Indian Institute of Engineering Science & Technology, Shibpur
Department of Computer Science & Technology
Artificial Intelligence Laboratory 2025 CS 4271

ASSIGNMENT – 02
NAME - RAKSHA PAHARIYA
ENROLLMENT NO - 2021CSB029
https://colab.research.google.com/drive/193QXvrsS81FqGd8wQeOgTP32Wt92GWE6?usp=sharing

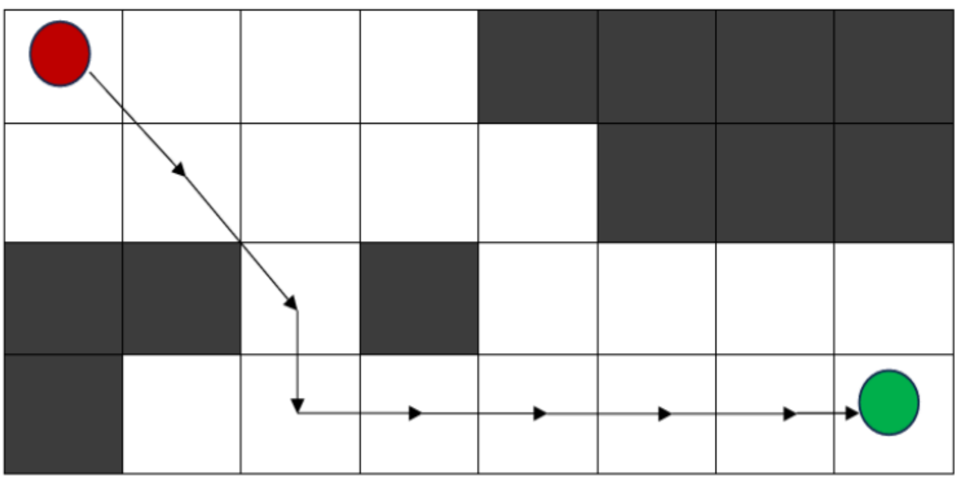
---

Question 1 : In a spatial context defined by a square grid featuring numerous obstacles, a task is presented wherein a starting cell, and a target cell are specified. The objective is to efficiently traverse from the starting cell to the target cell, optimizing for expeditious navigation. In this scenario, the A* Search algorithm proves instrumental.

The A* Search algorithm operates by meticulously selecting nodes within the grid, employing a parameter denoted as 'f.' This parameter, critical to the decision-making process, is the summation of two distinct parameters – 'g' and 'h.' At each iterative step, the algorithm strategically identifies the node with the lowest 'f' value and progresses the exploration accordingly. The allowed actions are: left, right, top, bottom, and diagonal.

The parameters 'g' and 'h' are delineated as follows:
- 'g': Represents the cumulative movement cost incurred in traversing the path from the designated starting point to the current square on the grid.
- 'h': Constitutes the estimated movement cost anticipated for the traversal from the current square on the grid to the specified destination, by using either Manhattan or Euclidean distance.

This element, often denoted as the heuristic, embodies an intelligent estimation. The A* Search algorithm, distinguished by its ability to efficiently find optimal or near optimal paths amidst obstacles, holds significant applicability in diverse domains such as robotics, gaming, and route planning.



Solution 1 :

```
import math
import heapq

def find_start_and_goal(matrix):
    """
    Find the start ('S') and goal ('G') positions in the matrix.
    """
    start = None
    goal = None
    for i in range(len(matrix)):
        for j in range(len(matrix[0])):
            if matrix[i][j] == 'S':
                start = (i, j)
            elif matrix[i][j] == 'G':
                goal = (i, j)
    if start is None or goal is None:
        raise ValueError("Matrix must contain both 'S' (start) and 'G' (goal).")
    return start, goal

def calculate_heuristic(current, target, heuristic_type):
    """
    Calculate the heuristic value (h) for the A* algorithm.
    """
    if heuristic_type == "manhattan":
        return abs(current[0] - target[0]) + abs(current[1] - target[1])
    elif heuristic_type == "euclidean":
        return math.sqrt((current[0] - target[0])**2 + (current[1] - target[1])**2)
    else:
        raise ValueError("Invalid heuristic type. Use 'manhattan' or 'euclidean'.")

def get_neighbors(node, grid):
    """
    Get valid neighbors of the current node.
    """
    neighbors = []
    directions = [
        (-1, 0), (1, 0), (0, -1), (0, 1),  # Left, Right, Up, Down
        (-1, -1), (1, 1), (-1, 1), (1, -1)  # Diagonal directions
    ]
    for dx, dy in directions:
        x, y = node[0] + dx, node[1] + dy
        if 0 <= x < len(grid) and 0 <= y < len(grid[0]) and grid[x][y] != -1:
            neighbors.append((x, y))
    return neighbors

def a_star_search(matrix, heuristic_type):
```

```
    """
    A* Search Algorithm to find the optimal path.
    """
    start, goal = find_start_and_goal(matrix)
    open_set = []  # Priority queue for nodes to explore
    heapq.heappush(open_set, (0, start))

    g_cost = {start: 0}  # Cost from start to a node
    f_cost = {start: calculate_heuristic(start, goal, heuristic_type)}  # Total cost
    parents = {start: None}  # Track the path

    while open_set:
        current = heapq.heappop(open_set)[1]  # Node with lowest f-cost

        # If goal is reached, reconstruct the path
        if current == goal:
            path = []
            while current:
                path.append(current)
                current = parents[current]
            return path[::-1]  # Return reversed path

        # Explore neighbors
        for neighbor in get_neighbors(current, matrix):
            tentative_g = g_cost[current] + 1  # Assume uniform cost for all moves

            if neighbor not in g_cost or tentative_g < g_cost[neighbor]:
                g_cost[neighbor] = tentative_g
                h = calculate_heuristic(neighbor, goal, heuristic_type)
                f_cost[neighbor] = g_cost[neighbor] + h
                parents[neighbor] = current

                # Add to open set
                heapq.heappush(open_set, (f_cost[neighbor], neighbor))

    return None  # No path found

def visualize_path(matrix, path):
    """
    Visualize the matrix with the path.
    """
    for x, y in path:
        if matrix[x][y] not in ('S', 'G'):  # Don't overwrite start/goal
            matrix[x][y] = '*'
    for row in matrix:
        print(" ".join(str(cell) for cell in row))
```

OUTPUTS :

```
--- Input Grid ---
S 1 1 0 0
1 0 1 1 0
1 1 1 0 0
0 1 0 0 0
0 1 1 1 G

Heuristic Used: Manhattan

--- Output Grid with Path ---
S 1 1 0 0
1 * 1 1 0
1 1 * 0 0
0 1 0 * 0
0 1 1 1 G

Heuristic Used: Euclidean

--- Output Grid with Path ---
S 1 1 0 0
1 * 1 1 0
1 1 * 0 0
0 1 0 * 0
0 1 1 1 G
```

```
--- Input Grid ---
S 1 1 1 0 0 0 0
1 1 1 1 1 0 0 0
0 0 1 0 1 1 1 1
0 1 1 1 1 1 1 G

Heuristic Used: Manhattan

--- Output Grid with Path ---
S 1 1 1 0 0 0 0
1 * 1 1 1 0 0 0
0 0 * 0 1 1 1 1
0 1 1 * * * * G

Heuristic Used: Euclidean

--- Output Grid with Path ---
S 1 1 1 0 0 0 0
1 * 1 1 1 0 0 0
0 0 * 0 1 1 1 1
0 1 1 * * * * G
```

Question 2 : In a spatial context defined by a square matrix of order N * N, a rat is situated at the starting point (0,0), aiming to reach the destination at (N-1, N-1). The task at hand is to enumerate all feasible paths that the rat can undertake to traverse from the source to the destination.

The permissible directions for the rat's movement are denoted as 'U' (up), 'D' (down), 'L' (left), and 'R' (right). Within this matrix, a cell assigned the value 0 signifies an obstruction, rendering it impassable for the rat, while a value of 1 indicates a traversable cell.

The objective is to furnish a list of paths in lexicographically increasing order, with the constraint that no cell can be revisited along the path. Moreover, if the source cell is assigned a value of 0, the rat is precluded from moving to any other cell.

To accomplish this, the AO* Search algorithm is employed to systematically explore the AND-OR graph and evaluate all conceivable paths from source to destination (with path cost = 1, and heuristic values given in the diagram).

The algorithm dynamically adapts its heuristic function during the search, optimizing the exploration process. The resultant list of paths reflects a meticulous exploration of the matrix, ensuring lexicographical order and adherence to the specified constraints.

Source **(A)**	B = 4	C
D = 10	E = 3	F
G	H = 2	I = 8
J	K = 3	Destination **(L = 5)**

Solution 2 :

```
def is_valid(x, y, visited, matrix):
    """Check if a cell is valid for movement."""
    rows, cols = len(matrix), len(matrix[0])
    return (
        0 <= x < rows and 0 <= y < cols and
        matrix[x][y] is not None and matrix[x][y][1] != 0 and
        (x, y) not in visited
```

```
    )

def explore_paths(matrix, x, y, path, cost, visited, results):
    """Recursive function to explore all paths."""
    rows, cols = len(matrix), len(matrix[0])

    # If destination is reached, add the path and cost to results
    if x == rows - 1 and y == cols - 1:
        results.append((path, cost))
        return

    # Mark the current cell as visited
    visited.add((x, y))

    # Define movement directions
    directions = {
        'U': (-1, 0),
        'D': (1, 0),
        'L': (0, -1),
        'R': (0, 1)
    }

    # Explore all valid moves
    for direction, (dx, dy) in sorted(directions.items()):
        nx, ny = x + dx, y + dy
        if is_valid(nx, ny, visited, matrix):
            next_cell = matrix[nx][ny]
            explore_paths(
                matrix, nx, ny, path + f" → {next_cell[0]}", cost + next_cell[1], visited,
results
            )

    # Backtrack: unmark the current cell as visited
    visited.remove((x, y))

def find_all_paths(matrix):
    """Find all paths from the top-left to the bottom-right of the matrix."""
    if matrix[0][0] is None or matrix[0][0][1] == 0:
        return []  # If the start cell is not traversable

    results = []
    start_cell = matrix[0][0]
    explore_paths(matrix, 0, 0, start_cell[0], start_cell[1], set(), results)

    # Sort results lexicographically by path
    results.sort(key=lambda x: x[0])
    return results

def print_paths(results):
    """Print the paths and their costs, and display the shortest path."""
```

```
    if not results:
        print("No valid paths found.")
        return

    for path, cost in results:
        print(f"Path: {path}, Total Cost: {cost}")

    # Find and print the shortest path
    shortest_path = min(results, key=lambda x: x[1])
    print(f"\nShortest Path: {shortest_path[0]}, Total Cost: {shortest_path[1]}")
```

OUTPUTS :

```
def main():
    """Main function to execute the program."""
    # Input: Matrix with tuples (alphabet, cost)
    matrix = [
    [('A', 1), ('B', 4), (None, 0)],  # First row
    [('D', 10), ('E', 3), (None, 0)], # Second row
    [(None, 0), ('H', 2), ('I', 8)],  # Third row
    [(None, 0), ('K', 3), ('L', 5)]   # Fourth row (Destination: L)
]

    # Find all paths
    results = find_all_paths(matrix)

    # Print the results
    print_paths(results)

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

```
Path: A → B → E → H → I → L, Total Cost: 23
Path: A → B → E → H → K → L, Total Cost: 18
Path: A → D → E → H → I → L, Total Cost: 29
Path: A → D → E → H → K → L, Total Cost: 24
```