Department of CSE (AI)

PATHFINDING WITH A* ALGORITHM

Name: Rakhi Garhwal
Branch: CSE(AI)
Section: C
University Roll No.: 202401100300194

Introduction

*Introduction to A Pathfinding Algorithm**

The A* (A-Star) algorithm is a widely used pathfinding technique in artificial intelligence and computer science. It is designed to find the shortest path between two points efficiently by combining elements of Dijkstra's algorithm and Greedy Best-First Search. A* is particularly useful in navigation, robotics, and game development.

*How A Works**

A* maintains an open list of nodes to explore, each with three values:

- **g(n):** Cost from the start node to the current node.
- **h(n):** Estimated cost from the current node to the goal.
- **f(n):** Total estimated cost, calculated as f(n) = g(n) + h(n).

The algorithm selects the node with the lowest f(n) and explores its neighbors. If a better path is found, the neighbor's cost values are updated, and the process continues until the goal is reached.

Applications of A*

A* is widely used in:

- **Video Games:** AI character movement and path navigation.
- **Robotics:** Autonomous navigation in complex environments.
- **GPS Navigation:** Finding the shortest driving routes.

Conclusion

A* is an efficient and widely used algorithm for pathfinding. By using a heuristic function, it optimizes the search process, making it more effective than traditional algorithms.

Mythodology

*Introduction to A Pathfinding Algorithm**

The A* (A-Star) algorithm is a widely used pathfinding technique in artificial intelligence and computer science. It efficiently finds the shortest path between two points by combining Dijkstra's algorithm and Greedy Best-First Search. A* is commonly used in navigation, robotics, and game development.

Methodology

A* follows a structured approach:

1. **Initialization:** Define start and goal nodes. Use an open list (nodes to explore) and a closed list (visited nodes).
2. **Cost Calculation:** Compute:
    - **g(n):** Cost from start to current node.
    - **h(n):** Estimated cost to goal (heuristic function).
    - **f(n) = g(n) + h(n):** Total estimated cost.
3. **Node Selection:** Choose the node with the lowest f(n). If it is the goal, reconstruct the path.
4. **Exploring Neighbors:** Identify valid neighboring nodes, update costs, and add them to the open list.
5. **Repeat Until Goal is Reached:** Continue selecting and updating nodes until the goal is found or no path exists.

Applications of A*

- **Video Games:** AI path navigation.
- **Robotics:** Autonomous movement.
- **GPS Navigation:** Finding shortest routes.

Conclusion

A* is an efficient pathfinding algorithm that optimizes searches using heuristics, making it faster and more effective than traditional methods.

CODE Typed

```
import heap

class Node:
    def __init__(self, position, parent=None):
        self.position = position  # (x, y) coordinates of the node
        self.parent = parent  # Parent node to trace back the path
        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):
        return self.f < other.f  # Compare nodes based on their f value for priority queue

def heuristic(a, b):
    # Calculate Manhattan distance as the heuristic (absolute difference in x and y coordinates)
    return abs(a[0] - b[0]) + abs(a[1] - b[1])

def astar(maze, start, end):
    open_list = []  # Priority queue to store nodes to be explored
    closed_set = set()  # Set to store visited nodes
    start_node = Node(start)  # Create start node
    goal_node = Node(end)  # Create goal node
    heapq.heappush(open_list, start_node)  # Add start node to priority queue

    while open_list:
```

```
        current_node = heapq.heappop(open_list)  # Get node with lowest f
value

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

        closed_set.add(current_node.position)  # Mark node as visited

        # Explore neighboring nodes (up, down, left, right)
        for dx, dy in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
            neighbor_pos = (current_node.position[0] + dx,
current_node.position[1] + dy)

            # Skip if the neighbor is out of bounds or is a wall (1 in the
maze)
            if (neighbor_pos in closed_set or
                neighbor_pos[0] < 0 or neighbor_pos[0] >= len(maze) or
                neighbor_pos[1] < 0 or neighbor_pos[1] >= len(maze[0]) or
                maze[neighbor_pos[0]][neighbor_pos[1]] == 1):
                continue

            # Create a new node for the neighbor
            neighbor = Node(neighbor_pos, current_node)
            neighbor.g = current_node.g + 1  # Increment g cost (movement
cost)
            neighbor.h = heuristic(neighbor_pos, goal_node.position)  #
Compute heuristic
            neighbor.f = neighbor.g + neighbor.h  # Compute total cost

            heapq.heappush(open_list, neighbor)  # Add neighbor to the
priority queue

    return None  # No path found

# Example usage
```

```python
maze = [
    [0, 1, 0, 0, 0],  # 0 = open path, 1 = obstacle
    [0, 1, 0, 1, 0],
    [0, 0, 0, 1, 0],
    [0, 1, 1, 1, 0],
    [0, 0, 0, 0, 0]
]
start = (0, 0)  # Start position (row, column)
end = (4, 4)  # Goal position (row, column)
path = astar(maze, start, end)  # Find the shortest path
print("Path:", path)  # Print the path from start to goal
```

ScreenShot Of Code Output

```
path = astar(maze, start, end)  # Find the shortest path
print("Path:", path)  # Print the path from start to goal
```

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