Pathfinding with A* Algorithm

SUBJECT : ARTIFICIAL INTELLIGENCE

SUBMITTED BY : HARSHIT

UNIVERSITY ROLL NO : 202401100400095

INSTITUTION : KIET GROUP OF INST.

DATE : 10 MARCH 2025

1. Introduction

1. **Purpose**: The A* (A-Star) algorithm is a powerful and widely used search algorithm designed to find the shortest path from a start point to a goal point in a graph or grid, while avoiding obstacles. It is highly effective for navigation tasks in robotics, gaming, and real-world applications like GPS systems.
2. **Key Concept**: A* balances the cost of movement (g-score) and the estimated cost to the goal (heuristic h-score) to determine the most promising path. The combination, `f = g + h`, ensures both efficiency and accuracy in reaching the target.
3. **Strengths**: A* is optimal and complete, meaning it guarantees finding the shortest path if one exists, and it does so efficiently by intelligently exploring nodes with the lowest total cost.

2. Problem Statement

- **Objective**: To find the shortest and most efficient path from a given starting point to a desired goal point in a grid or graph, considering potential obstacles and constraints.

- **Challenges**: The algorithm must navigate through a grid containing traversable paths (open cells) and blocked paths (obstacles), all while minimizing the computational cost and ensuring optimality.

- **Application Context**: Useful in scenarios like GPS navigation, robot motion planning, and game development, where real-time and accurate pathfinding is crucial for achieving desired outcomes.

3.Algorithm approach

- **Initialization**:
  - Start by defining the open set (nodes to be evaluated) and closed set (evaluated nodes).
  - Assign g-score (cost from start) and f-score (estimated total cost = g + h) for the starting node. Initialize all other nodes with infinite scores.
- **Exploration**:
  - Select the node with the lowest f-score from the open set (priority queue).
  - Check if this node is the goal. If yes, reconstruct and return the path; otherwise, move the node to the closed set.

- Identify all neighbors of the current node, skipping invalid or blocked ones.

 Update Scores:

- For each valid neighbor:
  - Calculate the tentative g-score (cost to the neighbor through the current node).
  - If this score is better than the previously recorded one or the neighbor hasn't been visited, update its scores (g, f) and mark the current node as its predecessor.
- Add the neighbor to the open set if not already present.

 Termination:

- Continue until the open set is empty or the goal is reached.
- If the open set is empty and the goal hasn't been found, conclude that no path exists.

4.Complexity analysis

 Time Complexity:

- In the worst case, A* may need to explore all nodes in the grid or graph, leading to a time complexity of **O(V + E)** where:
  - **V** is the number of vertices (nodes).
  - **E** is the number of edges (connections between nodes).
- Using a priority queue (e.g., a min-heap), the operation of extracting the minimum and updating scores takes **O(log V)** for each node. If the heuristic function is efficient, the performance can significantly improve by focusing only on the most promising nodes.

 Space Complexity:

- A* requires storing:
  - The open set (priority queue): **O(V)**.
  - The closed set (explored nodes): **O(V)**.
  - Scores like g-score and f-score: **O(V)**.
- Total space complexity is **O(V)**, as it depends on the size of the graph or grid.

 Optimality:

- A* is guaranteed to find the shortest path if the heuristic used is **admissible** (never overestimates the cost to reach the goal) and **consistent** (obeys the triangle inequality). The time complexity is heavily influenced by the quality of the heuristic.

- **Efficiency in Practice**:

- The algorithm’s efficiency depends on the ratio of traversable cells to obstacles and the accuracy of the heuristic function. A well-designed heuristic ensures fewer nodes are explored, reducing time and space costs.

CODE :

```
import heapq

import pandas as pd


# Define the grid map (0: traversable, 1: obstacle) as a pandas DataFrame

data = [

    [0, 1, 0, 0, 0],
```

```
    [0, 1, 0, 1, 0],
    [0, 0, 0, 1, 0],
    [1, 1, 0, 0, 0],
    [0, 0, 0, 1, 0]
]
grid = pd.DataFrame(data)

start = (0, 0)  # Starting position (row, column)
goal = (4, 4)  # Goal position (row, column)

# Heuristic function (Manhattan distance)
def heuristic(a, b):
    return abs(a[0] - b[0]) + abs(a[1] - b[1])

def a_star_search(grid, start, goal):
    rows, cols = grid.shape
```

```
open_set = []
heapq.heappush(open_set, (0, start))
came_from = {}
g_score = {start: 0}
f_score = {start: heuristic(start, goal)}

print("Starting A* algorithm...")
print(f"Start: {start}, Goal: {goal}\n")

while open_set:
    _, current = heapq.heappop(open_set)
    print(f"Processing node: {current}")

    if current == goal:
        path = []
        while current in came_from:
```

```
            path.append(current)
            current = came_from[current]
        path.append(start)
        print("\nPath found! Reconstructing path...\n")
        return path[::-1]  # Return reversed path

    neighbors = [
        (current[0] + dr, current[1] + dc)
        for dr, dc in [(-1, 0), (1, 0), (0, -1), (0, 1)]
    ]

    for neighbor in neighbors:
        r, c = neighbor
        if 0 <= r < rows and 0 <= c < cols and grid.iloc[r, c] == 0:
```

```
            tentative_g_score = g_score[current] + 1

            if neighbor not in g_score or tentative_g_score < g_score[neighbor]:

                came_from[neighbor] = current

                g_score[neighbor] = tentative_g_score

                f_score[neighbor] = tentative_g_score + heuristic(neighbor, goal)

                heapq.heappush(open_set, (f_score[neighbor], neighbor))

                print(f"  Neighbor {neighbor} updated with g_score={g_score[neighbor]}, f_score={f_score[neighbor]}")


    print("\nNo path found.")

    return None  # No path found
```

```
# Run A* algorithm
path = a_star_search(grid, start, goal)

if path:
    print("\nPath found:")
    for node in path:
        print(node)
else:
    print("\nNo path found.")
```

OUTPUT :

```
Start: (0, 0), Goal: (4, 4)

Processing node: (0, 0)
  Neighbor (1, 0) updated with g_score=1, f_score=8
Processing node: (1, 0)
  Neighbor (2, 0) updated with g_score=2, f_score=8
Processing node: (2, 0)
  Neighbor (2, 1) updated with g_score=3, f_score=8
Processing node: (2, 1)
  Neighbor (2, 2) updated with g_score=4, f_score=8
Processing node: (2, 2)
  Neighbor (1, 2) updated with g_score=5, f_score=10
  Neighbor (3, 2) updated with g_score=5, f_score=8
Processing node: (3, 2)
  Neighbor (4, 2) updated with g_score=6, f_score=8
  Neighbor (3, 3) updated with g_score=6, f_score=8
Processing node: (3, 3)
  Neighbor (3, 4) updated with g_score=7, f_score=8
Processing node: (3, 4)
  Neighbor (2, 4) updated with g_score=8, f_score=10
  Neighbor (4, 4) updated with g_score=8, f_score=8
Processing node: (4, 2)
  Neighbor (4, 1) updated with g_score=7, f_score=10
Processing node: (4, 4)

Path found! Reconstructing path...
```

```
Processing node: (4, 2)
  Neighbor (4, 1) updated with g_score=7, f_score=10
Processing node: (4, 4)

Path found! Reconstructing path...


Path found:
(0, 0)
(1, 0)
(2, 0)
(2, 1)
(2, 2)
(3, 2)
(3, 3)
(3, 4)
(4, 4)
```