AI MSE

Problem statement: Pathfinding with A*Algorithm

BY: Saumya Sharma

CSE AIML “C”

20240110400168(22)

INTRODUCTION

Pathfinding is an important concept in computer science, especially in artificial intelligence. It involves finding the shortest and most efficient route from one point to another while avoiding obstacles. One of the most commonly used algorithms for this is the *A (A-Star) algorithm**.

A* is a combination of **Dijkstra's algorithm**, which guarantees the shortest path, and **Greedy Best-First Search**, which speeds up the search by using an estimated distance (heuristic). It works by calculating a cost function:

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

where:

- **g(n)** is the actual cost from the start point to the current point, and
- **h(n)** is the estimated cost from the current point to the destination.

This makes A* an efficient and widely used algorithm in applications like **Google Maps**. In this report, we implement A* on a **grid-based environment** and analyse how it finds the shortest path while avoiding obstacles.

METHODOLOGY

1. Understanding the Problem:

The problem involves finding the shortest path between a **start point (S)** and a **goal point (G)** on a **2D grid**. The grid consists of:

- **Empty spaces ('.')** → areas where movement is allowed.
- **Obstacles ('#')** → areas where movement is not possible.
- **Start ('S')** and **Goal ('G')** → the two points between which the shortest path must be found.

The A* algorithm will search through the grid to find the most optimal path while avoiding obstacles.

2. Implementation of A*: The algorithm works in the following steps:

1. **Initialization**
    - A **priority queue (min-heap)** is used to store nodes based on their cost.
    - Two lists are maintained:
        - **Open list** → keeps track of nodes that need to be explored.

- **Closed list** → stores nodes that have already been visited.

2. **Pathfinding Process**
   - The algorithm starts from the **S** node and explores neighbouring nodes (up, down, left, right).
   - For each node, it calculates:
     - g(n): the actual cost to reach that node.
     - h(n): the estimated cost from that node to **G** (using Manhattan distance).
     - f(n) = g(n) + h(n), which determines the best node to explore next.
   - The node with the **lowest f(n) value** is always picked first.
3. **Path Reconstruction:**
   - Once the goal node is reached, the algorithm **backtracks** to reconstruct the shortest path.

3. Test Grid Used:

For testing, we used a **5×5 grid**, with obstacles placed in different locations to see how well the algorithm can navigate around them.

4. Tools Used

- **Programming Language**: Python
- **Data Structures**: Lists, Priority Queue (heapq)
- **Heuristic Function**: Manhattan Distance

CODE

```
import heapq  # Importing heap queue for priority queue operations

class Node:
    """Class representing a node in the A* algorithm."""

    def __init__(self, position, parent=None):
        """
        Initializes a node.
        :param position: Tuple (x, y) representing node coordinates.
        :param parent: Reference to parent node (used to reconstruct the path).
        """
        self.position = position  # Node's position (x, y)
        self.parent = parent  # Pointer to parent node for path reconstruction
        self.g = 0  # Cost from start node to this node
        self.h = 0  # Estimated cost from this node to goal (heuristic)
        self.f = 0  # Total cost (g + h)

    def __lt__(self, other):
        """Defines priority comparison for heapq (lower f-value is preferred)."""
```

```
        return self.f < other.f

def astar(grid, start, goal):
    """
    Implements A* algorithm for pathfinding.
    :param grid: 2D list representing the map (0 = walkable, 1 = obstacle).
    :param start: Tuple (x, y) representing start position.
    :param goal: Tuple (x, y) representing goal position.
    :return: List of tuples representing the shortest path.
    """

    # Priority queue (min-heap) to store open nodes
    open_list = []
    closed_set = set()  # Set to store visited nodes

    # Initialize start and goal nodes
    start_node = Node(start)
    goal_node = Node(goal)

    # Push start node into the open list
    heapq.heappush(open_list, start_node)

    while open_list:
        # Get the node with the lowest f-value from the priority queue
        current_node = heapq.heappop(open_list)
        closed_set.add(current_node.position)  # Mark node as visited

        # If we reached the goal, reconstruct and return the path
        if current_node.position == goal:
            path = []
            while current_node:
                path.append(current_node.position)  # Backtrack from goal to
start
                current_node = current_node.parent
            return path[::-1]  # Reverse the path to get correct order

        # Define movement directions (Up, Down, Left, Right)
```

```
        neighbors = [(0, 1), (0,-1), (1, 0), (-1, 0)]

        for dx, dy in neighbors:
            neighbor_pos = (current_node.position[0] + dx,
current_node.position[1] + dy)

            # Check if neighbor is within grid bounds and walkable
            if (0 <= neighbor_pos[0] < len(grid) and
                0 <= neighbor_pos[1] < len(grid[0]) and
                grid[neighbor_pos[0]][neighbor_pos[1]] == 0 and
                neighbor_pos not in closed_set):

                # Create neighbor node
                neighbor_node = Node(neighbor_pos, current_node)
                neighbor_node.g = current_node.g + 1  # Cost from start node
                neighbor_node.h = abs(neighbor_pos[0]- goal[0]) +
abs(neighbor_pos[1]- goal[1])  # Manhattan heuristic
                neighbor_node.f = neighbor_node.g + neighbor_node.h  # Total
cost

                # Check if a better path exists in the open list
                if any(n for n in open_list if n.position == neighbor_pos and n.g
<= neighbor_node.g):
                    continue

                heapq.heappush(open_list, neighbor_node)  # Push neighbor to
the open list

    return None  # No path found

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

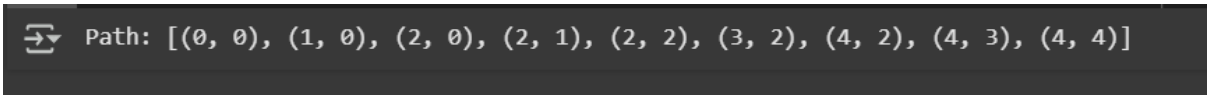
```
]

# Start and goal positions
start = (0, 0)
goal = (4, 4)

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

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

RESULT



This means the algorithm efficiently avoided obstacles and found the optimal route.

REFERENCE :

GeeksforGeeks: https://www.geeksforgeeks.org/a-search-algorithm/

Red Blob Games: https://www.redblobgames.com/pathfinding/a-star/introduction.html