Pathfinding Algorithm Practice

A* algorithm practice: try to solve the two problems introduced in the following, example solution can be found from the Github repo: https://github.com/Enigma-li/ILP24-25_practice.

Task One - Solving the Shortest Path in a Maze

Problem Description:

Imagine you have a 2D maze represented as a grid, where some cells are walls (impassable), and others are open paths. You need to find the shortest path from a start cell to a goal cell while avoiding the walls.

Step-by-Step Tutorial:

1. Maze Representation:

Start by representing the maze as a 2D grid, where each cell can be either:

- Open path (denoted as '.')
- Wall (denoted as '#')
- Start point (denoted as 'S')
- Goal point (denoted as 'G')

2. A* Algorithm Implementation:

Implement the A* algorithm: this algorithm requires the following components:

- Priority Queue (Min Heap) for open nodes
- Closed list to track explored nodes
- Evaluation function (f-cost) for each cell

3. Pseudocode:

Provide a pseudocode for the A* algorithm. Here's a simple example:

```
function AStar(maze, start, goal):
    openSet = {start}
```

```
cameFrom = {}
gScore = {cell: infinity for cell in maze}
gScore[start] = 0
fScore = {cell: infinity for cell in maze}
fScore[start] = heuristic(start, goal)

while openSet is not empty:
    current = cell in openSet with lowest fScore
    if current == goal:
        return reconstructPath(cameFrom, current)
    openSet.remove(current)
    for neighbor in getNeighbors(current):
        tentativeGScore = gScore[current] + distance(current, neighbor)
        if tentativeGScore < gScore[neighbor]:
            cameFrom[neighbor] = current
            gScore[neighbor] = tentativeGScore
            fScore[neighbor] = gScore[neighbor] + heuristic(neighbor, goal)
            if neighbor not in openSet:
                openSet.add(neighbor)
return "No path found"
```

4. Heuristic Function:

Common heuristics include Manhattan distance, Euclidean distance, and diagonal distance.

5. Visualization:

Use a graphical tool or code to visualize the A* algorithm as it progresses through the maze. **For Jave, just print the results maze**.

6. Implementation:

Implement the A* algorithm to find the shortest path from 'S' to 'G' in a given maze. Pseudocode can used as a guideline.

7. Testing:

Provide different maze scenarios to test the A* algorithm implementation. Ensure that you consider edge cases, such as mazes with no solution or mazes with multiple paths.

8. Performance Analysis:

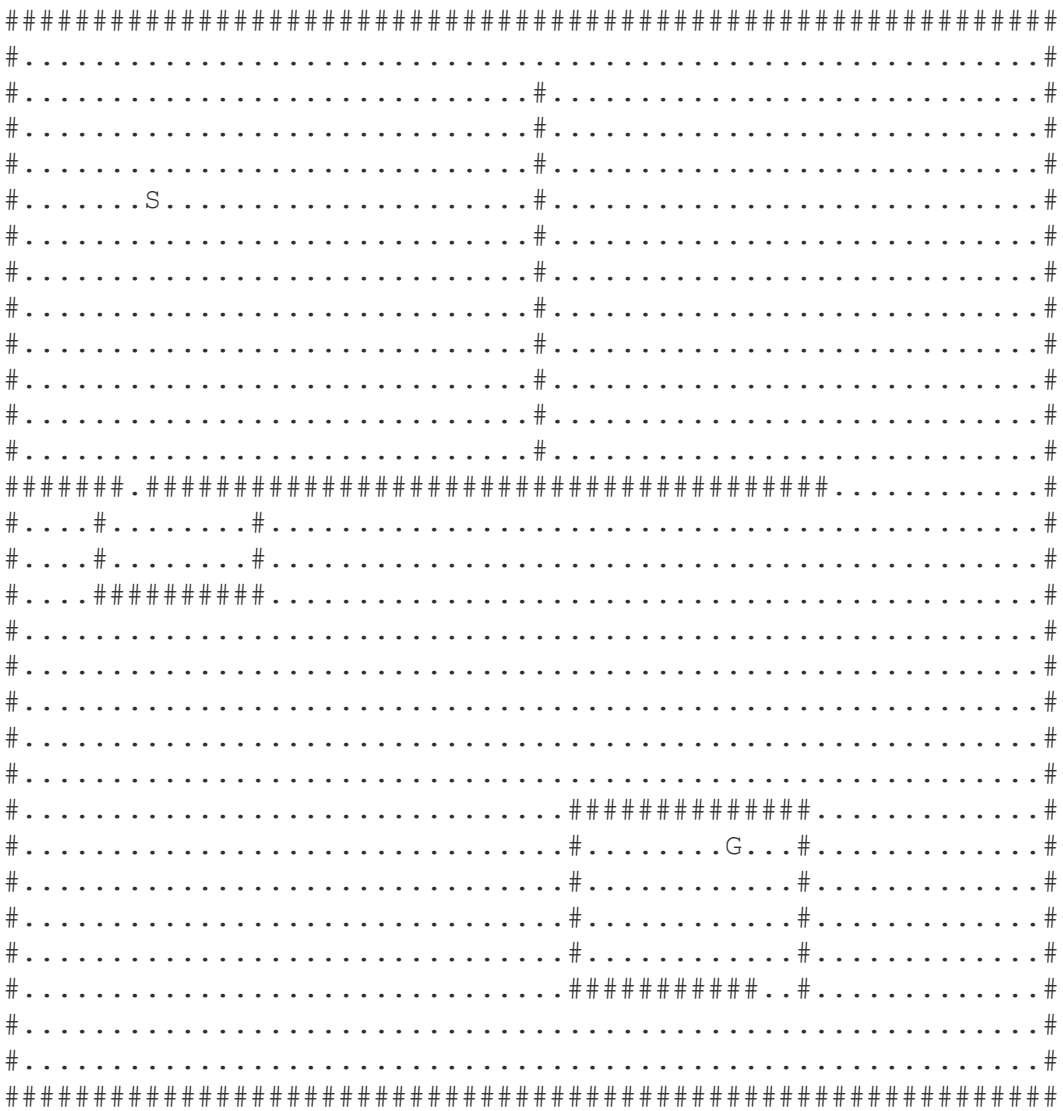
Discuss the time and space complexity of the A* algorithm, as well as its advantages and limitations.

Sample Code: download from GitHub.

- *Task*: implement the find shortest path function

Example Input:

Map:



Task Two (Extension) - Solving the 15-puzzle problem

The 15-puzzle, also known as the "Game of Fifteen," is a sliding puzzle that consists of a 4x4 grid with 15 numbered tiles and one empty space. The objective of the game is to rearrange the tiles by sliding them into the empty space to achieve a specific goal configuration, typically with the tiles arranged in ascending order, starting from the top-left corner.

The rules for the 15-puzzle are as follows:

- You can only slide a tile into the empty space, and you can't move tiles diagonally.
- The puzzle is typically solved when the tiles are arranged in ascending order from left to right, top to bottom, with the empty space in the bottom-right corner.
- The tiles can be shuffled into various configurations, making it a challenging puzzle.

The 15-puzzle is a classic example of a sliding puzzle and is often used as a recreational puzzle game and a simple problem for teaching algorithms like search and heuristic methods, such as the A* algorithm, to find an optimal solution. Solving the 15-puzzle can be done through a sequence of moves that minimize a specific cost function or heuristic.

Example:

Start Status:	Goal Status:
5 1 2 4	1 2 3 4
9 6 3 8	5 6 7 8
13 15 10 11	9 10 11 12
14 0 7 12	13 14 15 0

General Tips:

You can treat the status after one movement as a grid position, from the start status, you can move tiles to achieve the goal status.

g function: calculate the moving cost from the start status to the current status

h function: calculate the number of misplaced tiles

Sample Code: download from GitHub.

- *Task*: implement the find shortest path function.