Bio Inspired System

Slno	Experiment	Date	Signature	Marks
1.	Genetic Algorithm	18|8|25	Asry	09
2.	Gene Expression Algo	25|8|25	Asry	09
3.	Particle Swarm Optimization	01|9|25	Spit	09
4.	Ant Colony optimization for TSP	08|9|25	Spit	09
5.	Cuckoo Search	15|9|25	Spit	09
6.	Grey wolf Optimizer	29|9|25	Spit	08
7.	Parallel cellular Algorithm	13|10|25	Spit	09

VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB RECORD

Bio Inspired Systems (23CS5BSBIS)

Submitted by

Jayashree Tarai (1BM24CS407)

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
Aug-2025 to Dec-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 **Jayashree Tarai (1BM24CS407),** 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.

Soumya T	Dr. Kavitha Sooda
Assistant Professor	Professor & HOD
Department of CSE, BMSCE	Department of CSE, BMSCE

Index

Github Link:

https://github.com/Jayashreecse/Bio-Inspired-System-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:

Crossover

Crossover point is choosen randomly

Mutation

String no	offspring after cross over	Mutation chromosome	offspring after mutation	x Value	fitness
1	01101	10000	11101	29	841
2	11000	00000	11000	24	576
3	11011	00000	11011	27	729
4	10001	00101	10100	20	400

Sum	2546
Avg	636.5
Max	841

Output

Genetic Algorithm for optimization Problem Using python

Generation 1, Best fitness : 0.6235
generation 2, Best fitney : 0.6100

Generation 50, Best fitness : 0.6550

Final Accuracy on Validation Set : 0.6700

Pseudo code:

Note : chromosome : set of binary numbers
M   Mutation : change in genetic of offspring

Function fitness(x)
   Return x * x

function decode (chromosome)
   Convert the binary list to decimal number
   Return decimal_value

function create_population ()
   population = []
   for i = 1 to 10
    chromosome = random list of 5 bits (0 to 1)
    Add chromosome to population
   Return population

Function evaluate population (population)
   fitness_list = []
   for each chromosome in population
    x = decode (chromosome)
    f = fitness (x)
    Add f to fitness_list
   Return fitness_list

function select parents (population, fitness_list)
   Use roulette wheel selection based on fitness value
   Return selected parents

function Crossover (parent1, parent2)
   If random < 0.7
   choose a random crossover point
   child1 = first part of parent1 + second part of parent2
   Child2 = first part of parent2 + second part of part1

Else :
   child 1 = copy of parent1
   child 2 = copy of parent2
Return child1, child2

Function mutate (chromosome)
  for each bit in chromosome
   if random < 0.1
   flip the bit (0 becomes 1, 1 becomes 0)
  Return chromosome

function genetic algorithm ()
  population = create_population()
  best chromosome = None
  best_fitness = infinity

for for generation = 1 to 10
  fitness_list = evaluate_population (population)

  find chromosome with highest fitness
  If this fitness > best_fitness
   best_chromosome = that chromosome
   best_fitness = its fitness

Print generation number, best x & best fitness
  selected = select_parents (population, fitness_list)
  next_generation = []
  for i = 0 to population_size
   parent 1 = selected [i]
   parent 2 = selected [i+1]
   child1, child2 = crossover (parent1, parent2)
   Child1 = mutate (child1)
   child2 = mutate (child2)
   Add child1 and child2 next generation

population = next generation
Return decode (best_chromosome, best fitness)

Call genetic algorithm ()

Code:

```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

# Creating a sample dataset
X, y = make_classification(n_samples=500, n_features=10, n_informative=8, n_classes=2)
X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2)

# Neural Network Structure
input_size = X.shape[1]
hidden_size = 5
output_size = 1

# Helper functions for the Neural Network
def sigmoid(x):
    return 1 / (1 + np.exp(-x))
def forward_pass(X, weights1, weights2):
    hidden_input = np.dot(X, weights1)
    hidden_output = sigmoid(hidden_input)
    output_input = np.dot(hidden_output, weights2)
    output = sigmoid(output_input)
    return output
def compute_fitness(weights):
    predictions = forward_pass(X_train, weights['w1'], weights['w2'])
    predictions = (predictions > 0.5).astype(int)
    accuracy = accuracy_score(y_train, predictions)
    return accuracy
# Genetic Algorithm Parameters
population_size = 20
generations = 10
mutation_rate = 0.1

# Initialize Population
population = []
for _ in range(population_size):
    individual = {
        'w1': np.random.randn(input_size, hidden_size),
        'w2': np.random.randn(hidden_size, output_size)
    }
    population.append(individual)

# Tracking performance
best_fitness_history = []
```

```python
average_fitness_history = []

# Main Genetic Algorithm Loop
for generation in range(generations):
    # Evaluate Fitness of each Individual
    fitness_scores = np.array([compute_fitness(individual) for individual in population])
    best_fitness = np.max(fitness_scores)
    average_fitness = np.mean(fitness_scores)
    best_fitness_history.append(best_fitness)
    average_fitness_history.append(average_fitness)

    # Selection: Select top half of the population
    sorted_indices = np.argsort(fitness_scores)[::-1]
    population = [population[i] for i in sorted_indices[:population_size//2]]
    # Crossover and Mutation
    new_population = []
    while len(new_population) < population_size:
        parents = np.random.choice(population, 2, replace=False)
        child = {
            'w1': (parents[0]['w1'] + parents[1]['w1']) / 2,
            'w2': (parents[0]['w2'] + parents[1]['w2']) / 2
        }
        # Mutation
        if np.random.rand() < mutation_rate:
            child['w1'] += np.random.randn(*child['w1'].shape) * 0.1
            child['w2'] += np.random.randn(*child['w2'].shape) * 0.1
        new_population.append(child)
    population = new_population
    print(f"Generation {generation+1}, Best Fitness: {best_fitness:.4f}")

# Evaluate the best individual on validation set
best_individual = population[np.argmax(fitness_scores)]
predictions = forward_pass(X_val, best_individual['w1'], best_individual['w2'])
predictions = (predictions > 0.5).astype(int)
final_accuracy = accuracy_score(y_val, predictions)
print(f"Final Accuracy on Validation Set: {final_accuracy:.4f}")
# Plotting the results
plt.figure(figsize=(10, 5))
plt.plot(best_fitness_history, label='Best Fitness')
plt.plot(average_fitness_history, label='Average Fitness')
plt.title('Fitness Over Generations')
plt.xlabel('Generation')
plt.ylabel('Fitness')
plt.legend()
plt.grid(True)
plt.show()
```

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:

Code:

```
import random

# Define the function to optimize
def objective_function(position):
    x, y = position
    return x**2 + y**2

# PSO parameters
num_particles = 30
num_dimensions = 2
max_iterations = 10   # Changed from 100 to 10

w = 0.5
c1 = 1.5
c2 = 1.5

# Initialize particles
particles = []
velocities = []
personal_best_positions = []
personal_best_scores = []

for _ in range(num_particles):
    position = [random.uniform(-10, 10) for _ in range(num_dimensions)]
    velocity = [random.uniform(-1, 1) for _ in range(num_dimensions)]
    particles.append(position)
    velocities.append(velocity)
    personal_best_positions.append(position[:])
    personal_best_scores.append(objective_function(position))

global_best_index = personal_best_scores.index(min(personal_best_scores))
global_best_position = personal_best_positions[global_best_index][:]
global_best_score = personal_best_scores[global_best_index]

for iteration in range(max_iterations):
    for i in range(num_particles):
        for d in range(num_dimensions):
            r1 = random.random()
            r2 = random.random()

            velocities[i][d] = (w * velocities[i][d] +
                        c1 * r1 * (personal_best_positions[i][d] - particles[i][d]) +
                        c2 * r2 * (global_best_position[d] - particles[i][d]))

            particles[i][d] += velocities[i][d]
```

```python
        fitness = objective_function(particles[i])

        if fitness < personal_best_scores[i]:
            personal_best_positions[i] = particles[i][:]
            personal_best_scores[i] = fitness

            if fitness < global_best_score:
                global_best_position = particles[i][:]
                global_best_score = fitness

    print(f"Iteration {iteration+1}/{max_iterations} — Best Score: {global_best_score:.5f}")

print("\nBest solution found:")
print(f"Position: {global_best_position}")
print(f"Value: {global_best_score}")
```

Program 3:

Ant Colony Optimization for the Traveling Salesman Problem:

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.

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:
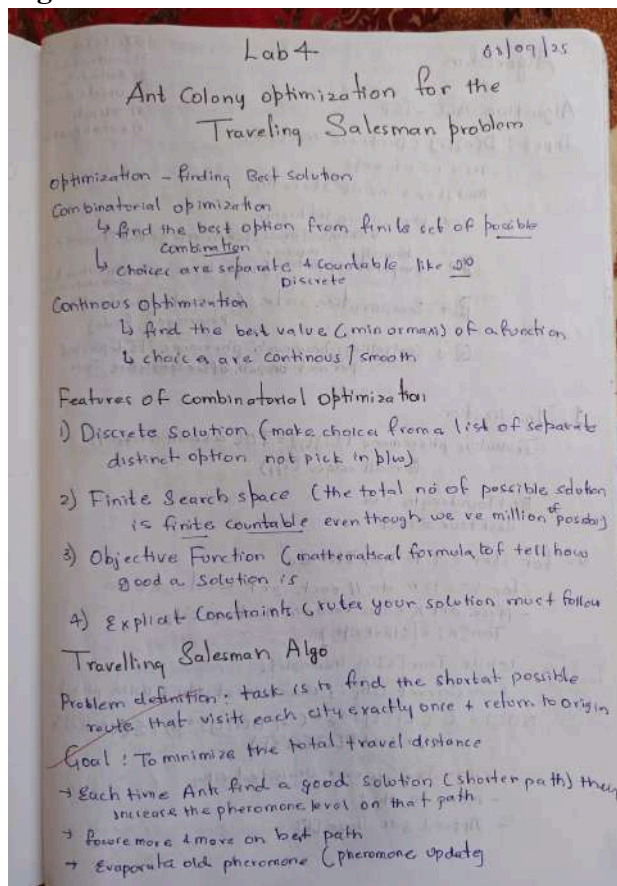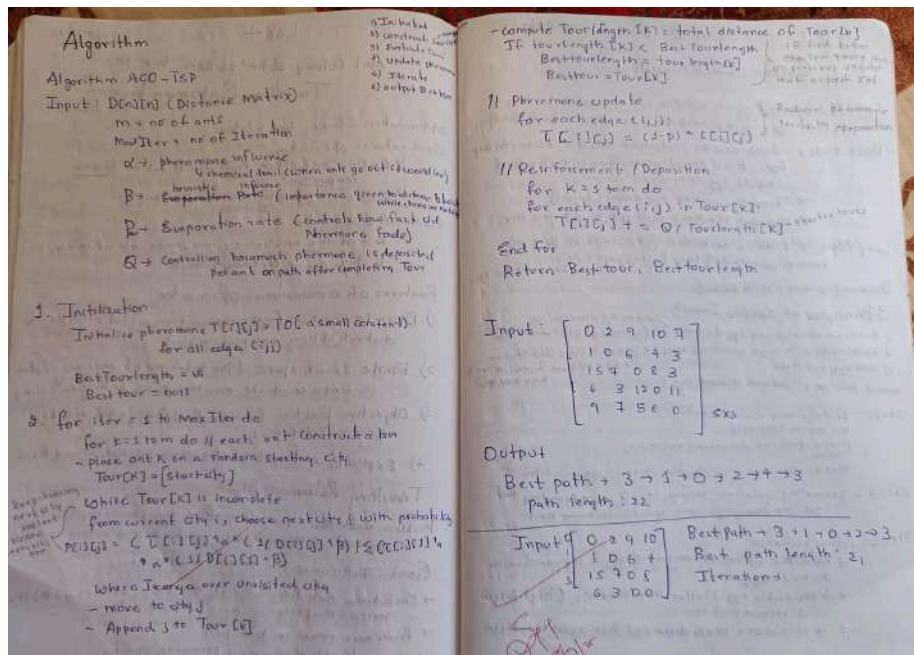
Code:

```python
import random
import math

class ACO_TSP:
    def __init__(self, distances, n_ants=10, n_iterations=100, alpha=1, beta=5, rho=0.5, Q=100):
        self.distances = distances
        self.n = len(distances)
        self.n_ants = n_ants
        self.n_iterations = n_iterations
        self.alpha = alpha
        self.beta = beta
        self.rho = rho
        self.Q = Q
        self.pheromone = [[1 for _ in range(self.n)] for _ in range(self.n)]

        self.best_length = float("inf")
        self.best_tour = None

    def run(self):
        for it in range(self.n_iterations):
            all_tours = []
            all_lengths = []

            for ant in range(self.n_ants):
                tour = self.construct_solution()
                length = self.compute_length(tour)
```

```python
            all_tours.append(tour)
            all_lengths.append(length)

            if length < self.best_length:
                self.best_length = length
                self.best_tour = tour

        self.update_pheromones(all_tours, all_lengths)

    return self.best_tour, self.best_length

def construct_solution(self):
    start = random.randint(0, self.n - 1)
    tour = [start]
    unvisited = set(range(self.n))
    unvisited.remove(start)

    current = start
    while unvisited:
        next_city = self.choose_next_city(current, unvisited)
        tour.append(next_city)
        unvisited.remove(next_city)
        current = next_city

    return tour

def choose_next_city(self, current, unvisited):
    probs = []
    total = 0
    for city in unvisited:
        tau = self.pheromone[current][city] ** self.alpha
        eta = (1.0 / self.distances[current][city]) ** self.beta
        value = tau * eta
        probs.append((city, value))
        total += value

    r = random.random() * total
    cumulative = 0
    for city, value in probs:
        cumulative += value
        if cumulative >= r:
            return city
    return probs[-1][0]

def compute_length(self, tour):
    length = 0
    for i in range(len(tour) - 1):
```

```python
            length += self.distances[tour[i]][tour[i+1]]
        length += self.distances[tour[-1]][tour[0]]
        return length

    def update_pheromones(self, all_tours, all_lengths):
        for i in range(self.n):
            for j in range(self.n):
                self.pheromone[i][j] *= (1 - self.rho)

        for tour, length in zip(all_tours, all_lengths):
            for i in range(len(tour) - 1):
                a, b = tour[i], tour[i+1]
                self.pheromone[a][b] += self.Q / length
                self.pheromone[b][a] += self.Q / length
            a, b = tour[-1], tour[0]
            self.pheromone[a][b] += self.Q / length
            self.pheromone[b][a] += self.Q / length


# Example usage
if __name__ == "__main__":
    distances = [
        [0, 2, 9, 10, 7],
        [1, 0, 6, 4, 3],
        [15, 7, 0, 8, 3],
        [6, 3, 12, 0, 11],
        [9, 7, 5, 6, 0]
    ]

    aco = ACO_TSP(distances, n_ants=10, n_iterations=50, alpha=1, beta=5, rho=0.5, Q=100)
    best_tour, best_length = aco.run()

    # Format tour as edges
    path_str = " -> ".join(map(str, best_tour)) + f" -> {best_tour[0]}"
    print("\n=== Final Best Solution ===")
    print("Best path:", path_str)
    print("Best path length:", best_length)
```
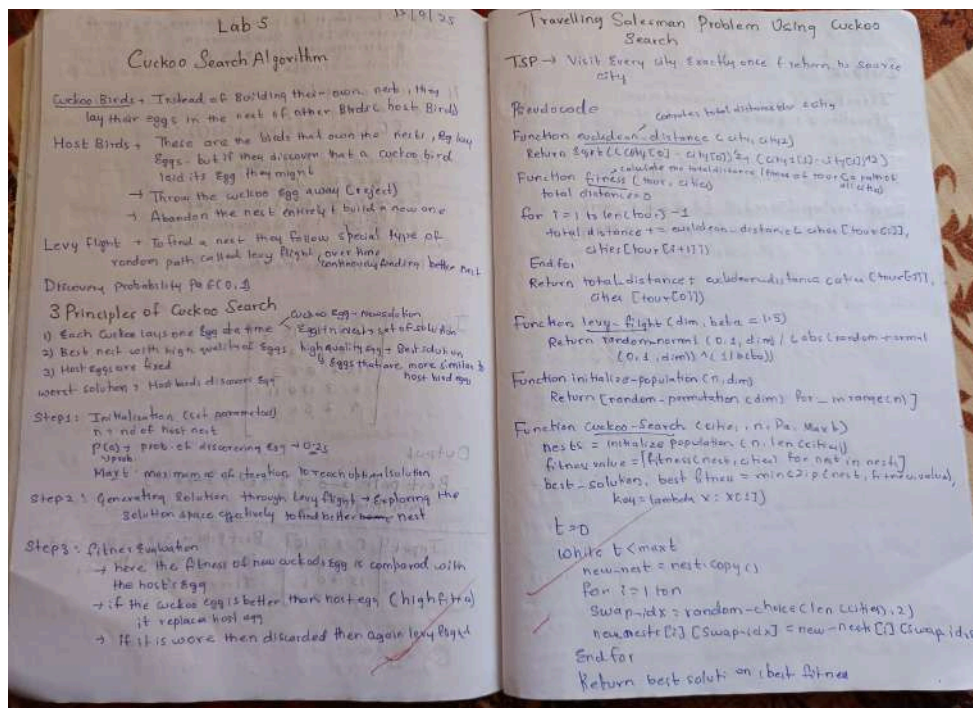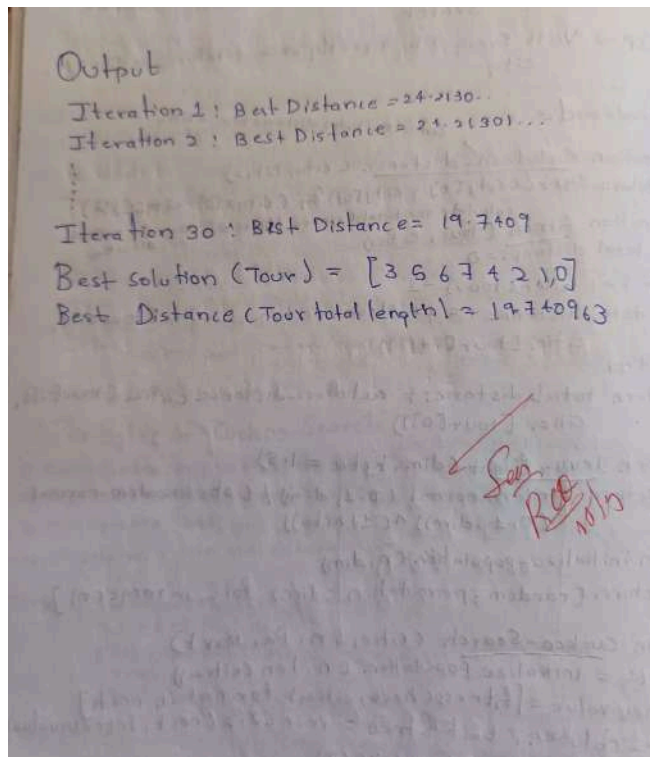
Program 4:

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:

Output

Iteration 1 : Best Distance = 24.2130.

Iteration 2 : Best Distance = 24.21301...

Iteration 30 : Best Distance = 19.7409

Best solution (Tour) = [3 5 6 7 4 2 1 0]

Best Distance (Tour total length) = 19.7409963

Code:

```python
import numpy as np

# Problem data (same as before)
weights = np.array([12, 7, 11, 8, 9])
values = np.array([24, 13, 23, 15, 16])
capacity = 26

n = 10      # Number of nests
Pa = 0.25    # Probability of abandoning worst nests
max_iter = 100

def fitness(solution):
    total_weight = np.sum(solution * weights)
    if total_weight > capacity:
        return 0
    else:
        return np.sum(solution * values)

def initial_nests(n, dim):
    return np.random.randint(0, 2, (n, dim))

def levy_flight(Lambda=1.5):
    sigma = (np.math.gamma(1 + Lambda) * np.sin(np.pi * Lambda / 2) /
        (np.math.gamma((1 + Lambda) / 2) * Lambda * 2 ** ((Lambda - 1) / 2))) ** (1 /
Lambda)
```

```python
    u = np.random.normal(0, sigma, size=weights.shape[0])
    v = np.random.normal(0, 1, size=weights.shape[0])
    step = u / np.abs(v) ** (1 / Lambda)
    return step

def get_new_solution(nest):
    step_size = levy_flight()
    new_sol_cont = nest + 0.01 * step_size * (nest - np.mean(nest))
    s = 1 / (1 + np.exp(-new_sol_cont))
    new_sol = np.array([1 if x > 0.5 else 0 for x in s])
    return new_sol

def abandon_worst_nests(nests, fitnesses, Pa):
    num_abandon = int(Pa * len(nests))
    worst_indices = np.argsort(fitnesses)[:num_abandon]
    for i in worst_indices:
        nests[i] = np.random.randint(0, 2, nests.shape[1])
        fitnesses[i] = fitness(nests[i])
    return nests, fitnesses

def cuckoo_search():
    dim = weights.shape[0]
    t = 0

    # Step 4 and 5: Initialize population and evaluate fitness
    nests = initial_nests(n, dim)
    fitnesses = np.array([fitness(nest) for nest in nests])

    while t < max_iter:
        for i in range(n):
            # Step 7 and 8: Generate cuckoo and evaluate fitness
            cuckoo = get_new_solution(nests[i])
            cuckoo_fit = fitness(cuckoo)

            # Step 9: Choose a nest randomly
            j = np.random.randint(n)

            # Step 10-12: Replace if cuckoo is better
            if cuckoo_fit > fitnesses[j]:
                nests[j] = cuckoo
                fitnesses[j] = cuckoo_fit

        # Step 13 and 14: Abandon fraction Pa of worst nests and build new ones
        nests, fitnesses = abandon_worst_nests(nests, fitnesses, Pa)

        # Step 15 and 16: Keep and rank the best solution
        best_index = np.argmax(fitnesses)
```

```python
        best_nest = nests[best_index].copy()
        best_fitness = fitnesses[best_index]

        print(f"Iteration {t+1}: Best fitness = {best_fitness}")

        t += 1

    # Step 19: Output the best solution
    return best_nest, best_fitness

best_solution, best_val = cuckoo_search()

print("\nBest solution found:")
print("Items selected:", best_solution)
print("Total value:", best_val)
print("Total weight:", np.sum(best_solution * weights))
```

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:

Code:

```python
import numpy as np
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import cross_val_score
from sklearn.ensemble import RandomForestClassifier

# Load dataset
data = load_breast_cancer()
X = data.data
y = data.target
num_features = X.shape[1]

# Gray Wolf Optimizer parameters
num_wolves = 10  # Population size
max_iter = 10    # Number of iterations

# Binary GWO helper functions
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def binary_transform(x):
    return np.where(sigmoid(x) > np.random.rand(len(x)), 1, 0)

# Fitness function: classification accuracy
def fitness(position):
    selected_features = np.where(position == 1)[0]
    if len(selected_features) == 0:
        return 0
    X_selected = X[:, selected_features]
    clf = RandomForestClassifier(n_estimators=50)
    score = cross_val_score(clf, X_selected, y, cv=5).mean()
    return score

# Initialize wolves
wolves = np.random.uniform(-1, 1, (num_wolves, num_features))
binary_wolves = np.array([binary_transform(w) for w in wolves])
fitness_vals = np.array([fitness(w) for w in binary_wolves])

# Initialize alpha, beta, delta
alpha_idx = np.argmax(fitness_vals)
alpha = wolves[alpha_idx].copy()
alpha_score = fitness_vals[alpha_idx]

beta_idx = np.argsort(fitness_vals)[-2]
beta = wolves[beta_idx].copy()
beta_score = fitness_vals[beta_idx]
```

```python
delta_idx = np.argsort(fitness_vals)[-3]
delta = wolves[delta_idx].copy()
delta_score = fitness_vals[delta_idx]

# Main loop
for t in range(max_iter):
    a = 2 - t * (2 / max_iter)  # Linearly decreasing a

    for i in range(num_wolves):
        for j in range(num_features):
            r1, r2 = np.random.rand(), np.random.rand()
            A1 = 2 * a * r1 - a
            C1 = 2 * r2
            D_alpha = abs(C1 * alpha[j] - wolves[i][j])
            X1 = alpha[j] - A1 * D_alpha

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

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

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

    # Update binary positions
    binary_wolves = np.array([binary_transform(w) for w in wolves])
    fitness_vals = np.array([fitness(w) for w in binary_wolves])

    # Update alpha, beta, delta
    sorted_idx = np.argsort(fitness_vals)[::-1]
    alpha, alpha_score = wolves[sorted_idx[0]].copy(), fitness_vals[sorted_idx[0]]
    beta, beta_score = wolves[sorted_idx[1]].copy(), fitness_vals[sorted_idx[1]]
    delta, delta_score = wolves[sorted_idx[2]].copy(), fitness_vals[sorted_idx[2]]

    print(f"Iteration {t+1}/{max_iter}, Best fitness: {alpha_score:.4f}")

# Best feature subset
best_features = np.where(binary_transform(alpha) == 1)[0]
print("Selected feature indices:", best_features)
print("Number of features selected:", len(best_features))
```

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:
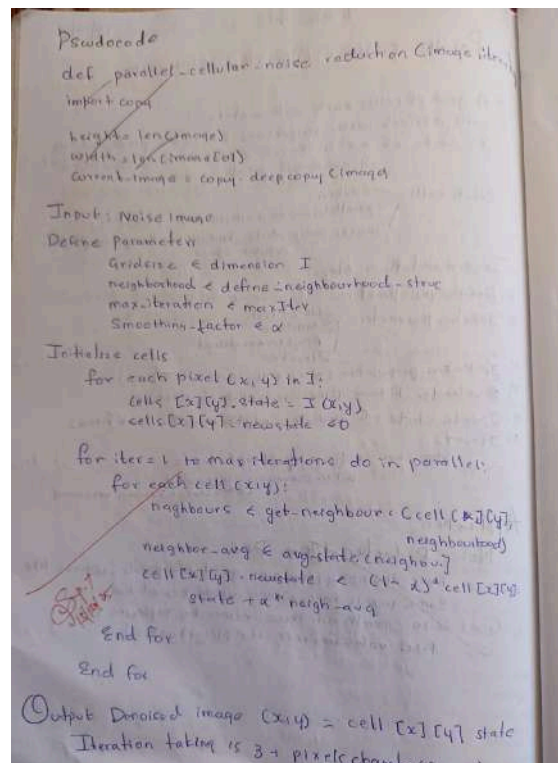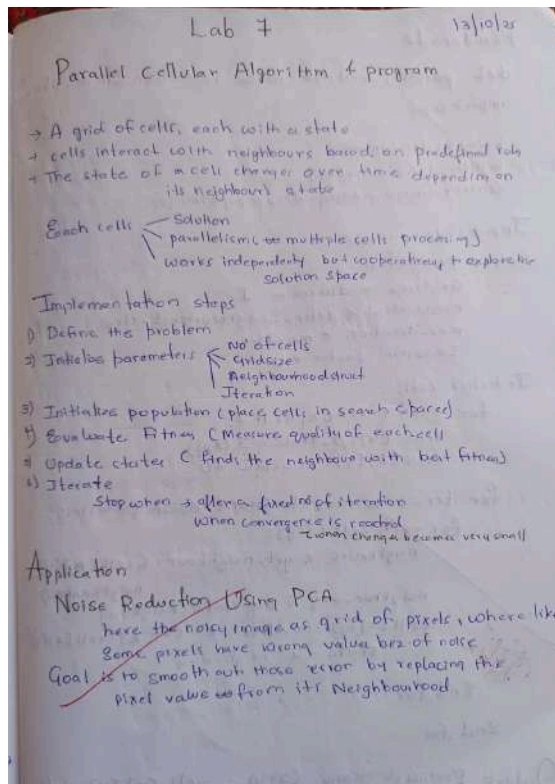
Code:

```python
import numpy as np
import matplotlib.pyplot as plt
from skimage import data, util

def get_neighbors_indices(row, col, max_row, max_col):
    neighbors = []
    for dr in [-1, 0, 1]:
        for dc in [-1, 0, 1]:
            if dr == 0 and dc == 0:
                continue
            nr, nc = row + dr, col + dc
            if 0 <= nr < max_row and 0 <= nc < max_col:
                neighbors.append((nr, nc))
    return neighbors

def pca_noise_reduction(image, max_iterations=10, sigma=15):
    rows, cols = image.shape
    denoised_image = image.copy().astype(float)

    for iteration in range(max_iterations):
        new_image = denoised_image.copy()

        for i in range(rows):
            for j in range(cols):
                neighbors = get_neighbors_indices(i, j, rows, cols)

                weights = []
                intensities = []

                for nr, nc in neighbors:
                    diff = abs(denoised_image[nr, nc] - denoised_image[i, j])
                    weight = np.exp(-diff / sigma)
                    weights.append(weight)
                    intensities.append(denoised_image[nr, nc])

                weights = np.array(weights)
                intensities = np.array(intensities)

                if weights.sum() > 0:
                    new_value = np.sum(weights * intensities) / np.sum(weights)
                    new_image[i, j] = new_value

        denoised_image = new_image

    return denoised_image.astype(np.uint8)
```

```python
# Load sample grayscale image: "camera"
image = data.camera()  # shape: (512, 512)

# Add salt & pepper noise
noisy_image = util.random_noise(image, mode='s&p', amount=0.05)
noisy_image = (noisy_image * 255).astype(np.uint8)

# Apply PCA-based denoising
denoised = pca_noise_reduction(noisy_image, max_iterations=10, sigma=20)

# 📌 Print only the 5×5 pixel values for comparison
print("\nOriginal Image 5x5 patch:")
print(image[100:105, 100:105])

print("\nNoisy Image 5x5 patch:")
print(noisy_image[100:105, 100:105])

print("\nDenoised Image 5x5 patch:")
print(denoised[100:105, 100:105])
```

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:

26
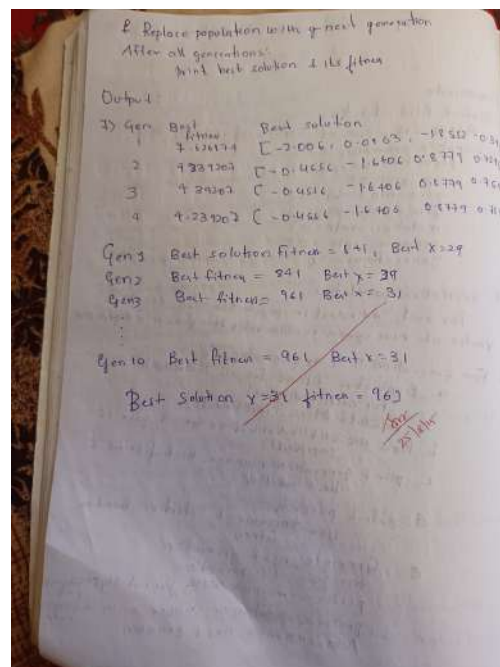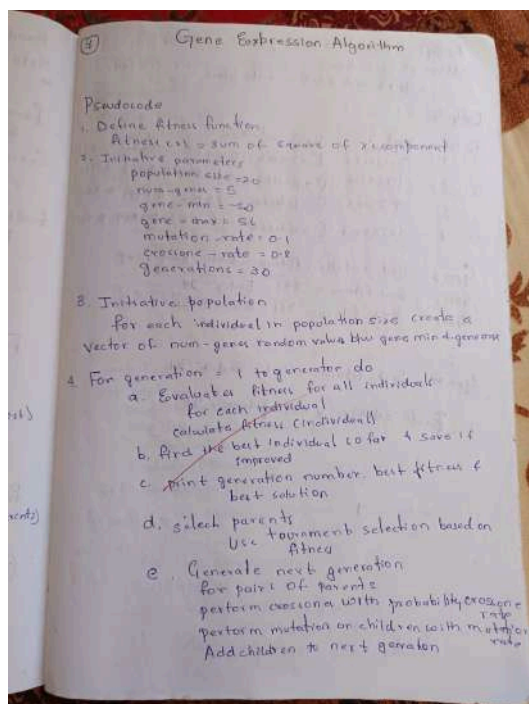
Code:

```python
import numpy as np
import matplotlib.pyplot as plt
from gplearn.genetic import SymbolicRegressor
from gplearn.functions import make_function
from gplearn.fitness import make_fitness

# Generate training data
X = np.linspace(-10, 10, 100).reshape(-1, 1)
y = X**2 + np.sin(X)  # True function to approximate

# Define symbolic regressor
est_gp = SymbolicRegressor(
    population_size=500,
    generations=20,
    stopping_criteria=0.01,
    p_crossover=0.7,
    p_subtree_mutation=0.1,
    p_hoist_mutation=0.05,
    p_point_mutation=0.1,
    max_samples=0.9,
    verbose=1,
    parsimony_coefficient=0.001,
    random_state=42
)

# Fit model
est_gp.fit(X, y)

# Predict on training data
y_pred = est_gp.predict(X)

# Print discovered expression
print("\nDiscovered expression:")
print(est_gp._program)

# Plot the results
plt.figure(figsize=(10, 6))
plt.plot(X, y, label='True Function', color='blue')
plt.plot(X, y_pred, label='GEP Prediction', color='red', linestyle='--')
plt.legend()
plt.title("Gene Expression Programming (Symbolic Regression)")
plt.xlabel("X")
plt.ylabel("y")
plt.grid(True)
plt.show()
```