INTERNATIONAL UNIVERSITY

School of Computer Science and Engineering

Algorithms & Data Structures

IT013IU

PROJECT

Path Finding Visualizer

Project's github:link

Submitted by

Lê Hưng - ITCSIU22271

Trần Đăng Nhất - ITITIU22115

Date submitted: 03/06/2025

Course Instructors: V. C. Thanh & T. T. Tin

Table of Content

Introduction

1. About Path Finding Visualizer Project

1.1. Main function

The Pathfinding Visualizer is a web-based application designed to interactively demonstrate the execution of pathfinding and maze-generation algorithms. Its core functionalities include:

- Pathfinding Algorithms: Real-time visualization of Dijkstra's algorithm, A* search, Breadth-First Search (BFS), and Depth-First Search (DFS).
- Maze Generation: Dynamic creation of mazes using Recursive Division and Binary Tree algorithms.
- User Interaction: Custom wall placement, adjustable animation speeds, and reset capabilities.
- Visual Feedback: Color-coded nodes to distinguish traversed paths, walls, start/end points, and optimal routes.

1.2. Project Highlights

The project distinguishes itself through:

- Educational Focus: Prioritizes clarity in visualizing algorithmic trade-offs (e.g., BFS guarantees shortest paths, while DFS prioritizes speed).
- Dynamic Adaptability: Users can modify grids mid-simulation and compare algorithm behaviors under varying constraints.
- Performance Optimization: Smooth animations achieved through asynchronous scheduling and React's state management.

2. Objectives

The project aims to:

Clarify Algorithmic Behavior: Translate abstract DSA concepts into intuitive visual workflows.

Facilitate Comparative Analysis: Enable users to observe differences in algorithmic efficiency (time/space complexity) and path optimality.

Promote Active Learning: Provide hands-on experimentation with parameters (e.g., speed, maze types) to reinforce theoretical knowledge.

Demonstrate Technical Implementation: Showcase modern web development practices (React, TypeScript, Tailwind CSS) in building responsive, maintainable systems.

3. Technical Methodology

3.1 Frontend Architecture

- React + TypeScript: Component-based architecture for modularity and type safety.
- State Management: Context API for centralized control of grid states, algorithm settings, and speed configurations.
- Styling: Tailwind CSS for responsive design and utility-first styling.

3.2 Algorithm Implementation

Pathfinding: Dijkstra, A*, BFS, DFS

Maze Generation: Recursive Division, Binary Tree

3.3 Performance Enhancements

Animation Scheduling: Asynchronous setTimeout loops to balance smooth rendering and computational load.

Memoization: Optimized re-renders using React.memo and useMemo hooks.

3.4 User Experience (UX)

Event Handling: Wall creation with mouse event listeners.

Error Boundaries: Graceful handling of edge cases (e.g., no path found).

4. Developer team

Full Name	Student ID	Contribution
Le Hung	**ITCSIU22271**	Team leader: Double check code and algorithm Data Structure: Recursive Division Algorithms: A-Star, Dijkstra Fix bugs
Tran Dang Nhat	**ITITIU22115**	Data Structure: Binary tree Algorithms: DFS, BFS Fix bugs

UI and Design

This is a single-page web application designed to visualize path traversal using various pathfinding algorithms, including BFS, DFS, Dijkstra, and A*. The user interface is built with TailwindCSS and ReactJS (with Vite), providing a clean, responsive, and interactive experience.

The application consists of three main components:

1. Tile:

The fundamental unit of the grid. Each tile is a square block that can represent a wall or an open path. Its state determines whether it is traversable by the algorithm during execution. There are the starting (green) and end (yellow) tiles, the wall tiles will be white.

2. Grid:

A large matrix composed of multiple tiles (39 ROWS and 49 COLUMNS). It serves as the visual and functional space where the algorithms perform pathfinding.

3. Navigation Bar:

Located at the top, this bar includes key controls such as the play button, speed timer, and algorithm, maze, speed selection dropdowns, allowing users to interact with and control the visualization process.

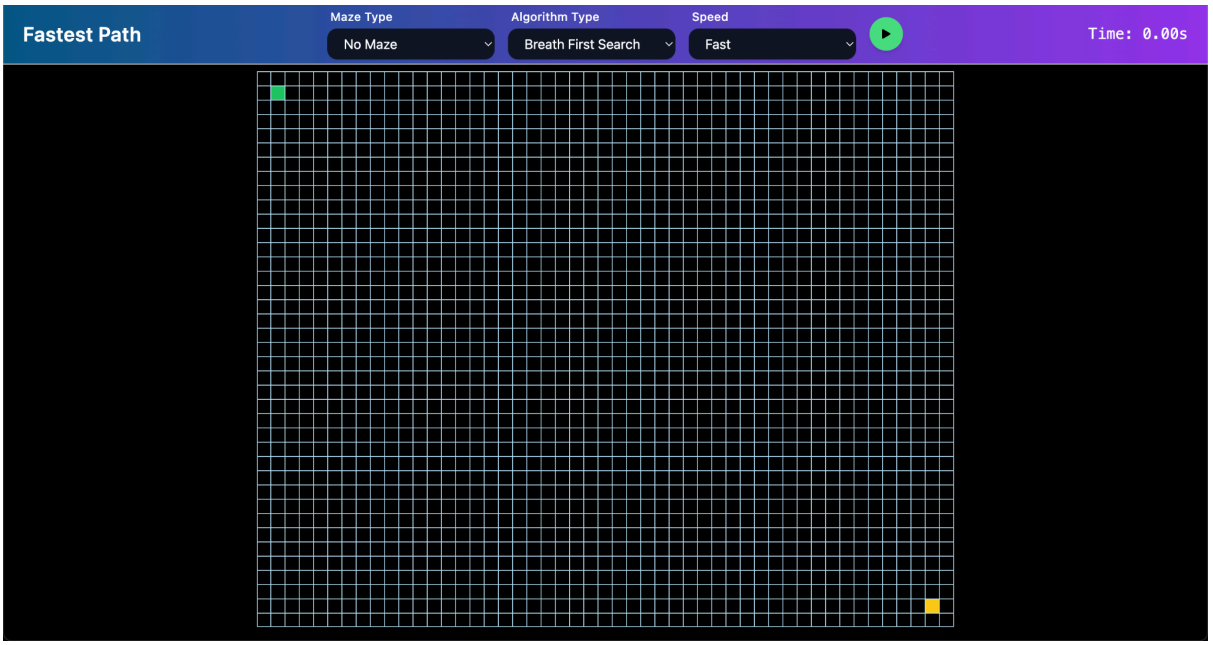

Figure 1. A grid

Data Structure and Grid Generation

- **2D Array** is used for storing the tiles in the grid
- Stack is used to execute the Depth First Search algorithm.
- Queue is used to execute the Breadth First Search algorithm.
- Min-Heap (Priority Queue) is used to optimize the time complexity for Dijkstra and A* algorithms.

1. No maze

Users can design the maze by what they want, and assign the wall tiles by themself.

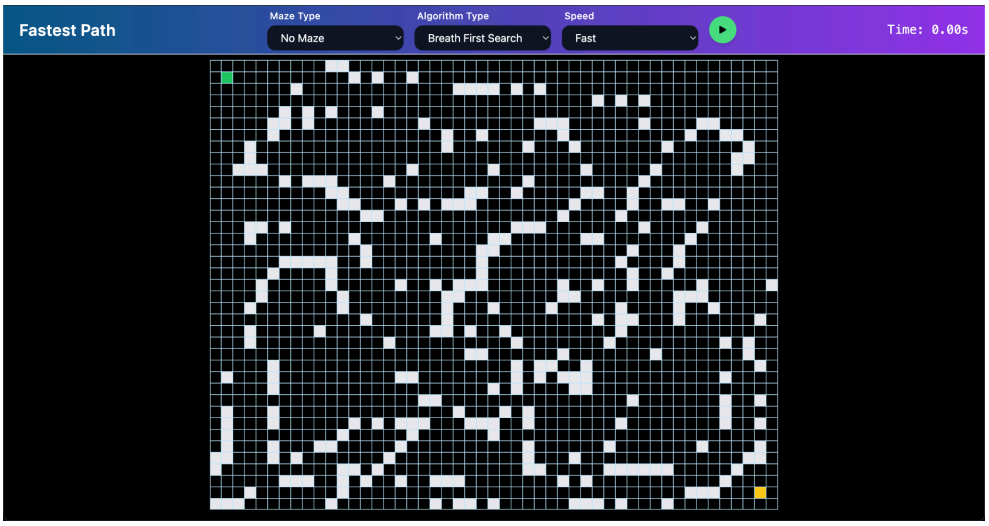

Figure 2. Arbitrarily generated matrix

2. Binary tree

The Binary Tree algorithm relies on a grid-based structure where each cell (or node) decides whether to carve a passage either north or west (or some other consistent pair of directions, depending on implementation). In this project, it is adapted to work with a 2D matrix representing the maze.

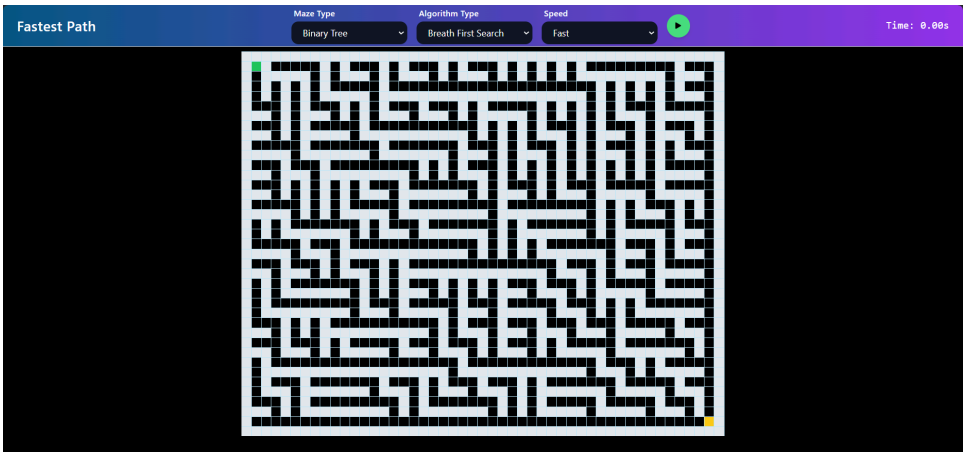

Figure 3. A binary tree maze

2.1 Purpose in this project

In the context of the Pathfinding Visualizer, the Binary Tree algorithm is used to generate perfect mazes - mazes with no loops and only one possible path between any two points. It serves as a lightweight yet visually intuitive method for users to test pathfinding algorithms under non-randomized, structured conditions.

2.2 How it work

Initialization

Start with a grid full of walls.

Iterate through each cell in the grid (except boundary cells if needed).

Decision Rule

For each cell, randomly choose to "carve" a passage either up or left (or down/right depending on orientation).

If the cell is at the top row or leftmost column, only one direction is allowed to avoid going out of bounds.

Repeat

Continue until all cells are processed.

2.3 Data Representation

The grid is represented as a 2D array of Tile objects.

Each Tile has properties like isWall, isStart, isEnd, etc.

The Binary Tree modifies the isWall property to carve out the maze.

2.4 Complexity

Time Complexity

$O(n \times m)$ where n is the number of rows and m is the number of columns. Each cell is visited once.

Space Complexity

$O(1)$ additional space - the algorithm operates in-place on the grid.

Advantage

Extremely fast and simple to implement.

Always produces a valid maze (perfect maze).

Good for illustrating basic pathfinding principles like directionality and search breadth.

Limitation

The resulting mazes are biased (e.g., tend to have longer passages in one direction).

Lack of variety compared to more advanced generators (e.g., Recursive Division).

No loops or multiple paths - makes it less challenging for algorithms like BFS or A*.

3. Recursive division

The Recursive division maze is randomly generated by two scenarios: horizontal division and vertical division. It splits the grid into smaller sections by adding walls with single gaps, recursively, until all sections are too small to divide. This creates a maze with a "rooms and corridors" look, animated in real time.

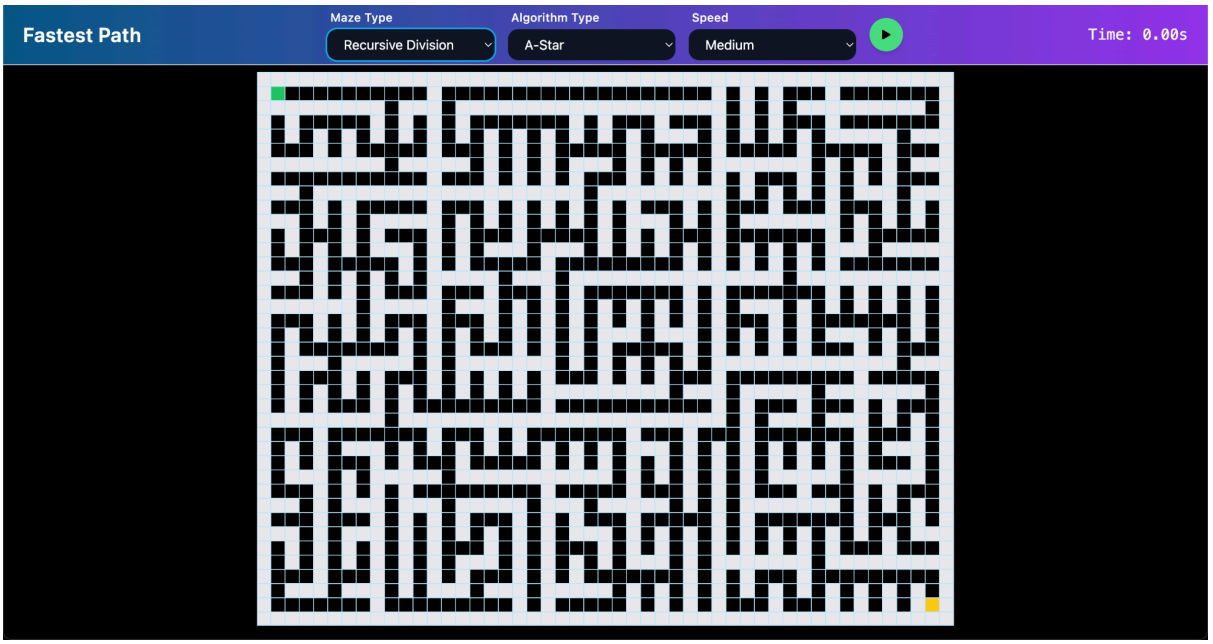

Figure 4. A maze created by recursive division

Pseudocode:

```
function recursiveDivision(row, col, height, width):
    if height <= 1 or width <= 1:
        return

    if height > width:
        horizontalDivision(row, col, height, width)
    else:
        verticalDivision(row, col, height, width)

////////////////////////////////////////////////////////////
function horizontalDivision(row, col, height, width):
    choose a random row → makeWallAt
    choose a random col for passage → makePassageAt

    for each col in width:
        if col != makePassageAt:
```

```
            place wall at (makeWallAt, col)

    // Recurse on upper and lower sections
    recursiveDivision(top half)
    recursiveDivision(bottom half)

////////////////////////////////////////////////////////
function verticalDivision(row, col, height, width):
    choose a random column → makeWallAt
    choose a random row for passage → makePassageAt

    for each row in height:
        if row ≠ makePassageAt:
            place wall at (row, makeWallAt)

    // Recurse on left and right sections
    recursiveDivision(left half)
    recursiveDivision(right half)
```

Algorithms

1. Depth First Search (DFS)

DFS operates by traversing deeper into the graph whenever possible. It uses a stack data structure - either explicitly with a stack or implicitly through recursion - to keep track of the path being explored. Nodes are marked as visited to prevent repeated processing.

1.1 Pseudocode

```
DFS(node):
    mark node as visited
    for each neighbor of node:
        if neighbor is not visited:
            DFS(neighbor)
```

1.2 Time and Space Complexity

Time Complexity: O(V + E), where V is the number of vertices and E is the number of edges, because each node and edge is explored at most once.

Space Complexity: O(V) due to the visited set and recursion stack.

1.3 Application in the Project

In this project, DFS is used for pathfinding on the grid-based graph. It explores nodes deeply, making it efficient when the path to the goal is expected to be found along a particular branch. However, it does not guarantee the shortest path.

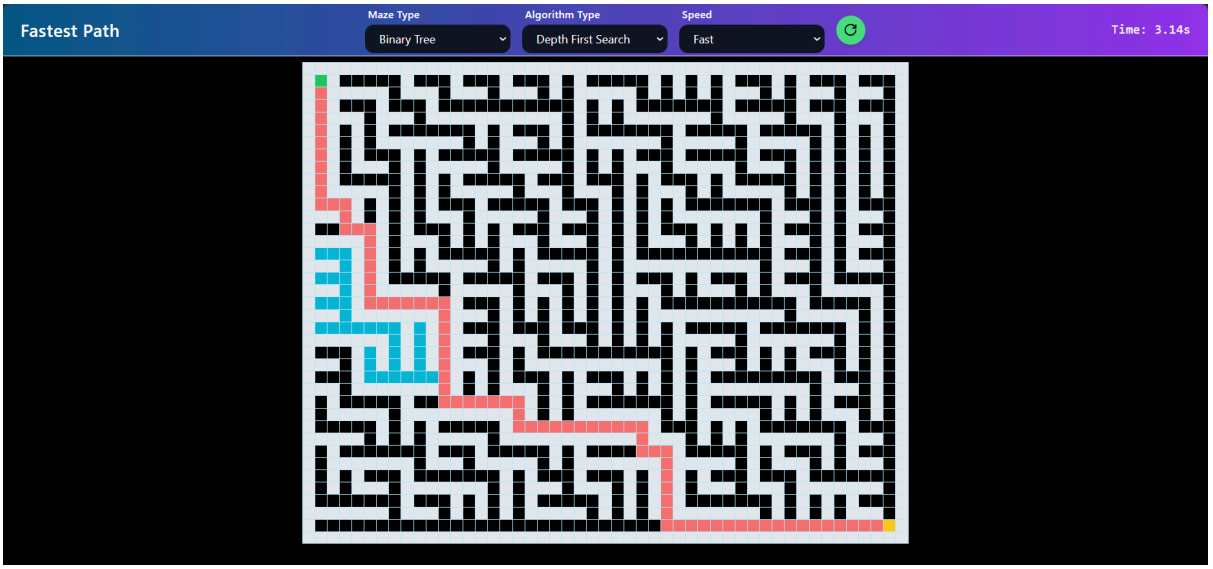

Figure 5. Optimal path finding by DFS

2. Breadth First Search (BFS)

BFS uses a queue data structure to explore neighbors in the order they are discovered. This guarantees the shortest path in an unweighted graph since it visits nodes in increasing order of their distance from the source.

2.1 Pseudocode

```
BFS(start_node):
    create a queue Q
    enqueue start_node onto Q
    mark start_node as visited
    while Q is not empty:
        node = dequeue Q
        for each neighbor of node:
            if neighbor is not visited:
                enqueue neighbor onto Q
                mark neighbor as visited
```

2.2 Time and Space Complexity

Time Complexity: O(V + E), similar to DFS.

Space Complexity: O(V), for the queue and visited tracking.

2.3 Application in the Project

BFS is implemented to find the shortest path between the start and end nodes on the grid, as BFS ensures the minimal number of steps in an unweighted graph. Its level-wise exploration is ideal for maze solving and shortest path visualization.

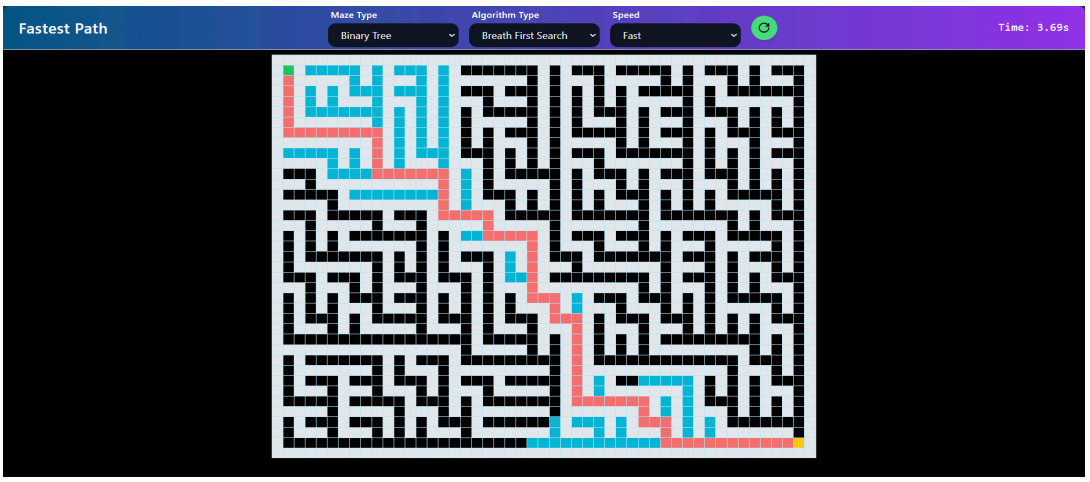

Figure 6. Optimal path finding by BFS

3. Dijkstra Algorithm

Dijkstra's algorithm explores the grid, always expanding the closest unvisited tile, updating distances, and tracking the shortest path. It stops when the end tile is reached, then reconstructs the path by following parent pointers.

3.1 Pseudocode

```
function dijkstra(grid, startTile, endTile):
    set startTile.distance = 0
    minHeap = new MinHeap ordered by tile.distance
    minHeap.push(startTile)
    traversedTiles = []

    while minHeap is not empty:
        currentTile = minHeap.pop()
        if currentTile is a wall or already traversed:
            continue
        if currentTile.distance == Infinity:
            break

        mark currentTile as traversed
        add currentTile to traversedTiles

        if currentTile is endTile:
            break

        for each untraversed neighbor of currentTile:
            if currentTile.distance + 1 < neighbor.distance:
                neighbor.distance = currentTile.distance + 1
                neighbor.parent = currentTile
                minHeap.push(neighbor)

    // Reconstruct path
    path = []
    current = endTile
    while current is not null and current.parent is defined:
        mark current as part of path
        insert current at start of path
        current = current.parent

    return traversedTiles, path
```

3.2 Time complexity and space complexity:

- **Time complexity**: O(E log V)

  V = number of tiles (vertices)

  E = number of edges (neighbor connections)

Each tile may be pushed/popped from the heap, each heap operation is O(log V).

For a grid, E is O(V), so overall: O(V log V) for a sparse grid.

- **Space complexity**: O(V)

  For the grid, MinHeap, traversedTiles, and path arrays.

  Each stores up to V tiles.

3.3 Application in project:

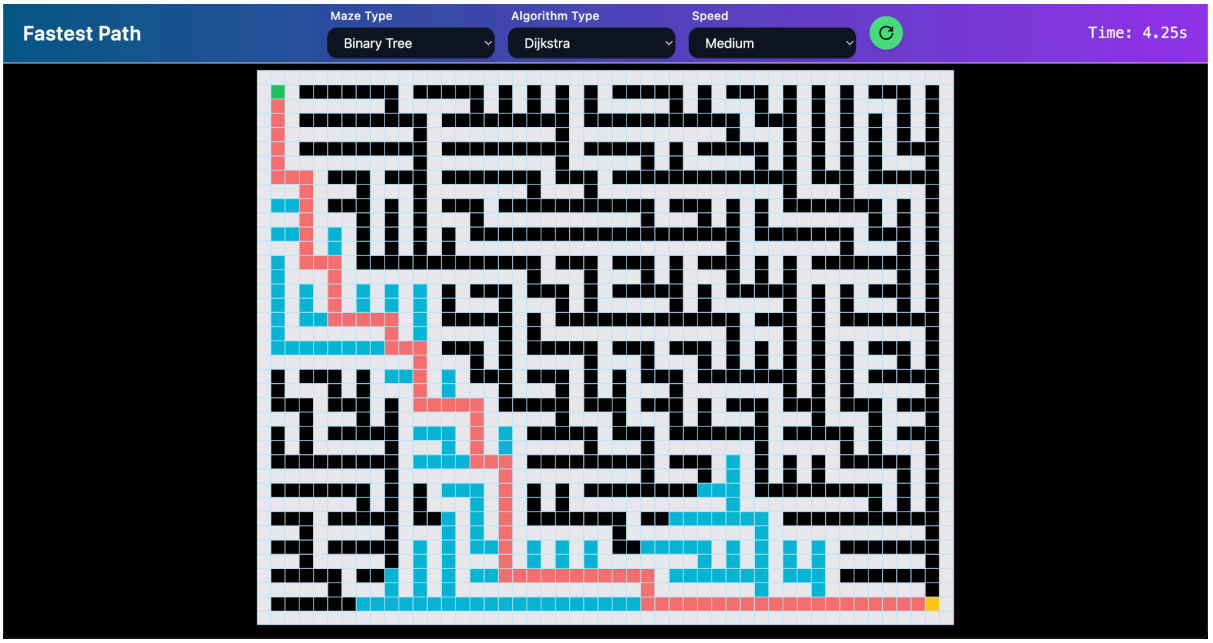

Figure 7. Optimal path finding by Dijkstra algorithm

4. A star Algorithm

A* (A-Star) is a best-first search algorithm that finds the shortest path from a start node to a goal node, using both:

g(n): the cost from start to current node.

h(n): a heuristic estimate of the cost to goal

It selects the next nodes based on: f(n) = g(n) + h(n)

→ This makes it complete, optimal (with an admissible heuristic), and efficient in many practical cases.

4.1 Pseudocode of A* algorithm:

```
function AStar(grid, start, end):
    heuristicCost ← compute h(n) for all nodes
    functionCost ← initialize all to ∞
    functionCost[start] ← h(start)
    start.distance ← 0
```

```
    openSet ← MinHeap ordered by functionCost
    openSet.push(start)

    while openSet is not empty:
        current ← openSet.pop()
        if current is end:
            break

        mark current as traversed
        for each neighbor of current:
            if neighbor is wall or already traversed:
                continue

            tentativeDistance ← current.distance + 1
            if tentativeDistance < neighbor.distance:
                neighbor.distance ← tentativeDistance
                functionCost[neighbor] ← tentativeDistance +
heuristicCost[neighbor]
                neighbor.parent ← current
                openSet.push(neighbor)

    path ← backtrack from end using parent pointers
    return path and traversed tiles
```

4.2 Time complexity and Space complexity

- **Time complexity:** O(E log V)

  V = number of tiles (vertices)

  E = number of edges (neighbor connections)

  Each tile can be pushed/popped from the MinHeap multiple times, each operation is O(log V).

  For a grid, E is O(V), so overall: O(V log V) for a sparse grid.

- **Space Complexity:** O(V), where V is the number of tiles in the grid.

4.3 Application in project:

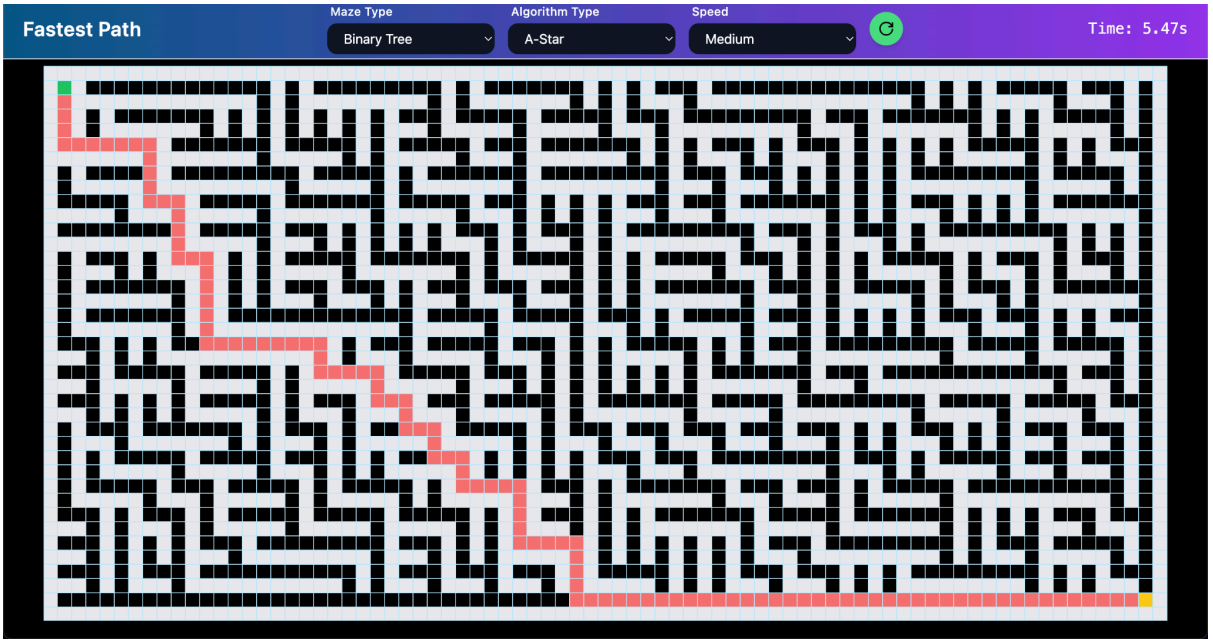

Figure 8. Optimal path finding by A* algorithm

5. Comparison the path finding algorithms

By implementing all of the path finding algorithms, here is the summary table of their characteristics:

Algorithm	Time Complexity	Space Complexity	Note
BFS	O(V + E)	O(V)	Just for an unweighted grid.
DFS	O(V + E)	O(V)	Not guaranteeing the fastest path.
Dijkstra	O(E log V)	O(V)	Explore all directions
A *	O(E log V)	O(V)	Use Heuristic, usually the fastest method.

Final Web Application

1. How to use

Step 1: Accessing the Web Application

Clone the project repository using the command:

```
git clone https://github.com/hungCS22hcmiu/Pathfinding-DSA.git
```

Navigate to the project folder:

```
cd Pathfinding-DSA
```

Install dependencies:

```
npm install
```

Start the development server:

```
npm run dev
```

Open your web browser and go to the local server address to access the application.

Step 2: Generating mazes or custom grid

Use the Maze Type dropdown menu to select a maze generation algorithm or choose an empty grid to start from scratch.

You can manually add or remove walls by clicking and dragging on the grid.

Step 3: Choosing pathfinding algorithms

Select your desired algorithm from the Algorithm Type dropdown, such as BFS, DFS, Dijkstra, or A*.

Step 4: Running the Visualization

Click the Play button to start the animation.

Watch the algorithm explore the grid and find a path from the start node to the end node.

Step 5: Resetting and start again

After the visualization finishes, use the Reset button to clear the grid and try different maze configurations or algorithms.

2. Demo video

To better understand how the Pathfinding Visualizer works, please watch the demo video at the following link: https://youtu.be/-rMLO_MHN4I

References

GeeksforGeeks. (2025, May 21). *Dijkstra's Algorithm to find Shortest Paths from a Source to all*. GeeksforGeeks. https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/

GeeksforGeeks. (2024, July 30). *A* search algorithm*. GeeksforGeeks. https://www.geeksforgeeks.org/a-search-algorithm/

Amit's a pages*. (2025). https://theory.stanford.edu/~amitp/GameProgramming/

Tech K. (n.d.). *DFS and DFS Traversal*. KungFuTech. https://kungfutech.edu.vn/bai-viet/algorithms/depth-first-search-dfs-depth-first-traversal

GeeksforGeeks. (2024a, February 19). *BFS vs DFS for Binary Tree*. GeeksforGeeks. https://www.geeksforgeeks.org/bfs-vs-dfs-binary-tree/

W3Schools.com. (n.d.). https://www.w3schools.com/REACT/DEFAULT.ASP

TypeScript Tutorial. (2024, August 24). *TypeScript Tutorial*. https://www.typescripttutorial.net/

Geeksforgeeks. (2020, June 24). *Introduction to Tailwind CSS. GeeksforGeeks.* https://www.geeksforgeeks.org/introduction-to-tailwind-css/

REFERENCE Video: https://www.youtube.com/watch?v=fLpvgCVYjTo