REPORT

NAME: YASHIKA TYAGI

ROLL NO:202401100300286

SEC: CSEAI-D

CLASS ROLLNO: 66

PROBLEM STATEMENT:-

Pathfinding with A* Algorithm

INTRODUCTION

The problem is about finding the shortest path on a 2D grid from a **start point** to a **goal point**, while avoiding obstacles. You need to implement the *A algorithm**, which uses a combination of actual movement cost and estimated distance (heuristic) to find the most efficient path. The algorithm moves only up, down, left, or right and should return the shortest path if possible, or indicate that no path exists if the goal is blocked.

Key Points:

1. **Grid**: Cells are either obstacles (`*`), free space (`.`), start (`S`), or goal (`G`).
2. **Movement**: Only vertical and horizontal movement is allowed.
3. **Heuristic**: Manhattan distance is used to estimate the distance from each point to the goal.
4. *A- Algorithm**: Finds the shortest path using a combination of actual cost and heuristic.

METHODOLOGY

The approach uses the *A algorithm** to find the shortest path on a grid. It starts by evaluating nodes based on their **g-value** (cost from start) and **h-value** (Manhattan distance to the goal). The node with the lowest **f-value** (g + h) is explored first. Neighbors are checked and added to the open list if they are valid and offer a lower cost. The process repeats until the goal is reached or no path exists, ensuring the shortest path is found while avoiding obstacles.

TYPED CODE

```python
class Node:
    def __init__(self, x, y):
        self.x = x
        self.y = y
        self.is_obstacle = False  # Can be used to mark obstacles
        self.parent = None  # Used to reconstruct the path

    def __eq__(self, other):
        return self.x == other.x and self.y == other.y

    def __hash__(self):
        return hash((self.x, self.y))  # For using Node in dictionaries
and sets

# Heuristic function: Manhattan distance
def heuristic(node, goal):
    return abs(node.x - goal.x) + abs(node.y - goal.y)

# Neighbor function for a 2D grid (up, down, left, right)
def get_neighbors(node, grid):
    neighbors = []
    directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]  # Up, Down, Left,
Right
    for dx, dy in directions:
        neighbor_x = node.x + dx
        neighbor_y = node.y + dy
        if 0 <= neighbor_x < len(grid) and 0 <= neighbor_y <
len(grid[0]):  # Within bounds
            neighbor = grid[neighbor_x][neighbor_y]
            if not neighbor.is_obstacle:  # Ignore obstacles
                neighbors.append(neighbor)
    return neighbors

# Function to reconstruct the path from the came_from dictionary
def reconstruct_path(goal):
    path = []
    current = goal
    while current:
        path.append((current.x, current.y))
        current = current.parent
    return path[::-1]  # Reverse the path
```

```
# A* algorithm
def a_star(start, goal, grid):
    open_list = [start]  # Nodes to be evaluated
    closed_list = set()  # Nodes already evaluated
    g_values = {start: 0}  # Cost from start to current node
    f_values = {start: heuristic(start, goal)}  # Estimated total cost

    while open_list:
        # Get the node with the lowest f value
        current_node = min(open_list, key=lambda node: f_values[node])

        if current_node == goal:
            return reconstruct_path(goal)

        open_list.remove(current_node)
        closed_list.add(current_node)

        for neighbor in get_neighbors(current_node, grid):
            if neighbor in closed_list:
                continue

            tentative_g = g_values[current_node] + 1  # Assuming each step
has a cost of 1

            if neighbor not in open_list:
                open_list.append(neighbor)
            elif tentative_g >= g_values.get(neighbor, float('inf')):
                continue

            neighbor.parent = current_node
            g_values[neighbor] = tentative_g
            f_values[neighbor] = g_values[neighbor] + heuristic(neighbor,
goal)

    return None  # If no path is found

# Create a simple grid (0 represents open space, 1 represents obstacle)
def create_grid():
    grid = []
    for i in range(5):
        row = []
        for j in range(5):
            node = Node(i, j)
            if (i == 1 and j != 0) or (i == 3 and j != 4):  # Adding
obstacles (replacing # with *)
```

```
                node.is_obstacle = True
            row.append(node)
        grid.append(row)
    return grid

# Function to print the grid for visualization
def print_grid(grid):
    for row in grid:
        print(" ".join(['*' if node.is_obstacle else '.' for node in
row]))

# Example usage
grid = create_grid()

start_node = grid[0][0]
goal_node = grid[4][4]

print("Grid:")
print_grid(grid)

path = a_star(start_node, goal_node, grid)

if path:
    print("\nPath found:")
    for node in path:
        print(f"({node[0]}, {node[1]})")
else:
    print("No path found.")
```

OUTPUT

```
Grid:
. . . . .
. * * * *
. . . . .
* * * * .
. . . . .

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

REFERENCES USED:

WEBSITE USED: CHAT-GPT