Artificial Intelligence CSCI-6511

Project Report

For Project 1

Instructor: Assoc. Prof. Amrinder Arora

Student: Emil Naghizade

Group Number: CSDA26

Date: 18/02/2025

Introduction

In AI and pathfinding problems, the task of navigating from one point to another in a given grid or maze is often a critical one. The goal is to find the most optimal path from a starting point to a destination, avoiding obstacles along the way. In this report, we explore how the A-star search algorithm (A*) can be applied to solve pathfinding problems in a grid-based maze. The algorithm is designed to find the shortest path while considering both the distance traveled so far and an estimate of the distance to the goal. It efficiently finds a path between two points on the grid using a combination of greedy best-first search and Dijkstra's algorithm.

Problem Statement

You are given a separate maze file in text format. In this maze, you can go up, down, left, or right. The maze is 81*81 binary matrix where each line in the file represents a row, and its values are separated with space. 1 indicates a block that you cannot pass. 0 indicates a clear space that you can pass from.

You should implement a program using Informed Search that reads this maze and takes any two points (start and end indices) as inputs and tells whether there is a path in this maze between such points. Please note that the index starts at 0 in the following points. For example, given the start point (1,1) and the end point (1, 8), your program outputs 'YES' if there exists at least one path between those two points and 'NO' if no path exists.

Explanation of A-star algorithm and Heuristic

The A-star search algorithm is a pathfinding and graph traversal algorithm used in AI. It finds the shortest path from a starting point to a goal point on a grid by combining two key components:

1. g(n) – The cost of the path from the start node to the current node ***n***
2. h(n) – A heuristic estimate of the cost from node ***n*** to the goal node

The total cost function is calculated as follows:

$$f(n) = g(n) + h(n)$$

The algorithm checks nodes with the lowest f(n) value first, ensuring the most promising paths are explored early.

The heuristic function is essential in directing the search process. For grid-based mazes, a commonly used heuristic is the Manhattan Distance, which is defined as follows:

$$h(n) = \left|x_{start} - x_{goal}\right| + \left|y_{start} - y_{goal}\right|$$

Here, x and y are the coordinates of the start and goal positions, respectively. This function estimates the minimum number of steps needed to reach the goal from the current position, considering only movements in horizontal and vertical directions.

Code Explanation

Below is the explanation of the core sections of the Python code for the A-star search algorithm applied to the maze.

1. **Reading the maze** – The maze is read from a text file into a 2D list using the function read_maze(file_path). Each row of the maze is split by spaces, and the values are mapped into integers (0 for passable, 1 for blocked).

```
3  # Function to read the maze file
4  def read_maze(file_name):
5      with open(file_name, 'r') as f:
6          return [list(map(int, line.strip().split())) for line in f]
```

2. **A-star search function –** We initialize the grid size (rows, cols) and define possible movements (up, down, left, right).

```
 8  # A* Search Algorithm
 9  def astar_search(maze, start, end):
10      rows, cols = len(maze), len(maze[0])
11      directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]  # Up, Down, Left, Right
```

3. **Heuristic –** The Manhattan Distance heuristic calculates the cost to reach the goal from the current node.

```
13      def heuristic(a, b):
14          """Manhattan distance heuristic"""
15          return abs(a[0] - b[0]) + abs(a[1] - b[1])
```

4. **Priority Queue –** The priority queue stores nodes prioritized by their f-cost. The visited set avoids revisiting the nodes, and g-cost tracks the shortest known path to each node.

```
17      # Priority queue: stores (f-cost, g-cost, (row, col))
18      pq = [(heuristic(start, end), 0, start)]
19      visited = set()
20      g_cost = {start: 0}
```

5. **Goal reached case –** The algorithm explores nodes with the lowest f-cost first. If the goal node is reached, it returns "YES".

```
22      while pq:
23          _, cost, (r, c) = heapq.heappop(pq)
24
25          if (r, c) == end:
26              return "YES"
```

6. **Explore other nodes –** We explore neighboring nodes in all four directions (up, down, left, right). If the new node is valid (within bounds and passable), we update its cost and add it to the priority queue.

Results

The code is tested by the start and end coordinate inputs provided by the instructor. The results are organized on the table below:

Table 1 Results

Test №	Start Point	End Point	Output
1	(1, 34)	(15, 47)	YES
2	(1, 2)	(3, 39)	YES
3	(0, 0)	(3, 77)	NO
4	(1, 75)	(8, 79)	NO
5	(1, 75)	(39, 40)	NO

```
C:\Users\emiln\OneDrive\Documents\ADA_University\Artificial_Intelligence>python Project_1.py maze1.txt 1 1 1 8
YES

C:\Users\emiln\OneDrive\Documents\ADA_University\Artificial_Intelligence>python Project_1.py maze1.txt 1 34 15 47
YES

C:\Users\emiln\OneDrive\Documents\ADA_University\Artificial_Intelligence>python Project_1.py maze1.txt 1 2 3 39
YES

C:\Users\emiln\OneDrive\Documents\ADA_University\Artificial_Intelligence>python Project_1.py maze1.txt 0 0 3 77
NO

C:\Users\emiln\OneDrive\Documents\ADA_University\Artificial_Intelligence>python Project_1.py maze1.txt 1 75 8 79
NO

C:\Users\emiln\OneDrive\Documents\ADA_University\Artificial_Intelligence>python Project_1.py maze1.txt 1 75 39 40
NO
```