AI-Based Pathfinding with A* Algorithm

Project: **Pathfinding with A* Algorithm**

Name: **Deepak Joshi**

Roll No.[Class}: **02**

Roll No.[University}: **202401100400075**

Course: **AI for Engineers**

Date: **10 March,2025**

Introduction

Our project basically focuses on finding the best path possible to reach our goal from wherever we start to the required goal position, in which certain obstacles will also be present, and we must not pass through those positions.

Methodology

The *A algorithm** is a smart way to find the shortest and most efficient route from a starting point to a target location while avoiding obstacles. It balances two key factors:

1. How far we've already traveled (actual cost).
2. How much distance is left to the goal (estimated cost).

By continuously evaluating these two aspects, the algorithm ensures that it takes the best possible path without unnecessary detours.

Step-by-Step Breakdown

1. Understanding the Map

First, we need a grid or network where movement can take place. Some areas are walkable, while others are blocked (obstacles). The grid also has:

- A starting point (where we begin).
- A goal position (our destination).
- Paths that can be taken to navigate towards the goal**.**

2. Creating a Plan

A* follows a priority-based searching approach, where it prioritizes paths that seem most promising. It does this using a formula:

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

Where:

- g(n) → The cost to reach a point from the start.
- h(n) → The estimated remaining cost to the goal.
- f(n) → The total estimated cost (used to pick the best route).

A* always expands the path with the lowest f(n) to ensure efficiency.

3. Exploring Possible Paths

The algorithm starts at the initial position and looks at all possible moves. It calculates f(n) for each possible path and prioritizes the one with the lowest value. It avoids obstacles and backtracking whenever possible.

- If a path reaches the goal, the search stops, and the best route is reconstructed.
- If a dead-end is encountered, the algorithm backtracks and explores alternative paths**.**

4. Tracking the Best Route

Each time a better (shorter) path to a location is found, A* updates the path. It keeps track of the previous steps so that once the goal is reached, it can easily retrace the shortest path taken.

<u>Full Code Implementation</u>

```
import heapq  # Heap queue (priority queue) to always pick the lowest-cost path first.
```

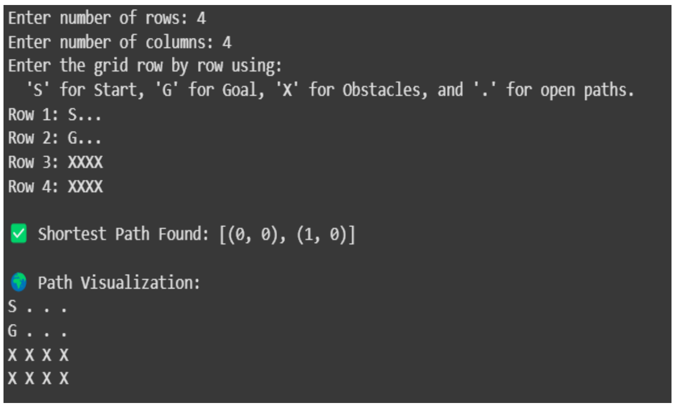
```
# Directions we can move in (right, left, down, up)

DIRECTIONS = [(0, 1), (0, -1), (1, 0), (-1, 0)]


class Node:

    """Represents a grid cell with details needed for pathfinding."""

    def __init__(self, position, g, h):

        self.position = position  # The (row, column) coordinates of this cell.

        self.g = g  # Distance from the start point.

        self.h = h  # Estimated distance to the goal (heuristic).

        self.f = g + h  # Total estimated cost of the path through this cell.

        self.parent = None  # Keeps track of how we got here (to reconstruct the path later).


    def __lt__(self, other):

        return self.f < other.f  # Ensures priority queue picks the lowest f-score.


def heuristic(a, b):

    """Estimates the shortest distance from one point to another (Manhattan Distance)."""

    return abs(a[0] - b[0]) + abs(a[1] - b[1])


def astar(grid, start, goal):

    """Finds the shortest path using the A* algorithm."""


    open_list = []  # Priority queue to explore the best paths first.

    closed_set = set()  # Keeps track of visited cells.

    node_map = {}  # Avoids duplicate nodes in the open list.
```

```
# Create the starting node and add it to the open list.
start_node = Node(start, 0, heuristic(start, goal))
heapq.heappush(open_list, start_node)
node_map[start] = start_node

while open_list:
    # Pick the node with the lowest f-score (best estimated path).
    current_node = heapq.heappop(open_list)

    # If we reached the goal, reconstruct the path and return it.
    if current_node.position == goal:
        path = []
        while current_node:
            path.append(current_node.position)
            current_node = current_node.parent
        return path[::-1]  # Return the path in correct order (start → goal).

    closed_set.add(current_node.position)  # Mark the cell as explored.

    # Explore the neighboring cells.
    for dx, dy in DIRECTIONS:
        neighbor_pos = (current_node.position[0] + dx, current_node.position[1] + dy)

        # Check if the neighbor is within the grid and not a blocked cell ('X').
        if 0 <= neighbor_pos[0] < len(grid) and 0 <= neighbor_pos[1] < len(grid[0]):
            if grid[neighbor_pos[0]][neighbor_pos[1]] == "X" or neighbor_pos in closed_set:
                continue  # Skip if it's an obstacle or already visited.
            # Calculate new cost values for the neighbor.
            g_cost = current_node.g + 1  # Moving one step costs 1.
            h_cost = heuristic(neighbor_pos, goal)  # Estimate distance to goal.
            f_cost = g_cost + h_cost  # Total estimated cost.

            # If we've seen this neighbor before, update its path if it's better.
            if neighbor_pos in node_map:
                existing_node = node_map[neighbor_pos]
                if g_cost < existing_node.g:
                    existing_node.g = g_cost
                    existing_node.f = f_cost
                    existing_node.parent = current_node  # Update path.
            else:
                # Create a new node and add it to the open list.
                neighbor_node = Node(neighbor_pos, g_cost, h_cost)
                neighbor_node.parent = current_node
                heapq.heappush(open_list, neighbor_node)
                node_map[neighbor_pos] = neighbor_node

    return None
   # No valid path found.
#Ask the user for grid size
rows = int(input("Enter number of rows: "))
cols = int(input("Enter number of columns: "))

print("Enter the grid row by row using:")
print("  'S' for Start, 'G' for Goal, 'X' for Obstacles, and '.' for open paths.")

grid = []
start = None
goal = None

# Get the grid from the user
for i in range(rows):
    while True:  # Keep asking until we get a valid row
        row = list(input(f"Row {i+1}: "))
        if len(row) == cols:
            break
        print(f"Error: Row must be exactly {cols} characters. Please re-enter row {i+1}.")

    grid.append(row)

    # Find and store the positions of 'S' (Start) and 'G' (Goal)
    if 'S' in row:
        start = (i, row.index('S'))
    if 'G' in row:
        goal = (i, row.index('G'))

# Ensure the grid has both a Start and a Goal
if start is None or goal is None:
    print("Error: The grid must contain one 'S' (Start) and one 'G' (Goal).")
    exit()
```

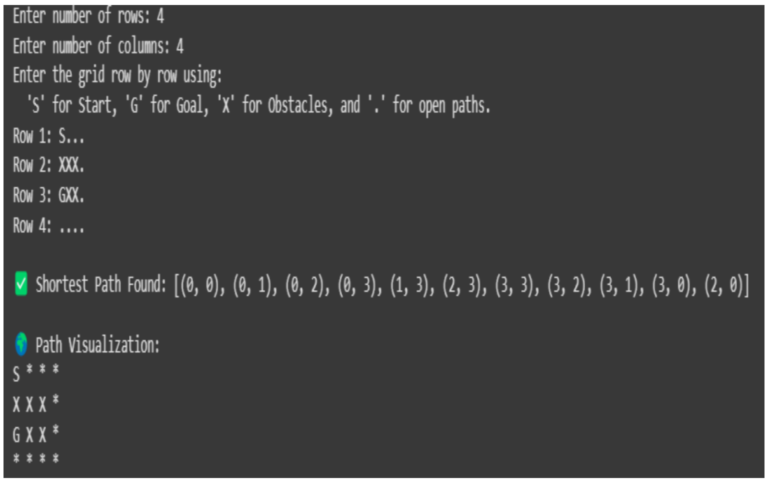
```
# Run A* and print the results
path = astar(grid, start, goal)

if path:
    print("\n✅ Shortest Path Found:", path)

    # Replace path cells with '*' for visualization
    for x, y in path:
        if grid[x][y] not in ('S', 'G'):
            grid[x][y] = '*'

    # Display the grid with the path
    print("\n🌍 Path Visualization:")
    for row in grid:
        print(" ".join(row))
else:
    print("\n❌ No path found.")
```

Output/Result

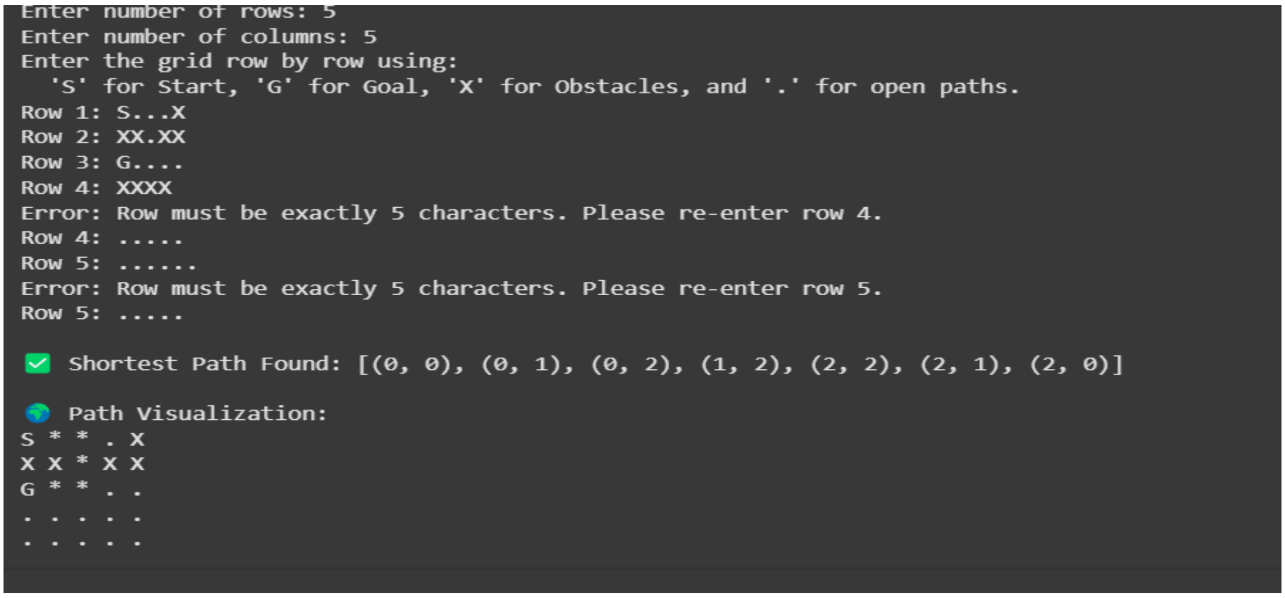
Below is a sample game output:



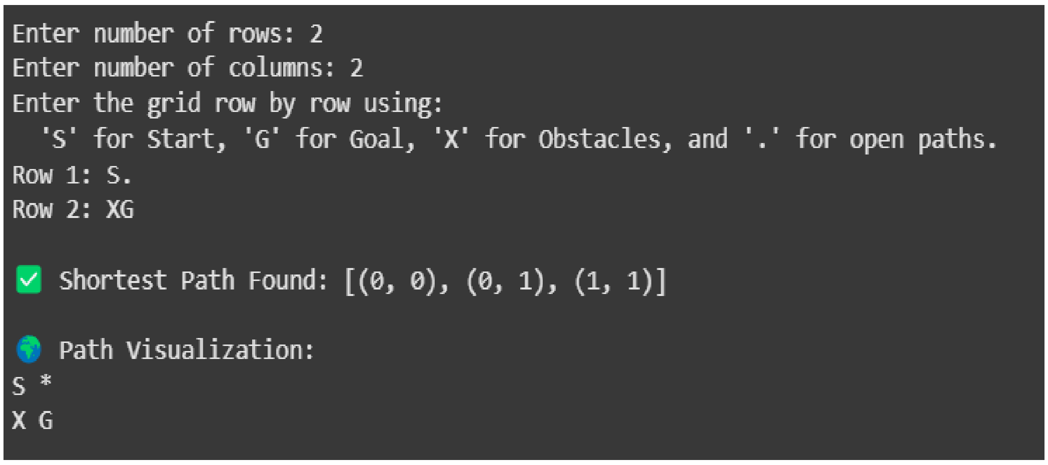
```
Enter number of rows: 4
Enter number of columns: 4
Enter the grid row by row using:
  'S' for Start, 'G' for Goal, 'X' for Obstacles, and '.' for open paths.
Row 1: S...
Row 2: G...
Row 3: XXXX
Row 4: XXXX

✅ Shortest Path Found: [(0, 0), (1, 0)]

🌍 Path Visualization:
S . . .
G . . .
X X X X
X X X X
```



```
Enter number of rows: 4
Enter number of columns: 4
Enter the grid row by row using:
  'S' for Start, 'G' for Goal, 'X' for Obstacles, and '.' for open paths.
Row 1: S...
Row 2: XXX.
Row 3: GXX.
Row 4: ....

✅ Shortest Path Found: [(0, 0), (0, 1), (0, 2), (0, 3), (1, 3), (2, 3), (3, 3), (3, 2), (3, 1), (3, 0), (2, 0)]

🌍 Path Visualization:
S * * *
X X X *
G X X *
* * * *
```



```
Enter number of rows: 5
Enter number of columns: 5
Enter the grid row by row using:
  'S' for Start, 'G' for Goal, 'X' for Obstacles, and '.' for open paths.
Row 1: S...X
Row 2: XX.XX
Row 3: G....
Row 4: XXXX
Error: Row must be exactly 5 characters. Please re-enter row 4.
Row 4: .....
Row 5: ......
Error: Row must be exactly 5 characters. Please re-enter row 5.
Row 5: .....

✅ Shortest Path Found: [(0, 0), (0, 1), (0, 2), (1, 2), (2, 2), (2, 1), (2, 0)]

🌍 Path Visualization:
S * * . X
X X * X X
G * * . .
. . . . .
. . . . .
```



```
Enter number of rows: 2
Enter number of columns: 2
Enter the grid row by row using:
  'S' for Start, 'G' for Goal, 'X' for Obstacles, and '.' for open paths.
Row 1: S.
Row 2: XG

✅ Shortest Path Found: [(0, 0), (0, 1), (1, 1)]

🌍 Path Visualization:
S *
X G
```

References/Credits

A* ALGORITHM: Wikipedia,GeeksForGeeks,TutorialsPoint