

VISVESVARAYA TECHNOLOGICAL UNIVERSITY

“JnanaSangama”, Belgaum -590014, Karnataka.



LAB RECORD

Bio Inspired Systems (23CS5BSBIS)

Submitted by

Shreya Raj (1BM23CS317)

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 Jan-2026**

**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 **Shreya Raj (1BM23CS317)**, who is 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.

Rohith Vaidya K Assistant Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE
--	--

Index

Sl. No.	Date	Experiment Title	Page No.
1	29/08/2025	Genetic Algorithm for Optimization Problems	4-9
2	29/08/2025	Optimization via Gene Expression Algorithms	10-14
3	12/09/2025	Particle Swarm Optimization for Function Optimization	15-17
4	10/10/2025	Ant Colony Optimization for the Traveling Salesman Problem	18-20
5	17/10/2025	Cuckoo Search (CS)	21-23
6	17/10/2025	Grey Wolf Optimizer (GWO)	24-27
7	07/11/2025	Parallel Cellular Algorithms and Programs	28-30

Github Link:

https://github.com/RajShreyaa/1BM23CS317_BIS_Lab

Program 1: Genetic Algorithm for Optimization Problems

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. Implement a Genetic Algorithm using Python to solve a basic optimization problem, such as finding the maximum value of a mathematical function.

Algorithm:

LAB-II							
29/8/2025							
5 main phases: Initialization, Fitness Assignment, Selection, Crossover, Termination							
Steps:							
1) Selecting encoding techniques 2 to 31							
2) Select initial population							
String No.	Initial popul'n	X value	Fitness $f(x) = x^2$	Prob $f(x)/\sum f(x)$	% Prob	Expected count $f(x)/\text{avg} f(x)$	Actual count
1	01100	12	144	0.1247	12.47	0.49	1
2	11001	25	625	0.5411	54.11	2.164	2
3	00101	5	25	0.0216	2.16	0.086	0
4	10011	19	361	0.3125	31.25	1.25	1
$\sum f(x) = 888.75$							
3) Select mating pool							
String No.	Mating pool	Crossover point	Offspring after crossover	X value	Fitness $f(x) = x^2$		
1	01100	4	01101	13	169		
2	11001		11000	24	576		
3	11001	2	11011	27	729		
4	10011	2	10001	17	289		
4) Crossover: Random 4 & 2 Max value 729							
5) Mutation							
String No.	Offspring after crossover	Mutat' chromosome for offspring	Offspring after mutat'	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		
$\sum f(x) = 2346$							

```

import random
def fitness(x):
    return x**2
POPULATION_SIZE = 4
CHROMOSOME_LENGTH = 5
MUTATION_RATE = 0.1
GENERATIONS = 10
def binary_to_decimal(binary):
    return int(binary, 2)
def decimal_to_binary(n):
    return format(n, f'{CHROMOSOME_LENGTH}b')
def initialize_population():
    return [decimal_to_binary(random.randint(0, 2**CHROMOSOME_LENGTH-1))
           for _ in range(POPULATION_SIZE)]
def evaluate_population(population):
    return [fitness(binary_to_decimal(individual)) for individual in population]
def select_parents(population, fitnesses):
    parents = []
    for _ in range(2):
        i, j = random.sample(range(len(population)), 2)
        if fitnesses[i] > fitnesses[j]:
            parents.append(population[i])
        else:
            parents.append(population[j])
    return parents
def crossover(parent1, parent2):
    point = random.randint(1, CHROMOSOME_LENGTH - 1)
    child1 = parent1[:point] + parent2[point:]
    child2 = parent2[:point] + parent1[point:]
    return child1, child2

```

```

def mutate(individual):
    mutated = individual
    for bit in individual:
        if random.random() < MUTATION_RATE:
            mutated += '1' if bit == '0' else '0'
        else:
            mutated += bit
    return mutated

def genetic_algorithm():
    population = initialize_population()
    for generation in range(GENERATIONS):
        fitnesses = evaluate_population(population)
        new_population = []
        while len(new_population) < POPULATION_SIZE:
            parent1, parent2 = select_parents(population, fitnesses)
            child1, child2 = crossover(parent1, parent2)
            child1 = mutate(child1)
            child2 = mutate(child2)
            new_population.append(child1)
            new_population.append(child2)
        population = new_population[:POPULATION_SIZE]
        best = max(population, key=lambda x: fitness(binary_to_decimal(x)))
        print(f'Generation {generation+1}: Best = {binary_to_decimal(best)}, '
              f'Fitness = {fitness(binary_to_decimal(best))}')
    best = max(population, key=lambda x: fitness(binary_to_decimal(x)))
    print('Best solution:', binary_to_decimal(best))
    print('Fitness:', fitness(binary_to_decimal(best)))
genetic_algorithm()

```

Code:

```
import random

# 1. Define problem (fitness function)
def fitness(x):
    return x ** 2

# 2. Initialize parameters
POPULATION_SIZE = 4
CHROMOSOME_LENGTH = 5 # We'll use 5-bit binary strings, i.e. values 0–31
MUTATION_RATE = 0.1
GENERATIONS = 10

# Helper: Binary to decimal
def binary_to_decimal(binary):
    return int(binary, 2)

# Helper: Decimal to binary
def decimal_to_binary(n):
    return format(n, f'0{CHROMOSOME_LENGTH}b')

# 3. Create initial population
def initialize_population():
    return [decimal_to_binary(random.randint(0, 2**CHROMOSOME_LENGTH - 1)) for _ in range(POPULATION_SIZE)]

# 4. Evaluate fitness of the population
def evaluate_population(population):
    return [fitness(binary_to_decimal(individual)) for individual in population]

# 5. Selection (Tournament Selection)
def select_parents(population, fitnesses):
    parents = []
    for _ in range(2):
        i, j = random.sample(range(len(population)), 2)
        if fitnesses[i] > fitnesses[j]:
            parents.append(population[i])
        else:
            parents.append(population[j])
    return parents
```

```

# 6. Crossover (Single-point)
def crossover(parent1, parent2):
    point = random.randint(1, CHROMOSOME_LENGTH - 1)
    child1 = parent1[:point] + parent2[point:]
    child2 = parent2[:point] + parent1[point:]
    return child1, child2

# 7. Mutation (Bit-flip)
def mutate(individual):
    mutated = ""
    for bit in individual:
        if random.random() < MUTATION_RATE:
            mutated += '1' if bit == '0' else '0'
        else:
            mutated += bit
    return mutated

# 8. Iteration
def genetic_algorithm():
    population = initialize_population()

    for generation in range(GENERATIONS):
        fitnesses = evaluate_population(population)

        new_population = []

        while len(new_population) < POPULATION_SIZE:
            # Select
            parent1, parent2 = select_parents(population, fitnesses)
            # Crossover
            child1, child2 = crossover(parent1, parent2)
            # Mutation
            child1 = mutate(child1)
            child2 = mutate(child2)
            # Add to new population
            new_population.extend([child1, child2])

        # Replace old population with new (trim if needed)
        population = new_population[:POPULATION_SIZE]

    # Debug info

```

```

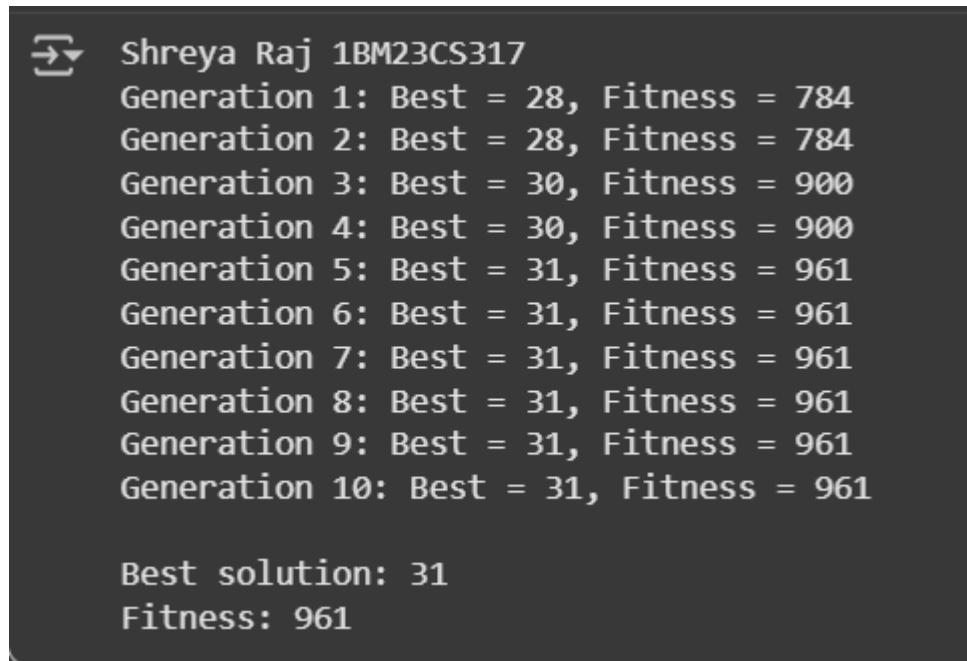
best = max(population, key=lambda x: fitness(binary_to_decimal(x)))
print(f"Generation {generation+1}: Best = {binary_to_decimal(best)}, Fitness =
{fitness(binary_to_decimal(best))}")

# 9. Output best solution
best = max(population, key=lambda x: fitness(binary_to_decimal(x)))
print("\nBest solution:", binary_to_decimal(best))
print("Fitness:", fitness(binary_to_decimal(best)))

# Run the genetic algorithm
genetic_algorithm()

```

Output:



```

→ Shreya Raj 1BM23CS317
Generation 1: Best = 28, Fitness = 784
Generation 2: Best = 28, Fitness = 784
Generation 3: Best = 30, Fitness = 900
Generation 4: Best = 30, Fitness = 900
Generation 5: Best = 31, Fitness = 961
Generation 6: Best = 31, Fitness = 961
Generation 7: Best = 31, Fitness = 961
Generation 8: Best = 31, Fitness = 961
Generation 9: Best = 31, Fitness = 961
Generation 10: Best = 31, Fitness = 961

Best solution: 31
Fitness: 961

```

Program 2: Optimization via Gene Expression Algorithms:

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:

```

        return {"g": g, "c": c}
    return make(a, b), make(b, a)
def select(pop, fit, k=3):
    i = min(random.sample(range(len(pop)), k), key=lambda j: fit[j])
    return {"g": pop[i]["g"][:], "c": dict(pop[i]["c"])}
def evolve(f, target=(-3, 3), pts=64, popn=100, head=10, gens=100,
          cx=0.7, seed=None):
    if seed: random.seed(seed)
    tail = head * (max(funcs.values()) - 1) + 1
    X = [target[0] + (target[1] - target[0]) * i / pts - 1 for i in range(pts)]
    Y = [f(x) for x in X]
    pop = [init_chrom(head, tail) for _ in range(popn)]
    best, fit = float("inf"), None
    for gen in range(gens):
        fits = [mse(ind, X, Y) for ind in pop]
        b = min(range(len(pop)), key=lambda i: fits[i])
        if fits[b] < best: best, fit = fits[b], {"g": pop[b]["g"][:], "c": dict(pop[b]["c"])}
        new = [ft]
        while len(new) < popn:
            p1 = select(pop, fits); p2 = select(pop, fits)
            c1, c2 = crossover(p1, p2) if random.random() < cx
            else (p1, p2)
            mutate(c1, head, tail); mutate(c2, head, tail)
            new += [c1, c2]
        pop = new
        if (gen+1) % 20 == 0: print("Gen", gen+1, "Best", best)
    return fit, best
if __name__ == "__main__":
    f = lambda x: x**3 - 0.5 * x + math.sin(x)
    best, err = evolve(f, gens=100)
    print("Best error:", err)
    print("Frobs:", [safe_eval(best, x) for x in [-2, -1, 0, 1, 2]])

```

Code:

```
print("Shreya Raj 1BM23CS317")
import math, random

# --- Function set ---
FUNCS = {"+":2,"-":2,"*":2,"/":2,"sin":1,"cos":1,"exp":1,"log":1}
APPLY = {
    "+":lambda a,b:a+b, "-":lambda a,b:a-b, "*":lambda a,b:a*b,
    "/":lambda a,b:a if abs(b)<1e-12 else a/b,
    "sin":lambda a:math.sin(a), "cos":lambda a:math.cos(a),
    "exp":lambda a:math.exp(max(-50,min(50,a))),
    "log":lambda a:math.log(abs(a)+1)
}
TERMS=["x","C"]

def init_chrom(head,tail):
    g=[random.choice(list(FUNCS)+TERMS) for _ in range(head)]
    g+=[random.choice(TERMS) for _ in range(tail)]
    c={i:random.uniform(-2,2) for i,s in enumerate(g) if s=="C"}
    return {"g":g,"c":c}

def decode(g):
    seq=[];need=1;i=0
    while need>0 and i<len(g):
        sym=g[i];seq.append((sym,i));need-=1
        if sym in FUNCS: need+=FUNCS[sym]
        i+=1
    return seq

def eval_expr(seq,i,x,c):
    s,pos=seq[i]
    if s in FUNCS:
        args=[];j=i+1
        for _ in range(FUNCS[s]):
            v,j=eval_expr(seq,j,x,c);args.append(v)
        try:return (APPLY[s](*args),j)
        except: return (float("inf"),j)
    return (x if s=="x" else c.get(pos,0),i+1)

def safe_eval(ch,x):
    try:v,_=eval_expr(decode(ch["g"]),0,x,ch["c"])
    except: v=None
```

```

except:return float("inf")
return v if math.isfinite(v) else float("inf")

def mse(ch,X,Y):
    s=0
    for x,y in zip(X,Y):
        d=safe_eval(ch,x)-y
        if not math.isfinite(d): return 1e12
        s+=d*d
    return s/len(X)

def mutate(ch,head,tail,pm=0.05,pc=0.1):
    g,c=ch["g"],ch["c"];n=len(g)
    for i in range(n):
        if random.random()<pm:
            g[i]=random.choice((list(FUNCS)+TERMS) if i<head else TERMS)
            if g[i]==="C":c[i]=random.uniform(-2,2)
            elif i in c:del c[i]
    for i in list(c):
        if random.random()<pc:c[i]+=random.gauss(0,0.1)

def crossover(a,b):
    n=len(a["g"]);cut=random.randint(1,n-1)
    def make(pa,pb):
        g=pa["g"][:cut]+pb["g"][cut:];c={}
        for i,s in enumerate(g):
            if s=="C":c[i]=(pa["c"] if i<cut else pb["c"]).get(i,random.uniform(-2,2))
        return {"g":g,"c":c}
    return make(a,b),make(b,a)

def select(pop,fit,k=3):
    i=min(random.sample(range(len(pop)),k),key=lambda j:fit[j])
    return {"g":pop[i]["g"][:],"c":dict(pop[i]["c"])}

def evolve(f,target=(-3,3),pts=64,popn=100,head=10,gens=100,cx=0.7,seed=None):
    if seed:random.seed(seed)
    tail=head*(max(FUNCS.values())-1)+1
    X=[target[0]+(target[1]-target[0])*i/(pts-1) for i in range(pts)]
    Y=[f(x) for x in X]
    pop=[init_chrom(head,tail) for _ in range(popn)]
    best,fit=float("inf"),None

```

```

for gen in range(gens):
    fits=[mse(ind,X,Y) for ind in pop]
    b=min(range(popn),key=lambda i:fits[i])
    if fits[b]<best:best,fit=fits[b],{"g":pop[b]["g"][:],"c":dict(pop[b]["c"])}
    new=[fit]
    while len(new)<popn:
        p1=select(pop,fits);p2=select(pop,fits)
        c1,c2=crossover(p1,p2) if random.random()<cx else (p1,p2)
        mutate(c1,head,tail);mutate(c2,head,tail)
        new+=[c1,c2]
    pop=new
    if (gen+1)%20==0:print("Gen",gen+1,"Best",best)
return fit,best

if __name__=="__main__":
    f=lambda x:x**3-0.5*x+math.sin(x)
    best,err=evolve(f,gens=100)
    print("Best error:",err)
    print("Preds:",[safe_eval(best,x) for x in [-2,-1,0,1,2]])

```

Output:

```

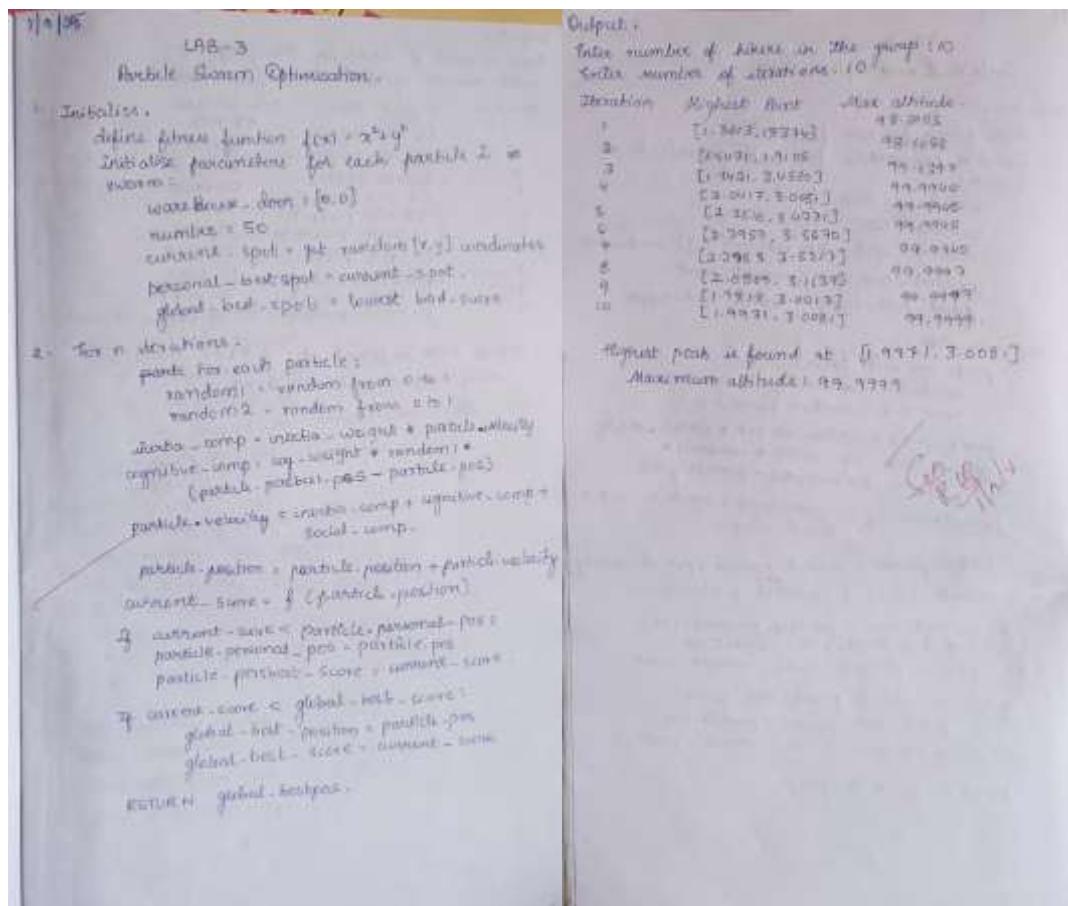
→ Shreya Raj 1BM23CS317
Gen 20 Best 0.2711388144567344
Gen 40 Best 0.08276309290483551
Gen 60 Best 0.02638736211355352
Gen 80 Best 0.0038234107968136122
Gen 100 Best 0.0035796711491496228
Best error: 0.0035796711491496228
Preds: [-7.947649551427884, -1.2578413596603832, 0.0, 1.2578413596603832, 7.947649551427884]

```

Program 3: Particle Swarm Optimization for Function Optimization:

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. Implement the PSO algorithm using Python to optimize a mathematical function.

Algorithm:



Code:

```

import numpy as np
# The peak of this mountain is at (2, 3) with an altitude of 100.
def get_altitude(coords):
    x, y = coords[0], coords[1]
    peak_x, peak_y = 2, 3
    # An inverted paraboloid: 100 - ((x-2)^2 + (y-3)^2)
    return 100 - np.power(x - peak_x, 2) - np.power(y - peak_y, 2)
print("--- Hiker Simulation Setup ---")
try:
  
```

```

n_hikers = int(input("Enter the number of hikers in the group (default: 20): "))
except ValueError:
    print("Invalid input. Using 20 hikers.")
    n_hikers = 20
try:
    n_iterations = int(input("Enter the number of search iterations (default: 50): "))
except ValueError:
    print("Invalid input. Using 50 iterations.")
    n_iterations = 50
print("\n--- Starting The Hike ---")
# --- PSO ALGORITHM PARAMETERS ---
w = 0.5 # Inertia: tendency to keep walking in the same direction
c1 = 0.8 # Cognitive: trust in one's own memory
c2 = 0.9 # Social: trust in the group's best-found spot
bounds = [(-10, 10), (-10, 10)] # The boundaries of the map
# --- INITIALIZE THE HIKERS (PARTICLES) ---
# Each hiker's current coordinates (position)
positions = np.random.rand(n_hikers, 2) * 20 - 10
# Each hiker's current direction and speed (velocity)
velocities = np.random.rand(n_hikers, 2) * 0.1
pbest_positions = np.copy(positions)
pbest_altitudes = np.array([get_altitude(p) for p in positions])
# The group knows the overall best spot found by any hiker
gbest_index = np.argmax(pbest_altitudes) # Note: argmax for maximization
gbest_position = pbest_positions[gbest_index]
gbest_altitude = pbest_altitudes[gbest_index]
print(f"\n{'Iteration':<12} {'Highest Point Found':<25} {'Max Altitude':<15}")
print("-" * 55)
# Main loop where the search happens
for i in range(n_iterations):
    for j in range(n_hikers):
        # UPDATE EACH HIKER'S DIRECTION AND SPEED (VELOCITY)
        r1, r2 = np.random.rand(2)
        cognitive_pull = c1 * r1 * (pbest_positions[j] - positions[j])
        social_pull = c2 * r2 * (gbest_position - positions[j])
        velocities[j] = w * velocities[j] + cognitive_pull + social_pull
        # HIKER MOVES TO A NEW POSITION
        positions[j] = positions[j] + velocities[j]
        # Ensure hikers don't walk off the map
        positions[j] = np.clip(positions[j], bounds[0][0], bounds[0][1])
        # HIKER CHECKS THE ALTITUDE AT THE NEW SPOT
        current_altitude = get_altitude(positions[j])
        if current_altitude > pbest_altitudes[j]: # Note: > for maximization
            pbest_positions[j] = positions[j]
            pbest_altitudes[j] = current_altitude
        # UPDATE GROUP'S BEST IF THIS HIKER FOUND A NEW HIGHEST POINT
        if current_altitude > gbest_altitude: # Note: > for maximization

```

```
gbest_position = positions[j]
gbest_altitude = current_altitude
# --- SHOW PROPER OUTPUT FOR THIS ITERATION ---
pos_str = f"[{gbest_position[0]:.4f}, {gbest_position[1]:.4f}]"
print(f'{i+1:<12} {pos_str:<25} {gbest_altitude:<15.4f}'")
# --- FINAL OUTPUT ---
print("\n" + "*55)
print("Search Complete!")
print(f"The highest peak found by the hikers is at: {gbest_position}")
print(f"Maximum altitude reached: {gbest_altitude}")
print(f"(The actual peak is at [2.0, 3.0] with an altitude of 100.0)")
```

Program4: Ant Colony Optimization for the Traveling Salesman Problem:

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:

LAB-4
Ant Colony Optimisation

Algorithm:

1. Initialise pheromone on all edges
2. For each iteration:
 - each ant builds a tour using pheromone + distance
 - evaluate tour length
 - update pheromone = evaporation + new deposit.
3. Repeat until max iterations
4. Return best tour found

α - influence of pheromone
 β - influence of $1/d$ distance
 ρ - pheromone evaporation rate
number of ants, iterations

Output:

Enter number of routers : 5

Enter network cost matrix:

0	2	0	0	10
2	0	3	0	0
0	3	0	4	0
0	0	4	0	5
10	0	0	5	0

Enter source router : 1

Enter destination router : 4

Best path from 1 to 4: [1, 0, 4]

Best path cost: 12.

Sol P10

Code:

```
import numpy as np
import random

# ----- PARAMETERS -----
NUM_ANTS = 20
NUM_ITER = 100
ALPHA = 1    # influence of pheromone
BETA = 5     # influence of heuristic (1/cost)
RHO = 0.5    # evaporation rate
Q = 100      # pheromone deposit factor

# ----- USER INPUT -----
n = int(input("Enter number of routers (nodes):"))

print("\nEnter the network cost matrix (0 for no link, >0 for link cost):")
cost_matrix = []
for i in range(n):
    row = list(map(int, input().split()))
    cost_matrix.append(row)

cost_matrix = np.array(cost_matrix)
pheromone = np.ones((n, n)) # initial pheromone levels

src = int(input("\nEnter source router (0 to n-1):"))
dst = int(input("Enter destination router (0 to n-1):"))

# ----- HELPER FUNCTIONS -----
def path_cost(path):
    """Compute total cost of a path"""
    return sum(cost_matrix[path[i]][path[i+1]] for i in range(len(path)-1))

# ----- MAIN ACO LOOP -----
best_path = None
best_cost = float('inf')

for iteration in range(NUM_ITER):
    all_paths = []

    for ant in range(NUM_ANTS):
        current = src
        path = [current]
        visited = set([current])

        while current != dst:
            # Possible next hops
```

```

neighbors = [j for j in range(n) if cost_matrix[current][j] > 0 and j not in visited]
if not neighbors: # Dead end
    break

# Probabilities for neighbors
probs = []
for j in neighbors:
    tau = pheromone[current][j] ** ALPHA
    eta = (1 / cost_matrix[current][j]) ** BETA
    probs.append(tau * eta)

probs = np.array(probs)
probs /= probs.sum()

# Choose next router
next_router = random.choices(neighbors, weights=probs)[0]
path.append(next_router)
visited.add(next_router)
current = next_router

if path[-1] == dst:
    all_paths.append(path)
    cost = path_cost(path)
    if cost < best_cost:
        best_path, best_cost = path, cost

# Evaporation
pheromone *= (1 - RHO)

# Deposit pheromone
for path in all_paths:
    cost = path_cost(path)
    deposit = Q / cost
    for i in range(len(path)-1):
        a, b = path[i], path[i+1]
        pheromone[a][b] += deposit
        pheromone[b][a] += deposit # assume undirected

print("\nBest path from router", src, "to", dst, ":", best_path)
print("Best path cost:", best_cost)

```

Program 5: Cuckoo Search (CS):

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:

110125 LAB - V
CUCKOO SEARCH OPTIMIZATION

Procedure:
Initialize value is set to no. of schedules n , probability $\epsilon \in [0,1]$
and maximum iterations MaxIter
Iteration counter $t = 0$
for $i = 1$ to n do
 generate an initial schedule S_i randomly
 evaluate the fitness func $f(S_i)$ = total completion time of
 schedule S_i
End for
while $t < \text{MaxIter}$ do
 Generate a new schedule S'_i from S_i using Levy Flight (small
 random changes in task order)
 Evaluate the fitness $f(S'_i)$
 Randomly select a schedule S_j among n schedules
 If $f(S'_i) < f(S_j)$ then
 Replace S_j with new schedule S'_i
 End if
 Abandon η fraction P_a of worst schedules and generate
 new schedules randomly
 Keep the best schedules found so far
 Rank all schedules by fitness & update the current best
 Increment iteration $t = t + 1$
End while
Output the best schedule S_{best}
Output:
Iteration 1 : Best Fitness = 2.29
Iteration 2 : Best Fitness = 2.28
:
Iteration 100 : Best Fitness = 2.11
Best schedule: [6, 1, 3, 5, 0, 4, 7, 2]
Best Fitness = 2.11

Code:

```
import numpy as np
import math
print('Shreya Raj 1BM23CS317')
def objective_function(x):
    return np.sum(x**2)

def initialize_nests(num_nests, dim, lower_bound, upper_bound):
    return np.random.uniform(lower_bound, upper_bound, size=(num_nests, dim))

def levy_flight(Lambda, size):
    sigma = (math.gamma(1 + Lambda) * math.sin(math.pi * Lambda / 2) /
             (math.gamma((1 + Lambda) / 2) * Lambda * 2**((Lambda - 1) / 2))) ** (1 / Lambda)
    u = np.random.randn(*size) * sigma
    v = np.random.randn(*size)
    step = u / np.abs(v) ** (1 / Lambda)
    return step

def cuckoo_search(num_nests=25, dim=2, lower_bound=-10, upper_bound=10,
                  pa=0.25, max_iter=100):

    nests = initialize_nests(num_nests, dim, lower_bound, upper_bound)
    fitness = np.apply_along_axis(objective_function, 1, nests)

    best_nest = nests[np.argmin(fitness)].copy()
    best_fitness = np.min(fitness)

    for t in range(max_iter):
        new_nests = nests + 0.01 * levy_flight(1.5, nests.shape) * (nests - best_nest)
        new_nests = np.clip(new_nests, lower_bound, upper_bound)

        new_fitness = np.apply_along_axis(objective_function, 1, new_nests)

        mask = new_fitness < fitness
        nests[mask] = new_nests[mask]
        fitness[mask] = new_fitness[mask]

        rand = np.random.rand(num_nests, dim)
        new_nests = np.where(rand > pa, nests,
                             initialize_nests(num_nests, dim, lower_bound, upper_bound))
```

```

new_fitness = np.apply_along_axis(objective_function, 1, new_nests)
mask = new_fitness < fitness
nests[mask] = new_nests[mask]
fitness[mask] = new_fitness[mask]

if np.min(fitness) < best_fitness:
    best_nest = nests[np.argmin(fitness)].copy()
    best_fitness = np.min(fitness)

print(f"Iteration {t+1}/{max_iter} | Best Fitness: {best_fitness:.6f}")

return best_nest, best_fitness

```

best_solution, best_value = cuckoo_search()
print("\nBest solution found:", best_solution)
print("Best fitness value:", best_value)

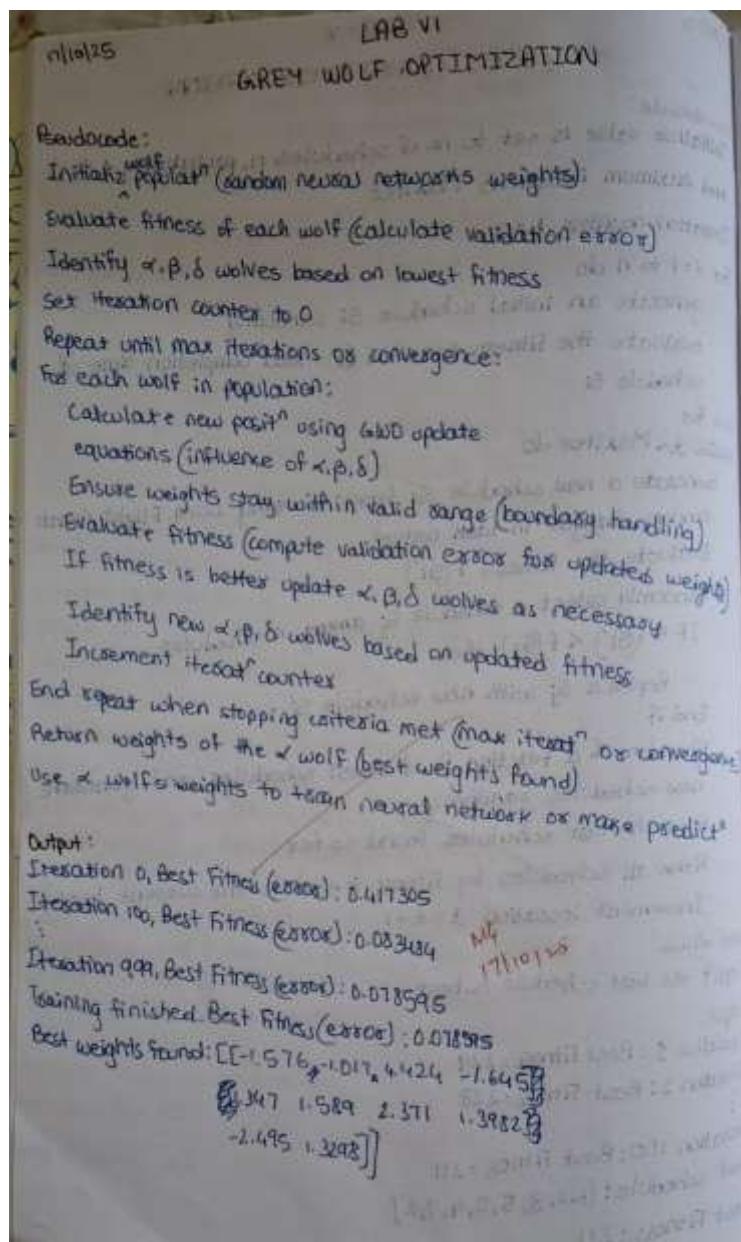
Output:

...	Shreya Raj 1BM23CS317
Iteration 1/100	Best Fitness: 14.471185
Iteration 2/100	Best Fitness: 14.471185
Iteration 3/100	Best Fitness: 14.471185
Iteration 4/100	Best Fitness: 0.894298
Iteration 5/100	Best Fitness: 0.894298
Iteration 6/100	Best Fitness: 0.894298
Iteration 7/100	Best Fitness: 0.894298
Iteration 8/100	Best Fitness: 0.894298
Iteration 9/100	Best Fitness: 0.564269
Iteration 10/100	Best Fitness: 0.564269
Iteration 11/100	Best Fitness: 0.564269
Iteration 12/100	Best Fitness: 0.457079
Iteration 13/100	Best Fitness: 0.457079
Iteration 14/100	Best Fitness: 0.457079
Iteration 15/100	Best Fitness: 0.457079
Iteration 16/100	Best Fitness: 0.457079
Iteration 17/100	Best Fitness: 0.457079
Iteration 18/100	Best Fitness: 0.457079
Iteration 19/100	Best Fitness: 0.457079
Iteration 20/100	Best Fitness: 0.457079
Iteration 21/100	Best Fitness: 0.457079
Iteration 22/100	Best Fitness: 0.457079
Iteration 23/100	Best Fitness: 0.457079
Iteration 24/100	Best Fitness: 0.457079
Iteration 25/100	Best Fitness: 0.457079
Iteration 26/100	Best Fitness: 0.457079
Iteration 27/100	Best Fitness: 0.457079
Iteration 80/100	Best Fitness: 0.000133
Iteration 81/100	Best Fitness: 0.000133
Iteration 82/100	Best Fitness: 0.000133
Iteration 83/100	Best Fitness: 0.000133
Iteration 84/100	Best Fitness: 0.000133
Iteration 85/100	Best Fitness: 0.000133
Iteration 86/100	Best Fitness: 0.000133
Iteration 87/100	Best Fitness: 0.000133
Iteration 88/100	Best Fitness: 0.000133
Iteration 89/100	Best Fitness: 0.000133
Iteration 90/100	Best Fitness: 0.000133
Iteration 91/100	Best Fitness: 0.000133
Iteration 92/100	Best Fitness: 0.000133
Iteration 93/100	Best Fitness: 0.000133
Iteration 94/100	Best Fitness: 0.000133
Iteration 95/100	Best Fitness: 0.000133
Iteration 96/100	Best Fitness: 0.000133
Iteration 97/100	Best Fitness: 0.000133
Iteration 98/100	Best Fitness: 0.000133
Iteration 99/100	Best Fitness: 0.000133
Iteration 100/100	Best Fitness: 0.000133
Best solution found: [-0.01150773 0.00076445]	
Best fitness value: 0.0001330122491796386	

Program 6: Grey Wolf Optimizer (GWO):

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:



Code:

```
import numpy as np
print('Shreya Raj 1BM23CS317')
def objective_function(x):
    return np.sum(x**2)

def grey_wolf_optimizer(num_wolves=30, dim=2, max_iter=50, lower_bound=-10,
upper_bound=10):
    wolves = np.random.uniform(lower_bound, upper_bound, (num_wolves, dim))

    Alpha_pos = np.zeros(dim)
    Beta_pos = np.zeros(dim)
    Delta_pos = np.zeros(dim)

    Alpha_score = float("inf")
    Beta_score = float("inf")
    Delta_score = float("inf")

    for t in range(max_iter):
        for i in range(num_wolves):
            wolves[i] = np.clip(wolves[i], lower_bound, upper_bound)

            fitness = objective_function(wolves[i])

            if fitness < Alpha_score:
                Delta_score = Beta_score
                Delta_pos = Beta_pos.copy()
                Beta_score = Alpha_score
                Beta_pos = Alpha_pos.copy()
                Alpha_score = fitness
                Alpha_pos = wolves[i].copy()
            elif fitness < Beta_score:
                Delta_score = Beta_score
                Delta_pos = Beta_pos.copy()
                Beta_score = fitness
                Beta_pos = wolves[i].copy()
            elif fitness < Delta_score:
                Delta_score = fitness
                Delta_pos = wolves[i].copy()

    a = 2 - t * (2 / max_iter)
```

```

for i in range(num_wolves):
    for j in range(dim):
        r1 = np.random.rand()
        r2 = np.random.rand()

        A1 = 2 * a * r1 - a
        C1 = 2 * r2
        D_alpha = abs(C1 * Alpha_pos[j] - wolves[i][j])
        X1 = Alpha_pos[j] - A1 * D_alpha

        r1 = np.random.rand()
        r2 = np.random.rand()
        A2 = 2 * a * r1 - a
        C2 = 2 * r2
        D_beta = abs(C2 * Beta_pos[j] - wolves[i][j])
        X2 = Beta_pos[j] - A2 * D_beta

        r1 = np.random.rand()
        r2 = np.random.rand()
        A3 = 2 * a * r1 - a
        C3 = 2 * r2
        D_delta = abs(C3 * Delta_pos[j] - wolves[i][j])
        X3 = Delta_pos[j] - A3 * D_delta

        wolves[i][j] = (X1 + X2 + X3) / 3

    print(f"Iteration {t+1}/{max_iter} | Best Fitness: {Alpha_score:.6f}")

return Alpha_pos, Alpha_score

best_position, best_score = grey_wolf_optimizer()
print("\nBest solution found:", best_position)
print("Best fitness value:", best_score)

```

Output:

```
Shreya Raj IBM23CS317
Iteration 1/50 | Best Fitness: 1.827641
Iteration 2/50 | Best Fitness: 0.889744
Iteration 3/50 | Best Fitness: 0.587311
Iteration 4/50 | Best Fitness: 0.211439
Iteration 5/50 | Best Fitness: 0.005600
Iteration 6/50 | Best Fitness: 0.000891
Iteration 7/50 | Best Fitness: 0.000070
Iteration 8/50 | Best Fitness: 0.000045
Iteration 9/50 | Best Fitness: 0.000014
Iteration 10/50 | Best Fitness: 0.000001
Iteration 11/50 | Best Fitness: 0.000001
Iteration 12/50 | Best Fitness: 0.000000
Iteration 13/50 | Best Fitness: 0.000000
Iteration 14/50 | Best Fitness: 0.000000
Iteration 15/50 | Best Fitness: 0.000000
Iteration 16/50 | Best Fitness: 0.000000
Iteration 17/50 | Best Fitness: 0.000000
Iteration 18/50 | Best Fitness: 0.000000
Iteration 19/50 | Best Fitness: 0.000000
Iteration 20/50 | Best Fitness: 0.000000
Iteration 21/50 | Best Fitness: 0.000000
Iteration 22/50 | Best Fitness: 0.000000
Iteration 23/50 | Best Fitness: 0.000000
Iteration 24/50 | Best Fitness: 0.000000
Iteration 25/50 | Best Fitness: 0.000000
Iteration 26/50 | Best Fitness: 0.000000
Iteration 27/50 | Best Fitness: 0.000000
Iteration 28/50 | Best Fitness: 0.000000
Iteration 29/50 | Best Fitness: 0.000000
```

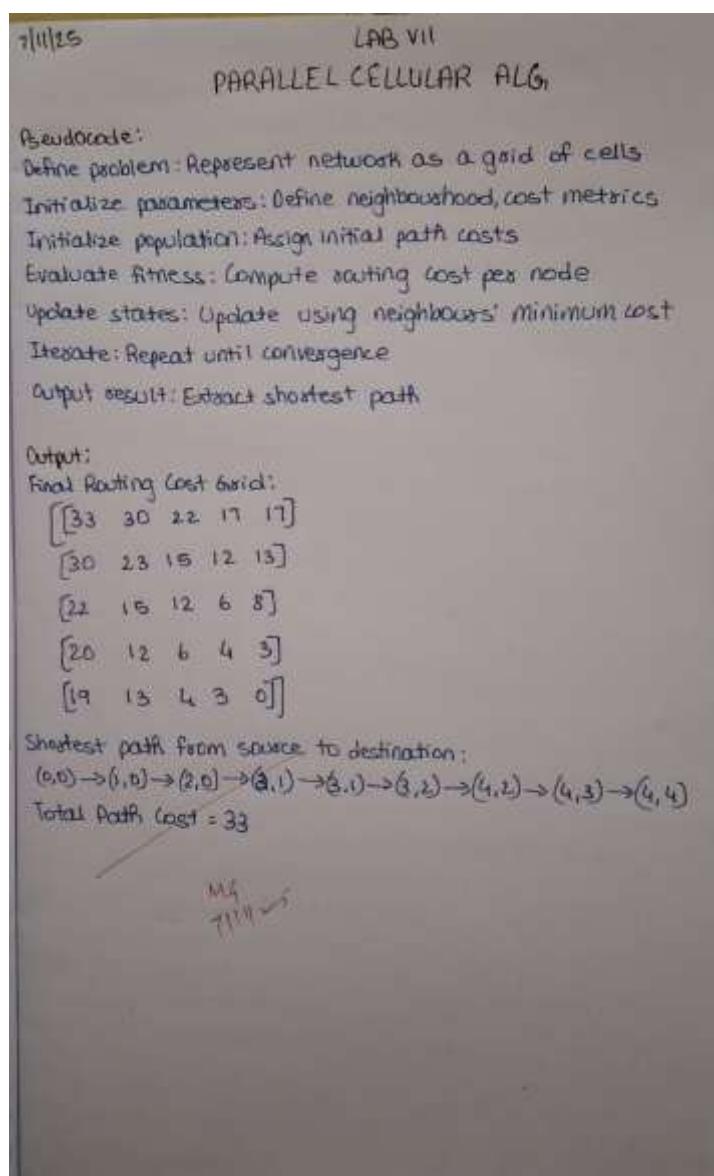
```
Iteration 29/50 | Best Fitness: 0.000000
Iteration 30/50 | Best Fitness: 0.000000
Iteration 31/50 | Best Fitness: 0.000000
Iteration 32/50 | Best Fitness: 0.000000
Iteration 33/50 | Best Fitness: 0.000000
Iteration 34/50 | Best Fitness: 0.000000
Iteration 35/50 | Best Fitness: 0.000000
Iteration 36/50 | Best Fitness: 0.000000
Iteration 37/50 | Best Fitness: 0.000000
Iteration 38/50 | Best Fitness: 0.000000
Iteration 39/50 | Best Fitness: 0.000000
Iteration 40/50 | Best Fitness: 0.000000
Iteration 41/50 | Best Fitness: 0.000000
Iteration 42/50 | Best Fitness: 0.000000
Iteration 43/50 | Best Fitness: 0.000000
Iteration 44/50 | Best Fitness: 0.000000
Iteration 45/50 | Best Fitness: 0.000000
Iteration 46/50 | Best Fitness: 0.000000
Iteration 47/50 | Best Fitness: 0.000000
Iteration 48/50 | Best Fitness: 0.000000
Iteration 49/50 | Best Fitness: 0.000000
Iteration 50/50 | Best Fitness: 0.000000

Best solution found: [5.98100037e-14 3.38310133e-14]
Best fitness value: 4.721774001251392e-27
```

Program 7: Parallel Cellular Algorithms and Programs

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
print('Shreya Raj 1BM23CS317')
GRID_SIZE = 5 # 5x5 network grid
MAX_ITER = 100
INF = 1e9

# Define source and destination
source = (0, 0)
destination = (4, 4)

# Create cost matrix for each link (random or distance-based)
np.random.seed(42)
cost_matrix = np.random.randint(1, 10, size=(GRID_SIZE, GRID_SIZE))

# Initialize state matrix (each cell's current best cost to destination)
state = np.full((GRID_SIZE, GRID_SIZE), INF)
state[destination] = 0 # destination cost = 0

# Define 4-neighborhood (up, down, left, right)
neighbors = [(-1, 0), (1, 0), (0, -1), (0, 1)]
def get_neighbors(i, j):
    """Return valid neighboring cells"""
    valid_neighbors = []
    for dx, dy in neighbors:
        ni, nj = i + dx, j + dy
        if 0 <= ni < GRID_SIZE and 0 <= nj < GRID_SIZE:
            valid_neighbors.append((ni, nj))
    return valid_neighbors

for iteration in range(MAX_ITER):
    new_state = state.copy()
    for i in range(GRID_SIZE):
        for j in range(GRID_SIZE):
            if (i, j) == destination:
                continue
            neighbor_costs = []
            for ni, nj in get_neighbors(i, j):
                # Update rule: min(cost to neighbor + neighbor's state)
                total_cost = cost_matrix[ni, nj] + state[ni, nj]
                neighbor_costs.append(total_cost)
            state[i, j] = min(neighbor_costs)
```

```

neighbor_costs.append(total_cost)
if neighbor_costs:
    new_state[i, j] = min(neighbor_costs)
# Check for convergence
if np.allclose(new_state, state):
    print(f"Converged after {iteration} iterations.")
    break
state = new_state

path = [source]
current = source
while current != destination:
    i, j = current
    nbs = get_neighbors(i, j)
    next_cell = min(nbs, key=lambda n: state[n])
    path.append(next_cell)
    current = next_cell
print("Final Routing Cost Grid:")
print(np.round(state, 2))
print("\nShortest Path from Source to Destination:")
print(" → ".join([str(p) for p in path]))
print(f"\nTotal Path Cost: {state[source]}")

```

Output:

```

Shreya Raj 1BM23CS317
Converged after 8 iterations.
Final Routing Cost Grid:
[[33. 30. 22. 17. 17.]
 [30. 23. 15. 12. 13.]
 [22. 15. 12. 6. 8.]
 [20. 12. 6. 4. 3.]
 [19. 13. 4. 3. 0.]]

Shortest Path from Source to Destination:
(0, 0) → (1, 0) → (2, 0) → (2, 1) → (3, 1) → (3, 2) → (4, 2) → (4, 3) → (4, 4)

Total Path Cost: 33.0

```