Department of Computer Engineering

Academic Term II : 22-23

Class: B.E (Comp A), SemVI **Subject Name: Artificial Intelligence**

Student Name: **Roll No: 9211**

Practical No:	**8**
Title:	Travelling salesman problem solving using Genetic Algorithm
Date of Performance:	
Date of Submission:	

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(submitted)	
Total					

Signature of the Teacher:

Experiment No: 08

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 $1 \leq r \leq n$
    a. Do {
    b. Choose pointer *j* with
    c. minimum = *c* (*e*, *j*) = min{*c* (*e*, *k*); visits (*k*) = 0 and $1 \leq k \leq n$ }
    d. cost ← cost + minimum - cost
    e. *e* = *j*
7. *C*(*r*) ← j
8. *C*(*n*) = 1
9. cost = cost + *c* (*e*, 1)

Output

1. List of visited cities and total cost.

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.

CODE:

```
import random

# Define the list of cities and their coordinates
cities = {
    'City A': (2, 6),
    'City B': (6, 2),
    'City C': (8, 10),
    'City D': (14, 8),
    'City E': (10, 12)
}

# Set the number of iterations and population size
num_generations = 100
pop_size = 50

# Define the fitness function for each possible route
def fitness(route):
    total_distance = 0
    for i in range(len(route)):
```

```
        current_city = route[i]
        next_city = route[(i+1)%len(route)]
        total_distance += ((cities[current_city][0]-
cities[next_city][0])**2
                           + (cities[current_city][1]-
cities[next_city][1])**2)**0.5
    return 1/total_distance

# Define the initial population randomly
def initial_population(pop_size):
    population = []
    cities_list = list(cities.keys())
    for i in range(pop_size):
        route = random.sample(cities_list, len(cities_list))
        population.append(route)
    return population

# Define the selection function using tournament selection
def selection(population):
    tournament_size = 3
    tournament = random.sample(population, tournament_size)
    tournament.sort(key=lambda x: fitness(x), reverse=True)
    return tournament[0]

# Define the crossover function using ordered crossover
def crossover(parent1, parent2):
    child = [None]*len(parent1)
    start_index = random.randint(0, len(parent1)-2)
    end_index = random.randint(start_index+1, len(parent1)-1)
    # Copy the slice from parent1 to the child
    child[start_index:end_index+1] =
parent1[start_index:end_index+1]
    # Fill in the remaining cities from parent2
    parent2_index = 0
    for i in range(len(child)):
```

```
        if child[i] == None:
            while parent2[parent2_index] in child:
                parent2_index += 1
            child[i] = parent2[parent2_index]
            parent2_index += 1
    return child

# Define the mutation function using swap mutation
def mutation(route):
    index1 = random.randint(0, len(route)-1)
    index2 = random.randint(0, len(route)-1)
    route[index1], route[index2] = route[index2], route[index1]
    return route

# Define the main genetic algorithm function
def genetic_algorithm(num_generations, pop_size):
    population = initial_population(pop_size)
    for generation in range(num_generations):
        new_population = []
        for i in range(pop_size):
            parent1 = selection(population)
            parent2 = selection(population)
            child = crossover(parent1, parent2)
            if random.random() < 0.1:
                child = mutation(child)
            new_population.append(child)
        population = new_population
    best_route = max(population, key=lambda x: fitness(x))
    return best_route

# Run the genetic algorithm
best_route = genetic_algorithm(num_generations, pop_size)

# Print the best route and its distance
print('Best route:', best_route)
```

```
print('Distance:', 1/fitness(best_route))
```

Output:

```
Best route: ['City E', 'City D', 'City B', 'City A', 'City C']
Distance: 31.353238174658927
```

Post Lab Assignment:

1. How does a genetic algorithm work, and what are its key components?
2. What are the advantages and disadvantages of using genetic algorithms for optimization problems?
3. What is the role of fitness function in a genetic algorithm, and how is it chosen?
4. How can crossover and mutation operators be used to create new solutions in a genetic algorithm, and what is their impact on the search process?
5. How can population size and generation count be selected in a genetic algorithm to balance exploration and exploitation of the search space?
6. What are some real-world applications of genetic algorithms, and how have they been used to solve specific problems in these domains?