**A logo for a company

Description automatically generated**

**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**

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:")

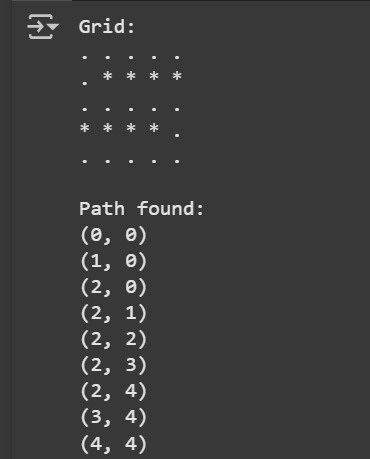
    for node in path:

        print(f"({node[0]}, {node[1]})")

else:

    print("No path found.")

**OUTPUT**

****

**REFERENCES USED:**

WEBSITE USED: CHAT-GPT