

VISVESVARAYA TECHNOLOGICAL UNIVERSITY

“JnanaSangama”, Belgaum -590014, Karnataka.



LAB RECORD

Bio Inspired Systems (23CS5BSBIS)

Submitted by

Bhoomi Suresh Kota (1BM23CS065)

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



CERTIFICATE

This is to certify that the Lab work entitled “ Bio Inspired Systems (23CS5BSBIS)” carried out by **Bhoomi Suresh Kota (1BM23CS065)**, who is a 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.

Dr Raghavendra C K Associate Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE
--	---

Index

Sl. No.	Date	Experiment Title	Page No.
1	18-8-'25	Program 1 - Genetic Algorithm	4
2	25-8-'25	Program 2 - Gene Expression Algorithm	6
3	1-9-'25	Program 3 - Particle Swarm Algorithm	9
4	8-9-'25	Program 4 - Ant Colony Algorithm	12
5	15-9-'25	Program 5 - Cuckoo Search Algorithm	15
6	29-9-'25	Program 6 - Grey Wolf Optimisation Algorithm	17
7	13-10-'25	Program 7 - Parallel Cellular Algorithm	19

Github Link:

<https://github.com/BhoomiSuresh/BIS-LAB.git>

Program 1

To find the shortest route visiting all the cities exactly once and returning to start, using Genetic algorithm

Algorithm:

	1	25	2510	21280	...	10110
	11000	0	252	...	10110	10110
<u>21 Pseudocode:</u>						
pop-size ← 4	Initial population					
chrom-length ← 5	Population					
generations ← 3	Loop					
P01	11	10110	1	10110	1	10110
P02	fitness(x)	00011	(10110)	11001	2	11001
P03	return x ³	11011	3	11011	3	11011
P04	decode(chromosome)	10001	4	11001	4	11001
P05	return bin-to-int(chromosome)					
PS1						
init-pop()						
for i ← 1 to pop-size						
Chromosome ← random 5-bit string						

```

ADD chromosome To population
endfor
return population
selection(pop)
fits ← [fitness(decode(c)) FOR c IN population]
total_fit ← sum(fits)
probs ← [f / total_fit FOR f IN fits]
new_pop ← roulette_selection(population, probs, pop-size)
return new_pop
crossover(parents) : parents = (2510) 10110, 11001
offspring ← []
FOR i ← 0 to pop-size - 1:
    p1 ← parents[i]
    p2 ← parents[i+1]
    cp ← random(1, chrom_length-1)
    o1 ← p1[0:cp] + p2[cp:1]
    o2 ← p2[0:cp] + p1[cp:1]
    ADD o1, o2 To offspring
ENDFOR
return offspring

mutation(pop)
mutated ← []
FOR each chrom in pop:
    if random() < 0.2:
        pos ← random(0, chrom_length-1)
        chrom[pos] ← flip_bit(chrom[pos])
    endif
    add chrom to mutated

```

Date:	10/10/2019
# Main	Genetic algorithm for Traveling Salesman Problem
pop ← init-pop()	Initial population
FOR gen ← 1 to generations	Loop
population ← selection(pop)	Selection
pop ← crossover(pop)	Crossover
pop ← mutation(pop)	Mutation
best ← argmax(pop, fitness(decode(chrom)))	Best solution
PRINT best, decode(best), fitness(decode(best))	Print results
Output: Generation 1: ['00011', '10111', '10101', '10001']	Generation 1
Generation 2: ['10111', '10101', '10100', '10111']	Generation 2
Generation 3: ['10100', '10100', '10111', '10101']	Generation 3
Best solution: 10111, x=23, fitness = 12167	Best solution
ADD o1, o2 To mutated	Add mutated
ENDFOR	End loop
return mutated	Return mutated

Code:

```
import random
import math

# Parameters
POP_SIZE = 20
MUTATION_RATE = 0.1
GENERATIONS = 4
X_MIN, X_MAX = 0, 10

# Fitness function
def fitness(x):
    return math.sin(x) * x

# Create initial population (list of real numbers)
def initial_population():
    return [random.uniform(X_MIN, X_MAX) for _ in range(POP_SIZE)]

# Tournament selection
def select(population):
    contenders = random.sample(population, 3)
    return max(contenders, key=fitness)

# Crossover (average)
def crossover(p1, p2):
    return (p1 + p2) / 2

# Mutation (small random change)
def mutate(x):
    if random.random() < MUTATION_RATE:
        x += random.uniform(-0.5, 0.5)
        x = max(min(x, X_MAX), X_MIN)
    return x

# Genetic Algorithm
def genetic_algorithm():
    population = initial_population()

    for generation in range(GENERATIONS):
        new_population = []

        for _ in range(POP_SIZE):
            parent1 = select(population)
            parent2 = select(population)
            child = crossover(parent1, parent2)
            child = mutate(child)
            new_population.append(child)

    return new_population
```

```

population = new_population
best = max(population, key=fitness)
print(f"Gen {generation}: Best x = {best:.4f}, f(x) = {fitness(best):.4f}")

return best

# Run
best_solution = genetic_algorithm()
print("\nBest solution found:")
print(f"x = {best_solution:.4f}")
print(f"f(x) = {fitness(best_solution):.4f}")

```

Program 2

To find the shortest route visiting all the cities exactly once and returning to start, using Gene Expression algorithm

Algorithm:

$[3] \text{sq}^*(x_0 - 1) + [3] \text{lg}^*x_0 = [2] \text{lab}_1$ $[3] \text{lg}^*(x_0 - 1) + \text{LAB}_2 * x_0 = [2] \text{lab}_2$ <p>Optimization via Gene Expression Algorithms</p> <p>input: city-coords, pop-size, generations, mutation-rate, crossover-rate</p> <p>start_mutation > crossover + i</p> <p>express(genotype):</p> <p>return order of city indices by ascending gene values</p> <p>tour.length(tour):</p> <p>total = 0</p> <p>for i in 0..n-cities-1: total += dist(tour[i], tour[(i+1) mod n-cities])</p> <p>return total</p> <p>fitness(genotype):</p> <p>tour = EXPRESS(genotype)</p> <p>length = TOUR.LENGTH(tour)</p> <p>return 1 / (1 + length), length, tour, qog, cmin</p> <p>init-population():</p> <p>return list of pop-size random real-valued vectors</p> <p>(qog, lab1, lab2, mutation_rate = 1, size = n-cities)</p> <p>(qog, lab1, lab2, mutation_rate = 1)</p> <p>tournament-select(pop, fitnesses, k):</p> <p>choose k random individuals</p> <p>return one with best fitness</p> <p>crossover(p1, p2):</p> <p>child1, child2 = copies of p1, p2</p> <p>if random < crossover_rate:</p> <p>for i in each gene:</p> <p>if random < 0.5:</p> <p>$\alpha = \text{random}(0, 1)$</p>	$\text{child1}[i] = \alpha * p1[i] + (1 - \alpha) * p2[i]$ $\text{child2}[i] = \alpha * p2[i] + (1 - \alpha) * p1[i]$ <p>return child1, child2</p> <p>mutate(genotype):</p> <p>for each gene:</p> <p>if random < mutation_rate:</p> <p>gene += Gaussian-noise</p> <p>return genotype</p> <p># Main</p> <p>pop = init-population()</p> <p>best = None</p> <p>for gen in 1..Generations:</p> <p>evaluate fitness for each individual in pop</p> <p>update best if a shortest tour is found</p> <p>new.pop[gen] = tournament-select(pop)</p> <p>add best individual (elitism)</p> <p>while size(new.pop) < pop_size:</p> <p>p1 = tournament-select(pop)</p> <p>p2 = tournament-select(pop)</p> <p>c1, c2 = crossover(p1, p2)</p> <p>c1 = mutate(c1)</p> <p>c2 = mutate(c2)</p> <p>add c1, c2 to new.pop</p> <p>pop = new.pop</p> <p>OUTPUT best-tour and best-length</p>
--	---

Code:

```
import random
import math

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

def tour_length(tour, cities):
    total = 0
    for i in range(len(tour)):
        total += distance(cities[tour[i]], cities[tour[(i+1) % len(tour)]])

    return total

# Step 2: Parameters
POP_SIZE = 50
N_GENES = 6
N_GENERATIONS = 100
MUT_RATE = 0.2
CROSS_RATE = 0.7

# Step 3: Cities (coordinates)
cities = [(random.randint(0, 100), random.randint(0, 100)) for _ in range(N_GENES)]

# Step 4: Initialize population (random tours)
def create_individual():
    tour = list(range(N_GENES))
    random.shuffle(tour)
    return tour

population = [create_individual() for _ in range(POP_SIZE)]

# Step 5: Evaluate fitness
def fitness(ind):
    return 1 / (1 + tour_length(ind, cities))

# Step 6: Selection (roulette wheel)
def selection(pop):
    weights = [fitness(ind) for ind in pop]
    return random.choices(pop, weights=weights, k=2)

# Step 7: Crossover (Order Crossover for TSP)
def crossover(parent1, parent2):
    if random.random() > CROSS_RATE:
        return parent1[:]

    start, end = sorted(random.sample(range(N_GENES), 2))
    child = [None]*N_GENES
    child[start:end] = parent1[start:end]
```

```

pos = end
for gene in parent2:
    if gene not in child:
        if pos >= N_GENES: pos = 0
        child[pos] = gene
        pos += 1
return child

# Step 8: Mutation (swap two cities)
def mutate(ind):
    if random.random() < MUT_RATE:
        i, j = random.sample(range(N_GENES), 2)
        ind[i], ind[j] = ind[j], ind[i]

# Step 9: Gene Expression Algorithm loop
best = None
for gen in range(N_GENERATIONS):
    new_population = []
    for _ in range(POP_SIZE):
        p1, p2 = selection(population)
        child = crossover(p1, p2)
        mutate(child)
        new_population.append(child)
    population = new_population

    # Track best solution
    current_best = min(population, key=lambda ind: tour_length(ind, cities))
    if best is None or tour_length(current_best, cities) < tour_length(best, cities):
        best = current_best

# Step 10: Output best solution
print("Cities (coordinates):")
for i, c in enumerate(cities):
    print(f'City {i}: {c}')

print("\nBest tour order (by city indices):", best)
print("Best tour length:", tour_length(best, cities))

print("\nBest tour path (with coordinates):")
for idx in best:
    print(f'City {idx} -> {cities[idx]}')
print(f'Back to start -> {cities[best[0]]}')

```

Program 3

Implement Particle Swarm Optimization (PSO) to find the minimum of a simple two-dimensional objective function: $f(x, y) = x^2 + y^2$.

Algorithm:

LAB 3: Particle Swarm Optimization	
Date: 01/08/2023	
Parameters:	
f: Objective function	$f(x, y) = x^2 + y^2$
x_i : Position of particles or agents	
v_i : Velocity - "velocity of agents" or "agent's speed"	
A: Population of particles or agents	10 particles
w: Inertia weight	0.9
c_1 : Cognitive constant	2.0
r_1, r_2 : Random numbers	
c_2 : Social constant	2.0
Steps:	
① Create a population of agents uniformly distributed over X	
② Evaluate each particle's position	$y = f(x) = -x^2 + 5x + 20$
③ If posn is better than prvn posn, update it	
④ Determine the best particle	
⑤ $v_i^{t+1} = v_i^t + c_1 u_i^t (p_{best}^t - p_i^t) + c_2 u_s^t (g_{best}^t - p_i^t)$	
⑥ Update velocity	
⑦ Go to step 2	

Fitness function: $f(x, y) = x^2 + y^2$
De Jong function: $f(x) = x^2 + y^2$
or any other function $w = []$ psolav
(1) $m1 = []$ eq. 3.6.1.1
Inertia weight: $0.4 + 0.95 \cdot t$
$c_1 \times c_2 : [1.5 - 2.0]$
$r_1, r_2 : [0.1 - 1.0] \text{ random}$
$\Delta x = []$ m1
$\Delta y = []$ m1
Eg: $f(x) = x^2 + y^2$
Pseudocode:
Initialize: $w \leftarrow 0.5$, $c_1 \leftarrow 1.5$, $c_2 \leftarrow 1.5$, $m1 \leftarrow []$, $m2 \leftarrow []$, $num_particles \leftarrow 3$, $num_iterations \leftarrow 2$
for each particle i in $[1..num_particles]$:
position[i] = (x_i, y_i)
velocity[i] = $(0, 0)$
personal_best_pos[i] = position[i]
personal_best_val[i] = $f(position[i])$
global_best_pos = position of particle with minimum $personal_best_val$
global_best_val = minimum $personal_best_val$

Page No.:	Date:	YOUVA
-----------	-------	-------

```

for iter in [1..num_iterations]:
    for each particle i in [1..num_particles]:
        Generate random r1, r2 in [0,1] prob 30
        velocity[i] = w* velocity[i] + c1*r1*o
        (personal_best_pos[i] - position[i])
        + c2 * r2 * (global_best_pos - position[i])
        position[i] = position[i] + velocity[i]
        current_val = f(position[i]) = position[i] * x^2
        + position[i] * y^2
        if current_val < personal_best_val[i]:
            personal_best_val[i] = current_val
            personal_best_pos[i] = position[i]
        find particle j with minimum personal_best_val[j]
        if personal_best_val[j] < global_best_val:
            global_best_val = personal_best_val[j]
            global_best_pos = personal_best_pos[j]
        print iteration number, all particles positions,
        velocities, and global best
        (x,y) = [i] position
        (0,0) = [i] position
    
```

Code:

```
import numpy as np
```

```
# Objective function
```

```
def f(position):
```

```
    x, y = position
```

```
    return x**2 + y**2
```

```
# Parameters
```

```
w = 0.5 # inertia weight
```

```
c1 = 1.5 # cognitive coefficient
```

```
c2 = 1.5 # social coefficient
```

```
num_particles = 3
```

```
num_iterations = 2
```

```
dim = 2 # dimensions: x and y
```

```
# Initialize particles' positions and velocities (as per example)
```

```
positions = np.array([[2.0, 2.0],
```

```
[-3.0, -1.0],
```

```
[1.0, -4.0]])
```

```
velocities = np.zeros((num_particles, dim))
```

```
personal_best_positions = positions.copy()
```

```
personal_best_values = np.array([f(pos) for pos in positions])
```

```

# Initialize global best
best_idx = np.argmin(personal_best_values)
global_best_position = personal_best_positions[best_idx].copy()
global_best_value = personal_best_values[best_idx]

print(f"Initial global best: position={global_best_position}, value={global_best_value}\n")

for iter in range(1, num_iterations + 1):
    print(f"Iteration {iter}:")

    for i in range(num_particles):
        r1, r2 = np.random.rand(2)

        # Update velocity
        cognitive = c1 * r1 * (personal_best_positions[i] - positions[i])
        social = c2 * r2 * (global_best_position - positions[i])
        velocities[i] = w * velocities[i] + cognitive + social

        positions[i] = positions[i] + velocities[i]

        fitness = f(positions[i])

        if fitness < personal_best_values[i]:
            personal_best_values[i] = fitness
            personal_best_positions[i] = positions[i].copy()

    best_idx = np.argmin(personal_best_values)
    if personal_best_values[best_idx] < global_best_value:
        global_best_value = personal_best_values[best_idx]
        global_best_position = personal_best_positions[best_idx].copy()

    for i in range(num_particles):
        print(f" Particle {i+1}: position={positions[i]}, velocity={velocities[i]},\nfitness={f(positions[i])}")

    print(f" Global best position: {global_best_position}, value: {global_best_value}\n")

```

Program 4

To find the shortest route visiting all the cities exactly once and returning to start, using Ant Colony Optimisation algorithm.

Algorithm:

LAB 4 ANT COLONY OPTIMISATION	
Date:	YOUVA
• Parameters:	(α, β, Q) : ant dist ρ : deposit rate
α - Pheromone importance factor β - Heuristic importance factor ρ - Pheromone evaporation rate Q - Pheromone deposit factor n - Number of ants m - Number of iterations	
• Pseudocode:	
1. Initialize: - Set parameters: $n, \alpha, \beta, \rho, Q, m$ - Initialize pheromone on all edges to Q - Calculate heuristic information (η) as $1/\text{dist between cities}$	
2. For each iteration: a. For each ant: - Construct a tour starting from a random city - Choose next city based on probability (pheromone $\cdot \alpha * \text{heuristic} \cdot \beta$) - Complete the tour	
b. Evaluate tours: - Calculate total distance of each ant's tour - Keep track of the best tour found so far	
c. Update pheromones: - Evaporate all pheromones by $(1 - \rho)$ - Deposit new pheromones based on tour quality eq: pheromone += $Q / \text{tour_length}$	
3. Output the best tour & its total distance	

Code:

```
import numpy as np
import random
```

```
class ACO_TSP:
    def __init__(self, distances, n_ants=10, n_iterations=100, alpha=1.0, beta=5.0, rho=0.5, Q=100):
        self.distances = distances
        self.n_cities = 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 = np.ones((self.n_cities, self.n_cities)) # initial pheromone

self.heuristic = 1 / (distances + np.eye(self.n_cities)) # avoid divide-by-zero on diagonal
np.fill_diagonal(self.heuristic, 0)

def run(self):
    best_length = float('inf')
    best_tour = []

    for _ in range(self.n_iterations):
        all_tours = []
        all_lengths = []

        for _ in range(self.n_ants):
            tour = self.construct_solution()
            length = self.tour_length(tour)
            all_tours.append(tour)
            all_lengths.append(length)

            if length < best_length:
                best_length = length
                best_tour = tour

        self.update_pheromones(all_tours, all_lengths)

    return best_tour, best_length

def construct_solution(self):
    tour = []
    visited = set()
    current_city = random.randint(0, self.n_cities - 1)
    tour.append(current_city)
    visited.add(current_city)

    while len(tour) < self.n_cities:
        probs = []
        for city in range(self.n_cities):
            if city not in visited:
                pher = self.pheromone[current_city][city] ** self.alpha
                heur = self.heuristic[current_city][city] ** self.beta
                probs.append((city, pher * heur))

        total = sum(p for _, p in probs)
        r = random.uniform(0, total)
        s = 0
        for city, p in probs:
            s += p
            if s >= r:
                next_city = city
                break

    return tour

```

```

tour.append(next_city)
visited.add(next_city)
current_city = next_city

return tour

def tour_length(self, tour):
    return sum(self.distances[tour[i]][tour[(i + 1) % self.n_cities]] for i in range(self.n_cities))

def update_pheromones(self, all_tours, all_lengths):
    self.pheromone *= (1 - self.rho) # Evaporation
    for tour, length in zip(all_tours, all_lengths):
        for i in range(len(tour)):
            a = tour[i]
            b = tour[(i + 1) % len(tour)]
            self.pheromone[a][b] += self.Q / length
            self.pheromone[b][a] += self.Q / length # symmetric

# Example usage:
if __name__ == "__main__":
    distance_matrix = np.array([
        [0, 2, 9, 10],
        [1, 0, 6, 4],
        [15, 7, 0, 8],
        [6, 3, 12, 0]
    ])

    aco = ACO_TSP(distance_matrix, n_ants=10, n_iterations=100)
    best_tour, best_length = aco.run()
    print("Best Tour:", best_tour)
    print("Best Length:", best_length)

```

Program 5

To find the shortest route visiting all the cities exactly once and returning to start, using the Cuckoo Search algorithm.

Algorithm:

<p style="text-align: right;">M T W T F S S Page No.: _____ Date: _____ YOUVA</p> <p>LAB 5 CUCKOO SEARCH <small>method frogini</small> : (random, task, nest) task, Nest, task : (i, i+1) task, (i, i+1) nest, task Input: Distance matrix 'D', number of nests 'n', discovery rate pa, max_iterations Output: Best tour and its distance</p> <p>1. Initialize population of n nests (random permutations of cities) 2. Evaluate fitness (tour length) of each nest (city) 3. Find the current best solution [i, i+1] nest, task 4. Repeat until max iterations: For each nest (city) i: (i) Generate new solution using Lévy flight (permutation) (ii) Evaluate fitness of new solution (iii) If new solution is better, replace nest i else (if (random.uniform(0, 1)) < pa): A fraction (pa) of worst nests are abandoned and replaced with new random solutions Update the best solution found best_tour = worst_tour, worst_fitness 5. Return the best tour and its tour length</p> <p>Output: Distance matrix : [[0, 2, 9, 10, 7], [2, 0, 6, 4, 3], [9, 6, 0, 8, 5], [10, 4, 8, 0, 6], [7, 8, 5, 6, 0]] Best tour found: [2 0 1 3 4] Tour length: 26</p>	<p>Pseudocode:</p> <pre> import random def total_dist(tour, dist_matrix): return sum(dist_matrix[tour[i]][tour[(i+1) % len(tour)]] for i in range(len(tour)-1)) + dist_matrix[tour[-1], tour[0]] def levy_flight(tour): i, j = sorted(random.sample(range(len(tour)), 2)) new_tour = tour[:i] + tour[i:j][::-1] + tour[j:] new_tour[i:j] = reversed(new_tour[i:j]) return new_tour def cuckoo_search_top(dist_matrix, n_nests=25, pa=0.25, max_iter=1000): n = len(dist_matrix) nests = [random.sample(range(n), n)] for _ in range(n_nests): fitness = [total_dist(t, dist_matrix) for t in nests] best_idx = fitness.index(min(fitness)) best_tour, best_fitness = nests[best_idx] if random.uniform(0, 1) < pa: nests[best_idx] = random.sample(range(n), n) else: new_tour = levy_flight(nests[best_idx]) new_fitness = total_dist(new_tour, dist_matrix) if new_fitness < fitness[best_idx]: nests[best_idx], fitness[best_idx] = new_tour, new_fitness if new_fitness < best_fitness: best_tour, best_fitness = new_tour, new_fitness return best_tour </pre>
---	--

Code:

```

import numpy as np
import random

def tour_length(tour, dist_matrix):
    #Calculate total distance of a TSP tour.
    return sum(dist_matrix[tour[i], tour[(i+1) % len(tour)]] for i in range(len(tour)))

```

```

def levy_flight(Lambda=1.5):
    #Generate step size using Lévy distribution.
    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)
    u = np.random.normal(0, sigma)

```

```

v = np.random.normal(0, 1)
step = u / abs(v)**(1/Lambda)
return step

def cuckoo_search_tsp(dist_matrix, n=20, pa=0.25, max_iter=500):
    num_cities = len(dist_matrix)

    # Initialize nests (random tours)
    nests = [random.sample(range(num_cities), num_cities) for _ in range(n)]
    fitness = [tour_length(t, dist_matrix) for t in nests]

    best_tour = nests[np.argmin(fitness)]
    best_fit = min(fitness)

    for _ in range(max_iter):
        for i in range(n):
            # Generate new solution by applying a random swap (levy inspired)
            new_tour = nests[i][:]
            step = int(abs(levy_flight()) * num_cities) % num_cities
            if step > 1:
                a, b = random.sample(range(num_cities), 2)
                new_tour[a], new_tour[b] = new_tour[b], new_tour[a]

            new_fit = tour_length(new_tour, dist_matrix)

            # Replace if better
            if new_fit < fitness[i]:
                nests[i], fitness[i] = new_tour, new_fit

            # Update best
            if new_fit < best_fit:
                best_tour, best_fit = new_tour, new_fit

    # Abandon some nests
    for i in range(n):
        if random.random() < pa:
            nests[i] = random.sample(range(num_cities), num_cities)
            fitness[i] = tour_length(nests[i], dist_matrix)

    return best_tour, best_fit

# Example usage
if __name__ == "__main__":
    # Distance matrix for 5 cities (symmetric TSP)
    dist_matrix = np.array([
        [0, 2, 9, 10, 7],
        [2, 0, 6, 4, 3],
        [9, 6, 0, 8, 5],

```

[10, 4, 8, 0, 6],
[7, 3, 5, 6, 0]
])

```
best_tour, best_length = cuckoo_search_tsp(dist_matrix, n=15, pa=0.3, max_iter=200)
print("Best tour found:", best_tour)
print("Tour length:", best_length)
```

Program 6

To schedule n jobs to m machines, using the Grey Wolf Optimiser algorithm.

Algorithm:

Page No.: Date: YOVA
<p>LAB 6 \rightarrow 10 - 15 * 5 * 5 = 8A</p> <p>GREY WOLF OPTIMIZATION \rightarrow 8D</p> <p>$(C1)x = [1]x[job] + 85 \rightarrow job = 10x[job] - 85$</p> <p>PSEUDOCODE: INPUT num-jobs, num-machines, jobs[], max-iter, pack-size FUNCTION fitness(schedule): machine-times = array of zeros (size=nummachines) FOR j = 0 TO num-jobs - 1: m = schedule[j] \rightarrow machine at j-th position machine-times[m] = machine-times[m] + jobs[j] RETURN max(machine-times) </p> <p>FUNCTION GWoL(): schedule \rightarrow sorted job to wolf map wolves = random integer arrays (packsize x num-jobs) in [0, num-machines - 1] fitness_vals = [fitness(w) FOR each w in wolves]</p> <p>SORT wolves by fitness \rightarrow alpha, beta, delta FOR t = 1 to max_iter: FOR i = 1 to pack_size: FOR j = 1 to num-jobs: r1, r2 = random(), random() \rightarrow wolf A1 = 2*a*r1 + a \rightarrow alpha[i] = alpha[i] + r1*(x[i] - alpha[i]) C1 = r1 + 2*r2 \rightarrow beta[i] = beta[i] + r2*(x[i] - beta[i]) D_alpha = abs(C1*alpha[i] - x[i]) x1 = alpha[i] - A1*D_alpha r1, r2 = random(), random() A2 = 2*a*r1 + a C2 = 2*r2 D_beta = abs(C2*beta[i] - x[i]) x2 = beta[i] - A2*D_beta </p>

<p>r1, r2 = random(), random()</p> <p>A3 = 2*a*r1 + a \rightarrow 8A</p> <p>C3 = 2*r2 \rightarrow 10 * 10 * 5 * 5 = 8A</p> <p>D_delta = abs(C3*delta[i] - x[i])</p> <p>x3 = delta[i] - A3*D_delta</p> <p>#new position : (alpha[i]) \rightarrow (alpha[i] + r1*(x1+x2+x3)/3) \rightarrow 8A</p> <p>#discretize to machines [1] \rightarrow 10x[job] - 85</p> <p>edge_wolves[i] = round(x[i] CLIPPED to [0, num-machines]) \rightarrow 8A</p> <p>fitness_vals = [fitness(w) FOR each w in wolves]</p> <p>SORT wolves by fitness \rightarrow alpha, beta, delta \rightarrow 8A</p> <p>alpha[i] = alpha[i] + r1*(x1+x2+x3)/3 \rightarrow 8A</p> <p>RETURN alpha, fitness(alpha) \rightarrow 8A</p> <p>Best wolf in w \rightarrow job at (w) \rightarrow 8A</p> <p>OUTPUT: sorted schedule \rightarrow sorted job to wolf map Jobs: [8 3 6 2 7 5] \rightarrow job \rightarrow machine at 1 = 4 8A Best schedule (job \rightarrow machine): [0 1 2 0 2 1] \rightarrow 8A Best completion time: 133 \rightarrow job at 1 = 8A [0] \rightarrow job at 1 = X</p> <p>ALGORITHM PARAMETERS: n: num of jobs at t = 1, 8A wolves: population \rightarrow 10x[job] = 8A max_iter: no. of iterations \rightarrow 5 * 5 = 1A a, p, s, w: best, second best, third best, fourth ([i]x - [i]alpha) rest of the population $\vec{D} = [\vec{x}_p(t) - \vec{x}_s(t)] - [i]alpha = 8A$ $\vec{x}_{(t+1)} = \vec{x}_p(t) - A \cdot \vec{D}$ (job at 0) \rightarrow 10x[job] = 8A $2 \cdot r2 \rightarrow$ const BN 0.2 * 10 * 5 * 5 = 8A C3 = 8A $([i]x - [i]alpha) \cdot C3 \rightarrow$ 8A alpha[i] = 10x[job] - 85 </p>
--

Code:

```
import numpy as np

# ---- Problem Setup ----
num_jobs = 6
num_machines = 3
jobs = np.random.randint(1, 10, size=num_jobs) # processing times

def fitness(schedule):
    """Compute makespan for a given schedule (list of machine assignments)."""
    machine_times = [0] * num_machines
    for j, m in enumerate(schedule):
        machine_times[m] += jobs[j]
    return max(machine_times) # makespan

# ---- GWO Algorithm ----
def gwo(max_iter=30, pack_size=10):
    # Initialize wolves (random schedules)
    wolves = [np.random.randint(0, num_machines, size=num_jobs) for _ in range(pack_size)]
    fitness_vals = [fitness(w) for w in wolves]

    # Identify alpha, beta, delta
    alpha, beta, delta = np.argsort(fitness_vals)[:3]
    alpha_wolf, beta_wolf, delta_wolf = wolves[alpha], wolves[beta], wolves[delta]

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

        for i in range(pack_size):
            X = wolves[i].copy().astype(float)

            for j in range(num_jobs):
                r1, r2 = np.random.rand(), np.random.rand()

                # Distances from alpha, beta, delta
                A1, C1 = 2*a*r1 - a, 2*r2
                D_alpha = abs(C1*alpha_wolf[j] - X[j])
                X1 = alpha_wolf[j] - A1*D_alpha

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

                r1, r2 = np.random.rand(), np.random.rand()
                A3, C3 = 2*a*r1 - a, 2*r2
                D_delta = abs(C3*delta_wolf[j] - X[j])
                X3 = delta_wolf[j] - A3*D_delta
```

```

 $X[j] = (X1 + X2 + X3) / 3$ 

wolves[i] = np.clip(np.round(X), 0, num_machines-1).astype(int)

# Re-evaluate fitness
fitness_vals = [fitness(w) for w in wolves]
alpha, beta, delta = np.argsort(fitness_vals)[:3]
alpha_wolf, beta_wolf, delta_wolf = wolves[alpha], wolves[beta], wolves[delta]

return alpha_wolf, fitness(alpha_wolf)

best_schedule, best_makespan = gwo()
print("Jobs:", jobs)
print("Best Schedule (job → machine):", best_schedule)
print("Best Makespan:", best_makespan)

```

Program 7

Implement a Parallel Cellular Algorithm to compute the shortest distance from a source cell (top-left corner) to all other cells in a 2D grid using uniform edge costs.

Algorithm:

LAB-7
PARALLEL CELLULAR

- PARAMETERS
 - 1) Number of cells: $n = \text{width} \times \text{height}$
 - 2) Grid size: width, height
 - 3) Neighbourhood Structure
 - 4) Iterations: maxiter
- PSEUDOCODE:


```

Initialize grid of size width x height
cells ← INF
source cell distance ← 0

def neighbours(y, x):
    return list of valid neighbour coordinates (up, down, left, right)

repeat for a fixed number of steps or until no changes:
    create a copy of the grid called new_grid

    For each cell (y, x) in the grid:
        For each neighbour (ny, nx) of (y, x):
            cost = 1
            new_dist = grid[ny][nx] + cost
            If new_dist < new_grid[y][x]
                new_grid[y][x] = new_dist
            Update grid = new_grid

End repeat
Result ← grid with shortest distance from source to each cell
            
```

Code:

```
import numpy as np

WIDTH, HEIGHT = 10, 10
INF = 9999

def init_grid():
    grid = np.full((HEIGHT, WIDTH), INF)
    # Source at top-left corner (0,0)
    grid[0,0] = 0
    return grid

def neighbors(y, x):
    # 4-neighborhood (up, down, left, right)
    for ny, nx in [(y-1,x), (y+1,x), (y,x-1), (y,x+1)]:
        if 0 <= ny < HEIGHT and 0 <= nx < WIDTH:
            yield ny, nx

def update_distances(grid):
    new_grid = grid.copy()
    for y in range(HEIGHT):
        for x in range(WIDTH):
            for ny, nx in neighbors(y, x):
                cost = 1 # uniform cost
                new_dist = grid[ny, nx] + cost
                if new_dist < new_grid[y, x]:
                    new_grid[y, x] = new_dist
    return new_grid

def print_grid(grid):
    for row in grid:
        print(''.join(f'{int(x):2d}' if x != INF else '∞' for x in row))
    print()

# Main loop
grid = init_grid()
print("Initial distances:")
print_grid(grid)

for step in range(15):
    grid = update_distances(grid)
    print(f"After step {step+1}:")
    print_grid(grid)
```