VISVESVARAYA TECHNOLOGICAL UNIVERSITY

“JnanaSangama”, Belgaum -590014, Karnataka.

LAB RECORD

Bio Inspired Systems (23CS5BSBIS)

Submitted by

Hemanth Kumar R (1BM23CS110)

in partial fulfilment 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 Technologica1University, Belgaum)
Department of Computer Science and Engineering

CERTIFICATE

This is to certify that the Lab work entitled "Bio Inspired Systems (23CS5BSBIS)" carried out by Hemanth Kumar R (1BM23CS110), who is Bonafide student of B.M.S. College of Engineering. It is in partial fulfilment 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.

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

Index

GitHub Link:

https://github.com/HK9876/BIS_1BM23CS110

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.

Algorithm:

WordGA - Evolving a Word to Match a Target

Goal: Evolve a population of random strings to match a target string ("CODE"), using genetic Algorithm.

Algorithm

1. Initialisation

Begin
    set target = "CODE"
    set pop-size = 20
    set Mutation-rate = 0.1
    set genes = Length of Target
    set Char-Pool = ABCDE . . . Z"

Initialise popln with POP-SIZE random strings of length GENES

SET generation = 0

REPEAT
    SORT population by descending fitness
    SET best = first individual in population
    PRINT generation, best, fitness(best)

    IF fitness(best) = GENES THEN
        print "Found match!"
        EXIT LOOP
    ENDIF

    CREATE new Population as empty list
    For i from 1 TO POP SIZE DO
        SELECT parent 1 using tournament selection
        SELECT parent 2 using tournament selection
        SET child 1 = crossover(parent1, parent2)
        SET child = mutate(child)
        ADD child to new Population
    END FOR

```
import random
import math

NUM_CITIES = 10
POPULATION SIZE = 100
GENERATIONS  = 500
MUTATION_RATE = 0.1

cities = [(random.randint(0, 100), random.randint(0, 100)) for   in range(NUM CITIES)]

def distance(city1, city2):
    return math.sqrt((city1[0] - city2[0])**2 + (city1[1] - city2[1])**2)

def total_distance(route):
    dist = 0
    for i in range(len(route)):
        dist += distance(cities[route[i]], cities[route[(i + 1) % NUM CITIES]])
    return dist

def fitness(route):
    return 1 / total distance(route)

def generate_population():
    return [random.sample(range(NUM_CITIES), NUM_CITIES) for_ in range(POPULATION_SIZE)]

def selection(population, fitnesses):
    selected = random.choices(population, weights=fitnesses, k=POPULATION_SIZE)
    return selected

def crossover(parent1, parent2):
    start, end = sorted(random.sample(range(NUM CITIES), 2))
    child = [None] “ NUM_CITIES
    child[start:end] = parent1[start:end]
    pointer = 0
    for gene in parent2:
        if gene not in child:
            while child[pointer] is not None:
                pointer += 1
            child[pointer] = gene
    return child
def mutate(route):
    if random.random() < MUTATION RATE:
        i, j = random.sample(range(NUM CITIES), 2)
        route[i], routes] = routes], route[i]
    return route
```

```
def genetic algorithm():
    population = generate_population()
    best_route = None
    best_distance  = float('inf')

    for generation in range(GENERATIONS):
        fitnesses = [fitness(ind) for ind in population]
        new_population = g

        for    in range(POPULATION   SIZE):
            parent1, parent2 = selection(population,  fitnesses)[:2]
            child = crossover(parent1, parent2)
            child = mutate(child)
            new population.append(child)

        population = new_population

        for route in population:
            dist = total  distance(route)
            if dist < best_distance:
                best distance = dist
                best  route = route

        if generation % 50 == 0:
            print(f"Generation {generation}:  Best Distance = (round(best_distance,  2))")

        t("\n⊖ Final  Best Route:")
    print("Route:", best_route)
    print("Distance:", round(best  distance, 2))

genetic_algorithm()
```

Program 2: 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 pmblems are encoded in a manner similar to genetic sequences. The algorithm evolves these solutions through selection, crossover, mutation, and gene expression to find optimal or near-optimal solutions. GEA is effective for solving complex optimization problems in various domains, including engineering, data analysis, and machine learning.

Algorithm:

Lab 2 Gene Expression Alg for Travelling SalesMan Problem

Pseudo code

1. Initialise Parameters
   - Population size (pop-size)
   - Number of generations (max-gene)
   - Crossover rate (crossover_rate)
   - Mutation rate (mutation_rate)
   - Distance Matrix (distances)
2. Generate Initial Population
   - Create a random initial population of solutions (tours)
   - Each individual in the population is a possible soln (tour)
3. Evaluate Fitness
   - For each individuals in the population:
     - Calculate the st of the tour (fitness)
     - Fitness function: shorter distances = higher fitness
4. Repeat for max-generations:
   For each generation:
   5. Selection
      - Select a subset of individuals based on their fitness
      - Use a selection method
      - The fittest individuals (lower distance) have a higher chance of being selected.
   6. Cross over
      - For each pair of selected parents, perform cross-over:
        - Use a crossover method like OX
        - Create a new off spring by combining parts of both parents
   7. Mutation
      - For each individual, apply mutation with prob mutation
        - Randomly swap two cities in the tour (2 opt mutation)
   8. Evaluate fitness of new Population
      - Calculate the fitness for each individual in the new population
   9. Survival selection
      - Combine the parent population and offspring selection
   10. Output the Best Solution

END

Output:

Generation 0:

Tour: [1, 3, 2, 0] | Dis = 80

Tour: [1 2 3 0] | Dis: 95

Tour: [1 3 2 0] | Dis: 80

Tour: [1 0 2 3] | Dis: 80

Generation 1

Tour: [2, 3, 0, | Dis: 95

Tour: [3, 1, 2, 0] | Dis: 95

Tour: [1, 3, 2, 0] | Dis: 80

Tour: [1, 2, 3, 0] | Dis: 80

for 2 generations the min dist is 80

Code:

```
import random
import math

# Parameters
NUM CITIES = 10
POPULATION_SIZE = 100
GENERATIONS = 500
MUTATION RATE = 0.1

# Generate random cities
cities = [(random.randint(0, 100), random.randint(0, 100)) for _ in range(NUM_CITIES)]

def distance(city1, city2):
   return math.sqrt((city1[0] - city2[0])**2 + (city1[1] - city2[1])**2)

def total distance(route):
   dist = 0
   for i in range(len(route)):
      dist += distance(cities[route[i]], cities[route[(i + 1) % NUM  CITIES]])
   return dist

def fitness(route):
   return 1 / total distance(route)

def generate population():
   return [random.sample(range(NUM  CITIES), NUM  CITIES) for   in range(POPULATION  SIZE)]

def selection(population, fitnesses):
   selected = random.choices(population, weights=fitnesses, k=POPULATION_SIZE)
   return selected

def crossover(parent1, parent2):
   start, end = sorted(random.sample(range(NUM_CITIES), 2))
   child = [None] * NUM_CITIES
   child[start:end] = parent1[start:end]
   pointer = 0
   for gene in parent2:
      if gene not in child:
         while child[pointer] is not None:
            pointer += 1
         child[pointer] = gene
   return child
def mutate(route):
   if random.random() < MUTATION_RATE:
      i, j = random.sample(range(NUM  CITIES), 2)
      route[i], routes] = routes], route[i]
   return route
```

```
def genetic algorithm():
    population = generate_population()
    best_route = None
    best_distance = float('inf')

    for generation in range(GENERATIONS):
        fitnesses = [fitness(ind) for ind in population]
        new_population = g

        for   in range(POPULATION  SIZE):
            parent1, parent2 = selection(population, fitnesses)[:2]
            child = crossover(parent1, parent2)
            child = mutate(child)
            new population.append(child)

        population = new_population

        # Track best
        for route in population:
            dist = total_distance(route)
            if dist < best distance:
                best distance = dist
                best route = route

        if generation % 50 == 0:
            print(f"Generation {generation}: Best Distance = (round(best_distance, 2))")

    print("\nO Final Best Route:")
    print("Route:", best route)
    print("Distance:", round(best distance, 2))

genetic_algorithm()
```

Program 3: Particle Swarm Optimization for Function Optimization:

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

Algorithm:

Particle Swarm Optimisation → exploration / → exploitation

Algo:

1. Initialise n_particles, w, c1, c2, max_iterations.
2. for each particle:
   - Initialise pos and vel randomly
   - pBest ← current pos
3. Evaluate fitness for all Best
   - gbest ← pbest
4. For iter = 1 to max_iterations:
   - For each particle:
     - Evaluate fitness of current position:
     - If current fitness better than pbest:
       - pbest ← current pos
     - If pbest > gBest:
       - gBest ← pBest
   - For each particle:
     - update velocity:
       - velocity = w * velocity + c_1 * r_1 * (pBest − pos) + c_2 * r_2 * (gBest − pos)
     - Update position:
       - position = position + velocity.

c^2

Return gBest as the best solution found

Code:

```
import random
import numpy as np

del fitness  function(position):
    x, y = position
    return -(x**2 + y**2 - 4*x - 6*y)

def particle_swarm_optimization(dimensions,  num_particles, max_iterations, threshold):
    w = 0.5
    c1 = 1.2
    c2 = 1.4

    swarm = []
    for    in range(num   articles):
        position  = np.random.uniform(-10, 10, size=dimensions)
        velocity = np.random.uniform(-1, 1, size=dimensions)
        pbest_position = position.copy()
        pbest fitness = fitness function(position)
        swarm.append(('position“. position, 'velocity". velocity,
                      'pbest_position’: pbest_position, 'pbest_fitness': pbest_fitness})

    gbest_position = np.zeros(dimensions)
    gbest fitness = -float(’inf’)

    for i in range(max   iterations):
        for p in swarm:
            fitness = fitness function(p['position'])

            if fitness > p['pbest_fitness’]:
                p['pbest_fitness'] = fitness
                p[’pbest_position’] = p[’position’].copy()

            if fitness > gbest_fitness:
                gbest_fitness  = fitness
                gbest   osition  = p['position'].copy()

        if gbest fitness >= threshold:
            print(f"Early stopping at iteration {i}")
            break

        for p in swarm:
            rand1 = random.random()
            rand2 = random.random()

            inertia = w * p[’velocity’]
            cognitive = c1 * rand1 * (p['pbest  position’] - p['position’])
            social = c2 * rand2 ” (gbest_position - p[’position’])

            p['velocity']  = inertia  + cognitive  + social
```

```
        p['position'] = p['position'] + p['velocity']

    print("SOLUTION FOUND:")
    print(f" Position: {gbest_position)")
    print(f" Fitness: (gbest_fitness}")
    return gbest   osition, gbest fitness

particle swarm optimization(dimensions=2, num particles=20, max iterations=5000, threshold=2)
```

Program4: 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.

Algorithm:

Ant Colony Op[illegible]

procedure AntColonyOptimization
    Initialise Graph G with nodes A, B, C, D
    Initialise pheromone level on each edge to a small constant value.

    For iteration = 1 to MAX. ITERATIONS do
        Place all ants at node A
        // Step1: Each ant chooses a neighbouring node to move to
        Choose next node from A based on pheromone level and distance
        Move from A to chosen node
        Record edge travelled (A→chosen node

    End for
        // Step2: Deposit pheromone on travelled paths
        For each ant do
            edge = Path traveled from A to chosen node
            Add pheremone to edge based on quality of path
            // Example: shorter distance → more pheromone added
        End for

    //Step3: Update probabilities for next move
        For each edge from A do
            Compute probability based on pheromone level and distance
            //More pheromone → higher probability for next ant choices
        End for

    Step4: Evaporate pheromone on all edges
        For each edge do
            $\boxed{\text{pheromone} = \text{pheromone} \times (1 - \text{evaporation rate})}$
        End for

    // Step5: Optionally, find best path based on pheromone and distance
        Update best path if current paths are better.

    End for
    Output the best path found and its length.
End Procedure

Input:

A, B, C, D

A-B	2
A-C	3
A-D	4
B-C	1
B-D	3
C-D	2

Initial pheromone levels: 1.0
No. of ants: 4
Evaporation rate = 0.1
Alpha (pheromone) : 1
Beta (distance) : 2
Iteration : 5

O/P:

Best Path Found : A→B→C→D→A
Total distance : 9
Final pheromone level
A-B : 3.5
A-C : 1.2
A-D : 0.8
B-C : 2.8
B.D : 1.5
C.D : 2.5

Code:

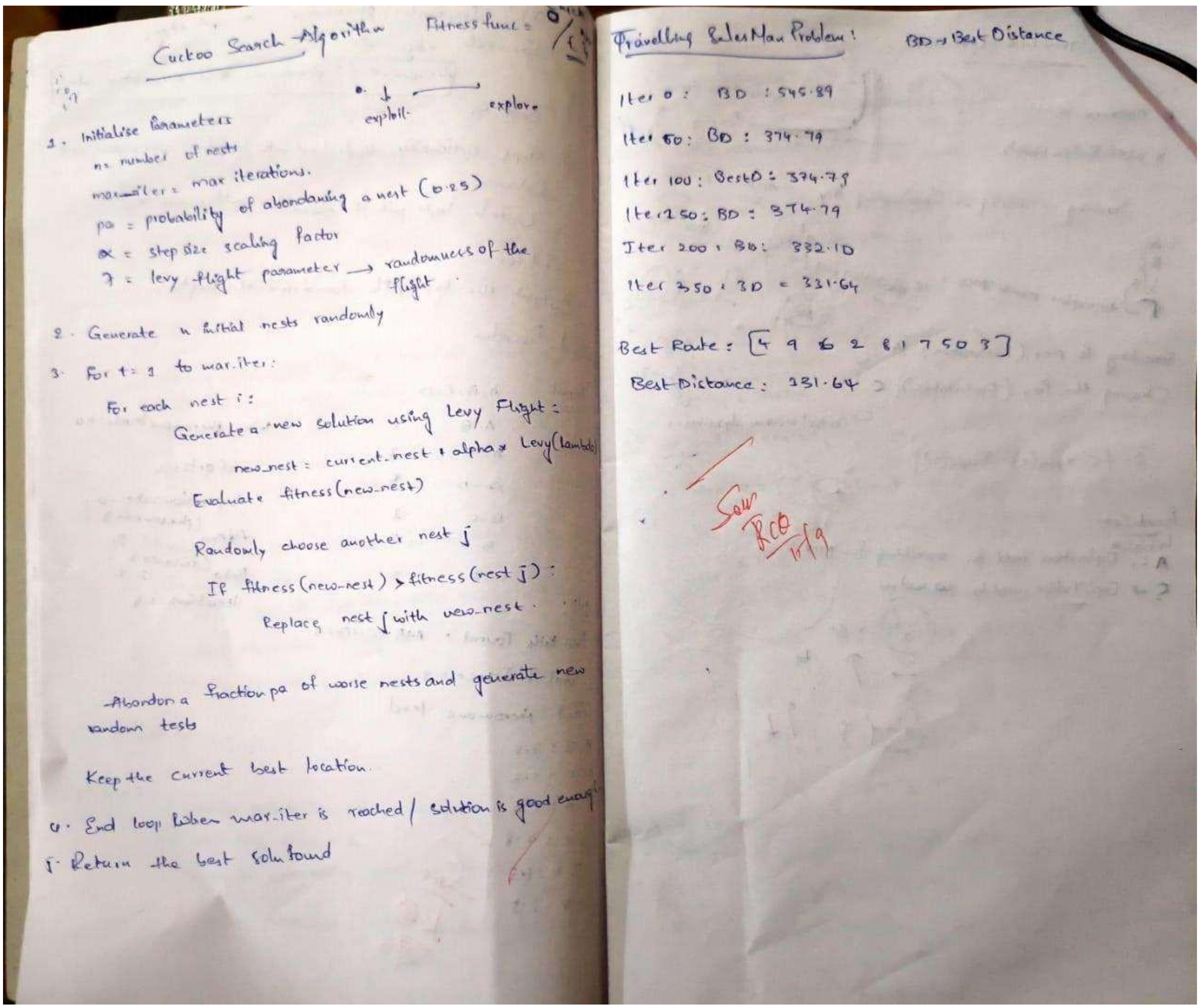
```
import numpy as np
import random


NUM_CITIES = 5
NUM_ANTS = 20
NUM_ITERATIONS = 100
ALPHA = 1.0
BETA = 5.0
RHO = 0.5
Q = 100


cities = np.random.rand(NUM_CITIES, 2)
distance_matrix = np.1ina1g.norm(cities[:, None] - cities, axis=2)
pheromone_matrix = np.ones((NUM_CITIES, NUM_CITIES))


def calculate    robabi1ities(current_city, visited):
    probabilities = []
    for next  city in range(NUM  CITIES):
        if next  city in visited:
            probabilities.append(0)
        else:
            pheromone = pheromone_matrix[current_city][next_city] ** ALPHA
            heuristic = (1  / distance_matrix[current_city][next_city]) ** BETA
            probabilities.append(pheromone * heuristic)
    total = sum(probabilities)
    return [p / total if total > 0 else 0 for p in probabilities]

def construct  tour():
    start_city= random.randint(0, NUM_CITIES - 1)
    tour = [start_city]
    while len(tour) < NUM  CITIES:
        probs = calculate   probabilities(tour[-l], tour)
        next_city = np.random.choice(range(NUM_CITIES), p     robs)
        tour.append(next_city)
    return tour

def compute  tour  length(tour):
    return sum(distance_matrix[tour[i]][tour[(i + 1) % NUM_CITIES]]  for i in range(NUM_CITIES))


best  tour = None
best  length = float('inf)

for iteration in range(NUM_ITERATIONS):
    all  tours = []

    for _  in range(NUM_ANTS):
```

```
        tour = construct_tour()
        length = compute_tour_length(tour)
        al1_tours.append((tour, length))

        if length < best_length:
            best_tour = tour
            best length = length


    pheromone  matrix *= (1 - RHO)


    for tour, length in al1_tours:
        for i in range(NUM_CITIES):
            a, b = tour[i], tour[(i + 1) % NUM_CITIES]
            pheromone_matrix[a][b] += Q / length
            pheromone  matrix[b][a] += Q / length # symmetric TSP


clean  tour= [int(city) for city in best  tour]
print("Best tour:", clean  tour)

print("Best length:", round(best_length, 4))
```

Program 5: 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 Levy flights to generate new solutions, promoting global search capabilities and avoiding local minima. The algorithm is widely used for solving continuous optimization problems and has applications in various domains, including engineering design, machine learning, and data mining.

Algorithm:



Cuckoo Search Algorithm

Fitness func =

exploit · explore

1. Initialise Parameters
   n = number of nests
   max_iter = max iterations.
   pa = probability of abondaning a nest (0.25)
   α = step size scaling factor
   λ = levy flight parameter → randomness of the flight

2. Generate n initial nests randomly

3. For t = 1 to max_iter:
   For each nest i:
      Generate a new solution using Levy Flight:
         new_nest = current_nest + alpha * Levy(lambda)
      Evaluate fitness(new_nest)
      Randomly choose another nest j
      If fitness(new_nest) > fitness(nest j):
         Replace nest j with new_nest.
   Abandon a fraction pa of worse nests and generate new random nests
   Keep the current best location.

4. End loop when max_iter is reached / solution is good enough

5. Return the best soln found

Travelling SalesMan Problem:

BD → Best Distance

Iter 0: BD: 545.89

Iter 50: BD: 374.79

Iter 100: BestD: 374.79

Iter 150: BD: 374.79

Iter 200: BD: 332.10

Iter 350: BD = 331.64

Best Route: [4 9 6 2 8 1 7 5 0 3]

Best Distance: 331.64

Code:

```
import numpy as np
import math

# --- Levy flight ---
del levy_flight(Lambda, dim):
    sigma = (math.gamma(1 + Lambda) * np.sin(np.pi * Lambda / 2) /
            (math.gamma((1 + Lambda) / 2) * Lambda * 2**((Lambda - 1) / 2)))**(1 / Lambda)
    u = np.random.normal(0, sigma, size=dim)
    v = np.random.normal(0, 1, size=dim)
    step = u / abs(v)””(1 / Lambda)
    return step

# --- Sigmoid for binary conversion ---
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

# --- Fitness function for knapsack ---
def fitness_function(x_bin, weights, values, capacity):
    total weight = np.sum(x bin * weights)
    total value = np.sum(x bin * values)
    if total weight > capacity:
        return -1 # Penalize overweight solutions heavily
    else:
        return total_value

# --- Cuckoo Search for Binary Knapsack ---
def cuckoo_search_knapsack(weights, values, capacity, n=25, Pa=0.25, Maxt=500):
    dim = len(weights)
    # Initialize nests (continuous vectors)
    nests = np.random.uniform(low=-1, high=1, size=(n, dim))
    # Convert to binary solutions
    nests_bin = np.array([sigmoid(nest) > np.random.rand(dim) for nest in nests])
    fitness = np.array([fitness function(x, weights, values, capacity) for x in nests bin])

    best_idx = np.argmax(fitness)
    best_nest = nests[best_idx].copy()
    best_bin = nests_bin[best_idx].copy()
    best fitness = fitness[best idx]

    t = 0
    while t < Maxt:
        for i in range(n):
            # Generate new solution by Levy flight
            step = levy_flight(1.5, dim)
            new_nest = nests[i] + 0.01 * step
            # Convert new_nest to binary
            new bin = sigmoid(new nest) > np.random.rand(dim)
            new_fitness = fitness_function(new_bin, weights, values, capacity)

            # If new solution is better, replace
```

```
            if new_fitness > fitness[i]:
                nests[i] = new_nest
                nests bin[i] = new bin
                fitness[i] = new_fitness

                if new fitness > best fitness:
                    best fitness = new fitness
                    best nest = new nest.copy()
                    best_bin = new_bin.copy()

        # Abandon fraction Pa of worst nests
        num_abandon = int(Pa * n)
        worst_indices = np.argsort(fitness)[:num_abandon]
        for idx in worst_indices:
            nests[idx] = np.random.uniform(-1, 1, dim)
            nests bin[idx] = sigmoid(nests[idx]) > np.random.rand(dim)
            fitness[idx] = fitness function(nests bin[idx], weights, values, capacity)

            if fitness[idx] > best_fitness:
                best fitness = fitness[idx]
                best nest = nests[idx].copy()
                best_bin = nests_bin[idx].copy()

        t +—— 1

    return best bin, best fitness

if name      == "main '.
  "print("Enter the number of items:")
  n items = int(input())

    weights = g
    values = g

    print("Enter the weights of the items (space-separated):")
    weights = np.array(list(map(float, input().split())))
    if len(weights) != n_items:
        raise ValueError("Number of weights does not match number of items.")

    print("Enter the values of the items (space-separated):")
    values = np.array(list(map(float, input().split())))
    if len(values) != n_items:
        raise ValueError("Number of values does not match number of items.")

    print("Enter the knapsack capacity:")
    capacity = float(input())

    print("Enter population size (default 25):")
    n = input()
    n = int(n) if n.strip() else 25

    print("Enter abandonment probability Pa (default 0.25):")
```

```
    Pa = input()
    Pa = float(Pa) if Pa.strip() else 0.25

    print("Enter maximum iterations Maxt (default 500):")
    Maxt = input()
    Maxt = int(Maxt) if Maxt.strip() else 500

    best solution, best value = cuckoo search knapsack(weights, values, capacity, n=n, Pa=Pa,
Maxt=Maxt)

    print("\nBest solution (items selected):", best_solution.astype(int))
    print("Total value:", best_value)
    print("Total weight:", np.sum(best_solution * weights))
```

Program 6: 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:

1) Initialise population of wolves (x_i) randomly
- n = no. of wolves
- d = no. of dimensions (variables)

2) Set initial parameters.
- a = 2 (explo factor)
- Max_iterations = max number of iterations
- Initialise random co-efficients A and C

3) Evaluate fitness of each wolf and identify.
- X_α = best wolf (α)
- X_β = ~~sec~~ beta
- X_δ = delta

4) For each iteration t=1 to Max_it;
- a = 2 * (1 - t / Max_iterations)
- For each wolf i in the population:
  - A = 2 * r_1 - a (~~…~~
  - C = 2 * r_2
  - Calculate the distance between the wolf and the 3 best wolves
    - $D_\alpha = |C_1 * X_\alpha - X_i|$
    - $D_\beta = |C_2 * X_\beta - X_i|$
    - $D_\delta = |C_3 * X_\delta - X_i|$
  - Update the pos of wolf:
    - $X_1 = X_\alpha - A_1 * D_\alpha$
    - $X_2 = X_\beta - A_2 * D_\beta$
    - $X_3 = X_\delta - A_3 * D_\delta$
    - $X_i = (X_1 + X_2 + X_3)/3$ average pos
  - Update fitness of all wolves after position change
  - Reassign X_α, X_β, X_δ based on new fitness values

5) After all iterations, return X_α as the best soln found.

Output:

Iteration 1, Best Distance: 10.0

Iter 2: Best Distance: 9.8

:

Iteration 100; 8.0

Best Route: [0, 1, 2, 3]

Best Distance = 8.0

Code:

```
import numpy as np
import random

def distance  matrix(cities):
   n = len(cities)
      dist = np.zeros((n,  n)) for i in range(n):  for j in range(n):
         dist[i]§]  = np.linalg.norm(np.array(cities[i]) - np.array(cities§]))
   return dist

def tour  length(tour,  dist):
   return sum(dist[tour[i]][tour[(i+1)%len(tour)]]  for i in range(len(tour)))

def initialize_population(num_wolves, num_cities):
   return [random.sample(range(num_cities), num_cities)  for _ in range(num_wolves)]

def gwo tsp(cities, num  wolves=20, max  iter=100):
   dist = distance_matrix(cities)
   population  = initialize_population(num_wolves, len(cities))
   fitness = [tour  length(tour,  dist) for tour in population]

   alpha,  beta,  delta  = sorted(zip(population,  fitness), key=lambda  x: x[1])[:3]

   for iter in range(max_iter):
      a = 2 - iter * (2 / max_iter)
      new_population = g

      for wolf in population:
         new_tour = []
         for i in range(len(cities)):
            r1, r2 = random.random(), random.random()
            A1 = 2 * a ” r1 - a
            C1 = 2 * r2
            D alpha = abs(C1 * alpha[0][i] - wolf[i])
            X1 = alpha[0][i]  - A1 * D_alpha

            # Repeat  for beta and delta
            # Combine X1, X2, X3 and discretize
            new  tour.append(int(X1)  % len(cities))

         # Ensure it's a valid permutation
         new  tour = list(dict.fromkeys(new  tour))
         while len(new  tour) < len(cities):
            new  tour.append(random.choice([i for i in range(len(cities))  if i not in new  tour]))

         new_population.append(new_tour)

      population  = new  population
      fitness = [tour_length(tour,  dist) for tour in population]
      alpha,  beta,  delta = sorted(zip(population,  fitness), key=lambda  x: x[1])[:3]
```

```
    return alpha[0], alpha[1]

# Example usage
cities = [(0,0), (1,5), (5,2), (6,6), (8,3)]
    best_tour,
    best distance =
    gwo tsp(cities)
print("Best tour:", best tour)
print("Distance:", best_distance)
```

<u>Program 7:</u> Parallel Cellular Algorithm for Routing :

It 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 PCA algorithm using Python to optimize a mathematical function.

Algorithm:

1. Parallel Cellular Alg for Routing

Find eff routes for a set of vehicles delivering goods to multiple locations minimising total travel distance

1. Define the problem:
   - Objective: Minimise total dis travelled by all vehicles.
   - Fitness fn: Total route length
2. Initialise Parameters:
   - Grid size: 2D grid of cells.
   - Cells: Each cell hold a set of routes.
   - Iterations: Max no of iterations.
3. Initialise Population:
   - Randomly generate routes for each vehicle in each cell.
4. Evaluate fitness:
   - For each cell, calculate total distance travell by all vehicles
   - Assign fitness based on route efficiency
5. Update states
   - For each cell:
     - a. Check neighbors' fitness
     - b. Identify neighbor with the shortest routes
     - c. Update own route, by incorporating parts from best neighbor's routes
     - d. Optionally mutate routes by randomly changing order
6. Iterate
   - Repeat evaluation & updating until max iteration/ Convergence
7. Output best solution:
   - Report the routing soln with the shortest total distance.

Code:

```
import numpy as np
from multiprocessing import Pool
from PIL import Image

# Load image and convert to grayscale
def load_image(path):
    img = Image.open(path).convert('L')  # 'L' mode = grayscale
    return np.array(img)

# Edge detection rule for a single pixel
def detect edge(args):
    grid, x, y, threshold = args
    rows, cols = grid.shape
    center = grid[x][y]
    for dx in [-1, 0, 1]:
        for dy in [-1, 0, 1]:
            if dx == 0 and dy == 0:
                continue
            nx, ny = x + dx, y + dy
            if 0 <= nx < rows and 0 <= ny < cols:
                if abs(int(center) - int(grid[nx][ny])) > threshold:
                    return 255  # Edge
    return 0  # Non-edge

# Parallel cellular edge detection
def parallel_edge_detection(image, threshold=20):
    rows, cols = image.shape
    args = [(image, x, y, threshold) for x in range(rows) for y in range(cols)]
    with Pool() as pool:
        edges = pool.map(detect edge, args)
    return np.array(edges).reshape((rows, cols))

# Save or display result
def save edge image(edge array, output path='edges.png'):
    edge_img = Image.fromarray(edge_array.astype(np.uint8))
    edge_img.save(output_path)
    edge_img.show()

# Example usage
if  name  == ' main ':
    image = load image('your image.jpg') # Replace with actual image path
    edges = parallel edge detection(image, threshold=30)
    save edge image(edges)
```