LAB 5

1) Simulated annealing algorithm for 8 puzzle problem

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)
    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):
            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
```

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

2)SUDOKU PROBLEM

CODE :

```
import numpy as np
import random
import math

def is_valid(puzzle, row, col, num):
   if num in puzzle[row] or num in puzzle[:, col]:
      return False
   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

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

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

def simulated_annealing(puzzle, max_iter=100000000, start_temp=1.0,
end_temp=0.01, alpha=0.99):
    current_state = initial_fill(puzzle)
    current_score = objective(current_state)
    temp = start_temp

    for iteration in range(max_iter):
        if current_score == 0:
            break

        row, col = random.randint(0, 8), random.randint(0, 8)
        while puzzle[row][col] != 0:
            row, col = random.randint(0, 8), random.randint(0, 8)

        new_state = current_state.copy()
        new_value = random.randint(1, 9)
        new_state[row][col] = new_value if is_valid(new_state, row, col,
new_value) else current_state[row][col]
        new_score = objective(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

        temp *= alpha

    return current_state

# Example usage:
```

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

3)MST (Minimum Spanning Tree)

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))

    def get_edges(self):
        return [(u, v, weight) for u in self.edges for v, weight in self.edges[u] if u < v]

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

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

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

    u, v, weight = random.choice(tree_list)
    new_tree = tree - {(u, v, weight)}

    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:
        return tree
```

```
    new_v, new_weight = random.choice(candidates)
    new_tree.add((u, new_v, new_weight))

    return new_tree

def simulated_annealing(graph):
    T = 1.0
    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):
            neighbor = generate_neighbor(current_solution, graph)
            current_energy = energy(current_solution)
            neighbor_energy = energy(neighbor)

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

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

        T *= cooling_factor

    return best_solution

if __name__ == "__main__":
    random.seed(42)
    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