GROUP – 17

Problem Statement – 2

Table of Contents

Analysis on Problem Statement

PEAS of Problem Statement

Complexity Analysis of given Algorithms

Output

Code

Analysis on Problem Statement

The task is to determine the safest and shortest path from an office (start) to a home (goal) in a city grid map. The grid features different types of cells, including buildings, roadblocks, and open paths. Your goal is to navigate this grid, avoiding obstacles and minimizing both travel distance and safety penalties.

Key Constraints

1. **Movement Restrictions**:
   - **Direct Movement**: You can move up, down, left, or right. Diagonal movements are only allowed if no direct paths are available, with a penalty of 3 points.
   - **Blockades**: You cannot traverse cells occupied by buildings.
   - **Obstacle Penalties**:
     - **Roadblocks**: Movement adjacent to roadblocks incurs a penalty of 3 points.
     - **Buildings**: Movement adjacent to buildings incurs a penalty of 5 points.
2. **Performance Measure**:
   - **Safety**: The path should incur the lowest possible penalties from adjacent buildings and roadblocks. Safety is prioritized over distance.
   - **Shortest Path**: Among the safest paths, the one with the fewest number of squares traveled should be chosen.
3. **Environment**:
   - **City Grid**: Represented as a matrix with different cell types:
     - **Open Cell**: Traversable with no penalties.
     - **Building ('B')**: Impassable and adjacent cells incur a penalty.
     - **Roadblock ('R')**: Impassable and adjacent cells incur a penalty.
   - **Start and Goal**: Defined by coordinates in the grid for your office and home.
4. **Given Algorithms**:
   - **Recursive Best First Search (RBFS)**: A heuristic-based search algorithm that explores the most promising paths first. It balances safety and distance to find the optimal path. It is simple recursive algorithm that attempts to mimic the operation of standard best – first search, but using only linear space.
   - **Uniform Cost Search (UCS)**: Expands paths in order of increasing cost. It ensures finding the least costly path while considering safety and distance.
5. **Cost Calculation**:
   - **Movement Cost**: Generally 1 point per valid move.
   - **Penalties**: Additional points are added if the path is adjacent to buildings or roadblocks.

PEAS of Problem Statement

Performance Measure:

- **Total Path Cost**: Minimize the total penalty points and number of squares in the path.
- **Path Quality**: Balance safety (penalty minimization) and shortest distance.

Environment:

- **City Grid Map**: Contains cells representing open paths, buildings, and roadblocks with varying penalties.

Actuators:

- **Movement**: Directions include up, down, left, right, and optionally diagonal (with a penalty if used).

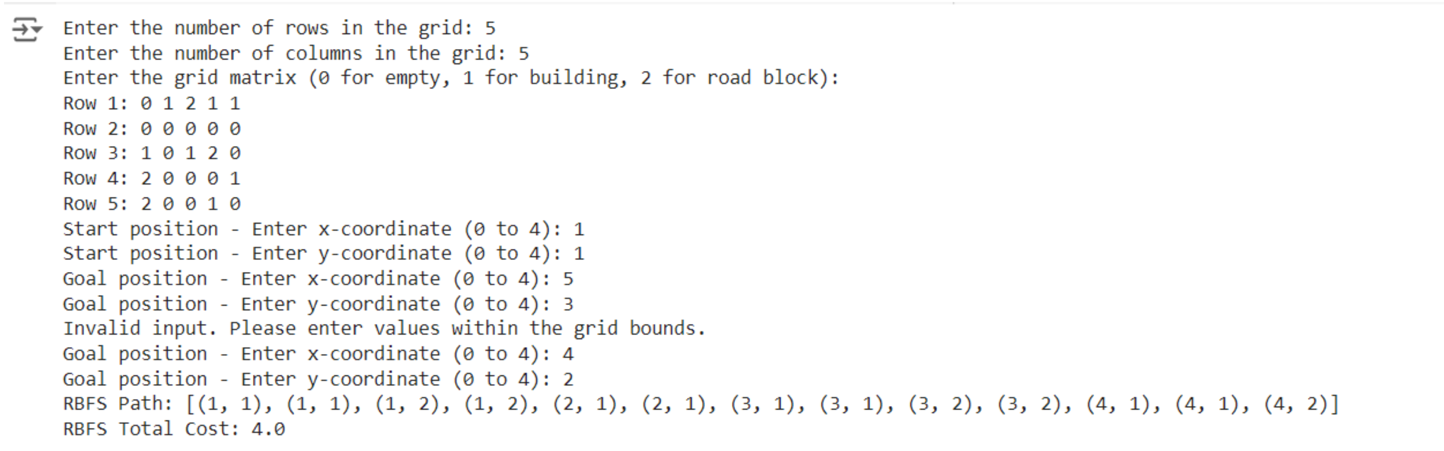
Sensor:

- **Cell Type Detection**: Identify cell types (open, building, roadblock) and adjacent cells to compute penalties.

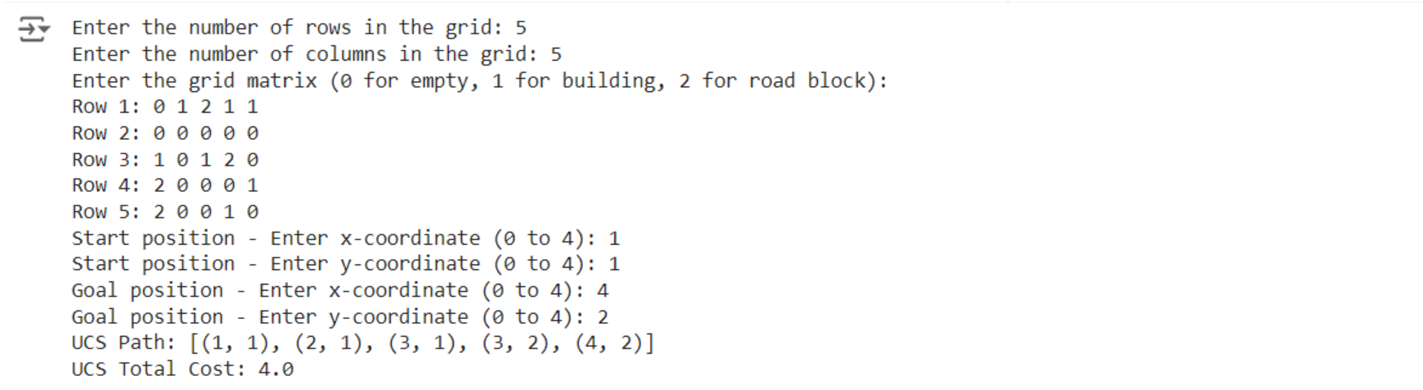
Complexity Analysis

- **Time Complexity**:
  - **RBFS**: Generally O(b^m), where b is the branching factor and m is the maximum depth of the search.
  - **UCS**: Generally O(b^d), where b is the branching factor and d is the depth of the solution.
- **Space Complexity**:
  - **RBFS**: Typically O(b * m), where b is the branching factor and m is the maximum depth.
  - **UCS**: Typically O(b * d), where b is the branching factor and d is the depth of the solution.

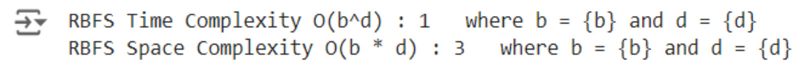
Output



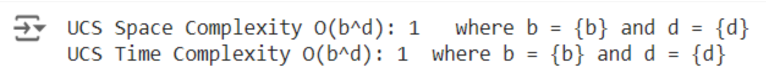
```
Enter the number of rows in the grid: 5
Enter the number of columns in the grid: 5
Enter the grid matrix (0 for empty, 1 for building, 2 for road block):
Row 1: 0 1 2 1 1
Row 2: 0 0 0 0 0
Row 3: 1 0 1 2 0
Row 4: 2 0 0 0 1
Row 5: 2 0 0 1 0
Start position - Enter x-coordinate (0 to 4): 1
Start position - Enter y-coordinate (0 to 4): 1
Goal position - Enter x-coordinate (0 to 4): 5
Goal position - Enter y-coordinate (0 to 4): 3
Invalid input. Please enter values within the grid bounds.
Goal position - Enter x-coordinate (0 to 4): 4
Goal position - Enter y-coordinate (0 to 4): 2
RBFS Path: [(1, 1), (1, 1), (1, 2), (1, 2), (2, 1), (2, 1), (3, 1), (3, 1), (3, 2), (3, 2), (4, 1), (4, 1), (4, 2)]
RBFS Total Cost: 4.0
```



```
Enter the number of rows in the grid: 5
Enter the number of columns in the grid: 5
Enter the grid matrix (0 for empty, 1 for building, 2 for road block):
Row 1: 0 1 2 1 1
Row 2: 0 0 0 0 0
Row 3: 1 0 1 2 0
Row 4: 2 0 0 0 1
Row 5: 2 0 0 1 0
Start position - Enter x-coordinate (0 to 4): 1
Start position - Enter y-coordinate (0 to 4): 1
Goal position - Enter x-coordinate (0 to 4): 4
Goal position - Enter y-coordinate (0 to 4): 2
UCS Path: [(1, 1), (2, 1), (3, 1), (3, 2), (4, 2)]
UCS Total Cost: 4.0
```



```
RBFS Time Complexity O(b^d) : 1   where b = {b} and d = {d}
RBFS Space Complexity O(b * d) : 3   where b = {b} and d = {d}
```



```
UCS Space Complexity O(b^d): 1   where b = {b} and d = {d}
UCS Time Complexity O(b^d): 1  where b = {b} and d = {d}
```

Artificial and Computational Intelligence Assignment 1

Problem solving by Uninformed & Informed Search

List only the BITS (Name) of active contributors in this assignment:

1.AADITI MILIND PIMPLEY

2.VIKAS MOHAN THAPLIYAL

3.SUDHAGANI VANDANA

4.THRUSHIKA.S

5.SONIA BENNY THOMAS

Things to follow

1. **Use appropriate data structures to represent the graph and the path using python libraries**
2. **Provide proper documentation**
3. **Find the path and print it**

Coding begins here

1. Define the environment in the following block

List the PEAS decription of the problem here in this markdown block

PEAS Description

- **Performance Measure: The agent should find the shortest and safest path from the office to home while considering penalties for diagonal movement, proximity to buildings, and road blocks.**
- **Environment: The environment is a grid representing a city with buildings, roadblocks, and blank spaces. The grid also includes the office and home locations.**
- **Actuator: The actuators are the movements in the grid: moving up, down, left, right, or diagonally with penalties.**
- **Sensor: The sensors detect the type of grid cell (empty, building, roadblock) and the current location in the grid.**

Design the agent as PSA Agent(Problem Solving Agent) Clear Initial data structures to define the graph and variable declarations is expected IMPORTATANT: Write distinct code block as below

In [1]:

```python
#Code Block : Set Initial State (Must handle dynamic inputs)
import numpy as np

def initialize_state():
    while True:
        try:
            rows = int(input("Enter the number of rows in the grid: "))
            cols = int(input("Enter the number of columns in the grid: "))

            if rows <= 0 or cols <= 0:
                print("Grid dimensions must be positive integers.")
                continue
            grid = np.zeros((rows, cols), dtype=int)

            print("Enter the grid matrix (0 for empty, 1 for building, 2 for road block)
:")
            for i in range(rows):
                row = list(map(int, input(f"Row {i + 1}: ").strip().split()))
                if len(row) != cols:
                    print("The number of columns does not match the grid width.")
                    return None
                grid[i] = row

            return grid
        except ValueError:
            print("Invalid input. Please enter integers only.")
```

In [2]:

```
#Code Block : Set the matrix for transition & cost (as relevant for the given problem)
def set_transition_and_cost_matrix(grid):
    rows, cols = grid.shape
    cost_matrix = np.full((rows, cols), np.inf)

    for i in range(rows):
        for j in range(cols):
            if grid[i, j] == 1 or grid[i, j] == 2:
                cost_matrix[i, j] = np.inf
            else:
                cost_matrix[i, j] = 1  # Default cost for moving to an empty cell

    return cost_matrix
```

In [3]:

```
#Code Block : Write function to design the Transition Model/Successor function. Ideally t
his would be called while search algorithms are implemented
def get_successors(state, grid, cost_matrix):
    rows, cols = grid.shape
    x, y = state
    successors = []
    moves = [(-1, 0), (1, 0), (0, -1), (0, 1)]  # Up, Down, Left, Right

    for dx, dy in moves:
        nx, ny = x + dx, y + dy
        if 0 <= nx < rows and 0 <= ny < cols and grid[nx, ny] != 1 and grid[nx, ny] !=
2:
            cost = cost_matrix[nx, ny]
            if (nx, ny) != (x, y):
                successors.append(((nx, ny), cost))

    return successors
```

In [4]:

```
#Code block : Write fucntion to handle goal test (Must handle dynamic inputs). Ideally th
is would be called while search algorithms are implemented
def is_goal_state(state, goal):
    return state == goal
```

2. Definition of Algorithm 1 (Mention the Name of the algorithm here)

In [5]:

```
#Code Block : Function for algorithm 1 implementation
import heapq

def rbfs(start, goal, grid, cost_matrix):
    def rbfs_recursive(node, g, f, heap, came_from):
        if is_goal_state(node, goal):
            return [node], g
        successors = get_successors(node, grid, cost_matrix)
        for (succ, cost) in successors:
            h = np.abs(succ[0] - goal[0]) + np.abs(succ[1] - goal[1])
            f = g + cost + h
            heapq.heappush(heap, (f, succ, g + cost))

        if not heap:
            return [], float('inf')

        _, current, g = heapq.heappop(heap)
        path, cost = rbfs_recursive(current, g, float('inf'), heap, came_from)
        return [node] + path, cost

    heap = []
    heapq.heappush(heap, (np.abs(start[0] - goal[0]) + np.abs(start[1] - goal[1]), start
, 0))
    path, cost = rbfs_recursive(start, 0, float('inf'), heap, {})
    return path, cost
```

3. Definition of Algorithm 2 (Mention the Name of the algorithm here)

In [6]:

```
#Code Block : Function for algorithm 2 implementation
def ucs(start, goal, grid, cost_matrix):
    from heapq import heappop, heappush

    rows, cols = grid.shape
    open_set = []
    heappush(open_set, (0, start))
    came_from = {}
    g_score = {start: 0}

    while open_set:
        current_cost, current = heappop(open_set)

        if is_goal_state(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 (neighbor, move_cost) in get_successors(current, grid, cost_matrix):
            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
                heappush(open_set, (f_score, neighbor))
                came_from[neighbor] = current

    return [], float('inf')
```

DYNAMIC INPUT

IMPORTANT : Dynamic Input must be got in this section. Display the possible states to choose from: This is applicable for all the relevent problems as mentioned in the question.

In [7]:

```
#Code Block : Function & call to get inputs (start/end state)
def get_dynamic_state(prompt, grid):
    while True:
        try:
            x = int(input(f"{prompt} - Enter x-coordinate (0 to {grid.shape[0] - 1}): ")
)
            y = int(input(f"{prompt} - Enter y-coordinate (0 to {grid.shape[1] - 1}): ")
)
            if 0 <= x < grid.shape[0] and 0 <= y < grid.shape[1]:
                if grid[x, y] != 1 and grid[x, y] != 2:
                    return (x, y)
                else:
                    print("Invalid position. The cell is blocked or occupied by a buildin
g.")
            else:
                print("Invalid input. Please enter values within the grid bounds.")
        except ValueError:
            print("Invalid input. Please enter integers only.")
```

4. Calling the search algorithms

(For bidirectional search in below sections first part can be used as per Hint provided. Under second section other combinations as per Hint or your choice of 2 algorithms can be called .As an analyst suggest suitable approximation in the comparitive analysis section)

In [12]:

```
#Invoke algorithm 1 (Should Print the solution, path, cost etc., (As mentioned in the pro
blem))
# Main execution
if __name__ == "__main__":
    # Initialize the grid
    grid = initialize_state()
if grid is not None:
    start = get_dynamic_state("Start position", grid)
    goal = get_dynamic_state("Goal position", grid)
    cost_matrix = set_transition_and_cost_matrix(grid)
    path, cost = rbfs(start, goal, grid, cost_matrix)
    print("RBFS Path:", path)
    print("RBFS Total Cost:", cost)
```

```
Enter the number of rows in the grid: 5
Enter the number of columns in the grid: 5
Enter the grid matrix (0 for empty, 1 for building, 2 for road block):
Row 1: 0 1 2 1 1
Row 2: 0 0 0 0 0
Row 3: 1 0 1 2 0
Row 4: 2 0 0 0 1
Row 5: 2 0 0 1 0
Start position - Enter x-coordinate (0 to 4): 1
Start position - Enter y-coordinate (0 to 4): 1
Goal position - Enter x-coordinate (0 to 4): 5
Goal position - Enter y-coordinate (0 to 4): 3
Invalid input. Please enter values within the grid bounds.
Goal position - Enter x-coordinate (0 to 4): 4
Goal position - Enter y-coordinate (0 to 4): 2
RBFS Path: [(1, 1), (1, 1), (1, 2), (1, 2), (2, 1), (2, 1), (3, 1), (3, 1), (3, 2), (3, 2)
, (4, 1), (4, 1), (4, 2)]
RBFS Total Cost: 4.0
```

In [13]:

```
#Invoke algorithm 2 (Should Print the solution, path, cost etc., (As mentioned in the pro
blem))
if __name__ == "__main__":
    # Initialize the grid
    grid = initialize_state()
if grid is not None:
    start = get_dynamic_state("Start position", grid)
    goal = get_dynamic_state("Goal position", grid)
    cost_matrix = set_transition_and_cost_matrix(grid)
    path, cost = ucs(start, goal, grid, cost_matrix)
    print("UCS Path:", path)
    print("UCS Total Cost:", cost)
```

```
Enter the number of rows in the grid: 5
Enter the number of columns in the grid: 5
Enter the grid matrix (0 for empty, 1 for building, 2 for road block):
Row 1: 0 1 2 1 1
Row 2: 0 0 0 0 0
Row 3: 1 0 1 2 0
Row 4: 2 0 0 0 1
Row 5: 2 0 0 1 0
Start position - Enter x-coordinate (0 to 4): 1
Start position - Enter y-coordinate (0 to 4): 1
Goal position - Enter x-coordinate (0 to 4): 4
Goal position - Enter y-coordinate (0 to 4): 2
UCS Path: [(1, 1), (2, 1), (3, 1), (3, 2), (4, 2)]
UCS Total Cost: 4.0
```

5. Comparitive Analysis

In [30]:

```
#Code Block : Print the Time & Space complexity of algorithm 1
# b = branching factor, d = depth of the solution

b = len(get_successors((0, 0), grid, set_transition_and_cost_matrix(grid)))  # Max branc
hing factor from the start node
d = np.max([np.abs(goal[0] - start[0]), np.abs(goal[1] - start[1])])  # Maximum depth in
the grid
m = grid.size  # Number of cells in the grid
Time_Complexity=pow(b,d)
Space_Complexity=b*d
print("RBFS Time Complexity O(b^d) :",Time_Complexity,"  where b = {b} and d = {d}")
print("RBFS Space Complexity O(b * d) :",Space_Complexity,"  where b = {b} and d = {d}")
```

```
RBFS Time Complexity O(b^d) : 1   where b = {b} and d = {d}
RBFS Space Complexity O(b * d) : 3   where b = {b} and d = {d}
```

In [31]:

```
#Code Block : Print the Time & Space complexity of algorithm 2
# b = branching factor, d = depth of the solution

b = len(get_successors((0, 0), grid, set_transition_and_cost_matrix(grid)))  # Max branc
hing factor from the start node
d = np.max([np.abs(goal[0] - start[0]), np.abs(goal[1] - start[1])])  # Maximum depth in
the grid
m = grid.size  # Number of cells in the grid
Complexity=pow(b,d)


print(f"UCS Space Complexity O(b^d):",Complexity,"  where b = {b} and d = {d}")
print(f"UCS Time Complexity O(b^d):",Complexity," where b = {b} and d = {d}")
```

```
UCS Space Complexity O(b^d): 1   where b = {b} and d = {d}
UCS Time Complexity O(b^d): 1  where b = {b} and d = {d}
```

6. Provide your comparitive analysis or findings in no more than 3 lines in below section

Comparison : Uniform Cost Search (UCS) ensures optimal solutions with O(b^d) time and space complexity due to storing all nodes. Recursive Best-First Search (RBFS) reduces space complexity to O(b*d) by only tracking the current path and recursion stack, though it can have similar time complexity due to extensive node exploration.