VISVESVARAYA TECHNOLOGICAL UNIVERSITY

“JnanaSangama”, Belgaum -590014, Karnataka.

LAB RECORD

Bio Inspired Systems (23CS5BSBIS)

Submitted by

Mohammed Uzair Obaid (1BM22CS159)

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 **Mohammed Uzair Obaid(1BM22CS159),** 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.

Radhika A D Assistant Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE

Index

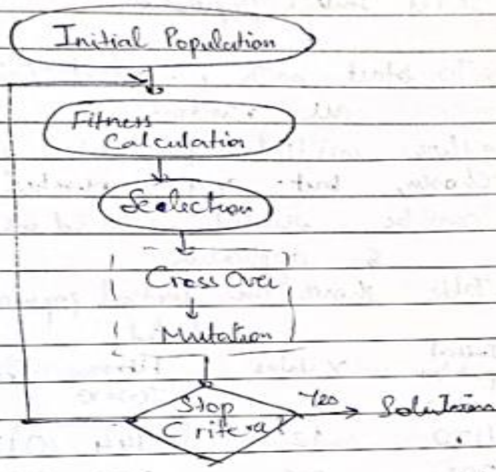
Github Link: github.com/uzairob/bis-lab

Program 1
Genetic Algorithm for Optimization Problems.
Problem Statement:
Genetic Algorithms (GA) are inspired by the process of natural selection and genetics, where the fittest individuals are selected for reproduction to produce the next generation. GAs are widely used for solving optimization and search problems. Implement a Genetic Algorithm using Python to solve a basic optimization problem, such as finding the maximum value of a mathematical function.

Algorithm:

PAGE NO :
DATE : 16/10.

Genetic Algorithms.



Genetic Algorithm Solved Example.

Consider the function of maximizing the function $f(x)=x^2$, where the permitted values of x vary between 0 to 31

Steps
1. Select Encoding Technique.
The min value is 0 & max value is 31
So to represent ~~this~~, we need 5-bit binary integer, numbers between 0(00000) & 31(11111) can be obtained.

The Objective function here is $f(x) = x$, which is to be maximized

DATE :

2. Select Initial population

- To start with, select initial population all random
- Here initial population of size 4 is chosen, but any number of populations can be selected based on the requirement & application
- Table shows an initial population randomly selected

String no.	Initial Population	X Value	Fitness $f(x)=x^2$	Prob	% Prob	Expected Count	Ranked Actual Count
1	01100	12	144	0.1247	12.47	0.4987	1
2	11001	25	625	0.5411	54.11	2.1645	2
3	00101	5	25	0.0216	2.16	0.0866	0
4	10011	19	181	0.3126	31.26	1.2504	1
Sum			1155	1.0	100	4	4
Average			288.75	0.25	25	1	1
Max			625	0.5411	54.11	2.1645	2.

$$Prob = \frac{f(x)}{\Sigma f(x)} \qquad \Sigma f(x) = 1155$$

$$\text{Expected Count} = \frac{f(x_i)}{Avg(\Sigma f(x))}$$

From above table, so each chromosome is taken in Actual count no of times.

PAGE NO: 2
DATE:

3. Selecting the correct initial population from the 4. This is done using actual count.

String no	Mating pool	Crossover point (Random)	Offspring after crossover	x value	fitness $f(x)=x^2$
1	0110|0	swap 4	01101	13	169
2	1100|1	4	11000	24	576
3	11|001	2	11011	27	729
4	10|011	2	10001	17	289
Sum					1763
Avg					440.75
max					729

initial max = 625 new = 729.

4. Mutation (final mutation chromosome)

String no	Offspring	mutation chromosome for flipp	Offspring after mutation	x val.	$f(x) = x^2$
1	01101	10000	11101	29	841
2	11000	00000	11000	24	576
3	11011	00000	11011	27	724
4	10001	00101	10100	20	400
Sum		flip bit		next gen	2546
Avg					636.5
Max					841

841 → fittest value

Result:- It is observed that on performing crossover selection, mutation we reach max on iterations i.e $f(x) = x^2$ reaches max.

3

CODE:

```python
import random

# Define the fitness function
def fitness_function(x):
    return x**2  # Example function: f(x) = x^2

# Generate initial population
def generate_population(size, x_min, x_max):
    return [random.uniform(x_min, x_max) for _ in range(size)]

# Selection process
def select_parents(population, fitnesses):
    total_fitness = sum(fitnesses)
    selection_probs = [f / total_fitness for f in fitnesses]
    parents = random.choices(population, weights=selection_probs, k=2)
    return parents

# Crossover process
def crossover(parent1, parent2):
    alpha = random.random()
    child = alpha * parent1 + (1 - alpha) * parent2
    return child

# Mutation process
def mutate(child, mutation_rate, x_min, x_max):
    if random.random() < mutation_rate:
        child = random.uniform(x_min, x_max)
    return child

# Genetic Algorithm
def genetic_algorithm(pop_size, generations, mutation_rate, x_min, x_max):
    population = generate_population(pop_size, x_min, x_max)
    for generation in range(generations):
        fitnesses = [fitness_function(ind) for ind in population]
        new_population = []
        for _ in range(pop_size):
            parent1, parent2 = select_parents(population, fitnesses)
            child = crossover(parent1, parent2)
            child = mutate(child, mutation_rate, x_min, x_max)
            new_population.append(child)
        population = new_population
    best_solution = max(population, key=fitness_function)
```

```python
    return best_solution

# User inputs
pop_size = int(input("Enter population size: "))
generations = int(input("Enter number of generations: "))
mutation_rate = float(input("Enter mutation rate (0-1): "))
x_min = float(input("Enter minimum value of x: "))
x_max = float(input("Enter maximum value of x: "))

# Run the genetic algorithm
best_solution = genetic_algorithm(pop_size, generations, mutation_rate, x_min,
x_max)
print(f"The best solution found is: {best_solution} with fitness value:
{fitness_function(best_solution)}")
```

OUTPUT:

```
Enter population size: 20
Enter number of generations: 3
Enter mutation rate (0-1): 2
Enter minimum value of x: 4
Enter maximum value of x: 8
The best solution found is: 7.871117037734696 with fitness value: 61.95448342171742
```

Program 2

Particle Swarm Optimization for Function Optimization.

Problem Statement:

Particle Swarm Optimization (PSO) is inspired by the social behavior of birds flocking or fish schooling. PSO is used to find optimal solutions by iteratively improving a candidate solution with regard to a given measure of quality. Implement the PSO algorithm using Python to optimize a mathematical function.

Algorithm:

Algorithm.

1. Initialize PSO parameters:
   - population size: N (No. of particles)
   - Inertia weight (w)
   - Cognitive coefficient c_1: controls the particle's inclination to its personal best position
   - Social coefficient c_2: controls the particle's inclination towards the global best
   - Max iterations T

2. Generate First Swarm:
   For each particle i, initialize.
   - Position x_i randomly within the search space
   - velocity v_i also randomly.

3. Evaluate the Fitness of All particles.
   For each particle, evaluate its fitness $f(x_i)$ based on the objective function which represent how good the solution at x_i.

4. Record Personal Best Fitness of all particles:
   update its personal best p_i, if its curr fitness is better than its previous personal best fitness:
   $$\text{if } f(x_i) > f(p_i) \text{ then } p_i = x_i$$

5. Find Global best particle:
   Identify particle with best fitness among all particles in the swarm
   $$g = \max_i f(p_i)$$
   where $f(p_i)$ is the personal best fitness of particle i

DATE :

6. Update the velocity of particles.

For each particle i, update its velocity v_i

$$v_i^{(t+1)} = w \cdot v_i(t) + C_1 \cdot r_1 \cdot (p_i - x_i) + C_2 \cdot r_2 \cdot (g - x_i)$$

w - inertia weight

C_1 & C_2 → Cognitive & Social coefficient

r_1 & r_2 → random values between 0 & 1.

7. Update the position of particles:

Update the position x_i of each position using the new velocity

$$x_i = x_i + v_i$$

8. Check Termination Criteria:

If max no. of iteration T is met, stop the algorithm. If not go back to step 3 & continue iterating.

Output:

when iteration set to 100

iteration 1/100 global best : 8.5399.

iteration 2/100 global best : 5.5

.

.

iteration 100/100 global best : 0.9944

Best Position : [-9.949e-01, -5.21608]

Best ~~velocity~~ Score : 0.99

13/11/24

CODE:

```
import numpy as np

# Objective function (Example: Rastrigin function)
def objective_function(position):
    return sum([x**2 - 10 * np.cos(2 * np.pi * x) + 10 for x in position])

# Particle Swarm Optimization
class Particle:
    def __init__(self, dimensions):
        self.position = np.random.uniform(-10, 10, dimensions)  # Initialize
position
        self.velocity = np.random.uniform(-1, 1, dimensions)    # Initialize
velocity
        self.best_position = self.position.copy()               # Personal
best position
        self.best_score = float('inf')                          # Best score
for personal best

    def update_velocity(self, global_best_position, inertia, cognitive_const,
social_const):
        r1, r2 = np.random.rand(), np.random.rand()
        cognitive = cognitive_const * r1 * (self.best_position -
self.position)
        social = social_const * r2 * (global_best_position - self.position)
        self.velocity = inertia * self.velocity + cognitive + social

    def update_position(self):
        self.position += self.velocity

# PSO Algorithm
def particle_swarm_optimization(objective_func, dimensions, num_particles,
max_iter):
    inertia = 0.5            # Inertia weight
    cognitive_const = 1.5    # Cognitive constant
    social_const = 1.5       # Social constant

    # Initialize particles
    swarm = [Particle(dimensions) for _ in range(num_particles)]
```

```
    global_best_position = np.random.uniform(-10, 10, dimensions)
    global_best_score = float('inf')

    for iteration in range(max_iter):
        for particle in swarm:
            # Evaluate fitness
            fitness = objective_func(particle.position)
            # Update personal best
            if fitness < particle.best_score:
                particle.best_score = fitness
                particle.best_position = particle.position.copy()

            # Update global best
            if fitness < global_best_score:
                global_best_score = fitness
                global_best_position = particle.position.copy()

        # Update velocity and position for each particle
        for particle in swarm:
            particle.update_velocity(global_best_position, inertia,
cognitive_const, social_const)
            particle.update_position()

        print(f"Iteration {iteration+1}/{max_iter}, Global Best Score:
{global_best_score}")

    return global_best_position, global_best_score

# Example usage
best_position, best_score = particle_swarm_optimization(objective_function,
dimensions=2, num_particles=30, max_iter=100)
print("Best Position:", best_position)
print("Best Score:", best_score)
```

OUTPUT:

```
Iteration 95/100, Global Best Score: 0.9949590570932898
Iteration 96/100, Global Best Score: 0.9949590570932898
Iteration 97/100, Global Best Score: 0.9949590570932898
Iteration 98/100, Global Best Score: 0.9949590570932898
Iteration 99/100, Global Best Score: 0.9949590570932898
Iteration 100/100, Global Best Score: 0.9949590570932898
Best Position: [ 9.94958639e-01 -1.58811738e-09]
Best Score: 0.9949590570932898
```

Program 3

Ant Colony Optimization for the Traveling Salesman Problem

Problem Statement:

The foraging behavior of ants has inspired the development of optimization algorithms that can solve complex problems such as the Traveling Salesman Problem (TSP). Ant Colony Optimization (ACO) simulates the way ants find the shortest path between food sources and their nest. Implement the ACO algorithm using Python to solve the TSP, where the objective is to find the shortest possible route that visits a list of cities and returns to the origin city.

Algorithm:

DATE: 23/10

Ant Colony Optimization

Ants → Usocial
→ pheromone → chemical
→ finding shortest path

1 $\Delta \tau_{ij}^{k} = \begin{cases} \frac{1}{L_k} & ; k\text{th ant travels on the edge } i,j \\ 0 & \text{otherwise} \end{cases}$

L_k is the length of the path

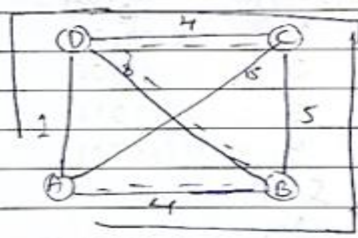
2) $\tau_{ij}^{k} = \sum_{k=1}^{m} \Delta \tau_{i,j}^{k}$ [without vapourization]

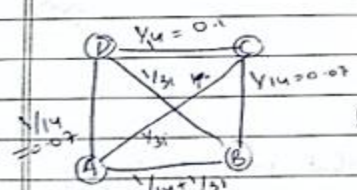
3 $\tau_{ij}^{k} = (1-\rho)\,\tau_{ij} + \sum_{k=1}^{m} \Delta \tau_{ij}^{k}$

ρ - is a constant → 0 - 1

Cost Matrix

	A	B	C	D
A	0	4	15	1
B	4	0	5	8
C	15	5	0	4
D	1	8	4	0

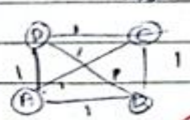


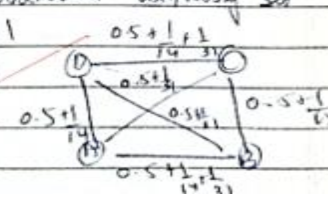


Traveller will choose path with higher pheromone level without vapourization

To Calculate the vapourization initially set all path to 1

$\rho = 0.5$





PAGE NO :
DATE :

$$\tau_{ij}^{k} = (1-\rho)\tau_{ik} + \sum_{k=1}^{m} \Delta\tau_{ij}^{k}$$

$$= 0.1 \times 1 + \frac{1}{14} + \frac{1}{31}$$

Starting from A check adjacent vertices probabilities

A→B $\quad P_{ij} = \dfrac{1 \times 1}{\left(1 \times \frac{1}{1} + 1 \times \frac{1}{15} + \frac{1}{4} \times 1\right)} = 0.754$

A→C $\quad P_{ij} = \dfrac{1 \times \frac{1}{4}}{\left(1 \times \frac{1}{1} + 1 \times \frac{1}{15} \times \frac{1}{4} \times 1\right)} = 0.18.$

Implementation

→ Algorithm:

1. Initialization :

- Define parameters : Number of ants (N), number of cities (M)
  τ_0 (Initial pheromone level on all edges), ρ (pheromone evaporation rate) $(0 < \rho < 1)$, α (Influence of pheromone on path selection)
  β → influence of heuristic information on path selection.
  ~~β~~ → Initialize pheromone levels $\tau_{ij} = \tau_0$ for all edges in τ

2. Mainloop.

For each ant :

Randomly choose a starting city.

Repeat until all cities are visited

Calculate the probability P_{ij} of moving from city i to city j using

PAGE NO :
DATE :

$$P_{ij} = \frac{\tau_{ij}^{\alpha} \cdot (\eta_{ij}^{\beta})}{\sum_{k \in \text{allowed/unvisited}} \tau_{ik}^{\alpha} \cdot \eta_{ik}^{\beta}}$$

$\tau \rightarrow$ pheromone tr.
$\eta \rightarrow$ heuristic info

Choose next city j based on P_{ij}.
Complete the ~~rout~~ tour by returning to the starting city

Update pheromone:
For edge (i,j):
evaporate the pheromone $\rho \rightarrow$ pheromone decay

$$\tau_{ij} \leftarrow (1-\rho) \cdot \tau_{ij}$$

Add pheromone based on the quality of solutions. For each ant k, if it includes edge(i,j) in its tour with length L_k:

$$\tau_{ij} \leftarrow \tau_{ij} + \frac{Q}{L_k}$$

$Q \rightarrow$ pheromone intensity.

Keep track of best solution found across all iterations.
Repeat main loop until stopping criteria is met
Return the best tour.

Cost Matrix:

	A	B	C	D
A	0	4	15	1
B	4	0	5	8
C	15	5	0	4
D	1	8	4	0

Output:
Best Tour: [2, 1, 0, 3, 2]
Best length = 14.

23/10/24

CODE:

```
import numpy as np
import random

class AntColony:
    def __init__(self, distance_matrix, n_ants, n_iterations, decay,
alpha=1, beta=1):
        self.distance_matrix = distance_matrix
```

```
        self.pheromone = np.ones(distance_matrix.shape) / 
len(distance_matrix)
        self.n_ants = n_ants
        self.n_iterations = n_iterations
        self.decay = decay
        self.alpha = alpha  # Pheromone importance
        self.beta = beta    # Distance importance
        self.all_indices = range(len(distance_matrix))

    def run(self):
        shortest_path = None
        all_time_shortest_path = ("path", np.inf)

        for _ in range(self.n_iterations):
            all_paths = self.generate_all_paths()
            self.update_pheromones(all_paths)
            shortest_path = min(all_paths, key=lambda x: x[1])
            if shortest_path[1] < all_time_shortest_path[1]:
                all_time_shortest_path = shortest_path

        return all_time_shortest_path

    def generate_all_paths(self):
        all_paths = []
        for _ in range(self.n_ants):
            path = self.generate_path(0)  # Start from city 0
            path_dist = self.calculate_path_distance(path)
            all_paths.append((path, path_dist))
        return all_paths

    def generate_path(self, start):
        path = [start]
        visited = set(path)
        while len(visited) < len(self.distance_matrix):
            move = self.select_next_city(path[-1], visited)
            path.append(move)
            visited.add(move)
        path.append(start)  # Return to starting city
        return path

    def select_next_city(self, current_city, visited):
        pheromone = np.copy(self.pheromone[current_city])
```

```
        pheromone[list(visited)] = 0  # Avoid visiting already visited
cities

        probabilities = pheromone ** self.alpha * ((1 /
self.distance_matrix[current_city]) ** self.beta)
        probabilities /= probabilities.sum()  # Normalize probabilities

        next_city = np.random.choice(self.all_indices, p=probabilities)
        return next_city

    def calculate_path_distance(self, path):
        total_dist = 0
        for i in range(len(path) - 1):
            total_dist += self.distance_matrix[path[i]][path[i + 1]]
        return total_dist

    def update_pheromones(self, all_paths):
        self.pheromone *= (1 - self.decay)  # Pheromone evaporation
        for path, dist in all_paths:
            for i in range(len(path) - 1):
                self.pheromone[path[i]][path[i + 1]] += 1 / dist  # Update
pheromone based on path quality

# Example: A 4-city TSP problem
if __name__ == "__main__":
    distance_matrix = np.array([[np.inf, 12, 12, 15],
                                [12, np.inf, 13, 14],
                                [12, 13, np.inf, 11],
                                [15, 14, 11, np.inf]])

    colony = AntColony(distance_matrix, n_ants=10, n_iterations=100,
decay=0.1, alpha=1, beta=2)
    best_path = colony.run()
    print("Best path found:", best_path)
```

OUTPUT:

```
Best path found: ([0, 1, 3, 2, 0], 49.0)
```

Program 4

Cuckoo Search (CS)

Problem Statement:

Cuckoo Search (CS) is a nature-inspired optimization algorithm based on the brood parasitism of some cuckoo species. This behavior involves laying eggs in the nests of other birds, leading to the optimization of survival strategies. CS uses Lévy flights to generate new solutions, promoting global search capabilities and avoiding local minima. The algorithm is widely used for solving continuous optimization problems and has applications in various domains, including engineering design, machine learning, and data mining.

Algorithm:

PAGE NO :
DATE :

Cuckoo Search Algorithm [CSA]

The CSA is a nature inspired optimization technique based on the brood ~~parasitism~~ parasitism behaviour of some cuckoo species. It combines Lévy flights for exploration with a ~~stav~~ strategy to replace worse sol(n) for exploitation.

Concepts:

1. Cuckoo Behaviour:

→ Cuckoos lay their eggs in the nests of other host birds

→ If a host bird discovers the cuckoo eggs, it may abandon the nest or build a new one elsewhere

2. Optimization:-

• The algo treats each nest as a potential sol^n

• The objective is to replace poorer sol^ns with better ones, ultimately finding the global optimum

3. Lévy flights:

A type of random walk with step sizes drawn from a Lévy distribution. This allows for occasional long jumps, enabling the algorithm to explore new regions in the search space

4. Abandon worst sol^ns :

Algorithm:

Objective: Maximize (or minimize) a given function $f(x)$; where $x = (x_1, x_2 \ldots x_d)^T$.

1. Initialization

1. Define the objective function $f(x)$

$$f(x) = \sum_{i=1}^{d} x_i^2.$$

$$f(x) = x_1^2 - x_1 x_2 + x_2^2 + 2x_1 + 4x_2 + 3$$

$$x = (x_1, x_2).$$

2. Generate an initial population of n host nests x_i, $i = 1 \ldots n$. where x_i is a potential sol^n.

3. Set parameters:

$t = 0$. Initial generation,

MaxGenerations

p_a: The fraction of nests to be abandoned

Step size α for Lévy flights.

While ($t <$ MaxGeneration)

Get a cuckoo randomly

Generate a new sol^n using Lévy flights.

$$x_{new}^{t+1} = x_i^t + \alpha L(\lambda).$$

where:

$\alpha > 0$ is the step size.

$$L(S) \approx s^{-\lambda} \; (\lambda \, [1, 3])$$

$\cancel{L(s)} =$

Stepsize =

$$\alpha = \frac{U}{|V|^{1/\lambda}}.$$

$$U \approx N(0, \sigma_u^2) \quad V \approx N(0, \sigma_v^2)$$

DATE :

$$\sigma_u^2 = \left[\frac{\Gamma(1+\lambda) \cdot \sin\left(\frac{\pi\lambda}{2}\right)}{\Gamma\left(\frac{1+\lambda}{2}\right) \times \lambda \times 2^{\frac{\lambda-1}{2}}} \right]^{1/\lambda} \qquad \sigma_v^2 = 1.$$

Evaluate the new fitness $f(x_j)$

Rule 2: Replace nests based on fitness.

If $f(x_j) > f(x_i)$ then $x_i = x_j$.

Rule 3: Abandon a fraction of the worst nests.

1. Abandon p_a fraction of the worst nests.
2. Generate new sol^ns for abandoned nests using local random walks.

$$x_i^{t+1} = x_i^t + H(p_a - \varepsilon) \otimes (x_j^t - x_k^t).$$

$H(u)$ is a Heaviside fn.

$p_a \rightarrow$ switching parameter.

$\varepsilon \rightarrow$ random number drawn from a uniform distribution.

x_j & x_k are two different sol^ns selected randomly by random permutation

Rank the sol^n & identify the current best.

Update counter: $t = t+1$.

Output:

Best sol^n : [0.3072, 0.1185; 1.1675, -0.8397, -1.5224]

Best objective value 4.49067

CODE:

```
import numpy as np

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

# Levy flight implementation
def levy_flight(Lambda, dim, alpha=1.0):
    u = np.random.normal(0, 1, size=dim)
    v = np.random.normal(0, 1, size=dim)
    step = alpha * (u / (np.abs(v) ** (1 / Lambda)))  # Lévy step
    return step

# Cuckoo Search Algorithm
def cuckoo_search(n, max_generations, pa, lower_bound, upper_bound, dim):
    # Step 1: Initialize nests randomly
    nests = np.random.uniform(lower_bound, upper_bound, size=(n, dim))
    fitness = np.array([objective_function(nest) for nest in nests])
    best_nest = nests[np.argmin(fitness)]
    best_fitness = np.min(fitness)

    # Iterative optimization
    for t in range(max_generations):
        # Rule 1: Generate new solutions via Lévy flight
        for i in range(n):
            new_nest = nests[i] + levy_flight(1.5, dim)
            new_nest = np.clip(new_nest, lower_bound, upper_bound)
            new_fitness = objective_function(new_nest)

            # Rule 2: Replace nests if better
            if new_fitness < fitness[i]:
                nests[i] = new_nest
                fitness[i] = new_fitness

                # Update global best
                if new_fitness < best_fitness:
                    best_nest = new_nest
                    best_fitness = new_fitness

        # Rule 3: Abandon some nests and create new random ones
        abandon = np.random.rand(n) < pa
        nests[abandon] = np.random.uniform(lower_bound, upper_bound,
```

```
size=(np.sum(abandon), dim))
        fitness[abandon] = np.array([objective_function(nest) for nest in
nests[abandon]])

    return best_nest, best_fitness

# Parameters
n = 25  # Number of nests
dim = 5  # Dimensionality of the problem
max_generations = 100  # Max iterations
pa = 0.25  # Abandonment probability
lower_bound = -10  # Lower bound of the search space
upper_bound = 10  # Upper bound of the search space

# Run Cuckoo Search
best_solution, best_value = cuckoo_search(n, max_generations, pa, lower_bound,
upper_bound, dim)

print("Best solution found:", best_solution)
print("Best objective value:", best_value)
```

OUTPUT:

```
Best solution found: [ 0.20497829 -0.33362922 -0.59123456 -0.74011305 -1.28606162]
Best objective value: 2.7046046892112336
```

Program 5

Grey Wolf Optimizer (GWO):

Problem Statement:

The Grey Wolf Optimizer (GWO) algorithm is a swarm intelligence algorithm inspired by the social hierarchy and hunting behavior of grey wolves. It mimics the leadership structure of alpha, beta, delta, and omega wolves and their collaborative hunting strategies. The GWO algorithm uses these social hierarchies to model the optimization process, where the alpha wolves guide the search process while beta and delta wolves assist in refining the search direction. This algorithm is effective for continuous optimization problems and has applications in engineering, data analysis, and machine learning.

Algorithm:

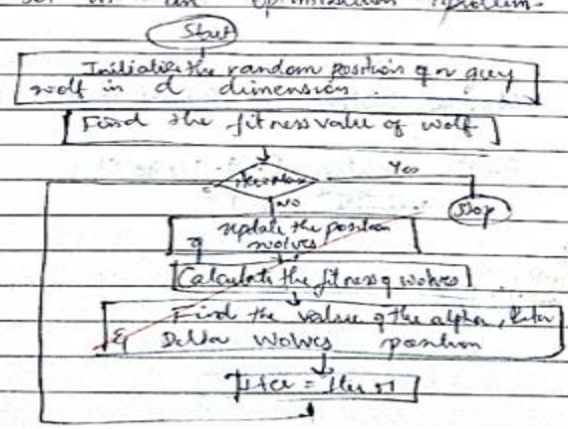
PAGE NO :
DATE : 28/11

Grey Wolf Optimization

Grey Wolf Optimization is a population based metaheuristic algorithm that mimics the hunting behaviour & social hierarchy of grey wolves.
Wolves hunt in packs, with a strict heirarchy
- Alpha (α) : The dominant wolf, representing the best solution
- Beta (β) : The second-in-command, who assists the alpha & is a candidate to replace the α.
- Delta (δ) : Subordinate to alpha & beta but hold more authority than omega.
- Omega (ω) :- The lowest ranking wolves, tasked with following the pack's order.

The hunting process involves tracking, chasing, approaching, encircling & finally attacking the prey.

This behaviour is used as inspiration for here the algorithm searches for optimal sol^n in an optimization problem.



A → Exploration & Exploitation
C → Prey's influence on the wolf position.

PAGE NO :
DATE :

1) Encircling the Prey.

Distance between wolf's position $X(t)$ & the prey's position $X_p(t)$

$$D = |C \cdot X_p(t) - X(t)|$$

• C is a random coefficient vector used to adjust the direction & speed of movement.

2) Position Update : Encircling.

$$X(t+1) = X_p(t) - A \cdot D.$$

3. Coefficient Calculation :

$$A = 2 \cdot a \cdot r_1 - a$$

$C = 2 \cdot r_2$ a → Linearly decreases from 2 to 0 over iteration.

r_1, r_2 : random values in [0, 1].

4) ~~Hunting~~ (Exploitation).

$$\cancel{X(t+1) = \frac{X_\alpha + X_\beta + X_\delta}{3}}$$

A : controls the step size & direction of movement toward or away from the prey. When

Exploitation → $|A| < 1$, wolves converge on the prey (exploitation).

Exploration → when $|A| > 1$, wolves diverge to explore other regions.

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

Where $X_1 = X_\alpha - A_1 \cdot D_\alpha$

$X_2 = X_\beta - A_2 \cdot D$ $X_3 = X_\delta - A_3 \cdot D\alpha.$

Fitness function: (Sphere)

$f(x) = \sum_{i=1}^{d} x_i^2$ $x \rightarrow$ position vect, d — dimension.

PAGE NO:
DATE:

O/P.

~~Iteration 1/100, Alpha Score: 8.0.217183~~

~~Iteration 2/10~~

Algo:-

1. Set initial values of the population size n, parameter a, coefficient vectors A & C & max iterations.
2. Set t = 0
3. for (i = 1; i ≤ n) do
4. Generate an initial population $X_i(t)$ randomly.
5. Evaluate the fitness function of each search agent solution $f(x)$
6. end for.
7. Assign the values of the first, second & the third best solution X_α, X_β & X_δ.
8. Repeat
   for(i=1; i ≤ n) do.
   Update each search population based
   Decrease the parameter a from 2 to 0.
   Update the coefficient A & C.
   Evaluate the fitness function of each search agent $f(x_i)$.
   end for.
9. Update the vectors X_α, X_β & X_δ
10. Set t = t+1.
11. Until (t<Max_itr)
12. Produce the best solⁿ.

O/P :-

It 1/100, Alpha Score: 8.02

:

It 100/100, Alpha Score: 3.96.

Best position [-4.5054, 4.3936]

Best Score: 3.96027e-63.

20/4/24

CODE:

```
import numpy as np

# Objective function (e.g., Sphere function)
def objective_function(position):
    return sum(x**2 for x in position)

# Grey Wolf Optimizer
def grey_wolf_optimizer(obj_function, dim, pop_size, max_iter, bounds=(-10, 10)):
    a = 2  # Coefficient, decreases linearly from 2 to 0
    alpha_position = np.zeros(dim)
    alpha_score = float('inf')  # Best fitness (alpha)
    beta_position = np.zeros(dim)
    beta_score = float('inf')  # Second-best fitness (beta)
    delta_position = np.zeros(dim)
    delta_score = float('inf')  # Third-best fitness (delta)

    # Initialize the positions of the wolves
    wolves = np.random.uniform(bounds[0], bounds[1], (pop_size, dim))

    for iteration in range(max_iter):
        for i, wolf in enumerate(wolves):
            fitness = obj_function(wolf)

            # Update alpha, beta, and delta
            if fitness < alpha_score:
                delta_position = beta_position.copy()
                delta_score = beta_score
                beta_position = alpha_position.copy()
                beta_score = alpha_score
                alpha_position = wolf.copy()
                alpha_score = fitness
            elif fitness < beta_score:
                delta_position = beta_position.copy()
                delta_score = beta_score
                beta_position = wolf.copy()
                beta_score = fitness
            elif fitness < delta_score:
                delta_position = wolf.copy()
                delta_score = fitness

        # Update positions
```

```python
        for i, wolf in enumerate(wolves):
            r1, r2 = np.random.rand(dim), np.random.rand(dim)
            A1 = 2 * a * r1 - a
            C1 = 2 * r2
            D_alpha = abs(C1 * alpha_position - wolf)
            X1 = alpha_position - A1 * D_alpha

            r1, r2 = np.random.rand(dim), np.random.rand(dim)
            A2 = 2 * a * r1 - a
            C2 = 2 * r2
            D_beta = abs(C2 * beta_position - wolf)
            X2 = beta_position - A2 * D_beta

            r1, r2 = np.random.rand(dim), np.random.rand(dim)
            A3 = 2 * a * r1 - a
            C3 = 2 * r2
            D_delta = abs(C3 * delta_position - wolf)
            X3 = delta_position - A3 * D_delta

            wolves[i] = (X1 + X2 + X3) / 3

        # Linearly decrease a
        a -= 2 / max_iter

        print(f"Iteration {iteration+1}/{max_iter}, Alpha Score: {alpha_score}")

    return alpha_position, alpha_score

# Example usage
best_position, best_score = grey_wolf_optimizer(objective_function, dim=2,
pop_size=30, max_iter=100)
print("Best Position:", best_position)
print("Best Score:", best_score)
```

OUTPUT:

```
Iteration 90/100, Alpha Score: 3.580478177703101e-60
Iteration 91/100, Alpha Score: 3.102988105420075e-60
Iteration 92/100, Alpha Score: 2.936828514858608e-60
Iteration 93/100, Alpha Score: 2.671008236898743e-60
Iteration 94/100, Alpha Score: 2.4811269749912955e-60
Iteration 95/100, Alpha Score: 2.3383305537762118e-60
Iteration 96/100, Alpha Score: 2.206454382871867e-60
Iteration 97/100, Alpha Score: 2.1121148046984019e-60
Iteration 98/100, Alpha Score: 2.0185177719072882e-60
Iteration 99/100, Alpha Score: 1.9403778098441208e-60
Iteration 100/100, Alpha Score: 1.9173501698915915e-60
Best Position: [9.19241999e-31 1.03554059e-30]
Best Score: 1.9173501698915915e-60
```

<u>Program 6</u>

<u>Parallel Cellular Algorithms and Programs:</u>

Problem Statement:

Parallel Cellular Algorithms are inspired by the functioning of biological cells that operate in a highly parallel and distributed manner. These algorithms leverage the principles of cellular automata and parallel computing to solve complex optimization problems efficiently. Each cell represents a potential solution and interacts with its neighbors to update its state based on predefined rules. This interaction models the diffusion of information across the cellular grid, enabling the algorithm to explore the search space effectively. Parallel Cellular Algorithms are particularly suitable for large-scale optimization problems and can be implemented on parallel computing architectures for enhanced performance.

Algorithm:

Parallel Cellular Algorithm

- It is a distributed nature inspired optimization method based on Cellular automata.
- It uses a grid of cells each representing a potential solution with cells updating simultaneously and interacting with neighbors
- Governed by ~~pre~~ predefined rules, it balances local & global cooperation. The algorithm excels in large scale complex optimization task leveraging parallelism & scalability.

Application.

i. Mathematical optimization ii Image processing

iii. Bioinformatics iv. Cryptography

Objective function:

$$f(x) = x^2 - 4x + 4.$$

this is function represents the problem to be optimized.

each cell uses this to update its state then is evaluated against the fitness function itself.

Algorithm

① Define Objective function: $f(x) = x^2 - 4x + 4$

② Initialize grid: 2D grid, random values in the search space

③ Evaluate fitness :- Compute the fitness of each cell using the objective function

④ Update States: based on the ~~objective~~ average of its neighbors each cells state is updated.

⑤ Repeat step ③ & ④ till predefined or till convergence

⑥ Output best solution: return the cell

Detailed Steps

1. Objective function definition

$$f(x) = x^2 - 4x + 4.$$

$x \rightarrow$ Current state of the cell

$f(x) \rightarrow$ fitness value of the cell

It is used to optimize max or min by changing each cell fitness after comparing with its neighbors.

2. Initialize the grid

$$X_{i,j} \in [X_{min}, X_{max}] \rightarrow \text{each cell location } i, j$$

3. formula used

$$fitness_{i,j} = f(x_{i,j})$$

$x_{i,j} \rightarrow$ Current State of the cell

$Fitness_{i,j}$: fitness value of the cell.

4. UpdateStates: (Neighborhood average)

$$x_{ij}^{new} = \frac{\sum \text{neighbors } x_{i'j'}}{\text{no. of neighbors}}$$

x_{ij}^{new}: updated state of the cell ij
$x_{i'j'}$: State of the neighboring cell.
no of neighbors: total no. of neighbors for each cells.

5. Stopping Criteria

$\Delta x < \epsilon$ or iteration count $\geq$ max iter.
Δx: Change in states between iteration
ϵ: Converged threshold

6. Output the best solution

$$x^* \text{ argmin}_{i,j} f(x_{ij})$$

x^* optimal Solution $f(x_{ij})$:- fitness values of all cells.

Output:
Best Value: 0.12646
Best fitness: 3.5101048
minimized grid size = 10
Search range = (-10, 10)
iteration = 100

CODE:

```
import numpy as np

# Define the objective function
def objective_function(x):
    return x**2 - 4*x + 4

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

# Compute fitness for the grid
def evaluate_fitness(grid, objective_function):
    return objective_function(grid)

# Update the grid based on neighborhood average
def update_grid(grid):
    new_grid = np.copy(grid)
    for i in range(grid.shape[0]):
        for j in range(grid.shape[1]):
            # Get neighbors' values
            neighbors = []
            for di in [-1, 0, 1]:
                for dj in [-1, 0, 1]:
                    ni, nj = i + di, j + dj
                    if 0 <= ni < grid.shape[0] and 0 <= nj < grid.shape[1]:
                        neighbors.append(grid[ni, nj])
            # Update state to the average of neighbors
            new_grid[i, j] = np.mean(neighbors)
    return new_grid

# Main function to run the algorithm
def parallel_cellular_algorithm(grid_size, search_range, iterations):
    grid = initialize_grid(grid_size, search_range)  # Step 2: Initialize grid
    for _ in range(iterations):
        fitness = evaluate_fitness(grid, objective_function)  # Step 3:
Evaluate fitness
        grid = update_grid(grid)  # Step 4: Update states
    # Find the best solution
```

```python
    best_value = grid[np.unravel_index(np.argmin(fitness), fitness.shape)]
    return best_value, objective_function(best_value)

# Parameters
grid_size = 10  # 10x10 grid
search_range = (-10, 10)  # Search range for cell values
iterations = 100  # Number of iterations

# Run the algorithm
best_value, best_fitness = parallel_cellular_algorithm(grid_size,
search_range, iterations)

# Output the results
print(f"Best Value: {best_value}")
print(f"Best Fitness: {best_fitness}")
```

OUTPUT:

```
Best Value: -0.7769646689183809
Best Fitness: 7.711532772420973
```

Program 7

Optimization via Gene Expression Algorithms:

Problem Statement:

Gene Expression Algorithms (GEA) are inspired by the biological process of gene expression in living organisms. This process involves the translation of genetic information encoded in DNA into functional proteins. In GEA, solutions to optimization problems are encoded in a manner similar to genetic sequences. The algorithm evolves these solutions through selection, crossover, mutation, and gene expression to find optimal or near-optimal solutions. GEA is effective for solving complex optimization problems in various domains, including engineering, data analysis, and machine learning.

Algorithm:

Optimization via Gene Expression Algorithm

It is inspired by the biological process of gene expression where genetic information is translated into functional proteins. GEAs model optimization problems by encoding solution as genetic sequences and evolve these sequences using genetic operators like selection cross over & mutation.

These algorithm are highly effective in solving complex optimization

Application → Automatic program generation ML eng design data analysis

Shortened Algorithm

1. Define the Rastrigin function

2. Initialize population: generate random individual within the search range

3. Evaluate fitness: Compute the rastrigin function value. ~~fort~~ for each individual.

4. Selection :- Use tournament selection to choose individuals based on fitness

Cross over: Combine gene of 2 parent to create offspring

mutation :- introduce random mutation to maintain diversity

Repeat & iterate until the stopping criterion (no of generation are hit)

Output:- Return the individual with the least fitness

Key Component

* Objective function:

$$f(x) = A \cdot N + \sum_{i=1}^{n} (x_i^2 - A \cdot \cos(2\pi x_i))$$

$n \to$ no of dimensions
x_i = gene of an individual.

* State space:- Set of all possible solution within the search range $(-5.12, 5.12]$ for each gene

* Fitness f^n:- -ve value of rastrigin f^n is used to guide optimization. lower fitness value indicate better solution

* Stopping Criteria:-
fixed no of generation (100 in ex)
Optional:- Convergence of fitness value.

* formulas & variables:-
Rastrigin function

$$f(x) = 10n + \sum_{i=1}^{n} (x_i^2 - 10\cos(2\pi x_i))$$

x_i gene values.

* Crossover (uniform):-
Child[i] = parent1[i] if random()<0.5
else parent2[i].
. x_i = random value is $(-5.12, 5.12]$ with probability mutation rate.

Best Soln: [0.988365 1.0378591 0.029949
0.93804573 -2.04281649
-1.12168452 -1.0219388 0.0277796
-1.0839781 -1.05373]

Best Fitness :- 18.2419393160531 5

16/12/24

CODE:

```
import numpy as np
import random

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

# Initialize population
def initialize_population(pop_size, gene_length, search_range):
    return [np.random.uniform(search_range[0], search_range[1], gene_length) for _ in
range(pop_size)]

# Evaluate fitness
def evaluate_fitness(population):
    return [rastrigin_function(ind) for ind in population]

# Selection (tournament selection)
def selection(population, fitness):
    selected = []
    for _ in range(len(population)):
        i, j = random.sample(range(len(population)), 2)
        selected.append(population[i] if fitness[i] < fitness[j] else population[j])
    return selected
```

```
# Crossover (uniform crossover)
def crossover(parent1, parent2):
    child = []
    for p1, p2 in zip(parent1, parent2):
        child.append(p1 if random.random() < 0.5 else p2)
    return np.array(child)

# Mutation (random mutation)
def mutate(individual, mutation_rate, search_range):
    for i in range(len(individual)):
        if random.random() < mutation_rate:
            individual[i] = random.uniform(search_range[0], search_range[1])
    return individual

# Gene Expression Algorithm
def gene_expression_algorithm(pop_size, gene_length, generations, mutation_rate,
search_range):
    # Step 1: Initialize population
    population = initialize_population(pop_size, gene_length, search_range)

    for generation in range(generations):
        # Step 2: Evaluate fitness
        fitness = evaluate_fitness(population)

        # Step 3: Selection
        selected_population = selection(population, fitness)

        # Step 4: Crossover and Mutation
        next_population = []
        for i in range(0, len(selected_population), 2):
            if i + 1 < len(selected_population):
                # Crossover
                child1 = crossover(selected_population[i], selected_population[i +
1])
                child2 = crossover(selected_population[i + 1],
selected_population[i])
            else:
                child1 = selected_population[i]
                child2 = selected_population[i]
            # Mutation
            next_population.append(mutate(child1, mutation_rate, search_range))
```

```python
            next_population.append(mutate(child2, mutation_rate, search_range))

        population = next_population[:pop_size]  # Maintain population size

    # Final fitness evaluation
    fitness = evaluate_fitness(population)
    best_individual = population[np.argmin(fitness)]
    return best_individual, rastrigin_function(best_individual)

# Parameters
pop_size = 50
gene_length = 10
generations = 100
mutation_rate = 0.1
search_range = (-5.12, 5.12)  # Search range for Rastrigin function

# Run the algorithm
best_solution, best_fitness = gene_expression_algorithm(pop_size, gene_length,
generations, mutation_rate, search_range)

print(f"Best Solution: {best_solution}")
print(f"Best Fitness: {best_fitness}")
```

OUTPUT:

```
Best Solution: [-1.01547995  0.07075383 -0.02144954 -1.02279118  1.05152707 -2.00221714
 -0.28127705 -1.02468392 -0.95537354 -0.99059453]
Best Fitness: 24.43384866081925
```