Pathfinding with A* Algorithm

Name : Mohammad Fahim

Branch : CSE(AI)

Sec : C

University roll no. : 202401100300154

Introduction:

The *A (A-star) algorithm* is a popular pathfinding algorithm used to find the **shortest path** between two points in a grid. It is widely used in **AI, robotics, and games**.

A* works by combining:

- **G-cost** (distance from the start point).
- **H-cost** (estimated distance to the goal, called the heuristic).
- **F-cost** (total cost: G + H).

Methodology:

The A* algorithm follows these steps to find the shortest path:

1. **Initialize**:
   - Define the **maze/grid** with obstacles.
   - Set the **start** and **goal** positions.
2. **Create Nodes**:
   - Each position is treated as a **node** with:
     - **G-cost** (distance from start)
     - **H-cost** (estimated distance to goal)
     - **F-cost** (total cost: G + H)
3. **Priority Queue (Open List)**:
   - Start with the **initial node** in a priority queue.
   - Pick the node with the **lowest F-cost** to explore first.
4. **Expand Neighbors**:
   - Check all possible **moves** (up, down, left, right).
   - Skip obstacles and already visited nodes.
   - Calculate new **G, H, and F values**.
5. **Path Construction**:
   - If the **goal is reached**, backtrack from the **goal node** to reconstruct the shortest path.
6. **Output**:
   - Return the path if found; otherwise, state that no path exists.

Code:

```
import heapq  # To use priority queue for open list


# A* algorithm for pathfinding


class Node:
    """
    Represents a node in the grid.
    It stores the position, its g, h, and f scores, and references to its parent.
    """
    def __init__(self, x, y):
        self.x = x
        self.y = y
        self.g = float('inf')  # g: cost from start to this node
        self.h = 0  # h: heuristic estimate from this node to the goal
        self.f = float('inf')  # f: f = g + h (used for sorting in the open list)
        self.parent = None  # Reference to parent node


    def __lt__(self, other):
        """
        Less-than operator to compare nodes based on their f score
        This is necessary for heapq to sort nodes by their f value in the priority queue.
        """
        return self.f < other.f


def a_star(start, goal, grid):
    """
    A* algorithm to find the shortest path from start to goal on a 2D grid.
    :param start: Tuple (x, y) for start node position
    :param goal: Tuple (x, y) for goal node position
    :param grid: 2D list representing the grid (1 for obstacles, 0 for free space)
    :return: List of nodes representing the path from start to goal, or an empty list if no path found
```

```python
"""
open_list = []  # Priority queue for nodes to be evaluated
closed_list = set()  # Set of nodes that have been evaluated

# Initialize the start and goal nodes
start_node = Node(start[0], start[1])
start_node.g = 0
start_node.h = abs(start_node.x - goal[0]) + abs(start_node.y - goal[1])  # Manhattan distance
start_node.f = start_node.g + start_node.h
heapq.heappush(open_list, start_node)  # Add start node to open list

goal_node = Node(goal[0], goal[1])

# Define possible movements (up, down, left, right)
movements = [(-1, 0), (1, 0), (0, -1), (0, 1)]

while open_list:
    current_node = heapq.heappop(open_list)  # Node with the lowest f value
    closed_list.add((current_node.x, current_node.y))  # Add current node to closed list

    # If we reach the goal node, reconstruct the path
    if (current_node.x, current_node.y) == (goal_node.x, goal_node.y):
        path = []
        while current_node:
            path.append((current_node.x, current_node.y))
            current_node = current_node.parent
        return path[::-1]  # Return reversed path (start to goal)

    # Evaluate each possible move
    for move in movements:
        neighbor_x = current_node.x + move[0]
        neighbor_y = current_node.y + move[1]
```

```
            # Check if the neighbor is within the grid bounds and is not an obstacle
            if 0 <= neighbor_x < len(grid) and 0 <= neighbor_y < len(grid[0]) and
grid[neighbor_x][neighbor_y] == 0:
                if (neighbor_x, neighbor_y) in closed_list:
                    continue  # Ignore already evaluated neighbors

                neighbor_node = Node(neighbor_x, neighbor_y)
                tentative_g = current_node.g + 1  # Assume the cost from current node to neighbor is
always 1

                # If neighbor is not in the open list, or we found a better g score
                if tentative_g < neighbor_node.g:
                    neighbor_node.g = tentative_g
                    neighbor_node.h = abs(neighbor_node.x - goal[0]) + abs(neighbor_node.y - goal[1])  #
Manhattan distance
                    neighbor_node.f = neighbor_node.g + neighbor_node.h
                    neighbor_node.parent = current_node

                    # If the neighbor is not in the open list, add it
                    if not any(neighbor.x == neighbor_node.x and neighbor.y == neighbor_node.y for
neighbor in open_list):
                        heapq.heappush(open_list, neighbor_node)

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

# Example grid (0 = free space, 1 = obstacle)
grid = [
    [0, 0, 0, 0, 0],
    [0, 1, 0, 1, 0],
    [0, 1, 0, 1, 0],
    [0, 0, 0, 0, 0],
    [0, 1, 0, 0, 0]
]
```

```python
start = (0, 0)  # Starting point (x, y)
goal = (4, 4)   # Goal point (x, y)

# Run the A* algorithm
path = a_star(start, goal, grid)

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

Screenshot of Output :

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