LAB-5:SIMULATED ANNEALING 29/10/24

SHREYA S RUDAGI
1BM22CS267

1)8-queens:

CODE:

```
import numpy as np
from scipy.optimize import dual_annealing

def queens_max(position):
    # This function calculates the number of pairs of queens that are not attacking each other
    position = np.round(position).astype(int)  # Round and convert to integers for queen
positions
    n = len(position)
    queen_not_attacking = 0

    for i in range(n - 1):
        no_attack_on_j = 0
        for j in range(i + 1, n):
            # Check if queens are on the same row or on the same diagonal
            if position[i] != position[j] and abs(position[i] - position[j]) != (j - i):
                no_attack_on_j += 1
        if no_attack_on_j == n - 1 - i:
            queen_not_attacking += 1
    if queen_not_attacking == n - 1:
        queen_not_attacking += 1
    return -queen_not_attacking  # Negative because we want to maximize this value

# Bounds for each queen's position (0 to 7 for an 8x8 chessboard)
bounds = [(0, 7) for _ in range(8)]

# Use dual_annealing for simulated annealing optimization
result = dual_annealing(queens_max, bounds)

# Display the results
best_position = np.round(result.x).astype(int)
best_objective = -result.fun  # Flip sign to get the number of non-attacking queens

print('The best position found is:', best_position)
print('The number of queens that are not attacking each other is:', best_objective)
```

OUTPUT:

```
The best position found is: [2 4 1 7 0 6 3 5]
The number of queens that are not attacking each other is: 8
```

2) SUDUKU:

CODE:

```
import numpy as np
import random
import math

# Function to check if a number can be placed in a cell without breaking Sudoku rules
```

```python
def is_valid(puzzle, row, col, num):
    # Check if num is not in the row and column
    if num in puzzle[row] or num in puzzle[:, col]:
        return False
    # Check 3x3 box
    box_x, box_y = row // 3 * 3, col // 3 * 3
    if num in puzzle[box_x:box_x + 3, box_y:box_y + 3]:
        return False
    return True
```

```python
# Initial random filling respecting the initial clues
def initial_fill(puzzle):
    filled = puzzle.copy()
    for row in range(9):
        for col in range(9):
            if filled[row][col] == 0:
                possible_values = [num for num in range(1, 10) if is_valid(filled, row, col,
num)]
                if possible_values:
                    filled[row][col] = random.choice(possible_values)
    return filled
```

```python
# Objective function: number of violations in the Sudoku grid
def objective(puzzle):
    conflicts = 0
    for row in range(9):
        conflicts += 9 - len(set(puzzle[row]))
    for col in range(9):
        conflicts += 9 - len(set(puzzle[:, col]))
    for box_x in range(0, 9, 3):
        for box_y in range(0, 9, 3):
            box = puzzle[box_x:box_x + 3, box_y:box_y + 3].flatten()
            conflicts += 9 - len(set(box))
    return conflicts
```

```python
# Simulated Annealing algorithm
def simulated_annealing(puzzle, max_iter=100000000, start_temp=1.0, end_temp=0.01, alpha=0.99):
    # Generate the initial state with a random valid fill
    current_state = initial_fill(puzzle)
    current_score = objective(current_state)
```

```python
    temp = start_temp
    for iteration in range(max_iter):
        # Terminate if the puzzle is solved
        if current_score == 0:
            break
        # Randomly choose a cell to modify
        row, col = random.randint(0, 8), random.randint(0, 8)
        while puzzle[row][col] != 0:  # Skip pre-filled cells
            row, col = random.randint(0, 8), random.randint(0, 8)
```

```python
        # Swap with another random value in the same row that respects Sudoku rules
        new_state = current_state.copy()
        new_value = random.randint(1, 9)
        if is_valid(new_state, row, col, new_value):
            new_state[row][col] = new_value

        new_score = objective(new_state)
```

```python
        # Determine if we should accept the new state
        delta_score = new_score - current_score
```

```
        if delta_score < 0 or random.uniform(0, 1) < math.exp(-delta_score / temp):
            current_state, current_score = new_state, new_score

        # Update temperature and iteration
        temp *= alpha

    return current_state

# Example usage:
# 0 represents an empty cell
puzzle = np.array([
    [5, 3, 0, 0, 7, 0, 0, 0, 0],
    [6, 0, 0, 1, 9, 5, 0, 0, 0],
    [0, 9, 8, 0, 0, 0, 0, 6, 0],
    [8, 0, 0, 0, 6, 0, 0, 0, 3],
    [4, 0, 0, 8, 0, 3, 0, 0, 1],
    [7, 0, 0, 0, 2, 0, 0, 0, 6],
    [0, 6, 0, 0, 0, 0, 2, 8, 0],
    [0, 0, 0, 4, 1, 9, 0, 0, 5],
    [0, 0, 0, 0, 8, 0, 0, 7, 9]
])
solved_puzzle = simulated_annealing(puzzle)
print("Solved Sudoku:\n", solved_puzzle)
```

OUTPUT:

```
Solved Sudoku:
 [[5 3 2 6 7 4 9 1 8]
 [6 7 4 1 9 5 3 2 0]
 [1 9 8 3 0 2 4 6 7]
 [8 1 9 7 6 0 5 4 3]
 [4 2 6 8 5 3 7 9 1]
 [7 5 3 9 2 1 8 0 6]
 [9 6 5 0 3 7 2 8 4]
 [2 8 7 4 1 9 6 3 5]
 [3 4 1 2 8 6 0 7 9]]
```

3) MST:

CODE:

```
import random
import math
from collections import defaultdict

class Graph:
    def __init__(self):
        self.edges = defaultdict(list)

    def add_edge(self, u, v, weight):
        self.edges[u].append((v, weight))
        self.edges[v].append((u, weight))  # Undirected graph

    def get_edges(self):
```

```python
        return [(u, v, weight) for u in self.edges for v, weight in self.edges[u] if u < v]
```

```python
def random_spanning_tree(graph):
    nodes = list(graph.edges.keys())
    random.shuffle(nodes)
    tree_edges = set()
    selected = {nodes[0]}

    while len(selected) < len(nodes):
        u = random.choice(list(selected))
        candidates = [(v, weight) for v, weight in graph.edges[u] if v not in selected]
        if candidates:
            v, weight = random.choice(candidates)
            tree_edges.add((u, v, weight))
            selected.add(v)

    return tree_edges
```

```python
def energy(tree):
    return sum(weight for u, v, weight in tree)
```

```python
def generate_neighbor(tree, graph):
    tree_list = list(tree)
    if len(tree_list) < 2:
        return tree

    # Select a random edge to remove
    u, v, weight = random.choice(tree_list)
    new_tree = tree - {(u, v, weight)}

    # Find a new edge to add
    candidates = [(x, w) for x, w in graph.edges[u] if (x, u, w) not in tree and (u, x, w) not in tree]
    if not candidates:
        # If no candidates are available, return the original tree
        return tree

    new_v, new_weight = random.choice(candidates)

    # Add the new edge
    new_tree.add((u, new_v, new_weight))

    # Ensure the new tree is valid (could add a check here if necessary)
    return new_tree
```

```python
def simulated_annealing(graph):
    T = 1.0  # Initial temperature
    final_temperature = 0.001
    cooling_factor = 0.95
    current_solution = random_spanning_tree(graph)
    best_solution = current_solution

    while T > final_temperature:
        for _ in range(100):  # Number of iterations at current temperature
            neighbor = generate_neighbor(current_solution, graph)
            current_energy = energy(current_solution)
            neighbor_energy = energy(neighbor)

            if neighbor_energy < current_energy:
                current_solution = neighbor
            else:
```

```python
                acceptance_probability = math.exp((current_energy - neighbor_energy) / T)
                if random.random() < acceptance_probability:
                    current_solution = neighbor
```

```python
        if energy(current_solution) < energy(best_solution):
            best_solution = current_solution

        T *= cooling_factor

    return best_solution
```

```python
# Example usage:
if __name__ == "__main__":
    random.seed(42)  # Set a fixed seed for reproducibility
    graph = Graph()
    edges = [(0, 1, 4), (0, 2, 1), (1, 2, 2), (1, 3, 5), (2, 3, 3)]

    for u, v, weight in edges:
        graph.add_edge(u, v, weight)

    mst = simulated_annealing(graph)

    print("Edges in the Minimum Spanning Tree:")
    for u, v, weight in mst:
        print(f"{u} -- {v} (weight: {weight})")

    print("Total weight:", energy(mst))
```

OUTPUT:

```
Edges in the Minimum Spanning Tree:
0 -- 2 (weight: 1)
2 -- 3 (weight: 3)
2 -- 1 (weight: 2)
Total weight: 6
```

4) BEST PATH FOR ROBOT:

CODE:

```python
import numpy as np
import random

# Define the grid (0 = open, 1 = obstacle)
grid = np.array([
    [0, 0, 0, 0, 1],
    [1, 1, 0, 1, 0],
    [0, 0, 0, 1, 0],
    [0, 1, 1, 0, 0],
    [0, 0, 0, 0, 0]
])
```

```python
# Start and target positions
start = (0, 0)
target = (4, 4)
```

```python
grid_size = grid.shape

# Define moves: up, down, left, right
moves = [(-1, 0), (1, 0), (0, -1), (0, 1)]

# Fitness function to evaluate a path
def path_cost(path):
    position = start
    cost = 0
    for move in path:
        # Calculate the next position
        next_position = (position[0] + moves[move][0], position[1] + moves[move][1])

        # Check if move is within bounds and not an obstacle
        if 0 <= next_position[0] < grid_size[0] and 0 <= next_position[1] < grid_size[1]:
            if grid[next_position] == 1:
                cost += 100  # High penalty for hitting obstacle
            else:
                position = next_position
        else:
            cost += 100  # High penalty for going out of bounds

        # Increment cost per move
        cost += 1

    # Final Manhattan distance to target
    cost += np.abs(position[0] - target[0]) + np.abs(position[1] - target[1])
    return cost

# Function to generate a random path
def generate_random_path(length):
    return [random.randint(0, 3) for _ in range(length)]

# Function to generate a neighbor by changing one move in the path
def generate_neighbor(path):
    new_path = path.copy()
    idx = random.randint(0, len(new_path) - 1)
    new_path[idx] = random.randint(0, 3)  # Change one move
    return new_path

# Simulated annealing function for pathfinding
def simulated_annealing(initial_path, max_iterations=1000, initial_temp=1.0, final_temp=0.001):
    current_path = initial_path
    current_cost = path_cost(current_path)
    best_path = current_path
    best_cost = current_cost
    temperature = initial_temp

    for iteration in range(max_iterations):
        if temperature < final_temp:
            break

        neighbor_path = generate_neighbor(current_path)
        neighbor_cost = path_cost(neighbor_path)

        # If the neighbor is better, accept it
        if neighbor_cost < current_cost:
            current_path, current_cost = neighbor_path, neighbor_cost
        else:
            # Accept the neighbor with a probability based on temperature
            acceptance_prob = np.exp((current_cost - neighbor_cost) / temperature)
```

```python
            if random.random() < acceptance_prob:
                current_path, current_cost = neighbor_path, neighbor_cost

        # Update the best path found
        if current_cost < best_cost:
            best_path, best_cost = current_path, current_cost

        # Cool down the temperature
        temperature *= 0.95

    return best_path, best_cost

# Number of moves in the initial path
path_length = 20
initial_path = generate_random_path(path_length)

# Run simulated annealing to find the best path
best_path, best_cost = simulated_annealing(initial_path)

print("Best path moves:", best_path)
print("Path cost:", best_cost)

# Show resulting path with positions visited
position = start
path_positions = [start]

for move in best_path:
    # Calculate the next position
    next_position = (position[0] + moves[move][0], position[1] + moves[move][1])

    # Check if move is valid (within bounds and not hitting an obstacle)
    if 0 <= next_position[0] < grid_size[0] and 0 <= next_position[1] < grid_size[1] and
grid[next_position] == 0:
        # Update position if the move is valid
        position = next_position
        path_positions.append(position)

    # Stop if the target is reached
    if position == target:
        break

print("Positions visited:", path_positions)
```

OUTPUT:

```
Best path moves: [3, 2, 3, 3, 1, 1, 0, 1, 2, 2, 0, 1, 1, 3, 2, 0, 1, 1, 3, 3]
Path cost: 222
Positions visited: [(0, 0), (0, 1), (0, 0), (0, 1), (0, 2), (1, 2), (2, 2), (1, 2), (2, 2), (2, 1), (2, 0), (3, 0), (4, 0), (4, 1), (4, 0), (3, 0), (4, 0), (4, 1), (4, 2)]
```