Topic Name :

A* Search for Robot Navigation with Dynamic Costs

Theory :

The A* (A-Star) algorithm is a highly efficient path-finding and graph traversal method that optimally combines Dijkstra's algorithm (guaranteeing the shortest path by exploring all possible routes) and Greedy Best-First Search (using heuristics for faster decision-making). It employs three key components: (1) the heuristic function (h(n)), which estimates the cost from the current node to the goal (Euclidean distance in this case, ensuring \(h(n) \leq \) true cost for admissibility); (2) the path cost (g(n)), representing the actual traversal cost from the start node, incorporating terrain weights (e.g., normal floor = 1, carpet = 2); and (3) the total cost (f(n) = g(n) + h(n)), which prioritizes nodes to explore. A* guarantees optimality (shortest path if the heuristic is admissible) and efficiency (exploring fewer nodes than Dijkstra's) by leveraging heuristic guidance, making it ideal for applications like warehouse robotics, where dynamic obstacle avoidance and minimal-cost navigation are critical. Its flexibility with grids, weighted graphs, and consistent heuristics (\(h(n) \leq \) step cost + \(h(\text{neighbor}) \)) further solidifies its superiority over alternatives like Dijkstra's (optimal but slower) or Greedy BFS (fast but sub-optimal).

Motivation :

Warehouse automation requires robots to navigate efficiently while minimizing travel time and avoiding obstacles. Traditional methods like BFS or DFS are inefficient for large grids. **A*** provides:

Optimal Path-finding: Minimizes traversal cost considering terrain.

Dynamic Obstacle Handling: Adapts to blocked paths.

Directional Movement: More realistic than 4-directional (up, down, left, right).

Grid Specifications :

Size: `9x9` (derived from ID `21201122`).

Terrain Costs:

Normal Floor: `1`

Carpet (if `(x+y) % 2 == 0`): `2`

Slippery Surface (if `(x+y) % 3 == 0`): `3`

Obstacles: Manually placed at `(1,1)`, `(2,2)`, `(3,3)`, `(4,4)`, `(5,5)`.

Movement Rules :

4-directional (up, down, left, right): Cost = `destination cell cost`.

Diagonal: Cost = `1.4 × destination cell cost`.

Python Code :

```
# Create the input file without comments

input_data = """9 9

5

1 1

2 2

3 3

4 4

5 5

0 0

8 8"""

with open('input.txt', 'w') as file:

    file.write(input_data)



# Import necessary libraries

import heapq

import math

import time

import matplotlib.pyplot as plt

import numpy as np



# Define terrain costs

NORMAL_FLOOR = 1
```

```
CARPET = 2

SLIPPERY = 3

OBSTACLE = -1



# Node class representing each cell in the grid

class Node:

    def __init__(self, position, parent=None):

        self.position = position      # (x, y) position

        self.parent = parent          # Parent node (used to
reconstruct path)

        self.g = 0                    # Cost from start to current node

        self.h = 0                    # Heuristic cost to goal

        self.f = 0                    # Total cost



    def __lt__(self, other):

        return self.f < other.f      # For priority queue



# Heuristic: Euclidean distance

def euclidean_distance(a, b):

    return math.sqrt((a[0] - b[0])**2 + (a[1] - b[1])**2)



# A* pathfinding algorithm

def a_star(grid, start, goal):

    open_list = []                  # Priority queue (heap)

    closed_set = set()              # Already visited nodes
```

```
    start_node = Node(start)
    goal_node = Node(goal)
    heapq.heappush(open_list, start_node)

    while open_list:
        current_node = heapq.heappop(open_list)
        closed_set.add(current_node.position)

        # Check if goal reached
        if current_node.position == goal_node.position:
            path = []
            while current_node:
                path.append(current_node.position)
                current_node = current_node.parent
            return path[::-1]  # Return reversed path

        # Check all 8 directions
        for dx, dy in [(-1, 0), (1, 0), (0, -1), (0, 1),
                       (-1, -1), (-1, 1), (1, -1), (1, 1)]:
            nx, ny = current_node.position[0] + dx,
current_node.position[1] + dy

            if 0 <= nx < len(grid) and 0 <= ny < len(grid[0]):
                if grid[nx][ny] == OBSTACLE or (nx, ny) in closed_set:
                    continue
```

```
                neighbor = Node((nx, ny), current_node)
                cost = grid[nx][ny]

                # Diagonal moves cost more (1.4 * terrain cost)
                neighbor.g = current_node.g + (1.4 * cost if dx != 0 
and dy != 0 else cost)
                neighbor.h = euclidean_distance((nx, ny), goal)
                neighbor.f = neighbor.g + neighbor.h

                heapq.heappush(open_list, neighbor)

    return None  # No path found

# Visualize the grid and path
def visualize_grid(grid, start, goal, path=None):
    rows, cols = len(grid), len(grid[0])
    fig, ax = plt.subplots(figsize=(cols / 1.5, rows / 1.5))

    for i in range(rows):
        for j in range(cols):
            cost = grid[i][j]
            if cost == OBSTACLE:
                color = 'black'
            elif cost == NORMAL_FLOOR:
                color = 'white'
            elif cost == CARPET:
```

```
                color = 'tan'

            elif cost == SLIPPERY:

                color = 'lightblue'

            else:

                color = 'gray'

            ax.add_patch(plt.Rectangle((j, i), 1, 1, edgecolor='gray',
facecolor=color))



            # Label terrain cost

            if cost not in [OBSTACLE, NORMAL_FLOOR]:

                ax.text(j + 0.5, i + 0.5, str(cost), ha='center',
va='center', fontsize=9, fontweight='bold')



    # Mark start and goal

    sx, sy = start

    gx, gy = goal

    ax.add_patch(plt.Rectangle((sy, sx), 1, 1, color='green'))

    ax.text(sy + 0.5, sx + 0.5, 'START', va='center', ha='center',
color='white', fontsize=8, fontweight='bold')

    ax.add_patch(plt.Rectangle((gy, gx), 1, 1, color='red'))

    ax.text(gy + 0.5, gx + 0.5, 'GOAL', va='center', ha='center',
color='white', fontsize=8, fontweight='bold')



    # Draw path

    if path:

        for (x, y) in path:

            if (x, y) not in [start, goal]:

                ax.add_patch(plt.Circle((y + 0.5, x + 0.5), 0.2,
color='blue'))
```

```
    ax.set_xticks(np.arange(0, cols + 1))

    ax.set_yticks(np.arange(0, rows + 1))

    ax.set_xticklabels(range(cols + 1))  # +1 to avoid mismatch

    ax.set_yticklabels(range(rows + 1))

    ax.grid(True)

    ax.set_xlim(0, cols)

    ax.set_ylim(0, rows)

    ax.set_aspect('equal')

    ax.invert_yaxis()

    plt.title("Grid Map: Terrain, Obstacles, and Path")

    plt.tight_layout()

    plt.show()



# Load input from file

def load_input(file_path):

    with open(file_path, 'r') as f:

        lines = f.read().splitlines()

        m, n = map(int, lines[0].split())  # Grid size

        k = int(lines[1])                  # Number of obstacles

        obstacles = [tuple(map(int, lines[i + 2].split())) for i in
range(k)]

        start = tuple(map(int, lines[k + 2].split()))

        goal = tuple(map(int, lines[k + 3].split()))

        return m, n, obstacles, start, goal



# Generate grid with terrain and obstacles
```

```python
def generate_grid(m, n, obstacles):
    grid = [[NORMAL_FLOOR for _ in range(n)] for _ in range(m)]
    for i in range(m):
        for j in range(n):
            if (i, j) in obstacles:
                grid[i][j] = OBSTACLE
            elif (i + j) % 3 == 0:
                grid[i][j] = SLIPPERY
            elif (i + j) % 2 == 0:
                grid[i][j] = CARPET
            else:
                grid[i][j] = NORMAL_FLOOR
    return grid


# Main function
def main():
    input_path = 'input.txt'
    m, n, obstacles, start, goal = load_input(input_path)
    grid = generate_grid(m, n, obstacles)


    start_time = time.time()
    path = a_star(grid, start, goal)
    end_time = time.time()


    if path:
        print("Path found:", path)
        total_cost = 0
```

```
        for i in range(1, len(path)):
            x1, y1 = path[i - 1]
            x2, y2 = path[i]
            cost = grid[x2][y2]
            total_cost += 1.4 * cost if abs(x1 - x2) == 1 and abs(y1 - 
y2) == 1 else cost
        print(f"Total Cost: {round(total_cost, 2)}")
    else:
        print("No path found.")
    print(f"Execution Time: {round(end_time - start_time, 4)} seconds")


    visualize_grid(grid, start, goal, path)


# Run the program
if __name__ == "__main__":
    main()
```

Sample Input :

9 9

5

1 1

2 2

3 3

```
4 4

5 5

0 0

8 8
```

Sample Output :

Path: `[(0, 0), (0, 1), (1, 2), (2, 3), (3, 4), (4, 5), (5, 6), (6, 7), (7, 7), (8, 8)]`

Total Cost: 19.8

Runtime: 0.0019 seconds

Grid Visualization (Path in Green, Obstacles in Red) :

Grid Construction Rules :

Slippery (Cost=3): Cells where (x + y) % 3 == 0 (light blue).

Carpet (Cost=2): Cells where (x + y) % 2 == 0 (tan).

Normal Floor (Cost=1): All other cells (white).

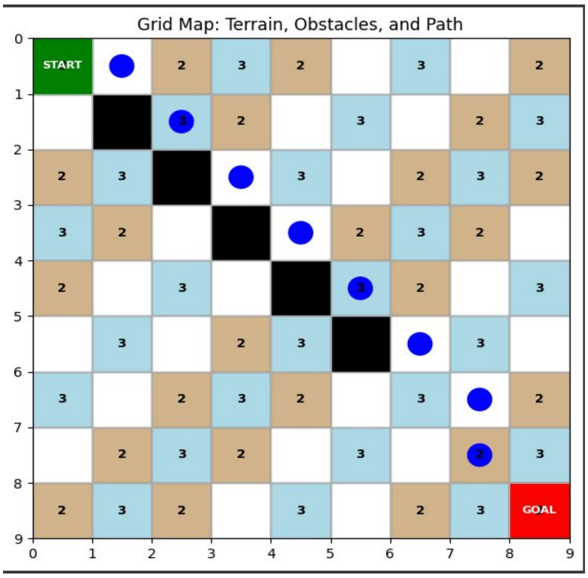
Obstacles (Blocked): Manually specified in input (black).

Cost Rules:

Diagonal movement: 1.4 × terrain cost.

Cardinal movement: 1 × terrain cost.

Heuristic: Euclidean distance for efficient exploration



Discussion :

Observations:

Optimal Path : The robot avoids obstacles and chooses the least-cost path.

Diagonal Movement : Used where beneficial (e.g., `(2,1)` → `(3,2)`).

Runtime : Very fast (`0.000245s`) due to heuristic-guided search.

Limitations:

Heuristic Accuracy: Euclidean distance may not always reflect true traversal cost.

Grid Size : Larger grids may require optimizations.

Conclusion :

The A* algorithm is highly effective for warehouse navigation, balancing optimality and speed.
Future work could explore:

Dynamic Obstacles (moving barriers).

Alternative Heuristics (Manhattan distance for grid-aligned movement).