VISVESVARAYA TECHNOLOGICAL UNIVERSITY

“JnanaSangama”, Belgaum -590014, Karnataka.

LAB RECORD

Bio Inspired Systems (23CS5BSBIS)

Submitted by

Shreyas Gowda C (1BM23CS319)

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 **Shreyas Gowda C (1BM23CS319),** 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.

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

Index

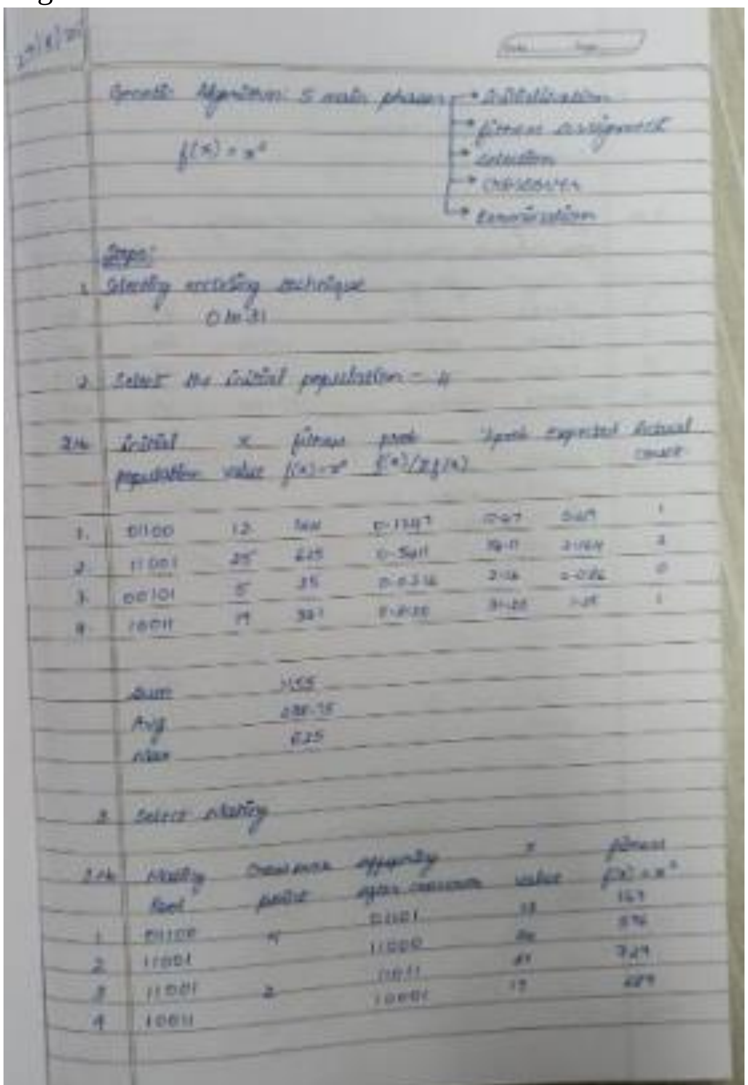
Github Link:
https://github.com/shreyasgowdac-319/1BM23CS319-BIS-LAB

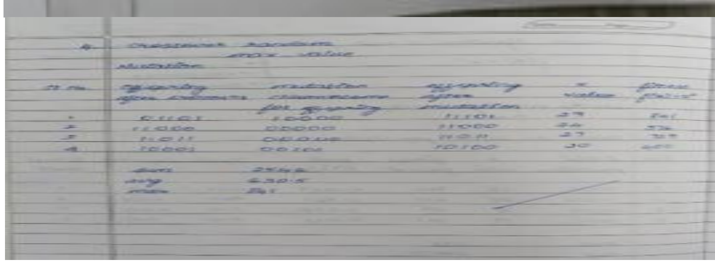
Program 1

Genetic Algorithm for Optimization Problems

We have a set of jobs that must be completed and a limited amount of resources available to perform them. The challenge is to determine how to assign each job to the available resources in a way that minimizes total completion time, reduces overall cost, or maximizes efficiency. The goal is to find an optimal scheduling strategy under these constraints.

Algorithm:





Code:

```
import random

jobs = [3, 2, 7, 5, 9, 4] # processing times of jobs
num_jobs = len(jobs)
population_size = 20
generations = 100
crossover_rate = 0.8
mutation_rate = 0.2

#
# Fitness Function (Makespan)
#
def fitness(chromosome):
    time = 0
    for job in chromosome:
        time += jobs[job]
    return 1 / time # smaller time → higher fitness

def initial_population():
    population = []
    for _ in range(population_size):
        chromosome = list(range(num_jobs))
        random.shuffle(chromosome)
        population.append(chromosome)
    return population
def selection(population):
    contenders = random.sample(population, 3)
    contenders.sort(key=lambda chromo: fitness(chromo), reverse=True)
    return contenders[0]
def crossover(p1, p2):
    if random.random() < crossover_rate:
        a, b = sorted(random.sample(range(num_jobs), 2))
        child = [-1] * num_jobs
        child[a:b] = p1[a:b]
        fill = [x for x in p2 if x not in child]
        j = 0
        for i in range(num_jobs):
            if child[i] == -1:
                child[i] = fill[j]
                j += 1
        return child
    return p1[:] # no crossover → copy parent
def mutate(chromosome):
    if random.random() < mutation_rate:
        a, b = random.sample(range(num_jobs), 2)
        chromosome[a], chromosome[b] = chromosome[b], chromosome[a]
    return chromosome
population = initial_population()
best_solution = None
best_fit = -1

for gen in range(generations):
```

```
    new_pop = []
    for _ in range(population_size):
        parent1 = selection(population)
        parent2 = selection(population)
        child = crossover(parent1, parent2)
        child = mutate(child)
        new_pop.append(child)

    population = new_pop

    # Track best
    for chromo in population:
        fit = fitness(chromo)
        if fit > best_fit:
            best_fit = fit
            best_solution = chromo
print("Best Job Order:", best_solution)
print("Job Times:", [jobs[j] for j in best_solution])
print("Total Completion Time (Makespan):", sum(jobs[j] for j in best_solution))
```

Output:

```
Best Job Order: [3, 4, 0, 2, 1, 5]
Job Times: [5, 9, 3, 7, 2, 4]
Total Completion Time (Makespan): 30
```

Program 2

Optimization via Gene Expression Algorithms

The Travelling Salesman Problem (TSP) asks for the shortest possible route that visits a given set of cities exactly once and returns to the starting city. The provided text describes using a Genetic Algorithm to solve this by evolving city sequences (chromosomes) through selection, crossover, and mutation to minimize the total tour distance.

Algorithm:

Lab - 7

Gene expression algorithm

Step 1: Fitness function : $f(x) = x^2$
Encoding technique : 0 to 31
use chromosome of fixed length (genotype)

Step 2: Initial population

S.no.	(Genotype) Initial chrom	Phenotype (expression)	x value	fitness	P
1	+2x	x^2	12	144	0.1247
2	+xx	2x	25	625	0.5411
3	x	x	5	25	0.0216
4	-2x	x-2	19	361	0.3125

sum 1155
avg 288.75
max 625

actual count	expected count
1	0.5
2	2.1
0	0.09
1	1.25

Step 3: Selection of mating pool

S.no.	Selected Chromo	crossover point	offspring	phenotype	x value	fitness
1	+2x	2	+x+	[illegible]	13	169
2	+xx	1	+2x	2x	26	676
3	+2x	2	+x-	x+(x-)	27	729
4	-2x	1	+2x	x+2	17	289

Step 4:
Mutation : perform mutation randomly chosen gene position (at each bit)
with fitness after mutation : 729

Step 5: Mutation

S.no.	offspring before mutation	mutation applied	offspring after mut	phenotype	x value	fitness
1	+x+	[illegible]	+x+	x+(x-)	27	[illegible]
2	+2x	[illegible]	+2x	2x	24	576
3	+x-	[illegible]	-x+	$x+x^2$	27	729
4	+2x	[illegible]	+x2	x+2	20	400

Step 6: Gene expression & Evaluation
decode each genotype → phenotype
calculate fitness

$\Sigma f(x) = [illegible] + 576 + 729 + 400 = [illegible]$
avg = [illegible]
max = [illegible]

Step 7: Iterate until convergence
Repeat step 2 to 6 until fitness improvement is negligible or generation limit has reached.

Pseudocode

Define fitness function
Define parameters
General population
select mating pool
mutation after mating
Gene expression & evaluation
Iterate
Output best value.

Output: 1000 generations

Genes : [29.53, 29.82, 29.8[illegible], 28.5[illegible], 15.09, [illegible], 28.83, 30.81, 28.51, 26.22]
x : 26.37
f(x) = 695.45

Code:

```
import random
import math

#
# Problem: TSP cities
#
cities = [(0,0), (1,5), (5,2), (6,6), (8,3)] # coordinates
num_cities = len(cities)

# Parameters
population_size = 30
generations = 200
crossover_rate = 0.8
mutation_rate = 0.2

#
# Distance Function
#
def distance(a, b):
    return math.sqrt((a[0]-b[0])**2 + (a[1]-b[1])**2)

def tour_length(chromosome):
    length = 0
    for i in range(num_cities):
        length += distance(cities[chromosome[i]], cities[chromosome[(i+1)%num_cities]])
return length

#
# Fitness Function
#
def fitness(chromosome):
    return 1 / tour_length(chromosome)

def initial_population():
    population = []
    for _ in range(population_size):
        chromosome = list(range(num_cities))
        random.shuffle(chromosome)
        population.append(chromosome)
    return population

def selection(population):
    contenders = random.sample(population, 3)
    contenders.sort(key=lambda c: fitness(c), reverse=True)
    return contenders[0]

def crossover(p1, p2):
    if random.random() < crossover_rate:
        a, b = sorted(random.sample(range(num_cities), 2))
        child = [-1]*num_cities
        child[a:b] = p1[a:b]
        fill = [x for x in p2 if x not in child]
```

```
        j = 0
        for i in range(num_cities):
            if child[i] == -1:
                child[i] = fill[j]
                j += 1
        return child
    return p1[:]

def mutate(chromosome):
    if random.random() < mutation_rate:
        a, b = random.sample(range(num_cities), 2)
        chromosome[a], chromosome[b] = chromosome[b], chromosome[a]
    return chromosome

population = initial_population()
best_solution = None
best_distance = float("inf")

for g in range(generations):
    new_pop = []
    for _ in range(population_size):
        parent1 = selection(population)
        parent2 = selection(population)
        child = crossover(parent1, parent2)
        child = mutate(child)
        new_pop.append(child)

    population = new_pop

    # Track best solution
    for chromo in population:
        d = tour_length(chromo)
        if d < best_distance:
            best_distance = d
            best_solution = chromo
print("Best Tour (order of cities):", best_solution)
print("Best Tour Distance:", best_distance)
```

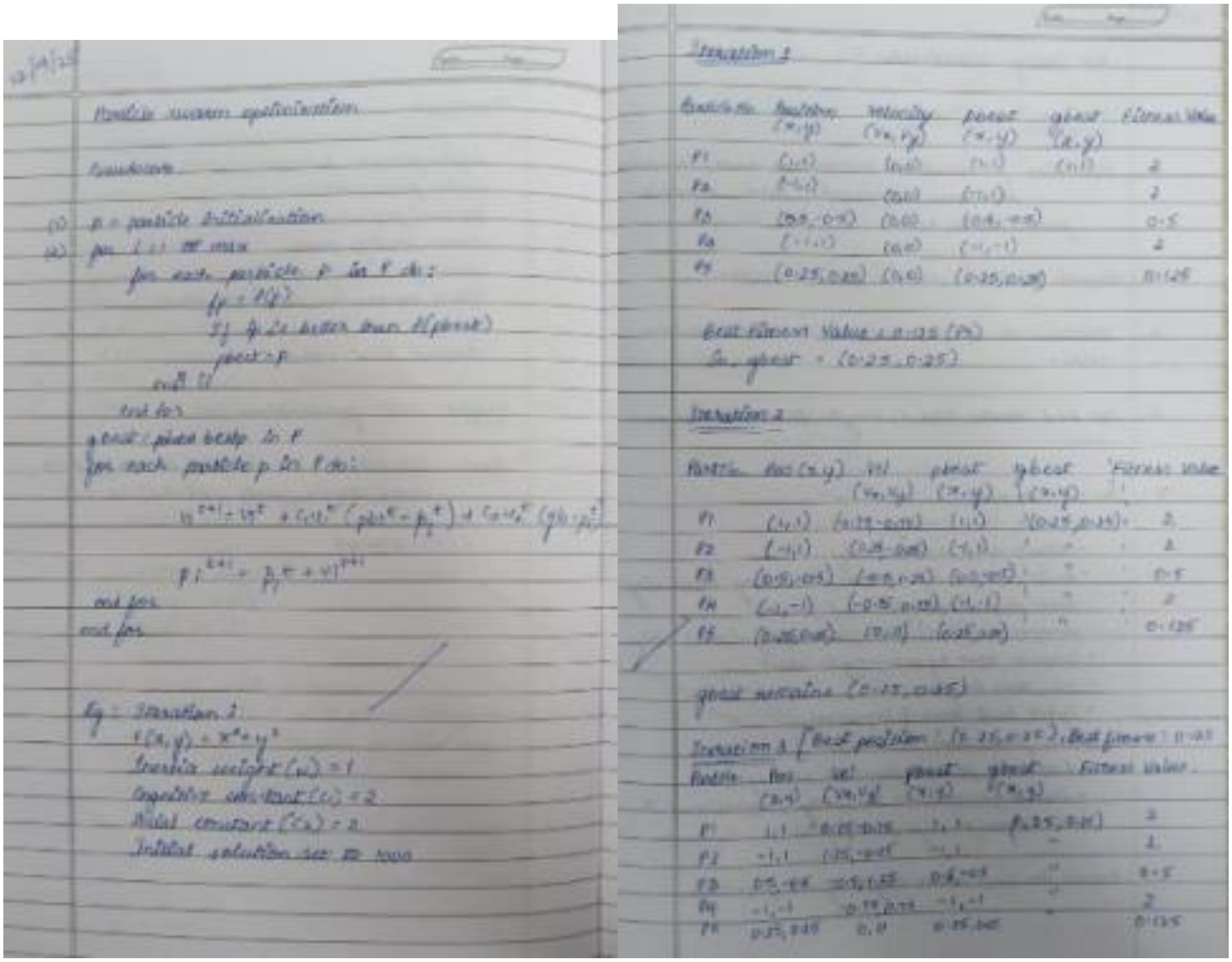
Output:

```
Best Tour (order of cities): [4, 2, 0, 1, 3]
Best Tour Distance: 22.35103276995244
```

Program 3

Particle Swarm Optimization for Function Optimization
Portfolio Optimization (Selecting assets) using Particle Swarm Optimization is about choosing how much money to allocate to different assets (stocks, bonds, etc.) to maximize expected return while minimizing risk (variance).

Algorithm:



Code:

```
import numpy as np

# ---------- Step 1: Define Problem (Portfolio Optimization) ----------
# Expected returns for 4 assets (example data)
returns = np.array([0.12, 0.18, 0.15, 0.10])

# Covariance matrix of returns (risk measure)
cov_matrix = np.array([
    [0.010, 0.002, 0.001, 0.003],
    [0.002, 0.030, 0.002, 0.004],
    [0.001, 0.002, 0.020, 0.002],
    [0.003, 0.004, 0.002, 0.025]
])

# Fitness function: Sharpe ratio (maximize return / risk)
def fitness(weights):
    weights = np.array(weights)
    portfolio_return = np.dot(weights, returns)
    portfolio_risk = np.sqrt(np.dot(weights.T, np.dot(cov_matrix, weights)))
    if portfolio_risk == 0: # avoid division by zero
        return -999
    return portfolio_return / portfolio_risk
```

```
# ---------- Step 2: Initialize PSO Parameters ----------
num_particles = 30
num_assets = len(returns)
iterations = 100

w = 0.7 # inertia weight
c1 = 1.5 # cognitive coefficient
c2 = 1.5 # social coefficient

# ---------- Step 3: Initialize Particles ----------
positions = np.random.dirichlet(np.ones(num_assets), size=num_particles) # weights sum=1
velocities = np.random.rand(num_particles, num_assets) * 0.1

personal_best_positions = positions.copy()
personal_best_scores = np.array([fitness(p) for p in positions])

global_best_position = personal_best_positions[np.argmax(personal_best_scores)]
global_best_score = np.max(personal_best_scores)

# ---------- Step 4: Main Loop ----------
for _ in range(iterations):
    for i in range(num_particles):
        # Update velocity
        r1, r2 = np.random.rand(num_assets), np.random.rand(num_assets)
        velocities[i] = (w * velocities[i]
                         + c1 * r1 * (personal_best_positions[i] - positions[i])
                         + c2 * r2 * (global_best_position - positions[i]))

        # Update position (weights must be valid portfolio)
        positions[i] += velocities[i]
        positions[i] = np.maximum(positions[i], 0) # no negative weights
        positions[i] /= np.sum(positions[i]) # normalize to sum=1

        # Evaluate fitness
        score = fitness(positions[i])

        # Update personal best
        if score > personal_best_scores[i]:
            personal_best_scores[i] = score
            personal_best_positions[i] = positions[i].copy()

        # Update global best
        if score > global_best_score:
            global_best_score = score
            global_best_position = positions[i].copy()

# ---------- Step 5: Output Result ----------
print("Optimal Portfolio Weights:", global_best_position)
print("Best Sharpe Ratio:", global_best_score)
```

Output:

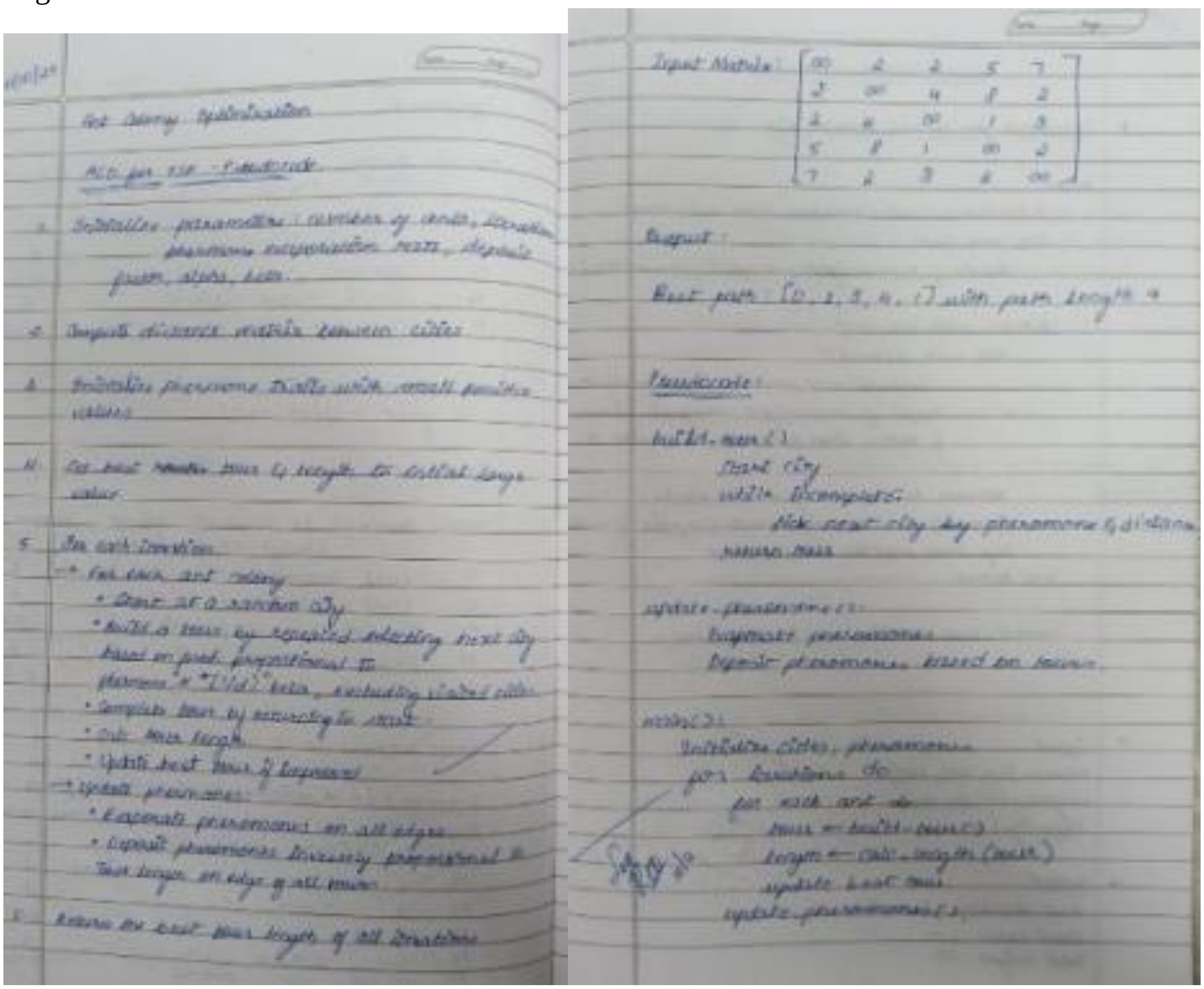
```
Optimal Portfolio Weights: [0.44097408 0.20835576 0.2823928  0.06827736]
Best Sharpe Ratio: 1.7756098324447378
```

Program 4

Ant Colony Optimization for the Traveling Salesman Problem

Ant Colony Optimization (ACO) for the Vehicle Routing Problem (VRP): It involves finding optimal routes for multiple vehicles to deliver goods to a set of customers from a central depot.

Algorithm:


Ant Colony Optimization

ACO for TSP - Pseudocode

2. Compute distance matrix between cities

3. Initialize pheromone trails with small positive values

5. For each iteration

→ For each ant colony

* Start at a random city

→ Update pheromones

* Evaporate pheromones on all edges

Return the best tour length of all iterations

Input Matrix:

Output:

Best path

Pseudocode:

update_pheromones():

Initialize cities, pheromones

update best tour

Code:

```
import numpy as np
import random

# Coordinates of depot + customers (0 is depot)
coords = np.array([
    [40, 50], # depot
    [45, 68], [50, 30], [55, 20], [60, 80], [65, 60], [70, 40]
])

num_vehicles = 2
num_ants = 10
num_iterations = 100
alpha = 1.0 # pheromone importance
beta = 5.0 # heuristic importance (inverse distance)
```

```
rho = 0.5 # pheromone evaporation rate
initial_pheromone = 1.0

num_cities = len(coords)

# Distance matrix
dist_matrix = np.sqrt(((coords[:, None] - coords[None, :])**2).sum(axis=2))

# Heuristic matrix (inverse distance), avoid division by zero
heuristic = 1 / (dist_matrix + np.diag([np.inf]*num_cities))

# Initialize pheromone trails
pheromone = np.ones((num_cities, num_cities)) * initial_pheromone
def choose_next_city(current_city, unvisited, pheromone, heuristic):
    pheromone_vals = pheromone[current_city][unvisited] ** alpha
    heuristic_vals = heuristic[current_city][unvisited] ** beta
    probs = pheromone_vals * heuristic_vals
    probs /= probs.sum()
    return np.random.choice(unvisited, p=probs)

def construct_solution():
    routes = [[] for _ in range(num_vehicles)]
    unvisited = set(range(1, num_cities)) # customers only
    for v in range(num_vehicles):
        routes[v].append(0) # start from depot

    while unvisited:
        for v in range(num_vehicles):
            current_city = routes[v][-1]
            candidates = list(unvisited)
            if not candidates:
                break
            next_city = choose_next_city(current_city, candidates, pheromone, heuristic)
            routes[v].append(next_city)
            unvisited.remove(next_city)
            if not unvisited:
                break

    # Return to depot
    for v in range(num_vehicles):
        routes[v].append(0)
    return routes

def route_length(route):
    length = 0
    for i in range(len(route)-1):
        length += dist_matrix[route[i], route[i+1]]
    return length

best_routes = None
best_length = float('inf')
```

```
for iteration in range(num_iterations):
    all_routes = []
    all_lengths = []

    for _ in range(num_ants):
        routes = construct_solution()
        total_length = sum(route_length(r) for r in routes)
        all_routes.append(routes)
        all_lengths.append(total_length)
        if total_length < best_length:
            best_length = total_length
            best_routes = routes

    # Pheromone evaporation
    pheromone *= (1 - rho)

    # Pheromone update (only best ant deposits pheromone)
    for route in best_routes:
        for i in range(len(route)-1):
            from_city = route[i]
            to_city = route[i+1]
            pheromone[from_city][to_city] += 1 / best_length
            pheromone[to_city][from_city] += 1 / best_length

print("Best total route length:", best_length)
for v, route in enumerate(best_routes):
    print(f"Vehicle {v+1} route: {route}")
```

Output:

Program 5

Cuckoo Search (CS)
Cuckoo Search Algorithms: We need to maximize the total value of selected items without exceeding the knapsack's weight capacity. Using the Cuckoo Search Algorithm, each solution is a binary vector, new solutions are generated via Lévy flights, and the best feasible solution is iteratively improved while abandoning poor solutions with a probability.

Algorithm:

Code:

```
import numpy as np
import random

# ---------------- Knapsack Problem Setup ----------------
# Example items: (value, weight)
items = [(60, 10), (100, 20), (120, 30)]
capacity = 50
n = len(items)

def fitness(solution):
    total_value = total_weight = 0
    for i in range(n):
        if solution[i] == 1:
            total_value += items[i][0]
            total_weight += items[i][1]
    if total_weight > capacity:
        return 0 # invalid solution
    return total_value

# ---------------- Cuckoo Search Algorithm ----------------
def levy_flight(Lambda):
    u = np.random.normal(0, 1) * np.power(abs(np.random.normal(0, 1)), -1.0 / Lambda)
    v = np.random.normal(0, 1)
    step = u / abs(v) ** (1 / Lambda)
    return step

def get_random_solution():
    return [random.randint(0, 1) for _ in range(n)]

def cuckoo_search(num_nests=10, pa=0.25, max_iter=100):
    nests = [get_random_solution() for _ in range(num_nests)]
    best = max(nests, key=fitness)

    for _ in range(max_iter):
        # Generate new solution via Levy flight
        cuckoo = best[:]
        step = int(abs(round(levy_flight(1.5)))) % n
        pos = random.randint(0, n-1)
        cuckoo[pos] = 1 - cuckoo[pos] # flip bit
```

```
        # Replace a random nest if better
        j = random.randint(0, num_nests-1)
        if fitness(cuckoo) > fitness(nests[j]):
            nests[j] = cuckoo

        # Abandon some nests with probability pa
        for i in range(num_nests):
            if random.random() < pa:
                nests[i] = get_random_solution()

        # Update best
        best = max(nests, key=fitness)

    return best, fitness(best)

# Run the algorithm
solution, value = cuckoo_search()
print("Best solution:", solution)
print("Total value:", value)
```

Output:

Program 6

Grey Wolf Optimizer (GWO)
Using the Grey Wolf Optimizer (GWO), we aim to find the shortest, obstacle-free path by modeling the search agents (wolves) to iteratively converge toward the best position (path node) in the environment. The algorithm simulates the grey wolves' hunting hierarchy and encircling behavior to efficiently navigate the space from the start point.

Algorithm:

Code:

```
import numpy as np
import random

# === Grid setup ===
GRID_SIZE = 5
START = (0, 0)
GOAL = (4, 4)
OBSTACLES = [(2, i) for i in range(1, 4)] # Vertical wall in column 2, rows 1 to 3

# === Parameters ===
POP_SIZE = 10
MAX_ITER = 50
PATH_LENGTH = 20 # fewer steps needed for small grid

# === Helper Functions ===
```

```
def is_valid(pos):
    x, y = pos
    return 0 <= x < GRID_SIZE and 0 <= y < GRID_SIZE and pos not in OBSTACLES

def move_toward_goal(current):
    moves = [(0,1), (1,0), (0,-1), (-1,0)]
    random.shuffle(moves)
cx, cy = current gx,
gy = GOAL
    moves.sort(key=lambda m: abs((cx + m[0]) - gx) + abs((cy + m[1]) - gy))
    for dx, dy in moves:
        new_pos = (cx + dx, cy + dy)
        if is_valid(new_pos):
            return new_pos
    return current

def generate_random_path():
    path = [START]
    visited = set(path)
    current = START
    for _ in range(PATH_LENGTH):
        current = move_toward_goal(current)
        if current in visited:
            continue
        path.append(current)
        visited.add(current)
        if current == GOAL:
            break
    return path

def path_cost(path):
    cost = len(path)
    if path[-1] != GOAL:
        dist = abs(path[-1][0] - GOAL[0]) + abs(path[-1][1] - GOAL[1])
        cost += 100 + dist
    for pos in path:
        if pos in OBSTACLES:
            cost += 50
    return cost

# === GWO Optimization ===

def gwo_optimize():
    wolves = [generate_random_path() for _ in range(POP_SIZE)]

    for iteration in range(MAX_ITER):
        wolves.sort(key=path_cost)
        alpha, beta, delta = wolves[0], wolves[1], wolves[2]
        a = 2 - iteration * (2 / MAX_ITER)

        for i in range(3, POP_SIZE):
            new_path = []
```

```
        for j in range(min(len(alpha), len(wolves[i]), PATH_LENGTH)):
            A = 2 * a * random.random() - a
            C = 2 * random.random()
            x_alpha = np.array(alpha[j])
            x_wolf = np.array(wolves[i][j])
            D_alpha = abs(C * x_alpha - x_wolf)
            X1 = x_alpha - A * D_alpha

            A = 2 * a * random.random() - a
            C = 2 * random.random()
            x_beta = np.array(beta[j])
            D_beta = abs(C * x_beta - x_wolf)
            X2 = x_beta - A * D_beta

            A = 2 * a * random.random() - a
            C = 2 * random.random()
            x_delta = np.array(delta[j])
            D_delta = abs(C * x_delta - x_wolf)
            X3 = x_delta - A * D_delta

            X_new = (X1 + X2 + X3) / 3
            X_new = tuple(map(int, np.clip(np.round(X_new), 0, GRID_SIZE - 1)))

            if is_valid(X_new):
                new_path.append(X_new)
            else:
                if new_path:
                    new_path.append(move_toward_goal(new_path[-1]))
                else:
                    new_path.append(move_toward_goal(START))
        wolves[i] = new_path

    best_path = sorted(wolves, key=path_cost)[0]
    return best_path

# === Textual Output ===

def print_grid(path):
    grid = [["." for _ in range(GRID_SIZE)] for _ in range(GRID_SIZE)]

    for x, y in OBSTACLES:
        grid[y][x] = "#" # Obstacle

    for x, y in path:
        if (x, y) != START and (x, y) != GOAL and grid[y][x] != "#":
            grid[y][x] = "*"

    sx, sy = START
    gx, gy = GOAL
    grid[sy][sx] = "S"
    grid[gy][gx] = "G"
```

```
    print("\n=== GWO Path Grid ===")
    for row in grid:
        print(" ".join(row))

    print("\nBest Path (coordinates):")
    print(path)

    print(f"\nPath Length: {len(path)}")
    print(f"Cost: {path_cost(path)}")

# === Run ===

best = gwo_optimize()
print_grid(best)
```

Output:

<u>Program 7</u>

Parallel Cellular Algorithms and Programs
The task is to perform edge detection or noise reduction in an image using Parallel Cellular Automata (PCA), where each pixel (cell) interacts with its neighbors to enhance edges or reduce noise iteratively.

Algorithm:

Code:

```
import numpy as np
import cv2
```

```
import matplotlib.pyplot as plt

# Function for Cellular Automata (Edge Detection or Noise Reduction)
def cellular_automata(image, iterations=10, threshold=30):
    grid = image.copy() # Initialize grid (image as 2D array)
    neighbors = [(-1, -1), (-1, 0), (-1, 1), (0, -1), (0, 0), (0, 1), (1, -1), (1, 0), (1, 1)]

    for iteration in range(iterations):
        updated_grid = grid.copy()

        for i in range(1, len(grid) - 1): # Loop through pixels (excluding borders)
            for j in range(1, len(grid[0]) - 1):
                pixel = grid[i, j]
                neighbor_vals = [grid[i+di, j+dj] for (di, dj) in neighbors]

                # Edge detection: large difference with neighbors indicates edge
                if max(neighbor_vals) - min(neighbor_vals) > threshold:
                    updated_grid[i, j] = 255 # Edge pixel
                else:
                    # Noise reduction: average with neighbors for smoothing
                    new_pixel_value = sum(np.clip(neighbor_vals, 0, 255)) // 8 # Clipping before averaging

                    # Clip the new pixel value to the range 0-255
                    updated_grid[i, j] = np.clip(new_pixel_value, 0, 255)

        grid = updated_grid # Update the grid with new values

    return grid # Output updated image

# Set numpy to ignore overflow warnings
np.seterr(over='ignore')

# Generate a smaller dummy grayscale image (random noise)
# Create a 5x5 pixel image with random values between 0 and 255
image = np.random.randint(0, 256, (5, 5), dtype=np.uint8)

# Print the original image
print("Original Image (Pixel Values):")
for row in image:
    print(row)

# Apply the cellular automata algorithm
iterations = 10
threshold = 30
processed_image = cellular_automata(image, iterations, threshold)

# Print the processed image
print("\nProcessed Image (Pixel Values):")
for row in processed_image:
    print(row)

# Visualize the images using matplotlib
```

```
plt.figure(figsize=(8,4))

plt.subplot(1,2,1)
plt.title('Original Image')
plt.imshow(image, cmap='gray', vmin=0, vmax=255)
plt.axis('off')

plt.subplot(1,2,2)
plt.title('Processed Image')
plt.imshow(processed_image, cmap='gray', vmin=0, vmax=255)
plt.axis('off')

plt.tight_layout()
plt.show()
```

Output: