M.Ahtisham

LAB 5

22F-3331

BCS-6B

Task 1

```
[1]: def manhattan_distance(x1, y1, x2, y2):
         return abs(x2 - x1) + abs(y2 - y1)
```

```
[2]: def get_neighbors(x, y, grid):
         neighbors = []
         rows, cols = len(grid), len(grid[0])
         directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]

         for dx, dy in directions:
             nx, ny = x + dx, y + dy
             if 0 <= nx < rows and 0 <= ny < cols and grid[nx][ny] == 0:
                 neighbors.append((nx, ny))

         return neighbors
```

```
[4]: def basic_hill_climb(grid, start, goal):
         x, y = start
         path = [start]
         loop_count = 0

         while (x, y) != goal:
             loop_count += 1
             neighbors = get_neighbors(x, y, grid)
             if not neighbors:
                 return path, False, loop_count, len(path)

             best_neighbor = min(neighbors, key=lambda pos: manhattan_distance(pos[0], pos[1], goal[0], goal

             if manhattan_distance(best_neighbor[0], best_neighbor[1], goal[0], goal[1]) >= manhattan_distar
                 return path, False, loop_count, len(path)

             x, y = best_neighbor
             path.append((x, y))

         return path, True, loop_count, len(path)
```

```
[5]: def random_restart_hill_climbing(grid, start, goal, max_restarts=10):
         for _ in range(max_restarts):
             path, success, loop_count, path_length = basic_hill_climb(grid, start, goal)
             if success:
                 return path, True, loop_count, path_length
             start = (random.randint(0, len(grid) - 1), random.randint(0, len(grid[0]) - 1))
         return [], False, 0, 0
```

```
[6]: def stochastic_hill_climbing(grid, start, goal):
    x, y = start
    path = [start]
    loop_count = 0

    while (x, y) != goal:
        loop_count += 1
        neighbors = get_neighbors(x, y, grid)
        if not neighbors:
            return path, False, loop_count, len(path)

        distances = [manhattan_distance(nx, ny, goal[0], goal[1]) for nx, ny in neighbors]
        total = sum(1.0 / (d + 1) for d in distances)  # Avoid division by zero
        probabilities = [(1.0 / (d + 1)) / total for d in distances]

        best_neighbor = random.choices(neighbors, probabilities)[0]
        x, y = best_neighbor
        path.append((x, y))

    return path, True, loop_count, len(path)
```

```
[7]: def first_choice_hill_climbing(grid, start, goal, max_attempts=10):
    x, y = start
    path = [start]
    loop_count = 0

    while (x, y) != goal:
        loop_count += 1
        neighbors = get_neighbors(x, y, grid)
        if not neighbors:
            return path, False, loop_count, len(path)

        for _ in range(max_attempts):
            candidate = random.choice(neighbors)
            if manhattan_distance(candidate[0], candidate[1], goal[0], goal[1]) < manhattan_distance(x,
                x, y = candidate
                path.append((x, y))
                break
        else:
            return path, False, loop_count, len(path)

    return path, True, loop_count, len(path)
```

[8]:
```python
grid = [
    [0, 0, 1, 0, 0],
    [0, 1, 1, 1, 0],
    [0, 0, 0, 1, 0],
    [1, 1, 0, 0, 0],
    [0, 0, 0, 1, 0]
]
```

[9]:
```python
start = (0, 0)
goal = (4, 4)
```

[10]:
```python
path, success, loop_count, path_length = basic_hill_climb(grid, start, goal)
```

[11]:
```python
if success:
    print("Success: Path found!")
    print("Path:", path)
else:
    print("Failure: Local maximum reached!")
    print("Path:", path)
```

```
Failure: Local maximum reached!
Path: [(0, 0), (1, 0), (2, 0), (2, 1), (2, 2), (3, 2), (4, 2)]
```

[12]:
```python
print("Total loops executed:", loop_count)
```

```
Total loops executed: 7
```

[13]:
```python
print("Path length (number of elements used):", path_length)
```

```
Path length (number of elements used): 7
```

[14]:
```python
print("Random Restart Hill Climbing:", random_restart_hill_climbing(grid, start, goal))
print("Stochastic Hill Climbing:", stochastic_hill_climbing(grid, start, goal))
print("First-Choice Hill Climbing:", first_choice_hill_climbing(grid, start, goal))
```

```
Random Restart Hill Climbing: ([(4, 3), (4, 4)], True, 1, 2)
Stochastic Hill Climbing: ([(0, 0), (1, 0), (2, 0), (1, 0), (2, 0), (2, 1), (2, 2), (2, 1), (2, 2),
(2, 1), (2, 2), (2, 1), (2, 0), (2, 1), (2, 0), (2, 1), (2, 0), (1, 0), (0, 0), (0, 1), (0, 0), (1,
0), (2, 0), (2, 1), (2, 2), (2, 1), (2, 0), (2, 1), (2, 2), (2, 1), (2, 0), (2, 1), (2, 0), (1, 0),
(0, 0), (1, 0), (2, 0), (1, 0), (2, 0), (1, 0), (2, 0), (1, 0), (0, 0), (0, 1), (0, 0), (1, 0), (2,
0), (1, 0), (0, 0), (0, 1), (0, 0), (0, 1), (0, 0), (0, 1), (0, 0), (0, 1), (0, 0), (1, 0), (2, 0),
(1, 0), (0, 0), (0, 1), (0, 0), (0, 1), (0, 0), (0, 1), (0, 0), (0, 1), (0, 0), (1, 0), (0, 0), (1,
0), (0, 0), (0, 1), (0, 0), (0, 1), (0, 0), (0, 1), (0, 0), (0, 1), (0, 0), (1, 0), (0, 0), (1, 0),
(0, 0), (0, 1), (0, 0), (0, 1), (0, 0), (0, 1), (0, 0), (1, 0), (0, 0), (0, 1), (0, 0), (0, 1), (0,
0), (0, 1), (0, 0), (0, 1), (0, 0), (0, 1), (0, 0), (1, 0), (2, 0), (2, 1), (2, 2), (3, 2), (3, 3),
(3, 4), (3, 3), (3, 4), (4, 4)], True, 112, 113)
First-Choice Hill Climbing: ([(0, 0), (1, 0), (2, 0), (2, 1), (2, 2), (3, 2), (3, 3), (3, 4), (4, 4)],
True, 8, 9)
```

[ ]:

TASK 2

```
[1]: import random
     import math
```

```
[2]: def calculate_total_distance(path, distance_matrix):
         return sum(distance_matrix[path[i]][path[i+1]] for i in range(len(path) - 1)) + distance_matrix[pat
```

```
[3]: def generate_neighbor(path):
         new_path = path[:]
         i, j = random.sample(range(len(path)), 2)
         new_path[i], new_path[j] = new_path[j], new_path[i]
         return new_path
```

```
[4]: def simulated_annealing(distance_matrix, initial_temperature, cooling_rate, max_iterations):
         num_cities = len(distance_matrix)
         current_path = list(range(num_cities))
         random.shuffle(current_path)
         current_energy = calculate_total_distance(current_path, distance_matrix)
         best_path, best_energy = current_path[:], current_energy
         temperature = initial_temperature
         loop_count = 0

         for _ in range(max_iterations):
             loop_count += 1
             new_path = generate_neighbor(current_path)
             new_energy = calculate_total_distance(new_path, distance_matrix)
             delta_energy = new_energy - current_energy

             if delta_energy < 0 or random.random() < math.exp(-delta_energy / temperature):
                 current_path, current_energy = new_path, new_energy
                 if current_energy < best_energy:
                     best_path, best_energy = current_path[:], current_energy

             temperature *= cooling_rate
             if temperature < 1e-10:
                 break

         return best_path, best_energy, loop_count, len(best_path)
```

```
[5]: cities = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J']
     distance_matrix = [
         [0, 12, 29, 22, 13, 24, 6, 15, 18, 20],
         [12, 0, 19, 3, 25, 17, 30, 4, 21, 23],
         [29, 19, 0, 21, 23, 4, 9, 16, 2, 26],
         [22, 3, 21, 0, 7, 18, 5, 11, 19, 14],
         [13, 25, 23, 7, 0, 6, 31, 8, 27, 10],
         [24, 17, 4, 18, 6, 0, 14, 20, 9, 5],
         [6, 30, 9, 5, 31, 14, 0, 28, 12, 7],
         [15, 4, 16, 11, 8, 20, 28, 0, 3, 2],
         [18, 21, 2, 19, 27, 9, 12, 3, 0, 16],
         [20, 23, 26, 14, 10, 5, 7, 2, 16, 0]
     ]

[6]: initial_temperature, cooling_rate, max_iterations = 1000, 0.95, 1000
     best_path_indices, best_energy, loop_count, path_length = simulated_annealing(distance_matrix, initial_
     best_path = [cities[i] for i in best_path_indices]

[7]: print("Best Path:", " -> ".join(best_path) + " -> " + best_path[0])
     print("Minimum Distance:", best_energy)
     print("Total loops executed:", loop_count)
     print("Path length (extra memory):", path_length)

     Best Path: I -> C -> F -> E -> D -> B -> A -> G -> J -> H -> I
     Minimum Distance: 52
     Total loops executed: 584
     Path length (extra memory): 10
```

TASK 3

```
[1]: def get_valid_neighbors(x, y, grid, obstacles):
         neighbors = []
         rows, cols = len(grid), len(grid[0])
         directions = [(0, 1), (0, -1), (1, 0), (-1, 0)]  # Up, Down, Right, Left

         for dx, dy in directions:
             nx, ny = x + dx, y + dy
             if 0 <= nx < rows and 0 <= ny < cols and (nx, ny) not in obstacles:
                 neighbors.append((nx, ny))

         return neighbors
```

```
[2]: def hill_climb(grid, start, obstacles):
         x, y = start
         path = [(x, y)]
         current_elevation = grid[x][y]
         steps = 0

         while True:
             neighbors = get_valid_neighbors(x, y, grid, obstacles)
             if not neighbors:
                 break

             best_neighbor = max(neighbors, key=lambda pos: grid[pos[0]][pos[1]], default=None)
             if best_neighbor and grid[best_neighbor[0]][best_neighbor[1]] > current_elevation:
                 x, y = best_neighbor
                 current_elevation = grid[x][y]
                 path.append((x, y))
                 steps += 1
             else:
                 break

         return path, current_elevation, steps
```

```python
[3]: grid = [
         [1, 2, 3, 4, 5],
         [2, 3, 4, 8, 7],
         [3, 6, 5, 9, 10],
         [4, 7, 8, 12, 11],
         [5, 9, 10, 13, 14]
     ]
```

```python
[4]: obstacles = {(2, 3), (3, 1)}
     start = (0, 0)
```

```python
[5]: path, peak_elevation, steps = hill_climb(grid, start, obstacles)
```

```python
[6]: print("Path Taken:", path)
     print("Peak Elevation:", peak_elevation)
     print("Steps Taken:", steps)
```

```
Path Taken: [(0, 0), (0, 1), (0, 2), (0, 3), (1, 3)]
Peak Elevation: 8
Steps Taken: 4
```