

VISVESVARAYA TECHNOLOGICAL UNIVERSITY

“JnanaSangama”, Belgaum -590014, Karnataka.



LAB RECORD

Bio Inspired Systems (23CS5BSBIS)

Submitted by

Shreyas K(1BM22CS271)

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 **Shreyas K(1BM22CS271)**, 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 mention subject and the work prescribed for the said degree.

Spoorthi D M Assistant Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE
---	--

Index

Sl. No.	Date	Experiment Title	Page No.
1	24-10-24	Genetic Algorithm for Optimization Problems	1
2	7-11-24	Particle Swarm Optimization for Function Optimization	4
3	14-11-24	Ant Colony Optimization for the Traveling Salesman Problem	7
4	21-11-24	Cuckoo Search (CS)	12
5	28-11-24	Grey Wolf Optimizer (GWO)	15
6	5-12-24	Parallel Cellular Algorithms and Programs	19
7	12-12-24	Optimization via Gene Expression Algorithms	21

Github Link:

<https://github.com/Shreyess7/BIS-LAB>

Program 1

Genetic Algorithm for Optimization Problems

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.

Implementation Steps:

1. Define the Problem: Create a mathematical function to optimize.
2. Initialize Parameters: Set the population size, mutation rate, crossover rate, and number of generations.
3. Create Initial Population: Generate an initial population of potential solutions.
4. Evaluate Fitness: Evaluate the fitness of each individual in the population.
5. Selection: Select individuals based on their fitness to reproduce.
6. Crossover: Perform crossover between selected individuals to produce offspring.
7. Mutation: Apply mutation to the offspring to maintain genetic diversity.
8. Iteration: Repeat the evaluation, selection, crossover, and mutation processes for a fixed number of generations or until convergence criteria are met.
9. Output the Best Solution: Track and output the best solution found during the generations.

Algorithm:

Genetic Algorithm for Optimization

Steps -

- 1) Initialization - Randomly initialize a population of candidate solutions
- 2) Selection - Select individuals based on their fitness
- 3) Crossover - combine pairs of individuals to form new offspring
- 4) Mutation - Randomly tweak offspring to maintain diversity in the population
- 5) Replacement - Replace part of the old population with the new offspring
- 6) Termination - Repeat the process until a stopping condition is met

Pseudocode

```
Function GeneticAlgo // initialization(parameters)
SET population_size = 100
SET mutation_rate = 0.01
SET num_generations = 50
SET x_bound = (-10, 10)
```

// Initialization
population = Randomly_Generate_population_size_within_x_bounds

For generation from 1 to num_generations
 fitness_values = Evaluate(population)
 // fitness evaluation

// selection
selected_parents = select(population, fitness_values)

Crossover and Mutation

```
new_population = []
for each pair of selected_parents
    (child1, child2) = crossover(parent1, parent2)
    child1 = mutate(child1)
    child2 = mutate(child2)
    Append child1 and child2 to new_population
```

// Replacement
population = new_population

// Return solution
Return solution(population)

Main point

- 1) Initialize a population of candidate solutions
- 2) Evaluate their fitness
- 3) Select the fittest individuals to create new offspring
- 4) Apply crossover & mutation to produce the next generation
- 5) Repeat the process for a specified no. of generations
- 6) Output - best solution

write all steps completely

CP - Best soln - $x = -9.99157$
Best fitness = 99.8275

24/07/24

Code:

```
#GENETIC ALGORITHM
```

```
import numpy as np
```

```
import random
```

```
# Define the fitness function
```

```
def fitness_function(x):
```

```
    return x ** 2
```

```
# Initialize parameters
```

```
population_size = 100
```

```
mutation_rate = 0.1
```

```
num_generations = 50
```

```
bounds = (-10, 10)
```

```
# Step 1: Create initial population
```

```
def create_initial_population(size, bounds):
```

```
    return [random.uniform(bounds[0], bounds[1]) for _ in range(size)]
```

```
# Step 2: Evaluate fitness of the population
```

```
def evaluate_population(population):
```

```
    return [fitness_function(individual) for individual in population]
```

```
# Step 3: Selection using roulette-wheel selection
```

```
def selection(population, fitness):
```

```
    total_fitness = sum(fitness)
```

```
    selection_probs = [f / total_fitness for f in fitness]
```

```
    return np.random.choice(population, size=2, p=selection_probs)
```

```
# Step 4: Crossover operation
```

```
def crossover(parent1, parent2):
```

```
    alpha = random.uniform(0, 1)
```

```
    offspring1 = alpha * parent1 + (1 - alpha) * parent2
```

```
    offspring2 = alpha * parent2 + (1 - alpha) * parent1
```

```
    return offspring1, offspring2
```

```
# Step 5: Mutation operation
```

```
def mutate(individual, bounds):
```

```
    if random.random() < mutation_rate:
```

```
        return random.uniform(bounds[0], bounds[1])
```

```
    return individual
```

```
# Main Genetic Algorithm loop
```

```
def genetic_algorithm(bounds):
```

```
    # Step 1: Create initial population
```

```
population = create_initial_population(population_size, bounds)
```

```
best_solution = None best_fitness
```

```
= float('-inf')
```

```
for generation in range(num_generations): # Step 2: Evaluate fitness
```

```
    fitness = evaluate_population(population)
```

```
    # Track the best solution
```

```
    current_best_fitness = max(fitness)
```

```
    if current_best_fitness > best_fitness:
```

```
        best_fitness = current_best_fitness
```

```
        best_solution = population[fitness.index(current_best_fitness)]
```

```
    # Step 3: Create new population
```

```
    new_population = []
```

```
    while len(new_population) < population_size:
```

```
        parent1, parent2 = selection(population, fitness)
```

```
        offspring1, offspring2 = crossover(parent1, parent2)
```

```
        new_population.append(mutate(offspring1, bounds))
```

```
        new_population.append(mutate(offspring2, bounds))
```

```
    # Replace the old population with the new population
```

```
    population = new_population[:population_size]
```

```
    return best_solution, best_fitness
```

```
# Run the Genetic Algorithm
```

```
best_solution, best_fitness = genetic_algorithm(bounds)
```

```
print(f"Best Solution: x = {best_solution}")
```

```
print(f"Best Fitness: f(x) = {best_fitness}")
```

Output:

```
Best Solution: x = 9.97704555295002
Best Fitness: f(x) = 99.54143796563977
```

Program 2

Particle Swarm Optimization for Function Optimization

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.

Implementation Steps:

1. Define the Problem: Create a mathematical function to optimize.
2. Initialize Parameters: Set the number of particles, inertia weight, cognitive and social coefficients.
3. Initialize Particles: Generate an initial population of particles with random positions and velocities.
4. Evaluate Fitness: Evaluate the fitness of each particle based on the optimization function.
5. Update Velocities and Positions: Update the velocity and position of each particle based on its own best position and the global best position.
6. Iterate: Repeat the evaluation, updating, and position adjustment for a fixed number of iterations or until convergence criteria are met.
7. Output the Best Solution: Track and output the best solution found during the iterations.

Algorithm:

Particle swarm Optimization

PSO algo inspired by the social behavior of birds flocking or fish schooling.

It is used to find approximate solution to complex optimization prob by simulating the movement of particles through a multidimensional search space.

Algo

- 1) define function which is needed to optimize.
- 2) Initialization of swarm
 - a) Initialize parameters:
 - P \rightarrow no. of particles
 - D \rightarrow no. of dimension
 - T \rightarrow no. of iterations
 - $c_1 \rightarrow$ cognitive coefficient (own influence / pbest)
 - $c_2 \rightarrow$ social coeff (social influence / gbest)
 - $w \rightarrow$ inertia weight (controls momentum)
 - b) initialize each particle:
 - particle position \rightarrow randomly initialize
 - particle velocity \rightarrow randomly initialize in range
 - c) Determine global best:
 - gbest - particle with best fitness
 - set gbest to the position of the particles with best fitness.
- 3) in loop from 1 to T
 - a) update velocity: generate r_1, r_2
 - $C = c_1 * r_1 * (pbest - x_i)$
 - $S = c_2 * r_2 * (gbest - x_i)$
 - $v_i = w * v_i + C + S$
 - b) update pos: $x_i = x_i + v_i$

calculate fitness of new pos x_i
update Pbest if new fitness is greater & also gbest

\rightarrow find the optimal solution

Summary

- 1) PSO is useful for solving complex optimization problems.
- 2) It works by initializing a group of particles with random pos & velocity.
- 3) Each particle evaluates its own position & tracks pbest.
- 4) Swarm tracks \rightarrow gbest.
- 5) particles update the pos based on Pbest & gbest.
- 6) Influenced by inertia (momentum) & cognitive (individual leaning) coeff.
- 7) Over multiple iterations the swarm converges towards the optimal solution.

write pseudocode properly of this algorithm

Op - Best pos = [1.9406, -1.1133]
Best fitness = 5.0855

Code:

#PARTICLE SWARM OPTIMIZATION

import numpy as np

Rastrigin function: A benchmark function for optimization problems

def rastrigin(x):

 A = 10

 # Calculate the Rastrigin function value based on the input vector x

 return A * len(x) + sum(x_i**2 - A * np.cos(2 * np.pi * x_i) for x_i in x)

Particle Swarm Optimization class

class ParticleSwarmOptimizer:

 def __init__(self, func, n_particles, n_dimensions, n_iterations, inertia_weight=0.7,
cognitive_coeff=1.5, social_coeff=1.5, bounds=(-5.12, 5.12)):

 self.func = func # The function to optimize

 self.n_particles = n_particles # Number of particles in the swarm

 self.n_dimensions = n_dimensions # Dimensions of the search space

 self.n_iterations = n_iterations # Number of iterations for the optimization

 self.lower_bound, self.upper_bound = bounds # Bounds for the search space

 # Initialize particle positions randomly within the specified bounds

 self.positions = np.random.uniform(self.lower_bound, self.upper_bound, (n_particles,
n_dimensions))

 # Initialize particle velocities randomly

 self.velocities = np.random.uniform(-1, 1, (n_particles, n_dimensions))

 # Personal best positions and scores for each particle

 self.pbest_positions = np.copy(self.positions)

 self.pbest_scores = np.array([func(p) for p in self.positions]) # Evaluate initial fitness

 # Global best position and score among all particles

 self.gbest_position = self.pbest_positions[np.argmin(self.pbest_scores)]

 self.gbest_score = np.min(self.pbest_scores)

 def optimize(self):

 # Main loop for the optimization process

 for _ in range(self.n_iterations):

 for i in range(self.n_particles):

 # Evaluate the fitness of the current position

 fitness = self.func(self.positions[i])

 # Update personal best if the current fitness is better

 if fitness < self.pbest_scores[i]:

 self.pbest_scores[i] = fitness

 self.pbest_positions[i] = self.positions[i]

 # Update global best if the current fitness is better

 if fitness < self.gbest_score:


```

self.gbest_score = fitness
self.gbest_position = self.positions[i]

# Generate random coefficients for cognitive and social components
r1, r2 = np.random.rand(self.n_dimensions),
np.random.rand(self.n_dimensions) # Update velocities based on inertia,
personal best, and global best self.velocities = (self.velocities * 0.7 + # Inertia
weight
1.5 * r1 * (self.pbest_positions - self.positions) + # Cognitive component
1.5 * r2 * (self.gbest_position - self.positions)) # Social component
# Update positions based on new velocities and clip to stay within bounds
self.positions = np.clip(self.positions + self.velocities, self.lower_bound, self.upper_bound)

# Print the best fitness found so far in this iteration
print(f"Best Fitness: {self.gbest_score}")

# Return the best position and score found after all iterations
return self.gbest_position, self.gbest_score

# Create and run the optimizer
pso = ParticleSwarmOptimizer(func=rastrigin, n_particles=30, n_dimensions=2, n_iterations=100)
best_position, best_score = pso.optimize()

# Print the best position and corresponding fitness score found
print("\nBest Position Found:", best_position)
print("Best Fitness Score:", best_score)

```

Output:

```

Best Fitness: 0.00042484148907107055
Best Fitness: 0.00042484148907107055
Best Fitness: 0.00019896058490331825
Best Fitness: 0.00019896058490331825
Best Fitness: 0.00019896058490331825
Best Fitness: 0.00019896058490331825
Best Fitness: 0.00019896058490331825
Best Fitness: 0.00019896058490331825
Best Fitness: 9.185587018123442e-06
Best Fitness: 9.185587018123442e-06
Best Fitness: 9.185587018123442e-06
Best Fitness: 9.185587018123442e-06
Best Fitness: 9.185587018123442e-06
Best Fitness: 9.185587018123442e-06
Best Fitness: 8.17354336390963e-06
Best Fitness: 8.17354336390963e-06
Best Fitness: 3.993851240835511e-06
Best Fitness: 3.993851240835511e-06
Best Fitness: 3.993851240835511e-06
Best Fitness: 3.993851240835511e-06
Best Fitness: 1.6462023069152565e-06
Best Fitness: 1.6462023069152565e-06
Best Fitness: 1.6462023069152565e-06
Best Fitness: 1.6462023069152565e-06
Best Fitness: 1.6462023069152565e-06
Best Fitness: 1.6462023069152565e-06
Best Fitness: 1.6462023069152565e-06
Best Fitness: 1.5764355048020207e-06
Best Position Found: [3.76308963e-05 8.08082678e-05]
Best Fitness Score: 1.5764355048020207e-06

```

Program 3

Ant Colony Optimization for the Traveling Salesman Problem

The foraging behaviour 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.

Implementation Steps:

1. Define the Problem: Create a set of cities with their coordinates.
2. Initialize Parameters: Set the number of ants, the importance of pheromone (alpha), the importance of heuristic information (beta), the evaporation rate (rho), and the initial pheromone value.
3. Construct Solutions: Each ant constructs a solution by probabilistically choosing the next city based on pheromone trails and heuristic information.
4. Update Pheromones: After all ants have constructed their solutions, update the pheromone trails based on the quality of the solutions found.
5. Iterate: Repeat the construction and updating process for a fixed number of iterations or until convergence criteria are met.
6. Output the Best Solution: Keep track of and output the best solution found during the iterations.

Algorithm:

LAB-3
Ant Colony Optimization for Travelling Salesman Problem

Algo

- 1) Represent the cities as nodes in a graph & construct a distance matrix.
- 2) Initialize parameters:
 - $m \rightarrow$ no of ants
 - $\alpha \rightarrow$ importance of pheromone
 - $\beta \rightarrow$ heuristic information
 - $\rho \rightarrow$ pheromone evaporation rate
 - $Q \rightarrow$ deposit constant
- 3) for each ant:
 - a) Randomly select start city
 - b) Build a complete tour by iteratively selecting next city
 - ↳ for each unvisited city j calculate probability of moving from i to j
 - c) Use the probability to select next city
 - d) Add selected city to tour & mark as visited
- 4) Calculate total length of each tour & keep track of best tour found so far.
- 5) Update pheromones:
 - pheromones $\times = (1 - \rho)$
 - pheromone increase $= Q / \text{tour-length}$

Pseudocode

- 1) Initialize parameters:
 - No. of ants
 - No. of iterations
 - Pheromone matrix with small initial values
 - α for pheromone influence
 - β for the heuristic influence
 - evaporation rate (ρ)
- 2) Initialize pheromone matrix & distance matrix
 - set initial & compute dist.
- 3) Repeat for each iteration
 - i) Place the ants at random city
 - ii) Construct a complete tour
 - calculate probabilities
 - calculate the length of tour
 - b) update pheromones
 - evaporate pheromones
 - reinforce pheromones
- 4) Return the best solution

Op - Best route - {3, 1, 0, 2, 4, 3}
Best dist - 22.50

Code:

```
#ANT COLONY OPTIMIZATION
```

```
import random
```

```
import numpy as np
```

```
# Distance calculation (Euclidean distance)
```

```
def euclidean_distance(city1, city2):
```

```
    return np.sqrt((city1[0] - city2[0])**2 + (city1[1] - city2[1])**2)
```

```
# Ant Colony Optimization Algorithm
```

```
class ACO:
```

```
    def __init__(self, cities, num_ants=10, num_iterations=100, alpha=1.0, beta=2.0, rho=0.5, Q=100):
```

```
        self.cities = cities
```

```
        self.num_ants = num_ants
```

```
        self.num_iterations = num_iterations
```

```
        self.alpha = alpha      # Importance of pheromone
```

```
        self.beta = beta        # Importance of heuristic information (distance)
```

```
        self.rho = rho          # Pheromone evaporation rate
```

```
        self.Q = Q              # Total pheromone deposited per ant per tour
```

```
        self.num_cities = len(cities)
```

```
        # Initialize pheromone matrix (for each pair of cities)
```

```
        self.pheromone = np.ones((self.num_cities, self.num_cities)) / self.num_cities
```

```
        self.distances = np.zeros((self.num_cities, self.num_cities))
```

```
        # Compute distance matrix
```

```
        for i in range(self.num_cities):
```

```
            for j in range(i + 1, self.num_cities):
```

```
                self.distances[i][j] = self.distances[j][i] = euclidean_distance(cities[i], cities[j])
```

```
    def _choose_next_city(self, ant, visited):
```

```
        # Calculate the probability of moving to each city
```

```
        current_city = ant[-1]
```

```
        probabilities = []
```

```
        for i in range(self.num_cities):
```

```
            if i not in visited:
```

```
                pheromone = self.pheromone[current_city][i] ** self.alpha
```

```
                heuristic = (1.0 / self.distances[current_city][i]) ** self.beta
```

```
                probabilities.append(pheromone * heuristic)
```

```
            else:
```

```
                probabilities.append(0)
```

```
        # Normalize probabilities
```

```

total = sum(probabilities)
if total == 0: # In case there's no valid path (shouldn't happen with good settings)
    return random.choice([i for i in range(self.num_cities) if i not in visited])

probabilities = [prob / total for prob in probabilities]

# Choose next city based on probabilities
next_city = random.choices(range(self.num_cities), probabilities)[0]
return next_city

def _construct_solution(self):
    # Each ant starts at a random city
    ant = [random.randint(0, self.num_cities - 1)]
    visited = set(ant)

    while len(ant) < self.num_cities:
        next_city = self._choose_next_city(ant, visited)
        ant.append(next_city)
        visited.add(next_city)

    # Return to the starting city
    ant.append(ant[0])

    return ant

def _evaluate_solution(self, solution):
    # Calculate the total distance of the tour
    total_distance = 0
    for i in range(len(solution) - 1):
        total_distance += self.distances[solution[i]][solution[i + 1]]
    return total_distance

def _update_pheromone(self, all_solutions):
    # Initialize pheromone update matrix
    pheromone_delta = np.zeros((self.num_cities, self.num_cities))

    # For each solution, deposit pheromone
    for solution in all_solutions:
        tour_length = self._evaluate_solution(solution)
        for i in range(len(solution) - 1):
            pheromone_delta[solution[i]][solution[i + 1]] += self.Q / tour_length

    # Evaporate pheromone
    self.pheromone = (1 - self.rho) * self.pheromone + pheromone_delta

```

```

def solve(self):
    best_solution = None
    best_distance = float('inf')
    for iteration in range(self.num_iterations):
        all_solutions = []

        # Each ant constructs a solution
        for ant in range(self.num_ants):
            solution = self._construct_solution()
            all_solutions.append(solution)
            tour_length = self._evaluate_solution(solution)

        # Update best solution if necessary
        if tour_length < best_distance:
            best_solution = solution
            best_distance = tour_length

        # Update pheromones based on solutions found
        self._update_pheromone(all_solutions)

        print(f"Iteration {iteration + 1}, Best Distance: {best_distance}")

    return best_solution, best_distance

# Function to take user input for cities
def get_user_input():
    num_cities = int(input("Enter the number of cities: "))
    cities = []

    print("Enter the coordinates of each city (x, y):")
    for i in range(num_cities):
        x, y = map(float, input(f"City {i+1}: ").split())
        cities.append((x, y))

    return cities

# Example usage:
if __name__ == "__main__": #
    Take user input for cities
    cities = get_user_input()

    # Take user input for ACO parameters
    num_ants = int(input("Enter the number of ants: "))

```

```

num_iterations = int(input("Enter the number of iterations: "))
alpha = float(input("Enter the value of alpha (pheromone importance): "))
beta = float(input("Enter the value of beta (distance importance): "))
rho = float(input("Enter the value of rho (pheromone evaporation rate): "))
Q = float(input("Enter the value of Q (pheromone deposit per ant): "))

# Create an instance of ACO and solve the problem

aco = ACO(cities, num_ants, num_iterations, alpha, beta, rho, Q)
best_solution, best_distance = aco.solve()

print(f"\nBest Solution (Tour): {best_solution}")
print(f"Best Distance: {best_distance}")

```

Output:

```

Enter the number of cities: 5
Enter the coordinates of each city (x, y):
City 1: 0 0
City 2: 1 3
City 3: 4 3
City 4: 6 1
City 5: 3 0
Enter the number of ants: 10
Enter the number of iterations: 100
Enter the value of alpha (pheromone importance): 1.0
Enter the value of beta (distance importance): 2.0
Enter the value of rho (pheromone evaporation rate): 0.5
Enter the value of Q (pheromone deposit per ant): 100

Best Solution (Tour): [1, 0, 4, 3, 2, 1]
Best Distance: 15.15298244508295

```


Program 4

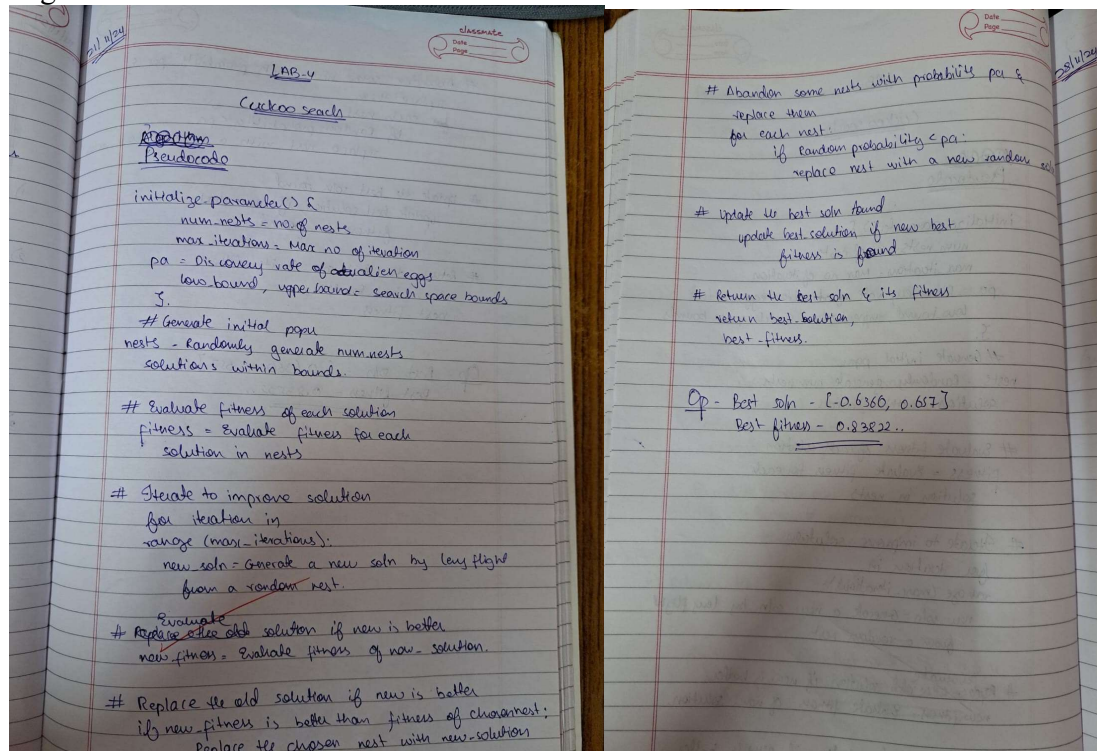
Cuckoo Search (CS)

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.

Implementation Steps:

1. Define the Problem: Create a mathematical function to optimize.
2. Initialize Parameters: Set the number of nests, the probability of discovery, and the number of iterations.
3. Initialize Population: Generate an initial population of nests with random positions.
4. Evaluate Fitness: Evaluate the fitness of each nest based on the optimization function.
5. Generate New Solutions: Create new solutions via Lévy flights.
6. Abandon Worst Nests: Abandon a fraction of the worst nests and replace them with new random positions.
7. Iterate: Repeat the evaluation, updating, and replacement process for a fixed number of iterations or until convergence criteria are met.
8. Output the Best Solution: Track and output the best solution found during the iterations.

Algorithm:



Code:

#CUCKOO SEARCH

```
import numpy as np
import math # Import the standard math module
def levy_flight(Lambda):
    sigma = (math.gamma(1 + Lambda) * math.sin(math.pi * Lambda / 2) /
              (math.gamma((1 + Lambda) / 2) * Lambda * 2 ** ((Lambda - 1) / 2))) ** (1 / Lambda)
    u = np.random.normal(0, sigma, 1)
    v = np.random.normal(0, 1, 1)
    step = u / abs(v) ** (1 / Lambda)
    return step
def cuckoo_search(obj_function, bounds, n=25, pa=0.25, max_iter=100):
    # Initialize nests
    dim = len(bounds)
    nests = np.random.rand(n, dim)
    for i in range(dim):
        nests[:, i] = nests[:, i] * (bounds[i][1] - bounds[i][0]) + bounds[i][0]
    fitness = np.array([obj_function(nest) for nest in nests])

    # Start optimization
    for _ in range(max_iter):
        for i in range(n):
            # Generate a new solution via Levy flight
            new_nest = nests[i] + levy_flight(1.5) * np.random.randn(dim)
            # Apply bounds
            new_nest = np.clip(new_nest, [b[0] for b in bounds], [b[1] for b in bounds])
            new_fitness = obj_function(new_nest)
            # Update if new solution is better
            if new_fitness < fitness[i]:
                nests[i] = new_nest
                fitness[i] = new_fitness

        # Abandon some nests and create new ones
        abandon_idx = np.random.rand(n) < pa
        for i in np.where(abandon_idx)[0]:
            nests[i] = np.random.rand(dim) * (np.array([b[1] for b in bounds]) - np.array([b[0] for b in
bounds])) + np.array([b[0] for b in bounds])
            fitness[i] = obj_function(nests[i])

    # Return the best solution
    best_idx = np.argmin(fitness)
    return nests[best_idx], fitness[best_idx]
```



```
# Example usage: Minimize  $f(x) = x^2$ 
def objective(x):
    return sum(xi**2 for xi in x)

bounds = [(-10, 10), (-10, 10)] # 2D problem
best_solution, best_fitness = cuckoo_search(objective, bounds)
print("Best Solution:", best_solution)
print("Best Fitness:", best_fitness)
```

Output:

```
Best Solution: [-0.14023741  0.59049343]
Best Fitness: 0.36834901994989167
```

Program 5

Grey Wolf Optimizer (GWO)

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.

Implementation Steps:

1. Define the Problem: Create a mathematical function to optimize.
2. Initialize Parameters: Set the number of wolves and the number of iterations.
3. Initialize Population: Generate an initial population of wolves with random positions.
4. Evaluate Fitness: Evaluate the fitness of each wolf based on the optimization function.
5. Update Positions: Update the positions of the wolves based on the positions of alpha, beta, and delta wolves.
6. Iterate: Repeat the evaluation and position updating process for a fixed number of iterations or until convergence criteria are met.
7. Output the Best Solution: Track and output the best solution found during the iterations.

Algorithm:

LAB-5
Grey Wolf Optimizer
Pseudocode

- 1) Initialize parameter:
 - population size
 - no. of iterations
 - low & upper bounds
 - Objective function to minimize
- 2) Randomly initialize the positions of grey wolves within the search space.
- 3) Evaluate the fitness of each wolf & identify:
 - Alpha (best soln)
 - Beta (second best soln)
 - Delta (third best soln)
- 4) Update the position of each wolf using the hunting mechanism:
 - compute the coefficient A & C :
 $A = 2a \times \text{random}() - a$
 $C = \text{rand}() \times \text{random}()$
 - calculate the position updates - α, β, δ
 $D_{\alpha} = |C_1 \times \text{Alpha-position} - \text{current-position}|$
 $X_1 = \text{Alpha-position} - A_1 \times D_{\alpha}$
 $\text{New-pos} = (X_1 + X_2 + X_3) / 3$
- 5) Clip the position to remain within bound
- 6) Evaluate the fitness of update pos & update Alpha, Beta & Delta accordingly.
- 7) Reduce the parameter 'a' linearly from 2 to 0 over iterations.
- 8) Repeat steps 4-7 until the stopping condition is met.
- 9) Return Alpha (best solution) & its fitness.

opt?
Optimal

$\text{Op - Optimal fitness} = 2.581673$
 $\text{Optimal soln} = [-1.08900, -15, -1.1814]$

Code:

```
#Grey Wolf Optimizer (GWO)
import numpy as np

def objective_function(x):
    """Example objective function: Sphere function."""
    return sum(x**2)

def initialize_population(dim, n_wolves, bounds):
    """Initialize the positions of the wolves randomly within the given bounds."""
    return np.random.uniform(bounds[0], bounds[1], (n_wolves, dim))

def gwo(objective_function, bounds, dim, n_wolves, n_iterations):

    # Initialize population

    wolves = initialize_population(dim, n_wolves, bounds)
    fitness = np.apply_along_axis(objective_function, 1, wolves)

    # Initialize alpha, beta, and delta
    alpha, beta, delta = np.argsort(fitness)[:3]
    alpha_pos, alpha_score = wolves[alpha], fitness[alpha]
    beta_pos, beta_score = wolves[beta], fitness[beta]
    delta_pos, delta_score = wolves[delta], fitness[delta]

    # Main optimization loop
    for iteration in range(n_iterations):
        a = 2 - 2 * (iteration / n_iterations) # Linearly decreasing a

        for i in range(n_wolves):
            for j in range(dim):
```

```

# Update each wolf's position
r1, r2 = np.random.rand(), np.random.rand()
A1, C1 = 2 * a * r1 - a, 2 * r2
D_alpha = abs(C1 * alpha_pos[j] - wolves[i, j])
X1 = alpha_pos[j] - A1 * D_alpha

r1, r2 = np.random.rand(), np.random.rand()
A2, C2 = 2 * a * r1 - a, 2 * r2
D_beta = abs(C2 * beta_pos[j] - wolves[i, j])
X2 = beta_pos[j] - A2 * D_beta

r1, r2 = np.random.rand(), np.random.rand()
A3, C3 = 2 * a * r1 - a, 2 * r2
D_delta = abs(C3 * delta_pos[j] - wolves[i, j])
X3 = delta_pos[j] - A3 * D_delta

# Average position update
wolves[i, j] = (X1 + X2 + X3) / 3.0

# Enforce bounds
wolves[i, :] = np.clip(wolves[i, :], bounds[0], bounds[1])

# Evaluate fitness and update alpha, beta, delta
fitness = np.apply_along_axis(objective_function, 1, wolves)
sorted_indices = np.argsort(fitness)
alpha, beta, delta = sorted_indices[:3]
alpha_pos, alpha_score = wolves[alpha], fitness[alpha]
beta_pos, beta_score = wolves[beta], fitness[beta]
delta_pos, delta_score = wolves[delta], fitness[delta]
return alpha_pos, alpha_score

# Example usage

dim = 5 # Number of dimensions
bounds = (-10, 10) # Search space bounds
n_wolves = 30 # Number of wolves
n_iterations = 100 # Number of iterations

best_solution, best_score = gwo(objective_function, bounds, dim, n_wolves, n_iterations)
print(f"Best solution: {best_solution}")
print(f"Best score: {best_score}")

```

Output:

```
Best solution: [-1.48263895e-11 -1.24732979e-11  1.51277899e-11  1.54330567e-11  
1.16834722e-11]  
Best score: 9.78937775690888e-22
```

Program 6

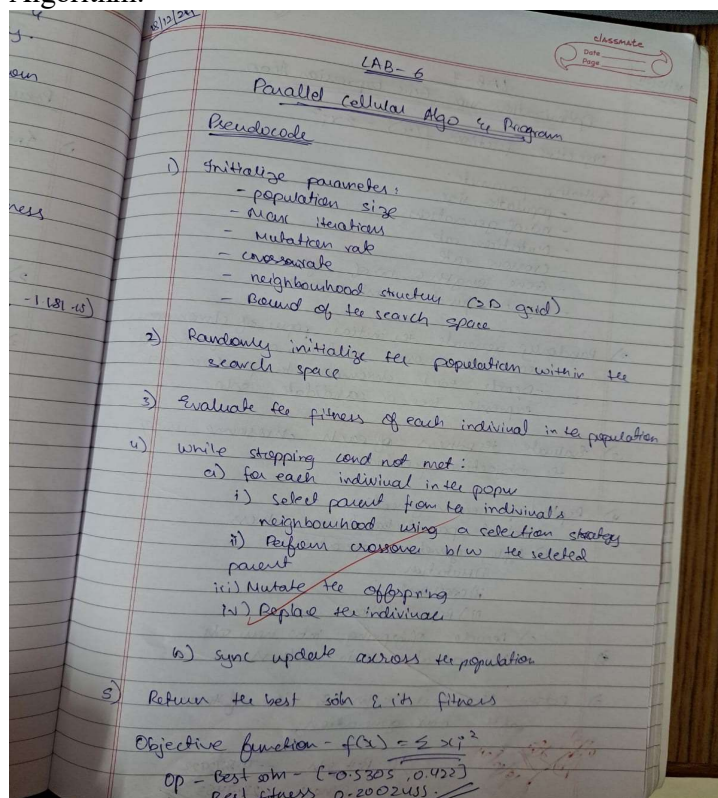
Parallel Cellular Algorithms and Programs

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.

Implementation Steps:

1. Define the Problem: Create a mathematical function to optimize.
2. Initialize Parameters: Set the number of cells, grid size, neighborhood structure, and number of iterations.
3. Initialize Population: Generate an initial population of cells with random positions in the solution space.
4. Evaluate Fitness: Evaluate the fitness of each cell based on the optimization function.
5. Update States: Update the state of each cell based on the states of its neighboring cells and predefined update rules.
6. Iterate: Repeat the evaluation and state updating process for a fixed number of iterations or until convergence criteria are met.
7. Output the Best Solution: Track and output the best solution found during the iterations.

Algorithm:



Code:

```
#Parallel Cellular Algorithms and Programs
import numpy as np

# Define the optimization function
def fitness_function(x):
    return x**2

# Initialize parameters
num_cells = 10
grid_size = 1.0
iterations = 100
neighborhood_size = 1

# Initialize population
cells = np.random.uniform(-grid_size, grid_size, num_cells)

# Main loop
for _ in range(iterations):
    # Evaluate fitness
    fitness = np.array([fitness_function(cell) for cell in cells])

    # Update states
    new_cells = np.copy(cells)
    for i in range(num_cells):
        # Get neighbors
        neighbors = cells[max(0, i-neighborhood_size):min(num_cells, i+neighborhood_size+1)]
        # Update cell based on neighbors
        new_cells[i] = np.mean(neighbors) + np.random.uniform(-0.1, 0.1) # Add some noise

    cells = new_cells

# Output the best solution
best_cell = cells[np.argmin(fitness)]
print(f"Best solution found: {best_cell}")
print(f"Fitness: {fitness_function(best_cell)}")
```

Output:

```
Best solution found: -0.11165744078455692
Fitness: 0.012467384082556834
```

Program 7

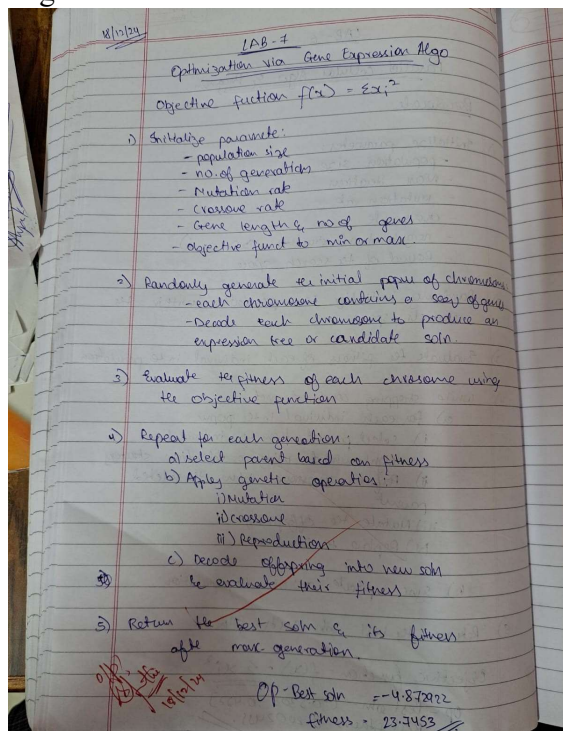
Optimization via Gene Expression Algorithms

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.

Implementation Steps:

1. Define the Problem: Create a mathematical function to optimize.
2. Initialize Parameters: Set the population size, number of genes, mutation rate, crossover rate, and number of generations.
3. Initialize Population: Generate an initial population of random genetic sequences.
4. Evaluate Fitness: Evaluate the fitness of each genetic sequence based on the optimization function.
5. Selection: Select genetic sequences based on their fitness for reproduction.
6. Crossover: Perform crossover between selected sequences to produce offspring.
7. Mutation: Apply mutation to the offspring to introduce variability.
8. Gene Expression: Translate genetic sequences into functional solutions.
9. Iterate: Repeat the selection, crossover, mutation, and gene expression processes for a fixed number of generations or until convergence criteria are met.
10. Output the Best Solution: Track and output the best solution found during the iterations.

Algorithm:



Code:

#Optimization via Gene Expression Algorithms

import numpy as np

Define the optimization function

```
def fitness_function(x):  
    return x**2
```

Convert binary string to decimal

```
def binary_to_decimal(binary_str):  
    return int(binary_str, 2) / (2**len(binary_str) - 1) * 10 - 5 # Scale to [-5, 5]
```

Initialize parameters

```
population_size = 20  
num_genes = 10  
mutation_rate = 0.1  
crossover_rate = 0.7  
generations = 100
```

Initialize population

```
population = ["".join(np.random.choice(['0', '1'], num_genes)) for _ in range(population_size)]
```

Main loop

```
for _ in range(generations):
```

```
    # Evaluate fitness
```

```
    fitness = [fitness_function(binary_to_decimal(ind)) for ind in population]
```

```
    # Selection (roulette wheel)
```

```
    total_fitness = sum(fitness)
```

```
    probabilities = [f / total_fitness for f in fitness]
```

```
    selected = np.random.choice(population, size=population_size, p=probabilities)
```

```
    # Crossover
```

```
    offspring = []
```

```
    for i in range(0, population_size, 2):
```

```
        if np.random.rand() < crossover_rate:
```

```
            point = np.random.randint(1, num_genes)
```

```
            offspring.append(selected[i][:point] + selected[i+1][point:])
```

```
            offspring.append(selected[i+1][:point] + selected[i][point:])
```

```
        else:
```

```
            offspring.append(selected[i])
```

```
            offspring.append(selected[i+1])
```

```

# Mutation
for i in range(population_size):
    if np.random.rand() < mutation_rate:
        point = np.random.randint(num_genes)
        offspring[i] = offspring[i][:point] + ('1' if offspring[i][point] == '0' else '0') + offspring[i]
        [point+1:]
    population = offspring

# Output the best solution

best_individual = min(population, key=lambda ind: fitness_function(binary_to_decimal(ind)))
best_fitness = fitness_function(binary_to_decimal(best_individual))
print(f"Best solution found: {binary_to_decimal(best_individual)}")
print(f"Fitness: {best_fitness}")

```

Output:

```

Best solution found: -4.872922776148583
Fitness: 23.74537638230761

```