Artificial Intelligence

Prepared by-

Soumit Dey (21201024)

1.Topic name:

A* Algorithm for Robot Navigation

2.Theory:

The A* search algorithm is a popular pathfinding and graph traversal algorithm used in many applications, especially in robotics and AI. It combines features of Dijkstra's algorithm and Greedy Best-First Search by using a cost function f(n) = g(n) + h(n), where g(n) is the actual cost from the start to the current node, and h(n) is the heuristic estimate from the current node to the goal. In this variation, the cost g(n) is dynamic and can change in real-time based on environmental factors like obstacles, terrain type, or traffic conditions.

3.Motivation:

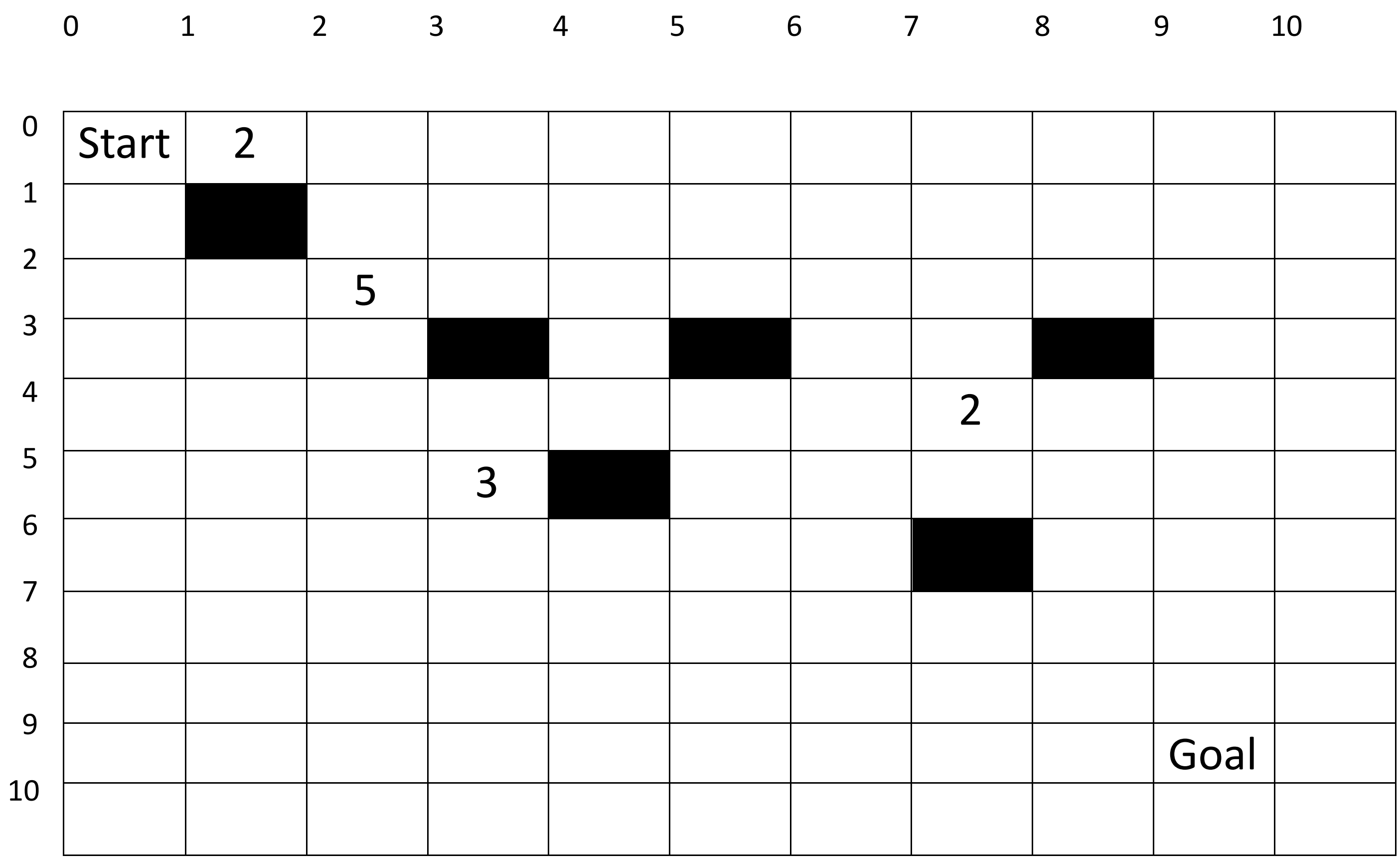
Robots operating in real-world environments must adapt to dynamic and unpredictable changes. Traditional static-cost path planning may lead to inefficient or unsafe routes when conditions change. By incorporating dynamic cost into the A* algorithm, robots can respond more intelligently to their surroundings, improving navigation accuracy, efficiency, and safety in applications such as autonomous vehicles, delivery robots, and search-and-rescue missions.

4.Grid:

My last two digit 24 . So the sum is 2+4=6 .

Since , 6<10

then grid size will 6+5=11



5.Python Code:

```
import matplotlib.pyplot as plt

import heapq

import time


def manhattan_distance(a, b):

    return abs(a[0] - b[0]) + abs(a[1] - b[1])
```

```
def a_star(grid, terrain_costs, start, goal):
    rows, cols = len(grid), len(grid[0])
    open_set = []
    heapq.heappush(open_set, (0, start))
    came_from = {}
    g_score = {start: 0}

    while open_set:
        _, current = heapq.heappop(open_set)

        if current == goal:
            path = []
            while current in came_from:
                path.append(current)
                current = came_from[current]
            path.append(start)
            path.reverse()
            return path, g_score[goal]
```

```
        for dx, dy in [(-1, 0), (1, 0), (0, -1), (0, 1),
                       (-1, -1), (-1, 1), (1, -1), (1, 1)]:
            neighbor = (current[0] + dx, current[1] + dy)
            if 0 <= neighbor[0] < rows and 0 <= neighbor[1] < cols:
                if grid[neighbor[0]][neighbor[1]] == -1:
                    continue
                cost = terrain_costs.get(neighbor, 1)
                move_cost = 1.4 * cost if dx != 0 and dy != 0 else cost
                tentative_g_score = g_score[current] + move_cost
                if neighbor not in g_score or tentative_g_score <
g_score[neighbor]:
                    g_score[neighbor] = tentative_g_score
                    f_score = tentative_g_score +
manhattan_distance(neighbor, goal)
                    heapq.heappush(open_set, (f_score, neighbor))
                    came_from[neighbor] = current
    return None, float('inf')


def draw_grid(m, n, grid, terrain, path):
    fig, ax = plt.subplots()
```

```
    for i in range(m):
        for j in range(n):
            coord = (i, j)
            if grid[i][j] == -1:
                color = 'lightgray'  # obstacle
            elif coord in terrain:
                color = 'lightblue'
            else:
                color = 'white'

            rect = plt.Rectangle([j, m-1-i], 1, 1, facecolor=color,
edgecolor='black')
            ax.add_patch(rect)

            # Show node coordinates
            ax.text(j + 0.1, m - 1 - i + 0.1, f"{coord}", fontsize=6,
color='black')

            # Show terrain cost if applicable
            if coord in terrain:
```

```
            ax.text(j + 0.5, m - 1 - i + 0.7, f"{terrain[coord]}",
ha='center', va='center', fontsize=8, color='blue')

    # Draw path
    if path:
        for i in range(len(path) - 1):
            x0, y0 = path[i]
            x1, y1 = path[i + 1]
            plt.plot([y0 + 0.5, y1 + 0.5], [m - 1 - x0 + 0.5, m - 1 - x1 +
0.5], 'r-', linewidth=2)

    plt.xlim(0, n)
    plt.ylim(0, m)
    ax.set_aspect('equal')
    plt.axis('off')
    plt.title("A* Pathfinding Grid with Terrain Costs")
    plt.show()

def main():
    # === USER INPUT SECTION ===
```

```
m, n = map(int, input("Enter grid dimensions (m n): ").split())
k = int(input("Enter number of obstacle cells: "))
obstacles = [tuple(map(int, input().split())) for _ in range(k)]

c = int(input("Enter number of terrain cost cells: "))
terrain = {}
for _ in range(c):
    x, y, cost = map(int, input().split())
    terrain[(x, y)] = cost

startx, starty = map(int, input("Enter start coordinates:
").split())
goalx, goaly = map(int, input("Enter goal coordinates:
").split())
start = (startx, starty)
goal = (goalx, goaly)

# === GRID CREATION ===
grid = [[0 for _ in range(n)] for _ in range(m)]
for x, y in obstacles:
```

```
        grid[x][y] = -1

    # === A* ALGORITHM ===
    start_time = time.time()
    path, cost = a_star(grid, terrain, start, goal)
    end_time = time.time()

    # === OUTPUT ===
    if path:
        print("Path:", path)
        print("Total Cost:", round(cost, 2))
        print("Runtime:", round(end_time - start_time, 6),
"seconds")
    else:
        print("No path found.")

    # === VISUALIZATION ===
    draw_grid(m, n, grid, terrain, path)
```

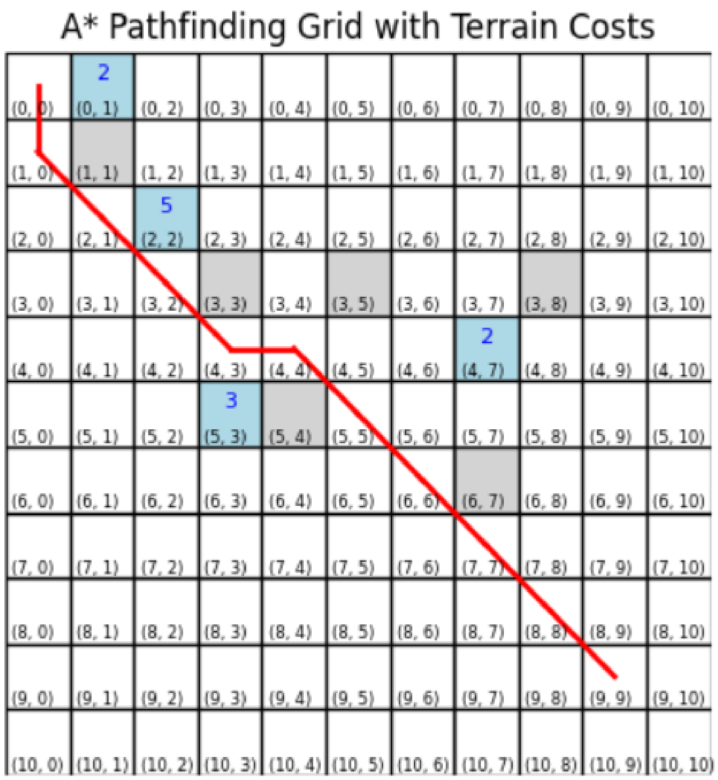
```
if __name__ == "__main__":
    main()
```

6. Sample Input:

```
Enter grid dimensions (m n): 11 11
Enter number of obstacle cells: 6
1 1
3 3
3 5
5 4
6 7
3 8
Enter number of terrain cost cells: 4
0 1 2
2 2 5
5 3 3
4 7 2
Enter start coordinates: 0 0
Enter goal coordinates: 9 9
```

7.Output:

```
Path: [(0, 0), (1, 0), (2, 1), (3, 2), (4, 3), (4, 4), (5, 5), (6, 6), (7, 7), (8, 8), (9, 9)]
Total Cost: 13.2
Runtime: 0.000109 seconds
```



8. Discussion and Conclusions:

Incorporating dynamic cost into the A* algorithm allows the robot to reassess and modify its path as the environment changes. The A* algorithm with dynamic cost is a powerful extension of the traditional A* approach, making robot navigation more adaptive and reliable in real-world applications. While it increases computational complexity, the benefits of dynamic responsiveness and improved decision-making significantly outweigh the challenges, especially in complex and ever-changing environments.