VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB RECORD

Bio Inspired Systems (23CS5BSBIS)

Submitted by

Parth Jain (1BM23CS357)

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 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 **Parth Jain (1BM23CS357),** who is a 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.

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

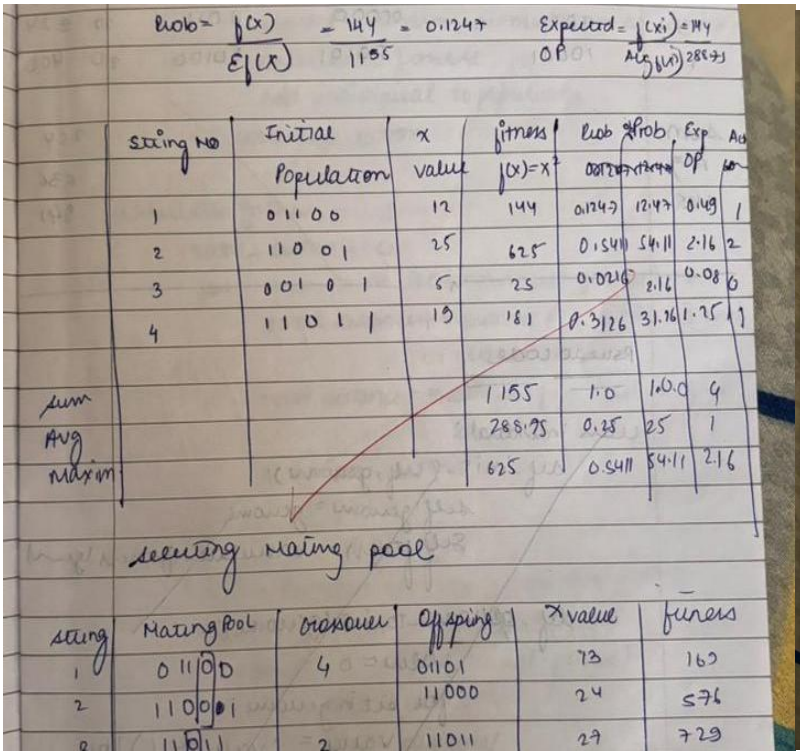
Index

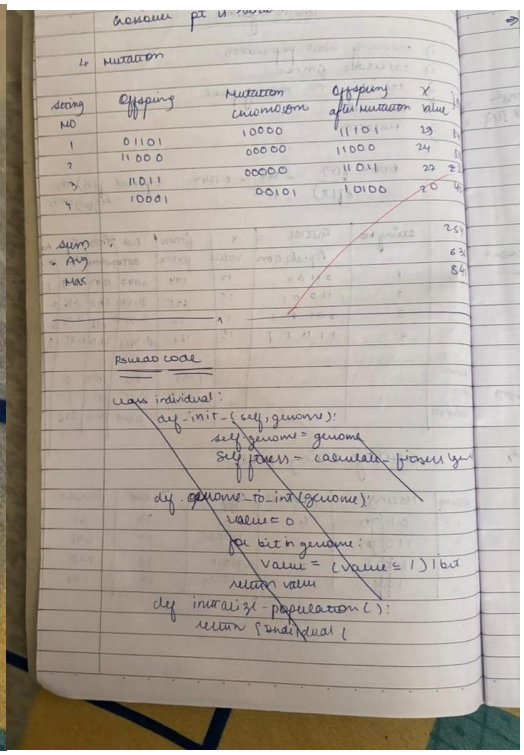
Github Link:
https://github.com/parthjain21108/BIS_LAB

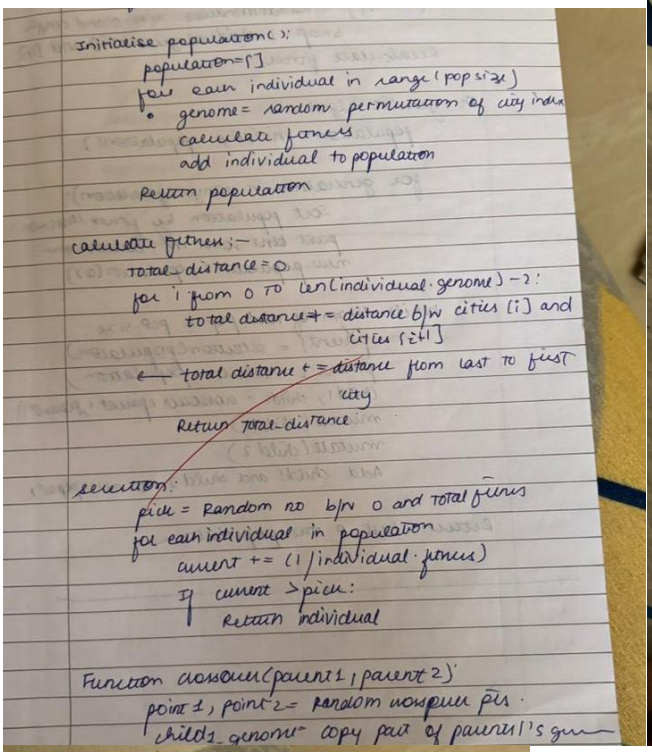
Program 1

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

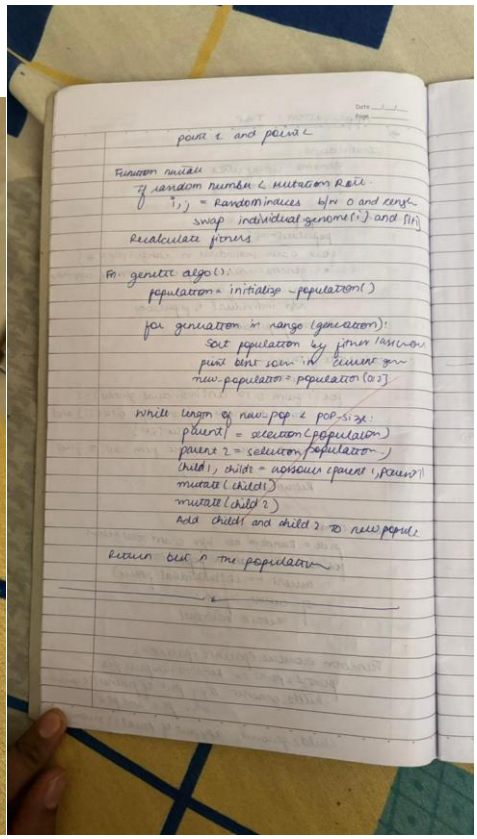
Algorithm:



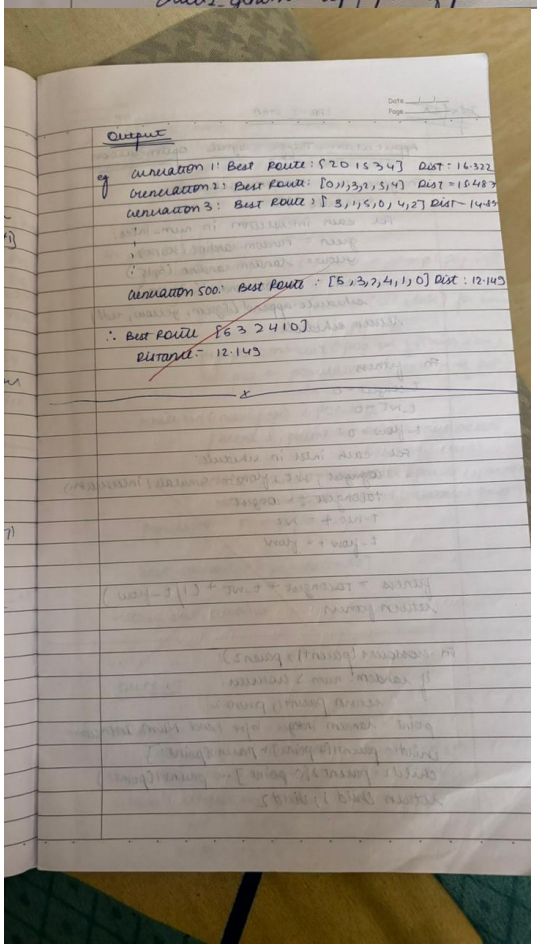




```
Initialise population():
    population=[]
    for each individual in range(pop size)
        genome = random permutation of city indices
        calculate fitness
        add individual to population
    Return population

calculate fitness:-
    total_distance = 0
    for i from 0 to len(individual.genome) - 2:
        total distance += distance b/w cities[i] and cities[i+1]
    ← total distance += distance from last to first city
    Return total_distance

selection:
    pick = Random no b/w 0 and total fitness
    for each individual in population
        current += (1/individual.fitness)
        If current > pick:
            Return individual

Function crossover(parent1, parent2)
    point 1, point 2 = random crossover pts.
    child1_genome = copy part of parent1's ge
```



```
point 1 and point 2

Function mutate
    If random number < mutation Rate:
        i, j = Random indices b/w 0 and length
        swap individual.genome[i] and [j]
    Recalculate fitness

Fn. genetic algo():
    population = initialize_population()

    for generation in range(generations):
        Sort population by fitness
        print best score in current gen
        new_population = population[0:2]

        While length of new_pop < POP_SIZE:
            parent1 = selection(population)
            parent2 = selection(population)
            child1, child2 = crossover(parent1, parent2)
            mutate(child1)
            mutate(child2)
            Add child1 and child2 to new popul

    Return best n the population
```



```
Output

eg  Generation 1: Best Route: [2 0 1 5 3 4]  Dist: 16.322
    Generation 2: Best Route: [0, 1, 3, 2, 5, 4]  Dist = 15.48
    Generation 3: Best Route: [3, 1, 5, 0, 4, 2]  Dist = 14.49
    .
    .
    .
    Generation 500: Best Route = [5, 3, 2, 4, 1, 0]  Dist: 12.143

∴ Best Route [5 3 2 4 1 0]
  Distance = 12.143
```

Code:

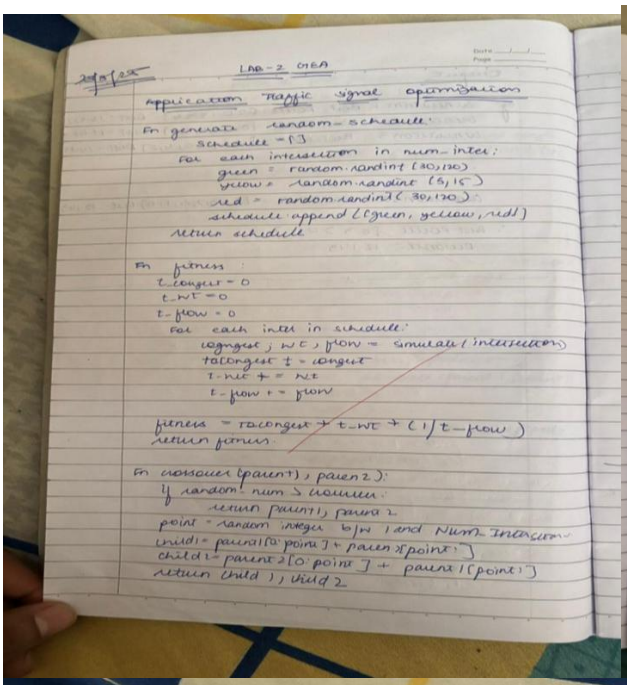
```
import random
import math

# 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)
```

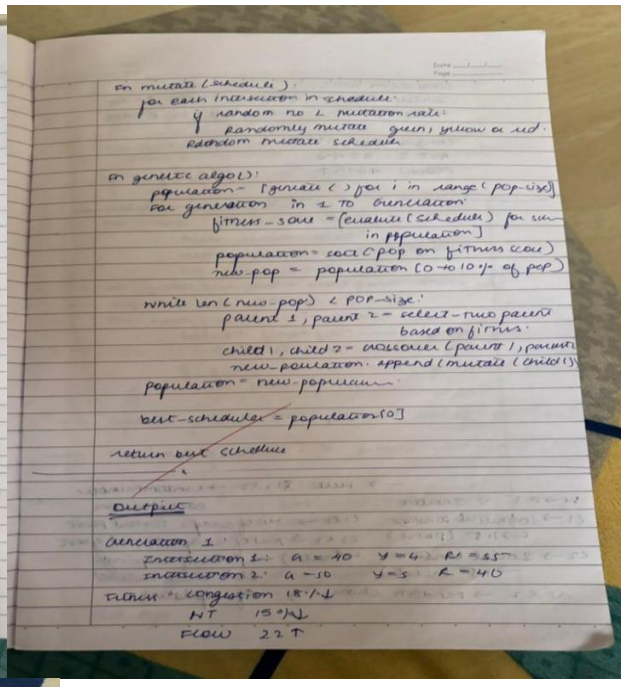
```
        population = new_population
        best = max(population, key=fitness)
        print(f"Gen {generation}: Best x = {best:.4f}, f(x) = {fitness(best):.4f}")

    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}")
```

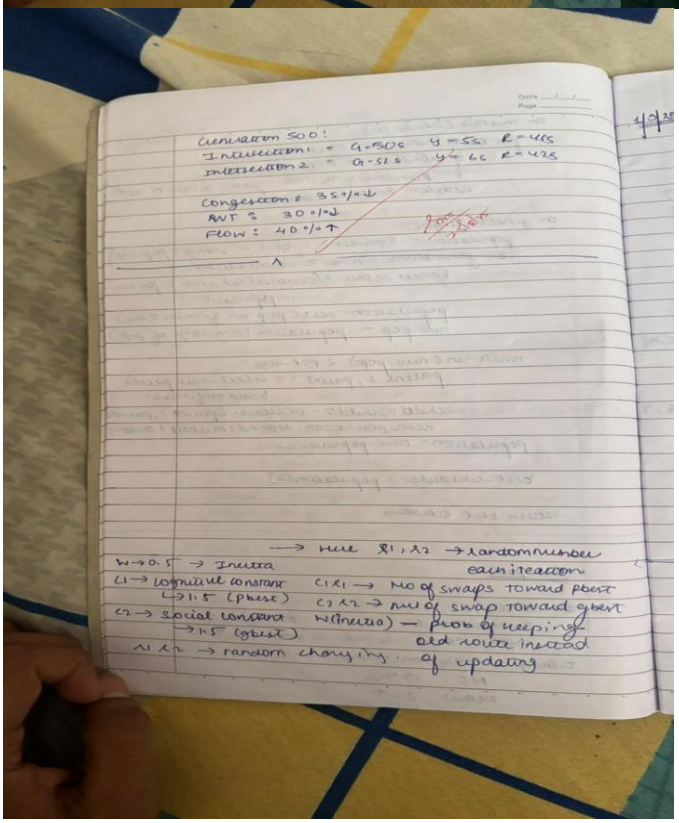
Program 2

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

Algorithm:







Code:

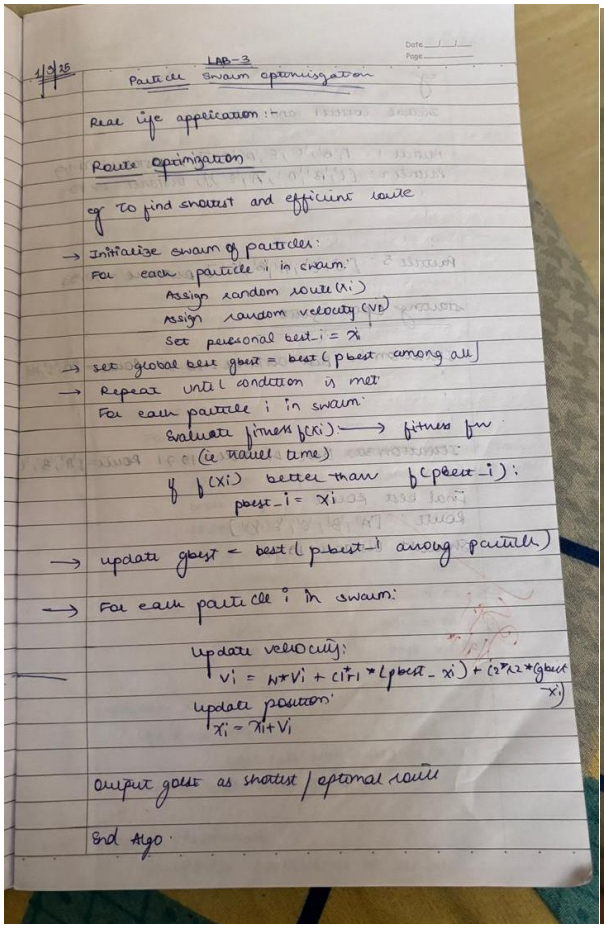
```
import random
import math

# 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]
```

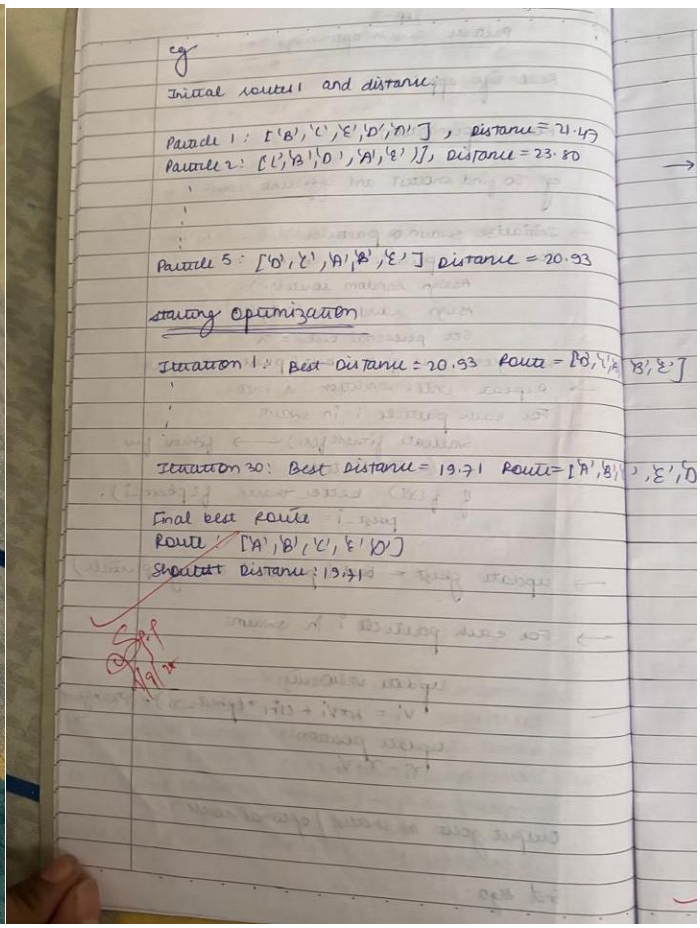
```
    pos = end
    for gene in parent2:
        if gene not in child:
            if pos >= N_GENES: pos = 0
            child[pos] = gene
            pos += 1
    return child

# 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:





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
```

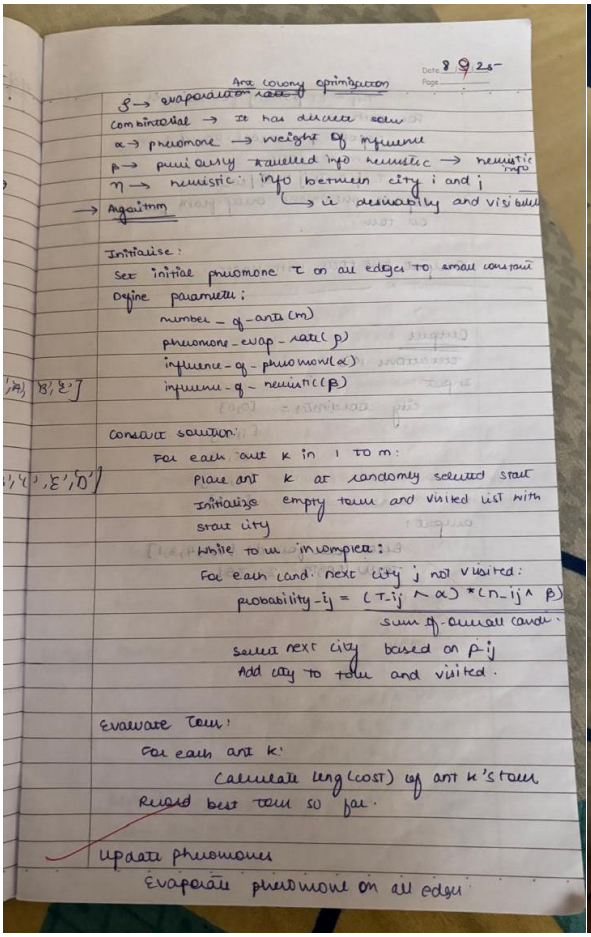
```
num_particles = 3
num_iterations = 2
dim = 2       # dimensions: x and y

# 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]},
fitness={f(positions[i])}")
```

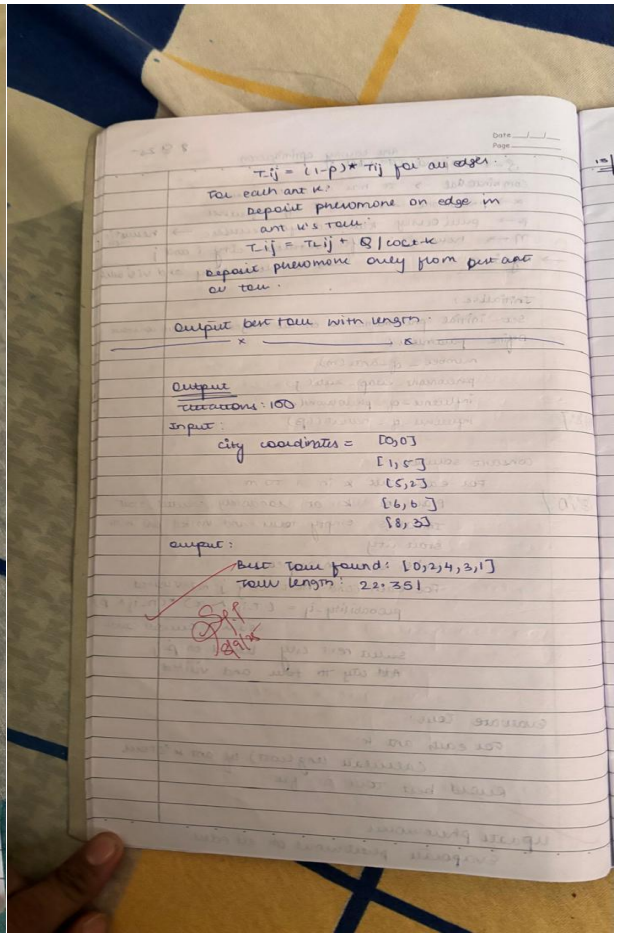
```
print(f" Global best position: {global_best_position}, value: {global_best_value}\n")
```

Program 4

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

Algorithm:





Code:

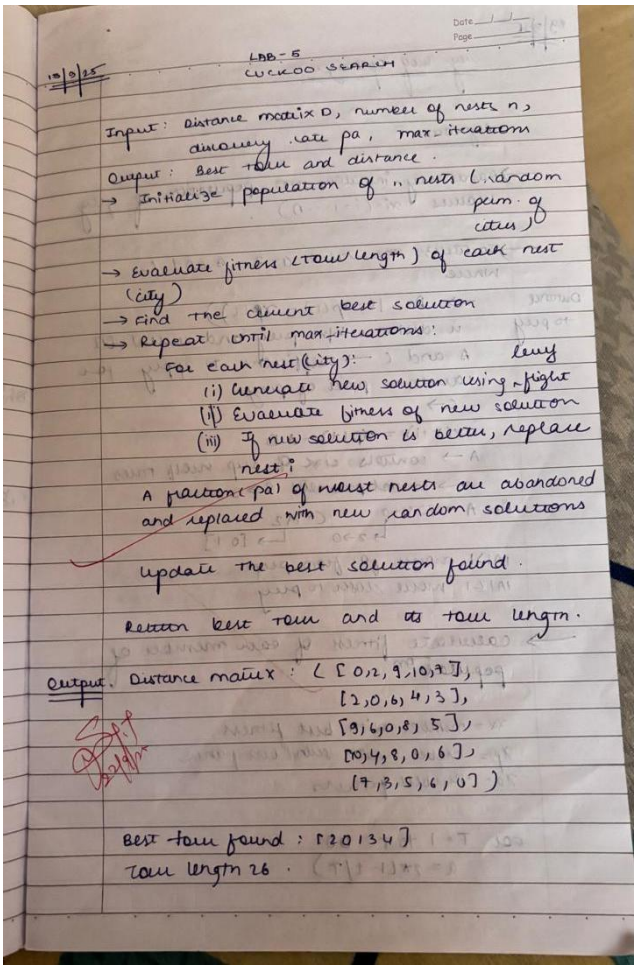
```
import numpy as np
import random

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
            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:



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)
```

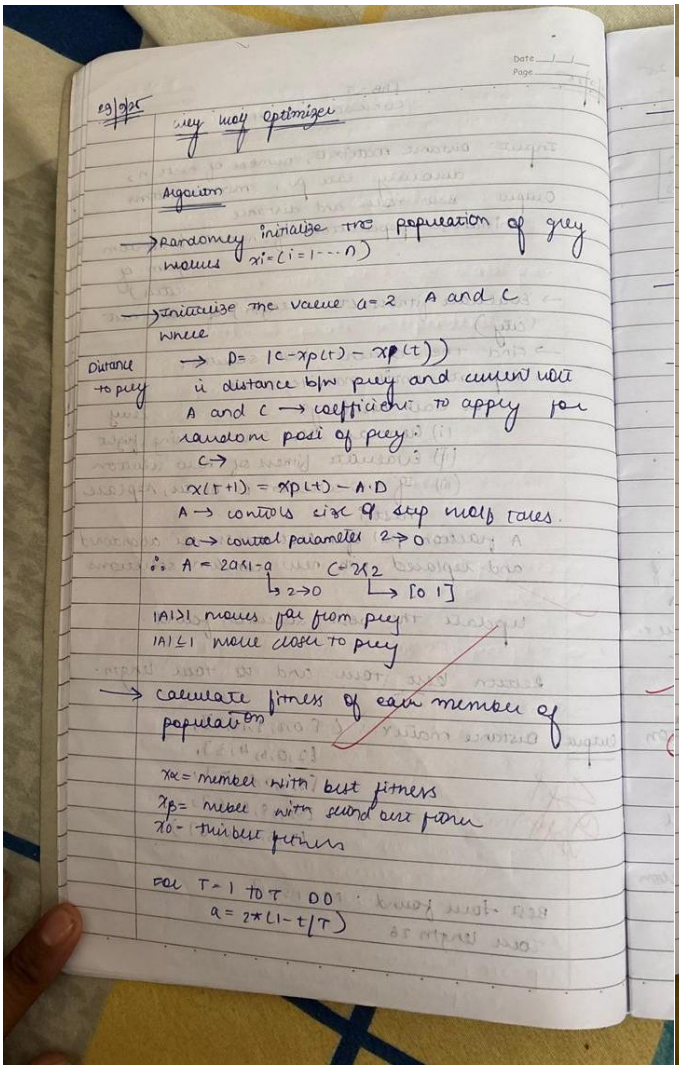
```
    v = np.random.normal(0, 1)
    step = u / abs(v)**(1/Lambda)
    return step

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],
```

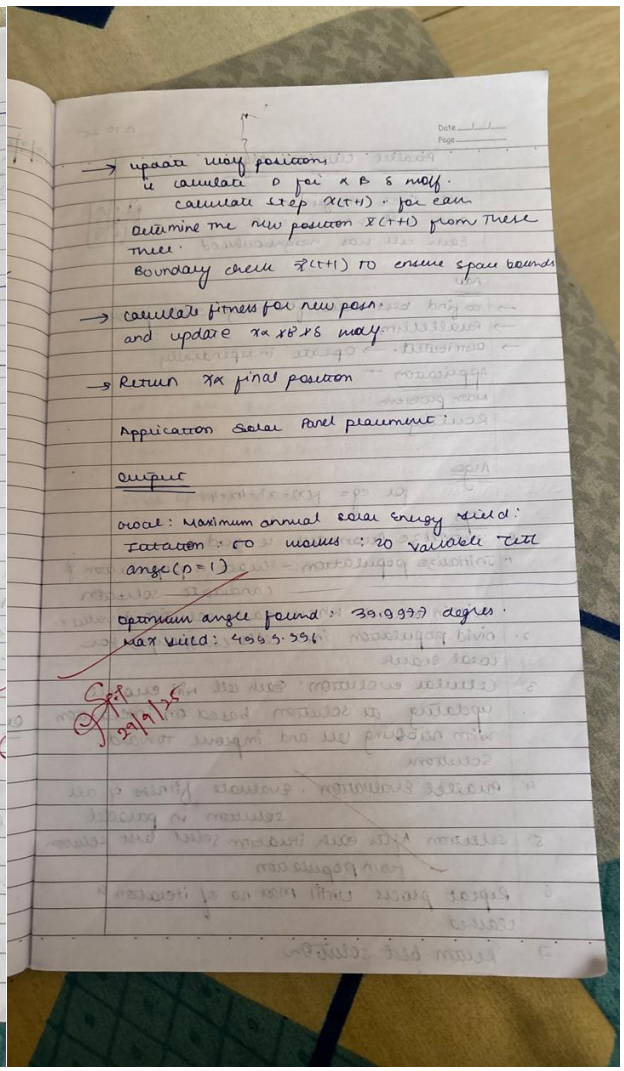
```
    [10, 4, 8, 0, 6],
    [7, 3, 5, 6, 0]
])

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:





Code:

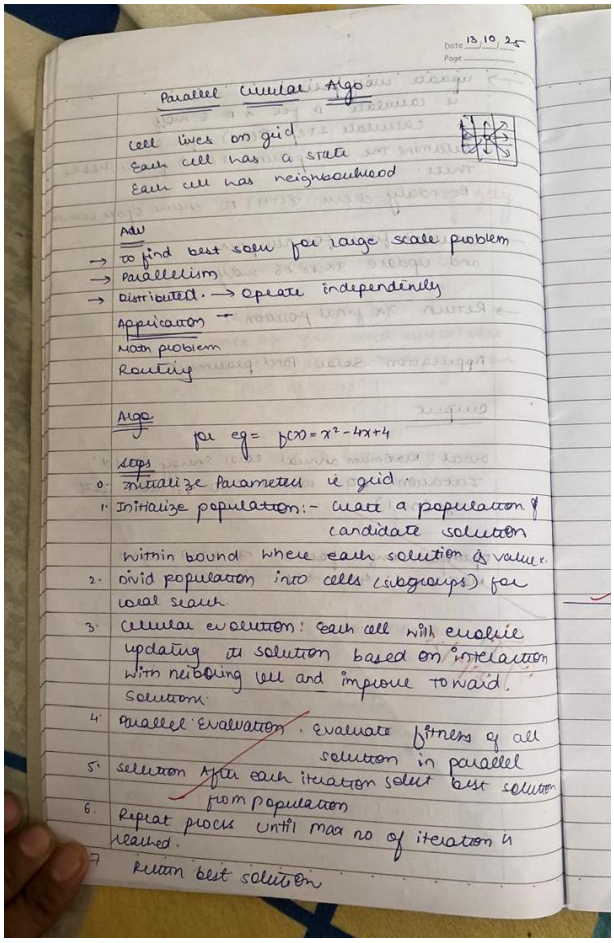
```
import numpy as np

# ---- 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
```

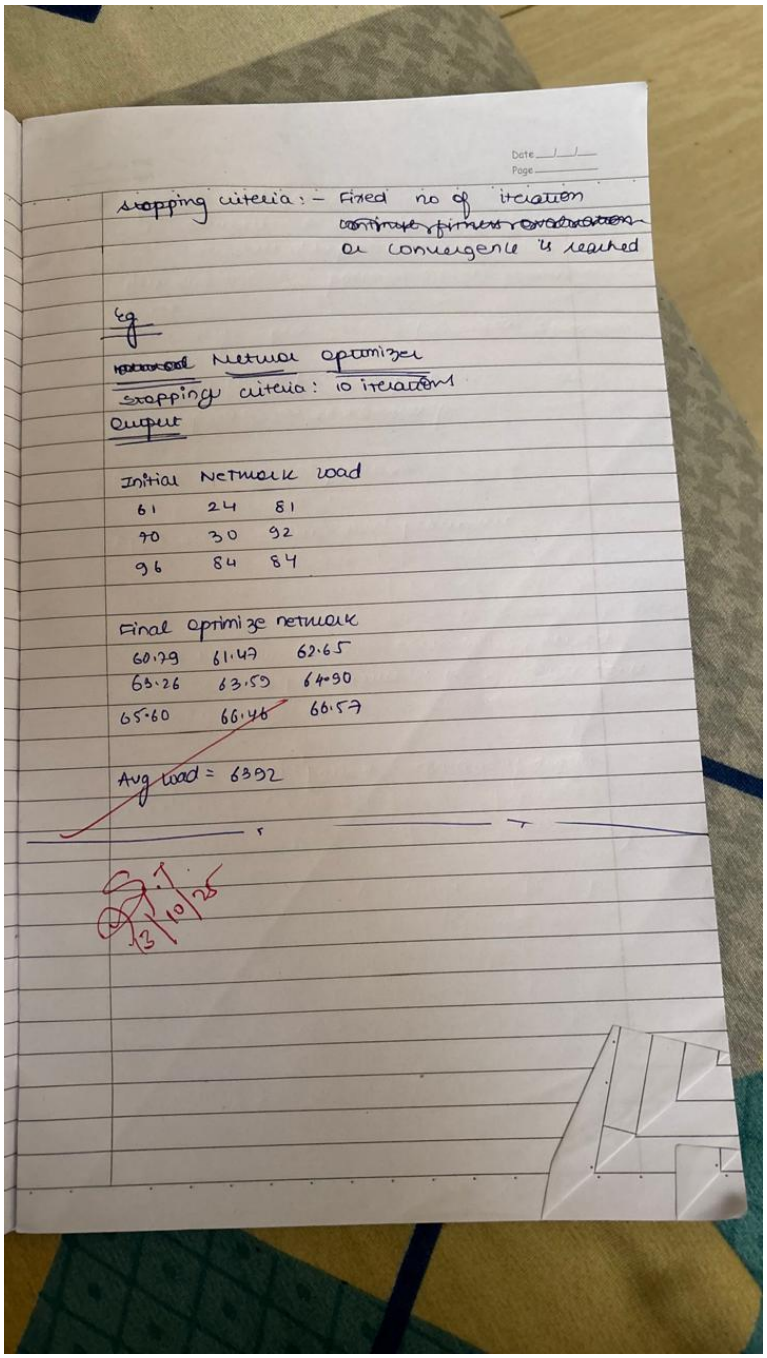
```
            X[j] = (X1 + X2 + X3) / 3

        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:





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)
```