grid environment. The goal is to navigate from a starting point (top-left corner) to a goal (bottom-right corner) while avoiding randomly placed obstacles. Developed in Python, the system leverages libraries such as 'numpy' for grid management, 'heapq' for efficient priority queuing, and 'tkinter' for an interactive graphical user interface (GUI). The project includes a 'Jupyter Notebook' for initial development and a standalone GUI application for real-time visualization. Performance metrics, including execution time and nodes explored, are analyzed to evaluate the efficiency of each algorithm. This document details the methodology, presents results, and provides visual evidence through screenshots, offering insights into practical pathfinding applications such as robotics and game development.

2. Methodology

2.1 Libraries and Dependencies

The implementation relies on several Python libraries to handle various aspects of the pathfinding system:

- ✓ heapq: Provides a priority queue for the A* algorithm, ensuring efficient selection of the next node based on cost.
- ✓ numpy: Facilitates numerical operations and array-based grid representation for quick manipulation and visualization.

- ✓ plotly.express: Used in the notebook for potential 3D plotting (though not fully utilized here), offering interactive visualization capabilities.
- ✓ time: Measures algorithm execution time for performance comparison.
- ✓ collections.deque: Implements a fast FIFO queue for BFS, optimizing neighbor exploration.
- ✓ random: Generates random obstacles in the grid with a specified probability.
- ✓ tkinter: Powers the GUI, enabling interactive grid editing and real-time algorithm visualization.
- ✓ tabulate: Formats the performance comparison table in a readable manner.

2.2 Cell Class Definition

The Cell class is the core building block of the grid, encapsulating the properties of each position. This describe the data structure that underpins the grid, making it clear how cells are represented and manipulated

- ✓ x and y: Integer coordinates (0-indexed) representing the cell's column and row.
- ✓ type: A string indicating the cell's role—'start', 'goal', 'obstacle', or 'empty' (default).
- ✓ parent: A reference to the previous cell in the path, used for path reconstruction.
- ✓ g_cost: The cost from the start node, initialized to infinity and updated during A* execution.

2.3 Grid Creation

The "create_grid" function constructs a 10x10 grid of Cell objects. The start position is set at (0,0) with type='start', and the goal at (9,9) with type='goal'. Obstacles are randomly placed with a default probability of 0.2 (20%). Here 's' denotes start, 'g' denotes goal, 'o' denotes obstacles, and 'e' denotes empty cells.

```
[217]: #Visualization Grid
       print(np.array([[cell.type[0] for cell in row] for row in grid]))
```

```
[['s' 'e' 'e' 'e' 'e' 'e' 'e' 'e' 'e' 'o']
 ['o' 'e' 'e' 'o' 'o' 'o' 'e' 'e' 'e' 'e']
 ['e' 'e' 'o' 'e' 'e' 'e' 'e' 'e' 'e' 'e']
 ['o' 'o' 'e' 'e' 'e' 'e' 'o' 'e' 'e' 'o']
 ['e' 'e' 'e' 'e' 'e' 'e' 'e' 'e' 'e' 'o']
 ['e' 'e' 'e' 'o' 'e' 'e' 'e' 'o' 'e' 'e']
 ['e' 'e' 'e' 'e' 'e' 'e' 'e' 'o' 'e' 'o']
 ['e' 'e' 'e' 'e' 'e' 'e' 'o' 'e' 'e' 'e']
 ['e' 'e' 'o' 'e' 'e' 'e' 'e' 'e' 'e' 'e']
 ['e' 'e' 'o' 'e' 'e' 'e' 'e' 'e' 'e' 'g']]
```


Figure 1-Visualization Grid

2.4 Pathfinding Algorithms

2.4.1 Breadth-First Search (BFS)

BFS finds the shortest path in an unweighted grid by exploring one level at a time. It starts from the starting cell (0,0) and uses a queue to keep track of which cell to visit next. At each step, it checks the neighboring cells (up, down, left, right). If a neighbor hasn't been visited and is not an obstacle, it is added to the queue. This continues until the goal (9,9) is found or there are no more cells to explore. The path is then built by going back from the goal to the start using parent pointers. BFS always gives the shortest path in terms of number of steps, but it may explore many cells, especially in big grids.

2.4.2 A* Algorithm

The A* algorithm finds the shortest path more efficiently than BFS by using a priority queue and a smart guess called a heuristic. It checks each cell based on a score:
f_cost = g_cost + h_cost

- g_cost is the distance from the start
- h_cost is the estimated distance to the goal (using Manhattan distance)

Manhattan distance is good here because we only move up, down, left, or right. A* always chooses the next cell with the lowest total cost (f_cost), so it avoids exploring unnecessary paths. Like BFS, it uses parent pointers to build the final path from goal to start. It’s faster than BFS in many cases and still gives the shortest path**.**

2.5 Visualization and GUI

Visualization is implemented in two forms. In the Jupyter Notebook, a simple text-based grid is printed using numpy arrays, showing the initial layout. The GUI, built with tkinter, offers an interactive 10x10 grid

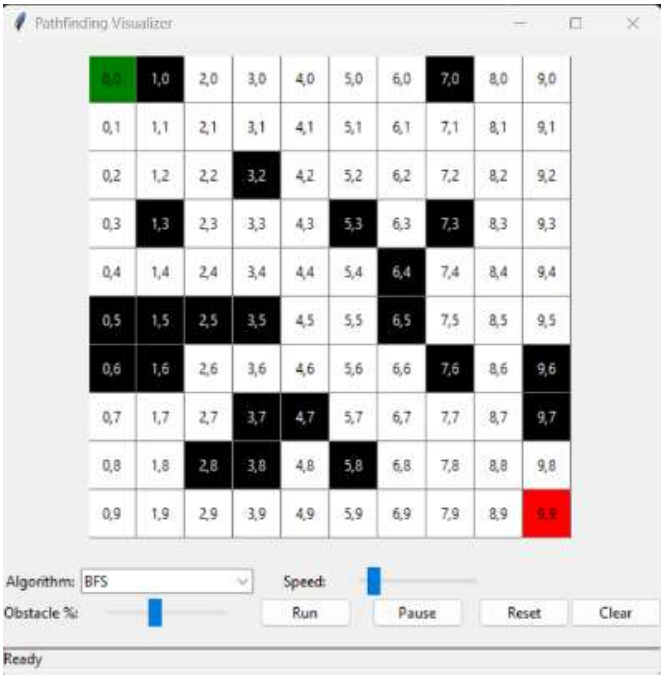

Figure 2-GUI

3. Results and Performance Comparison

The performance of BFS and A* was evaluated over 1000 runs on the same 10x10 grid. Results are summarized in the table below

Algorithm	Path Length	Avg.Time (ms)	Nodes Explored	Efficiency
BFS	18	0.13	73	26.0%
A*	18	0.06	33	57.6%

Figure 3-Output results

Both algorithms found an identical 18-step path from (0,0) to (9,9), as expected in an unweighted grid with 4-directional movement. However, A* executed faster (0.06 ms vs. 0.13 ms) and explored fewer nodes (33 vs. 73), achieving higher efficiency (57.6% vs. 26.0%). Efficiency is calculated as the ratio of path length to nodes explored. A*'s heuristic-guided approach reduces unnecessary exploration, making it more suitable for larger or complex grids, while BFS's exhaustive search ensures optimality but at a higher computational cost.

4.Demo Video

Video Link :-
https://drive.google.com/file/d/1WWgMkW4OTpsdLhFhqnAh3pMBn3jMJp2e/view?usp=sharing

5.References

[1]Python Software Foundation, "3.7.3 Documentation," *Python.org*, 2019. https://docs.python.org/3/

[2]Python Software Foundation, "tkinter — Python interface to Tcl/Tk — Python 3.7.2 documentation," *python.org*, 2019. https://docs.python.org/3/library/tkinter.html

[3]*Youtu.be*, 2025. https://youtu.be/-L-WgKMFuhE?si=KR1lCkQHuibfP46M (accessed Apr. 10, 2025).