Nature Inspired Computing Codes

Simulated Annealing (SA) optimization algorithm

```
from numpy import asarray, exp, randn, rand, seed
from matplotlib import pyplot

# Define the objective function, which is minimized
def objective(step):
    return step[0] ** 2.0

# Define the simulated annealing algorithm
def sa(objective, area, iterations, step_size, temperature):
    # Create an initial point within the search space
    start_point = area[:, 0] + rand(len(area)) * (area[:, 1] - area[:, 0])
    # Evaluate the starting point
    start_point_eval = objective(start_point)

    # Initialize best known solution to start point and its evaluation
    mia_start_point, mia_start_eval = start_point, start_point_eval
    outputs = []

    # Main loop over the number of iterations
    for i in range(iterations):
        # Generate a candidate solution by adding a random step to the current point
        mia_step = mia_start_point + randn(len(area)) * step_size
        mia_step_eval = objective(mia_step)

        # Check if the candidate solution is better than the best solution so far
```

```
        if mia_step_eval < mia_start_eval:
            mia_start_point, mia_start_eval = mia_step, mia_step_eval
            outputs.append(mia_start_eval)

        # Calculate the difference between candidate and best solution
        difference = mia_step_eval - mia_start_eval
        # Update temperature based on the current iteration
        t = temperature / float(i + 1)

        # Calculate Metropolis Acceptance Criterion
        metropolis = exp(-difference / t)

        # Accept the new solution if it meets certain criteria (either
better or random acceptance)
        if difference < 0 or rand() < metropolis:
            mia_start_point, mia_start_eval = mia_step, mia_step_eval

    return mia_start_point, mia_start_eval, outputs

# Set seed for reproducibility
seed(1)
# Define the search space
area = asarray([[-6.0, 6.0]])
# Set algorithm parameters
temperature = 12
iterations = 1200
step_size = 0.1

# Run simulated annealing
best_point, best_eval, outputs = sa(objective, area, iterations,
step_size, temperature)

# Plotting the objective function evaluations over time
```

```
pyplot.plot(outputs, 'ro-')
pyplot.xlabel('Iteration')
pyplot.ylabel('Evaluation of Objective Function')
pyplot.show()
```

Simulated Annealing to solve the Traveling Salesman Problem (TSP)

```
import numpy as np
import random
import math

# Distance matrix (in km) for Colombo, Kandy, Nuwara Eliya, Galle, Jaffna
distances = np.array([
    [0, 115, 180, 120, 396],  # Colombo
    [115, 0, 75, 80, 380],    # Kandy
    [180, 75, 0, 140, 435],   # Nuwara Eliya
    [120, 80, 140, 0, 340],   # Galle
    [396, 380, 435, 340, 0]   # Jaffna
])

# Function to calculate the total distance of a given path
def total_distance(path):
    return sum(distances[path[i], path[i+1]] for i in range(len(path) - 1)) + distances[path[-1], path[0]]
```

```
# Simulated Annealing algorithm
def simulated_annealing():
    num_cities = len(distances)
    current_path = list(range(num_cities))
    random.shuffle(current_path)
    current_distance = total_distance(current_path)

    T = 10000  # Initial temperature
    T_min = 1  # Minimum temperature
    alpha = 0.995  # Cooling rate

    best_path = current_path[:]
    best_distance = current_distance

    while T > T_min:
        new_path = current_path[:]
        i, j = random.sample(range(num_cities), 2)
        new_path[i], new_path[j] = new_path[j], new_path[i]

        new_distance = total_distance(new_path)

        if new_distance < best_distance:
            best_path = new_path[:]
            best_distance = new_distance

        delta_distance = new_distance - current_distance
        if  delta_distance  <  0  or  random.random()  <  math.exp(-
delta_distance / T):
            current_path = new_path[:]
            current_distance = new_distance

        T *= alpha
```

```
    return best_path, best_distance

# Run the simulated annealing function to find the best path and distance
best_path, best_distance = simulated_annealing()

# Mapping city indices to names
city_names = ['Colombo', 'Kandy', 'Nuwara Eliya', 'Galle', 'Jaffna']
best_path_names = [city_names[i] for i in best_path]

# Print the results
print("Optimal path:", " -> ".join(best_path_names))
print("Total distance:", best_distance)
```

Particle Swarm Optimization (PSO) to determine the optimal placement of distribution centers based on minimizing the weighted distance to customer locations

```
import numpy as np
import matplotlib.pyplot as plt

# Define customer locations (latitude, longitude) and their demand
customers = np.array([
    (40.7128, -74.0060, 100),  # New York
    (34.0522, -118.2437, 80),  # Los Angeles
    (41.8781, -87.6298, 70),   # Chicago
```

```
    (29.7604, -95.3698, 60),   # Houston
    (33.7490, -84.3880, 50),   # Atlanta
    (39.9526, -75.1652, 40),   # Philadelphia
    (42.3601, -71.0589, 30),   # Boston
    (47.6062, -122.3321, 20),  # Seattle
    (32.7157, -117.1611, 10),  # San Diego
])

# Haversine formula to calculate distance between two lat/lon points
def haversine_distance(lat1, lon1, lat2, lon2):
    R = 6371  # Earth radius in kilometers
    lat1, lon1, lat2, lon2 = map(np.radians, [lat1, lon1, lat2, lon2])
    dlat = lat2 - lat1
    dlon = lon2 - lon1
    a = np.sin(dlat/2)**2 + np.cos(lat1) * np.cos(lat2) *
np.sin(dlon/2)**2
    c = 2 * np.arcsin(np.sqrt(a))
    return R * c

# Objective function: Total weighted distance from distribution
centers to customers
def objective_function(positions):
    total_cost = 0
    for customer in customers:
        distances = haversine_distance(positions[:, 0], positions[:,
1], customer[0], customer[1])
        nearest_center_distance = np.min(distances)
        total_cost += nearest_center_distance * customer[2]    #
distance * demand
    return total_cost

# Particle class for PSO
class Particle:
```

```
    def __init__(self, bounds, n_centers):
        self.position = np.random.uniform(low=bounds[:, 0],
high=bounds[:, 1], size=(n_centers, 2))
        self.velocity = np.zeros_like(self.position)
        self.best_position = np.copy(self.position)
        self.best_score = float('inf')

# PSO implementation
def particle_swarm_optimization(objective, bounds, n_centers,
num_particles, num_iterations):
    particles = [Particle(bounds, n_centers) for _ in
range(num_particles)]
    global_best_position = np.zeros((n_centers, 2))
    global_best_score = float('inf')

    w = 0.5  # inertia weight
    c1 = 1   # cognitive parameter
    c2 = 2   # social parameter

    for _ in range(num_iterations):
        for particle in particles:
            score = objective(particle.position)
            if score < particle.best_score:
                particle.best_score = score
                particle.best_position = particle.position
            if score < global_best_score:
                global_best_score = score
                global_best_position = particle.position

        for particle in particles:
            r1, r2 = np.random.rand(2)
            particle.velocity = (w * particle.velocity +
```

```
                                c1 * r1 * (particle.best_position -
particle.position) +
                                c2 * r2 * (global_best_position -
particle.position))
            particle.position    =    np.clip(particle.position    +
particle.velocity, bounds[:, 0], bounds[:, 1])

    return global_best_position, global_best_score

# Set up optimization problem
bounds = np.array([[25, 50], [-125, -65]])  # Latitude and Longitude
bounds for USA
n_centers = 3  # Number of distribution centers to optimize
num_particles = 50
num_iterations = 200

# Run PSO
best_positions,                     best_score                     =
particle_swarm_optimization(objective_function,  bounds,  n_centers,
num_particles, num_iterations)

# Print results
print(f"Optimal distribution center locations:")
for i, pos in enumerate(best_positions):
    print(f"Center    {i+1}:    Latitude    {pos[0]:.4f},    Longitude
{pos[1]:.4f}")
print(f"Total weighted distance cost: {best_score:.2f}")

# Visualize the result
plt.figure(figsize=(12, 8))
plt.scatter(customers[:,    1],    customers[:,    0],    c='blue',
s=customers[:, 2], alpha=0.6, label='Customers')
```

```
plt.scatter(best_positions[:, 1], best_positions[:, 0], c='red',
s=100, marker='*', label='Distribution Centers')
plt.title('Optimal Distribution Center Locations')
plt.xlabel('Longitude')
plt.ylabel('Latitude')
plt.legend()
plt.grid(True)
plt.show()
```

Ant Colony Optimization (ACO) for playlist recommendations

```
import numpy as np
import random

# Sample Songs and their attributes
songs = [
    {"id": 1, "name": "Blinding Lights", "artist": "The Weeknd",
"genre": "Pop", "mood": "Energetic"},
    {"id": 2, "name": "Watermelon Sugar", "artist": "Harry Styles",
"genre": "Pop", "mood": "Happy"},
    {"id": 3, "name": "Levitating", "artist": "Dua Lipa", "genre":
"Pop", "mood": "Energetic"},
    {"id": 4, "name": "Peaches", "artist": "Justin Bieber", "genre":
"R&B", "mood": "Chill"},
    {"id": 5, "name": "drivers license", "artist": "Olivia Rodrigo",
"genre": "Pop", "mood": "Sad"}
```

```
]

# Define parameters
n_ants = 10
n_iterations = 50
decay_rate = 0.1
pheromone_increase = 1.0

# Initialize pheromone levels between each song pair
pheromone_levels = np.ones((len(songs), len(songs)))

# Calculate song compatibility based on mood and genre
def song_compatibility(song1, song2):
    genre_compatibility = 1 if song1["genre"] == song2["genre"] else
0.5
    mood_compatibility = 1 if song1["mood"] == song2["mood"] else 0.5
    return genre_compatibility * mood_compatibility

# Initialize desirability matrix based on compatibility
desirability = np.zeros((len(songs), len(songs)))
for i in range(len(songs)):
    for j in range(len(songs)):
        if i != j:
            desirability[i][j]    =    song_compatibility(songs[i],
songs[j])

# Ant function to generate playlist
def generate_playlist(start_song):
    playlist = [start_song]
    for _ in range(len(songs) - 1):
        current_song = playlist[-1]
        probabilities  =  pheromone_levels[current_song["id"]  -  1]  *
desirability[current_song["id"] - 1]
```

```
        probabilities /= probabilities.sum()
        next_song = np.random.choice(songs, p=probabilities)
        playlist.append(next_song)
    return playlist

# Update pheromones based on successful playlists
def update_pheromones(playlists):
    global pheromone_levels
    pheromone_levels *= (1 - decay_rate)  # Apply decay
    for playlist in playlists:
        for i in range(len(playlist) - 1):
            from_idx = playlist[i]["id"] - 1
            to_idx = playlist[i + 1]["id"] - 1
            pheromone_levels[from_idx][to_idx] += pheromone_increase

# Main ACO loop
for _ in range(n_iterations):
    ant_playlists = [generate_playlist(random.choice(songs)) for _ in
range(n_ants)]
    update_pheromones(ant_playlists)

# Show final playlist and pheromone levels
print("Final Pheromone Levels:")
print(pheromone_levels)
for i, song in enumerate(songs):
    print(f"Song    {song['name']}    transition    preferences:",
pheromone_levels[i])
```

Genetic Algorithms (GA) for Feature Selection

```
import numpy as np
import pandas as pd
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score

# Load Breast Cancer Dataset
data = load_breast_cancer()
X = data.data
y = data.target

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

# Initialize Population - Each chromosome represents a binary feature selection
def initialize_population(size, n_features):
    return np.random.randint(0, 2, (size, n_features))

# Fitness Function - Evaluates the fitness of each chromosome
def fitness_function(population, X_train, y_train, X_test, y_test):
    fitness_scores = []
    for chromosome in population:
        selected_features = X_train[:, chromosome == 1]
        model = RandomForestClassifier()
        model.fit(selected_features, y_train)
        y_pred = model.predict(X_test[:, chromosome == 1])
```

```
        fitness_scores.append(accuracy_score(y_test, y_pred))
    return np.array(fitness_scores)

# Selection Function - Selects top-performing chromosomes
def selection(population, fitness_scores, num_parents):
    parents_indices = np.argsort(fitness_scores)[-num_parents:]
    return population[parents_indices]

# Crossover Function - Combines two parents to produce offspring
def crossover(parents, offspring_size):
    offspring = np.empty(offspring_size)
    crossover_point = np.uint8(offspring_size[1] / 2)
    for k in range(offspring_size[0]):
        parent1_idx = k % parents.shape[0]
        parent2_idx = (k + 1) % parents.shape[0]
        offspring[k,   :crossover_point]   =   parents[parent1_idx,
:crossover_point]
        offspring[k,   crossover_point:]   =   parents[parent2_idx,
crossover_point:]
    return offspring

# Mutation Function - Mutates a gene in each offspring
def mutation(offspring, mutation_rate):
    for idx in range(offspring.shape[0]):
        if np.random.rand() < mutation_rate:
            gene_idx = np.random.randint(0, offspring.shape[1])
            offspring[idx, gene_idx] = 1 - offspring[idx, gene_idx]
    return offspring

# Genetic Algorithm for Feature Selection
X_train,   X_test,   y_train,   y_test   =   train_test_split(X,   y,
test_size=0.2, random_state=42)
n_features = X.shape[1]
```

```
population = initialize_population(population_size, n_features)

for gen in range(generations):
    fitness_scores = fitness_function(population, X_train, y_train, X_test, y_test)
    parents = selection(population, fitness_scores, num_parents=8)
    offspring = crossover(parents, offspring_size=(population_size - parents.shape[0], n_features))
    offspring = mutation(offspring, mutation_rate)
    population[0:parents.shape[0], :] = parents
    population[parents.shape[0]:, :] = offspring

# Get the best chromosome and evaluate its accuracy
best_chromosome = population[np.argmax(fitness_function(population, X_train, y_train, X_test, y_test))]
selected_features = X[:, best_chromosome == 1]
model = RandomForestClassifier()
model.fit(selected_features, y)
y_pred = model.predict(selected_features)
print("Final Model Accuracy with Selected Features:", accuracy_score(y, y_pred))
print("Selected Features:", data.feature_names[best_chromosome == 1])
```

```
import random

# Sample real-world songs with details
songs = {
    1: {'name': 'Blinding Lights', 'artist': 'The Weeknd', 'genre': 'Pop', 'popularity': 95},
    2: {'name': 'Watermelon Sugar', 'artist': 'Harry Styles', 'genre': 'Pop', 'popularity': 88},
    3: {'name': 'Levitating', 'artist': 'Dua Lipa', 'genre': 'Pop', 'popularity': 90},
    4: {'name': 'Peaches', 'artist': 'Justin Bieber', 'genre': 'R&B', 'popularity': 85},
    5: {'name': 'drivers license', 'artist': 'Olivia Rodrigo', 'genre': 'Pop', 'popularity': 92},
    6: {'name': 'Save Your Tears', 'artist': 'The Weeknd', 'genre': 'Pop', 'popularity': 89},
    7: {'name': 'MONTERO (Call Me By Your Name)', 'artist': 'Lil Nas X', 'genre': 'Hip-Hop', 'popularity': 87},
    8: {'name': 'Good 4 U', 'artist': 'Olivia Rodrigo', 'genre': 'Pop', 'popularity': 91},
    9: {'name': 'Kiss Me More', 'artist': 'Doja Cat', 'genre': 'R&B', 'popularity': 86},
    10: {'name': 'Stay', 'artist': 'The Kid LAROI & Justin Bieber', 'genre': 'Pop', 'popularity': 94},
}

# Initialize pheromone levels between songs
pheromones = {(i, j): 1.0 for i in songs for j in songs if i != j}

# Heuristic function based on popularity
def heuristic(song1, song2):
    # Preference given to songs with higher popularity
```

```
    return (songs[song2]['popularity'] + 1) / 100  # Heuristic scaled
for popularity

# Ant class to build playlists
class Ant:
    def __init__(self):
        self.playlist = []

    def   select_next_song(self,   current_song,   unvisited_songs,
pheromones):
        #  Calculate  probabilities  based  on  pheromone  levels  and
heuristic
        probabilities = []
        for song in unvisited_songs:
            pheromone_level = pheromones[(current_song, song)]
            heuristic_value = heuristic(current_song, song)
            probabilities.append(pheromone_level * heuristic_value)

        # Normalize probabilities
        total = sum(probabilities)
        probabilities = [p / total for p in probabilities]

        # Choose next song based on probabilities
        next_song = random.choices(unvisited_songs, probabilities)[0]
        return next_song

    def build_playlist(self, start_song, pheromones):
        self.playlist = [start_song]
        unvisited_songs = set(songs.keys()) - {start_song}

        while unvisited_songs:
            current_song = self.playlist[-1]
```

```
            next_song      =      self.select_next_song(current_song,
unvisited_songs, pheromones)
            self.playlist.append(next_song)
            unvisited_songs.remove(next_song)

# Update pheromones based on ant's playlist
def update_pheromones(pheromones, ants, decay=0.1, contribution=1.0):
    # Evaporate pheromones
    for key in pheromones:
        pheromones[key] *= (1 - decay)

    # Deposit new pheromones based on ant playlists
    for ant in ants:
        for i in range(len(ant.playlist) - 1):
            pheromones[(ant.playlist[i],  ant.playlist[i  +  1])]  +=
contribution / len(ant.playlist)

# Simulate ACO for generating playlists
def run_aco(num_ants=5, iterations=10):
    best_playlist = None
    best_score = float('inf')

    for _ in range(iterations):
        ants = [Ant() for _ in range(num_ants)]

        for ant in ants:
            start_song = random.choice(list(songs.keys()))
            ant.build_playlist(start_song, pheromones)

        # Evaluate playlists (for simplicity, sum of popularity)
        for ant in ants:
            score   =   sum(songs[song]['popularity']   for   song   in
ant.playlist)
```

```
            if score < best_score:
                best_score = score
                best_playlist = ant.playlist

        update_pheromones(pheromones, ants)

    return best_playlist, best_score

# Run the ACO simulation
best_playlist, best_score = run_aco()
print("Best Playlist:", [songs[s]['name'] for s in best_playlist])
print("Best Score:", best_score)
```