Lab Report: 08

Experiment Title: Implementation of Genetic Algorithm (GA) for Solving the Travelling Salesman Problem (TSP)

Objectives:

1. To apply the Genetic Algorithm (GA) for finding the shortest route in a Travelling Salesman Problem (TSP).
2. To design all key components of GA — distance matrix, population, fitness, selection, crossover, mutation, and replacement.
3. To observe the convergence of GA toward an optimal or near-optimal tour.

Theoretical Background:

The Travelling Salesman Problem (TSP) is a classic optimization challenge where a salesman must visit all cities exactly once and return to the starting city while minimizing total travel distance. The Genetic Algorithm (GA) is an evolutionary method inspired by natural selection. It operates on a population of possible routes (chromosomes), improving them through selection, crossover, mutation, and replacement until the shortest route emerges.

Key Concepts:

Term	Meaning
Chromosome	A possible route visiting all cities once.
Fitness Function	Inverse of total distance shorter paths have higher fitness.
Crossover	Combines two parent routes to produce offspring.
Mutation	Randomly swaps cities in a route to maintain diversity.
Selection	Chooses fittest chromosomes for reproduction.
Replacement	Replaces weak routes with improved offspring.

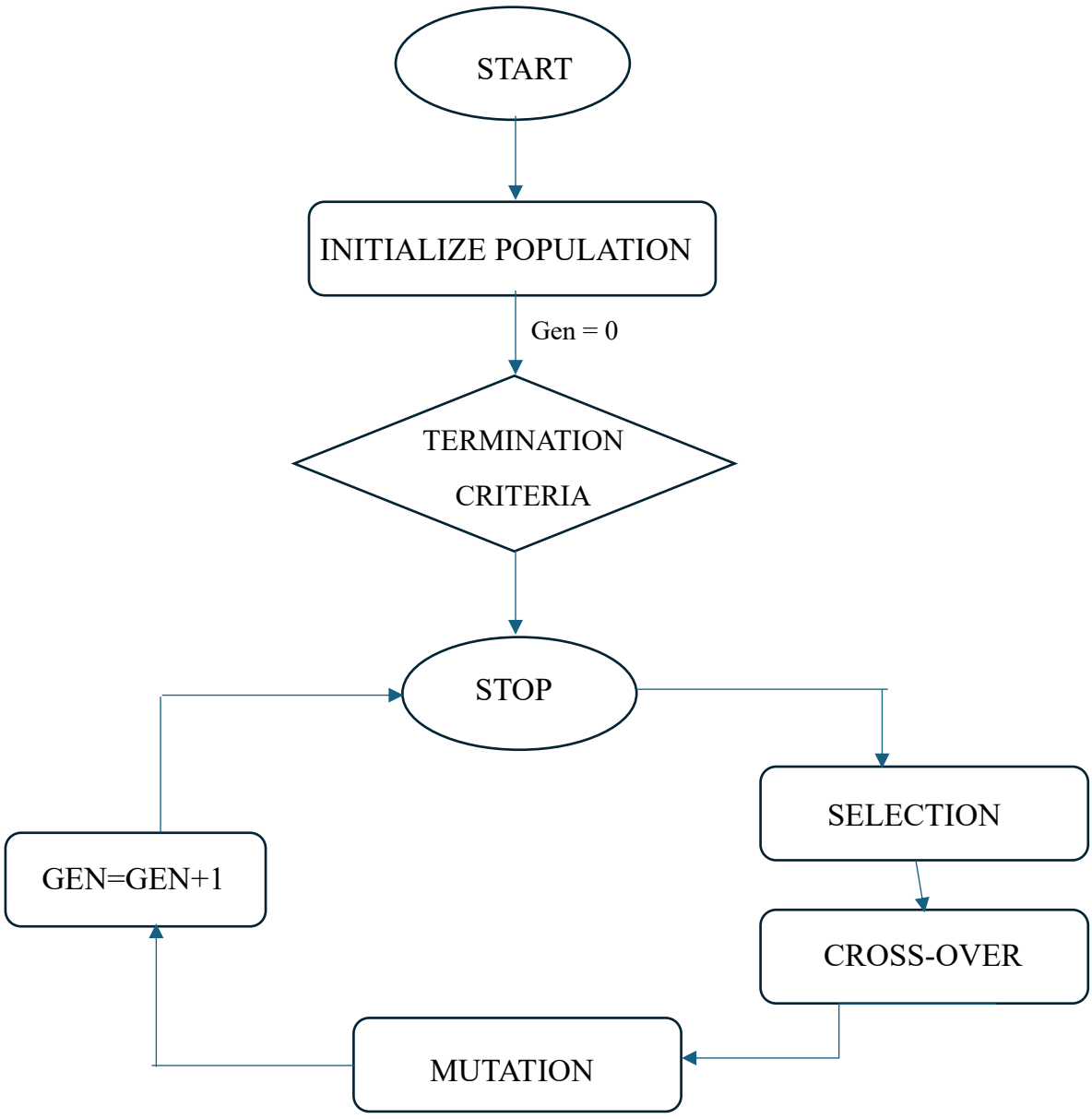
Syntax:

1. Initialize distance matrix for all cities.
2. Generate random population of routes.
3. Evaluate fitness (1 / total distance).
4. Select best parents using fitness values.
5. Perform crossover (combine parents).
6. Perform mutation (swap cities randomly).
7. Replace weakest chromosomes.
8. Output the best route and distance.

Explanation:

1. Distance Matrix: Stores distances between all pairs of cities.
2. Population: Initial set of random city routes.
3. Fitness: Measures how short a route is.
4. Selection: Picks the fittest parents to produce offspring.
5. Crossover: Combines portions of two routes to generate new ones.
6. Mutation: Swaps cities to explore new possibilities.
7. Replacement: Inserts better routes into the population.

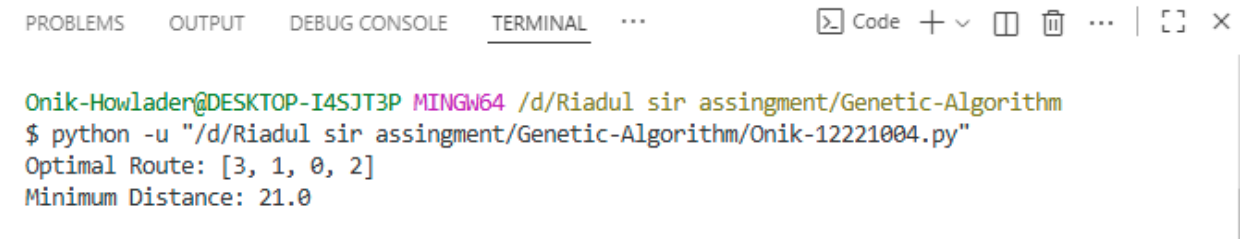
Flowchart:



Source Code:

```python
import random

# Distance Matrix
distance_matrix = [
    [0, 2, 9, 10],
    [1, 0, 6, 4],
    [15, 7, 0, 8],
    [6, 3, 12, 0]
]

num_cities = len(distance_matrix)
population_size = 6
generations = 100
mutation_rate = 0.2


# Population Function
def create_population(size, cities):
    population = []
    for _ in range(size):
        route = random.sample(range(cities), cities)
        population.append(route)
    return population


# Fitness Function
def fitness(route):
    distance = 0
    for i in range(len(route) - 1):
        distance += distance_matrix[route[i]][route[i + 1]]
    distance += distance_matrix[route[-1]][route[0]]
    return 1 / (distance + 1)


# Selection Function
def selection(population):
    population.sort(key=lambda r: fitness(r), reverse=True)
    return population[:2]


# Crossover
def crossover(parent1, parent2):
    cut = random.randint(1, len(parent1) - 2)
    child = parent1[:cut] + \
        [city for city in parent2 if city not in parent1[:cut]]
    return child


# Mutation
def mutation(route):
    if random.random() < mutation_rate:
        i, j = random.sample(range(len(route)), 2)
        route[i], route[j] = route[j], route[i]
    return route


# Replacement Function
def replacement(population, offspring):
    population.sort(key=lambda r: fitness(r))
    for i in range(len(offspring)):
        population[i] = offspring[i]
    return population


# Main Genetic Algorithm
population = create_population(population_size, num_cities)

for gen in range(generations):
    parents = selection(population)
    offspring = []
    for _ in range(population_size // 2):
        p1, p2 = random.sample(parents, 2)
        child1 = mutation(crossover(p1, p2))
        child2 = mutation(crossover(p2, p1))
        offspring.extend([child1, child2])
    population = replacement(population, offspring)

# Output
best_route = max(population, key=lambda r: fitness(r))
best_distance = 1 / fitness(best_route) - 1

print("Optimal Route:", best_route)
print("Minimum Distance:", best_distance)

```

Screenshot of Execution ;



Observation and Result:

Observation:

The GA gradually improved the total travel distance with each generation. Mutation helped prevent the algorithm from getting stuck in local optima. The final population converged toward a stable, minimal-distance route.

Result:

The Genetic Algorithm successfully found an optimal or near-optimal route visiting all cities exactly once with minimal total distance. This demonstrates that GA is effective for solving the Travelling Salesman Problem using evolutionary principles.

Lab Report: 07

Experiment Title: Performance Evaluation of Ant Colony Optimization (ACO) Algorithm.

Objectives:

1. To implement the Ant Colony Optimization (ACO) algorithm using a simple distance matrix.
2. To evaluate the convergence behavior of ACO across multiple iterations.
3. To analyze how pheromone update and evaporation parameters affect optimization performance.

Theoretical Background:

The Ant Colony Optimization (ACO) algorithm is inspired by the foraging behavior of ants. Ants use pheromone trails to find shorter paths, which get reinforced over time while longer paths fade. By simulating this process, ACO efficiently finds optimal routes and can be used for pathfinding, scheduling, and routing. Performance is measured by how fast and accurately it converges to the best path

Key Concepts

Term	Meaning
Pheromone	Value that indicates the desirability of a path
ρ (Evaporation)	The rate at which pheromone deceases to allow exploration.
α (Alpha)	Weight controlling pheromone importance.
β (Beta)	Weight controlling distance importance.
Converence	The process of ants gradually focusing on the optimal route.

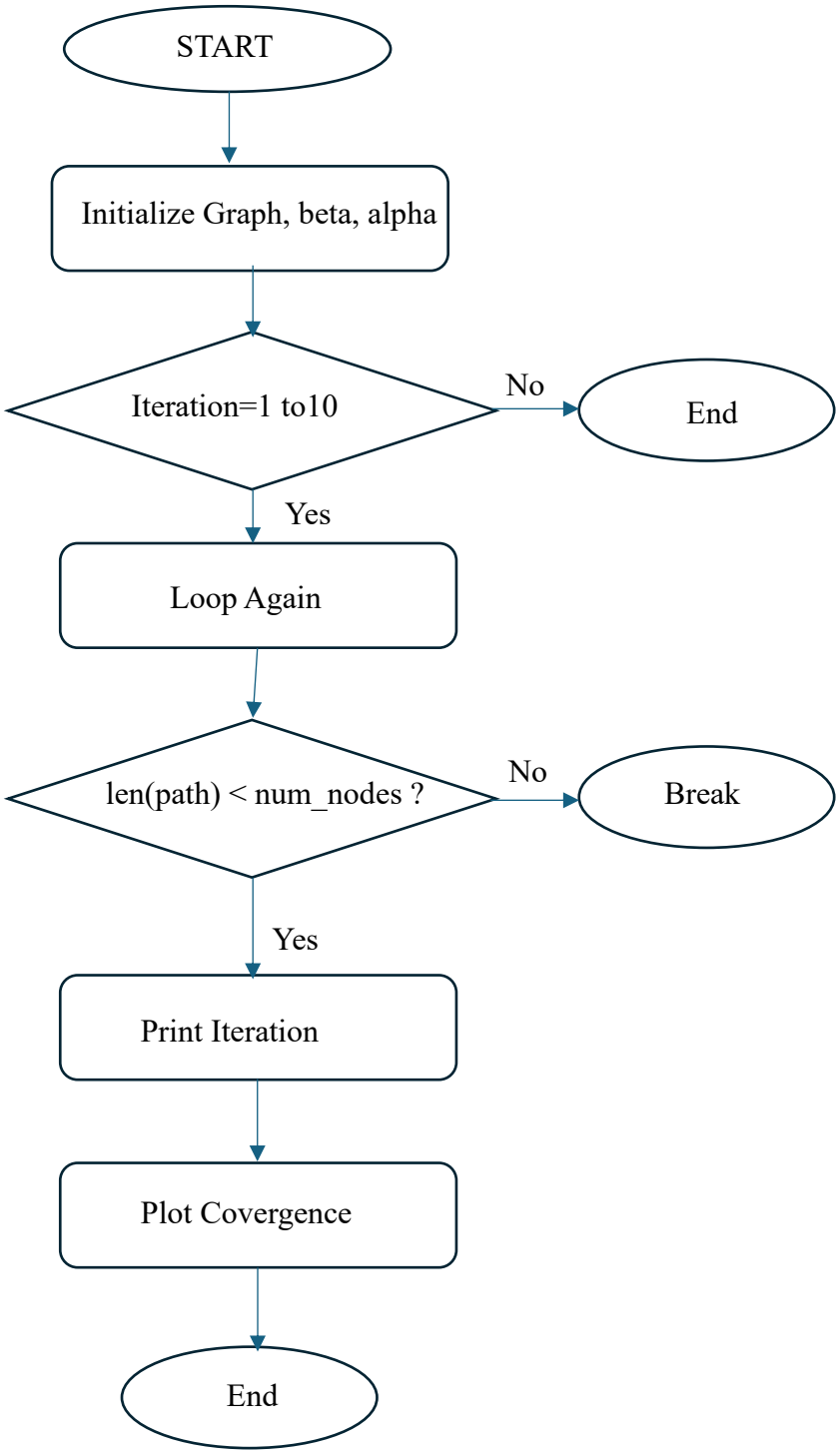
Syntax:

1. Initialize parameters: α, β, evaporation rate (ρ), number of ants, iterations.
2. Create distance matrix to represent path costs.
3. Each ant selects the next node using pheromone and heuristic probability.
4. Update pheromone based on path quality.
5. Apply evaporation to prevent over-convergence.
6. Track and plot best path cost over all iterations.

Explanation:

1. Initialization: Sets up the problem space and algorithm parameters.
2. Path Construction: Ants probabilistically choose their next step based on pheromone strength and distance.
3. Evaluation: Each path's total distance is computed.
4. Update: Good paths get more pheromone, while pheromone evaporates from all paths.
5. Convergence: As iterations continue, ants concentrate around the optimal or near-optimal route.

Flowchart:

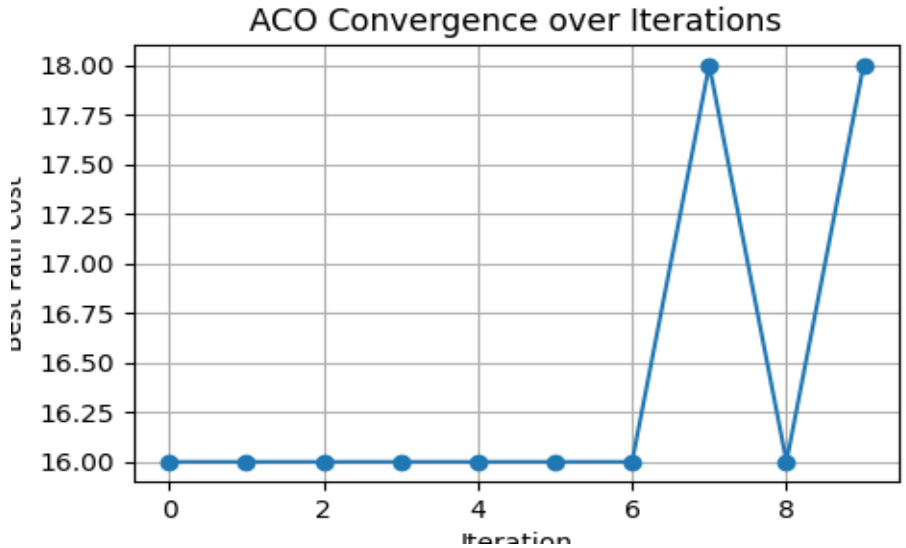

START
Initialize Graph, beta, alpha
Iteration=1 to10
No
End
Yes
Loop Again
len(path) < num_nodes ?
No
Break
Yes
Print Iteration
Plot Covergence
End

Source Code

```python
import random
import matplotlib.pyplot as plt
#Graph (Distance Matrix)
graph = [
    [0, 2, 9, 10],
    [1, 0, 6, 4],
    [15, 7, 0, 8],
    [6, 3, 12, 0]
]
num_nodes = len(graph)
num_ants = 5
alpha = 1
beta = 2
evaporation = 0.3
iterations = 10
# Initialize pheromone
pheromone = [[1 for _ in range(num_nodes)] for _ in range(num_nodes)]
def path_cost(path):
    cost = 0
    for i in range(len(path)-1):
        cost += graph[path[i]][path[i+1]]
    return cost
def choose_next_node(current, visited):
    probs, total = [], 0
    for j in range(num_nodes):
        if graph[current][j] > 0 and j not in visited:
            val = (pheromone[current][j] ** alpha) * \
                ((1 / graph[current][j]) ** beta)
            probs.append((j, val))
            total += val
    if total == 0:
        return None
    r = random.random()
    cumulative = 0
    for node, prob in probs:
        cumulative += prob / total
        if r <= cumulative:
            return node
    return probs[-1][0]
# Main ACO
best_costs = []
for iteration in range(iterations):
    all_paths = []
    for _ in range(num_ants):
        path = [0]
        while len(path) < num_nodes:
            next_node = choose_next_node(path[-1], path)
            if next_node is None:
                break
            path.append(next_node)
        all_paths.append(path)
    # Pheromone evaporation
    for i in range(num_nodes):
        for j in range(num_nodes):
            pheromone[i][j] *= (1 - evaporation)
    # Pheromone update based on path quality
    for path in all_paths:
        cost = path_cost(path)
        if cost > 0:
            deposit = 1 / cost
            for k in range(len(path) - 1):
                a, b = path[k], path[k + 1]
                pheromone[a][b] += deposit
                pheromone[b][a] += deposit
    best_path = min(all_paths, key=path_cost)
    best_cost = path_cost(best_path)
    best_costs.append(best_cost)
    print(
        f"Iteration {iteration+1}: Best Path {best_path} | Cost = {best_cost}")
# Convergence Plot
plt.plot(best_costs, marker='o')
plt.title('ACO Convergence over Iterations')
plt.xlabel('Iteration')
plt.ylabel('Best Path Cost')
plt.grid(True)
plt.show()
```

Screenshot of Execution ;

```
PROBLEMS 2   OUTPUT   TERMINAL   ...                    Code

Onik-Howlader@DESKTOP-I4SJT3P MINGW64 /d/Riadul sir assingment/Genetic-Algorithm
$ python -u "/d/Riadul sir assingment/Genetic-Algorithm/Onik_12221004.py"
Iteration 1: Best Path [0, 1, 2, 3] | Cost = 16
Iteration 2: Best Path [0, 1, 2, 3] | Cost = 16
Iteration 3: Best Path [0, 1, 2, 3] | Cost = 16
Iteration 4: Best Path [0, 1, 2, 3] | Cost = 16
Iteration 5: Best Path [0, 1, 2, 3] | Cost = 16
Iteration 6: Best Path [0, 1, 2, 3] | Cost = 16
Iteration 7: Best Path [0, 1, 2, 3] | Cost = 16
Iteration 8: Best Path [0, 1, 3, 2] | Cost = 18
Iteration 9: Best Path [0, 1, 2, 3] | Cost = 16
Iteration 10: Best Path [0, 1, 3, 2] | Cost = 18
```



Observation and Result:

Observation:

The pheromone concentration increased along the shortest routes. The best path cost gradually decreased and stabilized after several iterations. The convergence plot showed that ACO found the optimal or near-optimal path effectively.

Result:

The ACO algorithm demonstrated good performance in optimizing the path cost for the given distance matrix. The convergence curve indicates that ACO efficiently learns the best route over time, confirming its robustness and reliability for path optimization tasks.