Lab Assignment 4

Subject: Artificial Intelligence **Guided by:** Dr. Anuradha Yenkikar

Student name: Neha Sawant

Experiment Name: A* Algorithm Implementation for an application

Introduction:

The A* (A-star) algorithm is a popular pathfinding and graph traversal algorithm used to find the shortest path from a starting node to a target node. This implementation provides a clear understanding of how the algorithm operates in a grid environment, accounting for obstacles and utilizing heuristics to optimize the search.

Problem Statement:

The goal is to implement the A* algorithm to navigate a 2D grid, identifying a path from a specified starting point to a target point while avoiding obstacles. The grid is represented as a 2D array, where:

- `0` indicates a walkable cell.
- `1` indicates an obstacle.

Tools and Technologies:

- Programming Language: Java
- Data Structures Used:
  - Arrays for the grid representation.
  - Lists for open and closed sets of nodes.
  - A custom `Node` class to represent individual grid points.

Key Classes and Methods:

1] Class Node

The `Node` class represents a point in the grid and includes:

- `x` and `y`: coordinates of the node.
- `f`, `g`, `h`: cost values for pathfinding.
- `parent`: reference to the parent node for path reconstruction.

2] Class AStar

The `AStar` class encapsulates the algorithm's functionality:

- Constructor: Initializes the grid, closed list, and open list.

```
AStar(int[][] grid) {
    this.grid = grid;
    this.closedList = new boolean[grid.length][grid[0].length];
    this.openList = new ArrayList<>();
}
```

3] Heuristic Function

The heuristic function estimates the cost from the current node to the goal:

```
private double heuristic(Node a, Node b) {
    return Math.abs(a.x - b.x) + Math.abs(a.y - b.y);
}
```

This implementation uses the Manhattan distance.

4] Finding the Path

The `findPath` method implements the core A* logic:

```java
public class AStar {
    public List<Node> findPath(Node start, Node goal) {
        openList.add(start);
        while (!openList.isEmpty()) {
            Node current = openList.stream().min(Comparator.comparingDouble(n -> n.f)).orElse(other:null);
            if (current.x == goal.x && current.y == goal.y) return constructPath(current);

            openList.remove(current);
            closedList[current.x][current.y] = true;

            for (Node neighbor : getNeighbors(current)) {
                if (closedList[neighbor.x][neighbor.y]) continue;
                double tentativeG = current.g + 1;

                if (!openList.contains(neighbor)) {
                    neighbor.g = tentativeG;
                    neighbor.h = heuristic(neighbor, goal);
                    neighbor.f = neighbor.g + neighbor.h;
                    neighbor.parent = current;
                    openList.add(neighbor);
                } else if (tentativeG < neighbor.g) {
                    neighbor.g = tentativeG;
                    neighbor.h = heuristic(neighbor, goal);
                    neighbor.f = neighbor.g + neighbor.h;
                    neighbor.parent = current;
                }
            }
        }
        return Collections.emptyList(); // No path found
```

5] Constructing the Path

The `constructPath` method reconstructs the path from the goal node back to the start:

```java
private List<Node> constructPath(Node node) {
    List<Node> path = new ArrayList<>();
    while (node != null) {
        path.add(node);
        node = node.parent;
    }
    Collections.reverse(path);
    return path;
}
```

6] Getting Neighbors

The `getNeighbors` method identifies walkable neighboring nodes:

```java
private List<Node> getNeighbors(Node node) {
    List<Node> neighbors = new ArrayList<>();
    int[][] directions = {{-1, 0}, {1, 0}, {0, -1}, {0, 1}};

    for (int[] dir : directions) {
        int newX = node.x + dir[0], newY = node.y + dir[1];
        if (newX >= 0 && newY >= 0 && newX < grid.length && newY < grid[0].length && grid[newX][newY] == 0) {
            neighbors.add(new Node(newX, newY));
        }
    }
    return neighbors;
}
```

Example Usage:

In the `main` method, an example grid is defined, and the A* algorithm is executed:

```
public static void main(String[] args) {
    int[][] grid = {
        {0, 0, 0, 0, 0},
        {0, 1, 1, 1, 0},
        {0, 0, 0, 0, 0},
        {0, 1, 1, 0, 0},
        {0, 0, 0, 0, 0}
    };

    AStar aStar = new AStar(grid);
    Node start = new Node(x:0, y:0), goal = new Node(x:4, y:4);
    List<Node> path = aStar.findPath(start, goal);

    if (!path.isEmpty()) {
        System.out.println(x:"Path found:");
        path.forEach(node -> System.out.println("(" + node.x + ", " + node.y + ")"));
    } else {
        System.out.println(x:"No path found");
    }
}
```

Execution and Output:

When you run the program, it should display the path found from the starting point `(0, 0)` to the goal point `(4, 4)` avoiding obstacles:

```
a\jdt_ws\AI_Labs_44e150ae\bin'
Path found:
(0, 0)
(1, 0)
(2, 0)
(3, 0)
(4, 0)
(4, 1)
(0, 0)
(1, 0)
(2, 0)
(3, 0)
(4, 0)
(4, 1)
(2, 0)
(3, 0)
(4, 0)
(4, 1)
(3, 0)
(4, 0)
(4, 1)
(4, 0)
(4, 1)
(4, 1)
(4, 2)
(4, 3)
(4, 4)
```

Conclusion:

The A* algorithm implementation effectively navigates a grid, providing a clear and efficient way to find the shortest path while avoiding obstacles. Key learnings from this implementation include:

- Understanding the heuristic function's role in guiding the search.

- Utilizing open and closed lists to manage node processing.

- Constructing paths from goal nodes back to the start for clear output.

This implementation can be extended to various applications requiring pathfinding in grid-like environments.