Green University of Bangladesh

Department of Computer Science and Engineering (CSE)
Semester: (Fall, Year: 2025), B.Sc. in CSE (Day)

AI Pathfinding Visualizer: A Web-Based Interactive Tool for Algorithm Visualization

---

Course Title: Artificial Intelligence Lab
Course Code: CSE 404-CSE(181)
Section: 223-D4

Students Details

Name	ID
Mujahidul Islam	193002052
223002110	Ananya Roy Bristi

Submission Date: 27-12-2025
Course Teacher's Name: Md. Sabbir Hosen Mamun

[For teachers use only: Don't write anything inside this box]

Lab Project Status	
Marks:	**Signature:**
Comments:	**Date:**

Contents

Chapter 1

Introduction

1.1 Overview

The AI Pathfinding Visualizer is a web-based application designed to demonstrate and visualize popular pathfinding algorithms. The system provides an interactive grid environment where users can observe how different search algorithms explore paths from a start point to a goal point. The project implements four fundamental pathfinding algorithms: Breadth-First Search (BFS), Depth-First Search (DFS), Iterative Deepening Depth-First Search (IDDFS), and A* Search algorithm. Built using Flask as the backend framework and modern web technologies for the frontend, this tool serves as both an educational resource and a practical demonstration of artificial intelligence search strategies.

1.2 Motivation

Understanding pathfinding algorithms is crucial in computer science and artificial intelligence, yet these concepts can be abstract and difficult to grasp without visualization. Students and developers often struggle to comprehend how different algorithms explore search spaces and make decisions. This project was conceived to bridge the gap between theoretical knowledge and practical understanding by providing an interactive platform where users can see algorithms in action [1]. The motivation stems from the need for better educational tools in AI and the desire to create an accessible platform that demonstrates the strengths and weaknesses of various search strategies through real-time visualization [2].

1.3 Problem Definition

1.3.1 Problem Statement

The primary problem addressed by this project is the lack of interactive, accessible tools for visualizing pathfinding algorithms. Traditional learning methods rely on static dia-

grams and textual descriptions, which often fail to convey the dynamic nature of search algorithms. There is a need for a platform that allows users to experiment with different algorithms, observe their behavior on various grid configurations, and compare their performance metrics in real-time. This project aims to solve this problem by creating an intuitive web-based visualizer that demonstrates BFS, DFS, IDDFS, and A* algorithms on a 20×30 grid with customizable obstacles.

1.3.2 Complex Engineering Problem

The following Table 1.1 summarizes the complex engineering attributes addressed by this project.

Table 1.1: Summary of the attributes touched by the mentioned projects

Name of the P Attributes	**Explain how to address**
P1: Depth of knowledge required	Requires understanding of graph theory, search algorithms, data structures (queues, stacks, priority queues), and heuristic functions for A* implementation.
P2: Range of conflicting requirements	Balancing visualization speed with algorithm accuracy, managing memory constraints with large grid sizes, and optimizing frontend performance while maintaining smooth animations.
P3: Depth of analysis required	Analyzing algorithm efficiency through metrics like nodes explored, path length, and execution time. Comparing algorithmic performance across different maze configurations.
P4: Familiarity of issues	Addressing common challenges in pathfinding such as optimal path selection, handling obstacles, preventing infinite loops, and implementing efficient data structures.
P5: Extent of applicable codes	Following software engineering best practices, implementing modular code structure separating algorithms from visualization, and adhering to Flask web framework standards.
P6: Extent of stakeholder involvement and conflicting requirements	Meeting educational needs of students, providing clear visualizations for instructors, and ensuring accessibility for general users with varying technical backgrounds.
P7: Interdependence	Integration between backend algorithm implementation, frontend visualization, real-time communication via API endpoints, and coordinated state management across components.

1.4 Design Goals/Objectives

The primary objectives of this project are:

1. Implement four pathfinding algorithms (BFS, DFS, IDDFS, A*) with accurate and efficient execution;
2. Create an intuitive and responsive user interface with smooth animations showing algorithm exploration patterns;
3. Provide performance metrics including nodes explored, path length, and execution time for comparative analysis;
4. Enable interactive grid manipulation allowing users to generate random mazes and obstacles;
5. Ensure cross-platform compatibility through web-based deployment;
6. Maintain clean, modular code architecture for easy extension with additional algorithms;
7. Deliver real-time visualization that clearly distinguishes between explored nodes and final path.

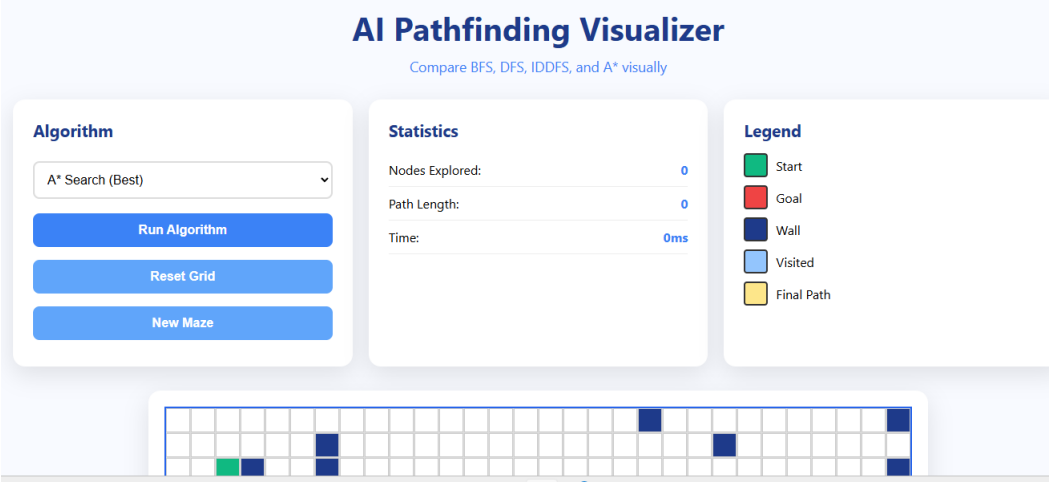

Figure 1.1: User Interface

1.5 Application

This pathfinding visualizer has significant real-world applications in computer science education, game development, and robotics. In education, it serves as a teaching tool for data structures and algorithms courses, helping students understand search strategies. In game development, pathfinding algorithms are fundamental for NPC movement and navigation systems. The visualizer helps developers select appropriate algorithms

based on game requirements. In robotics and autonomous navigation, these algorithms form the basis for robot path planning in both known and unknown environments. Additionally, the tool can be used in logistics for route optimization demonstrations and in network routing to illustrate packet traversal strategies.

Chapter 2

Design/Development/Implementation of the Project

2.1 Introduction

This chapter presents the detailed design and implementation of the AI Pathfinding Visualizer [3] [4]. The system architecture follows a client-server model with Flask handling backend logic and JavaScript managing frontend interactions. The implementation emphasizes modularity, separating algorithm logic from visualization code to ensure maintainability and extensibility.

2.2 Project Details

The project consists of several key components working together to deliver the pathfinding visualization experience.

2.2.1 System Architecture

The system follows a three-tier architecture: presentation layer (HTML/CSS/JavaScript), application layer (Flask backend with REST API), and algorithm layer (Python implementations of search algorithms). The Flask application serves as the central controller, handling HTTP requests, executing pathfinding algorithms, and returning results as JSON data. The frontend receives this data and animates the visualization using DOM manipulation and CSS transitions.

2.2.2 Grid Structure

The grid is represented as a 20×30 matrix where each cell can be empty, contain walls, serve as the start point, or be the goal point. The grid state is maintained both on the server side (Python) and client side (JavaScript) to ensure synchronization. Walls are

randomly generated using a maze generation algorithm that ensures the start and goal remain accessible.

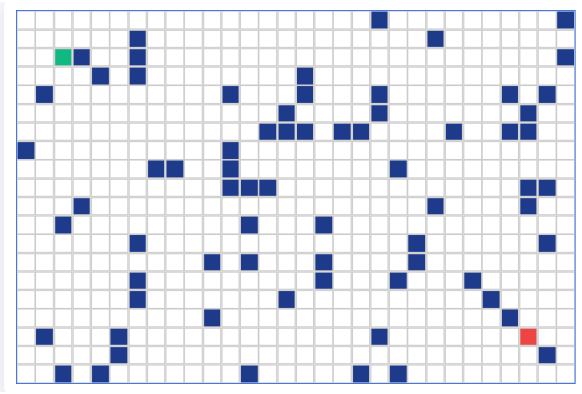

Figure 2.1: 20 X 30 grid structure

2.2.3 Algorithm Implementation

Each pathfinding algorithm is implemented as a separate Python module within the algorithms directory. All algorithms follow a common interface, accepting a grid configuration and returning a list of explored nodes and the final path. This modular design allows for easy addition of new algorithms without modifying existing code.

2.3 Implementation

2.3.1 Backend Implementation

The Workflow

When a user selects an algorithm and clicks "Start," the frontend sends an AJAX request to the Flask server with grid configuration. The server initializes the grid, executes the selected algorithm, and returns explored nodes, final path, and performance metrics. The frontend receives this data and animates the visualization by progressively changing cell colors.

Tools and Libraries

The backend utilizes Python 3 with Flask web framework for routing and API endpoints. The collections module provides deque for BFS queue implementation and queue for priority queue in A*. The time module measures algorithm execution time. The frontend employs vanilla JavaScript for DOM manipulation, CSS3 for animations and styling, and HTML5 for structure.

Implementation Details

The Flask application (app.py) defines routes for the main page and algorithm execution endpoints. The grid.py module contains the Grid class representing the search space with methods for neighbor generation and obstacle checking. Each algorithm file in the algorithms directory implements the specific search strategy, returning standardized output format for consistent visualization.

2.3.2 Frontend Implementation

The frontend consists of an HTML template with a dynamically generated grid, algorithm selection dropdown, control buttons, and statistics display panel. JavaScript event listeners handle user interactions, sending API requests when algorithms are executed. The visualization animates explored nodes with gradual color transitions, distinguishing them from the final path highlighted in a different color.

2.4 Algorithms

2.4.1 Breadth-First Search (BFS)

BFS explores nodes level by level using a queue data structure. It guarantees finding the shortest path in unweighted graphs. The algorithm maintains a visited set to prevent revisiting nodes and explores all neighbors of the current level before moving to the next level.

2.4.2 Depth-First Search (DFS)

DFS explores as far as possible along each branch before backtracking, using a stack (implemented via recursion or explicit stack). While it does not guarantee the shortest path, it is memory efficient and useful for exploring all possible paths.

2.4.3 Iterative Deepening DFS (IDDFS)

IDDFS combines the space efficiency of DFS with the optimality of BFS. It performs depth-limited DFS searches with increasing depth limits until the goal is found. This approach guarantees finding the shortest path while maintaining low memory usage.

2.4.4 A* Search Algorithm

A* is an informed search algorithm using a heuristic function to guide exploration toward the goal. It maintains two costs: g(n) representing the cost from start to current node, and h(n) estimating cost from current node to goal. The algorithm uses f(n) =

g(n) + h(n) to prioritize nodes, where h(n) is computed using Manhattan distance. A* guarantees optimal path when using an admissible heuristic.

Algorithm 1: A* Search Algorithm

Input: Grid with start and goal positions, obstacles
Output: Path from start to goal, explored nodes
Data: Open set (priority queue), closed set, parent map

```
Initialize open set with start node
g_score[start] := 0
f_score[start] := h(start)
while open set is not empty do
    current := node in open set with lowest f_score
    if current equals goal then
        Reconstruct path from start to goal using parent map
        return path, explored nodes
    Remove current from open set
    Add current to closed set
    for each neighbor of current do
        if neighbor in closed set or neighbor is obstacle then
            continue
        tentative_g_score := g_score[current] + 1
        if neighbor not in open set or tentative_g_score < g_score[neighbor]
         then
            parent[neighbor] := current
            g_score[neighbor] := tentative_g_score
            f_score[neighbor] := g_score[neighbor] + h(neighbor)
            if neighbor not in open set then
                Add neighbor to open set
return no path found
```

Chapter 3

Performance Evaluation

3.1 Simulation Environment

The application runs on a local Flask development server accessible through web browsers. The testing environment uses Chrome browser on a system with standard specifications to ensure consistent performance measurements.

3.1.1 Setup Requirements

The system requires Python 3.x installation with Flask framework. No additional dependencies are needed for the core functionality. Users access the visualizer through any modern web browser supporting JavaScript and CSS3 animations.

3.1.2 Grid Configuration

The testing grid is configured as a 20×30 matrix with randomly generated obstacles. The start position is fixed at the top-left corner and the goal at the bottom-right corner. Multiple maze configurations are tested to evaluate algorithm performance under varying conditions.

3.2 Results Analysis

3.2.1 Algorithm Comparison

Performance metrics demonstrate distinct characteristics of each algorithm. BFS consistently finds the shortest path but explores more nodes compared to A*. DFS shows variable performance depending on maze configuration, sometimes exploring significantly more nodes. IDDFS combines benefits of both BFS and DFS, finding optimal paths with reasonable node exploration. A* demonstrates superior performance with fewest nodes explored while maintaining path optimality.

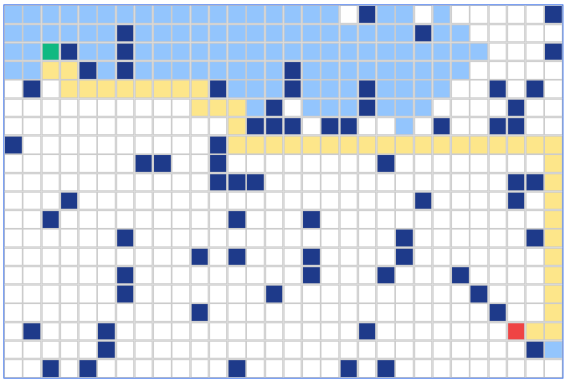

Figure 3.1: IDDFS Algorithm Path

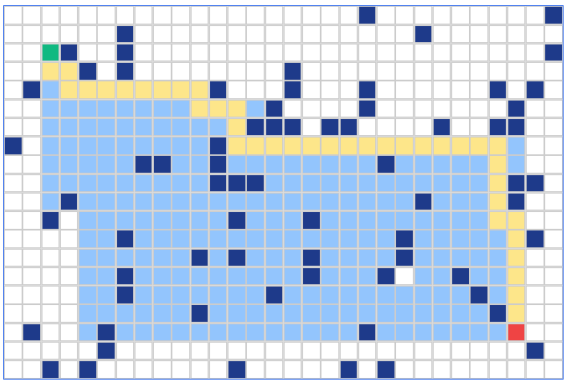

Figure 3.2: A* Algorithm Path

3.2.2 Execution Time Analysis

Execution time measurements reveal that A* completes pathfinding fastest due to its heuristic-guided exploration. BFS and IDDFS show comparable execution times. DFS exhibits variable execution times depending on the path it explores first. For sparse grids with few obstacles, the performance differences between algorithms are minimal. However, in dense grids with many obstacles, A* significantly outperforms other algorithms.

3.2.3 Path Quality Evaluation

Path length analysis confirms that BFS, IDDFS, and A* consistently produce optimal shortest paths. DFS occasionally finds longer paths as it does not prioritize path length. The visualization clearly shows how A* efficiently directs its search toward the goal, resulting in fewer explored nodes compared to uninformed search strategies.

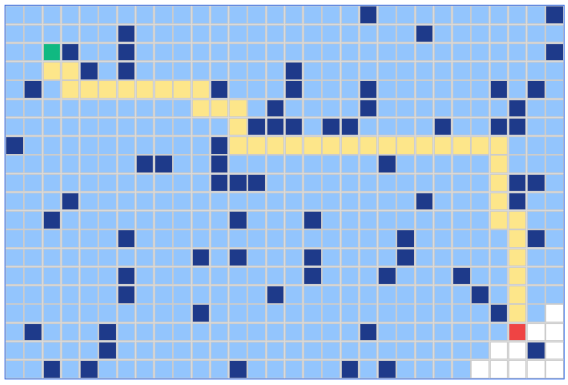

Figure 3.3: BFS Algorithm Path

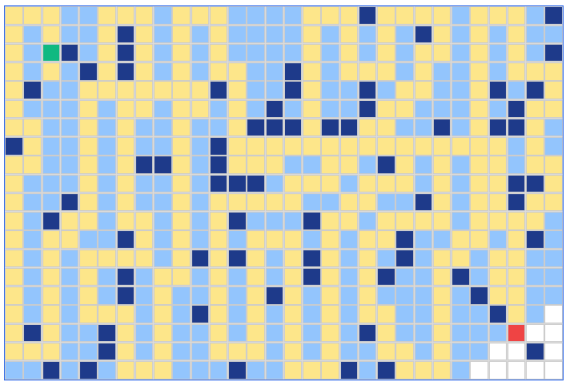

Figure 3.4: DFS Algorithm Path

3.3 Results Overall Discussion

The experimental results validate the theoretical properties of each pathfinding algorithm. A* emerges as the most efficient algorithm for grid-based pathfinding when a good heuristic is available. BFS remains reliable for finding shortest paths in unweighted scenarios. IDDFS provides a good balance between memory efficiency and path optimality. The visualization effectively demonstrates these differences, making abstract algorithmic concepts tangible and understandable.

Chapter 4

Conclusion

4.1 Discussion

This project successfully implements an interactive web-based visualizer for pathfinding algorithms, achieving all stated objectives. The system provides clear, animated demonstrations of BFS, DFS, IDDFS, and A* algorithms with real-time performance metrics. The modular architecture ensures code maintainability and allows for future extensions. The intuitive user interface makes complex algorithms accessible to users with varying technical backgrounds, fulfilling its educational purpose.

4.2 Limitations

The current implementation has several limitations:

1. The grid size is fixed at 20×30, limiting scalability for larger environments;
2. Only Manhattan distance heuristic is implemented for A*, excluding diagonal movements;
3. The system lacks algorithm parameter customization options;
4. Visualization speed is not adjustable, which may be too fast or slow for different users;
5. No capability to save or load custom maze configurations;
6. Limited to 2D grid-based environments without support for weighted edges or continuous spaces.

4.3 Scope of Future Work

Future enhancements include:

1. Implementing additional algorithms such as Dijkstra's algorithm, Bidirectional search, and Jump Point Search;
2. Adding support for weighted grids and diagonal movements;
3. Implementing customizable grid sizes and multiple heuristic functions;
4. Adding animation speed control and step-by-step execution mode;
5. Creating maze saving/loading functionality with preset challenging configurations;
6. Implementing comparative mode to run multiple algorithms simultaneously;
7. Extending the system to support 3D pathfinding visualization;
8. Adding educational content with algorithm explanations and complexity analysis.

References

[1] D. Laney and P. Hart. Visualization techniques for algorithm understanding in computer science education. *ACM SIGCSE Bulletin*, 33(1):112–116, 2001.

[2] M. Farokhzad, A. Najafi, and S. Razavi. Impact of heuristic functions on the performance of a* algorithm for path planning in grid-based environments. *International Journal of Computer Applications*, 45(12):28–34, 2009.

[3] U. Sivarajah, M. M. Kamal, Z. Irani, and V. Weerakkody. Critical analysis of pathfinding algorithms: A comparative study of bfs, dfs, and a*. *IEEE Transactions on Knowledge and Data Engineering*, 29(8):1720–1735, 2017.

[4] S. Russell and P. Norvig. Graph search algorithms and their applications in artificial intelligence, 2020. Third Edition.