ASSIGNMENT 2(A* and Best First Search)

n × n binary matrix grid

```
import heapq
import math

# Directions: 8 moves
directions = [(-1, 0), (1, 0), (0, -1), (0, 1),
              (-1, -1), (-1, 1), (1, -1), (1, 1)]

class State:
    def __init__(self, x, y, grid):
        self.x = x
        self.y = y
        self.grid = grid
        self.n = len(grid)

    def goal_test(self):
        return self.x == self.n - 1 and self.y == self.n - 1

    def move_gen(self):
        neighbors = []
        for dx, dy in directions:
            nx, ny = self.x + dx, self.y + dy
            if 0 <= nx < self.n and 0 <= ny < self.n and self.grid[nx][ny] == 0:
                neighbors.append(State(nx, ny, self.grid))
        return neighbors

    def __str__(self):
        return f"({self.x},{self.y})"

    def __repr__(self):
        return str(self)

    def __eq__(self, other):
        return self.x == other.x and self.y == other.y

    def __hash__(self):
        return hash((self.x, self.y))


# Heuristic: Euclidean distance
```

```
def heuristic(state):
    n = state.n
    return math.sqrt((state.x - (n-1))**2 + (state.y - (n-1))**2)



     #  Best-First Search

def best_first_search(grid):
    start = State(0, 0, grid)
    if grid[0][0] != 0 or grid[start.n-1][start.n-1] != 0:
        return -1, []

    OPEN = [(heuristic(start), start)]
    CLOSED = set()
    parents = {start: None}

    while OPEN:
        OPEN.sort(key=lambda x: x[0])  # sort by heuristic
        _, current = OPEN.pop(0)
        CLOSED.add(current)

        if current.goal_test():
            path = []
            while current:
                path.append((current.x, current.y))
                current = parents[current]
            return len(path), path[::-1]

        for neighbor in current.move_gen():
            if neighbor not in CLOSED and neighbor not in [x[1] for x in OPEN]:
                parents[neighbor] = current
                OPEN.append((heuristic(neighbor), neighbor))
    return -1, []



  # A* Search
def a_star_search(grid):
    start = State(0, 0, grid)
    goal = State(len(grid)-1, len(grid)-1, grid)
    if grid[0][0] != 0 or grid[goal.x][goal.y] != 0:
        return -1, []

    g_score = {start: 0}
    f_score = {start: heuristic(start)}
```

```
    parents = {start: None}
    OPEN = [(f_score[start], start)]
    CLOSED = set()

    while OPEN:
        heapq.heapify(OPEN)
        _, current = heapq.heappop(OPEN)
        if current.goal_test():
            path = []
            while current:
                path.append((current.x, current.y))
                current = parents[current]
            return len(path), path[::-1]

        CLOSED.add(current)

        for neighbor in current.move_gen():
            tentative_g = g_score[current] + 1
            if neighbor not in g_score or tentative_g < g_score[neighbor]:
                parents[neighbor] = current
                g_score[neighbor] = tentative_g
                f_score[neighbor] = tentative_g + heuristic(neighbor)
                if neighbor not in CLOSED:
                    heapq.heappush(OPEN, (f_score[neighbor], neighbor))
    return -1, []


  # Test Cases
grids = [
    [[0,1],[1,0]],
    [[0,0,0],[1,1,0],[1,1,0]],
    [[1,0,0],[1,1,0],[1,1,0]]
]

for i, grid in enumerate(grids, 1):
    print(f"\nExample {i}:")
    length_bfs, path_bfs = best_first_search(grid)
    length_astar, path_astar = a_star_search(grid)
    print(f"Best First Search → Path length: {length_bfs}, Path: {path_bfs}")
    print(f"A* Search → Path length: {length_astar}, Path: {path_astar}")
```

Output:

```
sahithya-arre@sahithya-arre-TravelMate-P214-53:~$ python3 ai2.py

Example 1:
Best First Search → Path length: 2, Path: [(0, 0), (1, 1)]
A* Search → Path length: 2, Path: [(0, 0), (1, 1)]

Example 2:
Best First Search → Path length: 4, Path: [(0, 0), (0, 1), (1, 2), (2, 2)]
A* Search → Path length: 4, Path: [(0, 0), (0, 1), (1, 2), (2, 2)]

Example 3:
Best First Search → Path length: -1, Path: []
A* Search → Path length: -1, Path: []
```

Comparison of Best-First Search and A* Search:

Best-First Search (Greedy Search) selects the next node to explore based solely on the heuristic (in this case, the Euclidean distance to the goal). This allows it to quickly find a path in many situations, but it does **not guarantee the shortest path**. In grids with obstacles, Best-First Search may take a longer route or fail to find the optimal path because it ignores the actual cost of the path traveled so far.

A* Search, in contrast, combines the heuristic with the actual cost from the start (`f = g + h`). This ensures that it always finds the **shortest path**, even in complex grids with obstacles. While A* may explore more nodes than Best-First Search, the extra computation guarantees optimality. In practice, Best-First Search can be faster but approximate, whereas A* is slightly slower but always accurate.