**AI MSE REPORT**

**Problem Title:** Pathfinding with A\* Algorithm

**Name:** Priya Chauhan

**Branch***:* CSE(AIML)

**Section***:* C

**Roll Number:** 202401100400148

**Date:** 10-03-2025

**1. Introduction**

Pathfinding is a method used in artificial intelligence to find the shortest path between two points. The *A (A-Star) Algorithm*\* is one of the best pathfinding algorithms. It is commonly used in video games, robotics, and navigation systems.

A\* works by calculating two main values:

1. **G-cost:** The actual distance from the starting position to the current position.
2. **H-cost:** An estimated distance from the current position to the goal.

By combining both values, A\* quickly finds the best possible path.

**2. Methodology**

The *A algorithm*\* follows these steps:

1. **Create a grid (map):** The environment is represented as a grid where:
   * 0 means a walkable path.
   * 1 means an obstacle.
2. **Select a heuristic function:** The **Manhattan Distance** formula is used to estimate how far a point is from the goal.
3. **Use a priority queue (min-heap):** This ensures that the best possible paths are explored first.
4. **Track visited positions:** This prevents checking the same position multiple times.
5. **Find the best path:** The algorithm backtracks from the goal to reconstruct the shortest path.
6. **Display the result:** If a path exists, it is shown; otherwise, a message is displayed saying no path was found.

**3. Code Implementation**

import heapq  # Import heapq for priority queue (min-heap)

import random  # Import random to generate obstacles in the grid

# Heuristic function: Manhattan distance (used to estimate cost from a point to the goal)

def heuristic(a, b):

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

# A\* Algorithm to find the shortest path from start to goal

def astar(grid, start, goal):

    rows, cols = len(grid), len(grid[0])  # Get grid size

    # Priority queue (min-heap) for open nodes, initialized with (cost, start\_position)

    open\_list = [(0, start)]

    came\_from = {start: None}  # Dictionary to store the path (where each node came from)

    g\_cost = {start: 0}  # Dictionary to store the cost from start to each node

    while open\_list:

        \_, current = heapq.heappop(open\_list)  # Get node with the lowest cost

        # If we reach the goal, reconstruct and return the path

        if current == goal:

            path = []

            while current:

                path.append(current)

                current = came\_from[current]

            return path[::-1]  # Reverse path to get it from start to goal

        # Explore all possible movement directions (left, right, up, down)

        for dx, dy in [(-1, 0), (1, 0), (0, -1), (0, 1)]:

            neighbor = (current[0] + dx, current[1] + dy)  # Calculate new position

            # Check if the neighbor is within grid boundaries and is a walkable path (0)

            if 0 <= neighbor[0] < rows and 0 <= neighbor[1] < cols and grid[neighbor[0]][neighbor[1]] == 0:

                new\_cost = g\_cost[current] + 1  # Cost of moving to neighbor

                # If the neighbor is unvisited OR we found a cheaper path to it, update it

                if neighbor not in g\_cost or new\_cost < g\_cost[neighbor]:

                    g\_cost[neighbor] = new\_cost  # Update cost

                    priority = new\_cost + heuristic(neighbor, goal)  # Calculate priority

                    heapq.heappush(open\_list, (priority, neighbor))  # Add to open list

                    came\_from[neighbor] = current  # Track where we came from

    return None  # No path found

# Function to generate a grid with obstacles

def generate\_grid(rows, cols, obstacle\_prob=0.2):

    """Generates a grid where:

    - 0 represents a walkable path

    - 1 represents an obstacle (blocked path)

    - obstacle\_prob controls the probability of an obstacle appearing"""

    return [[0 if random.random() > obstacle\_prob else 1 for \_ in range(cols)] for \_ in range(rows)]

# Get user input for grid size

rows = int(input("Enter number of rows: "))

cols = int(input("Enter number of columns: "))

# Generate the grid

grid = generate\_grid(rows, cols)

# Display the generated grid

print("\nGenerated Grid (0 = walkable, 1 = obstacle):")

for row in grid:

    print(row)

# Get user input for start and goal positions

while True:

    start\_x = int(input(f"\nEnter start row (0 to {rows-1}): "))

    start\_y = int(input(f"Enter start column (0 to {cols-1}): "))

    goal\_x = int(input(f"Enter goal row (0 to {rows-1}): "))

    goal\_y = int(input(f"Enter goal column (0 to {cols-1}): "))

    start = (start\_x, start\_y)

    goal = (goal\_x, goal\_y)

    # Ensure start and goal positions are not on obstacles

    if grid[start\_x][start\_y] == 0 and grid[goal\_x][goal\_y] == 0:

        break  # Valid input, exit loop

    else:

        print("❌ Invalid input! Start or goal is on an obstacle. Please enter different values.")

# Run A\* algorithm to find the shortest path

path = astar(grid, start, goal)

# Output the result

if path:

    print("\n✅ Shortest Path Found:", path)  # Display the shortest path

else:

    print("\n❌ No Path Found!")  # No valid path exists’’’

**4. Output Screenshots**

A screenshot of a computer

AI-generated content may be incorrect.

**5. References & Credits**

* The *A algorithm*\* is a widely used pathfinding algorithm.
* The **Manhattan Distance** heuristic is commonly used in grid-based pathfinding.
* Python’s **heapq library** is used for efficient priority queue implementation.

Bottom of Form