```
import random

def generate_chromosome(length):
    return [random.randint(0, 1) for _ in range(length)]

def generate_population(size, chromosome_length):
    return [generate_chromosome(chromosome_length) for _ in range(size)]

def fitness(chromosome):
    # Implement your fitness function here
    # This is a placeholder example, replace it with your actual fitness calculation
    return sum(chromosome)

def selection(population, fitnesses):
    # Implement your selection mechanism here
    # This is a simple roulette wheel selection example
    total_fitness = sum(fitnesses)
    probabilities = [f / total_fitness for f in fitnesses]
    return random.choices(population, weights=probabilities, k=2)

def crossover(parent1, parent2):
    crossover_point = random.randint(1, len(parent1) - 1)
    offspring1 = parent1[:crossover_point] + parent2[crossover_point:]
    offspring2 = parent2[:crossover_point] + parent1[crossover_point:]
    return offspring1, offspring2

def mutation(chromosome, mutation_rate):
    for i in range(len(chromosome)):
        if random.random() < mutation_rate:
            chromosome[i] = 1 - chromosome[i]
    return chromosome

def genetic_algorithm(population_size, chromosome_length, generations, mutation_rate):
    population = generate_population(population_size, chromosome_length)

    for generation in range(generations):
        fitnesses = [fitness(chromosome) for chromosome in population]
        new_population = []

        for _ in range(population_size // 2):
            parent1, parent2 = selection(population, fitnesses)
            offspring1, offspring2 = crossover(parent1, parent2)
            offspring1 = mutation(offspring1, mutation_rate)
            offspring2 = mutation(offspring2, mutation_rate)
            new_population.extend([offspring1, offspring2])

        population = new_population

        best_fitness = max(fitnesses)
        best_chromosome = population[fitnesses.index(best_fitness)]
        print(f"Generation {generation+1}: Best fitness = {best_fitness}, Chromosome = {best_chromosome}")

    return best_chromosome

# Example usage:
population_size = 100
chromosome_length = 20
generations = 100
mutation_rate = 0.01

best_solution = genetic_algorithm(population_size, chromosome_length, generations, mutation_rate)
print("Final best solution:", best_solution)
```

```
Generation 1: Best fitness = 15, Chromosome = [0, 1, 1, 1, 0, 1, 0, 1, 1, 1, 0, 0, 1, 1, 1, 0, 1, 1, 1, 1]
Generation 2: Best fitness = 16, Chromosome = [1, 0, 1, 0, 1, 0, 1, 0, 1, 1, 0, 1, 0, 1, 1, 1, 0, 1, 0, 0]
Generation 3: Best fitness = 16, Chromosome = [0, 0, 0, 1, 1, 1, 0, 0, 0, 1, 0, 1, 0, 1, 0, 1, 0, 0, 0, 1]
Generation 4: Best fitness = 16, Chromosome = [1, 1, 0, 0, 1, 1, 1, 0, 0, 1, 1, 0, 1, 0, 1, 0, 1, 1, 1, 0]
Generation 5: Best fitness = 16, Chromosome = [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 0, 0, 1, 1, 1, 0, 0]
Generation 6: Best fitness = 16, Chromosome = [0, 1, 1, 1, 0, 1, 0, 0, 1, 1, 0, 0, 1, 1, 0, 1, 0, 1, 1, 1]
Generation 7: Best fitness = 16, Chromosome = [0, 1, 1, 0, 1, 0, 0, 1, 1, 1, 0, 0, 1, 1, 1, 0, 1, 1, 1, 0]
Generation 8: Best fitness = 17, Chromosome = [1, 1, 1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1]
Generation 9: Best fitness = 17, Chromosome = [1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1]
Generation 10: Best fitness = 18, Chromosome = [0, 1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 0, 0, 0, 1, 1, 1, 1]
Generation 11: Best fitness = 19, Chromosome = [0, 1, 1, 1, 0, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 0, 1, 1, 0]
Generation 12: Best fitness = 18, Chromosome = [1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1]
Generation 13: Best fitness = 18, Chromosome = [0, 1, 1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 0, 1, 1, 1]
Generation 14: Best fitness = 19, Chromosome = [0, 1, 1, 1, 0, 1, 1, 1, 1, 0, 0, 1, 0, 1, 1, 1, 1, 1, 0, 1]
Generation 15: Best fitness = 19, Chromosome = [1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1]
Generation 16: Best fitness = 19, Chromosome = [1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1]
Generation 17: Best fitness = 19, Chromosome = [0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 1]
Generation 18: Best fitness = 19, Chromosome = [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1]
Generation 19: Best fitness = 19, Chromosome = [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 0, 0, 1, 1, 0, 1, 1]
Generation 20: Best fitness = 18, Chromosome = [1, 1, 1, 0, 1, 1, 1, 0, 1, 0, 1, 1, 0, 1, 1, 1, 0, 0, 1, 1]
```

```
Generation 21: Best fitness = 19, Chromosome = [0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 0]
Generation 22: Best fitness = 18, Chromosome = [1, 1, 0, 0, 1, 1, 0, 1, 0, 1, 1, 1, 1, 0, 1, 1, 0, 0, 1, 0]
Generation 23: Best fitness = 18, Chromosome = [1, 1, 0, 0, 1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1]
Generation 24: Best fitness = 19, Chromosome = [0, 1, 1, 0, 1, 0, 0, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1]
Generation 25: Best fitness = 19, Chromosome = [0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 0, 1, 0, 1, 1]
Generation 26: Best fitness = 18, Chromosome = [0, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1]
Generation 27: Best fitness = 19, Chromosome = [1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 0, 1]
Generation 28: Best fitness = 20, Chromosome = [1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 0, 1]
Generation 29: Best fitness = 19, Chromosome = [1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 1]
Generation 30: Best fitness = 19, Chromosome = [1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1]
Generation 31: Best fitness = 19, Chromosome = [0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 0, 1, 1, 1]
Generation 32: Best fitness = 19, Chromosome = [1, 1, 0, 0, 1, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 0, 1, 0, 1, 0]
Generation 33: Best fitness = 20, Chromosome = [1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1]
Generation 34: Best fitness = 19, Chromosome = [1, 0, 1, 1, 0, 1, 1, 0, 0, 1, 0, 0, 1, 0, 1, 1, 1, 1, 1, 1]
Generation 35: Best fitness = 19, Chromosome = [1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 0, 0, 1]
Generation 36: Best fitness = 20, Chromosome = [1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1]
Generation 37: Best fitness = 19, Chromosome = [1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 0, 1, 1, 1]
Generation 38: Best fitness = 20, Chromosome = [1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 0, 1]
Generation 39: Best fitness = 19, Chromosome = [0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 1, 1]
Generation 40: Best fitness = 19, Chromosome = [1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 0, 1]
Generation 41: Best fitness = 19, Chromosome = [0, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1]
Generation 42: Best fitness = 19, Chromosome = [1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1]
Generation 43: Best fitness = 20, Chromosome = [1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
Generation 44: Best fitness = 20, Chromosome = [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
Generation 45: Best fitness = 20, Chromosome = [1, 1, 1, 1, 1, 0, 0, 0, 1, 0, 1, 0, 1, 1, 1, 1, 1, 0, 0, 1]
Generation 46: Best fitness = 20, Chromosome = [1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 0, 1, 1]
Generation 47: Best fitness = 20, Chromosome = [1, 1, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1]
Generation 48: Best fitness = 19, Chromosome = [1, 0, 1, 0, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 0, 1, 1, 1]
Generation 49: Best fitness = 19, Chromosome = [0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 0, 1, 0]
Generation 50: Best fitness = 20, Chromosome = [1, 0, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 0, 1, 1]
Generation 51: Best fitness = 19, Chromosome = [0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 0]
Generation 52: Best fitness = 20, Chromosome = [1, 0, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 0, 0, 1]
Generation 53: Best fitness = 20, Chromosome = [1, 0, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 0]
Generation 54: Best fitness = 19, Chromosome = [0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
Generation 55: Best fitness = 20, Chromosome = [1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 0, 1, 0, 1, 1, 0, 1, 1]
Generation 56: Best fitness = 20, Chromosome = [1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
Generation 57: Best fitness = 20, Chromosome = [1, 0, 1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 0]
Generation 58: Best fitness = 20, Chromosome = [1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1]
```

```
import random
import numpy as np

class Particle:
    def __init__(self, dimensions):
        self.position = [random.uniform(-10, 10) for _ in range(dimensions)]
        self.velocity = [random.uniform(-1, 1) for _ in range(dimensions)]
        self.pbest_position = self.position.copy()
        self.pbest_value = float('inf')

class PSO:
    def __init__(self, dimensions, swarm_size, iterations, c1, c2, w):
        self.dimensions = dimensions
        self.swarm_size = swarm_size
        self.iterations = iterations
        self.c1 = c1  # Cognitive coefficient
        self.c2 = c2  # Social coefficient
        self.w = w  # Inertia weight

        self.swarm = [Particle(dimensions) for _ in range(swarm_size)]

        # Initialize gbest based on initial population fitness
        self.gbest_value = float('inf')
        for particle in self.swarm:
            fitness = self.evaluate(particle.position)
            if fitness < self.gbest_value:
                self.gbest_value = fitness
                self.gbest_position = particle.position.copy()

    def evaluate(self, position):
        # Replace this with your specific objective function
        return sum(x**2 for x in position)

    def update_velocity(self, particle):
        for i in range(self.dimensions):
            r1 = random.random()
            r2 = random.random()
            particle.velocity[i] = (self.w * particle.velocity[i]) + \
                                   (self.c1 * r1 * (particle.pbest_position[i] - particle.position[i])) + \
                                   (self.c2 * r2 * (self.gbest_position[i] - particle.position[i]))

    def update_position(self, particle):
        for i in range(self.dimensions):
            particle.position[i] += particle.velocity[i]

    def update_pbest(self, particle):
```

```
            fitness = self.evaluate(particle.position)
            if fitness < particle.pbest_value:
                particle.pbest_value = fitness
                particle.pbest_position = particle.position.copy()

        def update_gbest(self):
            for particle in self.swarm:
                fitness = self.evaluate(particle.position)
                if fitness < self.gbest_value:
                    self.gbest_value = fitness
                    self.gbest_position = particle.position.copy()

        def optimize(self):
            for _ in range(self.iterations):
                for particle in self.swarm:
                    self.update_velocity(particle)
                    self.update_position(particle)
                    self.update_pbest(particle)
                self.update_gbest()

            return self.gbest_position, self.gbest_value

    # Example usage:
    dimensions = 10
    swarm_size = 50
    iterations = 100
    c1 = 2.0
    c2 = 2.0
    w = 0.7

    pso = PSO(dimensions, swarm_size, iterations, c1, c2, w)
    best_position, best_value = pso.optimize()
    print("Best position:", best_position)
    print("Best value:", best_value)
```

```
Best position: [1.5804132675985314, 0.6945689028300615, -0.2134268303825733, -0.8610857084066699, -0.30811488979935336, 0.359736818
Best value: 5.275664017890163
```

```
    import numpy as np
    import random

    class AntHillOptimization:
        def __init__(self, distance_matrix, n_ants, n_iterations, alpha=1, beta=5, rho=0.5, q=100):
            self.distance_matrix = distance_matrix  # Distance matrix
            self.num_cities = len(distance_matrix)  # Number of cities
            self.n_ants = n_ants  # Number of ants
            self.n_iterations = n_iterations  # Number of iterations
            self.alpha = alpha  # Influence of pheromone
            self.beta = beta  # Influence of distance
            self.rho = rho  # Evaporation rate of pheromone
            self.q = q  # Constant for pheromone update
            self.pheromone = np.ones((self.num_cities, self.num_cities))  # Initial pheromone levels

        def _get_probabilities(self, current_city, visited_cities):
            # Calculate the probabilities for each possible next city based on pheromone and distance
            probabilities = []
            total = 0.0

            for i in range(self.num_cities):
                if i not in visited_cities:
                    pheromone = self.pheromone[current_city][i] ** self.alpha
                    distance = float(self.distance_matrix[current_city][i]) ** -self.beta  # Cast to float to allow negative powers
                    probabilities.append(pheromone * distance)
                    total += pheromone * distance
                else:
                    probabilities.append(0)

            # Normalize probabilities
            probabilities = [prob / total for prob in probabilities]
            return probabilities

        def _construct_solution(self):
            visited_cities = [random.randint(0, self.num_cities - 1)]  # Start from a random city
            while len(visited_cities) < self.num_cities:
                current_city = visited_cities[-1]
                probabilities = self._get_probabilities(current_city, visited_cities)
                next_city = np.random.choice(self.num_cities, p=probabilities)
                visited_cities.append(next_city)
            visited_cities.append(visited_cities[0])  # Return to the starting city
            return visited_cities
```

```
    def _calculate_total_distance(self, tour):
        # Calculate the total distance for a given tour
        total_distance = 0
        for i in range(len(tour) - 1):
            total_distance += self.distance_matrix[tour[i]][tour[i + 1]]
        return total_distance

    def _update_pheromone(self, all_solutions, all_distances):
        # Evaporate pheromone levels
        self.pheromone *= (1 - self.rho)

        # Deposit pheromone based on the quality of solutions
        for i in range(self.n_ants):
            tour = all_solutions[i]
            tour_distance = all_distances[i]
            pheromone_deposit = self.q / tour_distance  # Deposit pheromone inversely proportional to the distance
            for i in range(len(tour) - 1):
                self.pheromone[tour[i]][tour[i + 1]] += pheromone_deposit
                self.pheromone[tour[i + 1]][tour[i]] += pheromone_deposit  # Since the graph is undirected

    def run(self):
        best_tour = None
        best_distance = float('inf')

        for iteration in range(self.n_iterations):
            all_solutions = []
            all_distances = []

            for _ in range(self.n_ants):
                solution = self._construct_solution()
                total_distance = self._calculate_total_distance(solution)
                all_solutions.append(solution)
                all_distances.append(total_distance)

                if total_distance < best_distance:
                    best_tour = solution
                    best_distance = total_distance

            # Update pheromone based on the ants' solutions
            self._update_pheromone(all_solutions, all_distances)

            print(f"Iteration {iteration + 1}/{self.n_iterations}, Best Distance: {best_distance}")

        return best_tour, best_distance


# Example usage
if __name__ == "__main__":
    # Example distance matrix for 5 cities
    distance_matrix = np.array([
        [0, 10, 15, 20, 25],
        [10, 0, 35, 25, 30],
        [15, 35, 0, 30, 5],
        [20, 25, 30, 0, 15],
        [25, 30, 5, 15, 0]
    ])

    # Parameters: n_ants, n_iterations, alpha, beta, rho, q
    aco = AntHillOptimization(distance_matrix, n_ants=10, n_iterations=100, alpha=1, beta=5, rho=0.5, q=100)

    best_tour, best_distance = aco.run()

    print(f"Best tour: {best_tour}")
    print(f"Best distance: {best_distance}")
```

```
Iteration 1/100, Best Distance: 70
Iteration 2/100, Best Distance: 70
Iteration 3/100, Best Distance: 70
Iteration 4/100, Best Distance: 70
Iteration 5/100, Best Distance: 70
Iteration 6/100, Best Distance: 70
Iteration 7/100, Best Distance: 70
Iteration 8/100, Best Distance: 70
Iteration 9/100, Best Distance: 70
Iteration 10/100, Best Distance: 70
Iteration 11/100, Best Distance: 70
Iteration 12/100, Best Distance: 70
Iteration 13/100, Best Distance: 70
Iteration 14/100, Best Distance: 70
Iteration 15/100, Best Distance: 70
Iteration 16/100, Best Distance: 70
Iteration 17/100, Best Distance: 70
```

```
Iteration 18/100, Best Distance: 70
Iteration 19/100, Best Distance: 70
Iteration 20/100, Best Distance: 70
Iteration 21/100, Best Distance: 70
Iteration 22/100, Best Distance: 70
Iteration 23/100, Best Distance: 70
Iteration 24/100, Best Distance: 70
Iteration 25/100, Best Distance: 70
Iteration 26/100, Best Distance: 70
Iteration 27/100, Best Distance: 70
Iteration 28/100, Best Distance: 70
Iteration 29/100, Best Distance: 70
Iteration 30/100, Best Distance: 70
Iteration 31/100, Best Distance: 70
Iteration 32/100, Best Distance: 70
Iteration 33/100, Best Distance: 70
Iteration 34/100, Best Distance: 70
Iteration 35/100, Best Distance: 70
Iteration 36/100, Best Distance: 70
Iteration 37/100, Best Distance: 70
Iteration 38/100, Best Distance: 70
Iteration 39/100, Best Distance: 70
Iteration 40/100, Best Distance: 70
Iteration 41/100, Best Distance: 70
Iteration 42/100, Best Distance: 70
Iteration 43/100, Best Distance: 70
Iteration 44/100, Best Distance: 70
Iteration 45/100, Best Distance: 70
Iteration 46/100, Best Distance: 70
Iteration 47/100, Best Distance: 70
Iteration 48/100, Best Distance: 70
Iteration 49/100, Best Distance: 70
Iteration 50/100, Best Distance: 70
Iteration 51/100, Best Distance: 70
Iteration 52/100, Best Distance: 70
Iteration 53/100, Best Distance: 70
Iteration 54/100, Best Distance: 70
Iteration 55/100, Best Distance: 70
Iteration 56/100, Best Distance: 70
Iteration 57/100, Best Distance: 70
Iteration 58/100, Best Distance: 70
```

```
import numpy as np
import random

# Cuckoo Search for TSP
class CuckooSearch:
    def __init__(self, distance_matrix, n_nests, n_iterations, alpha=0.01, beta=1.5, pa=0.25):
        self.distance_matrix = distance_matrix  # Distance matrix
        self.num_cities = len(distance_matrix)  # Number of cities
        self.n_nests = n_nests  # Number of nests
        self.n_iterations = n_iterations  # Number of iterations
        self.alpha = alpha  # Step size (scaling factor for Levy flights)
        self.beta = beta  # Levy flight exponent
        self.pa = pa  # Discovery rate (probability of abandoning a nest)
        self.nests = np.array([self._generate_solution() for _ in range(n_nests)])  # Initial solutions (nests)
        self.best_solution = None  # Best solution
        self.best_distance = float('inf')  # Best distance (fitness)

    def _generate_solution(self):
        # Generate a random solution (random permutation of cities)
        return np.random.permutation(self.num_cities)

    def _calculate_total_distance(self, tour):
        # Calculate the total distance of the tour
        total_distance = 0
        for i in range(len(tour) - 1):
            total_distance += self.distance_matrix[tour[i]][tour[i + 1]]
        total_distance += self.distance_matrix[tour[-1]][tour[0]]  # Return to the starting city
        return total_distance

    def _levy_flight(self, solution):
        # Perform a Levy flight to create a new candidate solution
        step = np.random.normal(0, 1, size=solution.shape) * np.abs(np.random.normal(0, 1)) ** self.beta
        new_solution = solution + self.alpha * step

        # Ensure the new solution is a valid permutation of cities
        new_solution = np.clip(new_solution, 0, self.num_cities - 1).astype(int)

        # Fix invalid solutions (duplicates, out-of-bound values)
        unique_solution = np.unique(new_solution)
        if len(unique_solution) < self.num_cities:
            # If the solution contains duplicates, replace them with missing cities
            missing_cities = set(range(self.num_cities)) - set(unique_solution)
            new_solution = np.array(list(unique_solution) + list(missing_cities))
```

```
            # Ensure the solution has the correct length and is a valid permutation
            return np.random.permutation(new_solution)  # Ensure it's a permutation

        def _get_best_nests(self, fitness):
            # Get the best solutions (nests) based on fitness (shortest distance)
            sorted_indices = np.argsort(fitness)
            best_nests = self.nests[sorted_indices[:self.n_nests // 2]]
            return best_nests

        def _abandon_nest(self):
            # Abandon the worst nests with probability pa and generate a new solution
            new_nests = []
            for nest in self.nests:
                if random.random() > self.pa:
                    new_nests.append(nest)
                else:
                    new_nests.append(self._generate_solution())  # Generate a random new solution
            return np.array(new_nests)

        def run(self):
            # Main Cuckoo Search loop
            for iteration in range(self.n_iterations):
                fitness = np.array([self._calculate_total_distance(nest) for nest in self.nests])

                # Update the best solution found so far
                best_nest_idx = np.argmin(fitness)
                best_solution = self.nests[best_nest_idx]
                best_distance = fitness[best_nest_idx]

                if best_distance < self.best_distance:
                    self.best_solution = best_solution
                    self.best_distance = best_distance

                # Perform Levy flights to update nests
                new_nests = []
                for i, nest in enumerate(self.nests):
                    new_solution = self._levy_flight(nest)
                    new_nests.append(new_solution)

                # Ensure all solutions are valid and consistent in shape
                new_nests = np.array(new_nests)

                # Replace the worst nests with better ones from Levy flights
                fitness_new = np.array([self._calculate_total_distance(nest) for nest in new_nests])
                best_nests = self._get_best_nests(fitness_new)
                self.nests = np.vstack((best_nests, self._abandon_nest()))

                print(f"Iteration {iteration + 1}/{self.n_iterations}, Best Distance: {self.best_distance}")

            return self.best_solution, self.best_distance


    # Example usage
    if __name__ == "__main__":
        # Example distance matrix for 5 cities
        distance_matrix = np.array([
            [0, 10, 15, 20, 25],
            [10, 0, 35, 25, 30],
            [15, 35, 0, 30, 5],
            [20, 25, 30, 0, 15],
            [25, 30, 5, 15, 0]
        ])

        # Parameters: n_nests, n_iterations, alpha, beta, pa
        cuckoo = CuckooSearch(distance_matrix, n_nests=10, n_iterations=100, alpha=0.01, beta=1.5, pa=0.25)

        best_solution, best_distance = cuckoo.run()

        print(f"Best solution: {best_solution}")
        print(f"Best distance: {best_distance}")
```

```
Iteration 1/100, Best Distance: 95
Iteration 2/100, Best Distance: 95
Iteration 3/100, Best Distance: 85
Iteration 4/100, Best Distance: 85
Iteration 5/100, Best Distance: 85
Iteration 6/100, Best Distance: 85
Iteration 7/100, Best Distance: 85
Iteration 8/100, Best Distance: 85
Iteration 9/100, Best Distance: 70
Iteration 10/100, Best Distance: 70
Iteration 11/100, Best Distance: 70
```

```
Iteration 12/100, Best Distance: 70
Iteration 13/100, Best Distance: 70
Iteration 14/100, Best Distance: 70
Iteration 15/100, Best Distance: 70
Iteration 16/100, Best Distance: 70
Iteration 17/100, Best Distance: 70
Iteration 18/100, Best Distance: 70
Iteration 19/100, Best Distance: 70
Iteration 20/100, Best Distance: 70
Iteration 21/100, Best Distance: 70
Iteration 22/100, Best Distance: 70
Iteration 23/100, Best Distance: 70
Iteration 24/100, Best Distance: 70
Iteration 25/100, Best Distance: 70
Iteration 26/100, Best Distance: 70
Iteration 27/100, Best Distance: 70
Iteration 28/100, Best Distance: 70
Iteration 29/100, Best Distance: 70
Iteration 30/100, Best Distance: 70
Iteration 31/100, Best Distance: 70
Iteration 32/100, Best Distance: 70
Iteration 33/100, Best Distance: 70
Iteration 34/100, Best Distance: 70
Iteration 35/100, Best Distance: 70
Iteration 36/100, Best Distance: 70
Iteration 37/100, Best Distance: 70
Iteration 38/100, Best Distance: 70
Iteration 39/100, Best Distance: 70
Iteration 40/100, Best Distance: 70
Iteration 41/100, Best Distance: 70
Iteration 42/100, Best Distance: 70
Iteration 43/100, Best Distance: 70
Iteration 44/100, Best Distance: 70
Iteration 45/100, Best Distance: 70
Iteration 46/100, Best Distance: 70
Iteration 47/100, Best Distance: 70
Iteration 48/100, Best Distance: 70
Iteration 49/100, Best Distance: 70
Iteration 50/100, Best Distance: 70
Iteration 51/100, Best Distance: 70
Iteration 52/100, Best Distance: 70
Iteration 53/100, Best Distance: 70
Iteration 54/100, Best Distance: 70
Iteration 55/100, Best Distance: 70
Iteration 56/100, Best Distance: 70
Iteration 57/100, Best Distance: 70
Iteration 58/100, Best Distance: 70
```

```python
import numpy as np
import random

class GreyWolfOptimization:
    def __init__(self, distance_matrix, n_wolves, n_iterations, alpha=0.5, beta=1.5, delta=1.5):
        self.distance_matrix = distance_matrix  # Distance matrix
        self.num_cities = len(distance_matrix)  # Number of cities
        self.n_wolves = n_wolves  # Number of wolves
        self.n_iterations = n_iterations  # Number of iterations
        self.alpha = alpha  # Coefficient for alpha wolf
        self.beta = beta  # Coefficient for beta wolf
        self.delta = delta  # Coefficient for delta wolf
        self.wolves = np.array([self._generate_solution() for _ in range(n_wolves)])  # Initial solutions (wolves)
        self.best_solution = None  # Best solution
        self.best_distance = float('inf')  # Best distance (fitness)

    def _generate_solution(self):
        # Generate a random solution (random permutation of cities)
        return np.random.permutation(self.num_cities)

    def _calculate_total_distance(self, tour):
        # Calculate the total distance of the tour
        total_distance = 0
        for i in range(len(tour) - 1):
            total_distance += self.distance_matrix[tour[i]][tour[i + 1]]
        total_distance += self.distance_matrix[tour[-1]][tour[0]]  # Return to the starting city
        return total_distance

    def _update_wolf_position(self, wolf, A, C, alpha_wolf, beta_wolf, delta_wolf):
        # Update position of the wolf using the formula for alpha, beta, and delta wolves
        new_wolf_position = wolf + A * (alpha_wolf - wolf) + C * (beta_wolf - wolf) + C * (delta_wolf - wolf)
        # Make sure the solution is within the valid range [0, num_cities-1] and is a valid permutation
        new_wolf_position = np.clip(new_wolf_position, 0, self.num_cities - 1).astype(int)
        unique_wolf = np.unique(new_wolf_position)
        if len(unique_wolf) < self.num_cities:
            missing_cities = set(range(self.num_cities)) - set(unique_wolf)
            new_wolf_position = np.array(list(unique_wolf) + list(missing_cities))

        return np.random.permutation(new_wolf_position)  # Ensure it's a valid permutation
```

```
    def run(self):
        # Main Grey Wolf Optimization loop
        for iteration in range(self.n_iterations):
            fitness = np.array([self._calculate_total_distance(wolf) for wolf in self.wolves])

            # Update the best solution found so far
            best_wolf_idx = np.argmin(fitness)
            best_solution = self.wolves[best_wolf_idx]
            best_distance = fitness[best_wolf_idx]

            if best_distance < self.best_distance:
                self.best_solution = best_solution
                self.best_distance = best_distance

            # Get alpha, beta, and delta wolves
            sorted_indices = np.argsort(fitness)
            alpha_wolf = self.wolves[sorted_indices[0]]
            beta_wolf = self.wolves[sorted_indices[1]]
            delta_wolf = self.wolves[sorted_indices[2]]

            # Update positions of the wolves using the GWO formula
            A = 2 * np.random.random(self.wolves.shape) - 1  # Random vector for A
            C = 2 * np.random.random(self.wolves.shape)  # Random vector for C

            new_wolves = []
            for i, wolf in enumerate(self.wolves):
                new_position = self._update_wolf_position(wolf, A[i], C[i], alpha_wolf, beta_wolf, delta_wolf)
                new_wolves.append(new_position)
            self.wolves = np.array(new_wolves)

            print(f"Iteration {iteration + 1}/{self.n_iterations}, Best Distance: {self.best_distance}")

        return self.best_solution, self.best_distance


# Example usage
if __name__ == "__main__":
    # Example distance matrix for 5 cities
    distance_matrix = np.array([
        [0, 10, 15, 20, 25],
        [10, 0, 35, 25, 30],
        [15, 35, 0, 30, 5],
        [20, 25, 30, 0, 15],
        [25, 30, 5, 15, 0]
    ])

    # Parameters: n_wolves, n_iterations, alpha, beta, delta
    gwo = GreyWolfOptimization(distance_matrix, n_wolves=10, n_iterations=100, alpha=0.5, beta=1.5, delta=1.5)

    best_solution, best_distance = gwo.run()

    print(f"Best solution: {best_solution}")
    print(f"Best distance: {best_distance}")
```

```
Iteration 1/100, Best Distance: 70
Iteration 2/100, Best Distance: 70
Iteration 3/100, Best Distance: 70
Iteration 4/100, Best Distance: 70
Iteration 5/100, Best Distance: 70
Iteration 6/100, Best Distance: 70
Iteration 7/100, Best Distance: 70
Iteration 8/100, Best Distance: 70
Iteration 9/100, Best Distance: 70
Iteration 10/100, Best Distance: 70
Iteration 11/100, Best Distance: 70
Iteration 12/100, Best Distance: 70
Iteration 13/100, Best Distance: 70
Iteration 14/100, Best Distance: 70
Iteration 15/100, Best Distance: 70
Iteration 16/100, Best Distance: 70
Iteration 17/100, Best Distance: 70
Iteration 18/100, Best Distance: 70
Iteration 19/100, Best Distance: 70
Iteration 20/100, Best Distance: 70
Iteration 21/100, Best Distance: 70
Iteration 22/100, Best Distance: 70
Iteration 23/100, Best Distance: 70
Iteration 24/100, Best Distance: 70
Iteration 25/100, Best Distance: 70
Iteration 26/100, Best Distance: 70
Iteration 27/100, Best Distance: 70
Iteration 28/100, Best Distance: 70
Iteration 29/100, Best Distance: 70
```

```
Iteration 30/100, Best Distance: 70
Iteration 31/100, Best Distance: 70
Iteration 32/100, Best Distance: 70
Iteration 33/100, Best Distance: 70
Iteration 34/100, Best Distance: 70
Iteration 35/100, Best Distance: 70
Iteration 36/100, Best Distance: 70
Iteration 37/100, Best Distance: 70
Iteration 38/100, Best Distance: 70
Iteration 39/100, Best Distance: 70
Iteration 40/100, Best Distance: 70
Iteration 41/100, Best Distance: 70
Iteration 42/100, Best Distance: 70
Iteration 43/100, Best Distance: 70
Iteration 44/100, Best Distance: 70
Iteration 45/100, Best Distance: 70
Iteration 46/100, Best Distance: 70
Iteration 47/100, Best Distance: 70
Iteration 48/100, Best Distance: 70
Iteration 49/100, Best Distance: 70
Iteration 50/100, Best Distance: 70
Iteration 51/100, Best Distance: 70
Iteration 52/100, Best Distance: 70
Iteration 53/100, Best Distance: 70
Iteration 54/100, Best Distance: 70
Iteration 55/100, Best Distance: 70
Iteration 56/100, Best Distance: 70
Iteration 57/100, Best Distance: 70
Iteration 58/100, Best Distance: 70
```

```
import numpy as np
from multiprocessing import Pool
import random


def two_opt(tour, distance_matrix):
    """
    Performs the 2-opt optimization on the given tour.
    """
    best_tour = tour
    best_distance = calculate_distance(tour, distance_matrix)
    improved = True

    while improved:
        improved = False
        for i in range(1, len(best_tour) - 2):
            for j in range(i + 1, len(best_tour)):
                if j - i == 1:  # Skip adjacent nodes
                    continue
                new_tour = best_tour[:i] + best_tour[i:j + 1][::-1] + best_tour[j + 1:]
                new_distance = calculate_distance(new_tour, distance_matrix)
                if new_distance < best_distance:
                    best_tour = new_tour
                    best_distance = new_distance
                    improved = True
    return best_tour, best_distance


def calculate_distance(tour, distance_matrix):
    """
    Calculates the total distance of the given tour.
    """
    distance = 0
    for i in range(len(tour) - 1):
        distance += distance_matrix[tour[i]][tour[i + 1]]
    distance += distance_matrix[tour[-1]][tour[0]]  # Return to start
    return distance


class ParallelCellularOptimization:
    def __init__(self, distance_matrix, n_cells=4, n_iterations=100, alpha=0.5, migration_prob=0.2):
        self.distance_matrix = distance_matrix
        self.n_cells = n_cells
        self.n_iterations = n_iterations
        self.alpha = alpha
        self.migration_prob = migration_prob

    def _local_optimization(self, cell_solution):
        """
        Applies local optimization using 2-opt for a single cell.
        """
        return two_opt(cell_solution, self.distance_matrix)

    def _migrate(self, populations):
        """
```

```
            Handles migration between populations with the given probability.
            """
            for i in range(len(populations) - 1):
                if random.random() < self.migration_prob:
                    swap_idx = random.randint(0, len(populations[i]) - 1)
                    populations[i][swap_idx], populations[i + 1][swap_idx] = populations[i + 1][swap_idx], populations[i][swap_idx]
            return populations

        def run(self):
            """
            Runs the parallel cellular optimization.
            """
            n_cities = len(self.distance_matrix)
            populations = [
                random.sample(range(n_cities), n_cities) for _ in range(self.n_cells)
            ]
            best_solution = None
            best_distance = float('inf')

            for _ in range(self.n_iterations):
                # Parallel optimization
                with Pool(processes=self.n_cells) as pool:
                    results = pool.map(self._local_optimization, populations)

                # Extract the best solution
                for solution, distance in results:
                    if distance < best_distance:
                        best_solution = solution
                        best_distance = distance

                # Update populations for migration
                populations = [res[0] for res in results]
                populations = self._migrate(populations)

            return best_solution, best_distance


    if __name__ == "__main__":
        # Example distance matrix (symmetric TSP)
        distance_matrix = np.random.randint(10, 100, size=(10, 10))
        np.fill_diagonal(distance_matrix, 0)

        # Ensure the matrix is symmetric
        distance_matrix = (distance_matrix + distance_matrix.T) // 2

        pco = ParallelCellularOptimization(distance_matrix, n_cells=4, n_iterations=100, alpha=0.5, migration_prob=0.2)

        best_solution, best_distance = pco.run()

        print(f"Best solution: {best_solution}")
        print(f"Best distance: {best_distance}")
```

```
Best solution: [5, 2, 2, 2, 0, 3, 0, 4, 9, 9]
Best distance: 207
```

```
import numpy as np
import random


class GeneticAlgorithm:
    def __init__(self, fitness_function, chromosome_length, population_size, mutation_rate, crossover_rate, generations):
        self.fitness_function = fitness_function
        self.chromosome_length = chromosome_length
        self.population_size = population_size
        self.mutation_rate = mutation_rate
        self.crossover_rate = crossover_rate
        self.generations = generations

    def _initialize_population(self):
        """
        Randomly initialize the population with binary chromosomes.
        """
        return np.random.randint(2, size=(self.population_size, self.chromosome_length))

    def _decode_chromosome(self, chromosome, lower_bound, upper_bound):
        """
        Decode binary chromosome to a real number within a given range.
        """
        decimal_value = int("".join(map(str, chromosome)), 2)
        max_decimal = 2**self.chromosome_length - 1
        return lower_bound + (upper_bound - lower_bound) * decimal_value / max_decimal
```

```
        def _evaluate_fitness(self, population, lower_bound, upper_bound):
            """
            Evaluate the fitness of the population.
            """
            decoded_values = np.array(
                [self._decode_chromosome(chromosome, lower_bound, upper_bound) for chromosome in population]
            )
            return np.array([self.fitness_function(x) for x in decoded_values])

        def _select_parents(self, population, fitness):
            """
            Select parents using roulette wheel selection.
            """
            # Adjust fitness to ensure non-negative probabilities
            adjusted_fitness = fitness - fitness.min() + 1e-6
            probabilities = adjusted_fitness / adjusted_fitness.sum()
            indices = np.random.choice(len(population), size=2, replace=False, p=probabilities)
            return population[indices[0]], population[indices[1]]

        def _crossover(self, parent1, parent2):
            """
            Perform single-point crossover on two parents.
            """
            if random.random() < self.crossover_rate:
                point = random.randint(1, self.chromosome_length - 1)
                child1 = np.concatenate((parent1[:point], parent2[point:]))
                child2 = np.concatenate((parent2[:point], parent1[point:]))
                return child1, child2
            return parent1.copy(), parent2.copy()

        def _mutate(self, chromosome):
            """
            Mutate a chromosome by flipping bits with a given mutation rate.
            """
            for i in range(len(chromosome)):
                if random.random() < self.mutation_rate:
                    chromosome[i] = 1 - chromosome[i]
            return chromosome

        def optimize(self, lower_bound, upper_bound):
            """
            Run the genetic algorithm to optimize the fitness function.
            """
            population = self._initialize_population()
            best_solution = None
            best_fitness = float("-inf")

            for generation in range(self.generations):
                fitness = self._evaluate_fitness(population, lower_bound, upper_bound)

                # Track the best solution
                max_fitness_idx = np.argmax(fitness)
                if fitness[max_fitness_idx] > best_fitness:
                    best_fitness = fitness[max_fitness_idx]
                    best_solution = population[max_fitness_idx]

                # Create the next generation
                new_population = []
                while len(new_population) < self.population_size:
                    parent1, parent2 = self._select_parents(population, fitness)
                    child1, child2 = self._crossover(parent1, parent2)
                    new_population.append(self._mutate(child1))
                    if len(new_population) < self.population_size:
                        new_population.append(self._mutate(child2))

                population = np.array(new_population)

                # Print progress
                print(f"Generation {generation + 1}: Best Fitness = {best_fitness:.4f}")

            best_decoded = self._decode_chromosome(best_solution, lower_bound, upper_bound)
            return best_decoded, best_fitness


    if __name__ == "__main__":
        # Example fitness function: maximize f(x) = -x^2 + 5x + 10
        def fitness_function(x):
            return -x**2 + 5 * x + 10

        # GA parameters
        chromosome_length = 16  # Precision of the solution
        population_size = 20
```

```
mutation_rate = 0.01
crossover_rate = 0.8
generations = 50
lower_bound = 0  # Lower bound of the search space
upper_bound = 10  # Upper bound of the search space

ga = GeneticAlgorithm(
    fitness_function, chromosome_length, population_size, mutation_rate, crossover_rate, generations
)
best_solution, best_fitness = ga.optimize(lower_bound, upper_bound)

print(f"\nBest solution: {best_solution}")
print(f"Best fitness: {best_fitness:.4f}")
```

```
Generation 1: Best Fitness = 16.1584
Generation 2: Best Fitness = 16.1874
Generation 3: Best Fitness = 16.1874
Generation 4: Best Fitness = 16.1874
Generation 5: Best Fitness = 16.1874
Generation 6: Best Fitness = 16.2156
Generation 7: Best Fitness = 16.2156
Generation 8: Best Fitness = 16.2187
Generation 9: Best Fitness = 16.2317
Generation 10: Best Fitness = 16.2496
Generation 11: Best Fitness = 16.2496
Generation 12: Best Fitness = 16.2499
Generation 13: Best Fitness = 16.2500
Generation 14: Best Fitness = 16.2500
Generation 15: Best Fitness = 16.2500
Generation 16: Best Fitness = 16.2500
Generation 17: Best Fitness = 16.2500
Generation 18: Best Fitness = 16.2500
Generation 19: Best Fitness = 16.2500
Generation 20: Best Fitness = 16.2500
Generation 21: Best Fitness = 16.2500
Generation 22: Best Fitness = 16.2500
Generation 23: Best Fitness = 16.2500
Generation 24: Best Fitness = 16.2500
Generation 25: Best Fitness = 16.2500
Generation 26: Best Fitness = 16.2500
Generation 27: Best Fitness = 16.2500
Generation 28: Best Fitness = 16.2500
Generation 29: Best Fitness = 16.2500
Generation 30: Best Fitness = 16.2500
Generation 31: Best Fitness = 16.2500
Generation 32: Best Fitness = 16.2500
Generation 33: Best Fitness = 16.2500
Generation 34: Best Fitness = 16.2500
Generation 35: Best Fitness = 16.2500
Generation 36: Best Fitness = 16.2500
Generation 37: Best Fitness = 16.2500
Generation 38: Best Fitness = 16.2500
Generation 39: Best Fitness = 16.2500
Generation 40: Best Fitness = 16.2500
Generation 41: Best Fitness = 16.2500
Generation 42: Best Fitness = 16.2500
Generation 43: Best Fitness = 16.2500
Generation 44: Best Fitness = 16.2500
Generation 45: Best Fitness = 16.2500
Generation 46: Best Fitness = 16.2500
Generation 47: Best Fitness = 16.2500
Generation 48: Best Fitness = 16.2500
Generation 49: Best Fitness = 16.2500
Generation 50: Best Fitness = 16.2500

Best solution: 2.5058365758754864
Best fitness: 16.2500
```