Artificial Intelligence Assignment 1 – Pathfinding in Maze

Students: Murad Valiyev, Ilaha Behbudova

Date: 10 February 2026

Chosen Algorithm - A* Search Algorithm

Why A? – Explanation*

The problem asks us to determine, for several start–end coordinate pairs, whether it is possible to travel from the start cell to the end cell in a rectangular grid maze. The maze is given as a matrix of 0s and 1s, where 0 represents a passable (walkable) cell and 1 represents an impassable wall. Movement is allowed only in four directions - up, down, left, right - with no diagonal steps, and every move costs exactly 1 step.

A* was chosen as the most appropriate algorithm for the following reasons:

A* guarantees finding the **shortest path** (fewest steps) in grids with uniform movement cost when using an admissible and consistent heuristic. We use the **Manhattan distance** as the heuristic:

h(n) = |current_row − goal_row| + |current_col − goal_col|

Because only four-directional movement is allowed, this heuristic never overestimates the true remaining distance (it is admissible) and satisfies the consistency condition. Together, these properties ensure that the first time A* removes the goal from the priority queue, it has discovered a path with the minimal possible length. No shorter path can exist.

Compared to uninformed searches, A* is usually much more efficient in practice. BFS explores all directions equally and can generate a very large number of nodes when the goal is distant or the maze contains open spaces. DFS may find a path quickly in some cases but often produces long, winding, non-optimal routes and has poor worst-case performance. A* avoids both issues by using the evaluation function

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

where g(n) is the exact number of steps taken from the start to the current cell, and h(n) is the estimated steps remaining to the goal. This combination directs the search toward promising areas first, typically visiting far fewer cells than BFS in most maze configurations.

Implementation overview

The code reads the maze from a text file into a 2D list and validates that both start and end positions are inside the grid and not on walls. It then runs A* using a priority queue (implemented with Python's heapq) that orders nodes by their f-value, with a small counter added for stable tie-breaking.

A dictionary tracks the best known g-score (shortest path cost) to each visited cell. When a better path to a cell is found, its g-score is updated and a new entry is pushed to the priority queue (the standard "lazy" way to handle decrease-key in Python's heapq).

The algorithm continues after first reaching the goal. It keeps dequeuing nodes until it processes one whose f-value is at least as large as the best known path cost to the goal. At that moment it can safely stop and return YES, because no remaining path in the open set can possibly be shorter. If the queue empties without this condition being met and without reaching the goal, the answer is NO.

This combination of optimality, practical efficiency, and correct early stopping makes A* well suited for reliably answering multiple reachability queries on the same maze.

How to Run the Script

Save the code in a file named maze_solver.py.

Place the maze file p1_maze.txt in the same folder as the script. The file contains the grid as space-separated integers (0 = passable, 1 = wall).

Run the program with the command:

python3 maze_solver.py

(or python maze_solver.py on some systems)

The script automatically loads 'p1_maze.txt' from the current directory and tests five given start-end pairs. It prints the maze size and YES/NO result for each test case.