VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB RECORD

Bio Inspired Systems (23CS5BSBIS)

Submitted by

Aditya Singh (1BM22CS022)

in partial fulfillment for the award of the degree of

BACHELOR OF ENGINEERING
in
COMPUTER SCIENCE AND ENGINEERING

B.M.S. COLLEGE OF ENGINEERING
(Autonomous Institution under VTU)
BENGALURU-560019
Sep-2024 to Jan-2025

B.M.S. College of Engineering,

Bull Temple Road, Bangalore 560019

(Affiliated To Visvesvaraya Technological University, Belgaum)

Department of Computer Science and Engineering

<u>CERTIFICATE</u>

This is to certify that the Lab work entitled " Bio Inspired Systems (23CS5BSBIS)" carried out by **Aditya Singh (1BM22CS022),** who is bonafide student of **B.M.S. College of Engineering.** It is in partial fulfillment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the Visvesvaraya Technological University, Belgaum. The Lab report has been approved as it satisfies the academic requirements of the above mentioned subject and the work prescribed for the said degree.

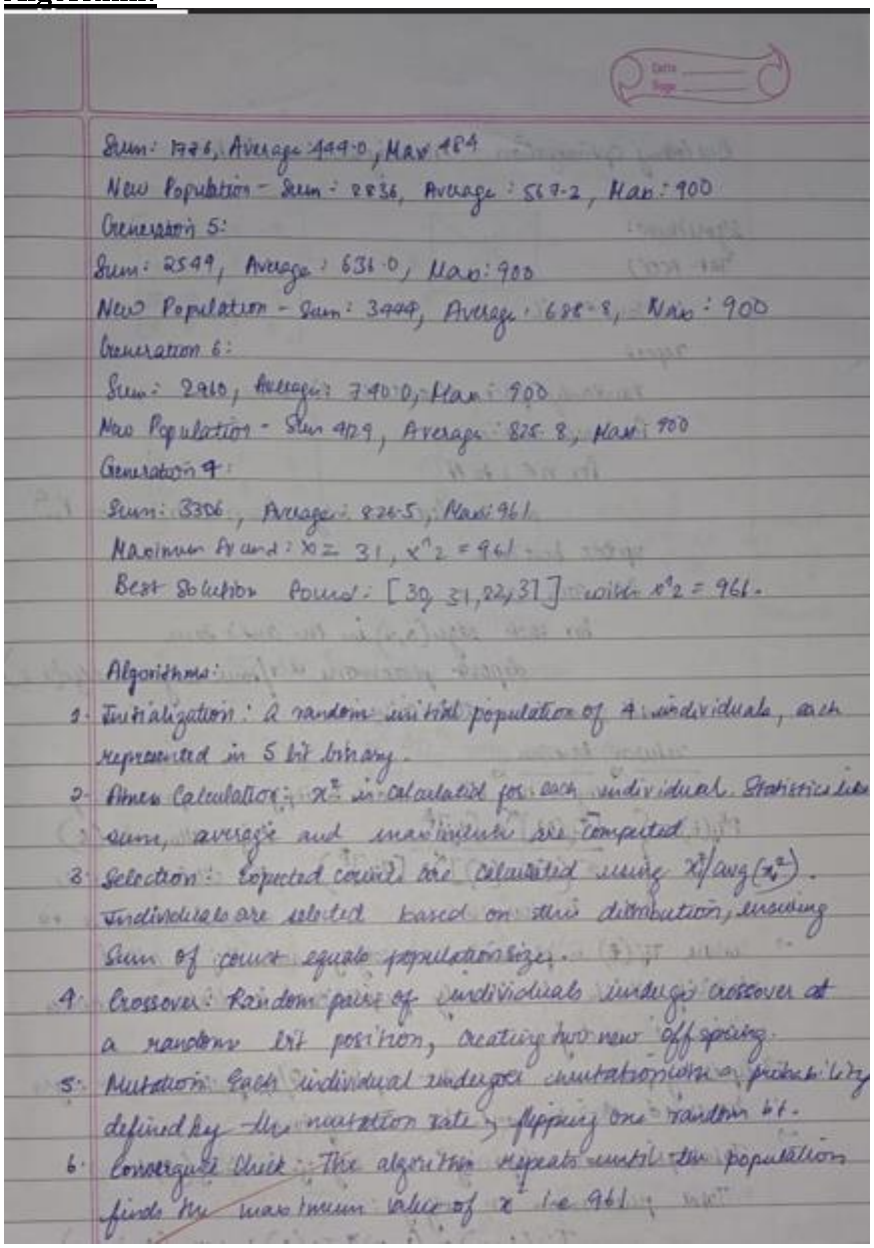
Lab faculty In charge: **DR. Shashikala** Assistant Professor Department of CSE, BMSCE	**Dr. Kavitha Sooda** Professor & HOD Department of CSE, BMSCE

Index

Github Link:
https://github.com/Singh12Aditya/1BM22CS022_BIS_LAB

Program 1
Genetic Algorithm

Algorithm:



Sum: 1776, Average 444.0, Max 484
New Population - Sum: 2836, Average: 567.2, Max: 900
Generation 5:
Sum: 2544, Average: 636.0, Max: 900
New Population - Sum: 3444, Average: 688.8, Max: 900
Generation 6:
Sum: 2960, Average: 740.0, Max: 900
New Population - Sum 4129, Average 825.8, Max: 900
Generation 7:
Sum: 3306, Average: 826.5, Max: 961
Maximum Found: x = 31, x^2 = 961
Best Solution Found: [30, 31, 22, 31] with x^2 = 961.

Algorithms:
1. Initialization: A random initial population of 4 individuals, each represented in 5 bit binary.
2. Fitness Calculation: x^2 is calculated for each individual. Statistics like sum, average and maximum are computed.
3. Selection: Expected counts are calculated using $x_i^2/\text{avg}(x_i^2)$. Individuals are selected based on this distribution, ensuring sum of count equals population size.
4. Crossover: Random pairs of individuals undergo crossover at a random bit position, creating two new offspring.
5. Mutation: Each individual undergoes mutation with a probability defined by the mutation rate, flipping one random bit.
6. Convergence Check: The algorithm repeats until the population finds the maximum value of x^2 i.e 961.

Code:

```
import random

# Step 1: Define fitness function (x^2)
def fitness(x):
    return x ** 2

# Step 2: Initialize population
def create_population(size, lower_bound, upper_bound):
    return [random.randint(lower_bound, upper_bound) for _ in range(size)]
```

```
# Step 3: Convert integer to 5-bit binary
def to_binary(x):
    return format(x, '05b')

# Step 4: Convert 5-bit binary back to integer
def from_binary(binary_str):
    return int(binary_str, 2)

# Step 5: Calculate the sum, average, and maximum of x^2 for the population
def calculate_statistics(population):
    fitness_values = [fitness(x) for x in population]
    total_sum = sum(fitness_values)
    average = total_sum / len(population)
    maximum = max(fitness_values)
    return fitness_values, total_sum, average, maximum

# Step 6: Selection using roulette wheel (fitness proportional)
def selection(population, fitness_values):
    total_fitness = sum(fitness_values)
    selection_probs = [fit / total_fitness for fit in fitness_values]

    # Roulette wheel selection
    selected_population = random.choices(population, weights=selection_probs, k=len(population))

    return selected_population

# Step 7: Perform crossover on pairs of individuals
def crossover(population):
    random.shuffle(population)
    new_population = []
    for i in range(0, len(population) - 1, 2):
        parent1, parent2 = population[i], population[i + 1]
        crossover_point = random.randint(1, 4)  # 5-bit crossover
        parent1_bin, parent2_bin = to_binary(parent1), to_binary(parent2)
        offspring1_bin = parent1_bin[:crossover_point] + parent2_bin[crossover_point:]
        offspring2_bin = parent2_bin[:crossover_point] + parent1_bin[crossover_point:]
        new_population.append(from_binary(offspring1_bin))
        new_population.append(from_binary(offspring2_bin))
    return new_population

# Step 8: Perform mutation on population (flip one bit)
def mutate(population, mutation_rate=0.05):
    mutated_population = []
    for individual in population:
        if random.random() < mutation_rate:
            individual_bin = list(to_binary(individual))
            mutation_point = random.randint(0, 4)
            individual_bin[mutation_point] = '1' if individual_bin[mutation_point] == '0' else '0'
```

```
            mutated_population.append(from_binary(''.join(individual_bin)))
        else:
            mutated_population.append(individual)
    return mutated_population

# Step 9: Genetic algorithm with elitism and biased towards 31
def genetic_algorithm(population_size, generations, lower_bound, upper_bound, mutation_rate):
    population = create_population(population_size, lower_bound, upper_bound)

    # Keep the best individual in each generation (elitism)
    best_solution = None

    for generation in range(generations):
        print(f"Generation {generation + 1}")

        # Step 4: Calculate fitness and statistics
        fitness_values, total_sum, average_fitness, max_fitness = calculate_statistics(population)
        print(f"Sum: {total_sum}, Average: {average_fitness}, Max: {max_fitness}")

        # If we find the maximum fitness, stop
        if max_fitness == 961:
            print(f"Maximum found: x = 31, x^2 = 961")
            return population

        # Step 5: Selection using fitness proportional selection (roulette wheel)
        selected_population = selection(population, fitness_values)

        # Step 6: Elitism - keep the best individual
        best_individual_idx = fitness_values.index(max(fitness_values))
        best_solution = population[best_individual_idx]  # Keep track of the best solution
        new_population = [best_solution]

        # Step 7: Crossover to create new population
        new_population.extend(crossover(selected_population))

        # Step 8: Mutate the population
        new_population = mutate(new_population, mutation_rate)

        # Recalculate fitness
        fitness_values, total_sum, average_fitness, max_fitness = calculate_statistics(new_population)
        print(f"New Population - Sum: {total_sum}, Average: {average_fitness}, Max: {max_fitness}")

        # Step 9: Update population for the next generation
        population = new_population[:population_size]

    # Return the best solution found
    return population

# Parameters
```

```
population_size = 4
generations = 50  # Increased generations
lower_bound = 0
upper_bound = 31
mutation_rate = 0.1

# Run the genetic algorithm
best_solution = genetic_algorithm(population_size, generations, lower_bound, upper_bound,
mutation_rate)
print(f"Best solution found: {best_solution} with x^2 = {max([fitness(x) for x in best_solution])}")
```

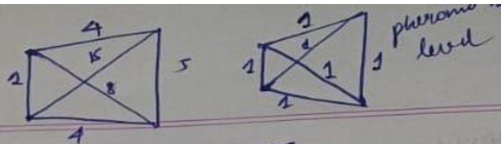
<u>Output:</u>

```
Generation 1
Sum: 494, Average: 123.5, Max: 484
New Population - Sum: 2305, Average: 461.0, Max: 529
Generation 2
Sum: 1776, Average: 444.0, Max: 484
New Population - Sum: 2260, Average: 452.0, Max: 484
Generation 3
Sum: 1776, Average: 444.0, Max: 484
New Population - Sum: 2836, Average: 567.2, Max: 900
Generation 4
Sum: 2352, Average: 588.0, Max: 900
New Population - Sum: 3028, Average: 605.6, Max: 900
Generation 5
Sum: 2544, Average: 636.0, Max: 900
New Population - Sum: 3444, Average: 688.8, Max: 900
Generation 6
Sum: 2960, Average: 740.0, Max: 900
New Population - Sum: 3860, Average: 772.0, Max: 900
Generation 7
Sum: 2960, Average: 740.0, Max: 900
New Population - Sum: 4129, Average: 825.8, Max: 900
Generation 8
Sum: 3229, Average: 807.25, Max: 900
New Population - Sum: 3790, Average: 758.0, Max: 961
Generation 9
Sum: 3306, Average: 826.5, Max: 961
Maximum found: x = 31, x^2 = 961
Best solution found: [30, 31, 22, 31] with x^2 = 961
```

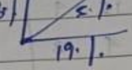
Program 2

Ant Colony Optimization

Algorithm:



$$P = \frac{1 \times 1/1}{(1 \times 1/1) + (1 \times 1/15) + (1 \times 1/4)} = 0.7595$$



$$P = \frac{1 \times 1/4}{(1 \times 1/1) + (1 \times 1/15) + (1 \times 1/4)}$$

Roulette wheel

Probalistic

0.76	0.19	0.05

Cumulative sum

1	0.24	0.05

A random number (r) in [0, 1]

$$\begin{cases} 0.24 < r \le 1.00 \\ 0.05 < r \le 0.24 \\ 0.00 \le r \le 0.05 \end{cases}$$

Algorithm

* Input:
  Problem specific parameters (graph, distance)
  Number of ants N
  pheromones evaporation rate (ρ)
  Initial pheromonic conc. (τ_0)
  Max. iterations

* Initializations
  Set pheromone conc. ~~to~~ τ to τ_0 for all paths
  define stopping criteria (max iteration or convergence)

* Generate ant population
  Create a list to store solutions of each ant
  For each ant in the colony: construct a solution by moving through the problem space based on pheromone and heuristic levels
  Store the constructed solution

* Calculate fitness value: For each solution generated by ants:
  Evaluate the fitness of the solution.
  (Calc. total distance for TSP) store fitness value.
* Use selection method to identify the best solution among ants. Keep track of best solution found so far.
* Update pheromone concentration
  Evaporation pheromones: $\tau = (1-\rho) * \tau$ (for all paths)
  Deposit new pheromone based on the quality of the solutions.
  For each ant, increase the pheromone levels on the paths used based on fitness of the solution. Return the best solution found

Code:

```
import numpy as np
import random

class AntColony:
    def __init__(self, distance_matrix, num_ants, num_iterations, alpha, beta, evaporation_rate):
        self.distance_matrix = distance_matrix
        self.num_ants = num_ants
        self.num_iterations = num_iterations
        self.alpha = alpha
        self.beta = beta
        self.evaporation_rate = evaporation_rate
        self.num_cities = distance_matrix.shape[0]
        self.pheromone_matrix = np.ones((self.num_cities, self.num_cities))

    def _calculate_probabilities(self, ant, visited):
        pheromone = self.pheromone_matrix[ant, :]
        visibility = 1 / self.distance_matrix[ant, :]
        probabilities = (pheromone ** self.alpha) * (visibility ** self.beta)
        # Convert visited set to array for indexing
        visited = np.array(list(visited))
        probabilities[visited] = 0
        return probabilities / np.sum(probabilities)

    def _update_pheromones(self, all_paths):
```

```
        self.pheromone_matrix *= (1 - self.evaporation_rate)
        for path, length in all_paths:
            for i in range(len(path) - 1):
                self.pheromone_matrix[path[i], path[i + 1]] += 1 / length

    def solve(self):
        best_path = None
        best_length = float('inf')

        for iteration in range(self.num_iterations):
            all_paths = []
            for ant in range(self.num_ants):
                path = [random.randint(0, self.num_cities - 1)]
                visited = set(path)

                for _ in range(self.num_cities - 1):
                    current_city = path[-1]
                    probabilities = self._calculate_probabilities(current_city, visited)
                    next_city = np.random.choice(range(self.num_cities), p=probabilities)
                    path.append(next_city)
                    visited.add(next_city)

                # Return to starting city
                path.append(path[0])
                length = self._calculate_path_length(path)
                all_paths.append((path, length))

                if length < best_length:
                    best_length = length
                    best_path = path

            self._update_pheromones(all_paths)
            # print(f"Iteration {iteration + 1}, Best Length: {best_length}")

        return best_path, best_length

    def _calculate_path_length(self, path):
        return sum(self.distance_matrix[path[i], path[i + 1]] for i in range(len(path) - 1))

if __name__ == "__main__":
    # Create a distance matrix for 5 cities
    distance_matrix = np.array([[0, 2, 9, 10, 3],
                                [1, 0, 6, 4, 5],
                                [15, 7, 0, 8, 9],
                                [6, 3, 12, 0, 4],
                                [10, 4, 8, 2, 0]])

    aco = AntColony(distance_matrix, num_ants=10, num_iterations=100, alpha=1.0, beta=2.0,
evaporation_rate=0.5)
```

```
best_path, best_length = aco.solve()
print("Best Path:", best_path)
print("Best Length:", best_length)
```

Output:

```
<ipython-input-16-57b8e868a3e4>:17: RuntimeWarning: divide by zero encountered in divide
  visibility = 1 / self.distance_matrix[ant, :]
Best Path: [3, 1, 0, 4, 2, 3]
Best Length: 23
```

Program 3
Particle Swarm Optimization

Algorithm:

Packet Swarm Optimization:

PSO algorithm

→ Initialize the controlling parameters (N, C_1, C_2, W_{min}, W_{max}, V_{max}, and MaxIter)
→ Initialize the population of N particles
→ do
  for each particle
    calculate the objective of the particle
    Update PBEST if required
    Update GBEST if required
  end for
  Update the inertia weight
  for each particle
    Update the velocity (V)
    Update the position (X)
  end for
while the end condition is not satisfied.
Return GBEST as the best estimation of global optimum.

Parameters:
Individual best position: C1 hyperparameter allows defining the ability of group to be influenced by best personal solutions found over the iterations. Also helps here exploration

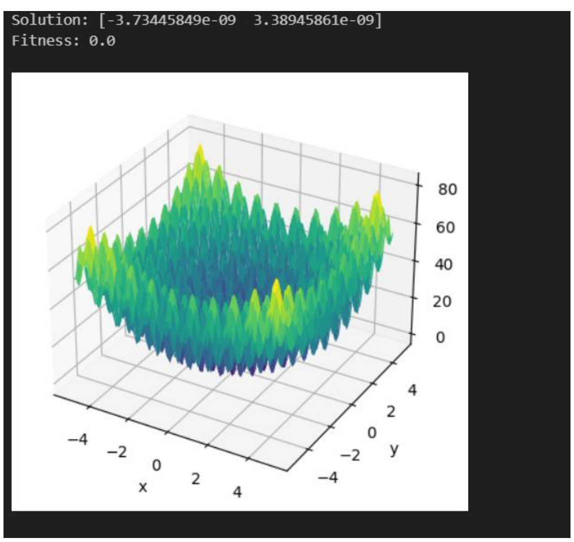
Swarm's best position: The hyperparameter C2 allows defining the ability of the ~~gro~~ group to influenced by best global solution found over the iterations. Helps in tuning exploitation

Code:

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

# Define the Rastrigin function
def rastrigin(x):
    n = len(x)
    return 10*n + sum([xi**2 - 10*np.cos(2*np.pi*xi) for xi in x])

# Define the PSO algorithm
def pso(cost_func, dim=2, num_particles=30, max_iter=100, w=0.5, c1=1, c2=2):
    # Initialize particles and velocities
    particles = np.random.uniform(-5.12, 5.12, (num_particles, dim))
    velocities = np.zeros((num_particles, dim))

    # Initialize the best positions and fitness values
    best_positions = np.copy(particles)
    best_fitness = np.array([cost_func(p) for p in particles])
    swarm_best_position = best_positions[np.argmin(best_fitness)]
    swarm_best_fitness = np.min(best_fitness)

    # Iterate through the specified number of iterations, updating the velocity and position of each
particle at each iteration
    for i in range(max_iter):
        # Update velocities
        r1 = np.random.uniform(0, 1, (num_particles, dim))
        r2 = np.random.uniform(0, 1, (num_particles, dim))
        velocities = w * velocities + c1 * r1 * (best_positions - particles) + c2 * r2 *
(swarm_best_position - particles)

        # Update positions
        particles += velocities

        # Evaluate fitness of each particle
        fitness_values = np.array([cost_func(p) for p in particles])

        # Update best positions and fitness values
        improved_indices = np.where(fitness_values < best_fitness)
        best_positions[improved_indices] = particles[improved_indices]
        best_fitness[improved_indices] = fitness_values[improved_indices]
        if np.min(fitness_values) < swarm_best_fitness:
            swarm_best_position = particles[np.argmin(fitness_values)]
            swarm_best_fitness = np.min(fitness_values)

    # Return the best solution found by the PSO algorithm
    return swarm_best_position, swarm_best_fitness

# Define the dimensions of the problem
```

```
dim = 2

# Run the PSO algorithm on the Rastrigin function
solution, fitness = pso(rastrigin, dim=dim)

# Print the solution and fitness value
print('Solution:', solution)
print('Fitness:', fitness)

# Create a meshgrid for visualization
x = np.linspace(-5.12, 5.12, 100)
y = np.linspace(-5.12, 5.12, 100)
X, Y = np.meshgrid(x, y)
Z = rastrigin([X, Y])

# Create a 3D plot of the Rastrigin function
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.plot_surface(X, Y, Z, cmap='viridis')
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('z')

# Plot the solution found by the PSO algorithm
ax.scatter(solution[0], solution[1], fitness, color='red')
plt.show()
```

<u>Output:</u>



Program 4
Cuckoo Search

Algorithm:

Cuckoo search algorithm

Purpose: The algo. was inspired by cuckoo birds. Cuckoo birds lay their eggs in the nests of other host birds. The first fundamental motivation for developing a new optimization algorithm is cuckoo laying egg and breeding. If the host bird recognizes the eggs are not being their own then it will either throw eggs away from its nest or simply empty the nest and build a new one. Each egg in a nest represents a solution. Cuckoo egg represents a new & good option. The answer obtained is a new option based on existing with some characteristics modified. It can be applied to a wide range of optimization problem because of its idealized breeding behaviour.

Applications of cuckoo search algorithm are:
image processing, pattern recognition, software testing, data mining, cyber security, cloud computing, IoT

Cuckoo Search Algorithm:

→ Initialization: Cuckoo birds prefer to lay their eggs in nest of other birds.

→ Levy flight: It is a random flight or walk. The steps are defined in terms of step lengths that have a certain probability distribution with random directions. The following movement is determined by current position.

→ Fitness Calculation: Calculation of fitness is achieved by using fitness function to find best solution. Nest is chosen randomly. The fitness of cuckoo egg (new solution) is then compared to that of host eggs (solution) in the nest. If the value of the cuckoo egg's fitness function is less than or equal to the value of the randomly chosen nest's fitness function, the randomly chosen nest is replaced by new solution

Termination: The fitness function compares the solutions in the current iteration and only the best solution is passed further. If the number of iteration is less than maximum, the best nest is retained. All cuckoo birds are ready for their next action after completing the initialization, levy flight and fitness calculation processes. The algorithm will be terminated once the max number of ~~iterating~~ iteration has been reached.

Output:
Optimal path: [0, 1, 3, 7, 9, 10]
Optimal path cost: 320.40997
CSA total Exec time => 35.984873

Code:

```
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.beta - 1) / 2))) ** (1 / self.beta)
        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")
```

Output:

```
...   Optimal path:  [0, 1, 3, 7, 9, 10]
      Optimal path cost:  320.40999999999997
      CSA total Exec time =>  13.131741
```

Program 5
Grey Wolf Optimizer

Algorithm:

Grey Wolf Optimizer
This algorithm is used for optimization problems. It is a population based meta heuristic algorithm that simulates the leadership hierarchy and hunting mechanism of grey wolves in nature.

Hierarchy:
α - dominant wolf of the pack, his/her orders should be followed
β - subordinate wolves, which help α in decision making
δ - submit to α & β, dominate ω.
ω - scapegoat & least important individuals in the pack.

Main phases of grey wolf hunting
→ Tracking, chasing & approaching the prey.
→ Pursuing, encircling & harassing the prey until it stops moving
→ attacks toward the prey.

Mathematical model & algorithm
→ Fittest solution as an Alpha wolf (α)
→ 2nd best solution as a Beta wolf (β)
→ 3rd best solution as a delta wolf (δ)
→ rest of candidate solution as Omega (ω) wolf

Application: scheduling, robotic & path planning, optimization and clustering in Machine learning

Algorithm.
→ Encircling the prey:

$$\vec{D} = |\vec{C} \cdot \vec{X_p}(t) - \vec{X}(t)|$$
$$\vec{X}_{(t+1)} = \vec{X_p}(t) - \vec{A} \cdot \vec{D}$$

$\vec{A}$, $\vec{C}$ are coefficient vectors
$\vec{X_p}$ is the position vector of the prey

$\vec{X}$ indicate position vector of grey wolf
vectors $\vec{A}$ & $\vec{C}$ are calculated as follow:

$$\vec{A} = 2\vec{a}.\vec{r_1} - \vec{a}$$

$\vec{C} = 2.\vec{r_2}$, where components of $\vec{a}$ are linearly decreased from 2 to 0 over the course of iterations. r_1 & r_2 are random vectors in [0,1]

Hunting.

$$\vec{D_\alpha} = |\vec{C_1}.\vec{X_\alpha} - \vec{X}|, \vec{D_\beta} = |\vec{C_2}.\vec{X_\beta} - \vec{X}|, \vec{D_\delta} = |\vec{C_3}.\vec{X_\delta} - \vec{X}|$$

$$\vec{X_1} = \vec{X_\alpha} - \vec{a_1}(\vec{D_\alpha}), \vec{X_2} = \vec{X_\beta} - \vec{a_2}(\vec{D_\beta}), \vec{X_3} = \vec{X_\delta} - \vec{a_3}.(\vec{D_\delta})$$

$$\vec{X}(t+1) = \frac{\vec{X_1} + \vec{X_2} + \vec{X_3}}{3}$$

attacking the prey:

$$\vec{D} = |\vec{C}.\vec{X_p}(t) - \vec{X}(t)|$$

$$\vec{X}(t+1) = \vec{X_p}(t) - \vec{A}.\vec{D}$$

$$\vec{A} = 2\vec{a}.\vec{r_1} - \vec{a}$$

$$\vec{C} = 2.\vec{r_2}$$

fluctuation range of $\vec{A}$ is also decreased by $\vec{a}$.
$\vec{A}$ is random value in interval [-2a, 2a]
when $-1 < A < 1$ → approaching prey

when $A > 1$ & $A < -1$ diverging from prey.

$\vec{C}$ vector contains random value from [0, 2]

Algorithm:

Initialize the grey wolf population X_i ($i = 1, 2 \ldots n$)
Initialize a, A and C
Calculate the fitness of each search agent.
X_α = the best search agent.
X_β - the second best search agent.
X_δ = the third best search agent.

Code:

```
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])
```

```python
        Iter += 1

    # 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")

print()
print()

# 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")
```

Output:

```
...
    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 = 0.000000

    End GWO for Sphere
```

Program 6
Parallel Cellular Algorithms

Algorithm:

Parallel Cellular Algorithms

Purpose:
Cellular automata model is a nature inspired parallel computational model that can be used for modelling and simulation of emergent phenomena and systems. Because of their inherent parallelism, cellular automata can be used to model large scale emergent systems on parallel computers.

Application
Parallel cellular automata models are successfully used in fluid dynamics, molecular dynamics, biology, genetic, chemistry, road traffic flow, cryptography, image processing, environmental modelling and finance.

Algorithm:
1. Initialization:
1. Initialize Grid:
→ Create a 2D grid where each cell represents a part of the forest:
→ Each cell has one of the following states:
 1 = No fuel
 2 = Burnable fuel
 3 = Burning
 4 = Burned
→ Setting initial conditions
Setting the center of grid (burning) to start the fire.
2. Simulation
1. Updating forest:
For each cell in grid:
→ If cell is already burned or not burnable, leaving it unchanged.

→ If cell is burning (3), randomly decide if it will burnout based on probability.

→ If cell is burnable, neighbours are checked
If any neighbour is burning, probability of cell catching fire is checked. If probability ~~is~~ exceeds a random threshold, set cell to burning

(2) Communicating with Neighbours (for parallel versions)

→ Exchanging boundary rows between neighbour process to keep grid consistent.
Repeating for next generation.

(3) Visualization after each generation

(4) Termination

```
Initialize_grid (grid)
Define TransitionRule (cell_state, neighbours)
For each iteration 1 to max_iteration
    For each cell (x,y) in grid:
        next_state (x,y) = TransitionRule (grid[x,y],
                               getNeighbours (x,y,grid))

    For each cell (x,y) in grid:
        grid (x,y) = next_state (x,y)

    If IsStable (grid)
        break.
Return grid.
```

Code:

```
#ParallelCellularAlgorithms
import numpy as np
from multiprocessing import Pool
```

```
def update_cell(cell_index, grid, size):
    # Unpack the cell index (cell_index should be a tuple (x, y))
    x, y = cell_index

    # Define neighbors' indices (using a 2D Moore neighborhood)
    neighbors = [
        ((x-1) % size, y), ((x+1) % size, y),
        (x, (y-1) % size), (x, (y+1) % size)
    ]

    # Compute new state (this could be any rule like majority, XOR, etc.)
    new_state = sum(grid[n[0], n[1]] for n in neighbors) % 2  # example: majority rule
    return (x, y, new_state)

def parallel_update(grid, size, num_iterations):
    pool = Pool(processes=4)  # Use 4 processes for parallel execution

    for iteration in range(num_iterations):
        print(f"Iteration {iteration + 1}:")

        indices = [(x, y) for x in range(size) for y in range(size)]
        result = pool.starmap(update_cell, [(i, grid, size) for i in indices])

        # Update the grid with the new states
        for x, y, new_state in result:
            grid[x, y] = new_state

        # Print the updated grid for this iteration
        print(grid)

    return grid

# Initialize grid with random states (0 or 1)
grid_size = 5
grid = np.random.randint(2, size=(grid_size, grid_size))

# Number of iterations to run
num_iterations = 10

# Parallel update and print iterations
updated_grid = parallel_update(grid, grid_size, num_iterations)
```

Output:

```
...   Iteration 1:
      [[1 1 1 0 0]
       [1 0 0 1 1]
       [0 1 1 0 1]
       [1 0 0 0 1]
       [0 0 1 0 0]]
      Iteration 2:
      [[0 0 0 0 0]
       [0 1 1 1 1]
       [0 1 1 1 0]
       [1 0 0 1 0]
       [0 0 1 1 1]]
      Iteration 3:
      [[0 1 0 0 0]
       [0 0 1 1 1]
       [0 0 1 1 0]
       [0 0 1 0 1]
       [0 1 1 1 1]]
      Iteration 4:
      [[1 1 1 0 0]
       [1 0 0 1 1]
       [0 1 1 0 1]
       [1 0 0 0 1]
       [0 0 1 0 0]]
      Iteration 5:
      ...
       [1 0 0 1 1]
       [0 1 1 0 1]
       [1 0 0 0 1]
       [0 0 1 0 0]]
```

Program 7
Gene Expression Algorithm

Algorithm:

Optimization via Gene Expression algorithms:

Gene optimization algorithm (GEP is an evolutionary algorithm that evolves computer programs or models to solve problem. It combines the strengths of Genetic algorithms & Genetic Programming to improve computational efficiency & accuracy.

Purpose: to automate the process of model or program discovery in a given problem domain.
→ Optimization and Modelling: Solves complex optimization problems in various scientific or engineering fields
→ Automatic Programming: Develops models & programs through evolutionary processes without manual intervention.
→ Function Discovery: Derives mathematical expression or decision rules for tasks like classification, regression & time series prediction.

Application: data mining & knowledge discovery, engineering optimization, biological and bioinformatics application, finance modelling, regression analysis, pattern recognition

Algorithm:
1. Initialization: Initialize a population of chromosomes randomly. Each chromosome encode a candidate solution to the optimization problem
2. Decoding: Convert the linear chromosome representation into an expression tree (phenotype)
3. Fitness calculation: Evaluate each candidate's solution's fitness using the defined fitness function (objective function to be optimized)
4. Selection: Select individuals with higher fitness scores as parents for producing the next generation.

5. Genetic operators:
Apply genetic operators to selected individual
→ Mutation: Randomly alter parts of a chromosome
→ Crossover: Combine parts of two parent chromosomes to produce offspring
→ Transposition: Move genes ~~into~~ segments within a chromosome
→ Recombination: Combine parts of two chromosomes to create a new solution.

6. Replacement: Form the new population by replacing less fit individual with offspring.

7. Termination: Repeat steps 2-6 for a fixed number of generation or until the desired optimization level is achieved

8. Return best solution

Pseudo-code

(1) input: - population size (P), chromosome length, maximum generation (G), mutation rate, crossover rate and other operator rate, fitness function (objective function to optimize)

(2.) Initialize Population: - Randomly generate P chromosomes.

(3) For generation = 1 to G:

(a) Decode each chromosome into its expression tree (phenotype)

(b) Evaluate fitness of each individual using fitness function

(c) Select fittest individuals as parents using a selection method.

(d) Apply genetic operators to produce new offspring

(e) Replace old population
- Combine offspring with parents or replace old population with new population generation

(f) check stopping criteria: If max. generations are reached or fitness threshold is satisfied, exit the loop.

(4) Output best fit individual.

Code:

```
#GeneExpresssionAlgorithm
import random

# Parameters
POPULATION_SIZE = 20
CHROMOSOME_LENGTH = 2  # Two variables: x and y
MAX_GENERATIONS = 500
MUTATION_RATE = 0.1
CROSSOVER_RATE = 0.7

# Objective Function (Fitness Function)
def objective_function(x, y):
    """Objective: Minimize f(x, y) = x^2 + y^2"""
    return x**2 + y**2

# Chromosome Class
class Chromosome:
    def __init__(self):
        # Randomly initialize x and y in the range [-5, 5]
        self.genes = [random.uniform(-5, 5) for _ in range(CHROMOSOME_LENGTH)]
        self.fitness = 0.0

    def evaluate_fitness(self):
        """Calculate fitness: Fitness = 1 / (1 + objective_function)"""
        x, y = self.genes
        self.fitness = 1 / (1 + objective_function(x, y))

# Initialize Population
def generate_population(size):
    return [Chromosome() for _ in range(size)]

# Tournament Selection
def select_parent(population):
    """Select the fittest parent between two randomly chosen individuals."""
    p1, p2 = random.sample(population, 2)
    return p1 if p1.fitness > p2.fitness else p2

# Crossover
def crossover(parent1, parent2):
    """Perform single-point crossover."""
    child = Chromosome()
    for i in range(CHROMOSOME_LENGTH):
        if random.random() < CROSSOVER_RATE:
            child.genes[i] = parent1.genes[i]
        else:
            child.genes[i] = parent2.genes[i]
    return child
```

```
# Mutation
def mutate(chromosome):
    """Randomly mutate genes with small changes."""
    for i in range(CHROMOSOME_LENGTH):
        if random.random() < MUTATION_RATE:
            chromosome.genes[i] += random.gauss(0, 1)  # Add small Gaussian noise

# Find Best Solution
def find_best_solution(population):
    """Return the chromosome with the highest fitness."""
    return max(population, key=lambda chromo: chromo.fitness)

# Main Genetic Algorithm
def genetic_algorithm():
    # Step 1: Initialize Population
    population = generate_population(POPULATION_SIZE)

    # Evaluate initial fitness
    for individual in population:
        individual.evaluate_fitness()

    # Evolution Loop
    for generation in range(MAX_GENERATIONS):
        new_population = []

        # Generate New Population
        while len(new_population) < POPULATION_SIZE:
            # Selection
            parent1 = select_parent(population)
            parent2 = select_parent(population)

            # Crossover
            child = crossover(parent1, parent2)

            # Mutation
            mutate(child)

            # Evaluate Child's Fitness
            child.evaluate_fitness()
            new_population.append(child)

        # Replace old population
        population = new_population

        # Find and Print Best Solution for Current Generation
        best_solution = find_best_solution(population)
        print(f"Generation {generation}: Best Fitness = {best_solution.fitness:.6f}, Genes =
{best_solution.genes}")
```

```python
    # Final Best Solution
    best_solution = find_best_solution(population)
    print("\nBest Solution Found:")
    print(f"Fitness: {best_solution.fitness:.6f}, Genes: {best_solution.genes}")

# Run the Genetic Algorithm
if __name__ == "__main__":
    genetic_algorithm()
```

Output:

```
...   Generation 0: Best Fitness = 0.891172, Genes = [-0.34827931128233836, -0.02861530914581678]
      Generation 1: Best Fitness = 0.891172, Genes = [-0.34827931128233836, -0.02861530914581678]
      Generation 2: Best Fitness = 0.891172, Genes = [-0.34827931128233836, -0.02861530914581678]
      Generation 3: Best Fitness = 0.767870, Genes = [-0.43720477443532946, -0.33339951226306375]
      Generation 4: Best Fitness = 0.947112, Genes = [0.23456974992404334, -0.02861530914581678]
      Generation 5: Best Fitness = 0.838950, Genes = [-0.43720477443532946, -0.02861530914581678]
      Generation 6: Best Fitness = 0.838950, Genes = [-0.43720477443532946, -0.02861530914581678]
      Generation 7: Best Fitness = 0.838950, Genes = [-0.43720477443532946, -0.02861530914581678]
      Generation 8: Best Fitness = 0.838950, Genes = [-0.43720477443532946, -0.02861530914581678]
      Generation 9: Best Fitness = 0.997927, Genes = [-0.03547770994166116, -0.02861530914581678]
      Generation 10: Best Fitness = 0.989647, Genes = [-0.03547770994166116, -0.09593112533595272]
      Generation 11: Best Fitness = 0.997927, Genes = [-0.03547770994166116, -0.02861530914581678]
      Generation 12: Best Fitness = 0.997927, Genes = [-0.03547770994166116, -0.02861530914581678]
      Generation 13: Best Fitness = 0.997927, Genes = [-0.03547770994166116, -0.02861530914581678]
      Generation 14: Best Fitness = 0.997927, Genes = [-0.03547770994166116, -0.02861530914581678]
      Generation 15: Best Fitness = 0.997927, Genes = [-0.03547770994166116, -0.02861530914581678]
      Generation 16: Best Fitness = 0.998217, Genes = [-0.03547770994166116, -0.02295837585093742]
      Generation 17: Best Fitness = 0.998217, Genes = [-0.03547770994166116, -0.02295837585093742]
      Generation 18: Best Fitness = 0.998217, Genes = [-0.03547770994166116, -0.02295837585093742]
      Generation 19: Best Fitness = 0.998467, Genes = [0.026759618678106264, -0.02861530914581678]
      Generation 20: Best Fitness = 0.998467, Genes = [0.026759618678106264, -0.02861530914581678]
      Generation 21: Best Fitness = 0.998467, Genes = [0.026759618678106264, -0.02861530914581678]
      Generation 22: Best Fitness = 0.998467, Genes = [0.026759618678106264, -0.02861530914581678]
      Generation 23: Best Fitness = 0.998467, Genes = [0.026759618678106264, -0.02861530914581678]
      Generation 24: Best Fitness = 0.998467, Genes = [0.026759618678106264, -0.02861530914581678]
      ...
      Generation 499: Best Fitness = 0.999765, Genes = [-0.005477642735498501, -0.01432034047625938]

      Best Solution Found:
      Fitness: 0.999765, Genes: [-0.005477642735498501, -0.01432034047625938]
```