*A Pathfinding Algorithm Report**

Name : Arpit Dixit

Roll No: 202401100300067

Course: B.Tech CSE (AI)

Introduction:

Pathfinding is a crucial technique in computer science, widely used in robotics, video games, and AI applications. The *A (A-Star) Algorithm** is a popular pathfinding algorithm that finds the shortest path from a start node to a goal node efficiently by combining **Dijkstra's algorithm** and **Greedy Best-First Search**.

This report presents the implementation of the A* algorithm on a grid-based environment to compute the shortest path. The algorithm considers both the actual movement cost and an estimated heuristic cost to reach the goal, ensuring optimal performance.

Methodology:

1. **Grid Representation:**
   - The environment is represented as a **2D grid**, where each cell is a node.
   - Nodes can be **walkable (white)** or **obstacles (black)**.
2. *A Algorithm Working:**
   - Each node maintains three costs:
     - **g(n):** Cost from the start node to the current node.
     - **h(n):** Estimated heuristic cost from current node to goal.
     - **f(n) = g(n) + h(n):** Total estimated cost.
   - A **priority queue (min-heap)** is used to expand the node with the lowest cost.
   - The **Manhattan Distance** heuristic is used: h(n)=∣x1−x2∣+∣y1−y2∣h(n) = |x_1 - x_2| + |y_1 - y_2|h(n)=∣x1−x2∣+∣y1−y2∣

- The algorithm terminates when the goal node is reached, and the shortest path is reconstructed.

Code:

```
import heapq

# Define the grid size
ROWS, COLS = 5, 5

# Directions for moving: right, left, down, up
DIRECTIONS = [(0, 1), (0,-1), (1, 0), (-1, 0)]

class Node:
    def __init__(self, row, col):
        self.row, self.col = row, col
        self.g = float("inf")  # Cost from start to current
node
        self.h = 0  # Heuristic cost to goal
        self.f = float("inf")  # Total cost
```

```
        self.parent = None  # Parent node for path
tracking

    def __lt__(self, other):
        return self.f < other.f  # Comparison for priority
queue

def heuristic(a, b):
    """ Manhattan Distance heuristic """
    return abs(a.row- b.row) + abs(a.col- b.col)

def a_star(grid, start, goal):
    open_set = []
    heapq.heappush(open_set, (0, start))  # Push start
node
    start.g, start.h, start.f = 0, heuristic(start, goal),
heuristic(start, goal)

    while open_set:
        current = heapq.heappop(open_set)[1]
```

```
        # Goal reached
        if current == goal:
            path = []
            while current:
                path.append((current.row, current.col))
                current = current.parent
            return path[::-1]  # Reverse path

        # Check neighbors
        for dr, dc in DIRECTIONS:
            r, c = current.row + dr, current.col + dc
            if 0 <= r < ROWS and 0 <= c < COLS:  # Within bounds
                neighbor = grid[r][c]
                temp_g = current.g + 1

                if temp_g < neighbor.g:  # Found a better path
```

```
                neighbor.g, neighbor.h = temp_g,
heuristic(neighbor, goal)
                neighbor.f = neighbor.g + neighbor.h
                neighbor.parent = current
                heapq.heappush(open_set, (neighbor.f,
neighbor))

    return None  # No path found

# Create grid
grid = [[Node(r, c) for c in range(COLS)] for r in
range(ROWS)]
start, goal = grid[0][0], grid[ROWS- 1][COLS- 1]

# Run A* Algorithm
path = a_star(grid, start, goal)

# Print the path
if path:
```

```
    print("Path found:", path)
else:
    print("No path found")
```

Result :

The program finds the shortest path from the start position (0,0) to the goal (4,4). Below is an example output of the execution:

Path found: [(0, 0), (1, 0), (2, 0), (3, 0), (4, 0), (4, 1), (4, 2), (4, 3), (4, 4)]

References/Credits:

1. A* Algorithm: https://en.wikipedia.org/wiki/A*_search_algorithm
2. Python Heapq (Priority Queue): https://docs.python.org/3/library/heapq.html
3. Artificial Intelligence: A Modern Approach by Stuart Russell & Peter Norvig

Conclusion:

The A* algorithm efficiently finds the shortest path in a grid environment. By balancing cost and heuristic estimation, it ensures an optimal and fast solution. This implementation can be extended to handle obstacles, larger grids, and real-world applications such as game development and robotics.