TOPIC: Pathfinding with A\* Algorithm

NAME: YASHASH TYAGI

UNIVERSITY ROLL NO. :202401100400218

INTRODUCTION

Pathfinding algorithms are designed to navigate a path from a starting point to a goal, traversing through a grid or graph while avoiding obstacles. These algorithms are the backbone of many real-world applications, enabling autonomous robots to navigate their environments, characters in video games to find their way through complex terrains, and GPS systems to provide the shortest route from one location to another. The primary goal of a pathfinding algorithm is to find the most efficient path, minimizing the cost, time, or distance traveled.

The A\* algorithm operates by maintaining a priority queue of nodes, known as the open list. It begins at the starting node and explores neighboring nodes, calculating the total cost for each node using the formula:

f(n)=g(n)+h(n)

 f(n) represents the total estimated cost of the path through node nn.

 g(n) denotes the actual cost from the starting node to node nn.

 h(n) is the heuristic function that estimates the cost from node nn to the goal node.

METHODOLOGY

**Methodology for Pathfinding Using the A\* Algorithm**

**Step 1: Define the Problem**

1. **Grid Representation**: Represent the environment as a grid or graph, where each cell or node can either be traversable or contain an obstacle. This can be a 2D grid, weighted graph, or any other suitable representation.
2. **Start and Goal Nodes**: Identify the starting point and the goal point within the grid. The objective is to find the most efficient path from the start node to the goal node while avoiding obstacles.

**Step 2: Initialize the Open and Closed Lists**

1. **Open List**: The open list contains nodes that need to be evaluated. It starts with only the start node.
2. **Closed List**: The closed list contains nodes that have already been evaluated. It starts empty.

**Step 3: Calculate Costs and Heuristics**

1. **Cost Calculation**: For each node, calculate the following costs:
   * g(n)g(n): The actual cost from the start node to node nn.
   * h(n)h(n): The heuristic estimate of the cost from node nn to the goal node. This can be calculated using various heuristic functions such as Euclidean distance, Manhattan distance, or Chebyshev distance.
2. **Total Cost**: Calculate the total cost f(n)=g(n)+h(n)f(n) = g(n) + h(n) for each node. The node with the lowest f(n)f(n) value will be prioritized for evaluation.

**Step 4: Evaluate Nodes**

1. **Current Node**: Select the node with the lowest f(n)f(n) value from the open list and set it as the current node.
2. **Goal Check**: If the current node is the goal node, the algorithm terminates, and the path is reconstructed by tracing back from the goal node to the start node.
3. **Move to Closed List**: Remove the current node from the open list and add it to the closed list, indicating that it has been evaluated.

**Step 5: Explore Neighbors**

1. **Neighbor Nodes**: Identify all traversable neighbor nodes of the current node. For each neighbor:
   * If the neighbor is already in the closed list, skip it.
   * If the neighbor is not in the open list, add it to the open list and calculate its g(n)g(n), h(n)h(n), and f(n)f(n) values.
2. **Path Update**: If the new path to the neighbor node is shorter than the previously known path, update the neighbor's g(n)g(n) and f(n)f(n) values, and set the current node as the neighbor's parent node.

**Step 6: Repeat**

1. **Iteration**: Repeat steps 4 and 5 until the open list is empty or the goal node is reached.
2. **Path Reconstruction**: Once the goal node is reached, reconstruct the path by tracing back from the goal node to the start node using the parent nodes.

CODE

import heapq  # Importing the heapq module for priority queue implementation

# A\* algorithm function

def a\_star(grid, start, goal):

    # Heuristic function to estimate the cost from a node to the goal (Manhattan distance)

    def heuristic(a, b):

        return abs(a[0] - b[0]) + abs(a[1] - b[1])

    open\_list = []  # Priority queue (open list) to store nodes to be explored

    heapq.heappush(open\_list, (0, start))  # Push the start node into the open list with f\_score of 0

    came\_from = {}  # Dictionary to track the path

    g\_score = {start: 0}  # Dictionary to store the cost from the start node to each node

    f\_score = {start: heuristic(start, goal)}  # Dictionary to store the total estimated cost (g\_score + heuristic)

    while open\_list:  # Loop while there are nodes to explore

        \_, current = heapq.heappop(open\_list)  # Pop the node with the lowest f\_score from the open list

        if current == goal:  # If the goal is reached

            path = []  # Initialize an empty path

            while current in came\_from:  # Reconstruct the path by backtracking

                path.append(current)

                current = came\_from[current]

            return path[::-1]  # Return the path in the correct order

        # Explore the neighbors (up, down, left, right) of the current node

        for dx, dy in [(-1, 0), (1, 0), (0, -1), (0, 1)]:

            neighbor = (current[0] + dx, current[1] + dy)

            # Check if the neighbor is within the grid bounds and is passable (not an obstacle)

            if 0 <= neighbor[0] < len(grid) and 0 <= neighbor[1] < len(grid[0]) and grid[neighbor[0]][neighbor[1]] == 0:

                tentative\_g\_score = g\_score[current] + 1  # Calculate the tentative g\_score for the neighbor

                # If the neighbor is not in g\_score or the tentative g\_score is better

                if neighbor not in g\_score or tentative\_g\_score < g\_score[neighbor]:

                    came\_from[neighbor] = current  # Update the path to the neighbor

                    g\_score[neighbor] = tentative\_g\_score  # Update the g\_score for the neighbor

                    f\_score[neighbor] = tentative\_g\_score + heuristic(neighbor, goal)  # Update the f\_score for the neighbor

                    heapq.heappush(open\_list, (f\_score[neighbor], neighbor))  # Push the neighbor into the open list with its f\_score

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

# Example grid: 0 represents passable cells, 1 represents obstacles

grid = [

    [0, 1, 0, 0, 0],

    [0, 1, 0, 1, 0],

    [0, 0, 0, 1, 0],

    [0, 1, 0, 0, 0],

    [0, 0, 0, 0, 0],

]

start = (0, 0)  # Starting position

goal = (4, 4)   # Goal position

# Execute the A\* algorithm to find the path from start to goal

path = a\_star(grid, start, goal)

print("Path found:", path)  # Print the path found

OUTPUT

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