EXPERIMENT 2.3

Student Name: Samuel	**UID: 24MAI10018**
Branch: AIML	**Section: 24MAI-1**
Semester: 2	**Date : / /2025**
Subject Name: SC Lab	**Subject Code: 24CSH–668**

AIM: Implementation Genetic Application – Travelling Salesman Problem

SOFTWARE USED :JUPYTER Notebook

THEORY:

- Genetic algorithm is a search technique used in computing to find true or approximate solutions to approximate solutions to optimization & search problems.
- Genetic algorithms are inspired by Darwin's theory about evolution. Solution to a problem solved by genetic algorithms is evolved.
- Algorithm is started with a set of solutions (represented by chromosomes) called population. Solutions from one population are taken and used to form a new population. This is motivated by a hope, that the new population will be better than the old one. Solutions which are selected to form new solutions (offspring) are selected according to their fitness - the more suitable they are the more chances they have to reproduce.
- This is repeated until some condition (for example number of populations or improvement of the best solution) issatisfied.

Working of Genetic Algorithm in TSP

- Initialization: Generate a population of random tours (chromosomes).
- Fitness Evaluation: Compute fitness based on inverse of total tour distance (shorter tours = higher fitness).
- Selection: Choose parent solutions using methods like Roulette Wheel, Tournament Selection, Rank-Based Selection, or Elitism.
- Crossover: Combine parents to produce offspring using Order Crossover (OX), Partially Mapped Crossover (PMX), or Cycle Crossover (CX).
- Mutation: Introduce diversity by applying Swap, Inversion, or Scramble Mutation.
- Replacement: Form a new population by selecting the best offspring and elite individuals.

- Stopping Condition: The algorithm terminates when a maximum number of generations is reached, no improvement is observed, or a near-optimal solution is found.

Algorithm:

Step 1: **Input the number of cities and distance matrix.**

- The user provides the number of cities and their

distances. **Step 2**: **Initialize Population**

- Generate a population of random permutations of cities.

Step 3: **Evaluate Fitness**

- Compute fitness based on the total tour distance (shorter tours are better).

Step 4: **Selection**

- Select parents using Roulette Wheel Selection, Tournament Selection, or Rank Selection based on fitness.

Step 5: **Crossover (Recombination)**

- Apply Order Crossover (OX), PMX, or Cycle Crossover (CX) to generate new

offspring. **Step 6**: **Mutation**

- Apply Swap Mutation, Inversion Mutation, or Scramble Mutation to maintain

diversity. **Step 7**: **Replacement**

- Replace the old population with the new offspring.

Step 8: **Check Stopping Condition**

- If the stopping condition is met (max iterations reached or no improvement in best tour for X generations), stop.
- Otherwise, go to **Step 3**.

Step 9: **Return the Best Solution Found**

- The individual with the shortest tour distance is the solution.

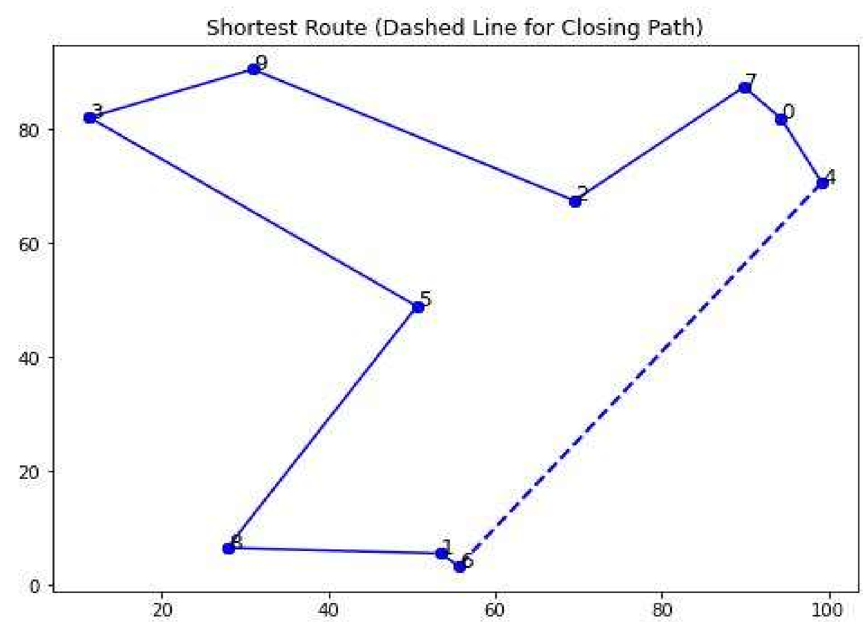
SOURCE CODE

```
import numpy as np
import random
import matplotlib.pyplot as plt
from itertools import permutations

# Generate random
cities num_cities = 10
cities = np.random.rand(num_cities, 2) * 100

def distance(route):
    """Calculate total distance of the route."""
```

```
    return sum(np.linalg.norm(cities[route[i]] - cities[route[i + 1]]) for i in range(len(route) - 1)) + \
           np.linalg.norm(cities[route[-1]] - cities[route[0]]) # Return to start

def create_population(size):
    """Generate a population of random routes."""
    return [random.sample(range(num_cities), num_cities) for _ in range(size)]

def tournament_selection(pop, k=3):
    """Select the best individual from k randomly chosen routes."""
    return min(random.sample(pop, k), key=distance)

def crossover(parent1, parent2):
    """Perform Order Crossover (OX)."""
    start, end = sorted(random.sample(range(num_cities), 2))
    child = [-1] * num_cities
    child[start:end] = parent1[start:end]

    remaining = [city for city in parent2 if city not in child]
    idx = 0
    for i in range(num_cities):
        if child[i] == -1:
            child[i] = remaining[idx]
            idx += 1

    return child

def mutate(route, mutation_rate=0.1):
    """Swap mutation."""
    if random.random() < mutation_rate:
        i, j = random.sample(range(num_cities), 2)
        route[i], route[j] = route[j], route[i]
    return route

def genetic_algorithm(generations=500, pop_size=100):
    """Main genetic algorithm."""
    population = create_population(pop_size)
    for _ in range(generations):
        new_population = [mutate(crossover(tournament_selection(population),
tournament_selection(population)))
                          for _ in range(pop_size)]
        population = new_population

    best_route = min(population, key=distance)
```

```
    return best_route

best_route = genetic_algorithm()

# Plot the cities and the shortest
path plt.figure(figsize=(8, 6))
for i in range(num_cities): plt.scatter(cities[i, 0],
    cities[i, 1], c='red') plt.text(cities[i, 0], cities[i,
    1], str(i), fontsize=12)

for i in range(num_cities - 1):
    plt.plot([cities[best_route[i], 0], cities[best_route[i + 1], 0]],
             [cities[best_route[i], 1], cities[best_route[i + 1], 1]], 'bo-')

# Close the loop with a dashed line
plt.plot([cities[best_route[-1], 0], cities[best_route[0], 0]],
         [cities[best_route[-1], 1], cities[best_route[0], 1]], 'b--', linewidth=2)

plt.title("Shortest Route (Dashed Line for Closing Path)")
plt.show()
```

SCREENSHOT OF OUTPUT:



LEARNING OUTCOME:

- Understand Genetic Algorithms (GA): Learn how selection, crossover, and mutation work to optimize solutions.
- Optimize Complex Problems: Apply GA to solve NP-hard problems like the Travelling Salesman Problem efficiently.
- Improve Route Planning: Learn how to find the shortest path in real-world applications like logistics and delivery systems.
- Visualize Optimization: Gain insights into how solutions evolve over generations through data visualization and graph plotting.