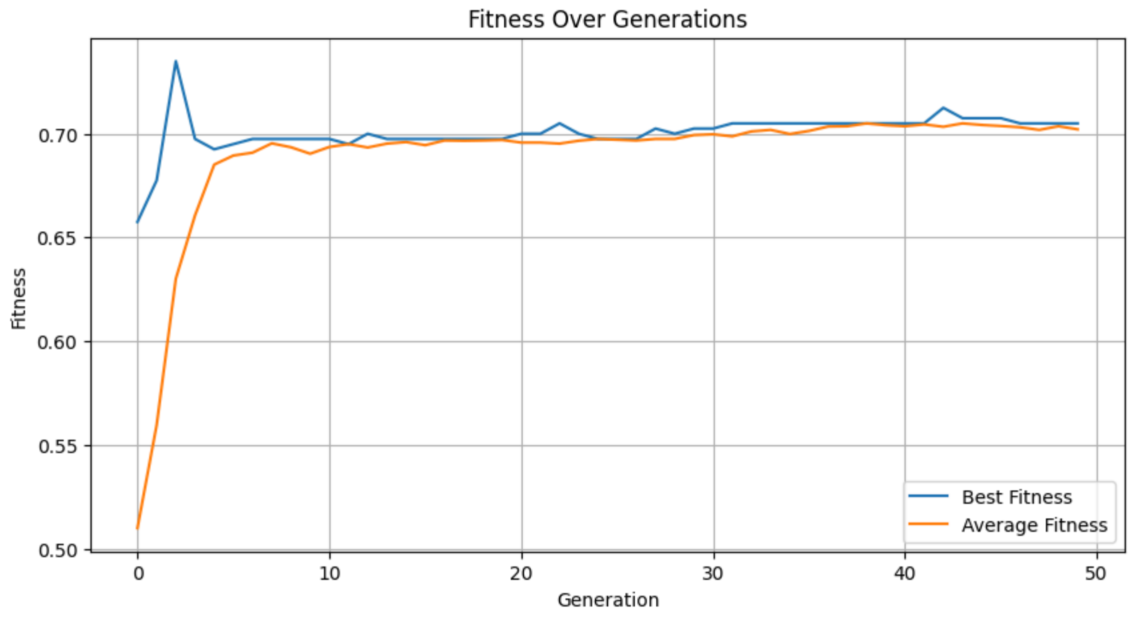
```
#genetic algorithm
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

# Creating a sample dataset
X, y = make_classification(n_samples=500, n_features=10, n_informative=8, n_classes=2)
X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2)

# Neural Network Structure
input_size = X.shape[1]
hidden_size = 5
output_size = 1

# Helper functions for the Neural Network
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def forward_pass(X, weights1, weights2):
    hidden_input = np.dot(X, weights1)
    hidden_output = sigmoid(hidden_input)
    output_input = np.dot(hidden_output, weights2)
    output = sigmoid(output_input)
    return output

def compute_fitness(weights):
    predictions = forward_pass(X_train, weights['w1'], weights['w2'])
    predictions = (predictions > 0.5).astype(int)
    accuracy = accuracy_score(y_train, predictions)
    return accuracy

# Genetic Algorithm Parameters
population_size = 20
generations = 50
mutation_rate = 0.1

# Initialize Population
population = []
for _ in range(population_size):
    individual = {
        'w1': np.random.randn(input_size, hidden_size),
        'w2': np.random.randn(hidden_size, output_size)
    }
    population.append(individual)

# Tracking performance
best_fitness_history = []
average_fitness_history = []

# Main Genetic Algorithm Loop
for generation in range(generations):
    # Evaluate Fitness of each Individual
    fitness_scores = np.array([compute_fitness(individual) for individual in population])
    best_fitness = np.max(fitness_scores)
    average_fitness = np.mean(fitness_scores)
    best_fitness_history.append(best_fitness)
    average_fitness_history.append(average_fitness)

    # Selection: Select top half of the population
    sorted_indices = np.argsort(fitness_scores)[::-1]
    population = [population[i] for i in sorted_indices[:population_size//2]]

    # Crossover and Mutation
    new_population = []
    while len(new_population) < population_size:
        parents = np.random.choice(population, 2, replace=False)
        child = {
            'w1': (parents[0]['w1'] + parents[1]['w1']) / 2,
            'w2': (parents[0]['w2'] + parents[1]['w2']) / 2
        }

        # Mutation
        if np.random.rand() < mutation_rate:
            child['w1'] += np.random.randn(*child['w1'].shape) * 0.1
            child['w2'] += np.random.randn(*child['w2'].shape) * 0.1

        new_population.append(child)

    population = new_population
    print(f"Generation {generation+1}, Best Fitness: {best_fitness:.4f}")

# Evaluate the best individual on validation set
best_individual = population[np.argmax(fitness_scores)]
predictions = forward_pass(X_val, best_individual['w1'], best_individual['w2'])
predictions = (predictions > 0.5).astype(int)
final_accuracy = accuracy_score(y_val, predictions)
print(f"Final Accuracy on Validation Set: {final_accuracy:.4f}")

# Plotting the results
plt.figure(figsize=(10, 5))
plt.plot(best_fitness_history, label='Best Fitness')
plt.plot(average_fitness_history, label='Average Fitness')
plt.title('Fitness Over Generations')
plt.xlabel('Generation')
plt.ylabel('Fitness')
plt.legend()
plt.grid(True)
plt.show()
```

```
Generation 1, Best Fitness: 0.6575
Generation 2, Best Fitness: 0.6775
Generation 3, Best Fitness: 0.7350
Generation 4, Best Fitness: 0.6975
Generation 5, Best Fitness: 0.6925
Generation 6, Best Fitness: 0.6950
Generation 7, Best Fitness: 0.6975
Generation 8, Best Fitness: 0.6975
Generation 9, Best Fitness: 0.6975
Generation 10, Best Fitness: 0.6975
Generation 11, Best Fitness: 0.6975
Generation 12, Best Fitness: 0.6950
Generation 13, Best Fitness: 0.7000
Generation 14, Best Fitness: 0.6975
Generation 15, Best Fitness: 0.6975
Generation 16, Best Fitness: 0.6975
Generation 17, Best Fitness: 0.6975
Generation 18, Best Fitness: 0.6975
Generation 19, Best Fitness: 0.6975
Generation 20, Best Fitness: 0.6975
Generation 21, Best Fitness: 0.7000
Generation 22, Best Fitness: 0.7000
Generation 23, Best Fitness: 0.7050
Generation 24, Best Fitness: 0.7000
Generation 25, Best Fitness: 0.6975
Generation 26, Best Fitness: 0.6975
Generation 27, Best Fitness: 0.6975
Generation 28, Best Fitness: 0.7025
Generation 29, Best Fitness: 0.7000
Generation 30, Best Fitness: 0.7025
Generation 31, Best Fitness: 0.7025
Generation 32, Best Fitness: 0.7050
Generation 33, Best Fitness: 0.7050
Generation 34, Best Fitness: 0.7050
Generation 35, Best Fitness: 0.7050
Generation 36, Best Fitness: 0.7050
Generation 37, Best Fitness: 0.7050
Generation 38, Best Fitness: 0.7050
Generation 39, Best Fitness: 0.7050
Generation 40, Best Fitness: 0.7050
Generation 41, Best Fitness: 0.7050
Generation 42, Best Fitness: 0.7050
Generation 43, Best Fitness: 0.7125
Generation 44, Best Fitness: 0.7075
Generation 45, Best Fitness: 0.7075
Generation 46, Best Fitness: 0.7075
Generation 47, Best Fitness: 0.7050
Generation 48, Best Fitness: 0.7050
Generation 49, Best Fitness: 0.7050
Generation 50, Best Fitness: 0.7050
Final Accuracy on Validation Set: 0.6200
```



```
# python implementation of particle swarm optimization (PSO)
# minimizing rastrigin and sphere function

import random
import math # cos() for Rastrigin
import copy # array-copying convenience
import sys    # max float
```

```
#-------fitness functions---------

# rastrigin function
def fitness_rastrigin(position):
    fitnessVal = 0.0
    for i in range(len(position)):
        xi = position[i]
        fitnessVal += (xi * xi) - (10 * math.cos(2 * math.pi * xi)) + 10
    return fitnessVal

#sphere function
def fitness_sphere(position):
    fitnessVal = 0.0
    for i in range(len(position)):
        xi = position[i]
        fitnessVal += (xi*xi);
    return fitnessVal;
#-------------------------

#particle class
class Particle:
    def __init__(self, fitness, dim, minx, maxx, seed):
        self.rnd = random.Random(seed)

        # initialize position of the particle with 0.0 value
        self.position = [0.0 for i in range(dim)]

        # initialize velocity of the particle with 0.0 value
        self.velocity = [0.0 for i in range(dim)]

        # initialize best particle position of the particle with 0.0 value
        self.best_part_pos = [0.0 for i in range(dim)]

        # loop dim times to calculate random position and velocity
        # range of position and velocity is [minx, max]
        for i in range(dim):
            self.position[i] = ((maxx - minx) * self.rnd.random() + minx)
            self.velocity[i] = ((maxx - minx) * self.rnd.random() + minx)

        # compute fitness of particle
        self.fitness = fitness(self.position) # curr fitness

        # initialize best position and fitness of this particle
        self.best_part_pos = copy.copy(self.position)
        self.best_part_fitnessVal = self.fitness # best fitness

# particle swarm optimization function
def pso(fitness, max_iter, n, dim, minx, maxx):
    # hyper parameters
    w = 0.729 # inertia
    c1 = 1.49445 # cognitive (particle)
    c2 = 1.49445 # social (swarm)

    rnd = random.Random(0)

    # create n random particles
    swarm = [Particle(fitness, dim, minx, maxx, i) for i in range(n)]

    # compute the value of best_position and best_fitness in swarm
    best_swarm_pos = [0.0 for i in range(dim)]
    best_swarm_fitnessVal = sys.float_info.max # swarm best

    # computer best particle of swarm and it's fitness
    for i in range(n): # check each particle
        if swarm[i].fitness < best_swarm_fitnessVal:
            best_swarm_fitnessVal = swarm[i].fitness
            best_swarm_pos = copy.copy(swarm[i].position)

    # main loop of pso
    Iter = 0
    while Iter < max_iter:

        # after every 10 iterations
        # print iteration number and best fitness value so far
        if Iter % 10 == 0 and Iter > 1:
            print("Iter = " + str(Iter) + " best fitness = %.3f" % best_swarm_fitnessVal)

        for i in range(n): # process each particle

            # compute new velocity of curr particle
            for k in range(dim):
                r1 = rnd.random() # randomizations
                r2 = rnd.random()

                swarm[i].velocity[k] = (
                                    (w * swarm[i].velocity[k]) +
                                    (c1 * r1 * (swarm[i].best_part_pos[k] - swarm[i].position[k])) +
                                    (c2 * r2 * (best_swarm_pos[k] -swarm[i].position[k]))
                                )


                # if velocity[k] is not in [minx, max]
                # then clip it
                if swarm[i].velocity[k] < minx:
                    swarm[i].velocity[k] = minx
                elif swarm[i].velocity[k] > maxx:
                    swarm[i].velocity[k] = maxx


            # compute new position using new velocity
            for k in range(dim):
                swarm[i].position[k] += swarm[i].velocity[k]

            # compute fitness of new position
            swarm[i].fitness = fitness(swarm[i].position)

            # is new position a new best for the particle?
            if swarm[i].fitness < swarm[i].best_part_fitnessVal:
                swarm[i].best_part_fitnessVal = swarm[i].fitness
                swarm[i].best_part_pos = copy.copy(swarm[i].position)

            # is new position a new best overall?
            if swarm[i].fitness < best_swarm_fitnessVal:
                best_swarm_fitnessVal = swarm[i].fitness
                best_swarm_pos = copy.copy(swarm[i].position)

        # for-each particle
        Iter += 1
    #end_while
    return best_swarm_pos
# end pso


#-----------------------------
# Driver code for rastrigin function

print("\nBegin particle swarm optimization on rastrigin function\n")
dim = 3
fitness = fitness_rastrigin


print("Goal is to minimize Rastrigin's function in " + str(dim) + " variables")
print("Function has known min = 0.0 at (", end="")
for i in range(dim-1):
    print("0, ", end="")
print("0)")

num_particles = 50
max_iter = 100

print("Setting num_particles = " + str(num_particles))
print("Setting max_iter = " + str(max_iter))
print("\nStarting PSO algorithm\n")



best_position = pso(fitness, max_iter, num_particles, dim, -10.0, 10.0)

print("\nPSO completed\n")
print("\nBest solution found:")
print(["%.6f"%best_position[k] for k in range(dim)])
fitnessVal = fitness(best_position)
print("fitness of best solution = %.6f" % fitnessVal)

print("\nEnd particle swarm for rastrigin function\n")


print()
print()


# Driver code for Sphere function
print("\nBegin particle swarm optimization on sphere function\n")
dim = 3
```

```
fitness = fitness_sphere


print("Goal is to minimize sphere function in " + str(dim) + " variables")
print("Function has known min = 0.0 at (", end="")
for i in range(dim-1):
    print("0, ", end="")
print("0)")

num_particles = 50
max_iter = 100

print("Setting num_particles = " + str(num_particles))
print("Setting max_iter = " + str(max_iter))
print("\nStarting PSO algorithm\n")



best_position = pso(fitness, max_iter, num_particles, dim, -10.0, 10.0)

print("\nPSO completed\n")
print("\nBest solution found:")
print(["%.6f"%best_position[k] for k in range(dim)])
fitnessVal = fitness(best_position)
print("fitness of best solution = %.6f" % fitnessVal)

print("\nEnd particle swarm for sphere function\n")
```

```
Goal is to minimize Rastrigin's function in 3 variables
Function has known min = 0.0 at (0, 0, 0)
Setting num_particles = 50
Setting max_iter = 100

Starting PSO algorithm

Iter = 10 best fitness = 8.463
Iter = 20 best fitness = 4.792
Iter = 30 best fitness = 2.223
Iter = 40 best fitness = 0.251
Iter = 50 best fitness = 0.251
Iter = 60 best fitness = 0.061
Iter = 70 best fitness = 0.007
Iter = 80 best fitness = 0.005
Iter = 90 best fitness = 0.000

PSO completed


Best solution found:
['0.000618', '0.000013', '0.000616']
fitness of best solution = 0.000151

End particle swarm for rastrigin function




Begin particle swarm optimization on sphere function

Goal is to minimize sphere function in 3 variables
Function has known min = 0.0 at (0, 0, 0)
Setting num_particles = 50
Setting max_iter = 100

Starting PSO algorithm

Iter = 10 best fitness = 0.189
Iter = 20 best fitness = 0.012
Iter = 30 best fitness = 0.001
Iter = 40 best fitness = 0.000
Iter = 50 best fitness = 0.000
Iter = 60 best fitness = 0.000
Iter = 70 best fitness = 0.000
Iter = 80 best fitness = 0.000
Iter = 90 best fitness = 0.000

PSO completed


Best solution found:
['0.000004', '-0.000001', '0.000007']
fitness of best solution = 0.000000

End particle swarm for sphere function
```

```
#ant colony
import numpy as np
import random

# Parameters
NUM_ANTS = 10          # Number of ants
NUM_CITIES = 5         # Number of cities
MAX_ITERATIONS = 100   # Number of iterations
ALPHA = 1.0            # Influence of pheromone
BETA = 2.0             # Influence of distance
EVAPORATION_RATE = 0.5 # Evaporation rate of pheromone
Q = 100                # Pheromone constant
INIT_PHEROMONE = 0.1   # Initial pheromone level

# Generate random distance matrix for TSP (symmetric graph)
def generate_distance_matrix(num_cities):
    distance_matrix = np.random.randint(1, 100, size=(num_cities, num_cities))
    np.fill_diagonal(distance_matrix, 0)  # Distance to itself is 0
    return (distance_matrix + distance_matrix.T) // 2  # Symmetric matrix

# Initialize pheromone matrix
def initialize_pheromones(num_cities, init_value):
    return np.full((num_cities, num_cities), init_value)

# Select next city based on probabilities
def select_next_city(current_city, visited, pheromones, distances):
    probabilities = []
    for city in range(len(distances)):
        if city not in visited:
            pheromone = pheromones[current_city][city] ** ALPHA
            heuristic = (1.0 / distances[current_city][city]) ** BETA
            probabilities.append(pheromone * heuristic)
        else:
            probabilities.append(0)

    probabilities = np.array(probabilities) / sum(probabilities)
    next_city = np.random.choice(range(len(distances)), p=probabilities)
    return next_city

# Update pheromones based on ant paths
def update_pheromones(pheromones, paths, distances):
    pheromones *= (1 - EVAPORATION_RATE)  # Evaporation
    for path in paths:
        path_length = calculate_path_length(path, distances)
        pheromone_deposit = Q / path_length
        for i in range(len(path) - 1):
            city_a, city_b = path[i], path[i + 1]
            pheromones[city_a][city_b] += pheromone_deposit
            pheromones[city_b][city_a] += pheromone_deposit  # Symmetric
    return pheromones

# Calculate total length of a path
def calculate_path_length(path, distances):
    total_length = 0
    for i in range(len(path) - 1):
        total_length += distances[path[i]][path[i + 1]]
    total_length += distances[path[-1]][path[0]]  # Return to start city
    return total_length

# Ant Colony Optimization
def ant_colony_optimization(num_cities, num_ants, max_iterations):
    distances = generate_distance_matrix(num_cities)
    pheromones = initialize_pheromones(num_cities, INIT_PHEROMONE)

    best_path = None
    best_path_length = float('inf')

    for iteration in range(max_iterations):
        all_paths = []
        for ant in range(num_ants):
            visited = []
            current_city = random.randint(0, num_cities - 1)
            visited.append(current_city)

            while len(visited) < num_cities:
                next_city = select_next_city(current_city, visited, pheromones, distances)
                visited.append(next_city)
                current_city = next_city

            visited.append(visited[0])  # Return to start city
            all_paths.append(visited)
            path_length = calculate_path_length(visited, distances)
            if path_length < best_path_length:
                best_path = visited
                best_path_length = path_length

        # Update pheromones
        pheromones = update_pheromones(pheromones, all_paths, distances)

        print(f"Iteration {iteration + 1}: Best Path Length = {best_path_length}")

    print("\nBest Path:", best_path)
    print("Best Path Length:", best_path_length)

# Run ACO for TSP
if __name__ == "__main__":
    ant_colony_optimization(NUM_CITIES, NUM_ANTS, MAX_ITERATIONS)
```

```
Iteration 1: Best Path Length = 216
Iteration 2: Best Path Length = 216
Iteration 3: Best Path Length = 216
Iteration 4: Best Path Length = 216
Iteration 5: Best Path Length = 216
Iteration 6: Best Path Length = 216
Iteration 7: Best Path Length = 216
Iteration 8: Best Path Length = 216
Iteration 9: Best Path Length = 216
Iteration 10: Best Path Length = 216
Iteration 11: Best Path Length = 216
Iteration 12: Best Path Length = 216
Iteration 13: Best Path Length = 216
Iteration 14: Best Path Length = 216
Iteration 15: Best Path Length = 216
Iteration 16: Best Path Length = 216
Iteration 17: Best Path Length = 216
Iteration 18: Best Path Length = 216
Iteration 19: Best Path Length = 216
Iteration 20: Best Path Length = 216
Iteration 21: Best Path Length = 216
Iteration 22: Best Path Length = 216
Iteration 23: Best Path Length = 216
Iteration 24: Best Path Length = 216
Iteration 25: Best Path Length = 216
Iteration 26: Best Path Length = 216
Iteration 27: Best Path Length = 216
Iteration 28: Best Path Length = 216
Iteration 29: Best Path Length = 216
Iteration 30: Best Path Length = 216
Iteration 31: Best Path Length = 216
Iteration 32: Best Path Length = 216
Iteration 33: Best Path Length = 216
Iteration 34: Best Path Length = 216
Iteration 35: Best Path Length = 216
Iteration 36: Best Path Length = 216
Iteration 37: Best Path Length = 216
Iteration 38: Best Path Length = 216
Iteration 39: Best Path Length = 216
Iteration 40: Best Path Length = 216
Iteration 41: Best Path Length = 216
Iteration 42: Best Path Length = 216
Iteration 43: Best Path Length = 216
Iteration 44: Best Path Length = 216
Iteration 45: Best Path Length = 216
Iteration 46: Best Path Length = 216
Iteration 47: Best Path Length = 216
Iteration 48: Best Path Length = 216
Iteration 49: Best Path Length = 216
Iteration 50: Best Path Length = 216
Iteration 51: Best Path Length = 216
Iteration 52: Best Path Length = 216
Iteration 53: Best Path Length = 216
Iteration 54: Best Path Length = 216
Iteration 55: Best Path Length = 216
Iteration 56: Best Path Length = 216
Iteration 57: Best Path Length = 216
Iteration 58: Best Path Length = 216
```

```
import random
import networkx as nx
import numpy as np
import math
import datetime
import pandas as pd

class Cuckoo:
    def __init__(self, path, G, eps = 0.9):
        self.path = path
        self.G = G
        self.nodes = list(G.nodes)
        self.eps = eps
        self.fitness = self.calculate_fitness()

    """
    Function to Compute fitness value.
    """
    def calculate_fitness(self):
        fitness = 0.0

        for i in range(1, len(self.path)):
            total_distance = 0
            curr_node = self.path[i-1]
            next_node = self.path[i]
            if self.G.has_edge(curr_node, next_node):
                fitness += self.G[curr_node][next_node]['weight']
            else:
                fitness += 0
        fitness = np.power(abs(fitness + self.eps), 2)
        return fitness

    def generate_new_path(self):
        """
        This function generates a random solution (a random path) in the graph
        """
        nodes = list(self.G.nodes)
        start = nodes[0]
        end = nodes[-1]
        samples = list(nx.all_simple_paths(self.G, start, end))
        for i in range(len(samples)):
            if len(samples[i]) != len(nodes):
                extra_nodes = [node for node in nodes if node not in samples[i]]
                random.shuffle(extra_nodes)
                samples[i] = samples[i] + extra_nodes

        sample_node = random.choice(samples)
        return sample_node

class CuckooSearch:
    def __init__(self, G, num_cuckoos, max_iterations, beta):
        self.G = G
        self.nodes = list(G.nodes)
        self.num_cuckoos = num_cuckoos
        self.max_iterations = max_iterations
        self.beta = beta
        self.cuckoos = [Cuckoo(random.sample(self.nodes, len(self.nodes)), self.G) for _ in range(self.num_cuckoos)]
        self.test_results = []
        self.test_cases = 0

    """
    Function to buld new nests at new location and abandon old ones using Levi flights.
    """
    def levy_flight(self):
        sigma = (math.gamma(1 + self.beta) * np.sin(np.pi * self.beta / 2) / (math.gamma((1 + self.beta) / 2) * self.beta * 2 ** ((self.
        u = np.random.normal(0, sigma, 1)
        v = np.random.normal(0, 1, 1)
        step = u / (abs(v) ** (1 / self.beta))
        return step

    def optimize(self):
        for i in range(self.max_iterations):
            for j in range(self.num_cuckoos):
                cuckoo = self.cuckoos[j]
                step = self.levy_flight()
                new_path = cuckoo.generate_new_path()
                new_cuckoo = Cuckoo(new_path, self.G)
                if new_cuckoo.fitness > cuckoo.fitness:
                    self.cuckoos[j] = new_cuckoo
                    self.test_cases+=1

            self.cuckoos = sorted(self.cuckoos, key=lambda x: x.fitness, reverse=True)
            best_path=self.cuckoos[0].path
            best_fitness=self.cuckoos[0].fitness

            self.test_results.append([i, best_fitness, self.test_cases])

        last_node = list(self.G.nodes)[-1]
        last_node_index = best_path.index(last_node) + 1

        return best_path[:last_node_index], best_fitness
```

```
if __name__ == "__main__":
    """
    Example usage
    """
    Gn = nx.DiGraph()

    #Add nodes to the graph
    for i in range(11):
        Gn.add_node(i)

    edges = [(0, 1,{'weight': 1}), (1, 3,{'weight': 2}), (1, 2,{'weight': 1}),(2, 4,{'weight': 2}),
            (3, 2,{'weight': 2}),(3, 4,{'weight': 1}),(3, 5,{'weight': 2}),(3, 7,{'weight': 4}),
            (4, 5,{'weight': 1}),(4, 6,{'weight': 2}),(5, 7,{'weight': 2}),(5, 8,{'weight': 3}),
            (6, 7,{'weight': 1}),(7, 9,{'weight': 2}),(8, 10,{'weight': 2}),(9, 10,{'weight': 1})]

    Gn.add_edges_from(edges)

    csa = CuckooSearch(Gn, num_cuckoos = 30, max_iterations=1000, beta=0.27)

    start = datetime.datetime.now()
    best_path, best_fitness = csa.optimize()
    end = datetime.datetime.now()

    csa_time = end - start

    csa_test_data = pd.DataFrame(csa.test_results,columns = ["iterations","fitness_value","test_cases"])

    print("Optimal path: ", best_path)
    print("Optimal path cost: ", best_fitness)
    print("CSA total Exec time => ", csa_time.total_seconds())
    csa_test_data.to_csv("csa_test_data_results.csv")
```

```
Optimal path:  [0, 1, 3, 7, 9, 10]
Optimal path cost:  320.40999999999997
CSA total Exec time =>  12.944212
```

```
# Python implementation of Grey Wolf Optimization (GWO)
# Minimizing Rastrigin and Sphere function

import random
import math  # cos() for Rastrigin
import copy  # array-copying convenience
import sys    # max float

#-------fitness functions---------

# Rastrigin function
def fitness_rastrigin(position):
    fitness_value = 0.0
    for i in range(len(position)):
        xi = position[i]
        fitness_value += (xi * xi) - (10 * math.cos(2 * math.pi * xi)) + 10
    return fitness_value

# Sphere function
def fitness_sphere(position):
    fitness_value = 0.0
    for i in range(len(position)):
        xi = position[i]
        fitness_value += (xi * xi)
    return fitness_value

#-------------------------

# Wolf class
class wolf:
    def __init__(self, fitness, dim, minx, maxx, seed):
        self.rnd = random.Random(seed)
        self.position = [0.0 for i in range(dim)]

        for i in range(dim):
            self.position[i] = ((maxx - minx) * self.rnd.random() + minx)

        self.fitness = fitness(self.position)  # current fitness

# Grey Wolf Optimization (GWO)
def gwo(fitness, max_iter, n, dim, minx, maxx):
    rnd = random.Random(0)

    # Create n random wolves
    population = [wolf(fitness, dim, minx, maxx, i) for i in range(n)]
    # On the basis of fitness values of wolves, sort the population in ascending order
    population = sorted(population, key=lambda temp: temp.fitness)

    # Best 3 solutions will be called as alpha, beta and gamma
    alpha_wolf, beta_wolf, gamma_wolf = copy.copy(population[:3])

    # Main loop of GWO
    Iter = 0
    while Iter < max_iter:

        # After every 10 iterations, print iteration number and best fitness value so far
        if Iter % 10 == 0 and Iter > 1:
            print("Iter = " + str(Iter) + " best fitness = %.3f" % alpha_wolf.fitness)

        # Linearly decreased from 2 to 0
        a = 2 * (1 - Iter / max_iter)

        # Updating each population member with the help of best three members
        for i in range(n):
            A1, A2, A3 = a * (2 * rnd.random() - 1), a * (2 * rnd.random() - 1), a * (2 * rnd.random() - 1)
            C1, C2, C3 = 2 * rnd.random(), 2 * rnd.random(), 2 * rnd.random()

            X1 = [0.0 for i in range(dim)]
            X2 = [0.0 for i in range(dim)]
            X3 = [0.0 for i in range(dim)]
            Xnew = [0.0 for i in range(dim)]

            for j in range(dim):
                X1[j] = alpha_wolf.position[j] - A1 * abs(C1 * alpha_wolf.position[j] - population[i].position[j])
                X2[j] = beta_wolf.position[j] - A2 * abs(C2 * beta_wolf.position[j] - population[i].position[j])
                X3[j] = gamma_wolf.position[j] - A3 * abs(C3 * gamma_wolf.position[j] - population[i].position[j])
                Xnew[j] += X1[j] + X2[j] + X3[j]

            for j in range(dim):
                Xnew[j] /= 3.0

            # Fitness calculation of new solution
            fnew = fitness(Xnew)

            # Greedy selection
            if fnew < population[i].fitness:
                population[i].position = Xnew
                population[i].fitness = fnew

        # On the basis of fitness values of wolves, sort the population in ascending order
        population = sorted(population, key=lambda temp: temp.fitness)

        # Best 3 solutions will be called as alpha, beta, and gamma
        alpha_wolf, beta_wolf, gamma_wolf = copy.copy(population[:3])

        Iter += 1
    # End-while

    # Returning the best solution
    return alpha_wolf.position

#----------------------------

# Driver code for Rastrigin function

print("\nBegin grey wolf optimization on Rastrigin function\n")
dim = 3
fitness = fitness_rastrigin

print("Goal is to minimize Rastrigin's function in " + str(dim) + " variables")
print("Function has known min = 0.0 at (", end="")
for i in range(dim-1):
    print("0, ", end="")
print("0)")

num_particles = 50
max_iter = 100

print("Setting num_particles = " + str(num_particles))
print("Setting max_iter = " + str(max_iter))
print("\nStarting GWO algorithm\n")

best_position = gwo(fitness, max_iter, num_particles, dim, -10.0, 10.0)

print("\nGWO completed\n")
print("\nBest solution found:")
print(["%.6f" % best_position[k] for k in range(dim)])
err = fitness(best_position)
print("Fitness of best solution = %.6f" % err)

print("\nEnd GWO for Rastrigin\n")

# Driver code for Sphere function
print("\nBegin grey wolf optimization on Sphere function\n")
dim = 3
fitness = fitness_sphere

print("Goal is to minimize Sphere function in " + str(dim) + " variables")
print("Function has known min = 0.0 at (", end="")
for i in range(dim-1):
    print("0, ", end="")
print("0)")

num_particles = 50
max_iter = 100

print("Setting num_particles = " + str(num_particles))
print("Setting max_iter = " + str(max_iter))
print("\nStarting GWO algorithm\n")

best_position = gwo(fitness, max_iter, num_particles, dim, -10.0, 10.0)

print("\nGWO completed\n")
print("\nBest solution found:")
print(["%.6f" % best_position[k] for k in range(dim)])
err = fitness(best_position)
print("Fitness of best solution = %.6f" % err)

print("\nEnd GWO for Sphere\n")
```

```
Begin grey wolf optimization on Rastrigin function

Goal is to minimize Rastrigin's function in 3 variables
Function has known min = 0.0 at (0, 0, 0)
Setting num_particles = 50
Setting max_iter = 100

Starting GWO algorithm

Iter = 10 best fitness = 6.636
Iter = 20 best fitness = 1.047
Iter = 30 best fitness = 1.012
Iter = 40 best fitness = 1.010
Iter = 50 best fitness = 1.008
Iter = 60 best fitness = 1.008
Iter = 70 best fitness = 1.008
Iter = 80 best fitness = 1.006
Iter = 90 best fitness = 1.005

GWO completed


Best solution found:
['-0.004395', '0.995042', '0.005800']
Fitness of best solution = 1.005465

End GWO for Rastrigin


Begin grey wolf optimization on Sphere function

Goal is to minimize Sphere function in 3 variables
Function has known min = 0.0 at (0, 0, 0)
Setting num_particles = 50
Setting max_iter = 100

Starting GWO algorithm

Iter = 10 best fitness = 0.000
Iter = 20 best fitness = 0.000
Iter = 30 best fitness = 0.000
Iter = 40 best fitness = 0.000
Iter = 50 best fitness = 0.000
Iter = 60 best fitness = 0.000
Iter = 70 best fitness = 0.000
Iter = 80 best fitness = 0.000
Iter = 90 best fitness = 0.000

GWO completed
```

```
Best solution found:
['0.000000', '0.000000', '-0.000000']
Fitness of best solution = 0.000000

End GWO for Sphere
```

```
import random
import multiprocessing

# Define the update rule function for each cell
# Here, a simple rule: New state = XOR of current cell and its two neighbors
def update_cell(index, grid, new_grid):
    # Handle the edge cells (leftmost and rightmost)
    left = grid[index - 1] if index > 0 else 0
    right = grid[index + 1] if index < len(grid) - 1 else 0

    # Rule: XOR of the current cell and its two neighbors
    new_grid[index] = left ^ grid[index] ^ right

# Parallel update function that processes all cells
def parallel_update(grid):
    # Create a new grid to store updated values
    new_grid = grid[:]

    # Create a pool of workers to update cells in parallel
    with multiprocessing.Pool(processes=multiprocessing.cpu_count()) as pool:
        pool.starmap(update_cell, [(i, grid, new_grid) for i in range(len(grid))])

    return new_grid

# Function to run the automaton for a number of generations
def run_automaton(num_generations, grid_size):
    # Initialize the grid with random states (0 or 1)
    grid = [random.randint(0, 1) for _ in range(grid_size)]

    print("Initial Grid: ", grid)

    for generation in range(num_generations):
        print(f"Generation {generation + 1}: {grid}")
        grid = parallel_update(grid)

    return grid

if __name__ == '__main__':
    grid_size = 20  # Size of the grid (number of cells)
    num_generations = 10  # Number of generations to run

    final_grid = run_automaton(num_generations, grid_size)
    print("Final Grid: ", final_grid)
```

```
Initial Grid:  [0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 0, 1, 0, 1, 0, 0, 1, 1, 0, 0]
Generation 1: [0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 0, 1, 0, 1, 0, 0, 1, 1, 0, 0]
Generation 2: [0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 0, 1, 0, 1, 0, 0, 1, 1, 0, 0]
Generation 3: [0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 0, 1, 0, 1, 0, 0, 1, 1, 0, 0]
Generation 4: [0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 0, 1, 0, 1, 0, 0, 1, 1, 0, 0]
Generation 5: [0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 0, 1, 0, 1, 0, 0, 1, 1, 0, 0]
Generation 6: [0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 0, 1, 0, 1, 0, 0, 1, 1, 0, 0]
Generation 7: [0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 0, 1, 0, 1, 0, 0, 1, 1, 0, 0]
Generation 8: [0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 0, 1, 0, 1, 0, 0, 1, 1, 0, 0]
Generation 9: [0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 0, 1, 0, 1, 0, 0, 1, 1, 0, 0]
Generation 10: [0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 0, 1, 0, 1, 0, 0, 1, 1, 0, 0]
Final Grid:  [0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 0, 1, 0, 1, 0, 0, 1, 1, 0, 0]
```

```
import random
import string
import operator
import math

class GEP:
    def __init__(self, population_size, max_depth, max_generations, mutation_rate, crossover_rate):
        self.population_size = population_size
        self.max_depth = max_depth
        self.max_generations = max_generations
        self.mutation_rate = mutation_rate
        self.crossover_rate = crossover_rate
        self.operators = ['+', '-', '*', '/', '**']
        self.terminals = ['x', 'y', '1', '2', '3', '4', '5']
        self.population = self.initialize_population()

    def initialize_population(self):
        population = []
        for _ in range(self.population_size):
            individual = self.generate_individual()
            population.append(individual)
        return population

    def generate_individual(self):
        individual = []
        depth = random.randint(1, self.max_depth)
        for _ in range(depth):
            individual.append(self.generate_expression())
        return individual

    def generate_expression(self):
        expression = random.choice(self.operators + self.terminals)
        return expression

    def fitness(self, individual, x_val, y_val):
        # Create a valid expression string
        expression = ''.join(individual)
        try:
            # Replace x and y with their values
            expression = expression.replace('x', str(x_val)).replace('y', str(y_val))
            # Evaluate the expression and return its absolute value as fitness (for simplicity)
            result = eval(expression)
            if math.isinf(result) or math.isnan(result):
                return float('inf')  # Return high fitness for invalid results
            return abs(result)
        except:
            # In case of an error in evaluation (e.g., division by zero), return a high fitness value
            return float('inf')

    def selection(self):
        fitness_values = [self.fitness(ind, 1, 1) for ind in self.population]
        total_fitness = sum(fitness_values)

        # Ensure that total fitness is non-zero and finite
        if total_fitness == 0 or not math.isfinite(total_fitness):
            return random.choice(self.population)

        # Normalize fitness values and select an individual based on fitness
        selection_probs = [fitness / total_fitness for fitness in fitness_values]
        return random.choices(self.population, selection_probs)[0]

    def crossover(self, parent1, parent2):
        if random.random() < self.crossover_rate:
            # Ensure that we have valid crossover points by checking lengths
            crossover_point = random.randint(1, min(len(parent1), len(parent2)) - 1) if min(len(parent1), len(parent2)) > 1 else 0
            if crossover_point == 0:
                # If crossover point is 0, return parents unchanged
                return parent1, parent2
            offspring1 = parent1[:crossover_point] + parent2[crossover_point:]
            offspring2 = parent2[:crossover_point] + parent1[crossover_point:]
            return offspring1, offspring2
        return parent1, parent2

    def mutation(self, individual):
        if random.random() < self.mutation_rate:
            mutation_point = random.randint(0, len(individual) - 1)
            individual[mutation_point] = self.generate_expression()
        return individual

    def evolve(self):
        best_fitness = float('inf')
        best_solution = None
        for generation in range(self.max_generations):
            print(f"Generation {generation + 1}/{self.max_generations}")
            new_population = []
            for _ in range(self.population_size // 2):
                # Selection
                parent1 = self.selection()
                parent2 = self.selection()

                # Crossover
                offspring1, offspring2 = self.crossover(parent1, parent2)

                # Mutation
                offspring1 = self.mutation(offspring1)
                offspring2 = self.mutation(offspring2)

                # Add offspring to the new population
                new_population.extend([offspring1, offspring2])

            # Replace the old population with the new population
            self.population = new_population

            # Evaluate the population and find the best fitness
            current_best_fitness = float('inf')
            for individual in self.population:
                fitness_value = self.fitness(individual, 1, 1)
                if fitness_value < current_best_fitness:
                    current_best_fitness = fitness_value
                    best_solution = individual

            print(f"Best fitness so far: {current_best_fitness}")
            best_fitness = current_best_fitness

        return best_solution

# Initialize GEP with parameters and evolve the population
gep = GEP(population_size=50, max_depth=3, max_generations=50, mutation_rate=0.1, crossover_rate=0.8)
best_solution = gep.evolve()

print("Best solution found:")
print(best_solution)
```

```
Generation 1/50
Best fitness so far: 1
Generation 2/50
Best fitness so far: 1
Generation 3/50
Best fitness so far: 4
Generation 4/50
Best fitness so far: 1
Generation 5/50
```