PATHFINDING WITH A* ALGORITHM

NAME : SAMVEG JAIN

BRANCH : CSE AI (C)

UNIVERSITY ROLL NO. : 202401100300214

Introduction

A* algorithm is a pathfinding algorithm which is generally used for finding the shortest path in graph, grid etc. It's speciality is that it will find a solution if one such exists only if the conditions satisfy :-

1) The graph should be finite.
2) The heuristic used for estimating the cost of the goal must be admissible.

It works by exploring the nodes of a graph, with node representating a state in search space.

Methodology

Step 1: Let's start with the initial setup

1) We'll take a start node.
2) We'll take a end node or the target position.
3) We'll represent the grid in 2D suppose.

Step 2: Node representation

1) Take the coordinates of a node as a position.
2) Take some cost function variables nodes.

3) And take a node which we will use to get a check upon which Node we a re currently on.

Step 3: The heuristic calculation which will be done by the heuristic function.

Step 4: Then comes the Algorithm steps which includes:-

1) Initialization
2) Main loop
3) Termination

CODE

```
import heapq

# A* algorithm to find the shortest path
class Node:
    def __init__(self, position, g_cost=0, h_cost=0, parent=None):
        self.position = position  # (x, y) coordinates
        self.g_cost = g_cost  # Cost from start to current node
        self.h_cost = h_cost  # Estimated cost from current node to end
        self.f_cost = g_cost + h_cost  # Total cost (f = g + h)
        self.parent = parent  # Parent node to track the path

    def __lt__(self, other):
        return self.f_cost < other.f_cost

def astar(start, end, grid):
    open_list = []
    closed_list = set()

    # Start node
    start_node = Node(start, g_cost=0, h_cost=heuristic(start, end))
    heapq.heappush(open_list, start_node)

    while open_list:
```

```
        current_node = heapq.heappop(open_list)

        if current_node.position == end:
            path = []
            while current_node:
                path.append(current_node.position)
                current_node = current_node.parent
            return path[::-1]  # Return reversed path (from start to end)

        closed_list.add(current_node.position)

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

            g_cost = current_node.g_cost + 1
            h_cost = heuristic(neighbor, end)
            neighbor_node = Node(neighbor, g_cost=g_cost, h_cost=h_cost,
parent=current_node)

            if not any(open_node.position == neighbor and
open_node.f_cost <= neighbor_node.f_cost for open_node in open_list):
                heapq.heappush(open_list, neighbor_node)

    return None  # Return None if no path found
```

```
def heuristic(a, b):
    # Manhattan distance heuristic
    return abs(a[0] - b[0]) + abs(a[1] - b[1])

def get_neighbors(node, grid):
    x, y = node.position
    neighbors = []

    # Check 4 possible directions (left, right, up, down)
    directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]
    for dx, dy in directions:
        nx, ny = x + dx, y + dy
        if 0 <= nx < len(grid) and 0 <= ny < len(grid[0]) and grid[nx][ny] == 0:
            neighbors.append((nx, ny))

    return neighbors

# Example usage
if __name__ == "__main__":
    grid = [
        [0, 1, 0, 0, 0],
        [0, 1, 0, 1, 0],
        [0, 0, 0, 1, 0],
        [0, 1, 0, 0, 0],
        [0, 0, 0, 1, 0]
```

```
]

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

path = astar(start, end, grid)

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

OUTPUT

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