Date:22.10.25

TASK:11

Implementation of Robot traversal

CO1, CO2, CO3 S3

PROBLEM STATEMENT

The primary challenge in robot traversal is Path Planning. A mobile robot, starting at a defined initial position (S) and orientation, must autonomously find the optimal, collision-free path to a specified goal state (G) within a known or partially known environment that contains static obstacles.

The problem is formally defined as: Given a 2D environment represented as a grid or graph with fixed obstacles, determine the sequence of movements and turns (operators) for a robot to move from S to G such that the path length is minimized and all obstacles are avoided.

AIM

The aim of this project is to design, implement, and analyze a path planning algorithm that enables an autonomous mobile robot to successfully navigate a workspace from a starting point to an endpoint while avoiding obstacles.

OBJECTIVE

To model the robot's workspace as a configuration space (C-space), typically a 2D grid map, where obstacles are clearly defined.

To implement a graph-based search algorithm (e.g., A* Search, Breadth-First Search) to find the shortest path between the start and goal nodes.

To ensure the algorithm generates a path that is guaranteed to be collision-free.

To visualize the robot's movement and the resultant optimal path on the map.

DESCRIPTION

1. **Environment Setup: The environment is represented as a grid map where each cell is a state. Cells are marked as either walkable (0) or obstacle (1).**
2. **Robot State: The robot's state is defined by its (x, y) coordinates on the grid.**
3. **Operators/Actions: Possible actions include moving one step Up, Down, Left, Right, and optionally, diagonal movements, each having a specific cost.**
4. **Path Planning Algorithm: The A* Search Algorithm is employed, which is an informed search technique. It finds the shortest path by evaluating the cost of a node n using the function:**

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

   - **$g(n)$: The actual cost from the start node to node n.**
   - **$h(n)$: The estimated heuristic cost from node n to the goal node (e.g., Euclidean or Manhattan distance).**

By prioritizing nodes with the lowest $f(n)$ value, the A* algorithm efficiently explores the map and determines the optimal path.

ALGORITHM

Step	Description
1. Initialization	Create an Open List (priority queue) containing only the start node S. Create a Closed List (empty). Set $g(S)=0$ and calculate $h(S)$. Set $f(S) = g(S) + h(S)$.
2. Loop	While the Open List is not empty:
3. Node Selection	Select the node n with the lowest $f(n)$ value from the Open List and move it to the Closed List.
4. Goal Check	If n is the Goal Node G, terminate and reconstruct the path from G back to S using parent pointers.

Step	Description
5. Neighbor Expansion	**For each neighbor n' of node n:**
6. Check Obstacle/Closed	**If n' is an obstacle or is already in the Closed List, skip it.**
7. Calculate Cost	**Calculate the tentative g score for n': $g_{tentative}(n') = g(n) + \text{cost}(n, n')$.**
8. Update/Add	**If $g_{tentative}(n')$ is less than the current $g(n')$ or if n' is not in the Open List, update its parent to n, update $g(n')$ to $g_{tentative}(n')$, calculate $f(n')$, and add/update n' in the Open List.**

PROGRAM

```
import heapq

import math


# 1. Define Node structure

class Node:

    def __init__(self, position, g=0, h=0, f=0, parent=None):

        self.position = position  # (x, y)

        self.g = g  # Cost from start

        self.h = h  # Heuristic to goal

        self.f = f  # Total cost

        self.parent = parent


    def __lt__(self, other):  # For heap comparison

        return self.f < other.f
```

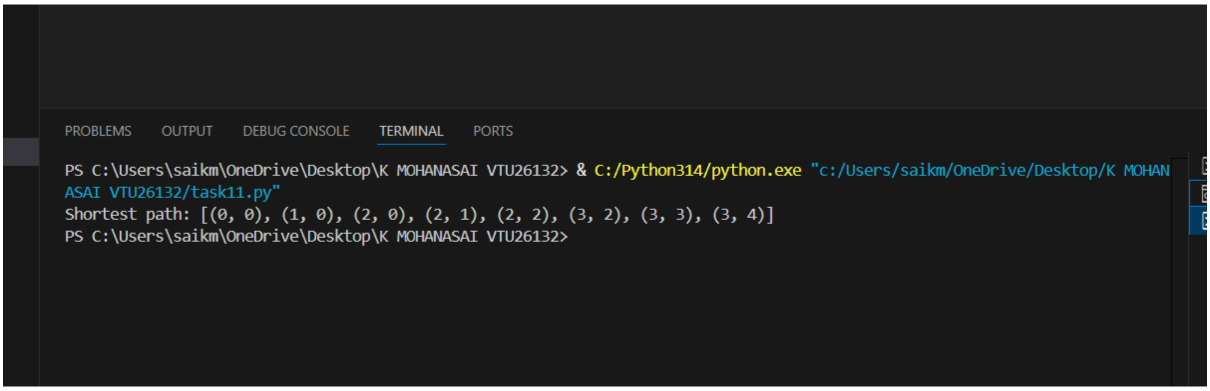
```python
# 2. Function: calculate_heuristic (Euclidean distance)
def calculate_heuristic(pos1, pos2):
    return math.sqrt((pos1[0] - pos2[0])**2 + (pos1[1] - pos2[1])**2)


# 3. Function: get_valid_neighbors
def get_valid_neighbors(grid, current_pos):
    neighbors = []
    rows, cols = len(grid), len(grid[0])
    directions = [(0,1), (1,0), (0,-1), (-1,0)]  # 4-directional movement

    for dx, dy in directions:
        nx, ny = current_pos[0] + dx, current_pos[1] + dy
        if 0 <= nx < rows and 0 <= ny < cols and grid[nx][ny] == 0:
            neighbors.append((nx, ny))
    return neighbors


# Helper: reconstruct path
def reconstruct_path(node):
    path = []
    while node:
```

```
        path.append(node.position)
        node = node.parent
    return path[::-1]  # reverse path


# 4. Main A* function
def A_star(grid, start, goal):
    start_node = Node(start, g=0, h=calculate_heuristic(start, goal))
    start_node.f = start_node.g + start_node.h

    open_list = []
    heapq.heappush(open_list, start_node)
    closed_list = set()

    while open_list:
        current_node = heapq.heappop(open_list)

        if current_node.position == goal:
            return reconstruct_path(current_node)

        closed_list.add(current_node.position)

        for neighbor_pos in get_valid_neighbors(grid, current_node.position):
            if neighbor_pos in closed_list:
```

```
                continue

            g_tentative = current_node.g + 1
            h = calculate_heuristic(neighbor_pos, goal)
            f = g_tentative + h

            neighbor_node = Node(neighbor_pos, g_tentative, h, f, 
current_node)
            heapq.heappush(open_list, neighbor_node)

    return None  # No path found


# Example usage
if __name__ == "__main__":
    grid = [
        [0, 1, 0, 0, 0],
        [0, 1, 0, 1, 0],
        [0, 0, 0, 1, 0],
        [1, 0, 0, 0, 0]
    ]
    start = (0, 0)
    goal = (3, 4)
```

```python
path = A_star(grid, start, goal)

print("Shortest path:", path)
```

OUTPUT



CONCLUSION

This foundational work in path planning is crucial for the development of real-world mobile robotics applications, such as autonomous warehouse vehicles, search and rescue robots, and planetary rovers. Future work could involve incorporating dynamic obstacle avoidance or using a more complex motion control model.