VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB RECORD

Bio Inspired Systems (23CS5BSBIS)

Submitted by

Akanksha Singa (1BM22CS027)

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

CERTIFICATE

This is to certify that the Lab work entitled " Bio Inspired Systems (23CS5BSBIS)" carried out by **Akanksha Singa(1BM22CS027),** 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.

Shashikala Assistant Professor Department of CSE, BMSCE	Dr. Joythi S Nayak Professor & HOD Department of CSE, BMSCE

Index

Sl. No.	Date	Experiment Title	Page No.
1		**Genetic Algorithm for Optimization Problems**	1-5
2		**Particle Swarm Optimization**	6-9
3		**Ant Colony Optimization**	10-14
4		**Cuckoo Search Optimization**	15-17
5		**Grey Wolf Optimizer**	18-20
6		**Prallel Cellular Algorithm**	21-24
7		**Gene Expression**	25-30

Github Link:
https://github.com/Akanksha-singa/Bis-_Lab_5A_B2

Program 1

Genetic Algorithm for Optimization Problems:

Problem Statement:
Genetic algorithms are a type of optimization algorithm, meaning they are used to find the optimal solution(s) to a given computational problem that maximizes or minimizes a particular function. Genetic algorithms represent one branch of the field of study called evolutionary computation, in that they imitate the biological processes of reproduction and natural selection to solve for the 'fittest' solutions.

Algorithm:

Genetic Algorithm

It is a optimization technique based on natural selection and evolution. It is done with the help of crossover we generally use MATLAB alongside example such as Travelling Salesman Problem. It employs history, current uses & future potential of genetic algorithm in various fields

1) Components, Structure & Terminology of genetic algorithms

Key components of genetic algorithm

1) Fitness Function - measure how well a chromosome solves problem

2) Population of Chromosomes - a set of candidates each represented by an array of parameter values

chromosome = $[P_1, P_2, \ldots P_{Npar}]$

3) Selection - chooses which chromosome reproduces based on fitness score calculated as

$$P(C_{53}) = \left| \frac{f(C_{53})}{\sum_{i=1}^{N_{pop}} f(C_i)} \right|.$$

4) Crossover - Combines two parent chromosomes to produce offspring

Ex: parent chromosomes are

11010111001000 & 01011010101010

are crossed over after 4th bit

01010111001000 & 11011101010010

5) Mutation - Introduces random change to chromosome to maintain diversity
From intial populatⁿ chromosomes are selected based on fitness function followed by crossover & mutatⁿ to create a new generatⁿ and the cycle continues until a satisfactory solⁿ is found

2) Preliminary Example
Given 3 examples demonstrate GA of incresing complexity

2.1) Maximizing a funcⁿ of one variable
Maximize the funcⁿ $f(x) = \frac{-x^2}{10} + 3x$ for x in range $[0, 31]$
Assume x as a chromosome and is repersented as 5 bits 0 to 31 i.e 00000 to 11111, generate random populatⁿ evaluate thier fitness and use probablity function
Pair of chromosome crossover to produce offspring
After one generatⁿ both max & avg fitness values of populatⁿ improved illustrating GA's abiliy to evolve towards fitter solⁿ

2.2) Number of 1's in a string
Maximiz

9/10 जे

Code:

```
import random

# Define the problem: maximize f(x) = x^2
def fitness_function(x):
    return x ** 2

# Initialize parameters
POPULATION_SIZE = 100
MUTATION_RATE = 0.01
CROSSOVER_RATE = 0.9
NUM_GENERATIONS = 50
X_RANGE = (-10, 10)  # The range for the x values

# Create an individual (a solution)
def create_individual():
    return random.uniform(X_RANGE[0], X_RANGE[1])

# Create the initial population
def create_population():
    return [create_individual() for _ in range(POPULATION_SIZE)]

# Evaluate the fitness of each individual
def evaluate_population(population):
    return [fitness_function(ind) for ind in population]

# Selection using roulette wheel method
def roulette_wheel_selection(population, fitness_values):
    total_fitness = sum(fitness_values)
    pick = random.uniform(0, total_fitness)
    current = 0
    for individual, fitness in zip(population, fitness_values):
        current += fitness
        if current > pick:
            return individual

# Crossover (linear crossover)
def crossover(parent1, parent2):
    if random.random() < CROSSOVER_RATE:
        alpha = random.random()  # Weighted combination
        offspring1 = alpha * parent1 + (1 - alpha) * parent2
        offspring2 = alpha * parent2 + (1 - alpha) * parent1
```

```
        return offspring1, offspring2
    else:
        return parent1, parent2

# Mutation
def mutate(individual):
    if random.random() < MUTATION_RATE:
        return random.uniform(X_RANGE[0], X_RANGE[1])
    return individual

# Genetic Algorithm function
def genetic_algorithm():
    # Step 1: Initialize the population
    population = create_population()

    for generation in range(NUM_GENERATIONS):
        # Step 2: Evaluate the fitness
        fitness_values = evaluate_population(population)

        # Track the best solution in this generation
        best_individual = population[fitness_values.index(max(fitness_values))]
        best_fitness = max(fitness_values)

        print(f"Generation {generation + 1}: Best Fitness = {best_fitness}, Best Individual =
{best_individual}")

        # Step 3: Create a new population
        new_population = []

        while len(new_population) < POPULATION_SIZE:
            # Step 4: Selection
            parent1 = roulette_wheel_selection(population, fitness_values)
            parent2 = roulette_wheel_selection(population, fitness_values)

            # Step 5: Crossover
            offspring1, offspring2 = crossover(parent1, parent2)

            # Step 6: Mutation
            offspring1 = mutate(offspring1)
            offspring2 = mutate(offspring2)

            new_population.extend([offspring1, offspring2])

        # Step 7: Replacement with the new population
        population = new_population[:POPULATION_SIZE]  # Ensure population size is maintained

    # After all generations, return the best solution found
```

```python
    fitness_values = evaluate_population(population)
    best_individual = population[fitness_values.index(max(fitness_values))]
    best_fitness = max(fitness_values)
    print(f"\nBest Solution: x = {best_individual}, f(x) = {best_fitness}")



# Run the genetic algorithm
if __name__ == "__main__":
    genetic_algorithm()
```

Output:

```
Generation 1: Best Fitness = 99.99346360752477, Best Individual = 9.999673175035511
Generation 2: Best Fitness = 99.99346360752477, Best Individual = 9.999673175035511
Generation 3: Best Fitness = 99.46662005154673, Best Individual = 9.973295345649136
Generation 4: Best Fitness = 97.69032186810577, Best Individual = 9.883841453003269
Generation 5: Best Fitness = 94.18389044049461, Best Individual = 9.704838506667414
Generation 6: Best Fitness = 93.3963817034628, Best Individual = 9.664180343074253
Generation 7: Best Fitness = 93.3963817034628, Best Individual = 9.664180343074253
Generation 8: Best Fitness = 89.52907041313442, Best Individual = 9.461980258547067
Generation 9: Best Fitness = 86.36485302207772, Best Individual = 9.293269232195833
Generation 10: Best Fitness = 86.36485302207772, Best Individual = 9.293269232195833
Generation 11: Best Fitness = 84.0653643753673, Best Individual = 9.168716615501173
Generation 12: Best Fitness = 81.47559520746516, Best Individual = 9.026383284985474
Generation 13: Best Fitness = 61.482462240096964, Best Individual = -7.841075324220331
Generation 14: Best Fitness = 60.19481608858275, Best Individual = -7.758531825582901
Generation 15: Best Fitness = 60.19481608858275, Best Individual = -7.758531825582901
Generation 16: Best Fitness = 58.59258803609685, Best Individual = -7.654579546656815
Generation 17: Best Fitness = 57.38968292186089, Best Individual = -7.575597859037984
Generation 18: Best Fitness = 55.86902614623868, Best Individual = -7.474558592066737
Generation 19: Best Fitness = 65.73651789932204, Best Individual = 8.107805985550101
Generation 20: Best Fitness = 65.73651789932204, Best Individual = 8.107805985550101
Generation 21: Best Fitness = 54.5966996255954, Best Individual = -7.388957952620613
Generation 22: Best Fitness = 98.44634815673862, Best Individual = 9.922013311659011
Generation 23: Best Fitness = 91.35472232908678, Best Individual = 9.55796643272442
```

```
Generation 40: Best Fitness = 52.247096331230296, Best Individual = -7.2282152936413215
Generation 41: Best Fitness = 52.240649550914924, Best Individual = -7.2277693343738445
Generation 42: Best Fitness = 52.17719810327338, Best Individual = -7.223378579534191
Generation 43: Best Fitness = 52.169401490827546, Best Individual = -7.222838880303751
Generation 44: Best Fitness = 52.12957476117624, Best Individual = -7.2200813541937485
Generation 45: Best Fitness = 52.12957476117624, Best Individual = -7.2200813541937485
Generation 46: Best Fitness = 92.37299201258797, Best Individual = -9.61108693190255
Generation 47: Best Fitness = 90.97563486883489, Best Individual = -9.538114848796637
Generation 48: Best Fitness = 92.21354426248418, Best Individual = 9.60278835872603
Generation 49: Best Fitness = 71.676499026507, Best Individual = -8.466197436069336
Generation 50: Best Fitness = 71.676499026507, Best Individual = -8.466197436069336

Best Solution: x = -8.353243892876788, f(x) = 69.77668353388336
```

Program 2

Particle Swarm Optimization

Problem Statement:
The objective is to optimize a given function using Particle Swarm Optimization (PSO). PSO is a population-based metaheuristic algorithm inspired by the social behavior of particles in a swarm. The algorithm aims to find the optimal solution by iteratively adjusting particle positions and velocities within a defined search space to minimize or maximize the objective function. The search considers constraints, if any, to ensure feasibility

Algorithm:

• Particle Swarm Optimization (PSO) for Function Optimization

PSO is inspired by social behaviour in nature where a group of candidate solutions collaboratively searches for the best solⁿ in the solution space.

• Uses:

• Optimization: Finding optimal solⁿ
• Functⁿ optimizatⁿ: Minimizing or maximizing funcⁿ
• Machine learning: Tuning model parameters

• Application Feilds:

1) Engineering: Design and control Optimizatⁿ
2) Finance: Portfolio and risk optimizatⁿ
3) Robotics: path planning & sensor optimizatⁿ
4) Telecommunications: network resource allocatⁿ
5) Data Mining: Feature selectⁿ & Clustering

• Optimization Techniques

1) Velocity update: Adjusting movement based on personal and global bests
2) Parameter Tuning: Optimization interia and coeffecients for better convergence
3) Hybrid Approches: Combining with other methods for enhanced performance
4) Adaptive PSO: Dynamic adjustment of parameters during iterations.

Code:

```
import random
# Objective function: f(x) = x^2
def fitness_function(x):
    return x**2

# Particle class to represent each particle in the swarm
class Particle:
    def __init__(self, min_x, max_x):
        self.position = random.uniform(min_x, max_x)  # Current position
        self.velocity = random.uniform(-1, 1)          # Current velocity
        self.best_position = self.position             # Best position found by the particle
        self.best_fitness = fitness_function(self.position)  # Best fitness value

    def update_velocity(self, global_best_position, inertia_weight, cognitive_coefficient, social_coefficient):
        r1, r2 = random.random(), random.random()
        cognitive_velocity = cognitive_coefficient * r1 * (self.best_position - self.position)
        social_velocity = social_coefficient * r2 * (global_best_position - self.position)
        self.velocity = (inertia_weight * self.velocity) + cognitive_velocity + social_velocity

    def update_position(self, min_x, max_x):
        self.position += self.velocity
        # Ensure the position is within bounds
        self.position = max(min_x, min(self.position, max_x))
        # Update the best position and fitness if needed
        fitness = fitness_function(self.position)
        if fitness < self.best_fitness:  # We want to minimize
            self.best_fitness = fitness
            self.best_position = self.position

# PSO algorithm
def particle_swarm_optimization(pop_size, min_x, max_x, generations, inertia_weight, cognitive_coefficient, social_coefficient):
    # Initialize particles
    swarm = [Particle(min_x, max_x) for _ in range(pop_size)]

    # Global best position initialized to None
    global_best_position = swarm[0].best_position
    global_best_fitness = swarm[0].best_fitness

    for generation in range(generations):
        for particle in swarm:
            # Update global best position
            if particle.best_fitness < global_best_fitness:
```

```
                global_best_fitness = particle.best_fitness
                global_best_position = particle.best_position

            # Update particle velocity and position
            particle.update_velocity(global_best_position, inertia_weight, cognitive_coefficient, social_coefficient)
            particle.update_position(min_x, max_x)

        # Print the best fitness in the current generation
        print(f"Generation {generation + 1}: Best solution = {global_best_position}, Fitness = {global_best_fitness}")

    return global_best_position

# Parameters
population_size = 30
min_value = -10
max_value = 10
num_generations = 50
inertia_weight = 0.5
cognitive_coefficient = 1.5
social_coefficient = 1.5

# Run Particle Swarm Optimization
best_solution = particle_swarm_optimization(population_size, min_value, max_value, num_generations, inertia_weight, cognitive_coefficient, social_coefficient)
print(f"Best solution found: {best_solution}, Fitness: {fitness_function(best_solution)}")
```

Output:

```
Generation 30: Best solution = 5.85702553210915e-07, Fitness = 3.4304748083778476e-13
Generation 31: Best solution = 5.85702553210915e-07, Fitness = 3.4304748083778476e-13
Generation 32: Best solution = 5.85702553210915e-07, Fitness = 3.4304748083778476e-13
Generation 33: Best solution = 5.85702553210915e-07, Fitness = 3.4304748083778476e-13
Generation 34: Best solution = 5.85702553210915e-07, Fitness = 3.4304748083778476e-13
Generation 35: Best solution = 5.85702553210915e-07, Fitness = 3.4304748083778476e-13
Generation 36: Best solution = 5.85702553210915e-07, Fitness = 3.4304748083778476e-13
Generation 37: Best solution = 9.609376734735526e-08, Fitness = 9.234012123007639e-15
Generation 38: Best solution = 9.609376734735526e-08, Fitness = 9.234012123007639e-15
Generation 39: Best solution = 9.609376734735526e-08, Fitness = 9.234012123007639e-15
Generation 40: Best solution = 9.609376734735526e-08, Fitness = 9.234012123007639e-15
Generation 41: Best solution = 9.609376734735526e-08, Fitness = 9.234012123007639e-15
Generation 42: Best solution = 9.609376734735526e-08, Fitness = 9.234012123007639e-15
Generation 43: Best solution = -2.2749074571960418e-08, Fitness = 5.17520393880616e-16
Generation 44: Best solution = -2.2749074571960418e-08, Fitness = 5.17520393880616e-16
Generation 45: Best solution = -2.2749074571960418e-08, Fitness = 5.17520393880616e-16
Generation 46: Best solution = -2.2749074571960418e-08, Fitness = 5.17520393880616e-16
Generation 47: Best solution = -2.2749074571960418e-08, Fitness = 5.17520393880616e-16
Generation 48: Best solution = -2.2749074571960418e-08, Fitness = 5.17520393880616e-16
Generation 49: Best solution = -2.2749074571960418e-08, Fitness = 5.17520393880616e-16
Generation 50: Best solution = -2.2749074571960418e-08, Fitness = 5.17520393880616e-16
Best solution found: -2.2749074571960418e-08, Fitness: 5.17520393880616e-16
```

Program 3

Ant Colony Optimization

Problem Statement:
Ant Colony Optimization (ACO) algorithm solves combinatorial optimization problem, such as finding the shortest path, optimizing resource allocation, or scheduling tasks. The algorithm should simulate the behavior of ants by using pheromone trails and heuristic information to iteratively discover and refine optimal or near-optimal solutions while adhering to the constraints of the problem.

Algorithm:

. Ant Colony Optimization (ACO) for Traveling SalesMan

ACO is a metaheuristic inspired by ants foraging behaviour, used to find the shortest route for the Traveling Salesman Problem

uses:

. Path optimization: Finding the shortest route visiting each city once

. Combinatorial optimizatⁿ: Solving routing and scheduling issues

Application Fields:

1) Logistics : Delivery route planning

2) Telecommunications: Network routing

3) Robotics : Path planning

4) Tourism: Travel itienary optimization

5) Manufacturing : Production scheduling

. Optimization Techniques:

1) Pheromone update: Adjusting pheromone levels based on solution quality

2) Heuristic informatⁿ: Combining pheromones with distance heuristics

3) Parameter Tuning : optimizing evaporatⁿ rates and ant nos.

4) Hybrid ACO: Combining with local search methods

1. **Initialization**:
    - Define the number of nodes, ants, and initialize parameters: pheromone matrix, distance matrix, and heuristic information.
    - Set initial pheromone levels to 1 on all edges.
2. **Ant Construction Phase**:
   For each ant:
    a. Start from a random node.
    b. Repeat until all nodes are visited:
        i. Compute the probability of moving to each unvisited node based on pheromone level and heuristic value (distance).
        ii. Select the next node probabilistically and move there.
        iii. Update the ant's path and distance.
    c. Complete the tour by returning to the starting node.
3. **Pheromone Update Phase**:
    - Evaporate pheromone on all edges by multiplying them with 1 - evaporation rate.
    - For each ant, deposit pheromone along its path proportionally to the inverse of its tour distance.
4. **Iterative Optimization**:
    - Repeat the **Ant Construction Phase** and **Pheromone Update Phase** for a fixed number of iterations.
    - Track the best solution (path and distance) found across all iterations.
5. **Return the Best Solution**:
    - Output the shortest path and its corresponding distance.

Code:
```
import numpy as np
import random

class Ant:
    def __init__(self, num_nodes):
        self.path = []
        self.distance = 0
        self.num_nodes = num_nodes

    def visit_node(self, node, distance_matrix):
        if len(self.path) > 0:
            self.distance += distance_matrix[self.path[-1]][node]
        self.path.append(node)

    def tour_complete(self, distance_matrix):
        return_to_start = distance_matrix[self.path[-1]][self.path[0]]
        self.distance += return_to_start
        self.path.append(self.path[0])  # return to start node

class AntColonyOptimizer:
```

```
def __init__(self, num_nodes, distance_matrix, num_ants, alpha=1, beta=2, evaporation=0.5, q=10):
    self.num_nodes = num_nodes
    self.distance_matrix = distance_matrix
    self.num_ants = num_ants
    self.alpha = alpha
    self.beta = beta
    self.evaporation = evaporation
    self.q = q
    self.pheromone = np.ones((num_nodes, num_nodes))

def _probability(self, i, j, visited):
    pheromone = self.pheromone[i][j] ** self.alpha
    heuristic = (1 / self.distance_matrix[i][j]) ** self.beta
    return pheromone * heuristic if j not in visited else 0

def _select_next_node(self, ant):
    unvisited = [node for node in range(self.num_nodes) if node not in ant.path]
    probabilities = [self._probability(ant.path[-1], node, ant.path) for node in unvisited]
    total = sum(probabilities)
    if total == 0: return random.choice(unvisited)
    probabilities = [p / total for p in probabilities]
    return np.random.choice(unvisited, p=probabilities)

def _update_pheromones(self, ants):
    self.pheromone *= (1 - self.evaporation)
    for ant in ants:
        contribution = self.q / ant.distance
        for i in range(len(ant.path) - 1):
            u, v = ant.path[i], ant.path[i + 1]
            self.pheromone[u][v] += contribution
            self.pheromone[v][u] += contribution

def run(self, iterations=100):
    best_distance = float('inf')
    best_path = []

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

        for ant in ants:
            ant.visit_node(random.randint(0, self.num_nodes - 1), self.distance_matrix)
            while len(ant.path) < self.num_nodes:
                next_node = self._select_next_node(ant)
                ant.visit_node(next_node, self.distance_matrix)
            ant.tour_complete(self.distance_matrix)

            if ant.distance < best_distance:
```

```
                best_distance = ant.distance
                best_path = ant.path

            self._update_pheromones(ants)

        return best_path, best_distance

# Example Usage
if __name__ == "__main__":
    num_nodes = 5
    distance_matrix = np.array([
        [0, 2, 2, 3, 4],
        [2, 0, 4, 5, 3],
        [2, 4, 0, 2, 3],
        [3, 5, 2, 0, 5],
        [4, 3, 3, 5, 0]
    ])

    optimizer = AntColonyOptimizer(num_nodes, distance_matrix, num_ants=10)
    best_path, best_distance = optimizer.run(iterations=100)
    print(f"Best Path: {best_path}")
    print(f"Best Distance: {best_distance}")
```

Output:

```
Best Path: [4, 1, 0, 3, 2, 4]
Best Distance: 13
```

Program 4

Cuckoo Search Optimization

Problem Statement:
Design a Cuckoo Search algorithm to solve an optimization problem by mimicking the behavior of cuckoo birds laying eggs in host nests. The algorithm should use Lévy flights to explore the solution space, evaluate the fitness of solutions, and iteratively replace weaker solutions with stronger ones, aiming to find the optimal result for the given objective.

Algorithm:

- Cuckoo Search (CS)

It is a natural inspired optimizatⁿ algorithm modeled on the parasitic behaviour of certain cuckoo species that lay their eggs in the nests of other birds. The algorithm uses levy flights to perform a random walk to explore the search space effectively.

- Uses:
- Global optimizatⁿ problems
- Function optimizatⁿ
- Multi-objective optimizatⁿ
- Application Feilds
- Engineering design
- Image processing
- Feature selectⁿ in machine learning
- Wireless sensor networks
- Optimizatⁿ Technique
- Hybridizatⁿ with other algorithms (eg. Particle Swarm Optimization, Genetic Algorithm)
- Parameter tuning for step size (levy flight) and population size
- Hybridizatⁿ with other algorithms
- Parameter tuning for stepsize and populatⁿ size
- Dynamic control of search balance between exploratⁿ and exploitation.

Code:

```
import numpy as np
# Objective function to minimize
def objective_function(x):
    return x[0]*2 + x[1]*2  # Example: simple quadratic function

# Lévy flight step generator
def levy_flight(Lambda):
    sigma = (np.math.gamma(1 + Lambda) * np.sin(np.pi * Lambda / 2) /
             (np.math.gamma((1 + Lambda) / 2) * Lambda * 2*((Lambda - 1) / 2)))*(1 / Lambda)
    u = np.random.normal(0, sigma, 2)
    v = np.random.normal(0, 1, 2)
    step = u / np.abs(v)(1 / Lambda)
    return step

# CS parameters
num_nests = 25
discovery_rate = 0.25
iterations = 100
Lambda = 1.5  # Parameter for Lévy flights
# Initialize nests
nests = np.random.uniform(-10, 10, (num_nests, 2))
best_nest = nests[0]
best_fitness = objective_function(best_nest)
# Main loop
for _ in range(iterations):
    # Generate new solutions using Lévy flight
    for i in range(num_nests):
        step_size = levy_flight(Lambda)
        new_solution = nests[i] + step_size * (nests[i] - best_nest)
        new_fitness = objective_function(new_solution)

        # If new solution is better, replace the current solution
        if new_fitness < objective_function(nests[i]):
            nests[i] = new_solution

        # Update best solution
        if new_fitness < best_fitness:
            best_fitness = new_fitness
            best_nest = new_solution

    # Abandon a fraction of worst nests
    num_abandoned = int(discovery_rate * num_nests)
    worst_indices = np.argsort([objective_function(nest) for nest in nests])[-num_abandoned:]
    nests[worst_indices] = np.random.uniform(-10, 10, (num_abandoned, 2))
```

```
print("Best solution:", best_nest)
print("Best fitness:", best_fitness)
```

Output:

```
Best solution: [-1.13040445e+197 -1.16606789e+053]
Best fitness: -2.2608088954230963e+197
```

Program 5

Grey Wolf Optimizer:

Problem Statement:
Develop a Grey Wolf Optimizer (GWO) algorithm to solve an optimization problem by mimicking the leadership hierarchy and hunting behavior of grey wolves. The algorithm should simulate the collaborative approach of alpha, beta, delta, and omega wolves to explore and exploit the search space, aiming to find the optimal solution for the given objective.

Algorithm:

5) Grey Wolf Optimizer

Algorithm:

1) Define obj func^n $f(x) = x^2$
2) Initialize n-wolves, n-iterations, dim & boundaries
3) Intialize wolves by randomly placing wolves within search space. Calculate fitness of each wolf using obj func^n
4) Identify leadership hierarchy & rank
   Alpha wolf : Best sol^n
   Beta wolf : 2nd best ~~b~~
   Delta wolf : 3rd
5) Update wolves position:
   For each wolf:
   Calc distance D to α, β, δ wolves.
   $D\alpha = |\gamma, \alpha - \text{wolfposit}^n|$
   update posit^n based on α, β, δ influences
   $$X_{new} = \frac{X\alpha + X\beta + X\delta}{3}$$

   Decrease parameter a linearly from 2 to 0 over iterations vs exploration
6) Make sure wolf remains within bounds & recalculate fitness of α, β, δ wolves based on new heirachy
7) Perform position updates & recalculate
8) Return alpha wolf posit^n & fitness as optimal sol^n

Code:

```
# Grey Wolf

import numpy as np

def obj_fn(x):
    """Objective function to minimize."""
    return np.sum(x**2)  # Example: Sphere function

def gwo(obj_fn, dim, wolves, iters, lb, ub):
    """Grey Wolf Optimm,kkl,k,,lkpppppppppizer (GWO) implementation."""
    # Initialize wolf positions
    pos = np.random.uniform(low=lb, high=ub, size=(wolves, dim))
    a_pos, b_pos, d_pos = np.zeros(dim), np.zeros(dim), np.zeros(dim)
    a_score, b_score, d_score = float("inf"), float("inf"), float("inf")

    for t in range(iters):
        for i in range(wolves):
            fit = obj_fn(pos[i])
            # Update Alpha, Beta, Delta
            if fit < a_score:
                d_score, d_pos = b_score, b_pos.copy()
                b_score, b_pos = a_score, a_pos.copy()
                a_score, a_pos = fit, pos[i].copy()
            elif fit < b_score:
                d_score, d_pos = b_score, b_pos.copy()
                b_score, b_pos = fit, pos[i].copy()
            elif fit < d_score:
                d_score, d_pos = fit, pos[i].copy()

        # Update wolf positions
        a = 2 - t * (2 / iters)  # Linearly decreasing factor
        for i in range(wolves):
            for j in range(dim):
                r1, r2 = np.random.rand(), np.random.rand()
                A1, C1 = 2 * a * r1 - a, 2 * r2
                D_a = abs(C1 * a_pos[j] - pos[i, j])
                X1 = a_pos[j] - A1 * D_a

                r1, r2 = np.random.rand(), np.random.rand()
                A2, C2 = 2 * a * r1 - a, 2 * r2
                D_b = abs(C2 * b_pos[j] - pos[i, j])
                X2 = b_pos[j] - A2 * D_b

                r1, r2 = np.random.rand(), np.random.rand()
```

```
            A3, C3 = 2 * a * r1 - a, 2 * r2
            D_d = abs(C3 * d_pos[j] - pos[i, j])
            X3 = d_pos[j] - A3 * D_d

            # Update position
            pos[i, j] = (X1 + X2 + X3) / 3

        # Keep wolves within bounds
        pos[i] = np.clip(pos[i], lb, ub)

    # Print progress
    print(f"Iter {t+1}/{iters}, Best Score: {a_score}, Best Pos: {a_pos}")

  return a_score, a_pos

# Parameters
dim = 5       # Problem dimension
wolves = 20   # Number of wolves
iters = 50   # Number of iterations
lb = -10      # Lower bound
ub = 10       # Upper bound

# Run GWO
best_score, best_pos = gwo(obj_fn, dim, wolves, iters, lb, ub)
print("\nFinal Best Score:", best_score)
print("Final Best Pos:", best_pos)
```

Output:

```
Iter 44/50, Best Score: 3.6384318663945005e-09, Best Pos: [-3.03609398e-05  2.85365921e-05  2.57152534e-05  2.68[illegible]e-05
  2.28284055e-05]
Iter 45/50, Best Score: 3.3320450798049916e-09, Best Pos: [-2.52105791e-05  2.68237075e-05  2.20522287e-05  3.18528402e-05
  2.18187139e-05]
Iter 46/50, Best Score: 3.0629745568651214e-09, Best Pos: [-2.73630476e-05  2.82810426e-05  1.98730573e-05  2.64678645e-05
  2.04678909e-05]
Iter 47/50, Best Score: 2.9639650951638374e-09, Best Pos: [-2.53901778e-05  2.67901836e-05  2.17957463e-05  2.69457229e-05
  2.00115839e-05]
Iter 48/50, Best Score: 2.9009306337631494e-09, Best Pos: [-2.45654095e-05  2.63794204e-05  2.22912615e-05  2.59449946e-05
  2.07738872e-05]
Iter 49/50, Best Score: 2.757516734618361e-09, Best Pos: [-2.43128984e-05  2.64208350e-05  2.11191357e-05  2.47965912e-05
  2.01853995e-05]
Iter 50/50, Best Score: 2.722932571502108e-09, Best Pos: [-2.46416651e-05  2.65663891e-05  2.05249416e-05  2.45481437e-05
  1.96484935e-05]

Final Best Score: 2.722932571502108e-09
Final Best Pos: [-2.46416651e-05  2.65663891e-05  2.05249416e-05  2.45481437e-05
  1.96484935e-05]
```

Program 6

Prallel Cellular Algorithm

Problem Statement:
Design a Parallel Cellular Algorithm to solve [specific optimization problem]. The algorithm should utilize a grid-based approach where each cell represents an independent entity capable of local computation. These cells communicate with their neighbors to iteratively improve the solution, leveraging parallel processing to accelerate convergence. The goal is to find the optimal or near-optimal solution for the given problem, ensuring efficiency and scalability across multiple processors.

Algorithm:

- Parallel Cellular Algorithm (PCA)

PCA extend the concept of Cellular Paradigm by including parallelism is computation. The algorithm operates on a grid where cells represent the solution space, and each cell interacts only with its local neighbour-hood, leading to a decentralized approach to optimizatⁿ

- Uses:
- Parallel and distributed optimizatⁿ
- Multi agent systems
- Local search optimizatⁿ
- Multi-agent system
- Local search optimizatⁿ
- Application Fields:
- Evolutionary algorithms
- Multi core Computing
- Genetic algorithms for large scale problems
- Traffic simulation and control
- Optimization Technique:
- Parallelizatⁿ across multiple processors or cores
- Dynamic load balancing to optimize resource utilization
- Hybridization with other parallel algorithms (eg. Parallel Genetic Algorithm)

Code:

```
import numpy as np
import random

# Objective function (Sphere function)
def objective_function(x):
    return np.sum(x ** 2)

# Initialize the grid (population)
def initialize_grid(grid_size, dim, bounds):
    return np.random.uniform(bounds[0], bounds[1], (grid_size, grid_size, dim))

# Evaluate fitness of the grid
def evaluate_grid(grid, objective_function):
    fitness = np.zeros((grid.shape[0], grid.shape[1]))
    for i in range(grid.shape[0]):
        for j in range(grid.shape[1]):
            fitness[i, j] = objective_function(grid[i, j])
    return fitness

# Selection using the best individual in the neighborhood
def select_best_neighbor(grid, fitness, x, y):
    neighbors = [
        ((x - 1) % grid.shape[0], y),   # Up
        ((x + 1) % grid.shape[0], y),   # Down
        (x, (y - 1) % grid.shape[1]),   # Left
        (x, (y + 1) % grid.shape[1]),   # Right
    ]
    best_pos = min(neighbors, key=lambda pos: fitness[pos[0], pos[1]])
    return grid[best_pos[0], best_pos[1]]

# Crossover operation
def crossover(parent1, parent2):
    alpha = np.random.rand()
    return alpha * parent1 + (1 - alpha) * parent2

# Mutation operation
def mutate(individual, bounds, mutation_rate=0.1):
    for i in range(len(individual)):
        if random.random() < mutation_rate:
            individual[i] += np.random.uniform(-1, 1)
            individual[i] = np.clip(individual[i], bounds[0], bounds[1])
    return individual

# Main Parallel Cellular Genetic Algorithm
```

```
def parallel_cellular_ga(objective_function, grid_size=5, dim=2, bounds=(-5, 5), max_iter=100,
mutation_rate=0.1):
    # Initialize the grid and fitness
    grid = initialize_grid(grid_size, dim, bounds)
    fitness = evaluate_grid(grid, objective_function)

    for iteration in range(max_iter):
        new_grid = np.copy(grid)

        for i in range(grid_size):
            for j in range(grid_size):
                # Select parents from the neighborhood
                parent1 = grid[i, j]
                parent2 = select_best_neighbor(grid, fitness, i, j)

                # Apply crossover and mutation
                offspring = crossover(parent1, parent2)
                offspring = mutate(offspring, bounds, mutation_rate)

                # Replace if offspring is better
                offspring_fitness = objective_function(offspring)
                if offspring_fitness < fitness[i, j]:
                    new_grid[i, j] = offspring
                    fitness[i, j] = offspring_fitness

        grid = new_grid

        # Output the best solution in the grid
        best_position = np.unravel_index(np.argmin(fitness), fitness.shape)
        best_fitness = fitness[best_position]
        print(f"Iteration {iteration + 1}: Best Fitness = {best_fitness}")

    # Return the best solution
    best_position = np.unravel_index(np.argmin(fitness), fitness.shape)
    return grid[best_position[0], best_position[1]], fitness[best_position]

# Parameters
grid_size = 5        # Size of the grid
dim = 2              # Dimensionality of the problem
bounds = (-5, 5)     # Search space boundaries
max_iter = 50        # Number of iterations
mutation_rate = 0.1  # Mutation rate

# Run PCGA
best_solution, best_fitness = parallel_cellular_ga(objective_function, grid_size, dim, bounds,
max_iter, mutation_rate)
```

# Output the best solution
print(f"\nBest solution: {best_solution}")
print(f"Best fitness: {best_fitness}")

Output:

```
Iteration 35: Best Fitness = 1.4700677504738945e-07
Iteration 36: Best Fitness = 1.4700677504738945e-07
Iteration 37: Best Fitness = 1.4700677504738945e-07
Iteration 38: Best Fitness = 1.4700677504738945e-07
Iteration 39: Best Fitness = 1.4700677504738945e-07
Iteration 40: Best Fitness = 1.4700677504738945e-07
Iteration 41: Best Fitness = 1.4700677504738945e-07
Iteration 42: Best Fitness = 1.4700677504738945e-07
Iteration 43: Best Fitness = 1.4700677504738945e-07
Iteration 44: Best Fitness = 1.4700677504738945e-07
Iteration 45: Best Fitness = 1.4700677504738945e-07
Iteration 46: Best Fitness = 1.4700677504738945e-07
Iteration 47: Best Fitness = 1.4700677504738945e-07
Iteration 48: Best Fitness = 1.4700677504738945e-07
Iteration 49: Best Fitness = 1.4700677504738945e-07
Iteration 50: Best Fitness = 1.4700677504738945e-07

Best solution: [ 0.00036358 -0.00012172]
Best fitness: 1.4700677504738945e-07
```

Program 7

Optimization via Gene Expression

Problem Statement:
Design an optimization system using the Gene Expression Algorithm to evolve mathematical expressions that minimize a given cost function. The problem requires creating a population of encoded mathematical expressions (genes) that are iteratively refined through genetic operations like selection, crossover, and mutation. The goal is to decode these expressions and evaluate their fitness based on how closely they approximate the desired output of the cost function, ensuring the algorithm converges to the most optimal solution over successive generations.

Algorithm:

- Gene Expression Programming (GEP)

Gene Expression Programming (GEP) is an evolutionary algorithm that encodes potential solutions as linear chromosomes, which are then expressed as nonlinear structures to solve complex problems. It is inspired by genetic programming and uses a genetic algorithm approach for evolution.

- Uses:
- Symbolic regression
- Function approximation
- Classification and prediction
- Application Fields:
- Financial modeling
- Control system
- Bioinformatics
- Robotics (evolution of controller)
- Optimization Technique
- Fitness function optimization (e.g, dynamic adaptation of selection pressure)
- Mutation and crossover parameter tuning
- Hybridization with neural networks or other evolutionary algorithms.

Code:
```
import random
import math

# --- PARAMETERS ---
POPULATION_SIZE = 50
GENE_LENGTH = 30
GENERATIONS = 100
MUTATION_RATE = 0.05
CROSSOVER_RATE = 0.7

# Terminals (constants, variable 'x') and Functions
TERMINALS = ['x', '1', '2', '3', '4', '5']
FUNCTIONS = ['+', '-', '*', '/', 'sin', 'cos']

# Target Cost Function (to minimize)
def cost_function(x):
    """ Example cost function to minimize. Replace with your target function. """
    return x**2 - 10 * math.sin(2 * x)

# --- GENE EXPRESSION CLASS ---
class GeneExpression:
    def __init__(self):
        self.gene = self._random_gene()
        self.cached_fitness = None  # To store fitness value

    def _random_gene(self):
        """ Initialize a random gene sequence. """
        return [random.choice(TERMINALS + FUNCTIONS) for _ in range(GENE_LENGTH)]

    def decode_gene(self, x):
        """ Decode the gene into a mathematical expression and evaluate it. """
        stack = []
        for token in self.gene:
            if token in TERMINALS:
                stack.append(float(x) if token == 'x' else float(token))
            elif token in FUNCTIONS:
                if len(stack) >= 1 and token in ['sin', 'cos']:
                    arg = stack.pop()
                    stack.append(math.sin(arg) if token == 'sin' else math.cos(arg))
                elif len(stack) >= 2:
                    b, a = stack.pop(), stack.pop()
                    if token == '+': stack.append(a + b)
                    elif token == '-': stack.append(a - b)
```

```
                elif token == '*': stack.append(a * b)
                elif token == '/' and b != 0: stack.append(a / b)
            else:
                return float('inf')  # Malformed gene
        return stack[0] if len(stack) == 1 else float('inf')

    def fitness(self, x):
        """ Evaluate fitness: minimize cost_function(output). """
        if self.cached_fitness is None:
            try:
                result = self.decode_gene(x)
                self.cached_fitness = abs(cost_function(result))
            except:
                self.cached_fitness = float('inf')
        return self.cached_fitness

# --- GENETIC OPERATIONS ---
def selection(population, fitnesses):
    """ Tournament selection: Select the best from random candidates. """
    tournament_size = 3
    candidates = random.sample(list(zip(population, fitnesses)), tournament_size)
    return min(candidates, key=lambda c: c[1])[0]

def crossover(parent1, parent2):
    """ Perform single-point crossover between two parents. """
    if random.random() < CROSSOVER_RATE:
        point = random.randint(1, GENE_LENGTH - 1)
        child1 = GeneExpression()
        child2 = GeneExpression()
        child1.gene = parent1.gene[:point] + parent2.gene[point:]
        child2.gene = parent2.gene[:point] + parent1.gene[point:]
        return child1, child2
    return parent1, parent2

def mutate(individual):
    """ Apply mutation by altering random parts of the gene. """
    for i in range(GENE_LENGTH):
        if random.random() < MUTATION_RATE:
            individual.gene[i] = random.choice(TERMINALS + FUNCTIONS)

# --- MAIN EVOLUTION FUNCTION ---
def geneExpression():
    # Initialization
    population = [GeneExpression() for _ in range(POPULATION_SIZE)]
    x_value = random.uniform(-10, 10)  # Random input to test optimization

    # Evolutionary loop
```

```
    for generation in range(GENERATIONS):
        fitnesses = [ind.fitness(x_value) for ind in population]
        best_idx = fitnesses.index(min(fitnesses))
        print(f"Generation {generation}: Best Fitness = {fitnesses[best_idx]:.5f}")

        # Elitism: Preserve the best individual
        new_population = [population[best_idx]]

        # Create next generation
        while len(new_population) < POPULATION_SIZE:
            parent1 = selection(population, fitnesses)
            parent2 = selection(population, fitnesses)
            child1, child2 = crossover(parent1, parent2)
            mutate(child1)
            mutate(child2)
            new_population.extend([child1, child2])

        population = new_population

    # Final Solution
    final_fitnesses = [ind.fitness(x_value) for ind in population]
    best_idx = final_fitnesses.index(min(final_fitnesses))
    print("\nOptimized Solution:")
    print(f"Best Gene: {population[best_idx].gene}")
    print(f"Best Fitness: {final_fitnesses[best_idx]:.5f}")

if __name__ == "__main__":
    geneExpression()
```

Output:

```
Generation 0: Best Fitness = inf
Generation 1: Best Fitness = inf
Generation 2: Best Fitness = inf
Generation 3: Best Fitness = inf
Generation 4: Best Fitness = 37.56666
Generation 5: Best Fitness = 37.56666
Generation 6: Best Fitness = 37.56666
Generation 7: Best Fitness = 37.56666
Generation 8: Best Fitness = 37.56666
Generation 9: Best Fitness = 37.56666
Generation 10: Best Fitness = 37.56666
Generation 11: Best Fitness = 37.56666
Generation 12: Best Fitness = 37.56666
Generation 13: Best Fitness = 26.06944
Generation 14: Best Fitness = 26.06944
Generation 15: Best Fitness = 26.06944
Generation 16: Best Fitness = 26.06944
Generation 17: Best Fitness = 26.06944
Generation 18: Best Fitness = 26.06944
```

```
Generation 81: Best Fitness = 0.00015
Generation 82: Best Fitness = 0.00015
Generation 83: Best Fitness = 0.00015
Generation 84: Best Fitness = 0.00015
Generation 85: Best Fitness = 0.00015
Generation 86: Best Fitness = 0.00015
Generation 87: Best Fitness = 0.00015
Generation 88: Best Fitness = 0.00015
Generation 89: Best Fitness = 0.00015
Generation 90: Best Fitness = 0.00015
Generation 91: Best Fitness = 0.00015
Generation 92: Best Fitness = 0.00015
Generation 93: Best Fitness = 0.00015
Generation 94: Best Fitness = 0.00015
Generation 95: Best Fitness = 0.00015
Generation 96: Best Fitness = 0.00015
Generation 97: Best Fitness = 0.00015
Generation 98: Best Fitness = 0.00015
Generation 99: Best Fitness = 0.00015

Optimized Solution:
Best Gene: ['-', '5', '*', '4', 'sin', '*', '+', '*', '+', '1', '5', '*', 'sin', '3', '-', 'x', '4', 'cos', '+', '*', '-', '4', '5', 'x', 'cos', 'x'
Best Fitness: 0.00015
```