

# **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.

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

# Index

<b>Sl. No.</b>	<b>Date</b>	<b>Experiment Title</b>	<b>Page No.</b>
1	18-8-'25	Program 1 - Genetic Algorithm	4
2	25-8-'25	Program 2 - Gene Expression Algorithm	8
3	1-9-'25	Program 3 - Particle Swarm Algorithm	11
4	8-9-'25	Program 4 - Ant Colony Algorithm	13
5	15-9-'25	Program 5 - Cuckoo Search Algorithm	16
6	29-9-'25	Program 6 - Grey Wolf Optimisation Algorithm	18
7	13-10-'25	Program 7 - Parallel Cellular Algorithm	21

**Github Link:**

[https://github.com/parthjain21108/BIS\\_LAB](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:

	$EvoLo = f(x) - 144 = 0.1247$	$Ef(x) = 1155$	$Expected = f(x_i) = 144$	
			$0.001 \text{ Avg}(f(x)) = 288.95$	
string No	Initial Population	x	fitness	Prob of selection
1	01100	12	144	0.1247 12.47 0.49 1
2	11001	25	625	0.15411 54.11 2.16 2
3	00101	5	25	0.0219 2.16 0.08 0
4	11011	13	181	0.3126 31.16 1.75 1
sum			1155	1.0 10.0 9
Avg			288.95	0.25 25 1
Maxim			625	0.5411 54.11 2.16
selecting mating pool				
string	Mating Pool	crossover	Offspring	X value
1	01100	4	01101	73
2	11001		11000	24
3	11011	2	11011	27
				729

Genetic pt 1				
string No	Offspring	Mutation count	Offspring X	Offspring after mutation value
1	01101	10000	1110	23 1
2	11000	00000	11000	24 2
3	11011	00000	11011	25 2
4	10001	00101	10100	20 4

Reward code				
Main individual:				
def init_(self, genome):				
self.genome = genome				
self.fitness = calculate_fitness(self)				
def genome_to_int(genome):				
value = 0				
for bit in genome:				
value = (value <= 1) * 2 +				
return value				
def initialize_population():				
return [individual()]				

Initialise population( ):

population = [ ]

- for each individual in range (pop size)
  - genome = random permutation of city index
  - calculate fitness
  - add individual to population

Return population

calculate fitness:

total distance = 0

for i from 0 to len(individual.genome) - 1:

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(parent 1, parent 2):

point 1, point 2 = Random nos b/w 0 and len(pop).

child's genome = copy part of parent 1's genome

Date \_\_\_\_\_  
Page \_\_\_\_\_

parent & child pairs

function muate

if random number < mutation rate:  
 $i,j$  = random indices b/w 0 and length  
 swap individual genome( $i$ ) and  $(j)$

Recalculate fitness

In genetic algos:

population = initialize\_population()

for generation in range(generations):  
 set population by fitness function  
 pick best seen fit element from  
 new\_population = population[0]

while length of new.pop < pop\_size:  
 parent 1 = selection(population)  
 parent 2 = selection(population)  
 child 1, child 2 = crossover(parent 1, parent 2)  
 mutate(child 1)  
 mutate(child 2)  
 Add child 1 and child 2 to new.pop

return best fit in the population

## Code:

```
import random
import math

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

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

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

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

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

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

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

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

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

    return new_population
```

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

return best

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

## Program 2

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

### Algorithm:

20/07/2023

LAB-2 GEA

Application: Magic signal optimization

```

function generate_random_schedule()
    schedule = []
    for each intersection in num_intersections:
        green = random.randint(0, 15)
        yellow = random.randint(0, 15)
        red = random.randint(0, 15)
        schedule.append([green, yellow, red])
    return schedule

```

fitness:

```

t_congestion = 0
t_wt = 0
t_flow = 0
for each int in schedule:
    congest, wt, flow = simulate(intersection)
    t_congestion += t_congest
    t_wt += wt
    t_flow += flow
fitness = t_congestion + t_wt + (1/t_flow)
return fitness

```

crossover (parent1, parent2):

```

if random num > mutation:
    return parent1, parent2
point = random integer b/w 1 and Num_Intersections
child1 = parent1[0:point] + parent2[point:]
child2 = parent2[0:point] + parent1[point:]
return child1, child2

```

on mutate (schedule):

```

for each intersection in schedule:
    if random no < mutation rate:
        randomly mutate green, yellow or red.
random mutate schedule

```

on generate (algo):

```

population = generate() for i in range(pop_size)
for generation in 1 to generations:
    fitness_surr = evaluate(schedule) for each
    in population
    population = select(pop on fitness score)
    new pop = population (0 to 10% of pop)
    while len(new pop) < pop_size:
        parent1, parent2 = select - two parents
        based on fitness
        child1, child2 = crossover (parent1, parent2)
        new_population.append(mutate (child1))
    population = new_population
    best_schedule = population[0]
return best_schedule

```

Output:

```

Generation 1: a=40 y=42 R=425
Intersection 1: a=40 y=42 R=425
Intersection 2: a=50 y=55 R=475
return congestion 3.5% ↓
NT 15% ↓
FLOW 22 ↑

```

Generation	Intersection 1	Intersection 2	Congestion	NT	Flow
0	a=40, y=42, R=425	a=50, y=55, R=475	3.5%	15%	22
1	a=40, y=42, R=425	a=50, y=55, R=475	3.0%	14%	22
2	a=40, y=42, R=425	a=50, y=55, R=475	2.5%	13%	22
3	a=40, y=42, R=425	a=50, y=55, R=475	2.0%	12%	22
4	a=40, y=42, R=425	a=50, y=55, R=475	1.5%	11%	22
5	a=40, y=42, R=425	a=50, y=55, R=475	1.0%	10%	22
6	a=40, y=42, R=425	a=50, y=55, R=475	0.5%	9%	22
7	a=40, y=42, R=425	a=50, y=55, R=475	0.2%	8%	22
8	a=40, y=42, R=425	a=50, y=55, R=475	0.1%	7%	22
9	a=40, y=42, R=425	a=50, y=55, R=475	0.05%	6%	22
10	a=40, y=42, R=425	a=50, y=55, R=475	0.02%	5%	22
11	a=40, y=42, R=425	a=50, y=55, R=475	0.01%	4%	22
12	a=40, y=42, R=425	a=50, y=55, R=475	0.005%	3%	22
13	a=40, y=42, R=425	a=50, y=55, R=475	0.002%	2%	22
14	a=40, y=42, R=425	a=50, y=55, R=475	0.001%	1%	22
15	a=40, y=42, R=425	a=50, y=55, R=475	0.0005%	0.5%	22
16	a=40, y=42, R=425	a=50, y=55, R=475	0.0002%	0.2%	22
17	a=40, y=42, R=425	a=50, y=55, R=475	0.0001%	0.1%	22
18	a=40, y=42, R=425	a=50, y=55, R=475	0.00005%	0.05%	22
19	a=40, y=42, R=425	a=50, y=55, R=475	0.00002%	0.02%	22
20	a=40, y=42, R=425	a=50, y=55, R=475	0.00001%	0.01%	22
21	a=40, y=42, R=425	a=50, y=55, R=475	0.000005%	0.005%	22
22	a=40, y=42, R=425	a=50, y=55, R=475	0.000002%	0.002%	22
23	a=40, y=42, R=425	a=50, y=55, R=475	0.000001%	0.001%	22
24	a=40, y=42, R=425	a=50, y=55, R=475	0.0000005%	0.0005%	22
25	a=40, y=42, R=425	a=50, y=55, R=475	0.0000002%	0.0002%	22
26	a=40, y=42, R=425	a=50, y=55, R=475	0.0000001%	0.0001%	22
27	a=40, y=42, R=425	a=50, y=55, R=475	0.00000005%	0.00005%	22
28	a=40, y=42, R=425	a=50, y=55, R=475	0.00000002%	0.00002%	22
29	a=40, y=42, R=425	a=50, y=55, R=475	0.00000001%	0.00001%	22
30	a=40, y=42, R=425	a=50, y=55, R=475	0.000000005%	0.000005%	22
31	a=40, y=42, R=425	a=50, y=55, R=475	0.000000002%	0.000002%	22
32	a=40, y=42, R=425	a=50, y=55, R=475	0.000000001%	0.000001%	22
33	a=40, y=42, R=425	a=50, y=55, R=475	0.0000000005%	0.0000005%	22
34	a=40, y=42, R=425	a=50, y=55, R=475	0.0000000002%	0.0000002%	22
35	a=40, y=42, R=425	a=50, y=55, R=475	0.0000000001%	0.0000001%	22
36	a=40, y=42, R=425	a=50, y=55, R=475	0.00000000005%	0.00000005%	22
37	a=40, y=42, R=425	a=50, y=55, R=475	0.00000000002%	0.00000002%	22
38	a=40, y=42, R=425	a=50, y=55, R=475	0.00000000001%	0.00000001%	22
39	a=40, y=42, R=425	a=50, y=55, R=475	0.000000000005%	0.000000005%	22
40	a=40, y=42, R=425	a=50, y=55, R=475	0.000000000002%	0.000000002%	22
41	a=40, y=42, R=425	a=50, y=55, R=475	0.000000000001%	0.000000001%	22
42	a=40, y=42, R=425	a=50, y=55, R=475	0.0000000000005%	0.0000000005%	22
43	a=40, y=42, R=425	a=50, y=55, R=475	0.0000000000002%	0.0000000002%	22
44	a=40, y=42, R=425	a=50, y=55, R=475	0.0000000000001%	0.0000000001%	22
45	a=40, y=42, R=425	a=50, y=55, R=475	0.00000000000005%	0.00000000005%	22
46	a=40, y=42, R=425	a=50, y=55, R=475	0.00000000000002%	0.00000000002%	22
47	a=40, y=42, R=425	a=50, y=55, R=475	0.00000000000001%	0.00000000001%	22
48	a=40, y=42, R=425	a=50, y=55, R=475	0.000000000000005%	0.000000000005%	22
49	a=40, y=42, R=425	a=50, y=55, R=475	0.000000000000002%	0.000000000002%	22
50	a=40, y=42, R=425	a=50, y=55, R=475	0.000000000000001%	0.000000000001%	22
51	a=40, y=42, R=425	a=50, y=55, R=475	0.0000000000000005%	0.0000000000005%	22
52	a=40, y=42, R=425	a=50, y=55, R=475	0.0000000000000002%	0.0000000000002%	22
53	a=40, y=42, R=425	a=50, y=55, R=475	0.0000000000000001%	0.0000000000001%	22
54	a=40, y=42, R=425	a=50, y=55, R=475	0.00000000000000005%	0.00000000000005%	22
55	a=40, y=42, R=425	a=50, y=55, R=475	0.00000000000000002%	0.00000000000002%	22
56	a=40, y=42, R=425	a=50, y=55, R=475	0.00000000000000001%	0.00000000000001%	22
57	a=40, y=42, R=425	a=50, y=55, R=475	0.000000000000000005%	0.000000000000005%	22
58	a=40, y=42, R=425	a=50, y=55, R=475	0.000000000000000002%	0.000000000000002%	22
59	a=40, y=42, R=425	a=50, y=55, R=475	0.000000000000000001%	0.000000000000001%	22
60	a=40, y=42, R=425	a=50, y=55, R=475	0.0000000000000000005%	0.0000000000000005%	22
61	a=40, y=42, R=425	a=50, y=55, R=475	0.0000000000000000002%	0.0000000000000002%	22
62	a=40, y=42, R=425	a=50, y=55, R=475	0.0000000000000000001%	0.0000000000000001%	22
63	a=40, y=42, R=425	a=50, y=55, R=475	0.00000000000000000005%	0.00000000000000005%	22
64	a=40, y=42, R=425	a=50, y=55, R=475	0.00000000000000000002%	0.00000000000000002%	22
65	a=40, y=42, R=425	a=50, y=55, R=475	0.00000000000000000001%	0.00000000000000001%	22
66	a=40, y=42, R=425	a=50, y=55, R=475	0.000000000000000000005%	0.000000000000000005%	22
67	a=40, y=42, R=425	a=50, y=55, R=475	0.000000000000000000002%	0.000000000000000002%	22
68	a=40, y=42, R=425	a=50, y=55, R=475	0.000000000000000000001%	0.000000000000000001%	22
69	a=40, y=42, R=425	a=50, y=55, R=475	0.0000000000000000000005%	0.000000000000000005%	22
70	a=40, y=42, R=425	a=50, y=55, R=475	0.0000000000000000000002%	0.000000000000000002%	22
71	a=40, y=42, R=425	a=50, y=55, R=475	0.0000000000000000000001%	0.000000000000000001%	22
72	a=40, y=42, R=425	a=50, y=55, R=475	0.00000000000000000000005%	0.000000000000000005%	22
73	a=40, y=42, R=425	a=50, y=55, R=475	0.00000000000000000000002%	0.000000000000000002%	22
74	a=40, y=42, R=425	a=50, y=55, R=475	0.00000000000000000000001%	0.000000000000000001%	22
75	a=40, y=42, R=425	a=50, y=55, R=475	0.000000000000000000000005%	0.000000000000000005%	22
76	a=40, y=42, R=425	a=50, y=55, R=475	0.000000000000000000000002%	0.000000000000000002%	22
77	a=40, y=42, R=425	a=50, y=55, R=475	0.000000000000000000000001%	0.000000000000000001%	22
78	a=40, y=42, R=425	a=50, y=55, R=475	0.0000000000000000000000005%	0.000000000000000005%	22
79	a=40, y=42, R=425	a=50, y=55, R=475	0.0000000000000000000000002%	0.000000000000000002%	22
80	a=40, y=42, R=425	a=50, y=55, R=475	0.0000000000000000000000001%	0.000000000000000001%	22
81	a=40, y=42, R=425	a=50, y=55, R=475	0.00000000000000000000000005%	0.000000000000000005%	22
82	a=40, y=42, R=425	a=50, y=55, R=475	0.00000000000000000000000002%	0.000000000000000002%	22
83	a=40, y=42, R=425	a=50, y=55, R=475	0.00000000000000000000000001%	0.000000000000000001%	22
84	a=40, y=42, R=425	a=50, y=55, R=475	0.000000000000000000000000005%	0.000000000000000005%	22
85	a=40, y=42, R=425	a=50, y=55, R=475	0.000000000000000000000000002%	0.000000000000000002%	22
86	a=40, y=42, R=425	a=50, y=55, R=475	0.000000000000000000000000001%	0.000000000000000001%	22
87	a=40, y=42, R=425	a=50, y=55, R=475	0.0000000000000000000000000005%	0.000000000000000005%	22
88	a=40, y=42, R=425	a=50, y=55, R=475	0.0000000000000000000000000002%	0.000000000000000002%	22
89	a=40, y=42, R=425	a=50, y=55, R=475	0.0000000000000000000000000001%	0.000000000000000001%	22
90	a=40, y=42, R=425	a=50, y=55, R=475	0.00000000000000000000000000005%	0.000000000000000005%	22
91	a=40, y=42, R=425	a=50, y=55, R=475	0.00000000000000000000000000002%	0.000000000000000002%	22
92	a=40, y=42, R=425	a=50, y=55, R=475	0.00000000000000000000000000001%	0.000000000000000001%	22
93	a=40, y=42, R=425	a=50, y=55, R=475	0.000000000000000000000000000005%	0.000000000000000005%	22
94	a=40, y=42, R=425	a=50, y=55, R=475	0.000000000000000000000000000002%	0.000000000000000002%	22
95	a=40, y=42, R=425	a=50, y=55, R=475	0.000000000000000000000000000001%	0.000000000000000001%	22
96	a=40, y=42, R=425	a=50, y=55, R=475	0.0000000000000000000000000000005%	0.000000000000000005%	22
97	a=40, y=42, R=425	a=50, y=55, R=475	0.0000000000000000000000000000002%	0.000000000000000002%	22
98	a=40, y=42, R=425	a=50, y=55, R=475	0.0000000000000000000000000000001%	0.000000000000000001%	22
99	a=40, y=42, R=425	a=50, y=55, R=475	0.00000000000000000000000000000005%	0.000000000000000005%	22
100	a=40, y=42, R=425	a=50, y=55, R=475	0.00000000000000000000000000000002%	0.000000000000000002%	22
101	a=40, y=42, R=425	a=50, y=55, R=475	0.00000000000000000000000000000001%	0.000000000000000001%	22
102	a=40, y=42, R=425	a=50, y=55, R=475	0.000000000000000000000000000000005%	0.000000000000000005%	22
103	a=40, y=42, R=425	a=50, y=55, R=475	0.000000000000000000000000000000002%	0.000000000000000002%	22
104	a=40, y=42, R=425	a=50, y=55, R=475	0.000000000000000000000000000000001%	0.000000000000000001%	22
105	a=40, y=42, R=425	a=50, y=55, R=475	0.0000000000000000000000000000000005%	0.000000000000000005%	22
106	a=40, y=42, R=425	a=50, y=55, R=475	0.0000000000000000000000000000000002%	0.00000	

## 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))

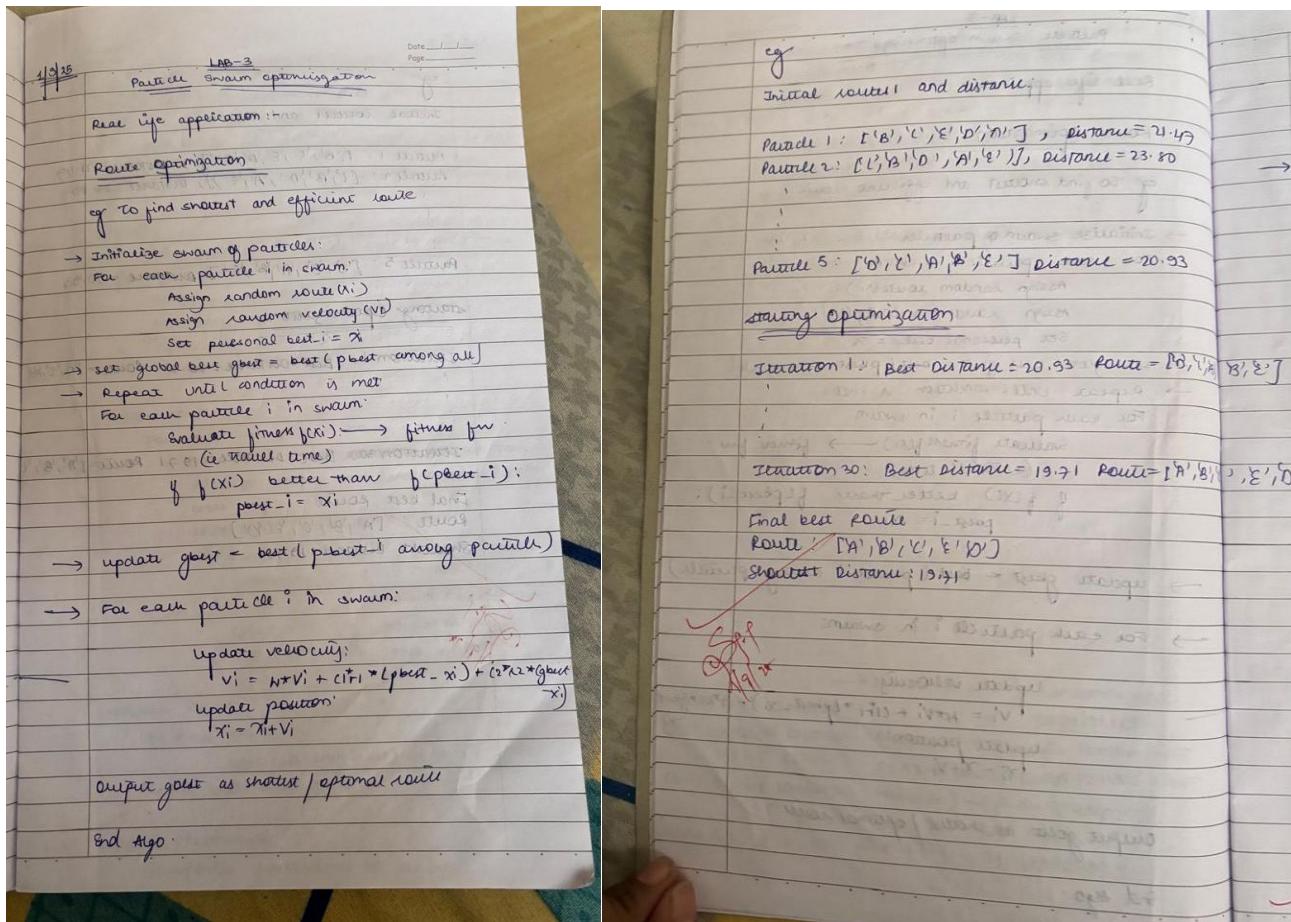
print("\nBest tour path (with coordinates):")
for idx in best:
    print(f'City {idx} -> {cities[idx]}')
print(f'Back to start -> {cities[best[0]]}')

```

# 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]},\nfitness={f(positions[i])}")

```

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

## Program 4

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

### Algorithm:

Ant colony optimization

→ evaporation rate  $\gamma$

combinatorial → it has discrete states

$\alpha \rightarrow$  pheromone → weight of influence

$\rho \rightarrow$  pheromony travelled info heuristic → heuristic info

$\eta \rightarrow$  heuristic info between city  $i$  and  $j$

→ algorithm many paths → it's desirability and visibility

Initialize:  
Set initial pheromone  $\tau$  on all edges to small constant  
Define parameters:  
number of ants ( $m$ )  
pheromone evap. rate ( $\rho$ )  
influence of pheromone ( $\alpha$ )  
influence of heuristic ( $\beta$ )

concrete solution:  
For each ant  $k$  in 1 to  $m$ :  
Place ant  $k$  at randomly selected start  
Initialize empty tour and visited list with start city  
While tour incomplete:  
For each land next city  $j$  not visited:  
$$\text{probability } p_{ij} = (\tau_{ij}^\alpha \cdot \eta_{ij}^\beta) / \text{sum of all candidate}$$
  
Select next city based on  $p_{ij}$   
Add city to tour and visited.  
Evaluate tour:  
For each ant  $k$ :  
Calculate length(cost) of ant  $k$ 's tour  
Record best tour so far.

Update pheromones  
Evaporate pheromone on all edges

$\tau_{ij} = (1 - \rho) * \tau_{ij}$  for all edges

For each ant  $k$ :  
Deposit pheromone on edge in ant  $k$ 's tour  
$$\tau_{ij} = \tau_{ij} + Q / m$$
  
Deposit pheromone only from best ant on tour

Output best tour with length

Output total pheromone

Iterations: 100

Input:  
city coordinates = [0, 0]  
[1, 5]  
[2, 3]  
[3, 2]  
[4, 1]  
[5, 6]  
[6, 5]  
[7, 4]  
[8, 3]

Output:  
Best tour found: [0, 2, 4, 3, 1]  
Tour length: 22.351

*Start* *End*

## 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]
    ])

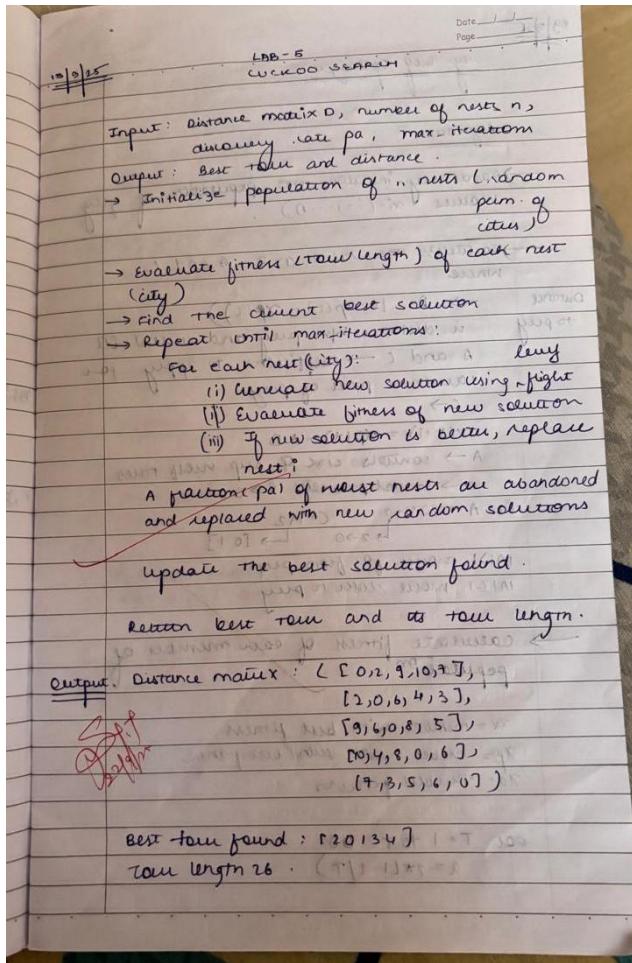
    aco = ACO_TSP(distance_matrix, n_ants=10, n_iterations=100)
    best_tour, best_length = aco.run()
    print("Best Tour:", best_tour)
    print("Best Length:", best_length)

```

# 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],

```

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

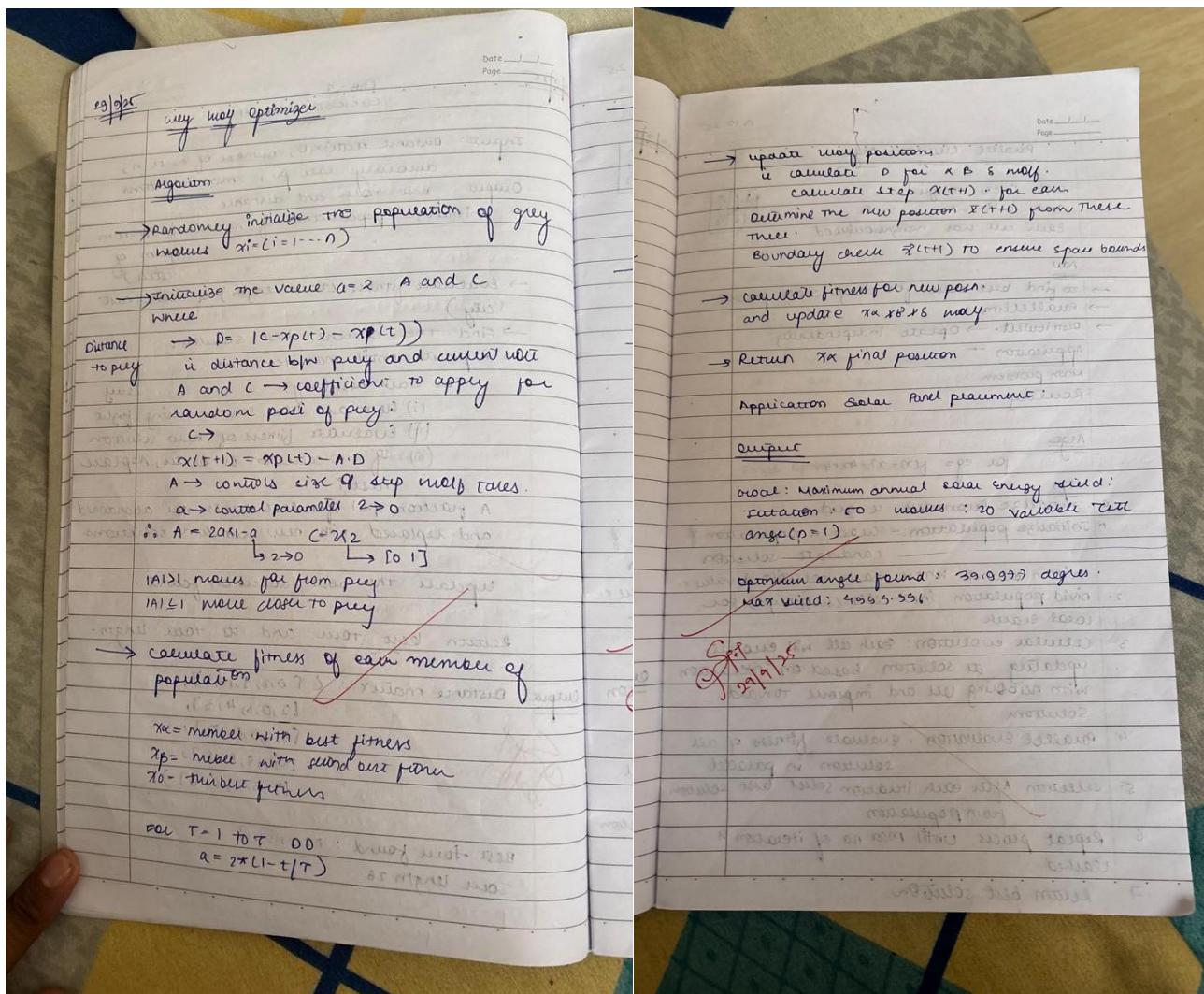
```

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)
```

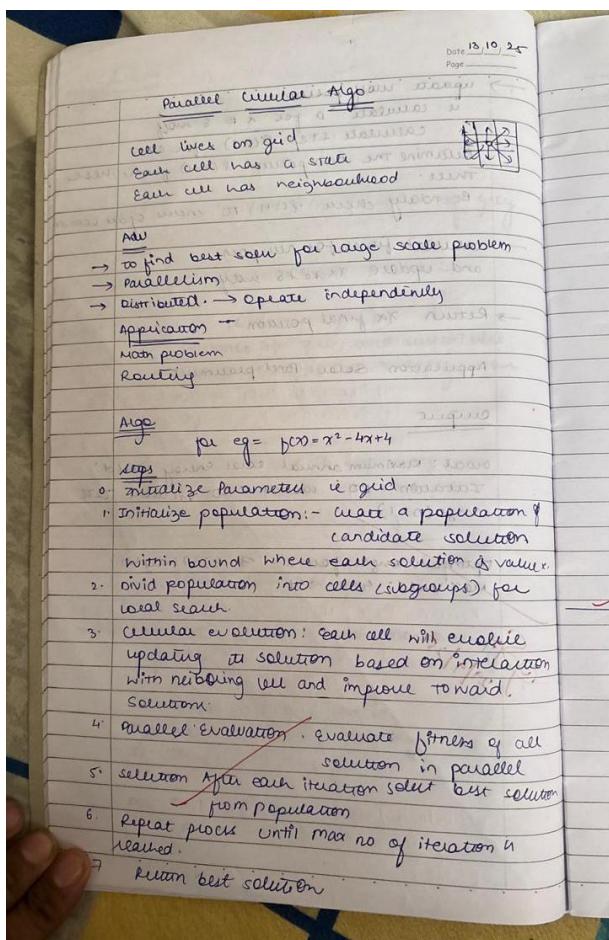
```
print("Best Schedule (job → machine):", best_schedule)
```

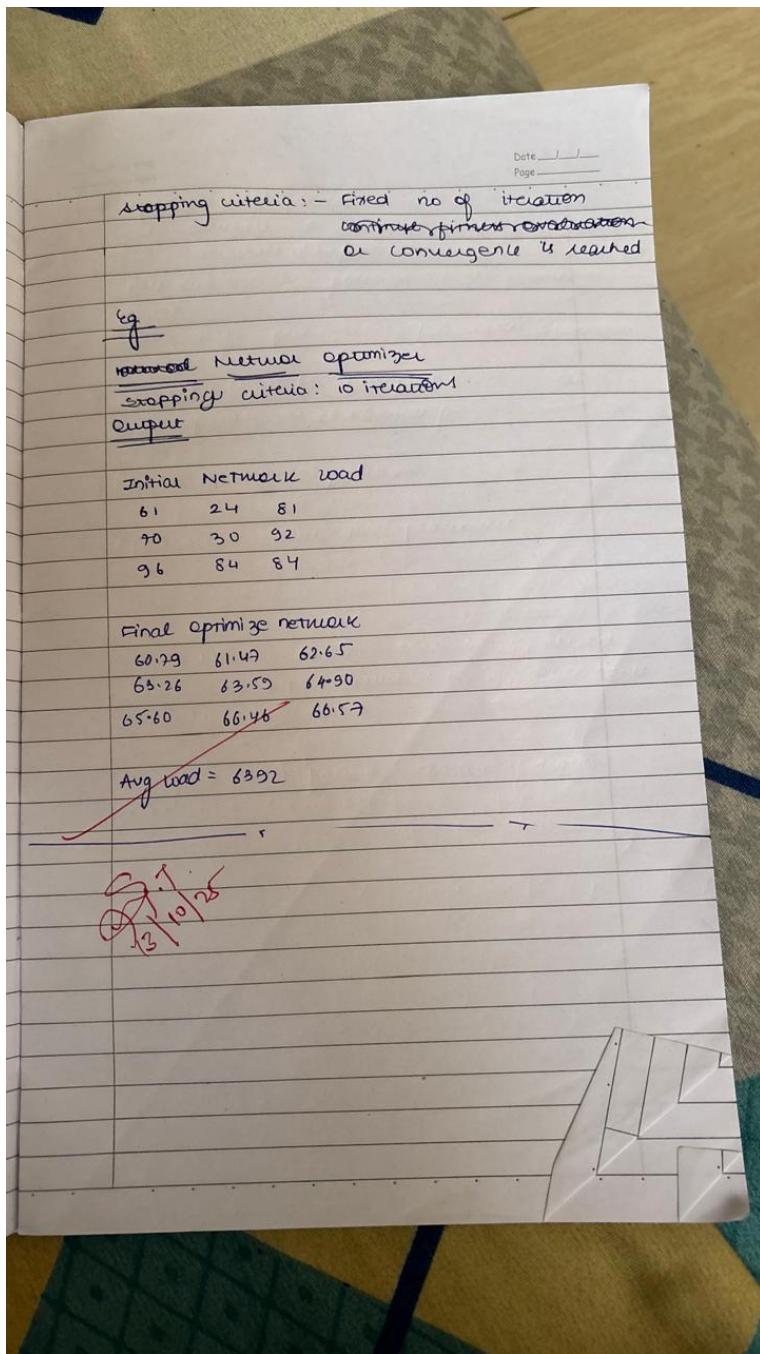
```
print("Best Makespan:", best_makespan)
```

## 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)

```