

Artificial Intelligence Assignment

Course:

Artificial Intelligence (CS-502)

Submitted to:

Mr. Muhammad Yasir Khan

Submitted by:

Muhammad Sadam Muneer

Roll No:

2022-uam-1972

Program:

BSCS - 6th Semester (Section A)

Institution:

MNS University of Agriculture, Multan

AI Assignment

What I Did

In this assignment, I created a program that solves three different mazes using four well-known search algorithms. Each maze is like a puzzle made of walls (#) and open paths, where the goal is to go from point A (start) to point B (destination) without hitting any walls.

To find the path, I used:

- BFS (Breadth-First Search)
- DFS (Depth-First Search)
- Greedy Search
- A* Search (A-Star)

For each maze, the program checks how each algorithm performs and how fast it finds the solution, how many steps it takes, and which one is the best among all.

Why I Did This & Purpose

The main reason for doing this task was to understand how different search algorithms behave in solving maze problems. By testing them on multiple mazes (easy and complex), I could clearly see the difference in speed, accuracy, and efficiency.

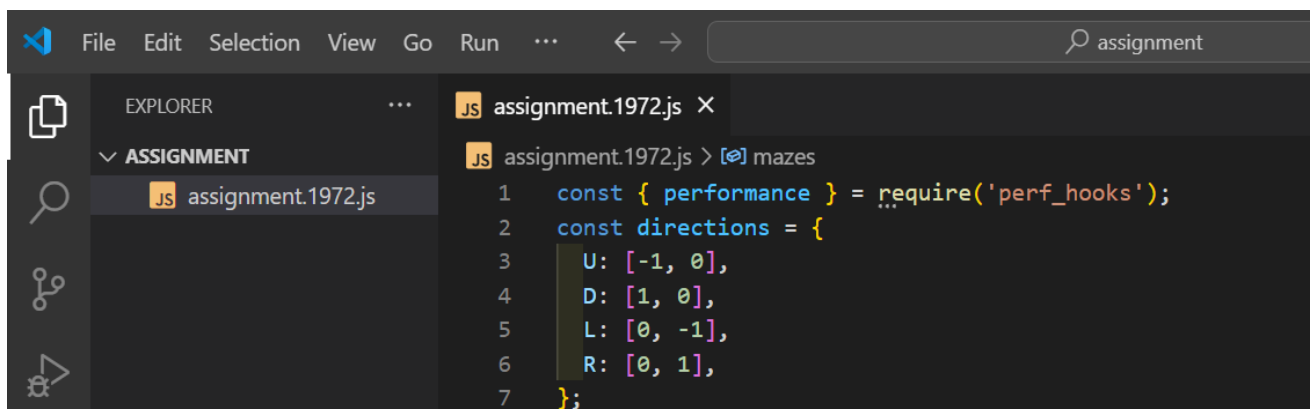
The purpose of the assignment was to give practical experience with search and pathfinding algorithms, which are very useful in fields like game development, robotics, AI, and maps navigation. It helped me learn how to write smarter code that makes decisions based on a situation.

Which Language I Used and Why

I chose **JavaScript (Node.js)** for this assignment because:

- It's beginner-friendly and good for logic-based tasks.
- Node.js runs the code fast and gives access to tools like `performance.now()` to measure speed.
- I'm already comfortable with JavaScript, which made development and testing easier.
- It's perfect for console-based programs like this, where visual design isn't the main focus.

Code Explanation With Screenshots

The screenshot shows a code editor interface. On the left, the 'EXPLORER' sidebar is open, showing a folder named 'ASSIGNMENT' with a file 'assignment.1972.js' inside. The main editor area displays the code for 'assignment.1972.js'. The code starts with a comment 'mazes' and then imports the 'performance' module using 'require'. It then defines a 'directions' object with four properties: 'U' (Up) with value [-1, 0], 'D' (Down) with value [1, 0], 'L' (Left) with value [0, -1], and 'R' (Right) with value [0, 1].

```
1 const { performance } = require('perf_hooks');
2 const directions = {
3   U: [-1, 0],
4   D: [1, 0],
5   L: [0, -1],
6   R: [0, 1],
7 };
```

In the file `assignment.1972.js`, the code imports `performance` to measure how long the algorithms take to run. It also defines the `directions` object with possible moves in the maze: Up (U), Down (D), Left (L), and Right (R), where each direction changes the row and column in the grid.

Maze Grids Definition

```

8  const mazes = [
9
10  #####B#
11  ##### #
12  ##### #
13  ##### ##
14  ##### ##
15  A#####
16  ~,
17
18  ###                      #####
19  # ##### # #
20  # ##### # # # #
21  # ##### # # # #
22  # ##### # # # #
23  ##### # # # #
24  # ## # # # #
25  # # ## ## ## ##### # # #
26  # # # # ##B# # # #
27  # # ## ##### # # #
28  ### ## ##### # # #
29  ### ##### ## # # # #
30  ### ## # # # #
31  ##### ##### ##### # # #
32  ##### ##### # #
33  A #####
34  ~,
35
36  ## #
37  ## ## #
38  #B # #
39  # ## ##
40  ##### ##
41  A#####
42  ~
43  ];

```

The mazes array holds three maze grids, with # indicating walls, spaces representing paths, A as the start point, and B as the goal. These grids are used to test the pathfinding algorithms.

Loading the Maze and Heuristic Calculation

```
44 function loadMaze(mazeStr) {  
45   const maze = mazeStr.trim().split('\n').map(r => r.split(''));  
46   let start, goal;  
47   maze.forEach((row, r) => row.forEach((cell, c) => {  
48     if (cell === 'A') start = { row: r, col: c };  
49     if (cell === 'B') goal = { row: r, col: c };  
50   }));  
51   return { maze, start, goal };  
52 }  
53 function heuristic(pos, goal) {  
54   return Math.abs(pos.row - goal.row) + Math.abs(pos.col - goal.col);  
55 }
```

loadMaze(mazeStr):

This function takes a maze as a string, processes it, and returns the maze in a more usable format. It:

- Splits the maze into rows and columns.
- Finds the starting point (A) and the goal (B) by looking for their positions.
- Returns the maze, along with the positions of A and B.

heuristic(pos, goal):

This function calculates the estimated distance between the current position and the goal. It uses the Manhattan distance formula, which adds the difference in rows and columns between the two points. This helps in finding the shortest path in pathfinding algorithms like A* or Greedy Search.

Pathfinding Search Function

```

56 function search(maze, start, goal, method = 'bfs') {
57   const queue = [{ pos: start, path: [], cost: 0 }];
58   const visited = new Set();
59
60   while (queue.length) {
61     if (method === 'greedy' || method === 'astar') {
62       queue.sort((a, b) => {
63         const fA = heuristic(a.pos, goal) + (method === 'astar' ? a.cost : 0);
64         const fB = heuristic(b.pos, goal) + (method === 'astar' ? b.cost : 0);
65         return fA - fB;
66       });
67     }
68     const current = method === 'dfs' ? queue.pop() : queue.shift();
69     const key = `${current.pos.row},${current.pos.col}`;
70     if (visited.has(key)) continue;
71     visited.add(key);
72
73     if (current.pos.row === goal.row && current.pos.col === goal.col)
74       return current.path;
75
76     for (const [dir, [dr, dc]] of Object.entries(directions)) {
77       const nr = current.pos.row + dr, nc = current.pos.col + dc;
78       if (nr >= 0 && nr < maze.length && nc >= 0 && nc < maze[0].length &&
79         maze[nr][nc] !== '#' && !visited.has(`${nr},${nc}`)) {
80         queue.push({
81           pos: { row: nr, col: nc },
82           path: [...current.path, dir],
83           cost: current.cost + 1,
84         });
85       }
86     }
87   }
88   return null;
89 }

```

The search function is used to find the shortest path from the start point (A) to the goal (B) in a maze using different pathfinding methods. It supports methods like **BFS (Breadth-First Search)**, **DFS (Depth-First Search)**, **Greedy Search**, and **A Search***

- It uses a queue to explore positions in the maze, starting from the start point.
- Depending on the search method, the queue is sorted differently (for Greedy and A* methods, it's sorted by distance).
- The function explores neighboring cells in all four directions (up, down, left, right), avoids revisiting cells, and checks if the goal is reached.
- If the goal is found, it returns the path; if no path is found, it returns null.

Performance Measurement Function

```
90 function runAndMeasure(name, fn) {
91   const start = performance.now();
92   const path = fn();
93   const end = performance.now();
94   return {
95     name,
96     path,
97     steps: path?.length || 0,
98     time: (end - start).toFixed(2),
99   };
100 }
```

The runAndMeasure function tracks the time it takes to execute a pathfinding algorithm. It calculates the time before and after the function runs, then returns the algorithm name, path, number of steps, and time taken in milliseconds.

Main Function to Run Algorithms

```
101 function main() {
102   mazes.forEach((mazeStr, i) => {
103     const { maze, start, goal } = loadMaze(mazeStr);
104     console.log(`\n===== Maze ${i + 1} =====`);
105
106     const results = [
107       runAndMeasure("BFS", () => search(maze, start, goal, 'bfs')),
108       runAndMeasure("DFS", () => search(maze, start, goal, 'dfs')),
109       runAndMeasure("Greedy Search", () => search(maze, start, goal, 'greedy')),
110       runAndMeasure("A* Search", () => search(maze, start, goal, 'astar')),
111     ];
112
113     results.forEach(({ name, path, steps, time }) => {
114       console.log(`\n${name} -> Steps: ${steps}, Time: ${time}ms`);
115       console.log(`Path: ${path ? path.join(' -> ') : "No path found"}`);
116     });
117
118     const best = results.reduce((a, b) => a.steps === b.steps ? (a.time < b.time ? a : b) : (a.steps < b.steps ? a : b));
119     console.log(`\n🏆 Best Algorithm: ${best.name} (Steps: ${best.steps}, Time: ${best.time}ms)`);
120   });
121 }
122
123 main();
```

The main function runs the four search algorithms (BFS, DFS, Greedy, A*) on each maze. For each maze, it measures the steps and time taken by each algorithm. It then displays the results, including the best-performing algorithm (based on the least steps and time). The best algorithm is chosen by comparing the steps and time of all the results.

➤ **Maze 3 Search Algorithms Result**

```
===== Maze 3 =====
```

```
BFS -> Steps: 4, Time: 0.10ms
```

```
Path: U -> R -> U -> U
```

```
DFS -> Steps: 16, Time: 0.18ms
```

```
Path: U -> R -> R -> R -> R -> U -> U -> R -> U -> U -> L -> L -> L -> D -> D -> L
```

```
Greedy Search -> Steps: 4, Time: 0.06ms
```

```
Path: U -> R -> U -> U
```

```
A* Search -> Steps: 4, Time: 0.05ms
```

```
Path: U -> R -> U -> U
```

```
💀 Best Algorithm: A* Search (Steps: 4, Time: 0.05ms)
```

Conclusion:

In this assignment, I used four search algorithms (BFS, DFS, Greedy, and A*) to find the shortest path in three different mazes. All algorithms successfully found the path, but A* Search performed the best, with the fewest steps and the fastest time. This assignment helped me understand how these algorithms work and how to compare their efficiency in pathfinding tasks.