VISVESVARAYA TECHNOLOGICAL UNIVERSITY

“JnanaSangama”, Belgaum -590014, Karnataka.

LAB RECORD

Bio Inspired Systems (23CS5BSBIS)

Submitted by

PALEPU VISHAL (1BM23CS224)

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 **StudentName (1BM23CS000),** 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.

Lab faculty In charge Name……….. Assistant Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE

Index

Github Link:
https://github.com/Palepuvishal/Bio-Inspired-Systems

Program 1

The goal is to find the **optimal binary string (chromosome)** of a fixed length that **maximizes the output** of a specific **fitness function**.

Details of the Problem:

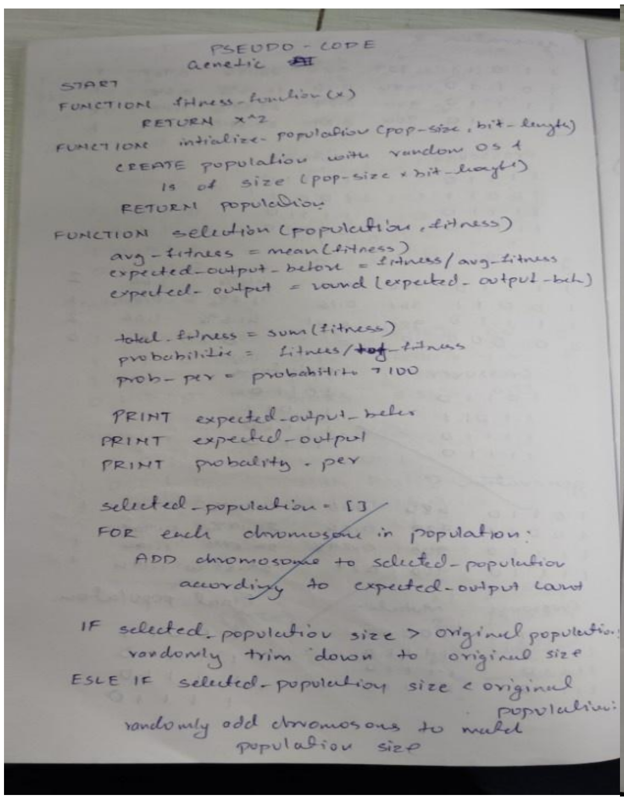
- **Optimization Type:** Maximization (since the goal of an optimization problem is typically to find the "best" solution, and the GA iteratively improves the fitness value).
- **Solution Representation:** Each potential solution is represented as a **binary string** (a "chromosome"). In the provided outputs, the strings appear to be **5 or 6 bits** long (e.g., 10101).
- Fitness Function: The fitness of a chromosome is calculated by first converting its binary representation to a decimal value, and then applying a specific mathematical function. The pseudocode explicitly defines the fitness function as:

  fitness_function(x) = x^2

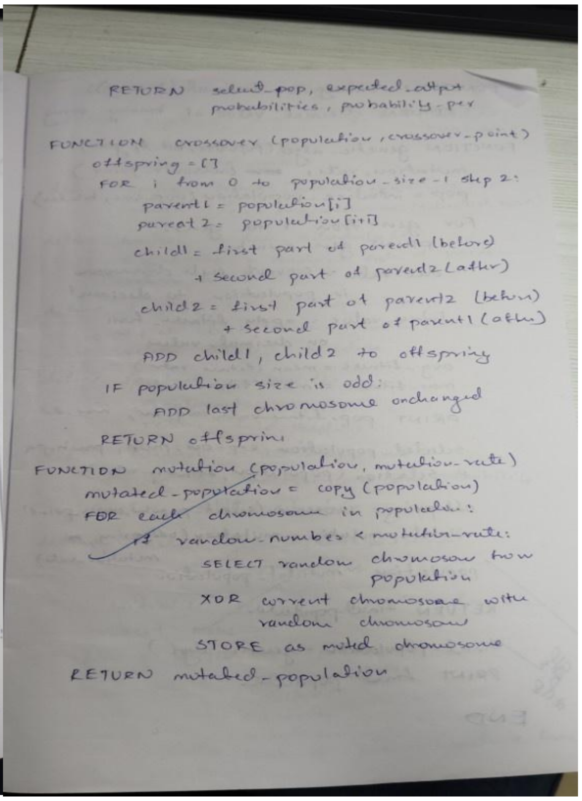
  where x is the decimal value of the binary chromosome.

- **Goal:** The Genetic Algorithm iteratively applies selection, crossover, and mutation over multiple generations to evolve a **population** of binary strings until a chromosome is found whose fitness value, {decimal_value}^2

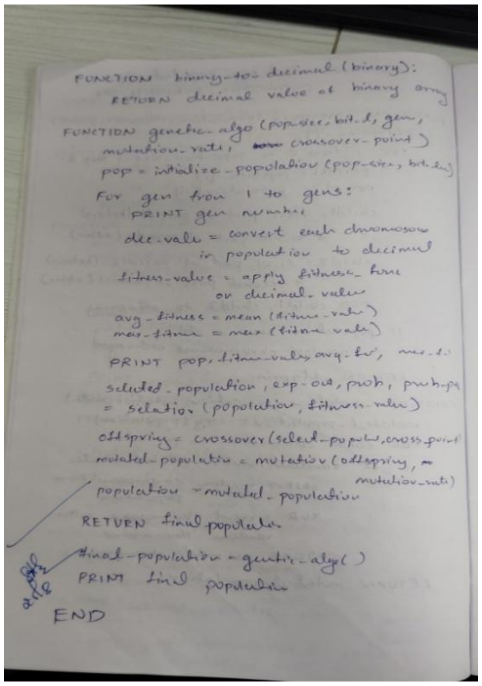
Algorithm:



```
PSEUDO - CODE
Genetic ~~AI~~

START
FUNCTION fitness-function(x)
        RETURN x^2
FUNCTION intialize-population (pop-size, bit-length)
    CREATE population with random 0s 1
        is of size (pop-size x bit_length)
    RETURN population
FUNCTION selection (population, fitness)
    avg-fitness = mean(fitness)
    expected-output-before = fitness/avg-fitness
    expected-output = round (expected-output-bef)

    total.fitness = sum(fitness)
    probabilitie = fitness/~~tot~~-fitness
    prob-per = probabilite * 100

    PRINT expected-output-before
    PRINT expected-output
    PRINT probality - per

    selected-population = []
    FOR each chromosome in population:
        ADD chromosome to selected-population
            according to expected-output count

  IF selected-population size > original population:
      randomly trim down to original size
  ESLE IF selected-population size < original population:
      randomly add chromosomes to match
            population size
```



```
    RETURN select-pop, expected-output
           probabilities, probability-per

FUNCTION crossover (population, crossover-point)
    offspring = []
    FOR i from 0 to population-size-1 step 2:
        parent1 = population[i]
        parent2 = population[i+1]
        child1 = first part of parent1 (before)
                 + second part of parent2 (after)
        child2 = first part of parent2 (before)
                 + second part of parent1 (after)
        ADD child1, child2 to offspring
    IF population size is odd:
        ADD last chromosome unchanged
    RETURN offspring
FUNCTION mutation (population, mutation-rate)
    mutated-population = copy (population)
    FOR each chromosome in population:
        if random-number < mutation-rate:
            SELECT random chromosome from
                          population
            XOR current chromosome with
                    random chromosome
            STORE as muted chromosome
  RETURN mutated-population
```



```
FUNCTION binary_to_decimal (binary):
    RETURN decimal value of binary array

FUNCTION genetic_algo (pop_size, bit_l, gen,
    mutation_rate, crossover_point)
    pop = initialize_population (pop_size, bit_len)
    For gen from 1 to gens:
        PRINT gen number
        dec_vals = convert each chromosome
                   in population to decimal
        fitness_value = apply fitness_func
                        on decimal_values
        avg_fitness = mean (fitness_value)
        max_fitness = max (fitness_value)
        PRINT pop, fitness_values, avg_fv, max_f
        selected_population, exp_out, prob, prob_pe
        = selection (population, fitness_value)
        offspring = crossover (select_popula, cross_point)
        mutated_population = mutation (offspring,
                                       mutation_rate)
        population = mutated_population
    RETURN final population
final_population = genetic_algo( )
PRINT final population
END
```

[illegible]

Code:

```
import numpy as np

def fitness_function(x):
 return x ** 2



def initialize_population(pop_size, bit_length):
    population = np.random.randint(0, 2, (pop_size, bit_length))
    return population



def selection(population, fitness):
    avg_fitness = np.mean(fitness)
    expected_output_before_rounding = fitness / avg_fitness
    expected_output = np.round(expected_output_before_rounding).astype(int)


    total_fitness = np.sum(fitness)
    probabilities = fitness / total_fitness

    probability_percentage = probabilities * 100



    print(f"Expected Output before rounding: {expected_output_before_rounding}")

    print(f"Selection Probabilities: {probabilities}")

    print(f"Selection Probability Percentages: {probability_percentage}")



    selected_population = []

    for idx, count in enumerate(expected_output):

        selected_population.extend([population[idx]] * count)



    selected_population = np.array(selected_population)



    pop_size = len(population)

    if len(selected_population) > pop_size:

        selected_population = selected_population[np.random.choice(len(selected_population),
pop_size, replace=False)]

    elif len(selected_population) < pop_size:

        extra_indices = np.random.choice(len(population), pop_size - len(selected_population))

        selected_population = np.vstack([selected_population, population[extra_indices]])



    return selected_population, expected_output, probabilities, probability_percentage



def crossover(population, crossover_point):

    offspring = []
    pop_size = len(population)

    for i in range(0, pop_size - 1, 2):
        parent1 = population[i]
        parent2 = population[i + 1]
        child1 = np.concatenate([parent1[:crossover_point], parent2[crossover_point:]])
        child2 = np.concatenate([parent2[:crossover_point], parent1[crossover_point:]])
        offspring.extend([child1, child2])

    if pop_size % 2 == 1:
        offspring.append(population[-1])

    return np.array(offspring)

def mutation(population, mutation_rate):
    mutated_population = population.copy()
    pop_size = len(population)

    for i in range(pop_size):
        if np.random.rand() < mutation_rate:
            rand_idx = np.random.randint(0, pop_size)
            random_chrom = population[rand_idx]
            mutated_population[i] = np.bitwise_xor(mutated_population[i], random_chrom)

    return mutated_population
```

```
def binary_to_decimal(binary):

    return int(''.join(map(str, binary)), 2)



def genetic_algorithm(pop_size=4, bit_length=5, generations=5, mutation_rate=0.05,
crossover_point=2):

    population = initialize_population(pop_size, bit_length)



    for generation in range(generations):

        print(f"\nGeneration {generation + 1}:")

        decimal_values = np.array([binary_to_decimal(ind) for ind in population])

        fitness_values = fitness_function(decimal_values)

        avg_fitness = np.mean(fitness_values)

        max_fitness = np.max(fitness_values)



        print(f"Population:\n{population}")

        print(f"Fitness Values: {fitness_values}")

        print(f"Avg Fitness: {avg_fitness}, Max Fitness: {max_fitness}")



        selected_population, expected_output, probabilities, probability_percentage =
selection(population, fitness_values)

        print(f"Expected Output (rounded): {expected_output}")



        offspring = crossover(selected_population, crossover_point)

        print(f"Offspring after Crossover:\n{offspring}")
        mutated_population = mutation(offspring, mutation_rate)

        print(f"Mutated Population:\n{mutated_population}")



        population = mutated_population



    return population



final_population = genetic_algorithm()

print(f"\nFinal Population:\n{final_population}")
```

Program 2

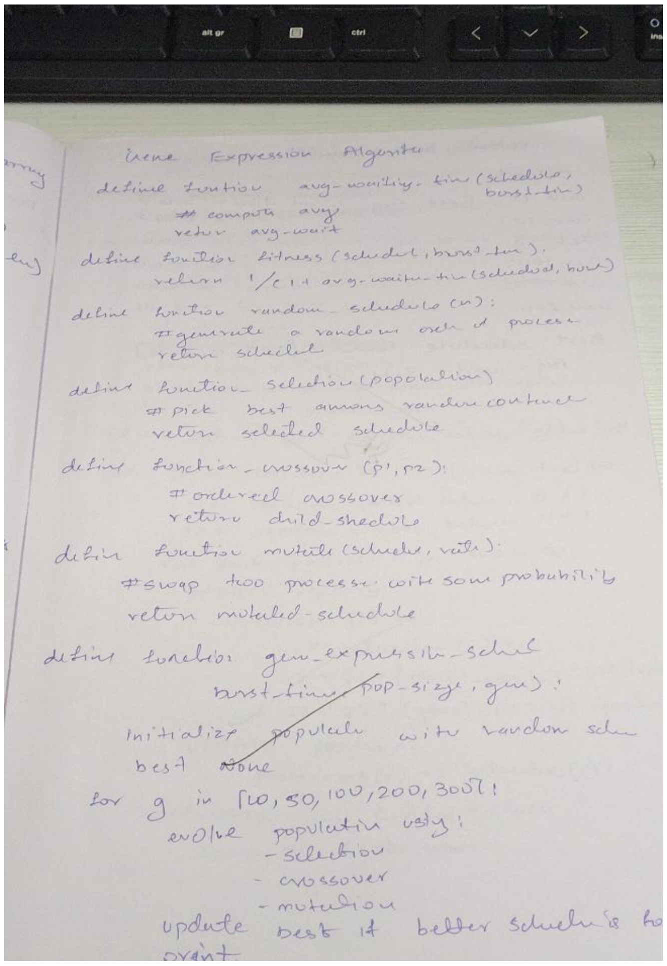
The goal is to find the **optimal schedule (order) of processes** to minimize the **Average Waiting Time** for all processes.

Details of the Problem:

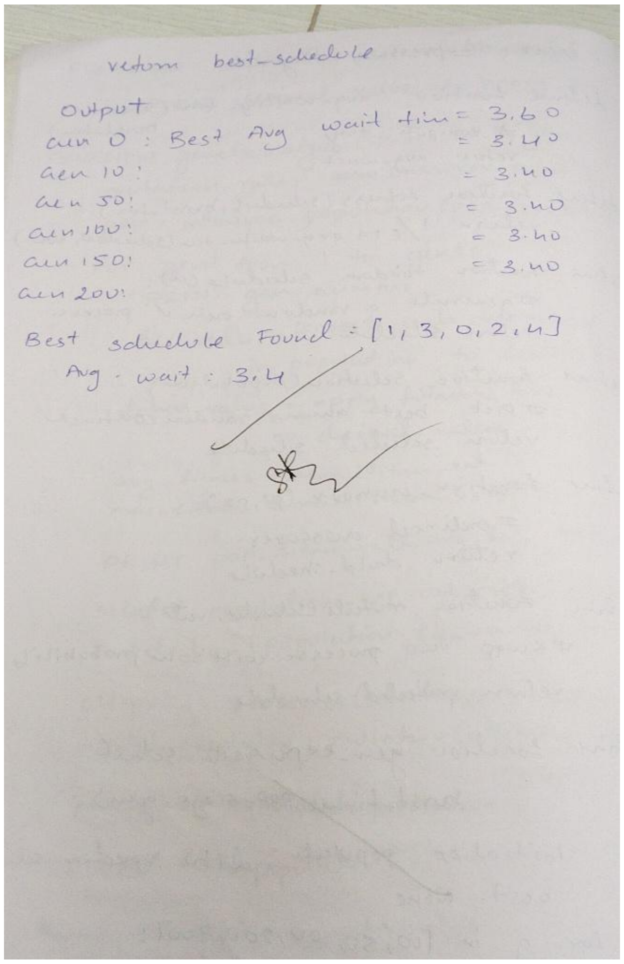
- **Problem Type:** Optimization (Minimization).
- **Domain:** Process Scheduling (e.g., in an operating system or manufacturing).
- **Goal:** Find a schedule (a permutation of process IDs) that yields the **minimum average waiting time**.
- **Inputs:** A set of processes with their corresponding **burst times** (processing times).
- **Solution Representation (Chromosome):** A **schedule**, which is an ordered sequence of process IDs (e.g., $[1, 3, 0, 2, 4]$).
- Fitness Function: The fitness is defined as the reciprocal of the average waiting time, meaning that minimizing the average waiting time maximizes the fitness.

  Fitness(schedule) = 1 / (1 + Average Waiting Time(schedule, burst_time))

Algorithm



```
Gene Expression Algorithm
define function avg-waiting-time (schedule,
                                  burst-time)
    # compute avg
    return avg-wait
define function fitness (schedule, burst-time):
    return 1/(1 + avg-waiting-time (schedule, burst)
define function random_schedule (n):
    # generate a random order of processes
    return schedule
define function_ selection (population)
    # pick best among random contest
    return selected schedule
define function_ crossover (p1, p2):
    # ordered crossover
    return child_shedule
define function mutate (schedule, rate):
    # swap two processes with some probability
    return mutated-schedule
define function gen_expression-sched
        burst-time, pop-size, gen):
    Initialize population with random sche
    best None
for g in [10, 50, 100, 200, 300]:
    evolve population using:
        - selection
        - crossover
        - mutation
    update best if better schedule is fo
    print
```



```
return best_schedule

Output
Gen 0 : Best Avg wait time = 3.60
Gen 10 :                   = 3.40
Gen 50 :                   = 3.40
Gen 100 :                  = 3.40
Gen 150 :                  = 3.40
Gen 200 :                  = 3.40

Best schedule Found : [1, 3, 0, 2, 4]
  Avg. wait : 3.4
```

[illegible]

Code:

```
import random
import numpy as np

def compute_avg_waiting_time(schedule, burst_times):
    num_processes = len(schedule)
    current_time = 0
    total_waiting_time = 0

    for process_id in schedule:
        waiting_time = current_time
        total_waiting_time += waiting_time
        current_time += burst_times[process_id]
    if num_processes == 0:
        return 0
    return total_waiting_time / num_processes

def fitness(schedule, burst_times):
    avg_wait = compute_avg_waiting_time(schedule, burst_times)
    return 1.0 / (1.0 + avg_wait)

def initialize_population(pop_size, num_processes):
    population = []
    process_ids = list(range(num_processes))
    for _ in range(pop_size):
        schedule = random.sample(process_ids, num_processes)
        population.append(schedule)
    return population

def selection(population, fitness_scores, num_parents):
    total_fitness = sum(fitness_scores)
    if total_fitness == 0:
        probabilities = [1.0 / len(population)] * len(population)
    else:
        probabilities = [f / total_fitness for f in fitness_scores]

    selected_indices = np.random.choice(
        len(population),
        size=num_parents,
        p=probabilities
    )
    parents = [population[i] for i in selected_indices]
    return parents

def crossover(p1, p2):
    size = len(p1)
    start, end = sorted(random.sample(range(size), 2))

    c1 = [None] * size
    c1[start:end+1] = p1[start:end+1]

    p2_genes_to_add = [gene for gene in p2 if gene not in c1]
    p2_idx = 0
    for i in range(size):
        if c1[i] is None:
            c1[i] = p2_genes_to_add[p2_idx]
            p2_idx += 1

    c2 = [None] * size
    c2[start:end+1] = p2[start:end+1]
    p1_genes_to_add = [gene for gene in p1 if gene not in c2]
    p1_idx = 0
    for i in range(size):
        if c2[i] is None:
            c2[i] = p1_genes_to_add[p1_idx]
            p1_idx += 1

    return c1, c2

def mutate(schedule, mutation_rate):
    if random.random() < mutation_rate:
        i, j = random.sample(range(len(schedule)), 2)
        schedule[i], schedule[j] = schedule[j], schedule[i]
    return schedule

def genetic_algorithm(burst_times, pop_size, num_generations, mutation_rate):
    num_processes = len(burst_times)
    population = initialize_population(pop_size, num_processes)
    best_schedule = None
    min_avg_wait = float('inf')

    for gen in range(num_generations):
        fitness_scores = [fitness(sch, burst_times) for sch in population]

        current_best_idx = np.argmax(fitness_scores)
        current_best_schedule = population[current_best_idx]
        current_min_avg_wait = compute_avg_waiting_time(current_best_schedule, burst_times)

        if current_min_avg_wait < min_avg_wait:
            min_avg_wait = current_min_avg_wait
            best_schedule = current_best_schedule

        num_parents = pop_size
        parents = selection(population, fitness_scores, num_parents)
        random.shuffle(parents)

        next_population = []
        for i in range(0, num_parents, 2):
            p1 = parents[i]
            if i + 1 < num_parents:
                p2 = parents[i+1]
                c1, c2 = crossover(p1, p2)
                c1 = mutate(c1, mutation_rate)
                c2 = mutate(c2, mutation_rate)
                next_population.extend([c1, c2])
            else:
                next_population.append(mutate(p1[:], mutation_rate))

        population = next_population[:pop_size]

    return best_schedule, min_avg_wait

# --- Example Execution ---
example_burst_times = [8, 4, 1, 6, 2]
pop_size = 50
num_generations = 200
mutation_rate = 0.1

best_schedule, min_avg_wait = genetic_algorithm(
    burst_times=example_burst_times,
    pop_size=pop_size,
    num_generations=num_generations,
    mutation_rate=mutation_rate
)

print("\n--- Final Results (Execution Summary) ---")
print(f"Processes (IDs 0-4) Burst Times: {example_burst_times}")
print(f"Best Schedule Found: {best_schedule}")
print(f"Minimum Average Waiting Time: {min_avg_wait:.2f}")\
```

Program 3

The goal is to find the **minimum value** of a given continuous **Objective Function** $f(x)$ over a specified search domain.

Details of the Problem:

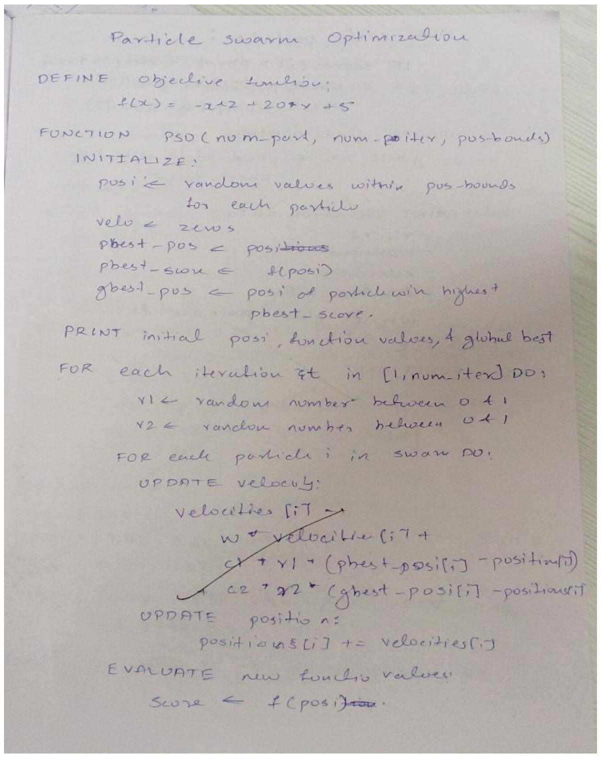
- **Optimization Type: Minimization** (since the algorithm is seeking the best position which typically corresponds to the lowest function value in standard PSO benchmarks).
- **Algorithm Used: Particle Swarm Optimization (PSO)**.
- Objective Function (Function to be Minimized):

  The function defined in the pseudocode is:

  f(x) = -x^2 + 20x + 5

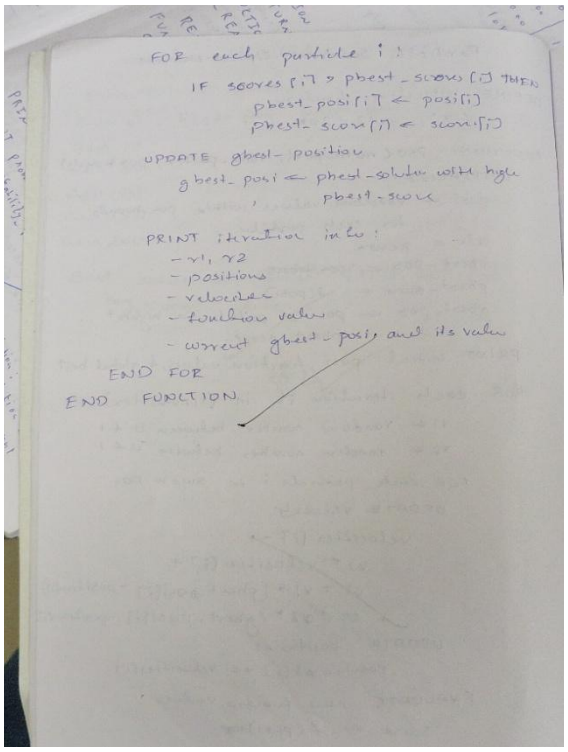
- **Solution Representation:** A **particle's position {pos}_i** represents a potential solution. Given the univariate objective function f(x), the position is a single real number (a scalar value, x).
- **Search Domain (Constraints):** The PSO algorithm requires defined boundaries for the initial particle positions. The specific boundaries are not listed in the output but are implied by the function parameter {pos_bounds} in the PSO function definition. The initial positions (e.g., 9.09, 13.23,) suggest a continuous domain, likely in the range of positive real numbers.
- **Goal:** Iteratively adjust the positions of the particles in the swarm, guided by their **personal best ({pbest})** scores and the **global best ({gbest})** score, until a particle's position yields the **minimum value** for the objective function f(x)

Algorithm

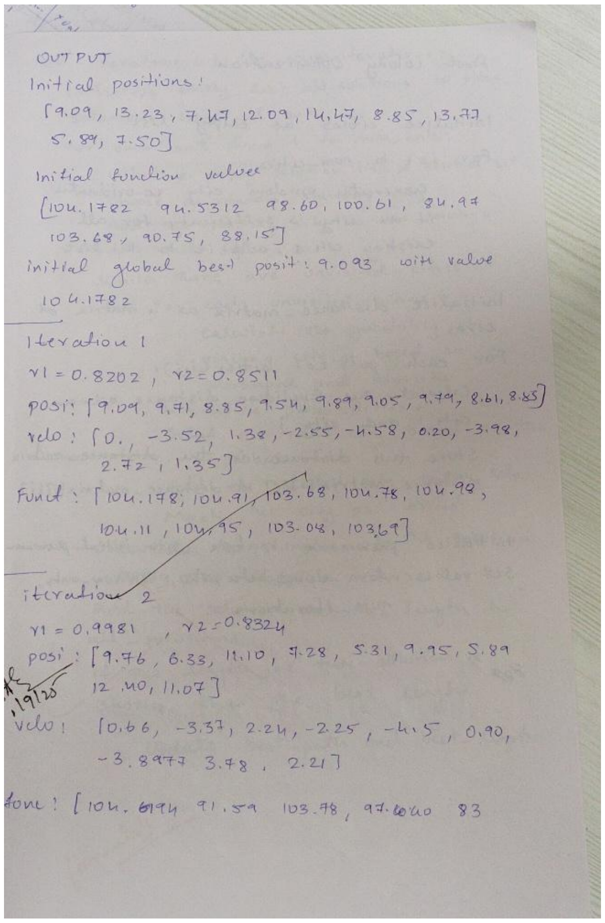


Particle swarm Optimization

```
DEFINE objective function:
    f(x) = -x^2 + 20*x + 5

FUNCTION PSO(num_part, num_~~po~~iter, pos_bounds)
  INITIALIZE:
    posi ← random values within pos_bounds
              for each particle
    velo ← zeros
    pbest_pos ← posi~~tions~~
    pbest_score ← f(posi)
    gbest_pos ← posi of particle with highest
                    pbest_score.
  PRINT initial posi, function values, & global best

  FOR each iteration it in [1, num_iter] DO:
      r1 ← random number between 0 & 1
      r2 ← random number between 0 & 1

      FOR each particle i in swarm DO:
        UPDATE velocity:
          velocities[i] ←
               w * velocities[i] +
               c1 * r1 * (pbest_posi[i] - positions[i])
             + c2 * r2 * (gbest_posi[i] - positions[i])
        UPDATE position:
             positions[i] += velocities[i]
      EVALUATE new function values:
        score ← f(posi)~~tion~~.
```



```
      FOR each particle i:
          IF scores[i] > pbest_scores[i] THEN
              pbest_posi[i] ← posi[i]
              pbest_score[i] ← score[i]

      UPDATE gbest_position
          gbest_posi ← pbest_solution with highe
                          pbest_score

      PRINT iteration info:
        - r1, r2
        - positions
        - velocities
        - function value
        - current gbest_posi, and its value
    END FOR
END FUNCTION
```



```
OUTPUT
Initial positions:
 [9.09, 13.23, 7.47, 12.09, 14.47, 8.85, 13.77
  5.89, 7.50]
Initial function values
 [104.1782  94.5312  98.60, 100.61, 84.97
  103.68, 90.75, 88.15]
initial global best posit: 9.093  with value
 104.1782

Iteration 1
r1 = 0.8202 , r2 = 0.8511
posi: [9.09, 9.71, 8.85, 9.54, 9.89, 9.05, 9.79, 8.61, 8.85]
velo: [0., -3.52, 1.38, -2.55, -4.58, 0.20, -3.98,
        2.72, 1.35]
Funct: [104.178, 104.91, 103.68, 104.78, 104.98,
        104.11, 104.95, 103.08, 103.69]

iteration 2
r1 = 0.9981 , r2 = 0.8324
posi: [9.76, 6.33, 11.10, 7.28, 5.31, 9.95, 5.89
       12.40, 11.07]
velo: [0.66, -3.37, 2.24, -2.25, -4.5, 0.90,
       -3.8977 3.78, 2.21]
func: [104.6194 91.59 103.78, 97.6040 83
```

[illegible]
19/25

Code:

```
import numpy as np

# Objective function
def f(x):
    return -x**2 + 20*x + 5

def particle_swarm_optimization(num_particles=9, num_iterations=5, pos_bounds=(0, 15)):
    # Generate random initial positions within bounds
    positions = np.random.uniform(pos_bounds[0], pos_bounds[1], size=num_particles)
    velocities = np.zeros_like(positions)

    # Initialize personal bests
    pbest_positions = positions.copy()
    pbest_scores = f(positions)

    # Initialize global best
    gbest_position = pbest_positions[np.argmax(pbest_scores)]

    # PSO hyperparameters
    c1 = c2 = 1
    w = 1

    print("Initial positions:\n", positions.round(4))
    print("Initial function values:\n", f(positions).round(4))
    print("Initial global best position:", round(gbest_position, 4), "with value:",
round(f(gbest_position), 4))
    print("-" * 60)

    # Iterate
    for t in range(num_iterations):
        r1 = np.random.rand()
        r2 = np.random.rand()

        for i in range(len(positions)):
            velocities[i] = (
                w * velocities[i] +
                c1 * r1 * (pbest_positions[i] - positions[i]) +
                c2 * r2 * (gbest_position - positions[i])
            )

        # Update positions
        positions += velocities
        scores = f(positions)

        # Update personal bests
        for i in range(len(positions)):
            if scores[i] > pbest_scores[i]:
                pbest_positions[i] = positions[i]
                pbest_scores[i] = scores[i]

        # Update global best
        gbest_position = pbest_positions[np.argmax(pbest_scores)]

        # Display iteration results
        print(f"Iteration {t + 1}")
        print("r1 =", round(r1, 4), ", r2 =", round(r2, 4))
        print("Positions:", positions.round(4))
        print("Velocities:", velocities.round(4))
        print("Function values:", scores.round(4))
        print("Global best position:", round(gbest_position, 4), "with value:", round(f(gbest_position),
4))
        print("-" * 60)

# Run the PSO function
particle_swarm_optimization()
```

Program 4

The goal is to find the minimum value of a given discrete **Objective Function** (the total tour length) over a specified search domain (all possible tours).

Details of the Problem:

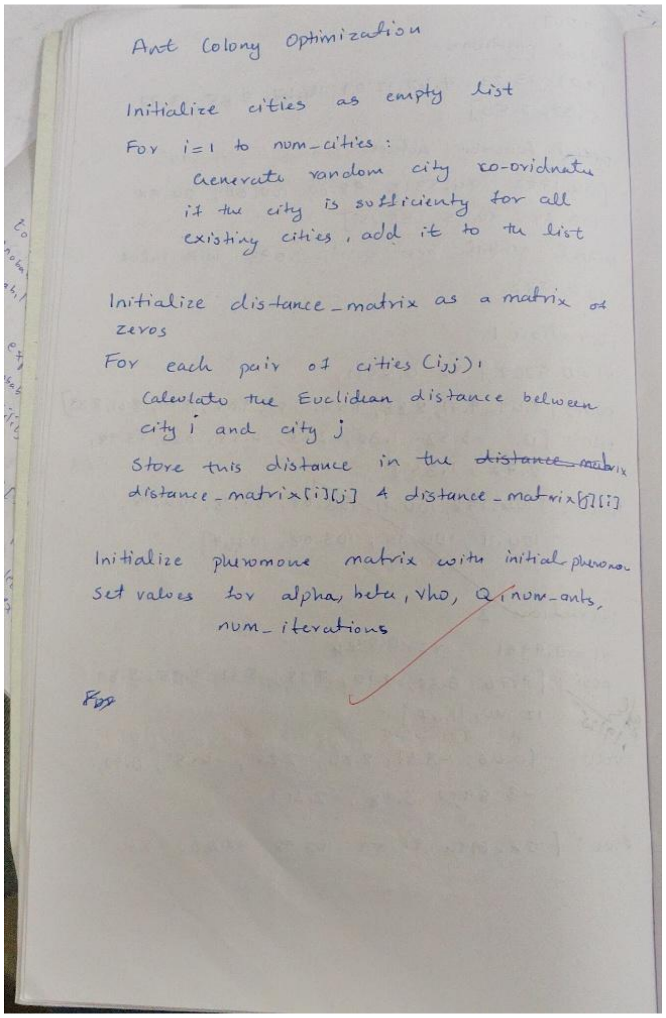
- **Optimization Type: Minimization** (The algorithm seeks the shortest path/minimum tour length).
- **Algorithm Used: Ant Colony Optimization (ACO)**.
- Objective Function (Function to be Minimized): The total length of the tour (path) taken by an ant, calculated as the sum of the distances between all consecutive cities in the tour:

  L(tour) = sum_{i=1}^{n} d(city_i, city_{i+1})

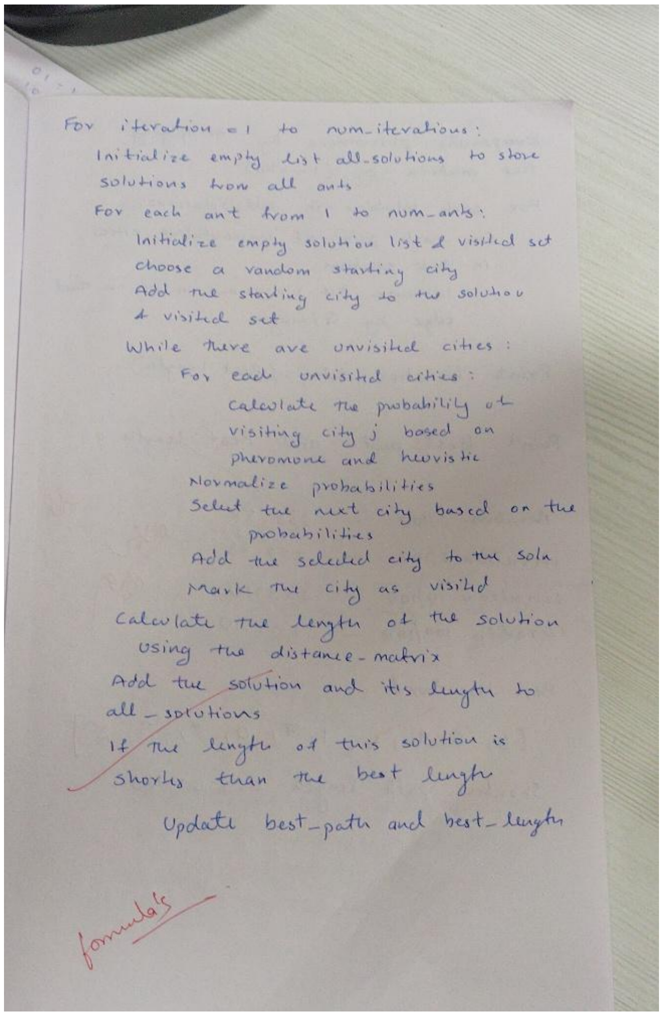
- **Solution Representation:** A **tour**, which is an ordered sequence of all cities that starts and ends at the same city (e.g., [6, 8, 9, 4, 1, 7, 0, 2, 3, 5, 6]).

- **Search Domain (Constraints):** The search domain consists of all possible **Hamiltonian cycles** (tours visiting every city exactly once) in the complete graph formed by the set of cities. The number of cities is not explicitly given, but the example solution shows 10 cities (0 to 9, plus the return to the start).
- **Goal:** Iteratively adjust the **pheromone matrix** on the edges of the city graph, guiding the artificial ants to construct a tour that yields the **minimum total tour length**, ultimately converging on the **shortest path** for the Traveling Salesman Problem. The notes show the final minimum value found was **255.09766**.

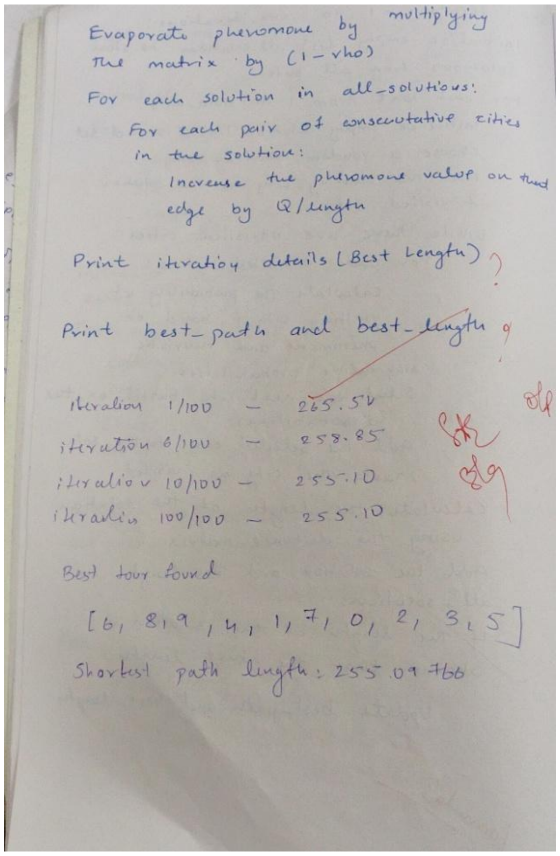
Algorithm



```
Ant Colony Optimization

Initialize cities as empty list
For i=1 to num_cities :
        Generate random city co-oridnate
        if the city is sufficienty for all
        existing cities , add it to the list

Initialize distance_matrix as a matrix of
 zeros
For each pair of cities (i,j),
    Calculate the Euclidean distance between
    city i and city j
    Store this distance in the ~~distance matrix~~
    distance_matrix[i][j] & distance_matrix[j][i]

Initialize pheromone matrix with initial pheromone
Set values for alpha, beta, rho, Q, num_ants,
        num_iterations

For
```



```
For iteration = 1 to num_iterations:
    Initialize empty list all_solutions to store
    solutions from all ants
    For each ant from 1 to num_ants:
        Initialize empty solution list & visited set
        choose a random starting city
        Add the starting city to the solution
        & visited set
        While there are unvisited cities :
            For each unvisited cities :
                Calculate the probability of
                visiting city j based on
                pheromone and heuristic
            Normalize probabilities
            Select the next city based on the
                probabilities
            Add the selected city to the soln
            Mark the city as visited
        Calculate the length of the solution
          using the distance_matrix
        Add the solution and it's length to
        all_solutions
        If the length of this solution is
        shorter than the best length
            Update best_path and best_length

formula's
```



```
Evaporate pheromone by multiplying
The matrix by (1 - rho)
For each solution in all_solutions:
    For each pair of conseautative cities
    in the solution:
        Increase the pheromone value on that
        edge by Q/length

Print iteration details (Best Length) ?

Print best_path and best_length ?
```

```
iteration 1/100   -   265.5v
iteration 6/100   -   258.85
iteration 10/100  -   255.10
iteration 100/100 -   255.10

Best tour found

[6, 8, 9, 4, 1, 7, 0, 2, 3, 5]

Shortest path length: 255.09766
```

Code:

```
import numpy as np
import random
import matplotlib.pyplot as plt

# Step 1: Define the Problem (cities and coordinates)
num_cities = 10
cities = []

# Minimum distance between cities to avoid clustering
min_distance = 5

# Generate cities ensuring they are sufficiently far apart
while len(cities) < num_cities:
    # Generate a random city
    new_city = np.random.rand(1, 2) * 100
    # Check if it's sufficiently far from all existing cities
    if all(np.linalg.norm(new_city - np.array(city)) >= min_distance for city in cities):
        cities.append(new_city[0])

cities = np.array(cities)
```

```
# Function to calculate the distance matrix
def calculate_distance_matrix(cities):
    n = len(cities)
    dist = np.zeros((n, n))
    for i in range(n):
        for j in range(n):
            dist[i][j] = np.linalg.norm(cities[i] - cities[j])
    return dist

distance_matrix = calculate_distance_matrix(cities)

# Step 2: Initialize Parameters for ACO
num_ants = 20
num_iterations = 100
alpha = 1.0         # pheromone importance
beta = 5.0          # heuristic importance
rho = 0.5           # evaporation rate
Q = 100             # constant for pheromone update
initial_pheromone = 1.0
pheromone = np.ones((num_cities, num_cities)) * initial_pheromone

# Step 3: Construct Solutions
# Function to calculate probability of visiting city j from city i
def probability(i, j, pheromone, distance_matrix, visited):
    if j in visited:
        return 0  # Don't revisit cities
    pher = pheromone[i][j] ** alpha
    heuristic = (1.0 / distance_matrix[i][j]) ** beta
    return pher * heuristic

# Function to construct a solution (tour) for an ant
def construct_solution(pheromone, distance_matrix):
    solution = []
    visited = set()
    current_city = random.randint(0, num_cities - 1)
    solution.append(current_city)
    visited.add(current_city)

    while len(visited) < num_cities:
        probs = []
        for j in range(num_cities):
            prob = probability(current_city, j, pheromone, distance_matrix, visited)
            probs.append(prob)
        probs = np.array(probs)
        probs /= probs.sum()  # normalize to form probability distribution
        next_city = np.random.choice(range(num_cities), p=probs)
        solution.append(next_city)
        visited.add(next_city)
        current_city = next_city

    return solution

# Step 4: Update Pheromones
# Function to update pheromones based on the solutions found
def update_pheromones(pheromone, all_solutions, distance_matrix):
    pheromone *= (1 - rho)  # Evaporate pheromones

    for path, length in all_solutions:
        for i in range(len(path)):
            from_city = path[i]
            to_city = path[(i + 1) % num_cities]  # Return to the starting city
            pheromone[from_city][to_city] += Q / length
            pheromone[to_city][from_city] += Q / length  # Undirected graph

# Step 5: Path Length Calculation
# Function to calculate the total length of a tour
def path_length(path, distance_matrix):
    length = 0
    for i in range(len(path)):
        from_city = path[i]
        to_city = path[(i + 1) % num_cities]  # Return to the starting city
        length += distance_matrix[from_city][to_city]
    return length

# Step 6: Main ACO Loop
best_path = None
best_length = float('inf')

for iteration in range(num_iterations):
    all_solutions = []
    for ant in range(num_ants):
        solution = construct_solution(pheromone, distance_matrix)
        length = path_length(solution, distance_matrix)
        all_solutions.append((solution, length))
        if length < best_length:
            best_length = length
            best_path = solution

    update_pheromones(pheromone, all_solutions, distance_matrix)
    print(f"Iteration {iteration+1}/{num_iterations} - Best Length: {best_length:.2f}")

# Step 7: Output the Best Solution
print("Best tour found:", best_path)
print("Shortest path length:", best_length)

# Step 8: (Optional) Plotting the Best Tour and All Cities
def plot_tour(cities, path):
    plt.figure(figsize=(8, 6))

    # Plot all the cities
    plt.scatter(cities[:, 0], cities[:, 1], color='red', marker='o', s=100, label="Cities")
    for i, (x, y) in enumerate(cities):
        plt.text(x + 1, y + 1, str(i), color='black', fontsize=12)

    # Plot the best route found by the ants (with arrows to show direction)
    tour = path + [path[0]]  # Return to the start
    for i in range(len(path)):
        start = cities[path[i]]
        end = cities[path[(i + 1) % num_cities]]
        plt.arrow(start[0], start[1], end[0] - start[0], end[1] - start[1],
                  head_width=2, head_length=3, fc='blue', ec='blue')  # Blue arrows to indicate direction

    plt.title("Best TSP Tour Found by ACO")
    plt.xlabel("X")
    plt.ylabel("Y")
    plt.legend()
    plt.grid(True)
    plt.show()

plot_tour(cities, best_path)
```

Program 5

The objective is to find the **shortest closed tour (path)** that visits every city in a given set exactly once and returns to the starting city.

This is the classic **Traveling Salesman Problem (TSP)**, which is a combinatorial optimization problem.

Detailed Formulation

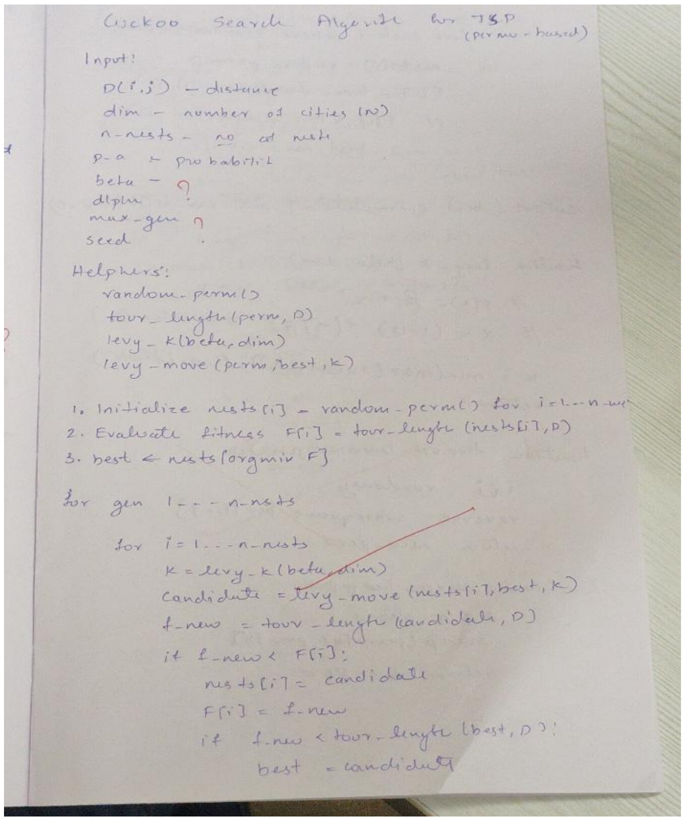
The goal is to find the minimum value of a given discrete **Objective Function** (the total tour length) over the search space of all possible permutations of city visits.

- **Optimization Type: Minimization** (The algorithm seeks the minimum total tour length).
- **Algorithm Used: Cuckoo Search Algorithm (CSA) for Permutations** (A nature-inspired metaheuristic).
- Objective Function (Function to be Minimized): The total length of the tour defined by the order of cities in a permutation.

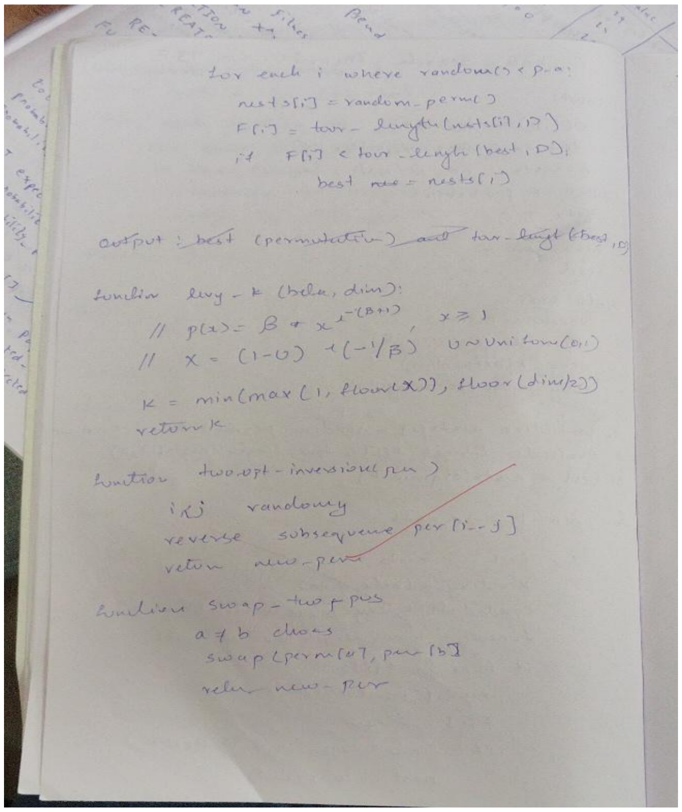
  L(tour) = sum_{i=1}^{n} d(city_i, city_{i+1})

- **Solution Representation:** A **nest** or **solution** is represented as a **permutation** ({perm}) of the cities {1, 2,……}This permutation defines the order in which the cities are visited.
- **Search Domain (Constraints):** The search domain consists of all possible permutations of the N cities, where N = (number of cities).
- **Goal:** Iteratively generate new and improved permutations using **Lévy flights** (simulated by permutation-based local search operations like {two-opt}, {insertion}, and {swap} within the {levy-move} function) until a permutation is found that yields the **minimum total tour length** (L). The final output is the {best} permutation and its {tour-length}{best},mathbf{D})

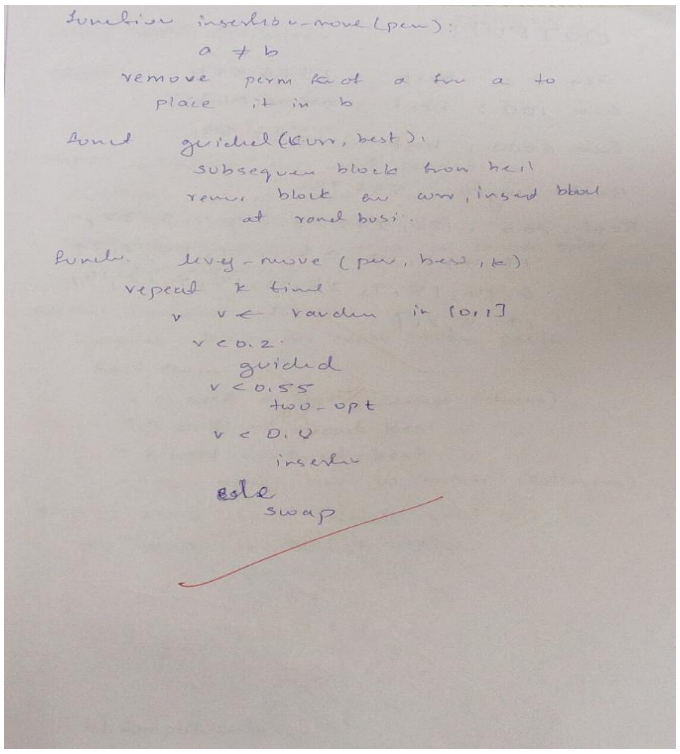
Algorithm



```
Cuckoo Search Algorithm for TSP
                              (permu - based)
Input:
    D(i,j) - distance
    dim - number of cities (N)
    n-nests - no of nests
   p-a - probabilit
   beta - ?
   alpha
  max-gen ?
  seed
Helphers:
    random-perm()
    tour_length(perm, D)
    levy_k(beta, dim)
    levy-move (perm, best, k)

1. Initialize nests[i] - random-perm() for i=1...n-nests
2. Evaluate fitness F[i] = tour-length (nests[i], D)
3. best ← nests[argmin F]

for gen 1 - - - n-nsts

    for i = 1 - - - n-nests
        k = levy_k (beta, dim)
        Candidate = levy-move (nests[i], best, k)
        f-new = tour-length (candidate, D)
        if f-new < F[i]:
            nests[i] = candidate
            F[i] = f-new
            if f-new < tour-length (best, D):
                best = candidate
```



```
        for each i where random() < p-a:
            nests[i] = random-perm()
            F[i] = tour-length(nests[i], D)
            if F[i] < tour-length(best, D):
                best ~~new~~ = nests[i]

Output: best (permutation) and tour-length (best, D)

function levy-k (beta, dim):
    // p(x) = β * x^(-(β+1)),  x ≥ 1
    // X = (1-U)^(-1/β)   U~Uniform(0,1)
    K = min(max(1, floor(X)), floor(dim/2))
    return K

function two-opt-inversion(perm)
    i<j randomly
    reverse subsequence per[i--j]
    return new-perm

function swap-two-pos
    a ≠ b chose
    swap(perm[a], perm[b])
    return new-per
```



```
function insertion-move(perm):
        a ≠ b
    remove perm at a for a to
      place it in b

funct   guided(curr, best):
        subsequen block from best
        remove block an curr, insert block
            at rand posi.

funct   levy-move (perm, best, k)
    repeat k time
        v  v ← random in [0,1]
        v < 0.2
            guided
        v < 0.55
            two-opt
        v < 0.8
            insertion
        else
            swap
```

Code:

```
"""
Cuckoo Search Algorithm (CSA) adapted for TSP (permutation-based)
- Input: distance matrix D (NxN)
- Output: best permutation (tour) and its length
- Author: concise CSA→TSP implementation
"""
import numpy as np
# ---------------------------
# Utility / operator functions
# --------------------------
def euclidean_distance_matrix(coords):
    coords = np.asarray(coords, dtype=float)
    n = coords.shape[0]
    dif = coords.reshape(n,1,-1) - coords.reshape(1,n,-1)
    return np.sqrt((dif**2).sum(axis=2))


def tour_length_from_perm(perm, D):
    perm = np.asarray(perm, dtype=int)
    return D[perm, np.roll(perm, -1)].sum()


def random_perm(n):
    return np.random.permutation(n)


def levy_k(beta, dim, cap=None):
    """
    Discrete heavy-tailed sampler (analogous to Lévy flight length).
    Returns an integer k >= 1 indicating how many permutation operators to apply.
    Uses a Pareto-like discrete sampling.
    """
    if cap is None:
        cap = max(1, dim // 2)
    # sample from pareto (numpy pareto's shape = alpha), shift by +1
    x = 1 + int(np.random.pareto(beta))
    return min(max(1, x), cap)


def two_opt_inversion(perm):
    """Random 2-opt: reverse a random subsequence."""
    n = len(perm)
    i, j = np.random.choice(n, 2, replace=False)
    if i > j:
        i, j = j, i
    new = perm.copy()
    new[i:j+1] = new[i:j+1][::-1]
    return new


def swap_two_positions(perm):
    """Swap two random positions."""
    n = len(perm)
    a, b = np.random.choice(n, 2, replace=False)
    new = perm.copy()
    new[a], new[b] = new[b], new[a]
    return new


def insertion_move(perm):
    """Remove an element and insert it at a random position."""
    n = len(perm)
    a, b = np.random.choice(n, 2, replace=False)
    new = list(perm)
    val = new.pop(a)
    new.insert(b, val)
    return np.array(new, dtype=int)


def guided_insert_from_best(curr, best):
    """
    Guided move: pick a random block from 'best' and insert it into 'curr' (preserving order),
    keeping a valid permutation. This nudges 'curr' towards 'best'.
    """
    n = len(curr)
    if n < 4:
        return two_opt_inversion(curr)
    i, j = np.random.choice(n, 2, replace=False)
    if i > j:
        i, j = j, i
    block = best[i:j+1].tolist()
    curr_list = [c for c in curr if c not in block]
    pos = np.random.randint(0, len(curr_list)+1)
    new = curr_list[:pos] + block + curr_list[pos:]
    return np.array(new, dtype=int)


def levy_move_permutation(perm, best, k):
    """
    Apply k permutation operators to perm.
    Operators are chosen probabilistically:
    - using guided insert occasionally to pull toward 'best'
    - using 2-opt, swap, insertion for diversity
    """
    new = perm.copy()
    for _ in range(k):
        r = np.random.rand()
        if r < 0.2:
            new = guided_insert_from_best(new, best)
        elif r < 0.55:
            new = two_opt_inversion(new)
        elif r < 0.8:
            new = insertion_move(new)
        else:
            new = swap_two_positions(new)
    return new
```

```
# ---------------------------
# Main CSA-TSP implementation
# ---------------------------

def cuckoo_search_tsp(D, n_nests=30, p_a=0.25, beta=1.5,
                      max_gen=1000, seed=None, verbose=False):
    """
    Cuckoo Search for TSP (permutation variant).
    D        : NxN distance matrix
    n_nests  : population size (number of nests)
    p_a      : discovery probability (fraction replaced each generation)
    beta     : Lévy-like exponent (discrete heavy-tail, e.g., 1.5)
    max_gen  : number of generations
    seed     : random seed (optional)
    Returns: best_perm (1D numpy array) and best_length (float)
    """
    if seed is not None:
        np.random.seed(seed)

    n_cities = D.shape[0]
    # initialize nests (list of numpy arrays)
    nests = [random_perm(n_cities) for _ in range(n_nests)]
    fitness = np.array([tour_length_from_perm(n, D) for n in nests])
    best_idx = np.argmin(fitness)
    best = nests[best_idx].copy()
```

```
best_f = fitness[best_idx]

if verbose:
    print(f"Initial best length: {best_f:.6f}")

for gen in range(1, max_gen + 1):
    # 1) Generate cuckoo proposals via discrete Lévy moves
    for i in range(n_nests):
        k = levy_k(beta, n_cities)
        candidate = levy_move_permutation(nests[i], best, k)
        f_new = tour_length_from_perm(candidate, D)
        if f_new < fitness[i]:
            nests[i] = candidate
            fitness[i] = f_new
            if f_new < best_f:
                best_f = f_new
                best = candidate.copy()

    # 2) Abandon fraction p_a of nests and replace with new random permutations
    replace_mask = np.random.rand(n_nests) < p_a
    for i in np.where(replace_mask)[0]:
        nests[i] = random_perm(n_cities)
        fitness[i] = tour_length_from_perm(nests[i], D)
        if fitness[i] < best_f:
            best_f = fitness[i]
```

```
                best = nests[i].copy()

        if verbose and (gen % (max(1, max_gen//10)) == 0 or gen == 1):
            print(f"Gen {gen:4d}  best = {best_f:.6f}")

    return best, best_f

# ---------------------------
# Example usage (if run as a script)
# ---------------------------

if __name__ == "__main__":
    # Example with random 30 cities
    n_cities = 30
    coords = np.random.rand(n_cities, 2) * 100.0
    D = euclidean_distance_matrix(coords)

    best_perm, best_len = cuckoo_search_tsp(
        D,
        n_nests=40,
        p_a=0.25,
        beta=1.5,
        max_gen=2000,
        seed=123,
        verbose=True
    )



    print("\nBest length:", best_len)

    print("Best tour:", best_perm.tolist())
```

Program 6

The goal is to find the minimum value of a given continuous **Objective Function** $f(x)$ over a specified search domain.

Details of the Problem:

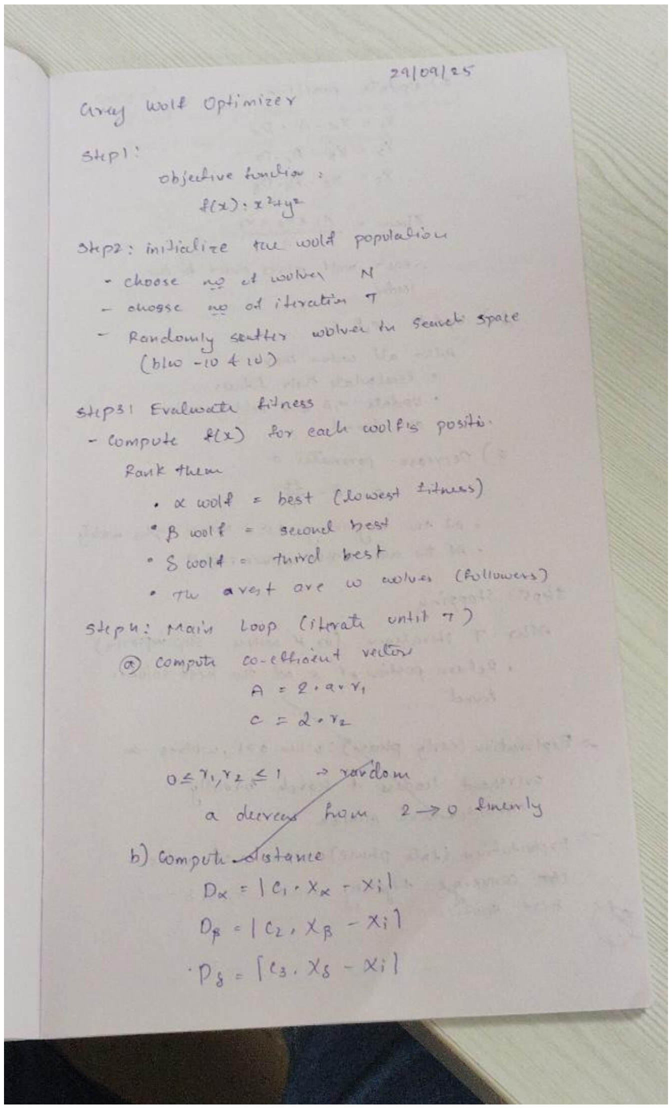
- **Optimization Type: Minimization** (The algorithm seeks the best position, which corresponds to the lowest function value, and the "best" wolf is defined as having the "lowest fitness").
- **Algorithm Used: Grey Wolf Optimizer (GWO)**.
- Objective Function (Function to be Minimized): A continuous, two-variable function:

  f(x) = x^2 y^2

  (Note: This is likely a simplification, as the main GWO steps use a single position vector X_i, which implies the function is f(X), where X is a vector of variables. The handwritten objective function f(x) = x^2 y^2 suggests a minimum value of 0 at x=0, y=0, which is a typical benchmark test case, though it is written in terms of two separate variables.)

- **Solution Representation:** A wolf's **position** (X_i) represents a potential solution within the search space. Given the objective function, the position is likely a vector of real numbers (e.g., X = [x, y]).
- **Search Domain (Constraints):** A continuous domain, specifically mentioned to be randomly initialized **between -10 and 10** for each dimension.
- **Goal:** Iteratively adjust the positions of the N wolves in the pack, guided by the best three positions (alpha, beta, and delta wolves), to converge on a position that yields the **minimum value** for the objective function f(x). The process continues until a stopping criterion (e.g., maximum number of iterations T or improvement stagnation) is met.

Algorithm



29/09/25

Grey wolf Optimizer

step1:

objective function :

$f(x) : x^2+y^2$

step2: initialize the wolf population

- choose no of wolves N
- choose no of iteration T
- Randomly scatter wolves in search space (b/w -10 & 10)

step3: Evaluate fitness

- compute $f(x)$ for each wolf's positi.

Rank them

- α wolf = best (lowest fitness)
- β wolf = second best
- δ wolf = third best
- The rest are ω wolves (followers)

step4: Main Loop (iterate until T)

ⓐ compute co-efficient vector

$A = 2 \cdot a \cdot r_1$

$C = 2 \cdot r_2$

$0 \le r_1, r_2 \le 1 \rightarrow$ random

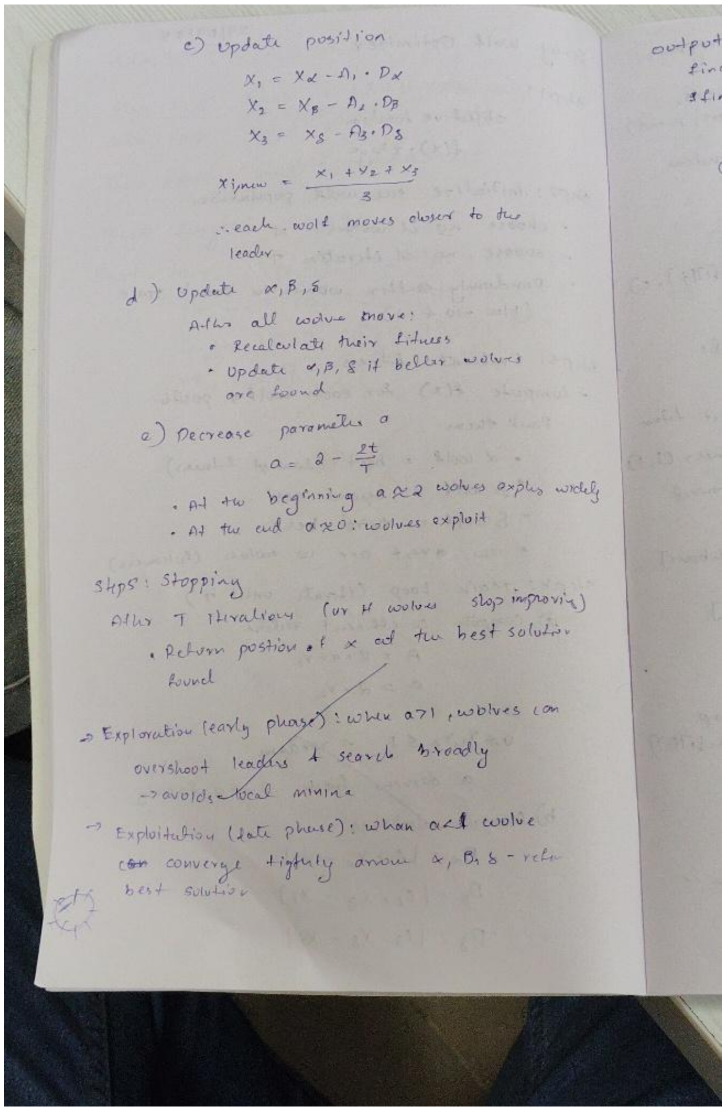
a decreas from $2 \rightarrow 0$ linearly

b) compute distance

$D_\alpha = |C_1 \cdot X_\alpha - X_i|$

$D_\beta = |C_2 \cdot X_\beta - X_i|$

$D_\delta = |C_3 \cdot X_\delta - X_i|$



c) Update position

$X_1 = X_\alpha - A_1 \cdot D_\alpha$

$X_2 = X_\beta - A_2 \cdot D_\beta$

$X_3 = X_\delta - A_3 \cdot D_\delta$

$X_{i,new} = \frac{X_1 + X_2 + X_3}{3}$

∴ each wolf moves closer to the leader

d) Update α, β, δ

After all wolves move:

- Recalculate their fitness
- Update α, β, δ if better wolves are found

e) Decrease parameter a

$a = 2 - \frac{2t}{T}$

- At the beginning $a \approx 2$ wolves explore widely
- At the end $a \approx 0$: wolves exploit

step5: Stopping

After T iterations (or if wolves stop improving)

- Return position of α as the best solution found

→ Exploration (early phase): when $a > 1$, wolves can overshoot leaders & search broadly → avoids local minima

→ Exploitation (late phase): when $a < 1$ wolve ~~can~~ converge tightly around α, β, δ - refin best solution

output

fin

3fi

Code;

```
import numpy as np




def tour_length(perm, D):

    n = len(perm)

    length = 0
    for i in range(n):
        city_from = perm[i]
        city_to = perm[(i + 1) % n]
        length += D[city_from, city_to]
    return length


def random_perm(dim):
    return np.random.permutation(dim)


def two_opt_inversion(perm, a, b):
    new_perm = perm.copy()
    sub_sequence = new_perm[a:b+1]
    new_perm[a:b+1] = sub_sequence[::-1]
    return new_perm


def insertion_move(perm, a, b):
    new_perm = list(perm.copy())
    city_to_move = new_perm.pop(a)
    new_perm.insert(b, city_to_move)
    return np.array(new_perm)


def swap_move(perm, a, b):
    new_perm = perm.copy()
    new_perm[a], new_perm[b] = new_perm[b], new_perm[a]
    return new_perm
```

```
def guided_move(curr_perm, best_perm):
    n = len(curr_perm)
    best_city_idx = np.random.randint(n)
    best_city = best_perm[best_city_idx]

    curr_city_idx = np.where(curr_perm == best_city)[0][0]

    new_perm = list(curr_perm.copy())
    new_perm.pop(curr_city_idx)
    new_perm.insert(best_city_idx, best_city)

    return np.array(new_perm)

def levy_k(beta, dim):

    sigma = (np.math.gamma(1 + beta) * np.sin(np.pi * beta / 2) /
             (np.math.gamma((1 + beta) / 2) * beta * 2**((beta - 1) / 2)))**(1 / beta)
    u = np.random.normal(0, sigma)
    v = np.random.normal(0, 1)

    X = np.abs(u / (v**(1 / beta)))

    k = np.floor(X)
    k = max(1, k)

    k = min(k, np.floor(dim / 2))


    return int(k)


def levy_move(perm, best_perm, k):

    new_perm = perm.copy()

    dim = len(perm)


    for _ in range(k):

        r = np.random.rand()


        if r < 0.2:

            new_perm = guided_move(new_perm, best_perm)

        elif r < 0.55:

            a, b = np.random.choice(dim, 2, replace=False)

            a, b = min(a, b), max(a, b)

            new_perm = two_opt_inversion(new_perm, a, b)

        elif r < 0.8:

            a, b = np.random.choice(dim, 2, replace=False)

            new_perm = insertion_move(new_perm, a, b)

        else:

            a, b = np.random.choice(dim, 2, replace=False)

            new_perm = swap_move(new_perm, a, b)
    return new_perm



def cuckoo_search_tsp(D, n_nests, p_a, beta, max_gen):
    dim = D.shape[0]

    nests = np.array([random_perm(dim) for _ in range(n_nests)])

    F = np.array([tour_length(nest, D) for nest in nests])

    best_idx = np.argmin(F)
    best_perm = nests[best_idx].copy()
    best_length = F[best_idx]

    print(f"Initial Best Length: {best_length:.4f}")

    for gen in range(1, max_gen + 1):

        for i in range(n_nests):
            k = levy_k(beta, dim)

            candidate_perm = levy_move(nests[i], best_perm, k)
            f_new = tour_length(candidate_perm, D)

            if f_new < F[i]:
                nests[i] = candidate_perm.copy()
                F[i] = f_new

                if f_new < best_length:
                    best_perm = candidate_perm.copy()
                    best_length = f_new

        for i in range(n_nests):
            if np.random.rand() < p_a:
                nests[i] = random_perm(dim)
                F[i] = tour_length(nests[i], D)

                if F[i] < best_length:
                    best_perm = nests[i].copy()
                    best_length = F[i]

        if gen % 10 == 0 or gen == max_gen:
            print(f"Generation {gen}/{max_gen} | Current Best Length: {best_length:.4f}")

    return best_perm, best_length



if __name__ == '__main__':
    np.random.seed(42)
    num_cities = 10
```

```
coords = np.random.rand(num_cities, 2) * 100


D = np.zeros((num_cities, num_cities))

for i in range(num_cities):

    for j in range(num_cities):

        if i != j:

            D[i, j] = np.linalg.norm(coords[i] - coords[j])


N_NESTS = 25

P_A = 0.25

BETA = 1.5

MAX_GEN = 100


print(f"Starting Cuckoo Search for TSP with {num_cities} cities...")


final_tour, final_length = cuckoo_search_tsp(D, N_NESTS, P_A, BETA, MAX_GEN)


print("\n==============================================")

print("Optimization Complete")

print(f"Final Shortest Path Length: {final_length:.4f}")

print(f"Best Tour Found: {final_tour.tolist()}")

print("==============================================")
```

Program 7

The goal is to find the minimum value of a discrete **Objective Function** (the total tour length) over the search space of all possible tours.

This is the Traveling Salesman Problem (TSP), defined as:

Given a set of cities and the Euclidean distance between each pair, find the shortest closed tour that visits every city exactly once and returns to the starting city.

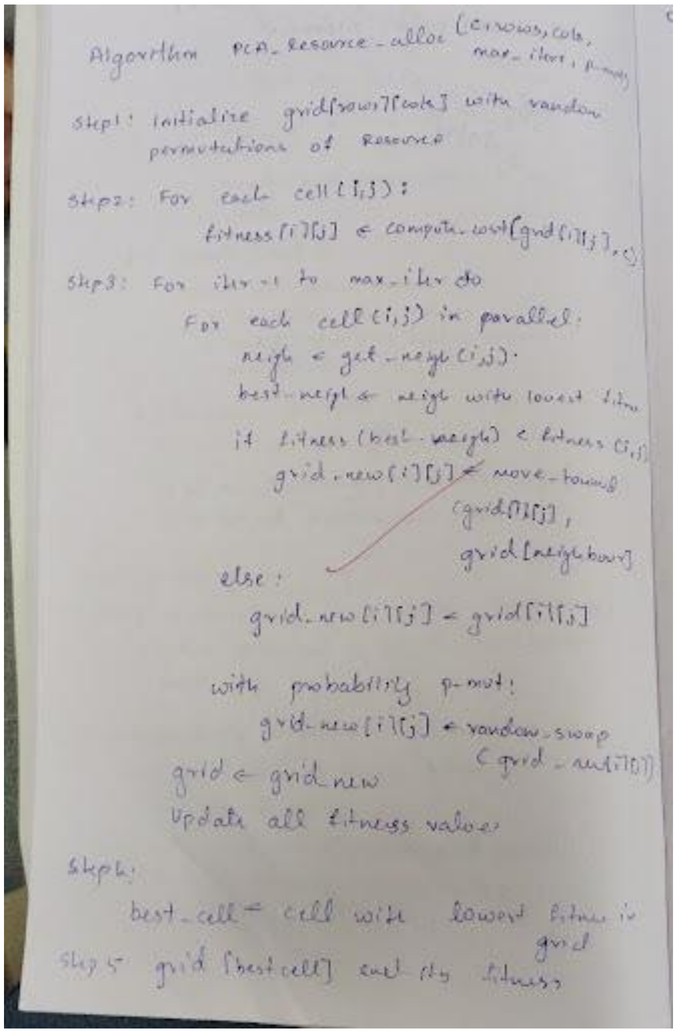
Details of the Problem:

- **Optimization Type: Minimization** (The algorithm seeks the shortest path/minimum tour length).
- **Algorithm Used: Ant Colony Optimization (ACO)**.
- Objective Function (Function to be Minimized): The total length of the tour taken by an ant:

  $$L(\text{tour}) = \sum_{i=1}^{n} d(\text{city}_i, \text{city}_{i+1})$$

- **Solution Representation:** A **tour**, which is an ordered sequence of all cities that starts and ends at the same city (e.g., `[6, 8, 9, 4, 1, 7, 0, 2, 3, 5]`).
- **Search Domain (Constraints):** All possible Hamiltonian cycles (closed paths visiting every node once) in the complete graph of cities.
- **Goal:** Iteratively adjust the **pheromone matrix** to guide artificial ants to converge on the permutation of cities that yields the **minimum total tour length**.

Algorithm



```
Algorithm PCA_Resource_alloc (C, rows, cols,
                              max_iters, p_mut)

Step1: Initialize grid[rows][cols] with random
       permutations of Resource

Step2: For each cell (i,j):
           fitness[i][j] ← compute_cost(grid[i][j], C)

Step3: For iter ← 1 to max_iter do
           For each cell (i,j) in parallel:
               neigh ← get_neigh (i,j).
               best_neigh ← neigh with lowest fitness
               if fitness (best_neigh) < fitness (i,j)
                   grid_new[i][j] ← move_towards
                                    (grid[i][j],
                                    grid[neighbour])
               else:
                   grid_new[i][j] ← grid[i][j]

               with probability p_mut:
                   grid_new[i][j] ← random_swap
                                    (grid_new[i][j])
           grid ← grid_new
           Update all fitness values

Step4:
    best_cell ← cell with lowest fitness in
                grid
Step5 grid [best cell] and its fitness
```

Code:

import random

import numpy as np

import math

from multiprocessing import Pool

# Generate random cities

def generate_cities(num_cities):

```
    return [(random.uniform(0, 100), random.uniform(0, 100)) for _ in range(num_cities)]


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


# Total distance of a route
def total_distance(route, cities):
    dist = 0
    for i in range(len(route) - 1):
        dist += distance(cities[route[i]], cities[route[i+1]])
    dist += distance(cities[route[-1]], cities[route[0]])  # return to start city
    return dist


# Fitness function (inverse of the total distance)
def fitness(route, cities):
    return 1 / total_distance(route, cities)


# Initialize a population (random routes)
def initialize_population(num_cells, num_cities):
    return [random.sample(range(num_cities), num_cities) for _ in range(num_cells)]


# Get neighbors of a cell (simplified to adjacent cells in the list)
def get_neighbors(population, index):
    neighbors = []
    if index > 0:
        neighbors.append(population[index - 1])
    if index < len(population) - 1:
        neighbors.append(population[index + 1])
    return neighbors


# Update state of a cell by moving towards the best neighbor (simplified approach)
def update_state(cell, neighbors, cities):
    best_neighbor = min(neighbors, key=lambda x: total_distance(x, cities))
    # Randomly swap a portion of the route to simulate an update
    swap_indices = random.sample(range(len(cell)), 2)
    new_cell = cell[:]
    new_cell[swap_indices[0]], new_cell[swap_indices[1]] = new_cell[swap_indices[1]],
new_cell[swap_indices[0]]
    return new_cell


# Parallel fitness evaluation function
def evaluate_cell(cell, cities):
    return fitness(cell, cities)


# Parallel update function (simplified)
def parallel_update_population(population, cities):
    with Pool() as pool:
        fitness_values = pool.starmap(evaluate_cell, [(cell, cities) for cell in population])
    updated_population = []
    for i in range(len(population)):
        neighbors = get_neighbors(population, i)
        updated_population.append(update_state(population[i], neighbors, cities))

    return updated_population

# Main parallel cellular algorithm
def parallel_cellular_algorithm(num_cities, num_cells, iterations):
    cities = generate_cities(num_cities)
    population = initialize_population(num_cells, num_cities)

    for _ in range(iterations):
        population = parallel_update_population(population, cities)

    # Track the best solution found
    best_solution = min(population, key=lambda x: total_distance(x, cities))
    return best_solution, total_distance(best_solution, cities)

# Running the algorithm
best_route, best_distance = parallel_cellular_algorithm(10, 50, 100)

print(f"Best Route: {best_route}")
print(f"Total Distance: {best_distance}")
```