INTRODUCTION TO ARTIFICIAL INTELLIGENCE

Course Project #2

SEARCHING

This document describes the course project content for the Introduction to Artificial Intelligence course in the direction of Computer Science major.

Group 01

23127008 Nguyen Trong Tai

23127181 Nguyen Tran Quoc Duy

23127207 Dang Dang Khoa

23127438 Dang Truong Nguyen

Faculty of Information Technology
University of Science - HCM

Ho Chi Minh, July 2025

Mục lục

1 Group Introduction

1.1 Member Information

23127008	**Nguyen Trong Tai**	**nttai23@clc.fitus.edu.vn**
23127181	Nguyen Tran Quoc Duy	ntqduy23@clc.fitus.edu.vn
23127207	Dang Dang Khoa	ddkhoa23@clc.fitus.edu.vn
23127438	Dang Truong Nguyen	dtnguyen23@clc.fitus.edu.vn

1.2 Overall Completion

Problem	Detail	Percentage
Problem Definition	Find the shortest or optimal path from a designated start point to a goal point in a 2D maze that contains obstacles (walls) using various search algorithms.	100%
Algorithm Implementation	Implement search algorithms including DFS, BFS, A*, UCS, IDDFS, Bi-directional Search, Beam Search, and IDA*, with integrated path visualization and performance tracking features.	100%
Graphical User Interface	Design an interactive web-based GUI that allows users to generate mazes, draw walls, select start and goal positions, and visually observe the execution of various search algorithms in real time.	100%
Presentation Slide	Presentation slides provide an overview of the problem, selected algorithms, implementation strategies, and demonstrate the results through visuals and comparisons.	100%
Demo Video	A demo video showcases the main system functionalities, user interactions with the interface, and the pathfinding results of each algorithm.	100%

2 Problem Modeling

State Definition:

- State: A state is defined as the current position of the agent in the maze, represented by coordinates (x, y)
- Initial State: Starting position S = (x_o, y_o)
- Goal State: Target position G = (x_m, y_m)

State Space Structure

State Space = {(x, y) | 0 ≤ x < width, 0 ≤ y < height, maze[x][y] ≠ wall}

Action Set

- From each state (x, y), the agent can perform the following actions:
  - **UP**: (x, y) → (x, y-1)
  - **DOWN**: (x, y) → (x, y+1)
  - **LEFT**: (x, y) → (x-1, y)
  - **RIGHT**: (x, y) → (x+1, y)
- **Validity Condition:** An action is valid only if the destination position is not a wall and remains within maze boundaries.

Transition Model

- *Result(s, a) = s' if action a is valid from state s*
- *Result(s, a) = undefined if action a leads to wall or out of bounds*

Path Cost Function

- Uniform Cost: Each movement step has cost = 1
- Variable Cost: Cost may vary depending on terrain type or distance

State Space Properties

- Finite: Number of states is limited by maze dimensions
- Discrete: States are discrete, not continuous
- Fully Observable: Agent knows its exact current position
- Deterministic: Action outcomes are predetermined

Grid Representation

Cell value in the maze grid	Meanings	Color (Hex)
0	Free space (Walkable)	#ffffff75
1	Wall/Obstacle (Blocked)	#01122C
2	Start position (Initial state)	#ffffff with Home icon
3	Goal position (Target state)	#ffffff with Target icon
4	Explored nodes (During search)	#77E2E4
5	Solution path (Final result)	#ffa500

State Space Size

- For an 25x25 maze with w walls (Can be random generated or drawed):
  - Total cells: 25 × 25
  - Valid states: (25 × 25) - w
  - Branching factor: Maximum 4 (up, down, left, right)

3 Algorithm Principles

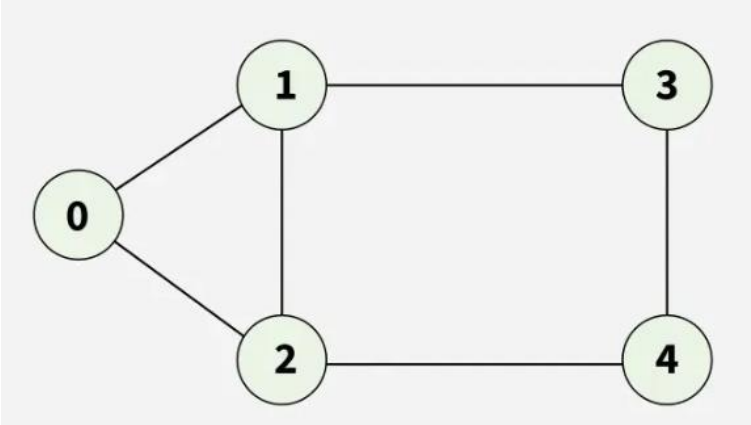
3.1 Breadth-First Search (BFS)

Theoretical Explanation

- Breadth-First Search explores the nodes level by level. It starts from the root node and explores all neighboring nodes before moving to the next level of neighbors.
- It uses a **queue (FIFO)** to keep track of nodes to be explored next.
- BFS guarantees the shortest path (in terms of the number of edges) if all edge costs are equal.

Simple Example

Input: adj[][] = [[1,2], [0,2,3], [0,1,4], [1,4], [2,3]]



Output: [0, 1, 2, 3, 4]
Explanation: Starting from 0, the BFS traversal will follow these steps:
Visit 0 → Output: [0]
Visit 1 (first neighbor of 0) → Output: [0, 1]
Visit 2 (next neighbor of 0) → Output: [0, 1, 2]
Visit 3 (next neighbor of 1) → Output: [0, 1, 2, 3]
Visit 4 (neighbor of 2) → Final Output: [0, 1, 2, 3, 4]

3.2 Depth-First Search (DFS)

Theoretical Explanation

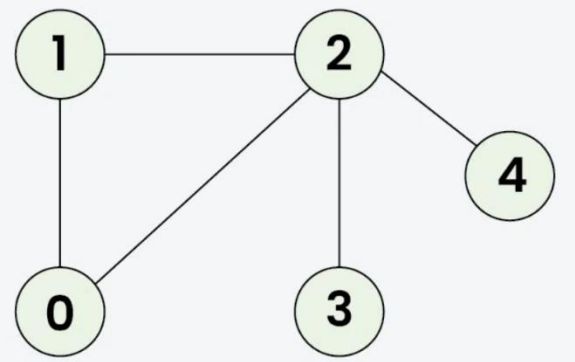
- Depth-First Search explores as far down a branch as possible before backtracking.
- It uses a **stack (LIFO)** — either explicitly or via recursion — to remember which nodes to visit next.

- DFS does not guarantee the shortest path, and in infinite-depth graphs, it may get stuck unless depth limits or visited tracking are used.

Simple Example

Note : There can be multiple DFS traversals of a graph according to the order in which we pick adjacent vertices. Here we pick vertices as per the insertion order.

Input: adj = [[1, 2], [0, 2], [0, 1, 3, 4], [2], [2]]



Output: [0 1 2 3 4]

Explanation: The source vertex s is 0. We visit it first, then we visit an adjacent.

Start at 0: Mark as visited. Output: 0
Move to 1: Mark as visited. Output: 1
Move to 2: Mark as visited. Output: 2
Move to 3: Mark as visited. Output: 3 (backtrack to 2)
Move to 4: Mark as visited. Output: 4 (backtrack to 2, then backtrack to 1, then to 0)

Multiple DFS traversals of a graph are possible. The traversal order depends on which adjacent node is visited first; for example, after visiting node 1, we could visit node 2 instead of 0, resulting in a different DFS traversal. The insertion order determines the selection.

3.3 Uniform Cost Search (UCS)

Theoretical Explanation

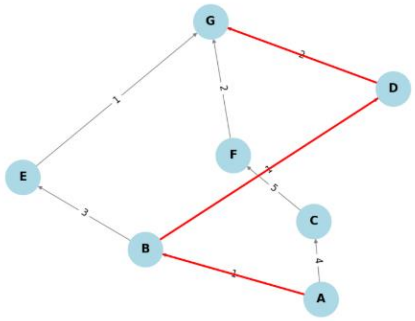
- Uniform Cost Search is a search algorithm that expands the node with the **lowest total path cost** from the start node.
- It uses a **priority queue**, where the node with the smallest cumulative cost g(n) is expanded first.
- UCS is **optimal** and **complete** if all costs are non-negative.

Simple Example

Example graph represented as an adjacency list

```
graph = {
  'A': [('B', 1), ('C', 4)],
```

```
    'B': [('D', 1), ('E', 3)],
    'C': [('F', 5)],
    'D': [('G', 2)],
    'E': [('G', 1)],
    'F': [('G', 2)],
    'G': []
}
```



1. Start at node A, with cost = 0. Frontier: [(A, 0)]
2. Expand A (cost 0): Neighbors:
    a. B with cost = 0 + 1 = 1
    b. C with cost = 0 + 4 = 4 Frontier: [(B, 1), (C, 4)]
3. Expand B (cost 1): Neighbors:
    a. D with cost = 1 + 1 = 2
    b. E with cost = 1 + 3 = 4 Frontier: [(D, 2), (C, 4), (E, 4)]
4. Expand D (cost 2): Neighbor:
    a. G with cost = 2 + 2 = 4 Frontier: [(C, 4), (E, 4), (G, 4)]
5. Expand C (cost 4): Neighbor:
    a. F with cost = 4 + 5 = 9 Frontier: [(E, 4), (G, 4), (F, 9)]
6. Expand E (cost 4): Neighbor:
    a. G with cost = 4 + 1 = 5 → but G already in frontier with cost 4 → skip Frontier: [(G, 4), (F, 9)]
7. Expand G (cost 4): Goal reached!

Final Result

- Path: A → B → D → G
- Total cost: 4

3.4 A* (A-star) Search

Theoretical Explanation

- A* search combines **Uniform Cost Search** and **greedy best-first search**.
- It selects the next node to expand based on the lowest estimated total cost:

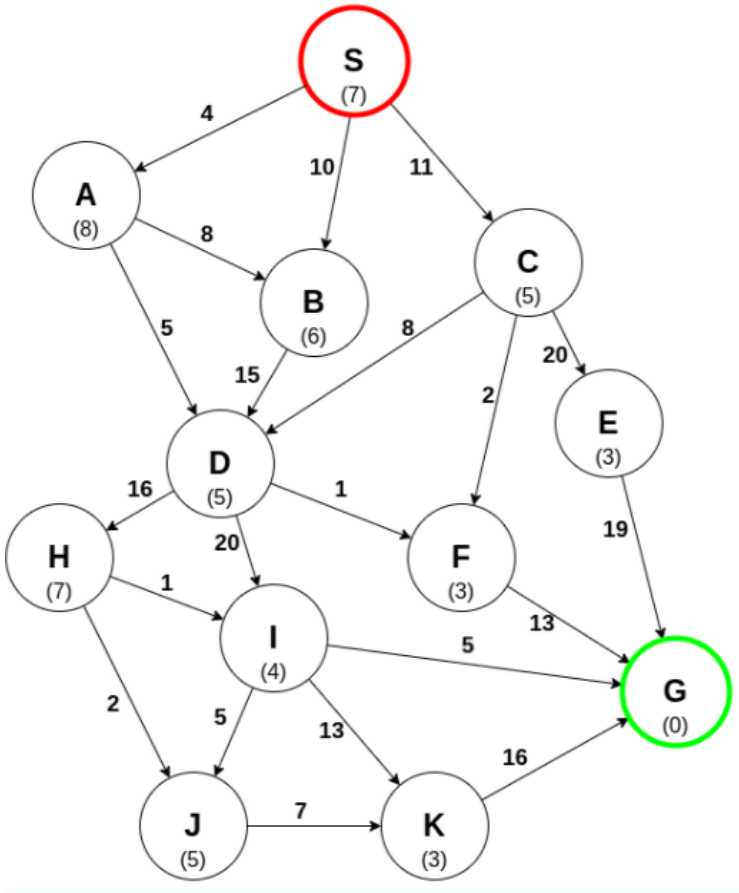
$$f(n)=g(n)+h(n)f(n) = g(n) + h(n)\text{f(n)=g(n)+h(n)}$$

- **g(n)** = actual cost from the start node to node *n*
- **h(n)** = heuristic estimate of the cost from *n* to the goal

If the heuristic **h(n)** is admissible (never overestimates), A* is **both optimal and complete**.

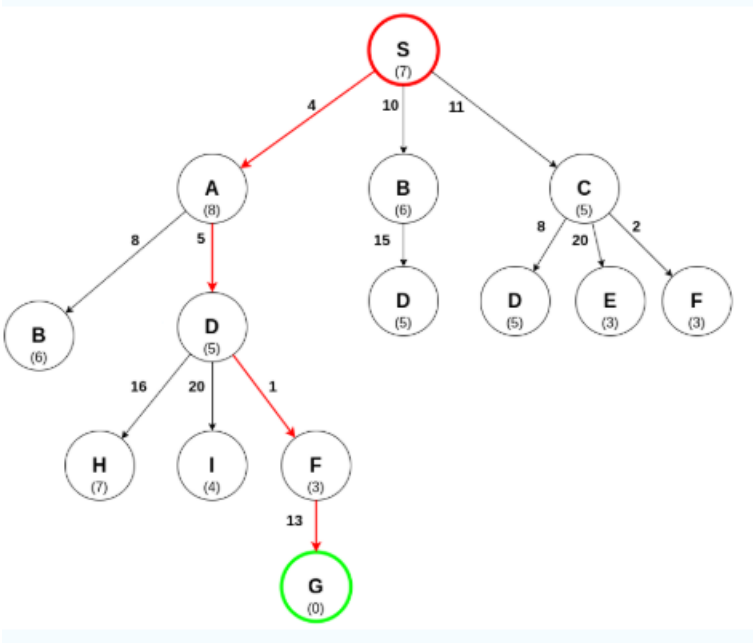
Simple Example:

Consider the following example of trying to find the shortest path from **S** to **G** in the following graph:



Each edge has an associated weight, and each node has a heuristic cost (in parentheses).

The result:

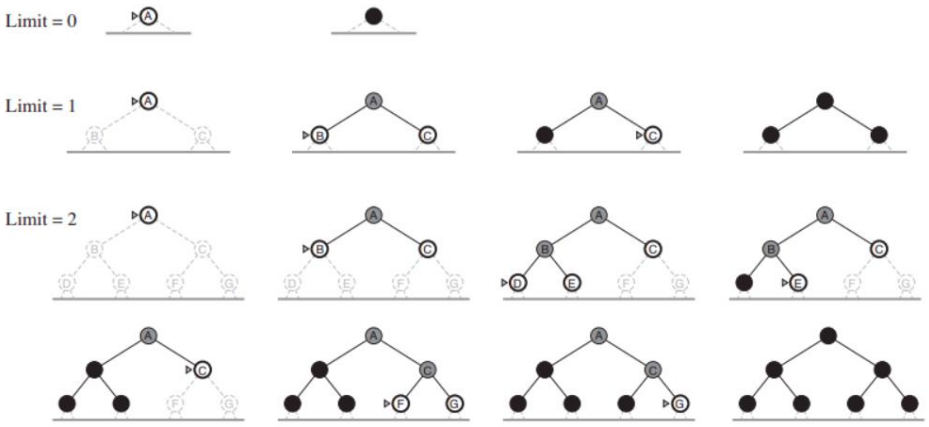


3.5 Iterative Deepening Depth-First Search (IDDFS)

Theoretical Explanation

- IDDFS performs repeated **depth-limited DFS** up to increasing depth limits until the goal is found.
- Each iteration performs a DFS with a deeper limit:
  - Depth 0
  - Depth 1
  - Depth 2
  - ... until goal is found.
- It combines the **space efficiency of DFS** with the **completeness and optimality (for uniform cost)** of BFS.

Simple Example:

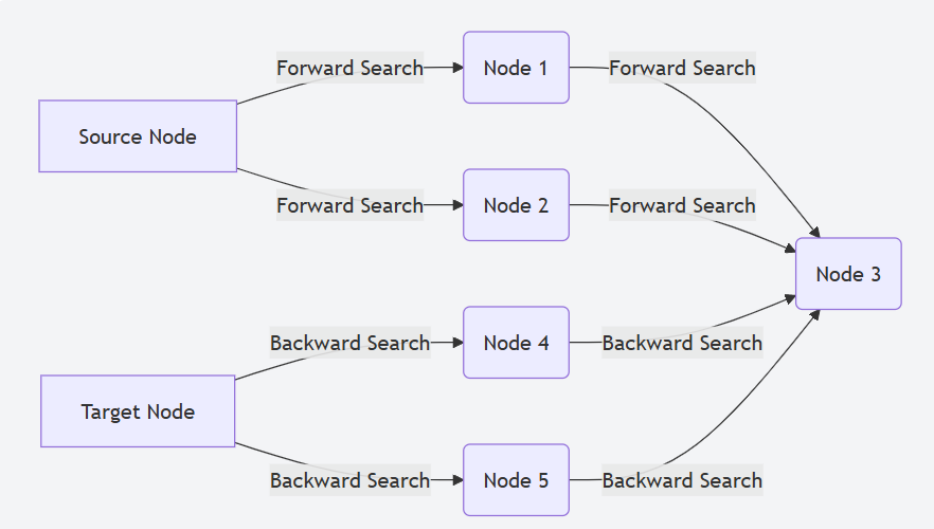


3.6 Bi-directional Search

Theoretical Explanation:

- Bi-directional search runs **two simultaneous searches**:
  - One forward from the **start node**
  - One backward from the **goal node**
- The search stops when the two frontiers **meet**.
- The biggest advantage: reduces the search space from $O(b^d)$ to approximately $O(b^{d/2})$ where b is the branching factor and d is the depth of the solution. This is a **massive** performance gain.
- Uses two BFS (or UCS/A*, depending on the implementation)
- Requires that the path from goal to start is **reversible**

Simple Example:

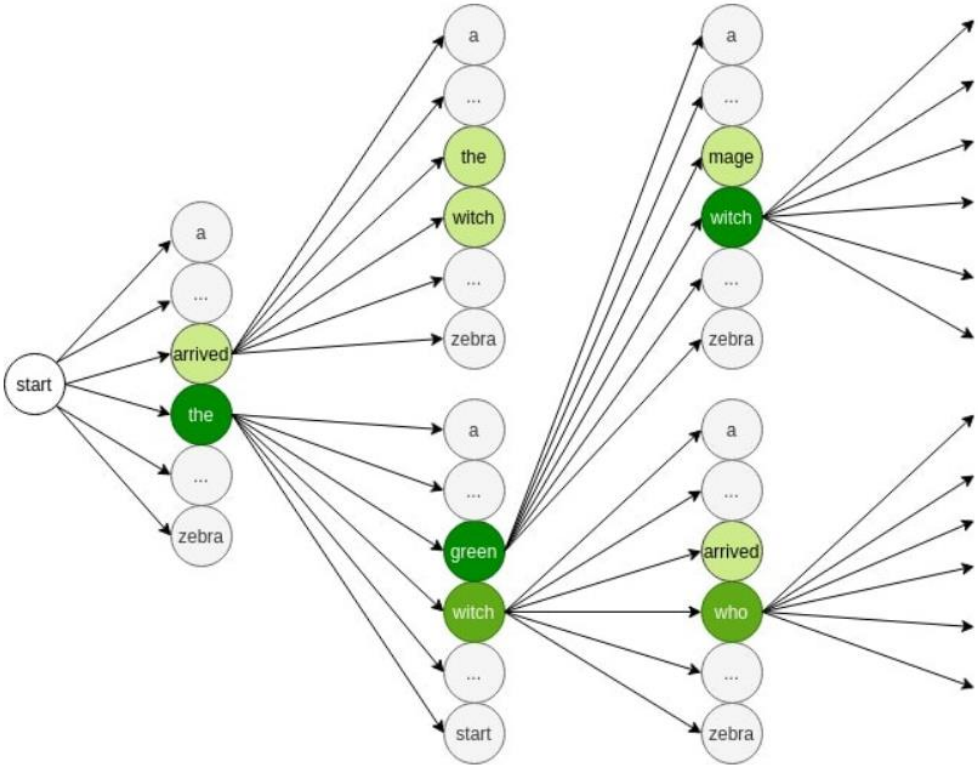


3.7 Beam Search

Theoretical Explanation:

- Beam Search is a **heuristic search algorithm** that explores a graph by expanding only the most promising nodes at each level.
- Instead of expanding **all** children like BFS, it maintains only a **fixed number k (beam width)** of best nodes at each depth, based on a heuristic score.
- It reduces memory and computation cost.
- It is **not complete** (may miss the goal).
- It is **not optimal** (may miss the best path).
- Commonly used in **natural language processing** (e.g., speech recognition, machine translation).

Simple Example:



3.8 Iterative Deepening A* (IDA*)

Theoretical Explanation

- IDA* is a combination of **A*** and **Iterative Deepening DFS**.
- Instead of maintaining a priority queue like A*, it performs **depth-first searches** with increasing thresholds on the f(n) value:

$$f(n)=g(n)+h(n) f(n) = g(n) + h(n) \text{f(n)=g(n)+h(n)}$$

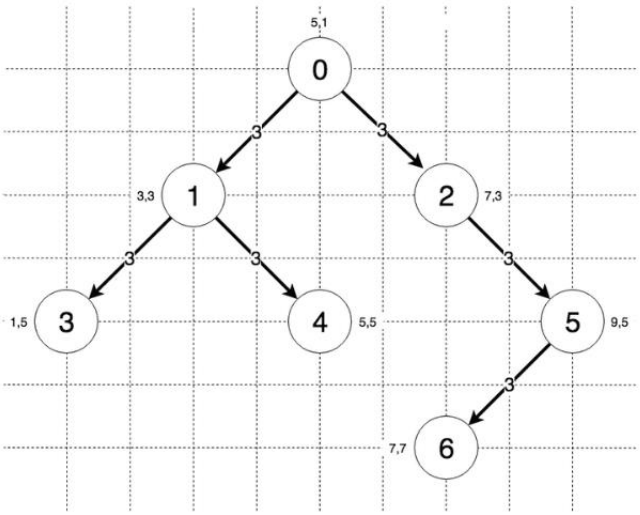
- Starts with a threshold equal to the heuristic of the start node.

- In each iteration, it performs DFS but only explores nodes where *f(n)≤thresholdf(n) \leq \text{threshold}*f(n)≤threshold.
- If the goal is not found, the threshold is increased to the minimum f(n) that exceeded the previous threshold.

Pros

- Much **lower memory** usage than A* (because it's DFS-based).
- Maintains **optimality and completeness** (with admissible heuristic).
- Especially useful in **large search spaces**.

Simple Example:



The steps the algorithm performs on this graph if given node 0 as a starting point and node 6 as the goal, in order, are:

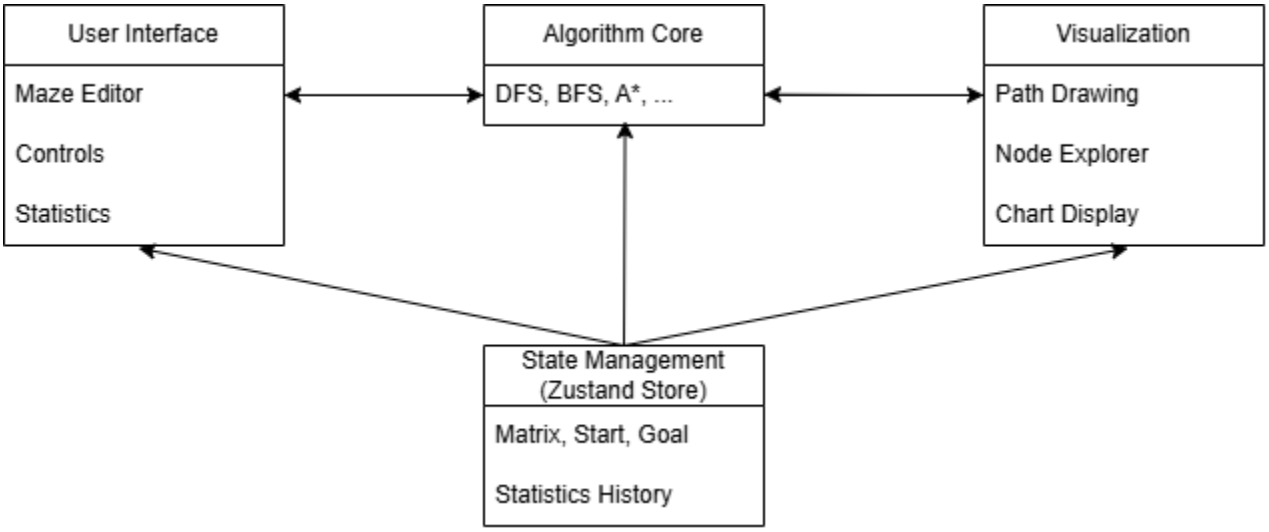
1. Iteration with threshold: 6.32
2. Visiting Node 0
3. Visiting Node 1
4. Breached threshold with heuristic: 8.66
5. Visiting Node 2
6. Breached threshold with heuristic: 7.00
7. Iteration with threshold: 7.00
8. Visiting Node 0
9. Visiting Node 1
10. Breached threshold with heuristic: 8.66
11. Visiting Node 2
12. Visiting Node 5
13. Breached threshold with heuristic: 8.83

14. Iteration with threshold: 8.66
15. Visiting Node 0
16. Visiting Node 1
17. Visiting Node 3
18. Breached threshold with heuristic: 12.32
19. Visiting Node 4
20. Breached threshold with heuristic: 8.83
21. Visiting Node 2
22. Visiting Node 5
23. Breached threshold with heuristic: 8.83
24. Iteration with threshold: 8.83
25. Visiting Node 0
26. Visiting Node 1
27. Visiting Node 3
28. Breached threshold with heuristic: 12.32
29. Visiting Node 4
30. Visiting Node 2
31. Visiting Node 5
32. Visiting Node 6
33. Found the node we're looking for!

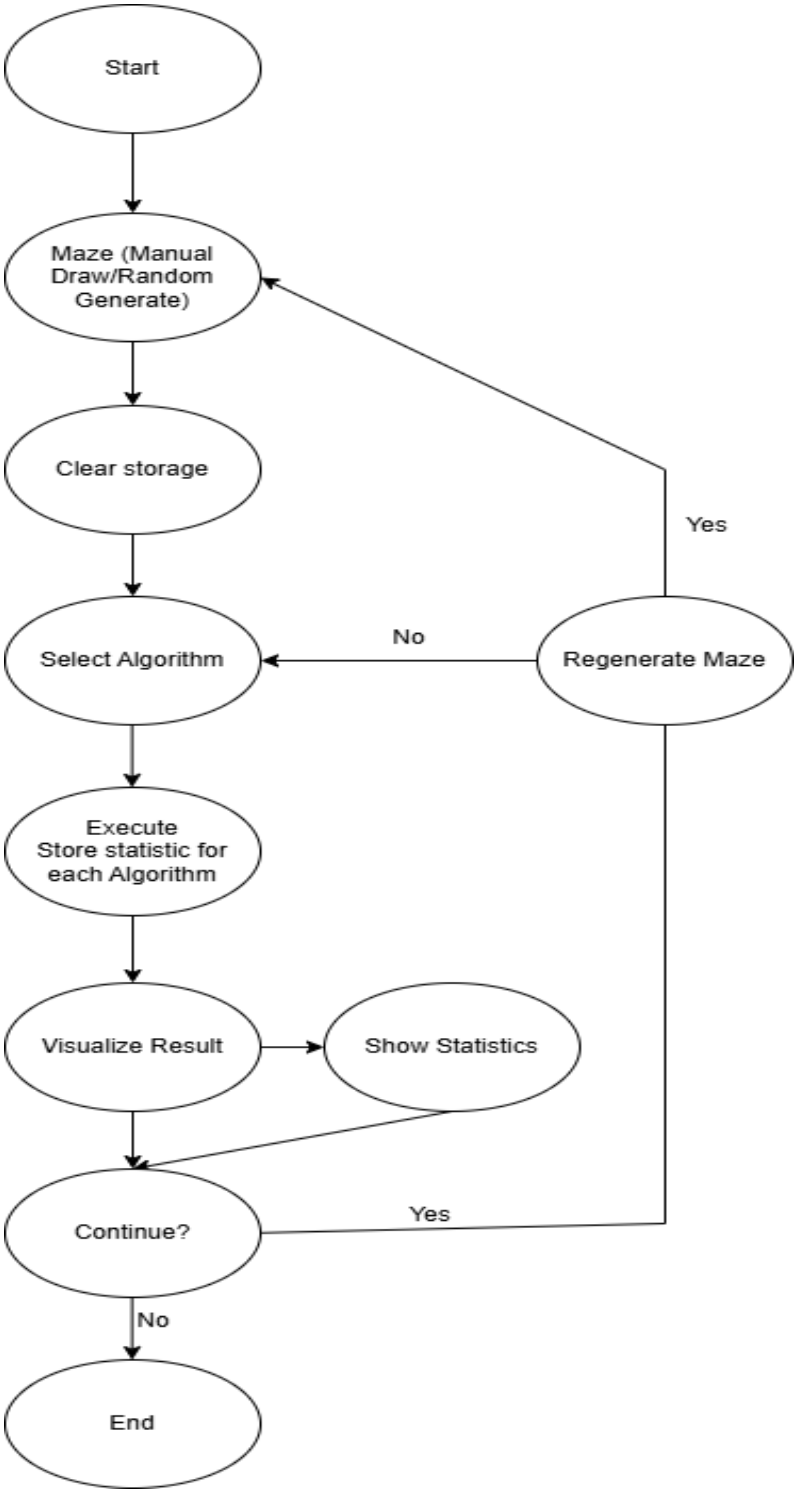
→ Final lowest distance from node 0 to node 6: 9

4 Program Flow

4. 1 Overall System Architecture Diagram


User Interface
Maze Editor
Controls
Statistics
Algorithm Core
DFS, BFS, A*, ...
Visualization
Path Drawing
Node Explorer
Chart Display
State Management
(Zustand Store)
Matrix, Start, Goal
Statistics History

4. 2 Main Application Flow



4.3 Explanation of program modules/functions.

4.3.1 Directory structure

```
AI-Searching/
├── Web/
│   └── client/
│       └── src/
│           └── app/
│               └── (pages)/
│                   └── (home)/page.js                    # Menu page
│                   └── analyse/
│                           └── cost/page.jsx             # Cost comparison page
│                           └── nodes-explored/page.jsx   # Nodes-explored comparison page
│                           └── path-length/page.jsx      # Path length comparison page
│                           └── processing-time/page.jsx  # Processing time comparison page
│                   └── game/page.jsx                     # Main maze game page
│                   └── team/page.jsx                     # Team's member information page
│           └── store/statStore.js                        # Zustand storage
│           └── helpers/algorithm.helper.js               # Searching algorithm
├── README.md
```

4.3.2 game/page.jsx

generateWalkableMatrix(size = 25): Generate random maze, ensuring there is a way from start to goal.

handleGenerate(): Toggle the click event of Generate button.

handleClear(): Toggle the click event of Clear button

drawPath(path, newMatrix): Set all the grid's value of helper function's return value to 5 (representing the path that connect start and goal)

resetMatrixStates(matrix): Clear all wall in maze grid (Set all grid value to 0).

findPoint(value, matrix): Find the coordinate of a value in grid.

handlePlay(): Capture the click event of Go button.

handleCellClick(): Capture the click event when user click to a cell of the maze.

handleCellDraw(): Capture user mouse click and drag events to the maze

4.3.3 helpers/algorithm.helper.js

runDFS(matrix, start, goal, onVisit, delay = 10):

- **Description**: Runs Depth-First Search on the matrix to find a path from start to goal.
- **Parameters**:
  - **matrix:** 2D array representing the maze (0=free, 1=wall, 2=start, 3=goal)
  - **start:** Starting position [row, col]
  - **goal:** Target position [row, col]
  - **onVisit:** Callback function called when visiting each node
  - **delay:** Animation delay in milliseconds (default: 10ms)
- **Returns:** Array of coordinates representing the path from start to goal, or null if no path exists

runBFS(matrix, start, goal, onVisit, delay = 10):

- **Description**: Runs Breadth-First Search on the matrix to find a path from start to goal.
- **Parameters**:
  - **matrix:** 2D array representing the maze (0=free, 1=wall, 2=start, 3=goal)
  - **start:** Starting position [row, col]
  - **goal:** Target position [row, col]
  - **onVisit:** Callback function called when visiting each node
  - **delay:** Animation delay in milliseconds (default: 10ms)
- **Returns:** Array of coordinates representing the path from start to goal, or null if no path exists

runIDDFS(matrix, start, goal, onVisit, delay = 5, maxDepth = 650)

- **Description**: Runs Iterative Deepening Depth-First Search, combining benefits of DFS and BFS.
- **Parameters**:
  - **matrix:** 2D array representing the maze (0=free, 1=wall, 2=start, 3=goal)
  - **start:** Starting position [row, col]
  - **goal:** Target position [row, col]
  - **onVisit:** Callback function called when visiting each node
  - **delay:** Animation delay in milliseconds (default: 5ms)
  - **maxDepth**: Maximum search depth limit (default: 650)
- **Returns:** Array of coordinates representing the path from start to goal, or null if no path exists

runAStar(matrix, start, goal, onVisit, delay = 10)

- **Description**: Runs A* search algorithm using Manhattan distance heuristic to find the optimal path.
- **Parameters**:
  - **matrix:** 2D array representing the maze (0=free, 1=wall, 2=start, 3=goal)
  - **start:** Starting position [row, col]
  - **goal:** Target position [row, col]
  - **onVisit:** Callback function called when visiting each node
  - **delay:** Animation delay in milliseconds (default: 10ms)
  - **maxDepth**: Maximum search depth limit (default: 650)
- **Returns:** Array of coordinates representing the path from start to goal, or null if no path exists

runUCS(matrix, start, goal, onVisit, delay = 10):

- **Description**: Runs Uniform Cost Search (Dijkstra's algorithm) to find the lowest-cost path.
- **Parameters**:
  - **matrix:** 2D array representing the maze (0=free, 1=wall, 2=start, 3=goal)
  - **start:** Starting position [row, col]
  - **goal:** Target position [row, col]
  - **onVisit:** Callback function called when visiting each node
  - **delay:** Animation delay in milliseconds (default: 10ms)
- **Returns:** Array of coordinates representing the path from start to goal, or null if no path exists

runBi_Directional_Search(matrix, start, goal, onVisit, delay = 10):

- **Description**: Runs bidirectional BFS from both start and goal simultaneously until they meet.
- **Parameters**:
  - **matrix:** 2D array representing the maze (0=free, 1=wall, 2=start, 3=goal)
  - **start:** Starting position [row, col]
  - **goal:** Target position [row, col]
  - **onVisit:** Callback function called when visiting each node
  - **delay:** Animation delay in milliseconds (default: 10ms)
- **Returns:** Array of coordinates representing the path from start to goal, or null if no path exists

runBeamSearch(matrix, start, goal, onVisit, delay = 10, beamWidth = 3)

- **Description**: Runs Beam Search with limited memory, keeping only the most promising nodes.
- **Parameters**:
  - **matrix:** 2D array representing the maze (0=free, 1=wall, 2=start, 3=goal)
  - **start:** Starting position [row, col]
  - **goal:** Target position [row, col]
  - **onVisit:** Callback function called when visiting each node
  - **delay:** Animation delay in milliseconds (default: 10ms)
  - **beamWidth**: Maximum number of nodes to keep at each level (default: 3)
- **Returns:** Array of coordinates representing the path from start to goal, or null if no path exists

runIDAStar(matrix, start, goal, onVisit, delay = 10)

- **Description**: Runs Iterative Deepening A* search combining IDA with A* heuristic evaluation.
- **Parameters**:
  - **matrix:** 2D array representing the maze (0=free, 1=wall, 2=start, 3=goal)
  - **start:** Starting position [row, col]
  - **goal:** Target position [row, col]
  - **onVisit:** Callback function called when visiting each node
  - **delay:** Animation delay in milliseconds (default: 10ms)
- **Returns:** Array of coordinates representing the path from start to goal, or null if no path exists

measurePath(algorithmFn, matrix, start, goal, onVisit, delay, ...args)

- **Description**: Wrapper function that measures performance statistics for any search algorithm.

- **Parameters**:
  - **algorithmFn**: The search algorithm function to execute
  - **matrix:** 2D array representing the maze (0=free, 1=wall, 2=start, 3=goal)
  - **start:** Starting position [row, col]
  - **goal:** Target position [row, col]
  - **onVisit:** Callback function called when visiting each node
  - **delay:** Animation delay in milliseconds (default: 10ms)
  - **…args**: Additional arguments passed to the algorithm

5 Algorithm Comparison

5.1 Comparison table

Algorithm	Time Complexity	Space Complexity	Optimality
BFS	O(V + E)	O(V)	Yes(due to its strategy of exploring nodes in order of increasing depth)
DFS	O(V + E)	O(H), H = max depth	No(may go deep along incorrect paths before finding the goal)
Uniform Cost Search (UCS)	$O(b^{C/\varepsilon})$ where C is cost of optimal solution, ε is minimum step cost	$O(b^{C/\varepsilon})$	Yes (if costs are non-negative)
A*	Exponential in worst case; faster with good heuristics	Exponential (stores open set)	Yes (if heuristic is admissible and consistent)
Iterative Deepening DFS (IDDFS)	$O(b^{d})$ – where d is depth of goal node	O(b·d)	Yes (if goal is at known or bounded depth)
Bi-directional Search	$O(b^{d/2})$	O(bd/2)	Yes (if both searches are complete and meet)
Beam Search	O(B·d) – B is beam width, d is depth	O(B·d)	No (may discard optimal path during pruning)
Iterative Deepening A* (IDA*)	Exponential, similar to A* with less overhead	O(d) (depth of solution)	Yes (if heuristic is admissible)

Notes:

- **b**: branching factor
- **d**: depth of the shallowest goal
- **C**: cost of the optimal path
- **ε**: minimum edge cost
- **B**: beam width
- **V** = number of vertices (maze cells)
- **E** = number of edges (connections between cells)

5.2 How each algorithm performed

5.2.1 Breadth-First Search (BFS) is a highly appropriate algorithm for solving mazes modeled as **undirected, unweighted graphs**.

Strengths:

- **Guarantees shortest path** in terms of the number of steps, which is perfect for mazes without weights.
- **Simple and efficient to implement** using a queue and a visited matrix or set.
- Works seamlessly with **undirected edges**, as long as visited nodes are tracked.
- Ideal when the goal is known and the shortest path is required.

Limitations:

- **High memory usage**, especially in large mazes, since BFS stores all frontier nodes at the same depth level.
- Explores **all nodes at the same depth** before going deeper, which can be inefficient if the goal is far and the maze has many branches.
- **Does not account for any environmental preferences**, such as traps, danger zones, or "easier" paths — since it treats all paths equally.

Special considerations:

- If the maze includes **multiple goals**, or if the objective is to find the closest of **many targets**, BFS still performs well.
- However, if the goal is unknown (e.g., "find any exit") or if the player only partially sees the maze (**fog-of-war scenarios**), BFS can be inefficient without adaptations.
- In **real-time gameplay**, BFS can be computationally expensive if triggered too often.

In summary, the Breadth-First Search algorithm proves to be highly effective for maze-solving tasks in static, unweighted environments where the shortest path is desired.

5.2.2 Depth-First Search (DFS) is generally less effective for solving mazes represented as **undirected, unweighted graphs**, particularly when the goal is to find the shortest path.

Strengths:

- DFS requires **less memory** than BFS, as it only stores the current path in a stack, rather than entire levels of nodes.
- It is **simple to implement**, either recursively or iteratively.
- DFS can be beneficial in applications such as **maze generation** or **full-space exploration**, where the objective is to traverse all reachable nodes.

Limitations:

- DFS **does not guarantee the shortest path**, since it explores one branch deeply before backtracking.
- Performance is **highly dependent on the graph's structure and neighbor visitation order**, which may lead to inefficient exploration in large or complex mazes.
- In **undirected graphs with cycles**—common in maze layouts—DFS must include cycle detection to prevent infinite loops.

Special considerations:

- In small-scale mazes or controlled scenarios, DFS may quickly reach the goal, but this is **non-deterministic** and unreliable for consistent shortest-pathfinding.
- DFS is often used in **recursive backtracking maze generation algorithms**, where deep traversal is preferred.
- For gameplay elements involving **hidden paths, dead ends, or collectible discovery**, DFS may be advantageous due to its thorough path-following behavior.

→ **To conclude**, although Depth-First Search offers memory efficiency and is well-suited for exploration or maze generation, it is suboptimal for shortest-path finding in static, unweighted, and undirected maze settings.

5.2.3 Uniform Cost Search (UCS) behaves similarly to Breadth-First Search (BFS) when applied to **undirected, unweighted graphs**, such as in the maze-solving scenario.

Strengths:

- UCS is **guaranteed to find the optimal path** in terms of total cost, making it suitable for graphs with weighted edges.
- It handles **variable-cost environments** well, which is beneficial when maze tiles might have different traversal costs (e.g., mud vs. road).
- Like BFS, UCS **systematically explores nodes** in order of path cost, ensuring no better route is missed.

Limitations:

- In **unweighted graphs**, UCS offers **no advantage over BFS**, as all edges have equal cost; it becomes functionally equivalent to BFS but with additional overhead from managing a priority queue.
- Requires **more computational resources** than BFS due to the need for a priority queue, even when unnecessary in unweighted settings.
- Like BFS, it may explore a large portion of the graph even if the goal is nearby but located through a higher-cost path (in weighted variants).

Special considerations:

- UCS becomes valuable if the maze environment is extended to include **variable-cost movement**, such as traps, terrain penalties, or power-ups affecting movement cost.
- If the maze is strictly unweighted and undirected, implementing UCS results in **extra complexity** with no performance gain compared to BFS.
- **Overall**, while UCS is a powerful and optimal pathfinding strategy in weighted graphs, its use in unweighted maze environments is **redundant** and incurs unnecessary overhead. BFS should be preferred unless cost-based traversal is introduced.

5.2.4 A* is a best-first search algorithm that uses both the actual path cost (g) and a heuristic estimate to the goal (h), making it highly effective in many pathfinding scenarios. In an **undirected, unweighted maze**, its performance is closely tied to the choice of heuristic.

Strengths:

- When equipped with a **consistent and admissible heuristic** (e.g., Manhattan distance for grid mazes), A* can **efficiently find the shortest path** while exploring fewer nodes than BFS.
- Combines the benefits of UCS (cost-awareness) and heuristic guidance, making it well-suited for **goal-directed search**.
- **Flexible**: easily adapted for different environments by changing the heuristic or incorporating penalties (e.g., trap zones, terrain).

Limitations:

- In **purely unweighted graphs**, if the heuristic provides little guidance (e.g., is zero or nearly uniform), A* can **degrade to UCS**, thus becoming no more efficient than BFS.
- Requires careful design of the heuristic; a poorly chosen heuristic can lead to **inefficiency or suboptimal paths** if it overestimates (i.e., non-admissible).
- Slightly more complex to implement due to the need for both a priority queue and heuristic calculations.

Special considerations:

- In grid-based mazes, commonly used heuristics like **Manhattan distance (for 4-directional movement)** or **Euclidean distance (for 8-directional movement)** often perform well and significantly reduce node expansions.
- A* performs best when the goal location is known and **heuristically close**, as it avoids unnecessary exploration typical of BFS or UCS.
- In **symmetric or open mazes**, heuristic effectiveness may drop, leading to broader exploration.

→ **In summary**, A* offers strong advantages in unweighted maze navigation if an effective heuristic is available. Its guided search allows it to outperform BFS in terms of speed and efficiency, despite being more complex in setup. This makes it a top candidate when shortest-path accuracy and real-time performance are both required.

5.2.5 Iterative Deepening DFS (IDDFS) combines the space-efficiency of DFS with the optimality of BFS. It repeatedly applies depth-limited DFS, increasing the depth limit step by step until the goal is found.

Strengths:

- **Finds the shortest path** in unweighted graphs, like BFS, but uses **less memory**, as it behaves like DFS in each iteration.
- Well-suited for large graphs or mazes where memory constraints are significant but shortest-path accuracy is still required.
- **Simple to implement** and naturally supports environments where the depth to the goal is not known in advance.

Limitations:

- Due to repeated traversals of the same nodes across iterations, IDDFS can be **less time-efficient** than BFS, especially in graphs with many shallow branches.
- In an undirected graph with **many cycles**, IDDFS must still track visited nodes within each depth-limited DFS to avoid infinite recursion, reducing its simplicity.
- Not as efficient as A* when a good heuristic is available, and not as fast as BFS when memory is not a constraint.

Special considerations:

- Particularly effective when the **maximum depth of the goal is relatively small**, and the maze has moderate branching.
- In real-time applications where a shallow solution is likely, IDDFS can often return results faster than BFS or UCS, especially in early iterations.
- Can be adapted to **anytime algorithms**, where approximate solutions are acceptable early, and refined over time.

→ **Overall**, IDDFS provides a **space-efficient alternative** to BFS for finding the shortest path in unweighted, undirected maze environments. Although it may perform redundant work, its low memory footprint makes it suitable in systems with limited resources.

5.2.6 Bi-directional Search operates by simultaneously running two searches: one forward from the start node, and one backward from the goal node, with the search terminating when the two meet.

Strengths:

- **Significantly reduces search space** in undirected, unweighted graphs — instead of searching all nodes up to depth *d*, it explores roughly up to *d/2* in both directions, leading to exponential time savings in large mazes.
- Guarantees finding the **shortest path**, assuming both searches are breadth-first.
- Particularly effective in **symmetric** environments, such as regular mazes, where forward and backward searches are likely to progress evenly.

Limitations:

- Requires the **goal state to be known in advance**, which may not always be practical depending on the game design.
- Implementation complexity increases, as two frontiers must be maintained, and the **meeting point** must be correctly handled to reconstruct the full path.
- In sparse or highly asymmetric graphs (e.g., one side is more constrained than the other), one search may progress much slower, reducing efficiency gains.
- Does **not scale well to dynamic environments**, where obstacles or paths may change during search — maintaining two synchronized frontiers becomes harder.

Special considerations:

- Works best when both start and goal are **known, static, and reachable** through symmetrical paths.
- In a maze with **multiple exits** or **unknown goal locations**, this method becomes less applicable unless adapted (e.g., multiple backward searches).
- Can be combined with other strategies (e.g., heuristic-based A*) to further optimize search direction and reduce node expansion.

→ **Overall**, Bi-directional Search is highly effective for shortest-pathfinding in large, undirected, unweighted mazes **when the goal is clearly defined and static**. It outperforms traditional BFS in terms of time complexity but comes with higher implementation and management costs.

5.2.7 Beam Search is a heuristic search algorithm that expands only a fixed number of the most promising nodes (defined by a beam width) at each level. It can be viewed as a memory-efficient variant of best-first search.

Strengths:

- **Reduces memory and computation cost** by limiting the number of nodes explored at each level.
- Can be effective in **large state spaces** where full BFS or A* would be too expensive.
- In maze-like environments, it provides **faster decision-making**, which may be suitable for time-constrained or real-time applications.

Limitations:

- Beam Search is **not optimal** — it may miss the shortest path due to early pruning of promising branches.
- Heavily **depends on the quality of the heuristic**; in an unweighted graph, without a strong guiding heuristic, its performance becomes unstable or even poor.
- The fixed beam width can lead to failure if the correct path lies just outside the top-*k* candidates in any level.
- In an **undirected, unweighted maze**, where all directions are equally valid in terms of cost, Beam Search may **prune optimal paths** due to superficial heuristic values.

Special considerations:

- Most effective in **well-structured environments** with strong heuristic signals and low branching factors.
- Beam width tuning is critical: **too narrow** leads to missed paths; **too wide** reduces its advantages over A* or BFS.
- May be more appropriate in **AI behavior modeling** or **approximate pathfinding**, rather than precise navigation in uniform-cost mazes.

→ **Overall**, Beam Search trades accuracy for speed and resource efficiency. In unweighted, undirected maze problems, its use should be justified by specific performance needs or real-time constraints, as it cannot guarantee optimality and may perform inconsistently without a well-designed heuristic.

5.2.8 Iterative Deepening A* (IDA*) is a heuristic search algorithm that applies **depth-first search with a cost-bound** derived from the A* evaluation function 𝑓. It iteratively increases this cost threshold until the goal is found.

Strengths:

- **Combines the memory efficiency of DFS** with the heuristic-guided optimality of A*, making it suitable for large graphs where traditional A* would consume too much memory.
- **Guaranteed to find the shortest path**, provided the heuristic is admissible.
- Performs well in **deep search spaces** where paths are long but memory is limited.

Limitations:

- In **unweighted graphs**, like undirected mazes, IDA* offers **no advantage over BFS or IDDFS** unless a highly effective heuristic is available.
- Suffers from **repeated state expansions** due to its iterative nature — each threshold increase may revisit the same nodes multiple times.
- Heavily **dependent on heuristic quality**; a poor heuristic leads to unnecessary iterations and degraded performance.
- More complex to implement and debug, especially in undirected graphs with cycles, where careful handling of visited states is required per iteration.

Special considerations:

- In a maze represented as an undirected, unweighted graph, with no significant heuristic advantage, IDA* may become **slower than A*** due to its repeated work and lack of weight differentiation.
- Useful when the **goal is far** and memory is constrained, assuming the heuristic provides good guidance.
- In practice, IDA* performs best in **tree-like or acyclic structures**, where state repetition is minimal.

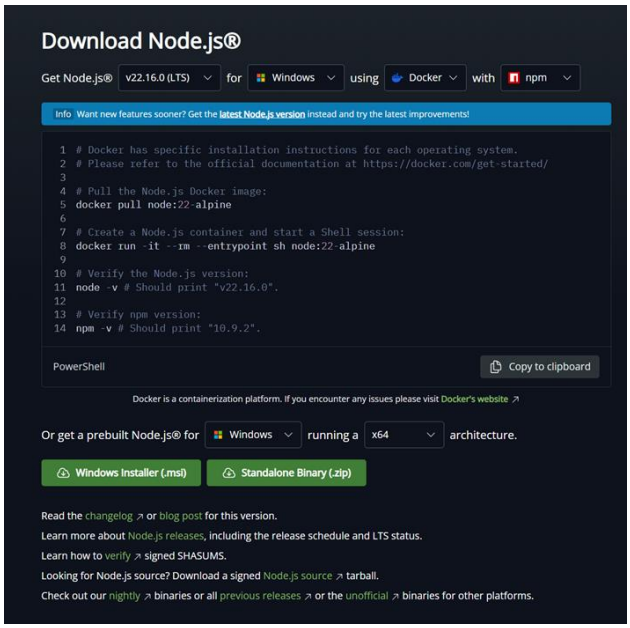
→ **Overall**, IDA* is a strong candidate when both **memory usage and optimality** are critical, and a **reliable heuristic** is available. However, in the context of unweighted, undirected mazes, its benefits over BFS, IDDFS, or even A* are limited and may not justify its overhead.

6 Program Instructions:

6.1 Accessing the Website

- Method 1: Set up the environment
  - Install NodeJS: go to Node.js — Download Node.js®, select **Windows Installer (.msi)**, then install Node.js. (Type node --version in the terminal. If the version appears, Node.js

has been successfully installed.)



- **Install Yarn:** Run the command npm install --global yarn
- Navigate to the project folder: Type cd Web/client and install all necessary libraries using yarn install
- Then start the development server by running yarn dev
  After that, open your browser and go to http://localhost:3000 to access the running web application.

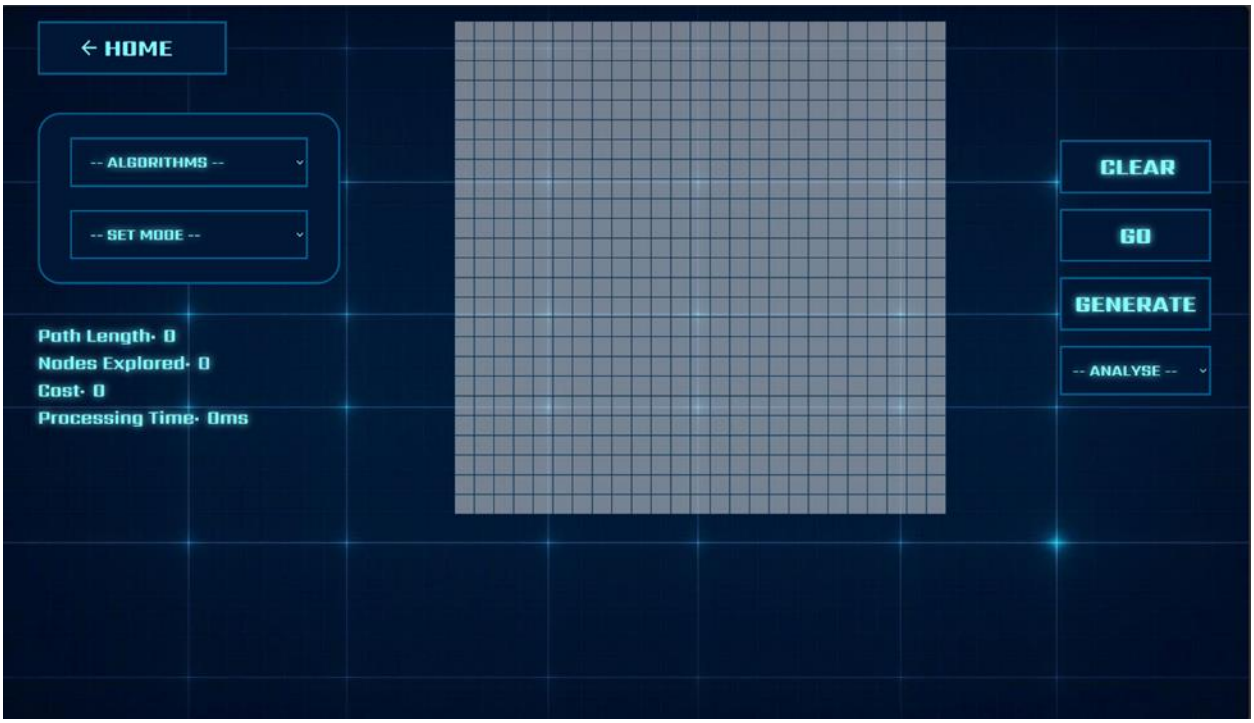
- Method 2: Visit Maze Game to access the pre-deployed website.

6.2 Website Usage Guide

- Startup Screen

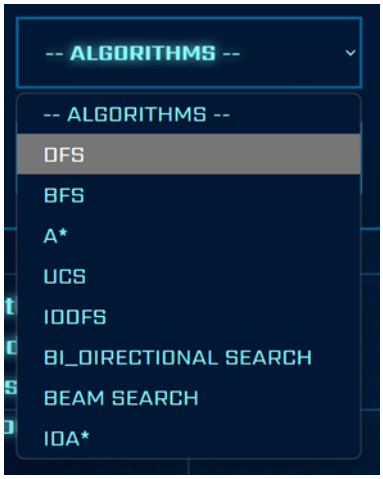
- The **Start button** launches the game
- The **Team button** displays information about the members of the group

- When the game starts

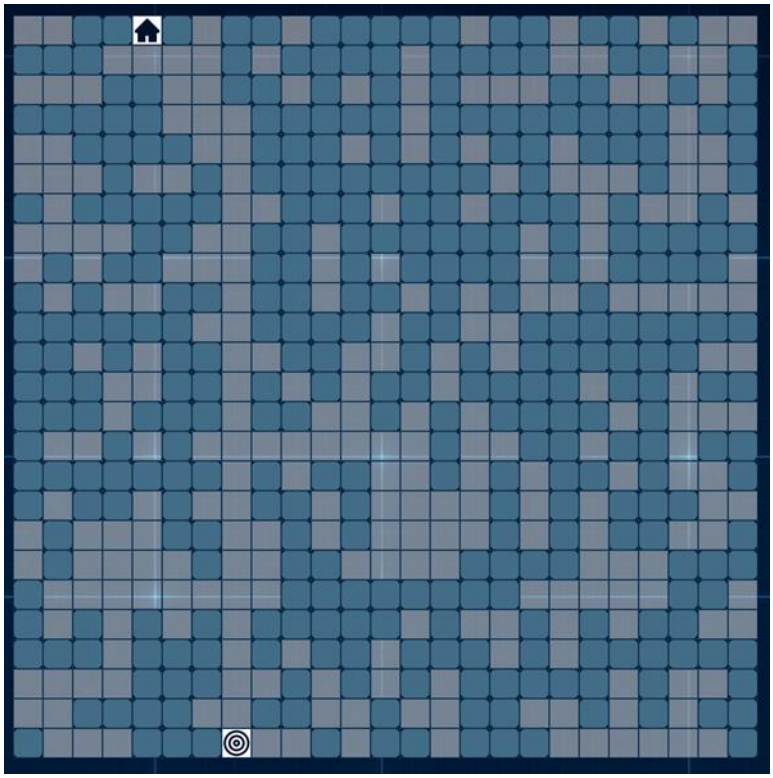


The interface at the beginning of game

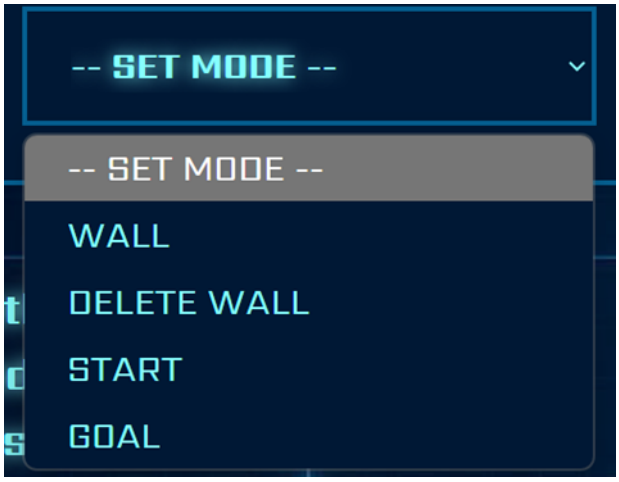
- Step 1: Choose Algorithms



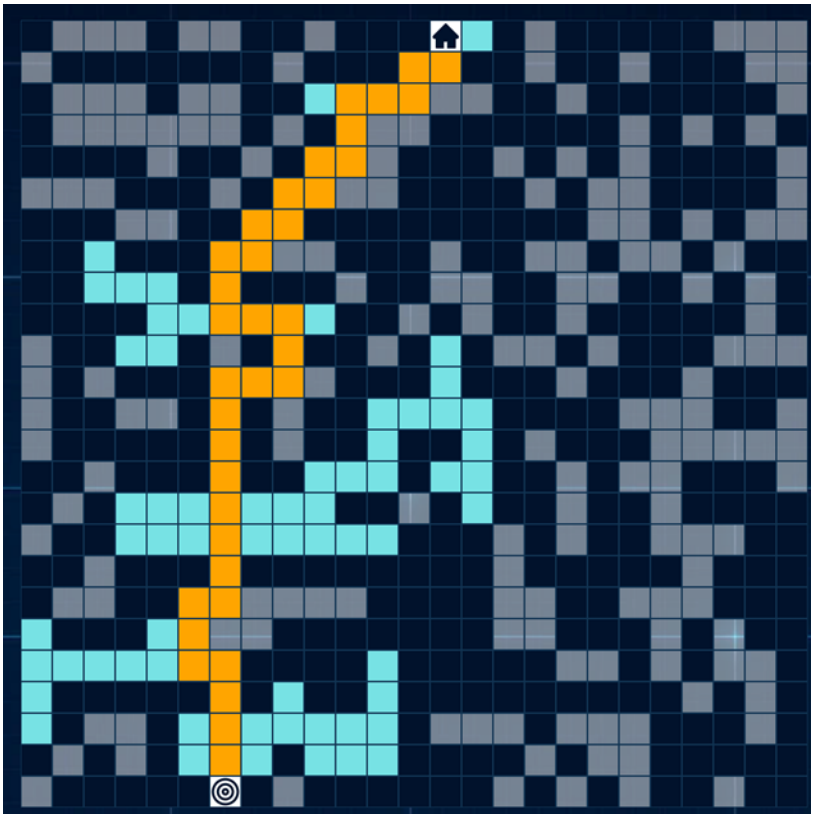
- Step 2: User can choose one of the following two options (depending on users' s preference)
  - Click the **Generate** button to create a Maze. The **Start** and **Goal** positions will be selected randomly.



- Go to **Set Mode** to manually choose the **Start** and **Goal** positions. (Additionally, you can draw or delete **Wall** as desired.)

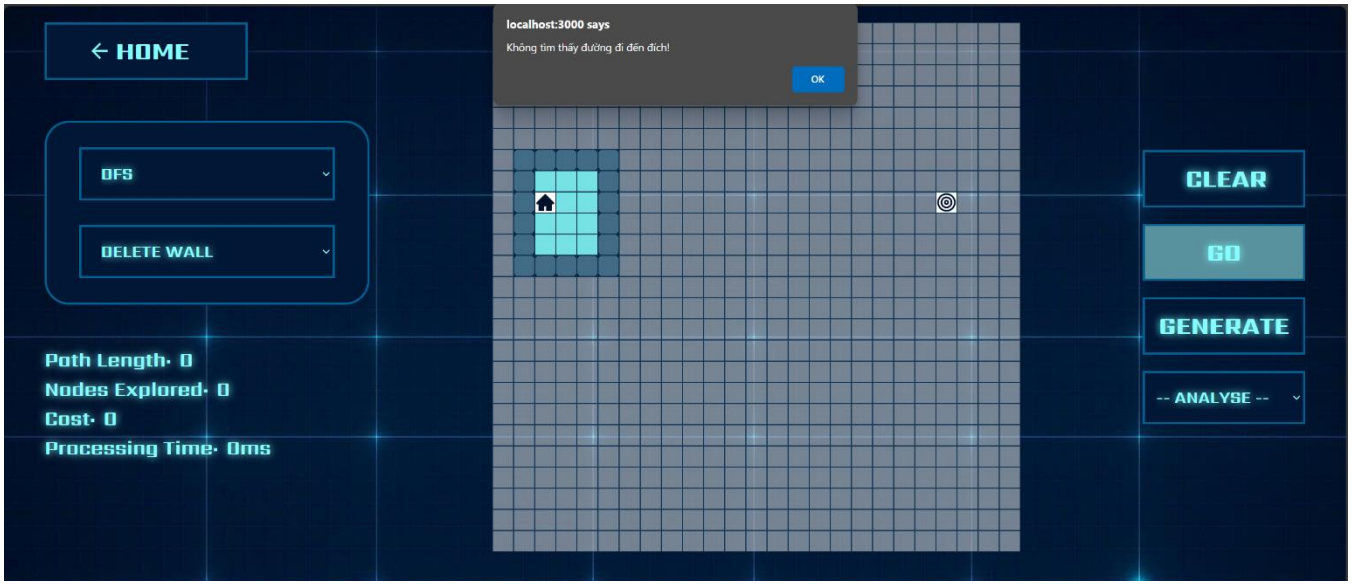


- Step 3: Click **Go** to run the algorithm selected by the user

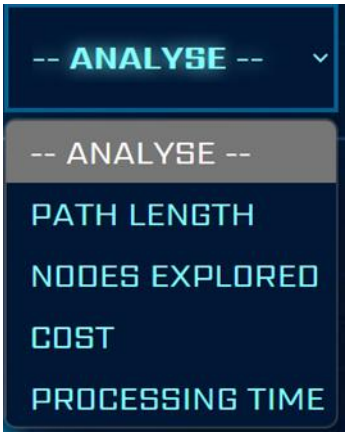


Running DFS algorithm

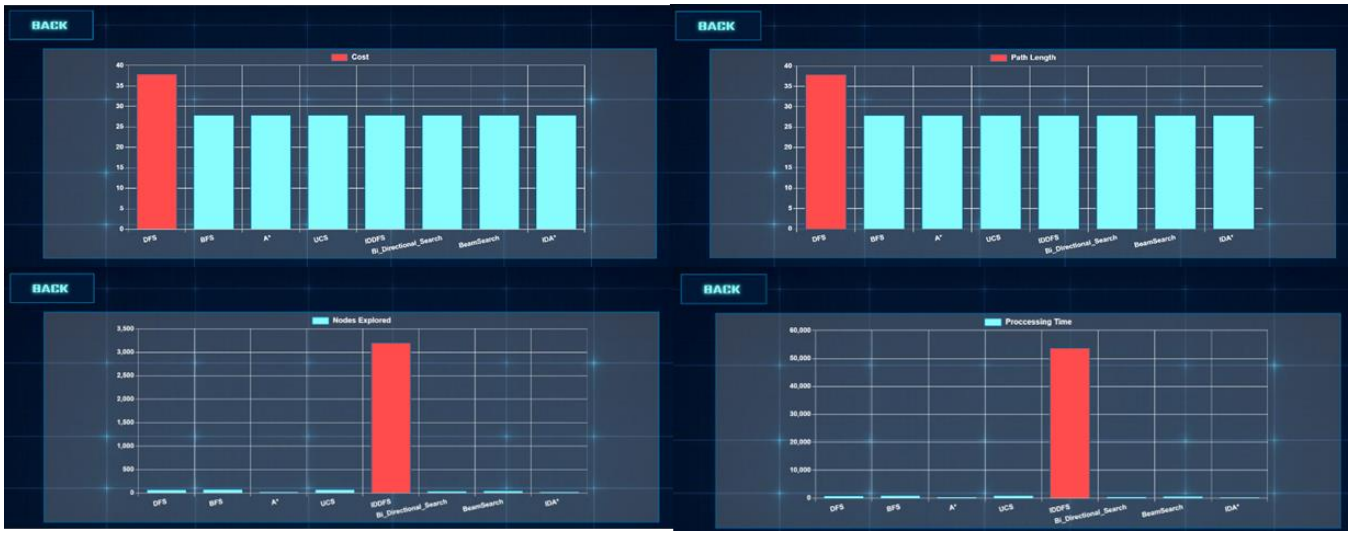
- When a valid path from **Start** to **Goal** is found, the following values will be measured: **Path length**, **Nodes explored**, **Cost**, and **Processing time** of the algorithm.
- If no path can be found, a notification will be displayed



→ After running different algorithms, you can view a **comparison section** that shows the performance metrics of the algorithms you have executed.



Example: A chart is displayed comparing the values when running 8 algorithms on the same maze. The **red column** represents the algorithm that consumes the most resources.


BACK
Cost
DFS
BFS
A*
UCS
IDDFS
Bi_Directional_Search
BeamSearch
IDA*
BACK
Path Length
DFS
BFS
A*
UCS
IDDFS
Bi_Directional_Search
BeamSearch
IDA*
BACK
Nodes Explored
DFS
BFS
A*
UCS
IDDFS
Bi_Directional_Search
BeamSearch
IDA*
BACK
Proccessing Time
DFS
BFS
A*
UCS
IDDFS
Bi_Directional_Search
BeamSearch
IDA*

7 Limitations and Future Work:

7. 1 Limitations

- **Infinite Loop Risk:** Algorithms like DFS, IDDFS, Beam Search, and IDA* may run indefinitely on complex mazes due to the absence of timeout or function-based protection.
- **Lack of Touch/Mobile Support:** The application relies on mouse interactions, hindering usability on touch devices.
- **No Save/Share Functionality:** Users cannot save maze configurations or share them via links.
- **Insufficient Algorithm Validation:** Missing unit tests in algorithm.helper.js prevent thorough verification of search algorithm correctness.
- It should be noted that certain limitations of the algorithm are not fully revealed within the current maze environment, as the graph is undirected and unweighted; this structure masks performance issues that would otherwise emerge in weighted or directed contexts, and may overstate the algorithm's efficiency or applicability.

7.2 Future Work

- Implement a timeout, cancellation token, or custom flag to prevent freezing in unsolvable mazes.
- Implement Undo / Redo button to undo/redo when user sets the wrong start/goal. Provide an option to export/import maze state (JSON or encoded URL)
- Implement `onTouchStart` and `onTouchMove` handlers to enhance mobile usability.
- Integrate advanced algorithms, including D* Lite (for dynamic environments), Genetic Algorithms (for optimization), HPA*, and RRT (for robotics). Evaluate performance based on path length, nodes explored, memory usage, scalability, and stability.
- Customizable weighted mazes:
  - Implement weighted paths for varied tile costs.
  - Enable custom terrain drawing (e.g., mud, grass, water, lava) with corresponding movement costs.
  - Incorporate diagonal movement or hex-grid options.
- Implement testing algorithms