VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB RECORD

# Bio Inspired Systems (23CS5BSBIS)

*Submitted by*

Rushi Hundiwala (1BM22CS224)

*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 Rushi Hundiwala (1BM22CS224), who is bonafide student of B.M.S. College of Engineering. It is in partial fulfillment for the award of Bachelor of Engineering in Computer Science and Engineering of the Visvesvaraya Technological University, Belgaum. The Lab report has been approved as it satisfies the academic requirements of the above mentioned subject and the work prescribed for the said degree.

| | |
|---|---|
| Lab faculty In charge Name:<br>Prof.Swathi Sridharan<br>Assistant Professor Department of CSE, BMSCE | Dr. Joythi S Nayak<br>Professor & HOD<br>Department of CSE, BMSCE |

# Index



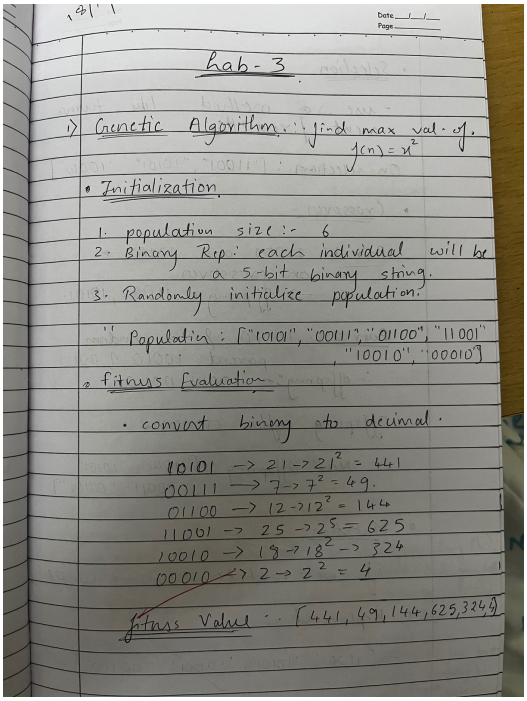
Github Link:

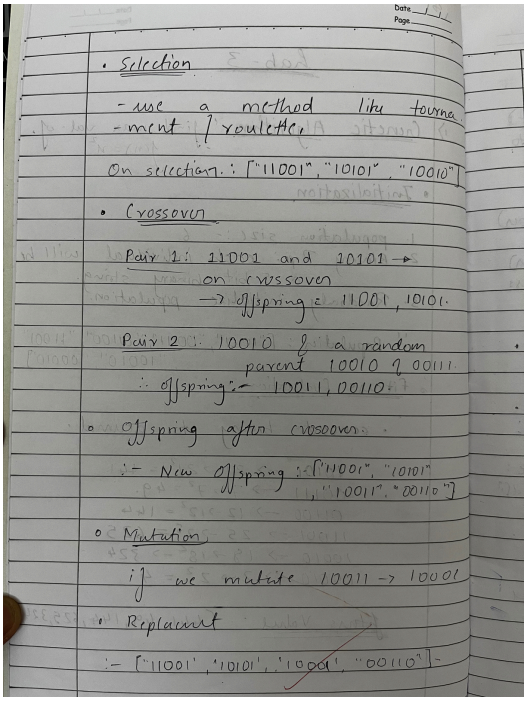
https://github.com/RishiHundiwala344/BISlab

**PROBLEM STATEMENT:**

**Genetic Algorithm for Optimization Problems:** "Implement a Genetic Algorithm using Python to solve optimization problems, such as finding the maximum value of a mathematical function. The problem statement should include fields such as engineering design, machine learning, and data mining where GA can be applied. The Genetic Algorithm will employ selection, crossover, mutation, and elitism techniques to iteratively improve candidate solutions. The goal is to find the best solution by mimicking the process of natural selection, where the fittest individuals are selected for reproduction, leading to the evolution of increasingly optimal solutions."

**ALGORITHM:**



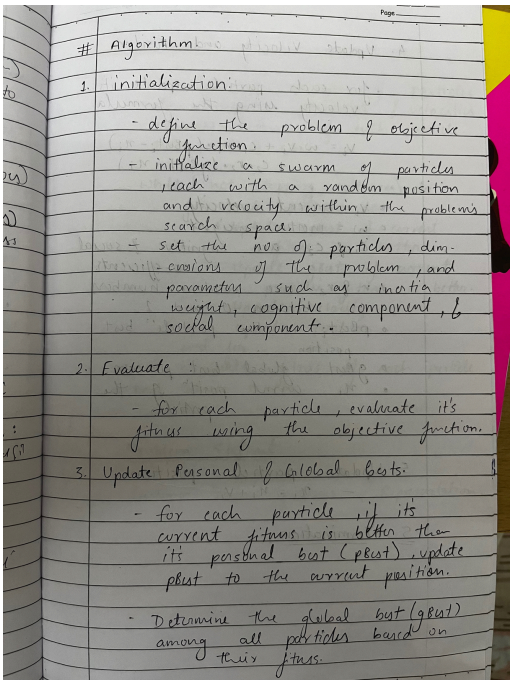
**CODE:**

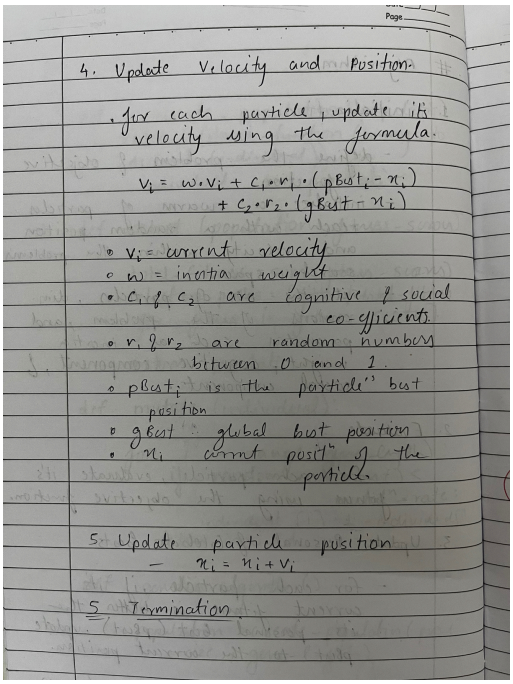
```
import numpy as np
import random

# Objective function (here we're maximizing f(x) = -x^2 + 10)
def fitness(x):
    return -x**2

# Create initial population
def create_population(size, low, high):
    return np.random.uniform(low, high, size)

# Select individuals based on their fitness
def select(population, fitness_values):
    idx = np.argsort(fitness_values)[-2:]  # Select top 2 individuals
    return population[idx]

# Crossover: combine two parents to produce an offspring
def crossover(parent1, parent2, crossover_rate=0.7):
    if random.random() < crossover_rate:
        alpha = random.random()
        return parent1 * alpha + parent2 * (1 - alpha)
    else:
        return parent1

# Mutation: introduce small random changes
def mutate(individual, mutation_rate=0.01, low=-5, high=5):
    if random.random() < mutation_rate:
        return np.random.uniform(low, high)
    else:
        return individual

# Genetic Algorithm main loop
def genetic_algorithm(pop_size=100, generations=50, crossover_rate=0.7, mutation_rate=0.01, low=-5, high=5):
    # Initialize population
    population = create_population(pop_size, low, high)

    for gen in range(generations):
        # Evaluate fitness
        fitness_values = np.array([fitness(x) for x in population])

        # Selection
        parents = select(population, fitness_values)

        # Generate new population through crossover and mutation
        new_population = []
        for _ in range(pop_size):
            parent1, parent2 = random.sample(parents, 2)
            offspring = crossover(parent1, parent2, crossover_rate)
            offspring = mutate(offspring, mutation_rate, low, high)
            new_population.append(offspring)

        population = np.array(new_population)

    # Best solution found
    best_solution = population[np.argmax(fitness_values)]
    return best_solution, fitness(best_solution)

    best_solution, best_fitness = genetic_algorithm()
    print(f"Best solution found: x = {best_solution}, f(x) = {best_fitness}")
```

**PROBLEM STATEMENT:**

**Particle Swarm Optimization for Function Optimization:** "Implement the Particle Swarm Optimization (PSO) algorithm using Python to optimize mathematical functions. This problem statement involves fields such as signal processing, neural networks, and control systems. The PSO algorithm will simulate the social behavior of birds flocking or fish schooling, using iterative improvement of candidate solutions based on individual and collective best experiences. The objective is to find optimal solutions by continuously adjusting the position and velocity of particles, leading them towards the best solution in the search space."

**ALGORITHM:**



**CODE:**

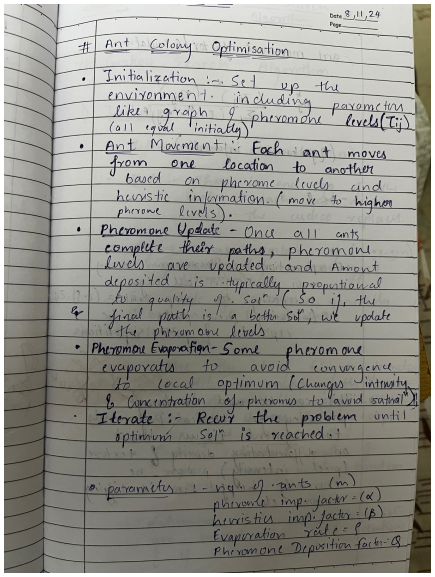
```
import numpy as np
import matplotlib.pyplot as plt

class Particle:
    def __init__(self, num_clusters, dimensions):
        # Initialize positions randomly within the data range
        self.position = np.random.rand(num_clusters, dimensions) * 10  # Assuming data range is 0-10
        self.velocity = np.random.rand(num_clusters, dimensions) * 2 - 1
        self.best_position = self.position.copy()
        self.best_value = float('inf')

def objective_function(data, cluster_centers):
    # Calculate the total distance (inertia) for K-means
    total_distance = 0
    for point in data:
        distances = np.linalg.norm(point - cluster_centers, axis=1)
        total_distance += np.min(distances)
    return total_distance

def pso_kmeans(data, num_clusters, num_particles, max_iterations):
    # Parameters
    w = 0.5
    c1 = 1.5
    c2 = 1.5

    swarm = [Particle(num_clusters, data.shape[1]) for _ in range(num_particles)]
    global_best_position = None
    global_best_value = float('inf')

    for iteration in range(max_iterations):
        for particle in swarm:
            # Evaluate fitness
            fitness_value = objective_function(data, particle.position)

            # Update personal best
            if fitness_value < particle.best_value:
                particle.best_value = fitness_value
                particle.best_position = particle.position.copy()
```

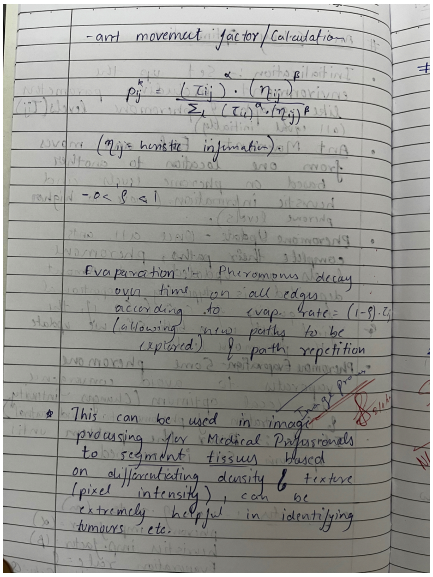
```
# Update global best
 if fitness_value < global_best_value:
 global_best_value = fitness_value
 global_best_position = particle.position.copy()

 for particle in swarm:
 # Update velocity
 r1, r2 = np.random.rand(num_clusters, data.shape[1]), np.random.rand(num_clusters, data.shape[1])
 particle.velocity = (w * particle.velocity +
 c1 * r1 * (particle.best_position - particle.position) +
 c2 * r2 * (global_best_position - particle.position))

 # Update position
 particle.position += particle.velocity

 # Ensure the positions are within bounds
 particle.position = np.clip(particle.position, 0, 10) # Assuming data range is 0-10

 return global_best_position

# Generate synthetic data
np.random.seed(42)
data = np.vstack([
 np.random.normal(loc=[2, 2], scale=0.5, size=(50, 2)),
 np.random.normal(loc=[8, 8], scale=0.5, size=(50, 2)),
 np.random.normal(loc=[5, 3], scale=0.5, size=(50, 2))
])

# Parameters
num_clusters = 3
num_particles = 30
max_iterations = 100

# Run PSO for K-means
best_centers = pso_kmeans(data, num_clusters, num_particles, max_iterations)

# Plotting
plt.scatter(data[:, 0], data[:, 1], color='lightgray')
plt.scatter(best_centers[:, 0], best_centers[:, 1], color='red', marker='x', s=200, label='Cluster Centers')
plt.title('PSO-based K-means Clustering')
plt.legend()
plt.show()
```

**PROBLEM STATEMENT:**

**Ant Colony Optimization for the Traveling Salesman Problem:** "Implement the Ant Colony Optimization (ACO) algorithm using Python to solve the Traveling Salesman Problem (TSP). The problem statement includes fields such as logistics, network design, and supply chain management where finding the shortest path is critical. The ACO algorithm will simulate the foraging behavior of ants to find the shortest possible route visiting a list of cities and returning to the origin. The ants will deposit pheromones on paths they traverse, reinforcing shorter paths based on the distance and pheromone levels, leading to increasingly optimized solutions."

**ALGORITHM:**



**CODE:**

```
import numpy as np
import cv2
import random
import math
import matplotlib.pyplot as plt

# Parameters for Ant Colony Optimization
n_ants = 50             # Number of ants
n_iterations = 100      # Number of iterations
alpha = 1               # Influence of pheromone
beta = 1                # Influence of distance/visibility (pixel similarity)
evaporation_rate = 0.1  # Pheromone evaporation rate
q = 1                   # Total pheromone deposited per ant

# Image loading and preprocessing
image = cv2.imread('medical_image.jpg', cv2.IMREAD_GRAYSCALE)  # Grayscale image
height, width = image.shape

# Initialize pheromone map and visibility map
pheromone_map = np.ones((height, width))  # Pheromone map, starts with 1
visibility_map = 255 - image  # Inverse of image intensity as visibility (lower intensity = higher visibility)

# Initialize ants' positions
ants_positions = np.random.randint(0, height, size=(n_ants, 2))

# Function to compute pixel similarity (e.g., Euclidean distance or intensity difference)
def compute_similarity(x, y):
    # Euclidean distance between two pixel intensities (foreground vs background)
    return np.abs(int(image[x, y]) - 128)  # Threshold intensity (128) for foreground/background

# Function to move an ant
def move_ant(ant_position, pheromone_map, visibility_map, alpha, beta):
    x, y = ant_position
    # Calculate probabilities for the four neighboring pixels (up, down, left, right)
    neighbors = [(x-1, y), (x+1, y), (x, y-1), (x, y+1)]
    probabilities = []

    for nx, ny in neighbors:
        if 0 <= nx < height and 0 <= ny < width:
            # Calculate the desirability (visibility) and pheromone level for each neighbor
            pheromone_level = pheromone_map[nx, ny] ** alpha
            visibility_level = visibility_map[nx, ny] ** beta
            probability = pheromone_level * visibility_level
            probabilities.append(probability)
        else:
            probabilities.append(0)

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

    # Select the next move based on probabilities
    move = random.choices(neighbors, probabilities)[0]
    return move

# Function to update pheromone levels
def update_pheromones(pheromone_map, ants_positions, evaporation_rate, q):
    pheromone_map *= (1 - evaporation_rate)  # Apply evaporation

    for x, y in ants_positions:
        pheromone_map[x, y] += q  # Increase pheromone in the visited cell
```

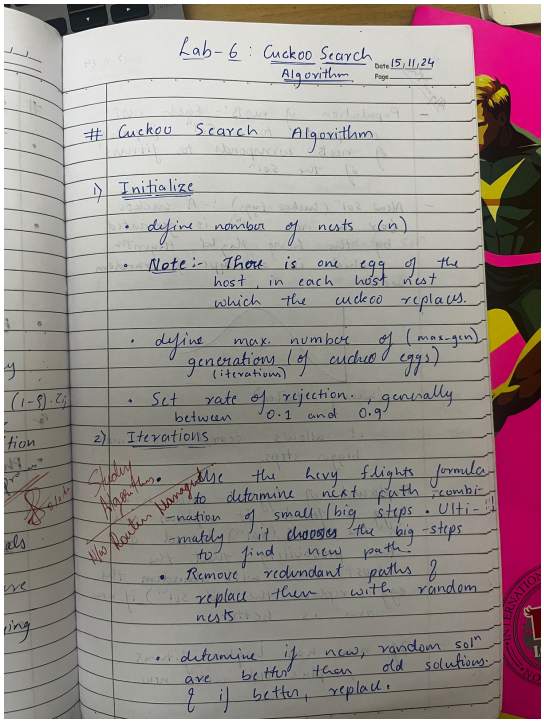
**CODE:**

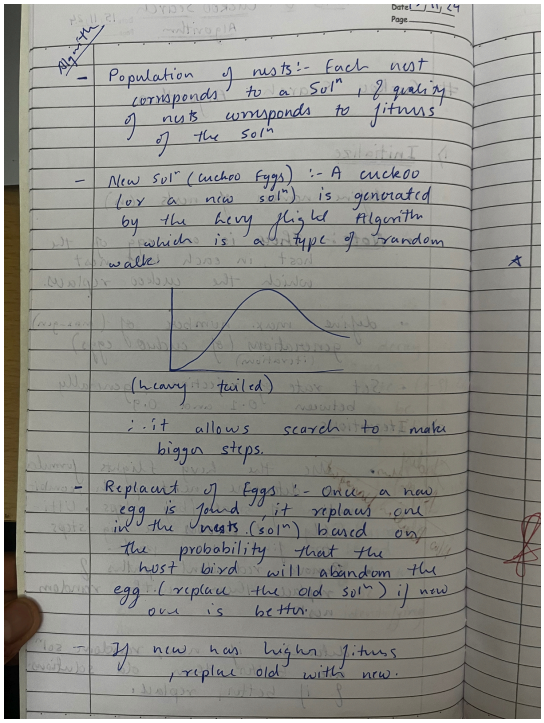
```
# Function to segment the image using ACO
def aco_segmentation(image, n_ants, n_iterations, alpha, beta, evaporation_rate, q):
    pheromone_map = np.ones((height, width))  # Initialize pheromone map
    best_segmentation = None
    best_score = float('inf')

    for iteration in range(n_iterations):
        ants_positions = np.random.randint(0, height, size=(n_ants, 2))  # Initialize ants' positions
        visited_positions = []

        # Simulate the movement of ants
        for i in range(n_ants):
            ant_position = tuple(ants_positions[i])
            visited_positions.append(ant_position)
            # Each ant moves through the image and creates a segmentation
            for _ in range(100):  # Maximum number of steps an ant can take
                ant_position = move_ant(ant_position, pheromone_map, visibility_map, alpha, beta)
                visited_positions.append(ant_position)

        # Update pheromone map based on visited positions
        update_pheromones(pheromone_map, visited_positions, evaporation_rate, q)

        # Evaluate segmentation quality (objective function)
        segmentation_score = evaluate_segmentation(visited_positions)
        if segmentation_score < best_score:
            best_score = segmentation_score
            best_segmentation = visited_positions

        # Optional: Print progress
        if iteration % 10 == 0:
            print(f"Iteration {iteration + 1}/{n_iterations}, Best Score: {best_score}")

    return best_segmentation

# Function to evaluate the segmentation quality (e.g., based on the number of errors)
def evaluate_segmentation(visited_positions):
    # Here, you can implement your custom evaluation function. This could be based on:
    # - Number of foreground vs. background pixels correctly classified
    # - Similarity to ground truth (if available)
    # - Edge sharpness, etc.
    return len(visited_positions)  # Example: the number of visited positions (just a placeholder)

# Run ACO for image segmentation
best_segmentation = aco_segmentation(image, n_ants, n_iterations, alpha, beta, evaporation_rate, q)

# Visualize the segmentation result
segmentation_image = np.zeros_like(image)
for (x, y) in best_segmentation:
    segmentation_image[x, y] = 255  # Mark foreground pixels

plt.imshow(segmentation_image, cmap='gray')
plt.title("ACO Image Segmentation")
plt.show()
```

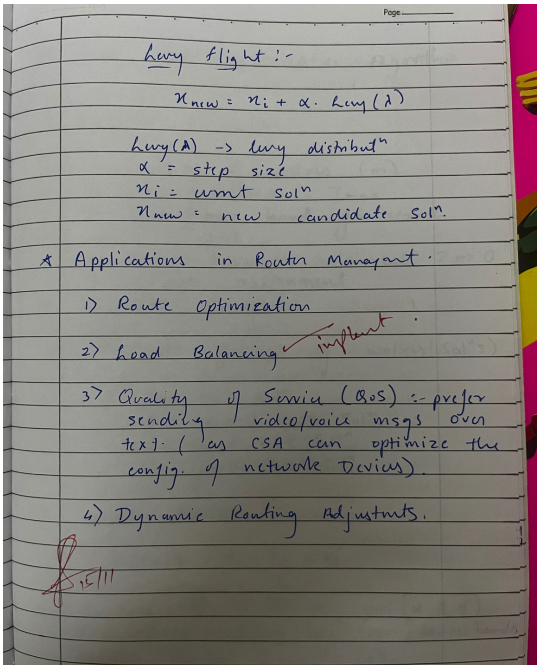
**PROBLEM STATEMENT:**

**Cuckoo Search (CS):** "Implement the Cuckoo Search (CS) algorithm using Python to solve continuous optimization problems, including engineering design, data analysis, and machine learning. The problem statement will focus on fields where the global search capabilities of CS are crucial. The CS algorithm will model the brood parasitism behavior of cuckoos, using Lévy flights for generating new solutions. It will promote exploration and avoid local minima by mimicking the egg laying process and the random walks of cuckoos. The objective is to find optimal solutions through a balance of exploration and exploitation."

**ALGORITHM:**



## CODE:

```
#implementation of load balancing in router networks using cuckoosearch

import numpy as np

# Parameters for the Cuckoo Search Algorithm
n = 20              # Number of cuckoos (solutions)
max_iter = 100      # Maximum number of iterations
pa = 0.25           # Probability of discovery of an alien egg
alpha = 0.01        # Step size for Lévy flight

# Define the network parameters
num_routers = 5      # Number of routers in the network
router_capacity = [100, 100, 100, 100, 100]  # Example router capacities
traffic_load = [300, 200, 400, 250, 150]  # Example traffic load demands

# Define the objective function to minimize (Max load in any router)
def objective_function(load_distribution, router_capacity, traffic_load):
    # Compute the load on each router based on the distribution
    router_loads = np.zeros(len(router_capacity))

    for i, load in enumerate(traffic_load):
        router_index = int(load_distribution[i])
        if router_index < len(router_capacity):
            router_loads[router_index] += load

    # Objective: Minimize the maximum load across routers (balance the load)
    max_load = np.max(router_loads)
    return max_load

# Initialize the cuckoo search algorithm
def initialize_population(n, num_routers):
    population = np.random.randint(0, num_routers, size=(n, len(traffic_load)))
    return population

# Lévy flight for exploration (random walk) without scipy
def levy_flight(alpha, size):
    # Use a power-law distribution (Lévy flight)
    beta = 1.5
    # Lévy flight uses the fact that step sizes follow a power-law distribution
    # Using a normal distribution and scaling to match Lévy distribution
    sigma = np.sqrt(alpha)  # Adjusted step size
    s = np.random.normal(0, sigma, size)
    t = np.random.normal(0, 1, size)
    return s / np.abs(t)  # This creates the "Lévy flight" distribution

# Cuckoo Search Algorithm for Load Balancing
def cuckoo_search(traffic_load, router_capacity, n, max_iter, pa, alpha):
    # Initialize population
    population = initialize_population(n, num_routers)
```

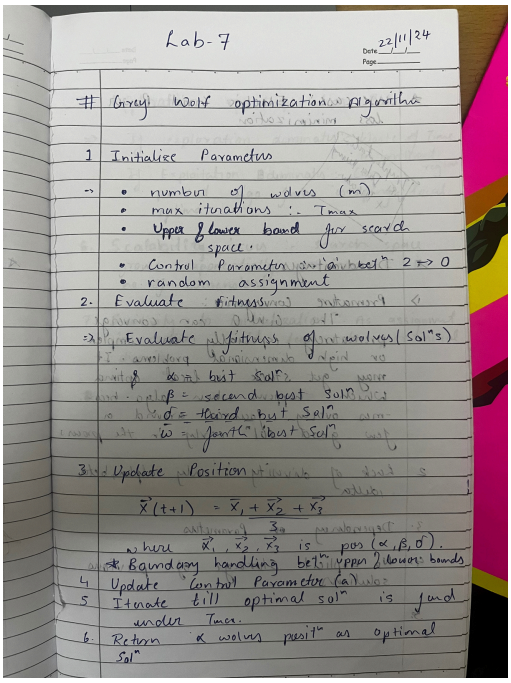
## CODE:

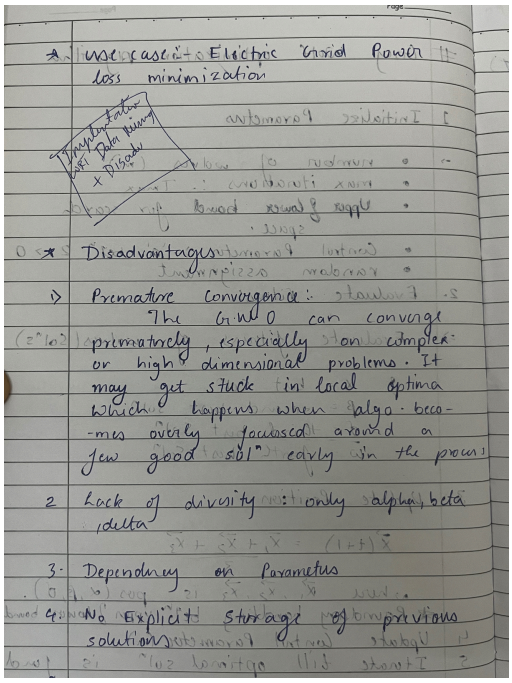
```
    # Initialize the best solution
    best_solution = None
    best_objective = float('inf')

    # Main optimization loop
    for iteration in range(max_iter):
        new_population = np.copy(population)

        # Evaluate the current population
        for i in range(n):
            objective = objective_function(population[i], router_capacity, traffic_load)

            # Update the best solution found so far
            if objective < best_objective:
                best_solution = population[i]
                best_objective = objective

        # Generate new solutions using Lévy flight
        for i in range(n):
            step_size = alpha * levy_flight(alpha, len(traffic_load))
            new_solution = population[i] + step_size
            new_solution = np.clip(new_solution, 0, num_routers - 1).astype(int)

            # Evaluate new solution
            new_objective = objective_function(new_solution, router_capacity, traffic_load)

            # If the new solution is better, replace the old one
            if new_objective < best_objective:
                population[i] = new_solution
                best_solution = new_solution
                best_objective = new_objective

        # Apply the probability of discovery of alien eggs
        for i in range(n):
            if np.random.rand() < pa:
                population[i] = np.random.randint(0, num_routers, size=len(traffic_load))

        # Display progress
        print(f"Iteration {iteration+1}/{max_iter}, Best Load Distribution: {best_solution}, Max Load: {best_objective}")

    return best_solution, best_objective

# Run the Cuckoo Search Algorithm for Load Balancing
best_solution, best_objective = cuckoo_search(traffic_load, router_capacity, n, max_iter, pa, alpha)

print(f"Optimal load distribution: {best_solution}")
print(f"Optimal objective (max load): {best_objective}")
```

**PROBLEM STATEMENT:**

**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.

**ALGORITHM:**



## CODE:

```
import numpy as np

# Objective function: Dummy function for demonstration
def fitness_function(solution):
    """Fitness function mimicking data mining - minimizes the number of selected features."""
    return np.sum(solution)  # Fewer selected features are better

# Grey Wolf Optimization Algorithm (Simplified)
class GreyWolfOptimizer:
    def __init__(self, n_agents, n_features, max_iter):
        self.n_agents = n_agents
        self.n_features = n_features
        self.max_iter = max_iter
        # Initialize binary positions (0 or 1) randomly
        self.positions = np.random.randint(2, size=(n_agents, n_features))
        self.alpha, self.beta, self.delta = None, None, None

    def optimize(self):
        for iteration in range(self.max_iter):
            # Evaluate fitness for all agents
            fitness = np.array([fitness_function(agent) for agent in self.positions])
            # Rank agents by fitness
            sorted_indices = np.argsort(fitness)
            self.alpha, self.beta, self.delta = (
                self.positions[sorted_indices[0]],
                self.positions[sorted_indices[1]],
                self.positions[sorted_indices[2]],
            )

            # Update positions
            for i in range(self.n_agents):
                for j in range(self.n_features):
                    r1, r2, r3 = np.random.random(), np.random.random(), np.random.random()
                    A1, A2, A3 = 2 * r1 - 1, 2 * r2 - 1, 2 * r3 - 1
                    C1, C2, C3 = 2 * np.random.random(), 2 * np.random.random(), 2 * np.random.random()

                    D_alpha = abs(C1 * self.alpha[j] - self.positions[i][j])
                    D_beta = abs(C2 * self.beta[j] - self.positions[i][j])
                    D_delta = abs(C3 * self.delta[j] - self.positions[i][j])

                    X1 = self.alpha[j] - A1 * D_alpha
                    X2 = self.beta[j] - A2 * D_beta
                    X3 = self.delta[j] - A3 * D_delta

                    # Update position with a threshold
                    self.positions[i][j] = 1 if (X1 + X2 + X3) / 3 > 0.5 else 0

            # Output progress
            print(f"Iteration {iteration + 1}, Best Fitness: {fitness[sorted_indices[0]]}")
```

**CODE:**

```
    # Example Usage
if __name__ == "__main__":
    # Simplified example with 5 agents, 10 features, and 10 iterations
    n_agents = 5
    n_features = 10
    max_iter = 10

    gwo = GreyWolfOptimizer(n_agents=n_agents, n_features=n_features, max_iter=max_iter)
    best_solution = gwo.optimize()
    print(f"Best Solution: {best_solution}")
```

## CODE:

```
   #use this if modules can be downloaded

import numpy as np
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score

# Grey Wolf Optimization (GWO) Algorithm
class GreyWolfOptimizer:
   def __init__(self, obj_function, n_wolves, max_iter, dim, lower_bound, upper_bound):
      self.obj_function = obj_function  # Objective function
      self.n_wolves = n_wolves  # Number of wolves
      self.max_iter = max_iter  # Maximum iterations
      self.dim = dim  # Dimension of the problem
      self.lower_bound = lower_bound  # Lower bound of the search space
      self.upper_bound = upper_bound  # Upper bound of the search space
      self.alpha_pos = np.zeros(dim)  # Position of the alpha wolf
      self.alpha_score = float("inf")  # Score of the alpha wolf
      self.beta_pos = np.zeros(dim)  # Position of the beta wolf
      self.beta_score = float("inf")  # Score of the beta wolf
      self.delta_pos = np.zeros(dim)  # Position of the delta wolf
      self.delta_score = float("inf")  # Score of the delta wolf
      self.positions = np.random.uniform(lower_bound, upper_bound, (n_wolves, dim))  # Initial positions of
wolves
      self.scores = np.zeros(n_wolves)  # Scores of the wolves

   def optimize(self):
      for t in range(self.max_iter):
         for i in range(self.n_wolves):
            self.scores[i] = self.obj_function(self.positions[i])  # Evaluate fitness function

            if self.scores[i] < self.alpha_score:
               self.alpha_score = self.scores[i]
               self.alpha_pos = self.positions[i]

            if self.scores[i] < self.beta_score and self.scores[i] > self.alpha_score:
               self.beta_score = self.scores[i]
               self.beta_pos = self.positions[i]

            if self.scores[i] < self.delta_score and self.scores[i] > self.beta_score:
               self.delta_score = self.scores[i]
               self.delta_pos = self.positions[i]

         a = 2 - t * (2 / self.max_iter)  # Decrease in a over time

         # Update positions of wolves
         for i in range(self.n_wolves):
            for j in range(self.dim):
               r1 = np.random.random()
               r2 = np.random.random()

               A1 = 2 * a * r1 - a  # Coefficient A1
               C1 = 2 * r2  # Coefficient C1
```

**CODE:**

```
    D_alpha = np.abs(C1 * self.alpha_pos[j] - self.positions[i, j])  # Distance to alpha
                X1 = self.alpha_pos[j] - A1 * D_alpha  # New position for alpha wolf

                A2 = 2 * a * r1 - a  # Coefficient A2
                C2 = 2 * r2  # Coefficient C2

                D_beta = np.abs(C2 * self.beta_pos[j] - self.positions[i, j])  # Distance to beta
                X2 = self.beta_pos[j] - A2 * D_beta  # New position for beta wolf

                A3 = 2 * a * r1 - a  # Coefficient A3
                C3 = 2 * r2  # Coefficient C3

                D_delta = np.abs(C3 * self.delta_pos[j] - self.positions[i, j])  # Distance to delta
                X3 = self.delta_pos[j] - A3 * D_delta  # New position for delta wolf

                # Update position of wolf
                self.positions[i, j] = (X1 + X2 + X3) / 3

        return self.alpha_pos, self.alpha_score

# Feature Selection Objective Function
def feature_selection_function(position):
    selected_features = [i for i, value in enumerate(position) if value > 0.5]
    if len(selected_features) == 0:
        return float("inf")  # No feature selected
    X_selected = X_train[:, selected_features]
    model = SVC(kernel='linear')
    model.fit(X_selected, y_train)
    y_pred = model.predict(X_selected)
    return -accuracy_score(y_train, y_pred)  # Minimize the negative accuracy

# Load Dataset
data = load_iris()
X = data.data
y = data.target

# Train-Test Split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Initialize Grey Wolf Optimizer
gwo = GreyWolfOptimizer(
    obj_function=feature_selection_function,
    n_wolves=30,  # Number of wolves
    max_iter=50,  # Maximum number of iterations
    dim=X_train.shape[1],  # Number of features
    lower_bound=0,  # Lower bound for feature selection (0 or 1)
    upper_bound=1  # Upper bound for feature selection (0 or 1)
)

# Run Optimization
best_position, best_score = gwo.optimize()

# Feature selection result
selected_features = [i for i, value in enumerate(best_position) if value > 0.5]
print(f"Selected Features: {selected_features}")
```

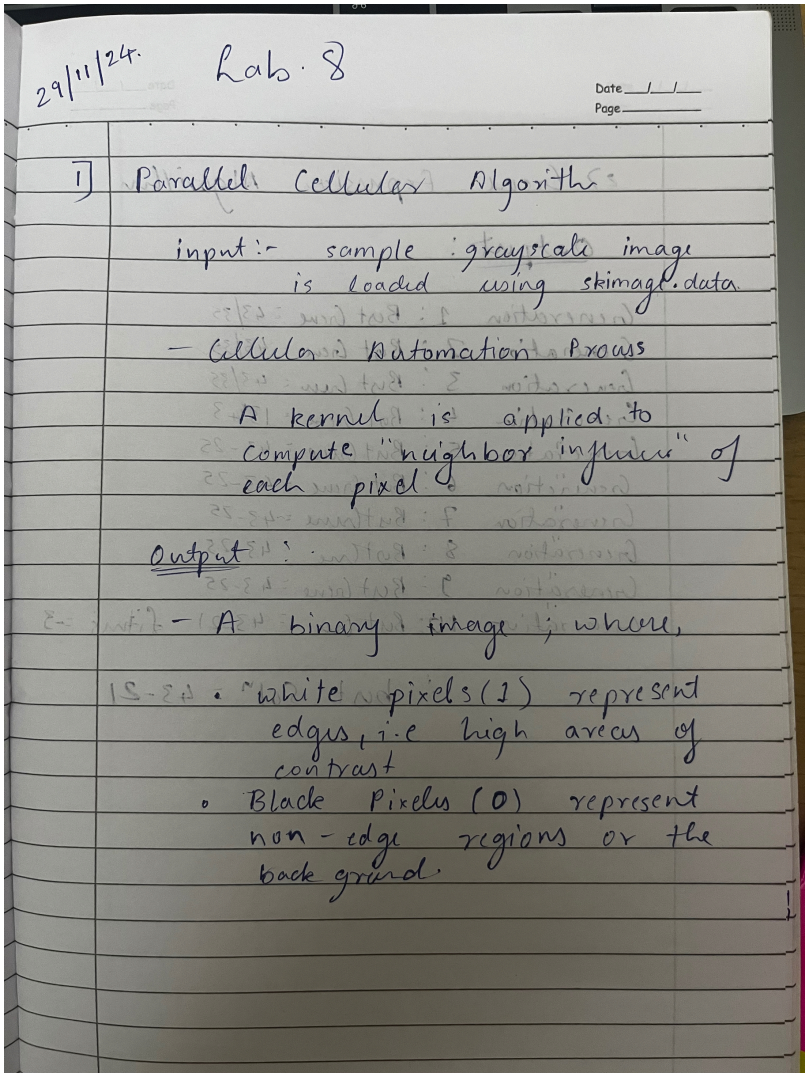
**CODE:**

```
   # Evaluate on Test Set with Selected Features
X_train_selected = X_train[:, selected_features]
X_test_selected = X_test[:, selected_features]

model = SVC(kernel='linear')
model.fit(X_train_selected, y_train)
y_pred = model.predict(X_test_selected)
print(f"Test Accuracy: {accuracy_score(y_test, y_pred)}")
```

**PROBLEM STATEMENT:**

**Parallel Cellular Algorithms and Programs:** "Implement Parallel Cellular Algorithms using Python to solve large-scale optimization problems efficiently, including parallel computing applications, complex system simulations, and data mining. The problem statement will focus on fields where parallel computing enhances performance, such as weather prediction, genetic algorithms, and network optimization. The algorithm will model the functioning of biological cells using cellular automata, allowing each cell to interact with its neighbors. By diffusing information across the cellular grid, it can explore the search space effectively and identify optimal solutions in real-time."

**ALGORITHM:**



29/11/24. Lab. 8

Date __/__/__
Page _____

1] Parallel Cellular Algorithm

input :- sample : grayscale image is loaded using skimage.data

- Cellular Automation Process

A kernel is applied to compute "neighbor influence" of each pixel.

Output :

- A binary image ; where,
  - White pixels (1) represent edges, i.e high areas of contrast
  - Black Pixels (0) represent non-edge regions or the back ground.

## CODE:

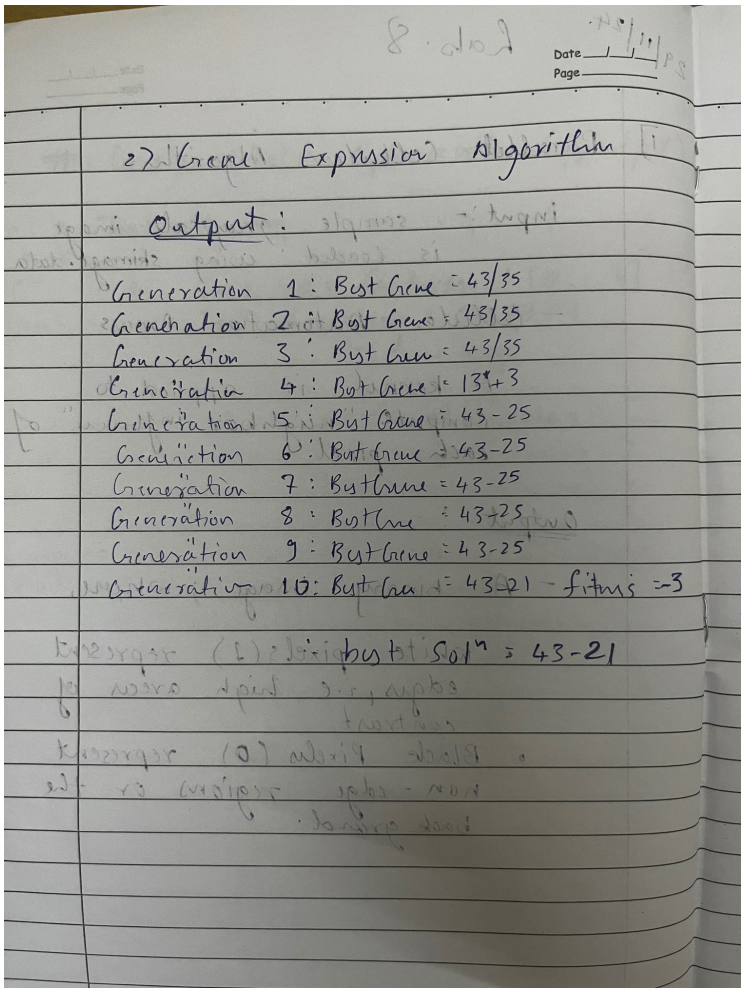
```
  #this code is used for edge detecting in an image and uses parallel cellular algorithm

import numpy as np
import matplotlib.pyplot as plt
from scipy.ndimage import convolve

# Cellular Automaton for Image Processing
class CellularAutomaton:
    def __init__(self, image, kernel, threshold):
        """
        Initialize the CA with the image and rules.
        :param image: Input grayscale image (2D numpy array).
        :param kernel: Kernel for neighbor computation.
        :param threshold: Threshold to determine state change.
        """
        self.image = image
        self.kernel = kernel
        self.threshold = threshold

    def apply_rules(self):
        """
        Apply CA rules in parallel to update the image.
        :return: Updated image (2D numpy array).
        """
        # Convolve image with the kernel to compute neighbor influence
        neighbor_sum = convolve(self.image, self.kernel, mode='constant', cval=0)

        # Apply threshold to determine new state
        new_image = (neighbor_sum > self.threshold).astype(int)

        return new_image

def load_sample_image():
    """Load a sample grayscale image."""
    from skimage import data, color
    img = color.rgb2gray(data.camera())  # Convert to grayscale
    return (img > 0.5).astype(int)  # Binarize for simplicity

# Main Execution
if __name__ == "__main__":
    # Load a sample image
    image = load_sample_image()

    # Define the CA kernel (e.g., Sobel-like kernel for edge detection)
    kernel = np.array([[1, 1, 1],
                       [1, -8, 1],
                       [1, 1, 1]])
```

```
# Initialize CA with a threshold

threshold = 4

ca = CellularAutomaton(image, kernel, threshold)
# Run CA for a few iterations

iterations = 5

for i in range(iterations):
    image = ca.apply_rules()
# Display the final result
    plt.imshow(image, cmap='gray')
    plt.title(f"Image after {iterations} CA iterations")
    plt.axis('off')
    plt.show()
```

**PROBLEM STATEMENT:**

**Optimization via Gene Expression Algorithms (GEA):**"Implement a Gene Expression Algorithm (GEA) using Python to solve complex optimization problems in fields such as engineering design, data analysis, and machine learning. The problem statement will involve encoding solutions as genetic sequences, analogous to DNA, which are expressed into functional solutions through a simulated biological gene expression process. The GEA will employ evolutionary operators such as selection, crossover, mutation, and gene expression to evolve and optimize solutions iteratively. The objective is to find optimal or near-optimal solutions to challenging problems by leveraging the robust and adaptive mechanisms of biological gene expression."

**ALGORITHM:**



## CODE:

```
  import numpy as np

# Define the target function
def target_function(x):
    return x**2  # The function to optimize

# Define the fitness function
def fitness_function(expression, target, x_value):
    """
    Evaluate the fitness of an expression.
    :param expression: The candidate solution (as a string).
    :param target: Target output to achieve (e.g., x^2).
    :param x_value: The value of x to plug into the function.
    :return: Fitness value (higher is better).
    """
    try:
        result = eval(expression)  # Evaluate the expression
        return -abs(result - target_function(x_value))  # Closer to x^2, better the fitness
    except:
        return float('-inf')  # Invalid expressions get very low fitness

# Gene Expression Algorithm
class GeneExpressionAlgorithm:
    def __init__(self, population_size, gene_length, target, generations, mutation_rate, x_value):
        self.population_size = population_size
        self.gene_length = gene_length
        self.target = target
        self.generations = generations
        self.mutation_rate = mutation_rate
        self.x_value = x_value
        self.operators = ['+', '-', '*', '/', '**']
        self.variables = ['x']
        self.constants = ['1', '2', '3', '4', '5']
        self.population = self._initialize_population()

    def _initialize_population(self):
        """
        Generate a random initial population.
        """
        population = []
        for _ in range(self.population_size):
            gene = ''.join(
                np.random.choice(self.variables + self.operators + self.constants, self.gene_length)
            )
            population.append(gene)
        return population
```

**CODE:**

```
def _mutate(self, gene):
      """
      Apply random mutation to a gene.
      """
      gene = list(gene)
      for i in range(len(gene)):
         if np.random.rand() < self.mutation_rate:
            gene[i] = np.random.choice(self.variables + self.operators + self.constants)
      return ''.join(gene)

   def evolve(self):
      """
      Evolve the population to optimize the function.
      """
      for generation in range(self.generations):
         # Evaluate fitness for each gene in the population
         fitness = [fitness_function(gene, self.target, self.x_value) for gene in self.population]

         # Select the best-performing genes
         sorted_indices = np.argsort(fitness)[::-1]  # Descending sort
         self.population = [self.population[i] for i in sorted_indices[:self.population_size // 2]]

         # Generate offspring by mutating the best genes
         offspring = [self._mutate(gene) for gene in self.population]
         self.population += offspring

         # Print the best gene of the generation
         print(f"Generation {generation + 1}: Best Gene = {self.population[0]}, Fitness =
{fitness[sorted_indices[0]]}")

      # Return the best solution
      best_gene = self.population[0]
      return best_gene

# Main Execution
if __name__ == "__main__":
   # Parameters
   population_size = 20
   gene_length = 5
   target = 25  # Target value of x^2 for x=5
   x_value = 5  # Use x = 5 for optimization
   generations = 10
   mutation_rate = 0.2

   # Initialize and run the algorithm
   gep = GeneExpressionAlgorithm(population_size, gene_length, target, generations, mutation_rate, x_value)
   best_solution = gep.evolve()
   print(f"Best Solution: {best_solution}")
```