Department of Computer Engineering
Academic Term II: 23-24

Class: B.E (Computer), Sem – VI **Subject Name: Artificial Intelligence**

Student Name: Jatin Kadu **Roll No: 9548**

Practical No:	**10**
Title:	Travelling Salesman Problem
Date of Performance:	**8/4/24**
Date of Submission:	**8/4/24**

Rubrics for Evaluation:

Sr. No	Performance Indicator	Excellent	Good	Below Average	Marks
1	On time Completion & Submission (01)	01 (On Time)	NA	00 (Not on Time)	
2	Logic/Algorithm Complexity analysis (03)	03(Correct)	02(Partial)	01 (Tried)	
3	Coding Standards (03): Comments/indention/Naming conventions Test Cases /Output	03(All used)	02 (Partial)	01 (rarely followed)	
4	Post Lab Assignment (03)	03(done well)	2 (Partially Correct)	1(submitte d)	
Total					

Signature of the Teacher:

Experiment No: 10

Title: Travelling salesman problem solving using Genetic Algorithm

Objective: To write a program that solves the traveling Salesman problem in an efficient manner.

Theory:

Given a collection of cities and the cost of travel between each pair of them, the **traveling salesman problem**, or **TSP** for short, is to find the cheapest way of visiting all of the cities and returning to your starting point. In the standard version It study, the travel costs are symmetric in the sense that traveling from city X to city Y costs just as much as traveling from Y to X. Clarity on the problem statement as it may sound simple and deceptive.

Algorithm:

Input

1. Number of cities *n*
2. Cost of traveling between the cities.
3. *c* (*i*, *j*) *i*, *j* = 1, . . , *n*.
4. Start with city 1

Main Steps

1. /*Initialization */
2. *c*← 0
3. Cost ← 0
4. visits ← 0
5. *e* = 1 /* pointer of the visited city */
6. For 1 ≤ *r* ≤ *n*

 . Do {
 a. Choose pointer *j* with
 b. minimum = *c* (*e*, *j*) = min{*c* (*e*, *k*); visits (*k*) = 0 and 1 ≤ *k* ≤ *n* }
 c. cost ← cost + minimum - cost
 d. *e* = *j*

7. *C*(*r*) ← j
8. *C*(*n*) = 1
9. cost = cost + *c* (*e*, 1)

Using Genetic Algorithm

Finding a solution to the travelling salesman problem requires setting up a genetic algorithm in a specialized way. For instance, a valid solution would need to represent a route where every location is included at least once and only once. If a route contain a single location more than once, or missed a location out completely it wouldn't be valid and it would be valuable computation time calculating its distance.

Step 1. Choose mutation method to shuffle the route. Note that method should not add routes else invalid solutions will be produced.
Step 2. Select swap mutation for the procedure.
Step3.Select two locations at random to swap their positions.

For example, if swap mutation is applied to the following list, [1,2,3,4,5] it might end up with, [1,2,5,4,3]. Here, positions 3 and 5 were switched creating a new list with exactly the same values, just a different order.

Step 4.Make sure that values are not created and pre-existing values are used.

1	2	3	4	5	6	7	8	9

1	2	8	4	5	6	7	3	9

Step 5. Pick a crossover method which can enforce the same constraint.
Step 6. Select ordered crossover. In this crossover method, select a subset from the first parent, and then add that subset to the offspring.

Step 7. Add any missing values to the offspring from the second parent in order that they are found.
To make this explanation a little clearer consider the following example:

Parents

1	2	3	4	5	6	7	8	9

9	8	7	6	5	4	3	2	1

Offspring

					6	7	8	

9	5	4	3	2	6	7	8	1

Explanation:

Here a subset of the route is taken from the first parent (6,7,8) and added to the offspring's route. Next, the missing route locations are adding in order from the second parent. The first location in the second parent's route is 9 which isn't in the offspring's route so it's added in the first available position. The next position in the parent’s route is 8 which is in the offspring's route so it's skipped. This process continues until the offspring has no remaining empty values. If implemented correctly the end result should be a route which contains all of the positions its parents did with no positions missing or duplicated.

Post Lab Assignment:

1. How to overcome combinatorial explosion in TSP?
2. What is learning from travelling salesperson problem?

Code:

```
import random

# Genetic Algorithm parameters
POPULATION_SIZE = 50
MUTATION_RATE = 0.01
NUM_GENERATIONS = 1000

# Example city distances
CITY_DISTANCES = [
    [0, 29, 20, 21],
    [29, 0, 15, 18],
    [20, 15, 0, 25],
    [21, 18, 25, 0]
]

def create_initial_population(num_cities):
    population = []
    for _ in range(POPULATION_SIZE):
        route = list(range(1, num_cities))
        random.shuffle(route)
        population.append(route)
    return population

def calculate_fitness(route):
    total_distance = 0
    for i in range(len(route) - 1):
        total_distance += CITY_DISTANCES[route[i] - 1][route[i + 1] - 1]
    return total_distance

def crossover(parent1, parent2):
    offspring = [-1] * len(parent1)
    start_index = random.randint(0, len(parent1) - 1)
    end_index = random.randint(start_index, len(parent1) - 1)
    subset = parent1[start_index:end_index]
    offspring[start_index:end_index] = subset
    remaining = [city for city in parent2 if city not in subset]
    offspring = [city if city == -1 else city for city in offspring]
    for i in range(len(offspring)):
        if offspring[i] == -1:
            offspring[i] = remaining.pop(0)
    return offspring
```

```python
def mutate(route):
    if random.random() < MUTATION_RATE:
        idx1, idx2 = random.sample(range(len(route)), 2)
        route[idx1], route[idx2] = route[idx2], route[idx1]

def genetic_algorithm(num_cities):
    population = create_initial_population(num_cities)
    for _ in range(NUM_GENERATIONS):
        population = sorted(population, key=lambda x: calculate_fitness(x))
        new_population = []
        for _ in range(POPULATION_SIZE // 2):
            parent1, parent2 = random.choices(population[:POPULATION_SIZE // 10], k=2)
            offspring = crossover(parent1, parent2)
            mutate(offspring)
            new_population.append(offspring)
        population = population[:POPULATION_SIZE // 10] + new_population
    return population[0]

# Example usage:
num_cities = 4
optimal_route = genetic_algorithm(num_cities)
print("Optimal Route:", optimal_route)
print("Total Distance:", calculate_fitness(optimal_route))
```

Ouptut:

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    PORTS    SEARCH ERROR

PS C:\Users\hacke\OneDrive\Desktop\SEM VI\AI> & C:/Users/hacke/AppData/Local/Programs/Python/Python311/python.exe "c:/Users/hacke/OneDrive/Desktop/SEM VI/AI/tsp_geneticAlgo.py"
Optimal Route: [1, 3, 2]
Total Distance: 35
PS C:\Users\hacke\OneDrive\Desktop\SEM VI\AI>
```