VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB RECORD

Bio Inspired Systems (23CS5BSBIS)

Submitted by

Disha H Jain (1BM23CS095)

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 **Disha H Jain (1BM23CS095),** who is a bonafide student of **B.M.S. College of Engineering.** It is in partial fulfillment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the Visvesvaraya Technological University, Belgaum. The Lab report has been approved as it satisfies the academic requirements of the above mentioned subject and the work prescribed for the said degree.

Assistant Professor
Department of CSE, BMSCE

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

Mayanka Gupta

Index

Github Link:

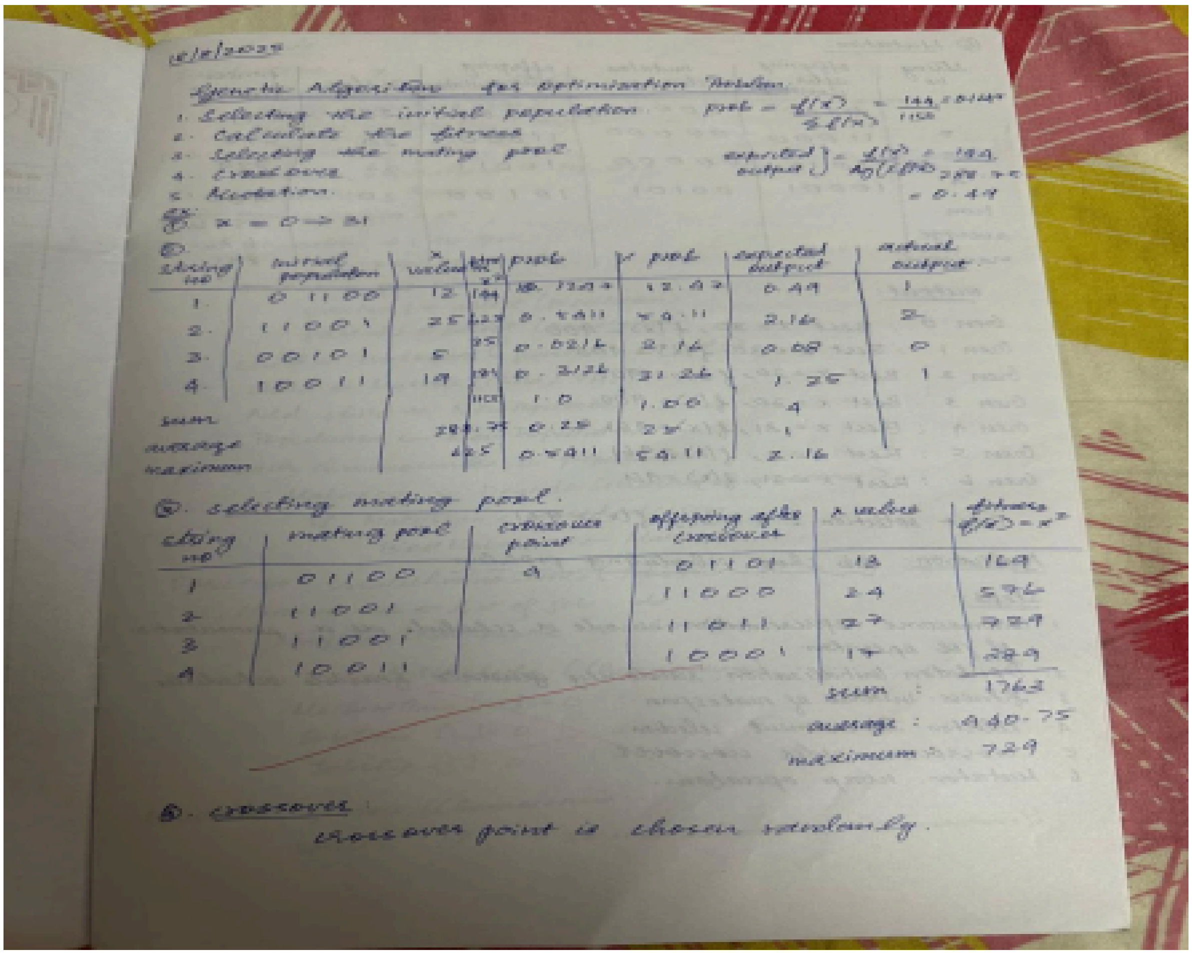
https://github.com/dishahjain/BIS-LAB

Program 1 : Genetic Algorithm

Problem statement:

Genetic Algorithms (GA) are inspired by the process of natural selection and genetics, where the fittest individuals are selected for reproduction to produce the next generation. GAs are widely used for solving optimization and search problems.

Algorithm:



Genetic Algorithm for optimization Problem

1. Selecting the initial population
2. Calculate the fitness
3. Selecting the mating pool
4. Crossover
5. Mutation

$\text{prob} = \frac{f(x)}{\sum f(x)} = \frac{144}{1155} = 0.1247$

$\left.\begin{matrix}\text{expected}\\ \text{output}\end{matrix}\right\} = \frac{f(x)}{Avg(\sum f(x))} = \frac{144}{288.75} = 0.49$

Ex: $x = 0 \rightarrow 31$

①

string no	initial population	x value	fitness x^2	prob	% prob	expected output	actual output
1.	0 1 1 0 0	12	144	0.1247	12.47	0.49	1
2.	1 1 0 0 1	25	625	0.5411	54.11	2.16	2
3.	0 0 1 0 1	5	25	0.0216	2.16	0.08	0
4.	1 0 0 1 1	19	361	0.3126	31.26	1.25	1
sum			1155	1.0	1.00	4	
average			288.75	0.25	25	1	
maximum			625	0.5411	54.11	2.16	

②. selecting mating pool.

string no	mating pool	crossover point	offspring after crossover	x value	fitness $f(x) = x^2$
1	0 1 1 0 0	4	0 1 1 0 1	13	169
2	1 1 0 0 1		1 1 0 0 0	24	576
3	1 1 0 0 1		1 1 0 1 1	27	729
4	1 0 0 1 1		1 0 0 0 1	17	289
				sum :	1763
				average :	440.75
				maximum :	729

③. crossover

crossover point is chosen randomly.

⑥ Mutation.

String no	offspring after crossover	mutation chromosome	offspring after mutation	x value	fitness $f(x) = x^2$
1	01101	10000	11101	29	841
2	11000	00000	11000	24	576
3	11011	00000	11011	27	729
4	10001	00101	10100	20	400
sum					2546
average					636.5
maximum					841

Output:

Gen 0 : Best $x = 30$, $f(x) = 900$
Gen 1 : Best $x = 30$, $f(x) = 900$
Gen 2 : Best $x = 30$, $f(x) = 900$
Gen 3 : Best $x = 30$, $f(x) = 900$
Gen 4 : Best $x = 31$, $f(x) = 961$
Gen 5 : Best $x = 31$, $f(x) = 961$
Gen 6 : Best $x = 31$, $f(x) = 961$

Best selection : $x = 31$, $f(x) = 961$

Application: Job shop scheduling problem

Steps:
1. Chromosome representation: encode a schedule as a permutation of job operation
2. Population initialization: randomly generate feasible schedules.
3. fitness: inverse of makespan
4. selection: Tournament selection.
5. crossover: order crossover
6. Mutation: swap operations.

Pseudocode:
Input: Jobs
Population
No. of g
Mutation
Function Generat
Initializ
Bestsolu
Best
For genera
For
Ch
C
A
For
Fun
Fun
Fun

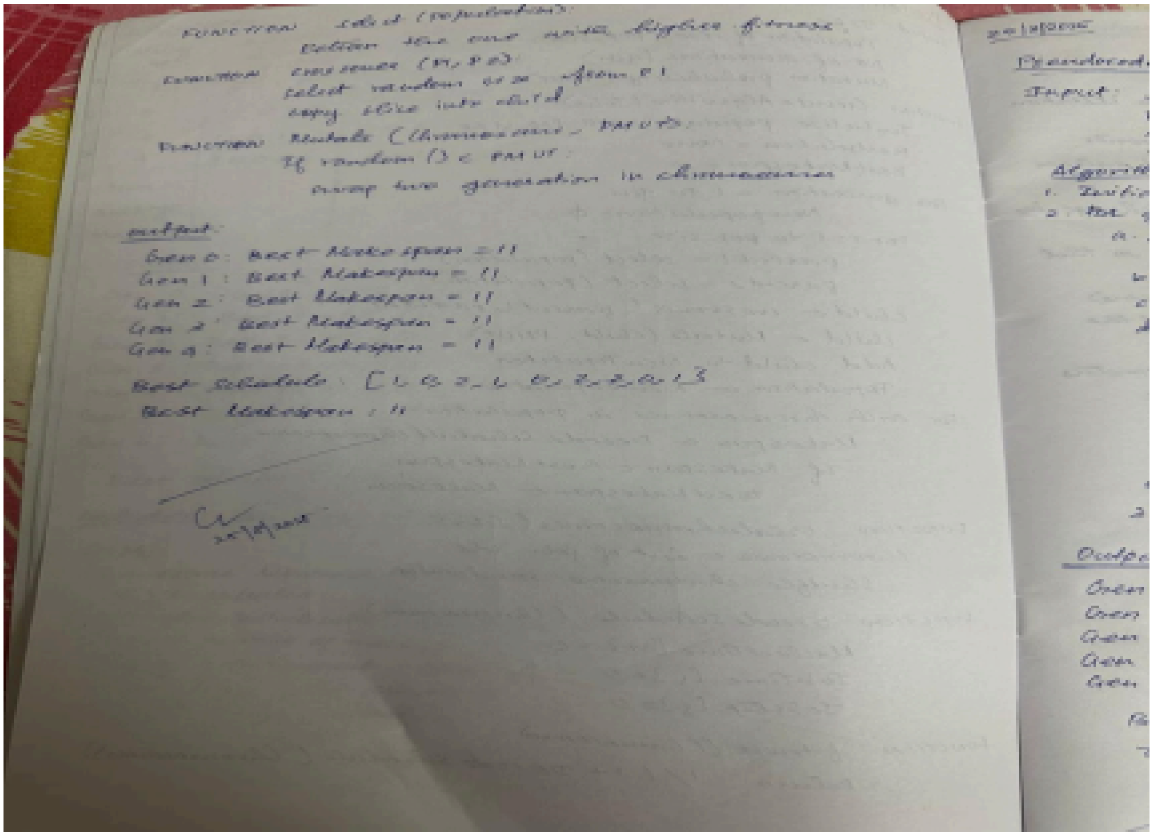
value	fitness $f(x)=x^2$
	841
	576
	729
	400
	2546
	636.5
	841

cumulative

schedules.

Pseudocode:

```
Input: Jobs
        Population of size (pop-size)
        No of generations (gen)
        Mutation probability (PMUT)
Function GeneticAlgorithm (Jobs)
    Initialize population with Pop-size
    BestSolution = None
    BestMakespan = ∞
  For generation = 1 to gen.
        Newpopulation = ϕ
    For i = 1 to pop-size
        parent 1 = select (population)
        parent 2 = select (population)
    child ← crossover (parent 1, parent 2)
    child ← Mutate (Child, PMUT)
    Add child to NewPopulation
    Population ← NewPopulation
  For each chromosomes in population:
        Makespan ← Decode-Schedule(Chromosome)
        If Makespan < BestMakespan:
            BestMakespan ← Makespan
FUNCTION createchromosomes (Jobs).
    Chromosome ← list of job ids
    Shuffle chromosome randomly
FUNCTION DecodeSchedule (Chromosome)
        MachineTime [m] = 0
        JobTime [j] = 0
        JobClip [j] = 0
FUNCTION fitness (Chromosome):
    Return 1/(1 + DecodeSchedule (Chromosome))
```



FUNCTION select (population):
return the one with highest fitness
FUNCTION crossover (P1, P2)
select random size from P1
copy slice into child
FUNCTION Mutate (chromosome, RATE):
If random () < RATE:
swap two genes in chromosome

output:
Gen 0: Best Make span = 11
Gen 1: Best Makespan = 11
Gen 2: Best Makespan = 11
Gen 3: Best Makespan = 11
Gen 4: Best Makespan = 11
Best Schedule:
Best Makespan : 11

Code:

```
import random
def fitness(x):
 return x**2
def int_to_bin(x):
 return format(x, '05b')
def bin_to_int(b):
 return int(b, 2)
def tournament_selection(pop, k=3):
 selected = random.sample(pop, k)
 selected.sort(key=lambda x: fitness(x), reverse=True)
 return selected[0]
def crossover(p1, p2):
 b1, b2 = int_to_bin(p1), int_to_bin(p2)
 point = random.randint(1, 4)
 child1 = bin_to_int(b1[:point] + b2[point:])
 child2 = bin_to_int(b2[:point] + b1[point:])
 return child1, child2
def mutate(x, mutation_rate=0.1):
 if random.random() < mutation_rate:
 b = list(int_to_bin(x))
 pos = random.randint(0, 4)
 b[pos] = '1' if b[pos] == '0' else '0'
 return bin_to_int("".join(b))
 return x
def genetic_algorithm(initial_population=None, pop_size=6, generations=20, crossover_rate=0.8,
mutation_rate=0.1):
 if initial_population:
```

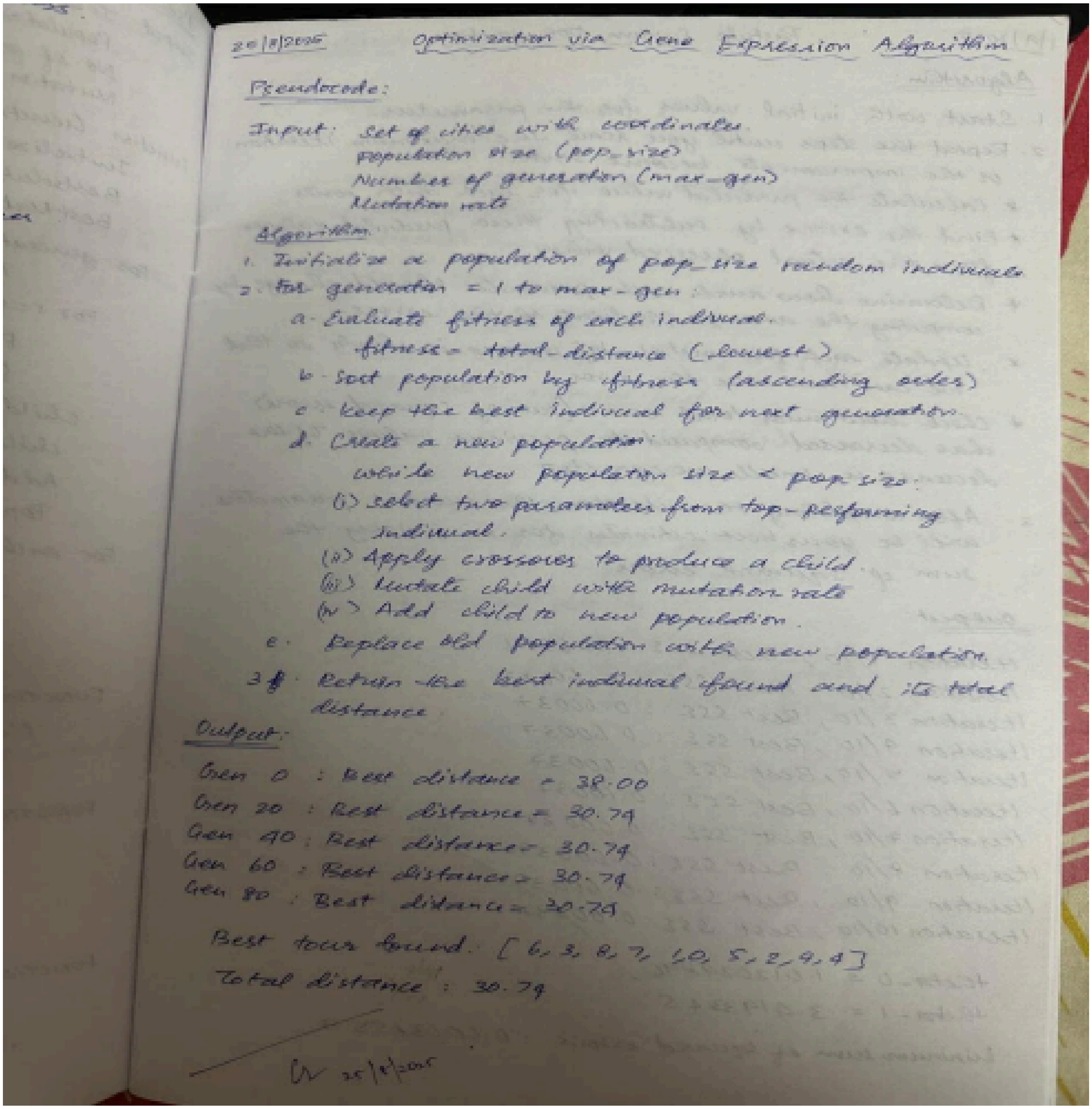
```
population = initial_population[:pop_size] # take only needed size
else:
population = [random.randint(0, 31) for _ in range(pop_size)]
for gen in range(generations):
population.sort(key=lambda x: fitness(x), reverse=True)
best = population[0]
print(f"Gen {gen}: Best x={best}, f(x)={fitness(best)}")
new_pop = [best]
while len(new_pop) < pop_size:
parent1 = tournament_selection(population)
parent2 = tournament_selection(population)
if random.random() < crossover_rate:
child1, child2 = crossover(parent1, parent2)
else:
child1, child2 = parent1, parent2
child1 = mutate(child1, mutation_rate)
child2 = mutate(child2, mutation_rate)
new_pop.extend([child1, child2])
population = new_pop[:pop_size]
population.sort(key=lambda x: fitness(x), reverse=True)
best = population[0]
print(f"\nBest Solution: x={best}, f(x)={fitness(best)}")
custom_population = [3, 7, 15, 20, 25, 30]
genetic_algorithm(initial_population=custom_population, generations=5)
```

Program 2 : Optimization via Gene expression

Problem statement:

Gene Expression Algorithms (GEA) are inspired by the biological process of gene expression in living organisms. This process involves the translation of genetic information encoded in DNA into functional proteins. In GEA, solutions to optimization problems are encoded in a manner similar to genetic sequences. The algorithm evolves these solutions through selection, crossover, mutation, and gene expression to find optimal or near-optimal solutions. GEA is effective for solving complex optimization problems in various domains, including engineering, data analysis, and machine learning.

Algorithm:


20/8/2025 Optimization via Gene Expression Algorithm

Pseudocode:

Input: Set of cities with coordinates
Population size (pop_size)
Number of generation (max_gen)
Mutation rate

Algorithm

1. Initialize a population of pop_size random individuals
2. for generation = 1 to max_gen:
   a. Evaluate fitness of each individual.
      fitness = total_distance (lowest)
   b. Sort population by fitness (ascending order)
   c. Keep the best individual for next generation
   d. Create a new population
      while new population size < pop_size:
      (i) select two parameters from top-performing individual.
      (ii) Apply crossover to produce a child.
      (iii) Mutate child with mutation rate
      (iv) Add child to new population.
   e. Replace old population with new population.
3. Return the best individual found and its total distance

Output:

Gen 0 : Best distance = 38.00
Gen 20 : Best distance = 30.79
Gen 40 : Best distance = 30.79
Gen 60 : Best distance = 30.79
Gen 80 : Best distance = 30.79

Best tour found: [6, 3, 8, 7, 10, 5, 2, 9, 4]
Total distance : 30.79

25/8/2025

Code:

```
import random
import math
cities = [
 (0, 0), (1, 5), (5, 2), (6, 6), (8, 3),
 (2, 1), (7, 7), (3, 3), (4, 4), (9, 0)
]
def distance(a, b):
 return math.sqrt((a[0]-b[0])**2 + (a[1]-b[1])**2)
def total_distance(tour):
 dist = 0
 for i in range(len(tour)):
 city_a = cities[tour[i]]
 city_b = cities[tour[(i+1) % len(tour)]]
 dist += distance(city_a, city_b)
```

```
 return dist
def create_individual(n):
 gene = list(range(n))
 random.shuffle(gene)
 return gene
def mutate(individual, rate=0.1):
 ind = individual[:]
 for i in range(len(ind)):
 if random.random() < rate:
 j = random.randint(0, len(ind)-1)
 ind[i], ind[j] = ind[j], ind[i]
 return ind
def crossover(parent1, parent2):
 size = len(parent1)
 a, b = sorted([random.randint(0, size-1) for _ in range(2)])
child = [None]*size
 child[a:b+1] = parent1[a:b+1]
 p2_index = 0
 for i in range(size):
 if child[i] is None:
 while parent2[p2_index] in child:
 p2_index += 1
 child[i] = parent2[p2_index]
 return child
def genetic_algorithm(generations=100, pop_size=100, mutation_rate=0.1):
num_cities = len(cities)
 population = [create_individual(num_cities) for _ in range(pop_size)]
best = None
 best_dist = float('inf')
 for gen in range(generations):
 scored = [(ind, total_distance(ind)) for ind in population]
scored.sort(key=lambda x: x[1])
 if scored[0][1] < best_dist:
 best = scored[0][0]
 best_dist = scored[0][1]
 new_pop = [best]
 while len(new_pop) < pop_size:
 p1 = random.choice(scored[:50])[0]
 p2 = random.choice(scored[:50])[0]
 child = crossover(p1, p2)
 child = mutate(child, mutation_rate)
 new_pop.append(child)
 population = new_pop
 if gen % 20 == 0:
 print(f"Gen {gen}: Best distance = {best_dist:.2f}")
 return best, best_dist
best_tour, best_dist = genetic_algorithm()
print("\nBest tour found:")
print(best_tour)
print(f"Total distance: {best_dist:.2f}")
```

Program 3 : Particle swarm Optimization

Problem statement:

Particle Swarm Optimization (PSO) is inspired by the social behavior of birds flocking or fish schooling. PSO is used to find optimal solutions by iteratively improving a candidate solution with regard to a given measure of quality.

Algorithm:

1/9/2025 Particle Swarm Optimization

Algorithm:

1. Start with initial values for the parameters.
2. Repeat the steps until you reach the maximum iteration or the improvements become small.
   * Calculate the predicted value for each data points
   * Find the errors by subtracting these predicted values from the actual observed values.
   * Determine how much each parameter should change by computing the average gradients of the errors.
   * Update each parameter by moving it slowly so that we can minimize the error.
   * Check how much total error (sum of squared errors) has decreased compared to previous values. If the decrease is small we can stop
3. After completing the iteration, the current parameters will be your best estimates for minimizing the sum of squared errors

Output:

Iteration 1/10, Best SSE : 30.63716
Iteration 2/10, Best SSE : 0.60037
Iteration 3/10, Best SSE : 0.60037
Iteration 4/10, Best SSE : 0.60037
Iteration 5/10, Best SSE : 0.60037
Iteration 6/10, Best SSE : 0.60037
Iteration 7/10, Best SSE : 0.60037
Iteration 8/10, Best SSE : 0.60037
Iteration 9/10, Best SSE : 0.60037
Iteration 10/10, Best SSE 0.60037

theta_0 = 1.012029896
theta_1 = 3.4193345

Minimum sum of squared errors : 0.60036557

MG 1/9

Pseudoco

Initialize

For

Pseudocode:

Initialize
- Number of particles: N
- D (length of θ)
- Particle position θi within bounds
- Particle velocities vi
- Personal best solution

For each particle i

1. evaluate fitness:
   - $J(\theta_i) = \Sigma (y_j - f(x_j ; \theta_i))^2$
2. update personal best
   If $J(\theta_i) < J(best\_i)$
   $pBest\_i = \theta_i$
3. update global best:
   If $J(pbest_i) < J(gbest)$.
   $gBest = pBest\_i$
4. update velocity
   $v_i = \omega * v_i$
   $+ c_1 * rand() * (pbest_i - \theta_i)$
   $+ c_2 * rand() * (gBest - \theta_i)$
5. update position
   $\theta_i = \theta_i + v_i$
6. Return gBest as the optimal θ

Code:

```
import numpy as np
x_data = np.array([1, 2, 3, 4, 5])
y_data = np.array([3, 5, 7, 9, 11])
def objective_function(theta):
 theta_0, theta_1 = theta
 predictions = theta_0 + theta_1 * x_data
 errors = y_data - predictions
 return np.sum(errors**2)
num_particles = 30
num_iterations = 10
w = 0.7
c1 = 1.5
c2 = 2.1
bounds = [(-10, 10), (-10, 10)]
positions = np.array([np.random.uniform(low, high, num_particles) for low, high in bounds]).T
```

```
velocities = np.random.uniform(-1, 1, (num_particles, 2))
personal_best_positions = np.copy(positions)
personal_best_values = np.array([objective_function(p) for p in personal_best_positions])
best_particle_index = np.argmin(personal_best_values)
global_best_position = personal_best_positions[best_particle_index]
global_best_value = personal_best_values[best_particle_index]
for iteration in range(num_iterations):
 for i in range(num_particles):
 fitness = objective_function(positions[i])
 if fitness < personal_best_values[i]:
 personal_best_values[i] = fitness
 personal_best_positions[i] = positions[i]
 if fitness < global_best_value:
 global_best_value = fitness
 global_best_position = positions[i]
 for i in range(num_particles):
 r1 = np.random.rand(2)
 r2 = np.random.rand(2)
 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] += velocities[i]
 for dim in range(2):
 positions[i, dim] = np.clip(positions[i, dim], bounds[dim][0], bounds[dim][1])
print(f"Iteration {iteration+1}/{num_iterations}, Best SSE: {global_best_value:.5f}")
print("\nBest parameters found:")
print("theta_0 =", global_best_position[0])
print("theta_1 =", global_best_position[1])
print("Minimum sum of squared errors:", global_best_value)
```

Program 4 : Ant Colony Optimization

Problem statement:

The foraging behavior of ants has inspired the development of optimization algorithms that can solve complex problems such as the Traveling Salesman Problem (TSP). Ant Colony Optimization (ACO) simulates the way ants find the shortest path between food sources and their nest. Implement the ACO algorithm using Python to solve the TSP, where the objective is to find the shortest possible route that visits a list of cities and returns to the origin city.

Algorithm:

8/9/2025 Ant Colony Optimization - Delivery Route

Pseudocode:

Input :

Num_cities - No of cities
Num_ants - No of ants
Num_Iteration - No of Iteration
Alpha - Influence of pheromone
Beta - Influence of distance
Q - constant for pheromone deposit.

best_path - Path with the shortest distance found.
best_distance - total distance of the best path

Function ACO_TSP(num_cities, num_ants, num_iterations, alpha, beta, evaporation, Q)

cities ← GenerateRandomCities (num_cities)
distance_matrix ← ComputeDistanceMatrix (cities)
pheromone_matrix ← InitializePheromoneMatrix (num_cities)
best_path ← [ ]
best_distance ← ∞
for iteration from 1 to num_iteration do:
all_path ← [ ]
all_distance ← [ ]
for ant from 1 to num_ants do:
path ← ConstructionSolution (pheromone_matrix, distance_matrix, alpha, beta)
distance ← CalculateTotalDistance (path, distance_matrix
all_paths.append (path)
all_distance ~~=~~ .append (distance)
If distance < best_distance
best_distance ← distance
best_path ← path

pheromone_...

Return b...

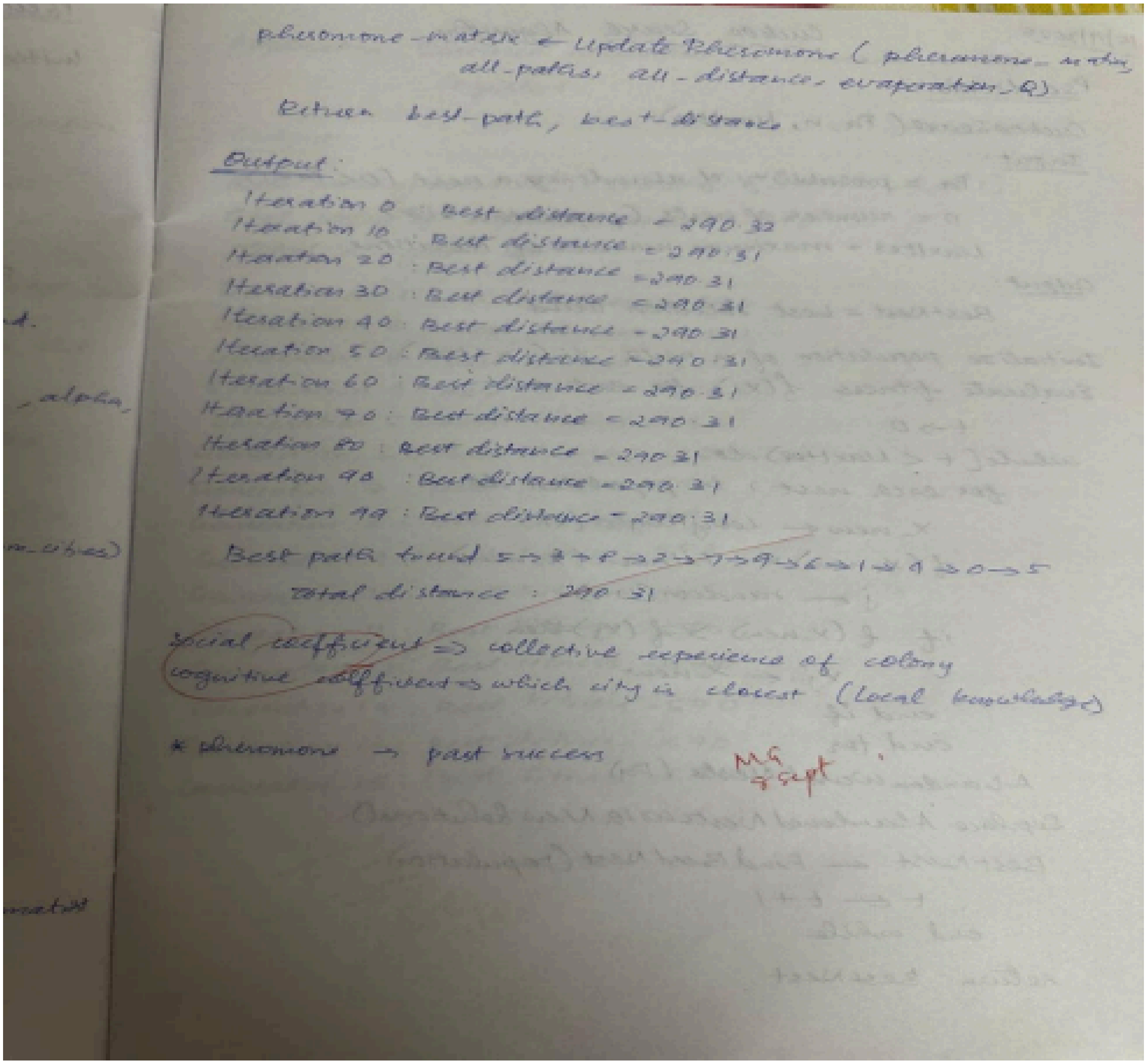
Output:

Iteration 0
Iteration 10
Iteration 20
Iteration 30
Iteration 40
Iteration 50
Iteration 60
Iteration 70
Iteration 80
Iteration 90
Iteration 9...

Best path...
Total...

social coeff...
cognitive co...

* pheromone...



pheromone_matrix ← UpdatePheromone (pheromone_matrix, all_paths, all_distance, evaporation, Q)

Return best_path, best_distance

Output:

Iteration 0 : Best distance = 290.32
Iteration 10 : Best distance = 290.31
Iteration 20 : Best distance = 290.31
Iteration 30 : Best distance = 290.31
Iteration 40 : Best distance = 290.31
Iteration 50 : Best distance = 290.31
Iteration 60 : Best distance = 290.31
Iteration 70 : Best distance = 290.31
Iteration 80 : Best distance = 290.31
Iteration 90 : Best distance = 290.31
Iteration 99 : Best distance = 290.31

Best path found 5→3→8→2→7→9→6→1→4→0→5
Total distance : 290.31

Social coefficient ⇒ collective experience of colony
cognitive coefficient → which city is closest (local knowledge)

* pheromone → past success

MG
8 sept

Code:

```
import numpy as np
import random
NUM_CITIES = 10
NUM_ANTS = 20
NUM_ITERATIONS = 100
ALPHA = 1.0
BETA = 5.0
EVAPORATION = 0.5
Q = 100
np.random.seed(42)
cities = np.random.rand(NUM_CITIES, 2) * 100
dist_matrix = np.sqrt(((cities[:, np.newaxis, :] - cities[np.newaxis, :, :]) ** 2).sum(axis=2))
pheromone = np.ones((NUM_CITIES, NUM_CITIES))
best_distance = float('inf')
best_path = []
for iteration in range(NUM_ITERATIONS):
```

```
all_paths = []
all_distances = []
for ant in range(NUM_ANTS):
path = [random.randint(0, NUM_CITIES - 1)]
while len(path) < NUM_CITIES:
current_city = path[-1]
probabilities = []
for next_city in range(NUM_CITIES):
if next_city not in path:
tau = pheromone[current_city][next_city] ** ALPHA
eta = (1 / dist_matrix[current_city][next_city]) ** BETA
probabilities.append(tau * eta)
else:
probabilities.append(0)
probabilities = np.array(probabilities)
probabilities /= probabilities.sum()
next_city = np.random.choice(range(NUM_CITIES), p=probabilities)
path.append(next_city)
path.append(path[0]) # Return to starting city
distance = sum(dist_matrix[path[i]][path[i + 1]] for i in range(NUM_CITIES))
all_paths.append(path)
all_distances.append(distance)
if distance < best_distance:
best_distance = distance
best_path = path
pheromone *= (1 - EVAPORATION)
for i in range(NUM_ANTS):
for j in range(NUM_CITIES):
from_city = all_paths[i][j]
to_city = all_paths[i][j + 1]
pheromone[from_city][to_city] += Q / all_distances[i]
pheromone[to_city][from_city] += Q / all_distances[i]
if iteration % 10 == 0 or iteration == NUM_ITERATIONS - 1:
print(f"Iteration {iteration}: Best Distance = {best_distance:.2f}")
print("\nBest Path Found:")
print(" -> ".join(map(str, best_path)))
print(f"Total Distance: {best_distance:.2f}")
```

Program 5 : Cuckoo search Optimization

Problem statement:

Cuckoo Search (CS) is a nature-inspired optimization algorithm based on the brood parasitism of some cuckoo species. This behavior involves laying eggs in the nests of other birds, leading to the optimization of survival strategies. CS uses Lévy flights to generate new solutions, promoting global search capabilities and avoiding local minima. The algorithm is widely used for solving continuous optimization problems and has applications in various domains, including engineering design, machine learning, and data mining.

Algorithm:

15/9/2025. Cuckoo Search Algorithm

Pseudocode:

CuckooSearch(Pa, n, MaxIter)

Input:

Pa = probability of abandoning a nest (0< Pa <1)

n = number of nests (population size)

MaxIter = maximum number of iterations.

Output:

BestNest = best solution found

Initialize population of n nests xi (i = 1 to n)

Evaluate fitness f(xi) for each nest

t -> 0

while(t < MaxIter) do

for each nest i in population do

x new ← LevyFlight (xi)

f (x_new) ← Fitness (x_new)

j ← random (1, n)

if f (x_new) > f (xj) then

xj ← x new

end if

end for

AbandonWorstNests (Pa)

Replace AbandonedNestsWith New Solutions()

BestNest ← FindBestNest (population)

t ← t+1

end while

Return BestNest

Applicat

Output

Best pa

Total va

Total

Generat

Generat

Generat

Generat

Genera

Generat

Generati

Generati

Genera

Genera

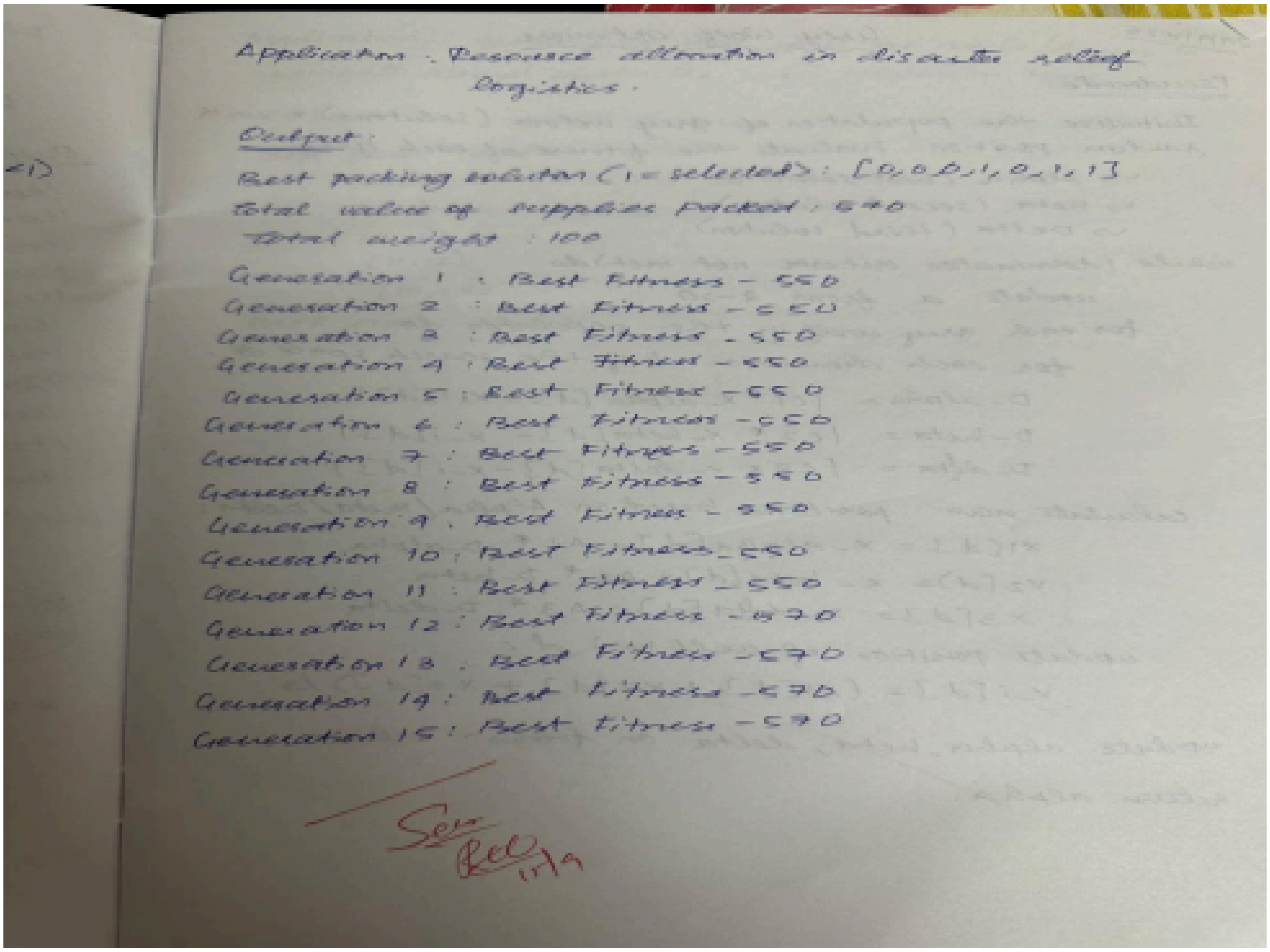
Generat

Genera

Genera

Genera

Genera


Application : Resource allocation in disaster relief logistics.

Output:

Best packing solution (1 = selected): [0,0,0,1,0,1,1]
Total value of supplies packed : 570
Total weight : 100

Generation 1 : Best Fitness - 550
Generation 2 : Best Fitness - 550
Generation 3 : Best Fitness - 550
Generation 4 : Best Fitness - 550
Generation 5 : Best Fitness - 550
Generation 6 : Best Fitness - 550
Generation 7 : Best Fitness - 550
Generation 8 : Best Fitness - 550
Generation 9 : Best Fitness - 550
Generation 10 : Best Fitness - 550
Generation 11 : Best Fitness - 550
Generation 12 : Best Fitness - 570
Generation 13 : Best Fitness - 570
Generation 14 : Best Fitness - 570
Generation 15 : Best Fitness - 570

Code:

```
import random
import math
weights = [10, 20, 30, 40, 15, 25, 35]
values = [60, 100, 120, 240, 80, 150, 200]
capacity = 100 # Max weight capacity of the truck
n_items = len(weights)
n_nests = 15
max_iter = 50
pa = 0.25
def fitness(solution):
 total_weight = sum(w for w, s in zip(weights, solution) if s == 1)
 total_value = sum(v for v, s in zip(values, solution) if s == 1)
 if total_weight > capacity:
 return 0 # Penalize overweight solutions
 else:
 return total_value
def generate_nest():
 return [random.randint(0, 1) for _ in range(n_items)]
def levy_flight(Lambda=1.5):
 sigma_u = (math.gamma(1 + Lambda) * math.sin(math.pi * Lambda / 2) /  (math.gamma((1 +
Lambda) / 2) * Lambda * 2 ** ((Lambda - 1) / 2))) ** (1 / Lambda)
 u = random.gauss(0, sigma_u)
 v = random.gauss(0, 1)
```

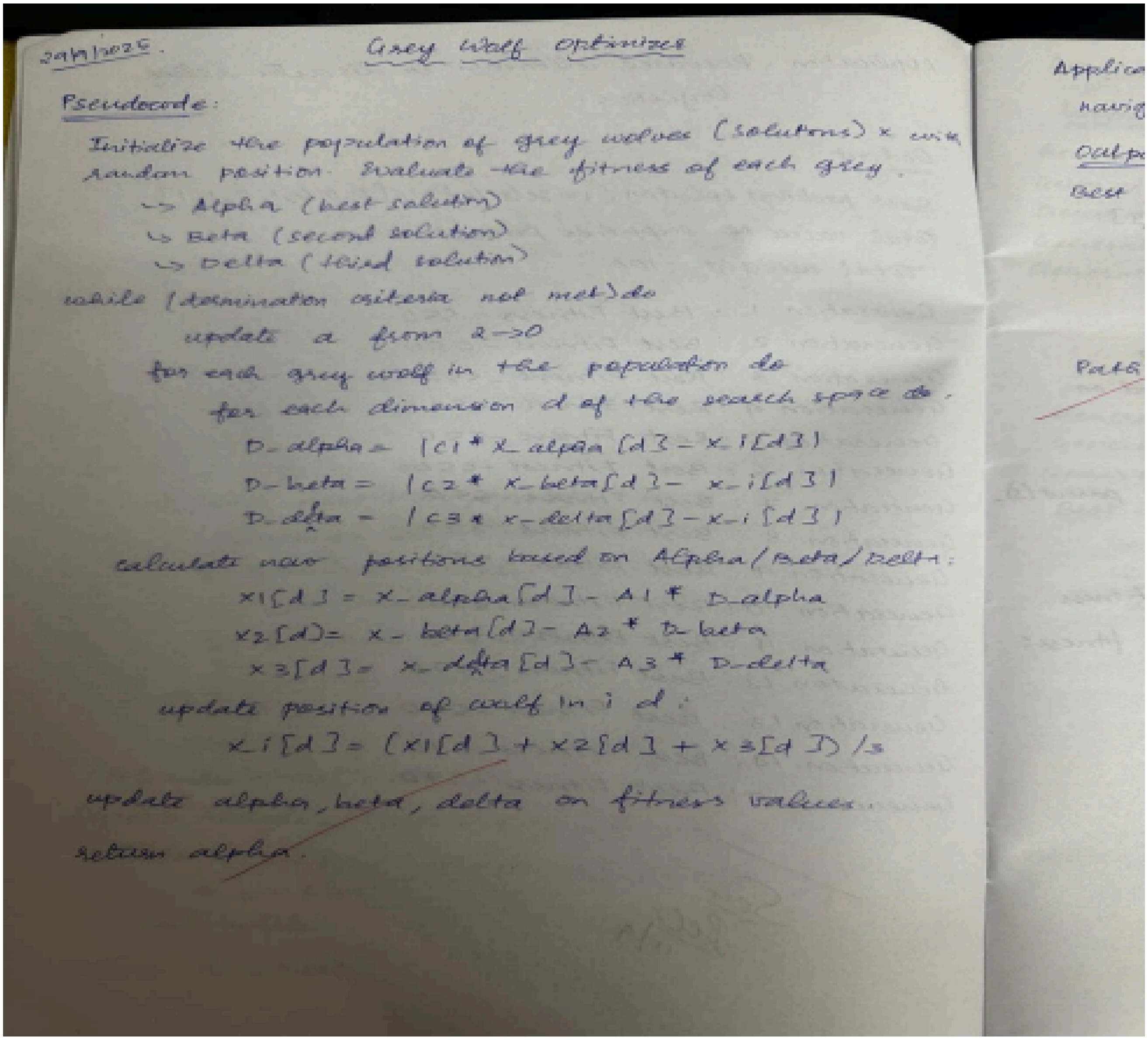
```
step = u / (abs(v) ** (1 / Lambda))
return step
def get_cuckoo(nest, best_nest):
new_nest = []
for xi, bi in zip(nest, best_nest):
step = levy_flight()
val = xi + step * (xi - bi)
s = 1 / (1 + math.exp(-val))
new_val = 1 if s > 0.5 else 0
new_nest.append(new_val)
return new_nest
def cuckoo_search():
nests = [generate_nest() for _ in range(n_nests)]
fitness_values = [fitness(nest) for nest in nests]
best_index = fitness_values.index(max(fitness_values))
best_nest = nests[best_index][:]
best_fitness = fitness_values[best_index]
for _ in range(max_iter):
for i in range(n_nests):
new_nest = get_cuckoo(nests[i], best_nest)
new_fitness = fitness(new_nest)
if new_fitness > fitness_values[i]:
nests[i] = new_nest
fitness_values[i] = new_fitness
for i in range(n_nests):
if random.random() < pa:
nests[i] = generate_nest()
fitness_values[i] = fitness(nests[i])
current_best_index = fitness_values.index(max(fitness_values))
current_best_fitness = fitness_values[current_best_index]
if current_best_fitness > best_fitness:
best_fitness = current_best_fitness
best_nest = nests[current_best_index][:]
return best_nest, best_fitness
if __name__ == "__main__":
best_solution, best_value = cuckoo_search()
total_weight = sum(w for w, s in zip(weights, best_solution) if s == 1)
print(f"Best packing solution (1 = selected): {best_solution}")
print(f"Total value of supplies packed: {best_value}")
print(f"Total weight: {total_weight}")
```

Program 6 : Grey Wolf Optimization

Problem statement:

The Grey Wolf Optimizer (GWO) algorithm is a swarm intelligence algorithm inspired by the social hierarchy and hunting behavior of grey wolves. It mimics the leadership structure of alpha, beta, delta, and omega wolves and their collaborative hunting strategies. The GWO algorithm uses these social hierarchies to model the optimization process, where the alpha wolves guide the search process while beta and delta wolves assist in refining the search direction. This algorithm is effective for continuous optimization problems and has applications in engineering, data analysis, and machine learning.

Algorithm:



24/4/2025.

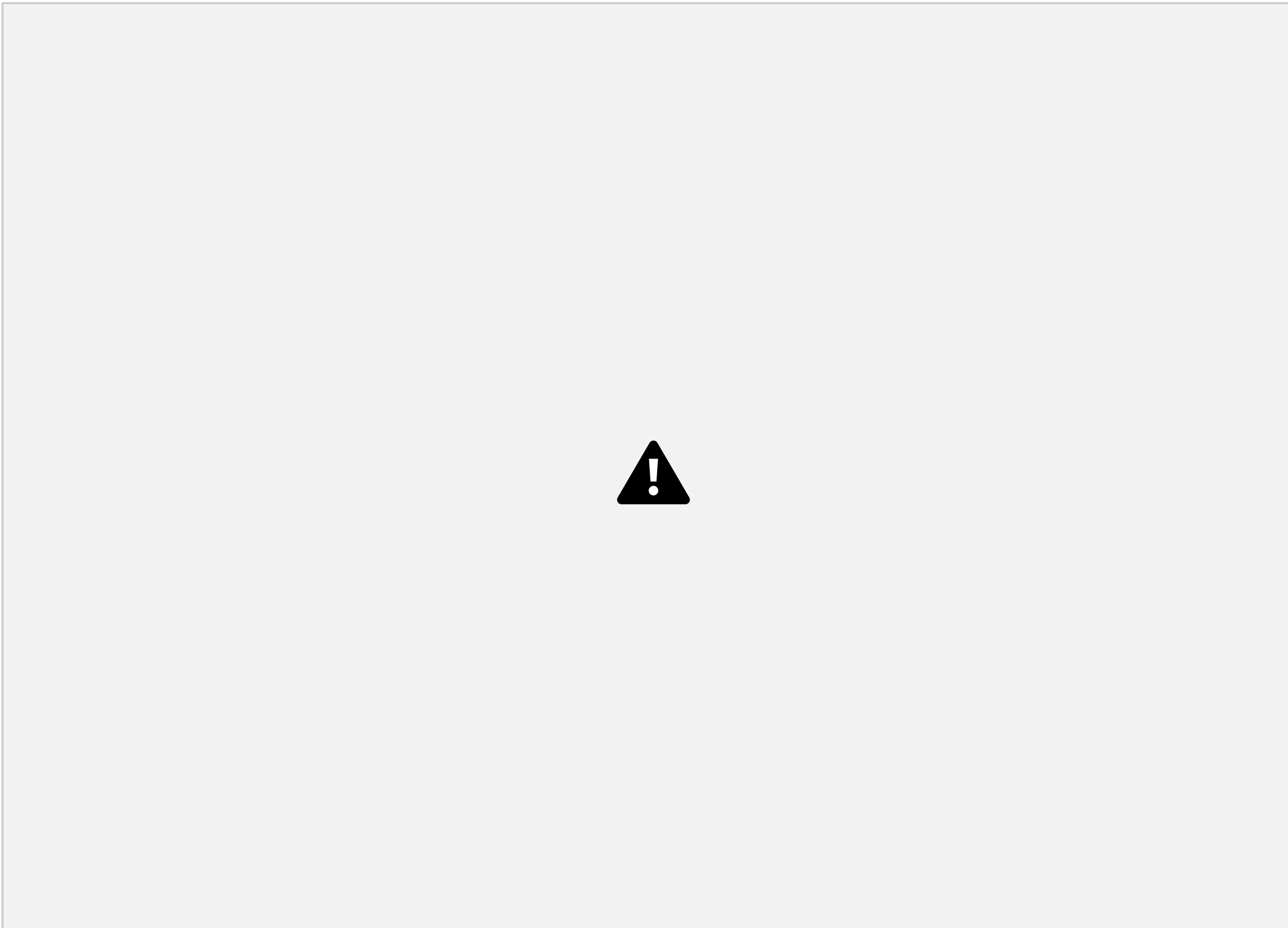
Grey Wolf Optimizer

Pseudocode:

Initialize the population of grey wolves (solutions) x with random position. Evaluate the fitness of each grey.

-> Alpha (best solution)
-> Beta (second solution)
-> Delta (third solution)

```
while (termination criteria not met) do
    update a from 2->0
    for each grey wolf in the population do
        for each dimension d of the search space do.
            D_alpha = |c1 * x_alpha[d] - x_i[d]|
            D_beta = |c2 * x_beta[d] - x_i[d]|
            D_delta = |c3 * x_delta[d] - x_i[d]|
    calculate new positions based on Alpha/Beta/Delta:
            x1[d] = x_alpha[d] - A1 * D_alpha
            x2[d] = x_beta[d] - A2 * D_beta
            x3[d] = x_delta[d] - A3 * D_delta
        update position of wolf in i d:
            x_i[d] = (x1[d] + x2[d] + x3[d]) /3
update alpha, beta, delta on fitness values.
return alpha.
```



Code:

```
import numpy as np
def gwo(obj_func, dim, search_space, n_agents=20, max_iter=100):
 lb, ub = search_space
 wolves = np.random.uniform(lb, ub, (n_agents, dim))
 alpha, beta, delta = None, None, None
 alpha_score, beta_score, delta_score = float("inf"), float("inf"), float("inf")
for t in range(max_iter):
 for i in range(n_agents):
 fitness = obj_func(wolves[i])
 if fitness < alpha_score:
 delta_score, delta = beta_score, beta
 beta_score, beta = alpha_score, alpha
 alpha_score, alpha = fitness, wolves[i].copy()
 elif fitness < beta_score:
 delta_score, delta = beta_score, beta
 beta_score, beta = fitness, wolves[i].copy()
 elif fitness < delta_score:
 delta_score, delta = fitness, wolves[i].copy()
 a = 2 - 2 * (t / max_iter)
 for i in range(n_agents):
 for j in range(dim):
 r1, r2 = np.random.rand(), np.random.rand()
 A1, C1 = 2 * a * r1 - a, 2 * r2
 D_alpha = abs(C1 * alpha[j] - wolves[i][j])
```

```
X1 = alpha[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[j] - wolves[i][j])
X2 = beta[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[j] - wolves[i][j])
X3 = delta[j] - A3 * D_delta
wolves[i][j] = np.clip((X1 + X2 + X3) / 3, lb, ub)
return alpha, alpha_score
grid_size = (20, 20)
start, goal = np.array([0, 0]), np.array([19, 19])
obstacles = [
(5, 5, 10, 10),
(12, 0, 14, 14),
(3, 15, 15, 17)
]
def is_collision(point):
x, y = point.astype(int)
if x < 0 or y < 0 or x >= grid_size[0] or y >= grid_size[1]:
return True
for ox1, oy1, ox2, oy2 in obstacles:
if ox1 <= x <= ox2 and oy1 <= y <= oy2:
return True
return False
waypoints = waypoints.reshape(-1, 2)
path = [start] + [w.astype(int) for w in waypoints] + [goal]
total_dist, penalty = 0, 0
for i in range(len(path) - 1):
dist = np.linalg.norm(path[i + 1] - path[i])
total_dist += dist
if is_collision(path[i + 1]):
penalty += 100
energy = 0
for i in range(1, len(path) - 1):
v1 = path[i] - path[i - 1]
v2 = path[i + 1] - path[i]
if np.linalg.norm(v1) > 0 and np.linalg.norm(v2) > 0:
cos_angle = np.dot(v1, v2) / (np.linalg.norm(v1) * np.linalg.norm(v2))  angle
= np.arccos(np.clip(cos_angle, -1, 1))
energy += angle
return total_dist + energy * 5 + penalty
n_waypoints = 5 # intermediate waypoints
dim = n_waypoints * 2
best_path, best_score = gwo(path_cost, dim, (0, grid_size[0]-1), n_agents=30, max_iter=200)
best_waypoints = best_path.reshape(-1, 2).astype(int)
final_path = np.vstack([start, best_waypoints, goal])
clean_path = []
for p in final_path:
```

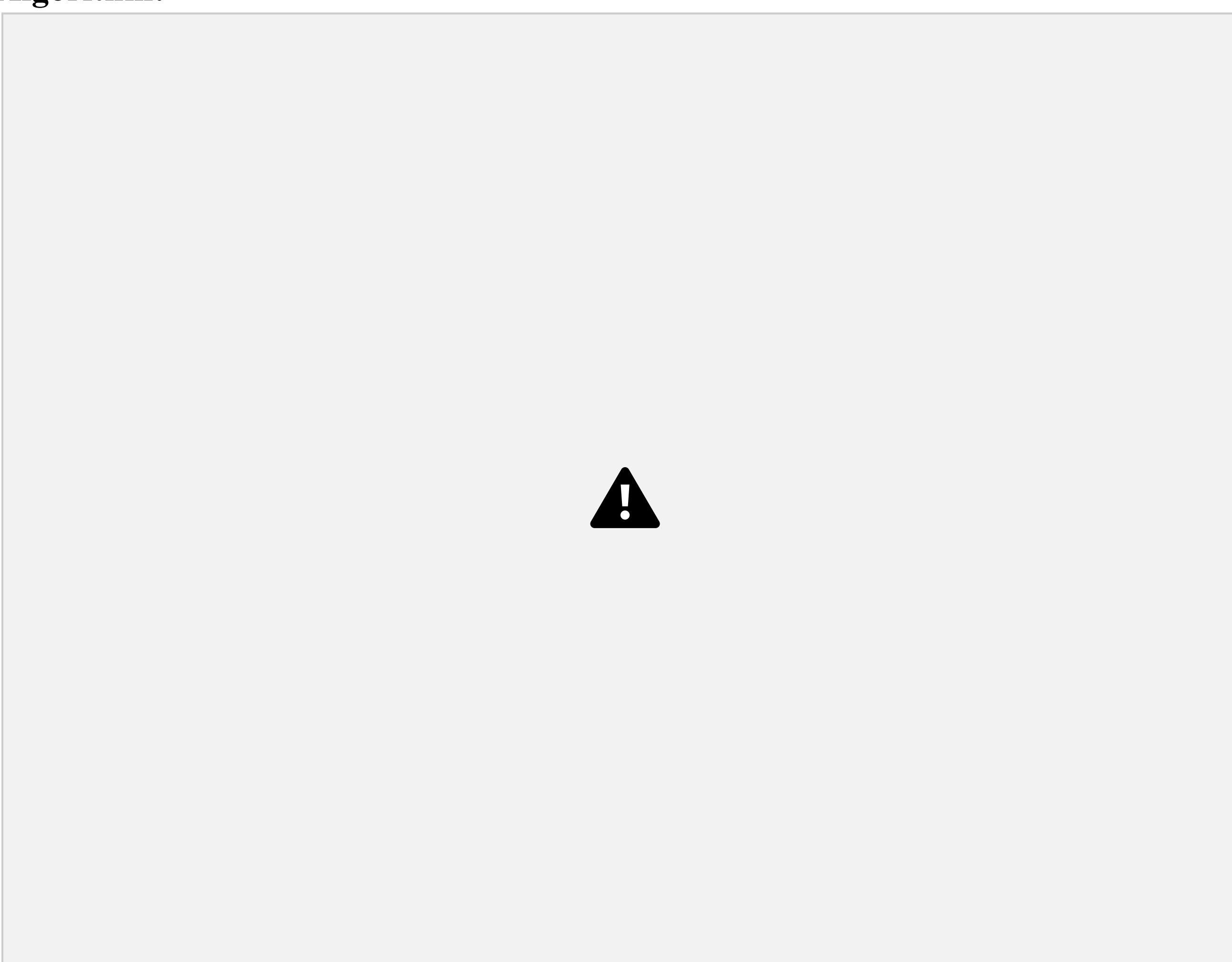
```
 pt = tuple(map(int, p))
 if len(clean_path) == 0 or pt != clean_path[-1]:
 clean_path.append(pt)
print("Best Path Found:")
for p in clean_path:
 print(p)
print("\nPath Cost:", round(best_score, 2))
```

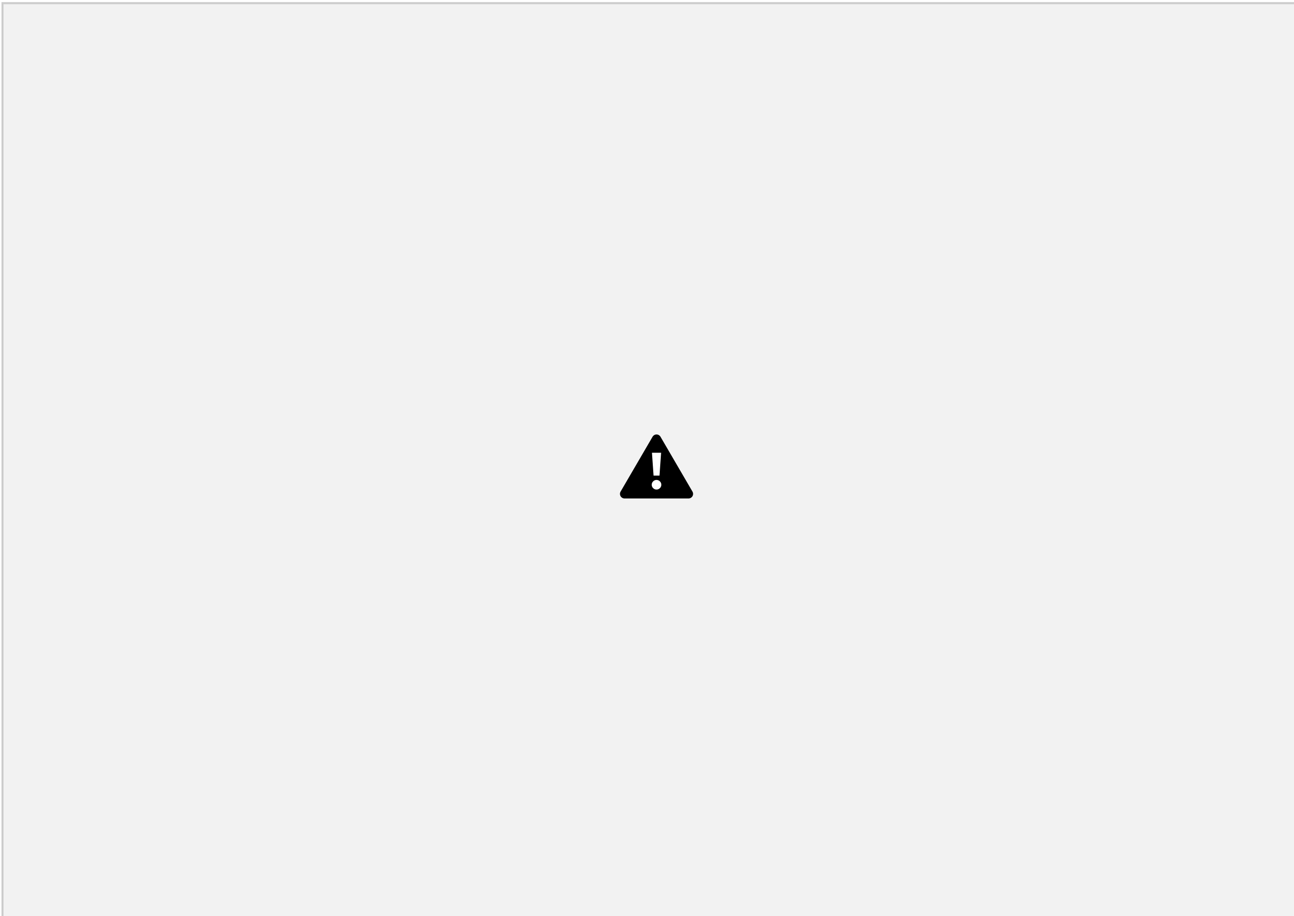
<u>Program 7 : Parallel cellular Optimization</u>

Problem statement:

Parallel Cellular Algorithms are inspired by the functioning of biological cells that operate in a highly parallel and distributed manner. These algorithms leverage the principles of cellular automata and parallel computing to solve complex optimization problems efficiently. Each cell represents a potential solution and interacts with its neighbors to update its state based on predefined rules. This interaction models the diffusion of information across the cellular grid, enabling the algorithm to explore the search space effectively. Parallel Cellular Algorithms are particularly suitable for large-scale optimization problems and can be implemented on parallel computing architectures for enhanced performance.

Algorithm:





Code:

```
import numpy as np
import random
from itertools import permutations
distance_matrix = np.array([
 [0, 2, 9, 10],
 [2, 0, 6, 4],
 [9, 6, 0, 8],
 [10, 4, 8, 0]
])
num_customers = distance_matrix.shape[0] - 1
population_size = 9
grid_dim = (3, 3)
num_vehicles = 2
def generate_individual():
 perm = list(range(1, num_customers + 1))
 random.shuffle(perm)
 return perm
population = [generate_individual() for _ in range(population_size)]
def fitness(individual):
 split_points = np.linspace(0, num_customers, num_vehicles + 1, dtype=int)
total_distance = 0
 for i in range(num_vehicles):
 route = [0] + individual[split_points[i]:split_points[i+1]] + [0] # depot at start and end  for
j in range(len(route) - 1):
```

```
 total_distance += distance_matrix[route[j], route[j+1]]
 return total_distance
def get_neighbors(idx):
 r, c = divmod(idx, grid_dim[1])
 neighbors = []
 for dr in [-1, 0, 1]:
 for dc in [-1, 0, 1]:
 nr, nc = r + dr, c + dc
 if 0 <= nr < grid_dim[0] and 0 <= nc < grid_dim[1]:
 n_idx = nr * grid_dim[1] + nc
 if n_idx != idx:
 neighbors.append(n_idx)
 return neighbors
def crossover(parent1, parent2):
 size = len(parent1)
 a, b = sorted(random.sample(range(size), 2))
 child = [None] * size
 child[a:b] = parent1[a:b]
 pointer = b
 for gene in parent2[b:] + parent2[:b]:
 if gene not in child:
 if pointer == size:
 pointer = 0
 child[pointer] = gene
 pointer += 1
 return child
def mutate(individual):
 a, b = random.sample(range(len(individual)), 2)
 individual[a], individual[b] = individual[b], individual[a]
 return individual
def pca_iteration(pop):
 new_pop = pop.copy()
 for idx in range(len(pop)):
 neighbors = get_neighbors(idx)
 partner_idx = random.choice(neighbors)
 parent1 = pop[idx]
 parent2 = pop[partner_idx]
 child = crossover(parent1, parent2)
 if random.random() < 0.2:
 child = mutate(child)
 if fitness(child) < fitness(pop[idx]):
 new_pop[idx] = child
 return new_pop
num_generations = 25
for gen in range(num_generations):
 population = pca_iteration(population)
 best_fitness = min(fitness(ind) for ind in population)
 print(f"Generation {gen+1}: Best total distance = {best_fitness}")
best_individual = min(population, key=fitness)
print("\nBest route assignment (split evenly):")
```

```
split_points = np.linspace(0, num_customers, num_vehicles + 1, dtype=int)
for i in range(num_vehicles):
 route = [0] + best_individual[split_points[i]:split_points[i+1]] + [0]
print(f"Vehicle {i+1} route: {route}")
print(f"Total distance: {fitness(best_individual)}")
```