```python
import numpy as np
import random

# Objective function to maximize
def objective_function(x):
    return x ** 2

# Initialize parameters
population_size = 100
num_generations = 50
mutation_rate = 0.1
crossover_rate = 0.7
range_min = -10
range_max = 10

# Create initial population
def initialize_population(size, min_val, max_val):
    return np.random.uniform(min_val, max_val, size)

# Evaluate fitness of the population
def evaluate_fitness(population):
    return np.array([objective_function(x) for x in population])

# Selection using roulette-wheel method
def selection(population, fitness):
    total_fitness = np.sum(fitness)
    probabilities = fitness / total_fitness
    # Select two parents based on fitness
    parents = population[np.random.choice(range(len(population)),
size=2, p=probabilities)]
    return parents[0], parents[1]

# Crossover between two parents
def crossover(parent1, parent2):
    if random.random() < crossover_rate:
        # Simple averaging for crossover, ensure offspring is within
range
        offspring = (parent1 + parent2) / 2
        return np.clip(offspring, range_min, range_max)
    return parent1 # No crossover

# Mutation of an individual
def mutate(individual):
    if random.random() < mutation_rate:
        return np.random.uniform(range_min, range_max)  # Random
mutation within range
    return individual

# Genetic Algorithm function
def genetic_algorithm():
```

```
    # Step 1: Initialize population
    population = initialize_population(population_size, range_min,
range_max)
    best_solution = None
    best_fitness = -np.inf

    # Evolutionary process
    for generation in range(num_generations):
        # Step 2: Evaluate fitness
        fitness = evaluate_fitness(population)

        # Track the best solution
        current_best_index = np.argmax(fitness)
        current_best_solution = population[current_best_index]
        current_best_fitness = fitness[current_best_index]

        if current_best_fitness > best_fitness:
            best_solution = current_best_solution
            best_fitness = current_best_fitness

        # Step 3: Create new population
        new_population = []
        for _ in range(population_size):
            # Select parents
            parent1, parent2 = selection(population, fitness)
            # Crossover to create offspring
            offspring = crossover(parent1, parent2)
            # Mutate offspring
            offspring = mutate(offspring)
            new_population.append(offspring)

        # Step 6: Replace old population with new population
        population = np.array(new_population)

    return best_solution, best_fitness

# Run the Genetic Algorithm
best_solution, best_fitness = genetic_algorithm()
print(f"Best Solution Found: {best_solution}, Fitness:
{best_fitness}")
```

```
Best Solution Found: 9.986850753164255, Fitness: 99.73718796597744
```

```
import numpy as np
import random
import math

# Define the problem (cities and vehicles)
num_vehicles = 3
locations = [
```

```
    (0, 0),  # Depot (starting point)
    (2, 4),
    (3, 1),
    (5, 2),
    (6, 4),
    (8, 3),
    (7, 7)
]
num_locations = len(locations)

# Euclidean distance function
def euclidean_distance(city1, city2):
    return math.sqrt((city1[0] - city2[0]) ** 2 + (city1[1] -
city2[1]) ** 2)

# Compute the distance matrix (distance between every pair of cities)
distance_matrix = np.zeros((num_locations, num_locations))
for i in range(num_locations):
    for j in range(num_locations):
        distance_matrix[i][j] = euclidean_distance(locations[i],
locations[j])

# Initialize parameters for the Genetic Algorithm
population_size = 100
num_generations = 500
mutation_rate = 0.05
crossover_rate = 0.7
elitism = 0.1  # Proportion of best individuals that are kept in the
next generation
max_route_length = num_locations - 1  # Maximum number of stops
(excluding depot)

# Generate a random solution (route)
def generate_random_route():
    route = list(range(1, num_locations))  # All cities except depot
    random.shuffle(route)
    return route

# Fitness function: calculate the total distance of the route
def calculate_fitness(route):
    total_distance = 0
    # Traverse through each vehicle's route
    current_city = 0  # Start at depot
    for i in route:
        total_distance += distance_matrix[current_city][i]
        current_city = i
    total_distance += distance_matrix[current_city][0]  # Return to
depot
    return total_distance
```

```
# Selection: Roulette wheel selection
def selection(population, fitness_scores):
    total_fitness = np.sum(fitness_scores)
    probabilities = fitness_scores / total_fitness
    selected_index = np.random.choice(range(population_size),
p=probabilities)
    return population[selected_index]

# Crossover: Ordered Crossover (OX)
def crossover(parent1, parent2):
    if random.random() < crossover_rate:
        start, end = sorted(random.sample(range(len(parent1)), 2))
        child = [-1] * len(parent1)

        # Copy a segment from parent1
        child[start:end] = parent1[start:end]

        # Fill in the remaining cities from parent2
        parent2_filtered = [city for city in parent2 if city not in
child]
        for i in range(len(parent1)):
            if child[i] == -1:
                child[i] = parent2_filtered.pop(0)
        return child
    return parent1  # No crossover

# Mutation: Swap mutation
def mutate(route):
    if random.random() < mutation_rate:
        idx1, idx2 = random.sample(range(len(route)), 2)
        route[idx1], route[idx2] = route[idx2], route[idx1]
    return route

# Genetic Algorithm: Main loop
def genetic_algorithm():
    population = [generate_random_route() for _ in
range(population_size)]

    # Evaluate initial population
    fitness_scores = np.array([1 / (calculate_fitness(route) + 1e-6)
for route in population])

    best_solution = None
    best_fitness = -np.inf

    for generation in range(num_generations):
        # Elitism: Preserve the best solutions
        elite_size = int(elitism * population_size)
        elite_indices = np.argsort(fitness_scores)[-elite_size:]
        elite_population = [population[i] for i in elite_indices]
```

```
        new_population = elite_population.copy()

        # Generate the rest of the population using selection,
crossover, and mutation
        while len(new_population) < population_size:
            parent1 = selection(population, fitness_scores)
            parent2 = selection(population, fitness_scores)

            child = crossover(parent1, parent2)
            child = mutate(child)

            new_population.append(child)

        # Evaluate the new population
        population = new_population
        fitness_scores = np.array([1 / (calculate_fitness(route) + 1e-
6) for route in population])

        # Track the best solution
        current_best_fitness = np.max(fitness_scores)
        if current_best_fitness > best_fitness:
            best_fitness = current_best_fitness
            best_solution = population[np.argmax(fitness_scores)]

        # Print progress (optional)
        if generation % 50 == 0:
            print(f"Generation {generation}, Best Fitness:
{best_fitness}")

    return best_solution, best_fitness

# Run the Genetic Algorithm
best_solution, best_fitness = genetic_algorithm()

# Convert the best solution to a route with the depot at the start and
end
best_route = [0] + best_solution + [0]

# Print the optimal route and its total distance
print("Optimal Route:", best_route)
print("Optimal Route Distance:", calculate_fitness(best_solution))
```

```
Generation 0, Best Fitness: 0.03856497041867355
Generation 50, Best Fitness: 0.04121661933407149
Generation 100, Best Fitness: 0.04121661933407149
Generation 150, Best Fitness: 0.04121661933407149
Generation 200, Best Fitness: 0.04121661933407149
Generation 250, Best Fitness: 0.04121661933407149
Generation 300, Best Fitness: 0.04121661933407149
```

```
Generation 350, Best Fitness: 0.04121661933407149
Generation 400, Best Fitness: 0.04121661933407149
Generation 450, Best Fitness: 0.04121661933407149
Optimal Route: [0, 2, 3, 5, 4, 6, 1, 0]
Optimal Route Distance: 24.2620567853496
```