KIET GROUP OF INSTITUTIONS

INTRODUCTION TO AI

MSE 1

PROBLEM STATEMENT:

Pathfinding with A* Algorithm

Name: SNEHA SAHU

Roll Number: 202401100400188

Branch: CSE(AIML)

Section: C

INTRODUCTION

Introduction to Pathfinding with A* Algorithm

Pathfinding is the process of finding a path from a start point to a goal point while navigating through obstacles or constraints. It's a crucial aspect of many fields, from robotics and game development to navigation systems. One of the most popular algorithms for pathfinding is **A* (A-star)**, which is widely used due to its efficiency and ability to find the shortest path in a variety of environments.

It works by evaluating paths based on two main factors:

1. **g(n)**: The cost it took to get to the current point.
2. **h(n)**: An estimate of the cost to reach the goal from the current point (called a heuristic).

A* combines these factors to calculate **f(n) = g(n) + h(n)**, and it chooses the path with the lowest f(n) value to explore first. This allows A* to find the shortest path quickly and efficiently, making it widely used in areas like game development, navigation systems, and robotics.

METHODOLOGY

To solve a pathfinding problem using the *A algorithm**, the approach is as follows:

1. **Initialize Lists**:
   - Create an **open list** to hold nodes that need to be evaluated.
   - Create a **closed list** to keep track of nodes that have already been evaluated.
2. **Start from the Initial Node**:
   - Add the starting node to the open list with a cost of 0 for g(n) (no movement yet) and compute its heuristic h(n).
3. **Explore Nodes**:
   - Pick the node with the lowest f(n) from the open list.
   - For the current node, examine its neighbors (adjacent nodes).
   - For each neighbor, calculate:
     - **g(n)** (cost to move to that neighbor),
     - **h(n)** (heuristic estimate to the goal),
     - **f(n)** (total cost, f(n) = g(n) + h(n)).

4. **Update Lists**:
   - If a neighbor is not in the open or closed list, add it to the open list with its calculated f(n).
   - If a neighbor is already in the open list and the new f(n) is lower, update its f(n).
5. **Repeat**:
   - Continue this process until the goal node is found or the open list is empty (which means no path exists).
6. **Reconstruct Path**:
   - Once the goal is reached, reconstruct the path by tracing from the goal node back to the start node, following parent nodes.

By evaluating nodes in this way, A* efficiently finds the shortest path while considering both the cost of moving and the estimated distance to the goal.

CODE

```python
import heapq

# Define a Node class for A* search
class Node:
    def __init__(self, position, parent=None):
        self.position = position  # Current node's position as a tuple (x, y)
        self.parent = parent      # Parent node (for backtracking the path)
        self.g = 0                # Cost from the start node to this node
        self.h = 0                # Heuristic cost (estimated cost to the goal)
        self.f = 0                # Total cost (g + h)

    def __lt__(self, other):
        return self.f < other.f  # For priority queue comparisons

# A* algorithm implementation
def astar(maze, start, end):
    """
    Perform A* search to find the shortest path in a 2D grid maze.
    :param maze: 2D list representing the grid (0 = open, 1 = obstacle)
    :param start: Tuple (x, y) representing the starting position
    :param end: Tuple (x, y) representing the target position
    :return: A list of tuples representing the path from start to end (or empty if no path exists)
    """
    # Initialize the open and closed lists
    open_list = []
    closed_list = set()

    # Add the start node to the open list
    start_node = Node(start)
    goal_node = Node(end)
    heapq.heappush(open_list, start_node)

    while open_list:
```

```python
# Get the node with the lowest f score
current_node = heapq.heappop(open_list)
closed_list.add(current_node.position)

# If we've reached the goal, backtrack the path and return it
if current_node.position == goal_node.position:
    path = []
    while current_node:
        path.append(current_node.position)
        current_node = current_node.parent
    return path[::-1]  # Reverse path to get it from start to goal

# Generate neighbors
neighbors = [
    (0, -1),  # Up
    (0, 1),   # Down
    (-1, 0),  # Left
    (1, 0)    # Right
]

for move in neighbors:
    # Calculate neighbor position
    neighbor_pos = (current_node.position[0] + move[0], current_node.position[1] + move[1])

    # Check if the neighbor is within the maze boundaries and is walkable
    if (
        0 <= neighbor_pos[0] < len(maze) and
        0 <= neighbor_pos[1] < len(maze[0]) and
        maze[neighbor_pos[0]][neighbor_pos[1]] == 0 and
        neighbor_pos not in closed_list
    ):
        neighbor_node = Node(neighbor_pos, current_node)
```

```python
                # Calculate the costs
                neighbor_node.g = current_node.g + 1  # Cost to move to neighbor
                neighbor_node.h = abs(neighbor_pos[0] - goal_node.position[0]) + abs(neighbor_pos[1] - goal_node.position[1])  # Manhattan distance heuristic
                neighbor_node.f = neighbor_node.g + neighbor_node.h

                # Add the neighbor to the open list if not already there with a lower f score
                if not any(open_node.position == neighbor_node.position and open_node.f <= neighbor_node.f for open_node in open_list):
                    heapq.heappush(open_list, neighbor_node)

    return []  # Return empty path if no path is found

# Example usage
maze = [
    [0, 1, 0, 0, 0],
    [0, 1, 0, 1, 0],
    [0, 0, 0, 1, 0],
    [0, 1, 1, 1, 0],
    [0, 0, 0, 0, 0]
]

start = (0, 0)  # Starting position
end = (4, 4)    # Goal position

path = astar(maze, start, end)
print("Path:", path)
```

OUTPUT / RESULT

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