VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB RECORD

Bio Inspired Systems (23CS5BSBIS)

Submitted by

Pranay Kommuri (1BM23CS242)

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
Sep-2024 to Jan-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

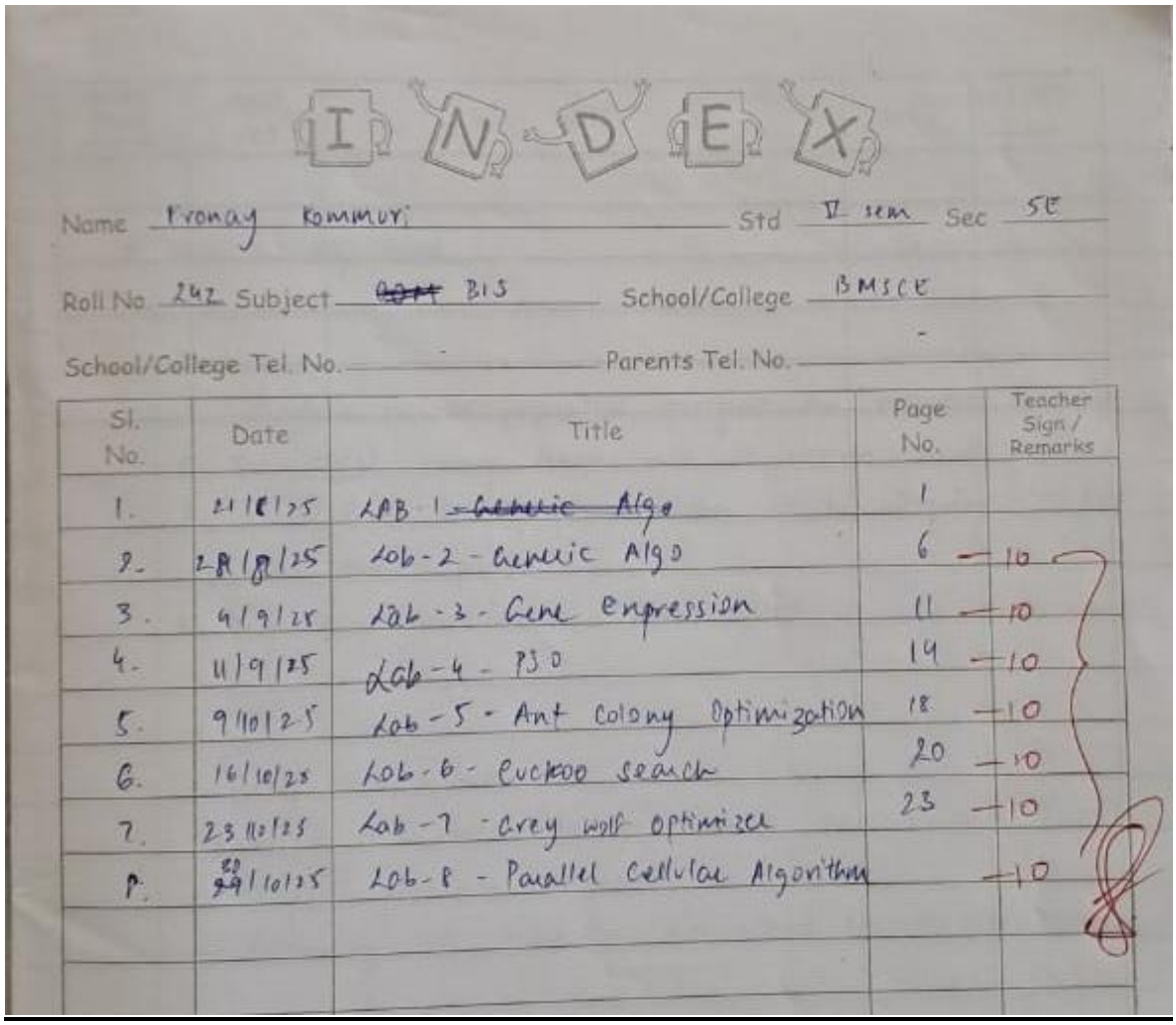
This is to certify that the Lab work entitled "Bio-Inspired Systems (23CS5BSBIS)" carried out by **Pranay Kommuri (1BM23CS242),** who is bonafide student of **B.M.S. College of Engineering.** It is in partial fulfillment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the Visvesvaraya Technological University, Belgaum. The Lab report has been approved as it satisfies the academic requirements of the above mentioned subject and the work prescribed for the said degree.

Prof. Swathi Sridharan Assistant Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE

Index

Github Link:
https://github.com/pranav-kommuri/1BM23CS242_BIS

Observation Index



INDEX

Name Pranay Kommuri Std V sem Sec 5E

Roll No. 242 Subject ~~OOPS~~ BIS School/College BMSCE

School/College Tel. No. Parents Tel. No.

Sl. No.	Date	Title	Page No.	Teacher Sign / Remarks
1.	21/8/25	LAB 1 - ~~Genetic Algo~~	1	
2.	28/8/25	Lab-2 - Genetic Algo	6 — 10	
3.	4/9/25	Lab-3 - Gene expression	11 — 10	
4.	11/9/25	Lab-4 - PSO	14 — 10	
5.	9/10/25	Lab-5 - Ant Colony Optimization	18 — 10	
6.	16/10/25	Lab-6 - Cuckoo search	20 — 10	
7.	23/10/25	Lab-7 - Grey wolf optimizer	23 — 10	
8.	29/10/25	Lab-8 - Parallel Cellular Algorithm	— 10	

Program 1

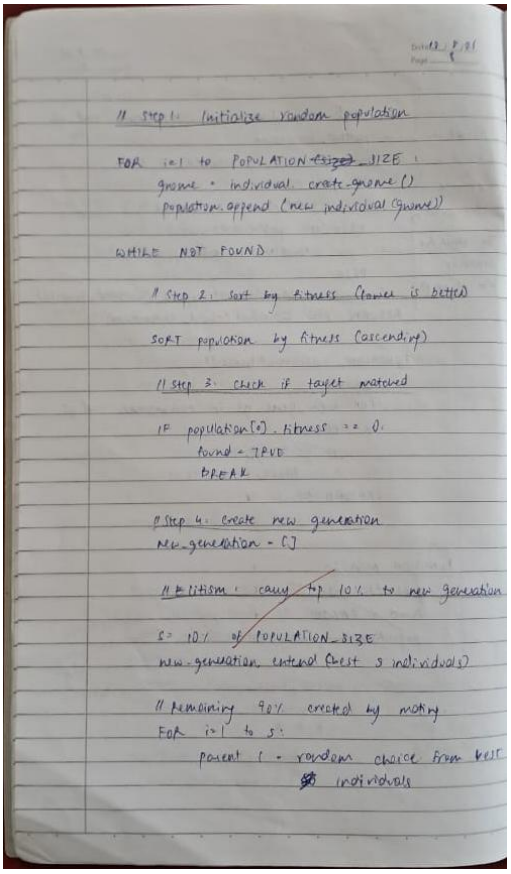
Genetic Algorithm for Optimization Problems:

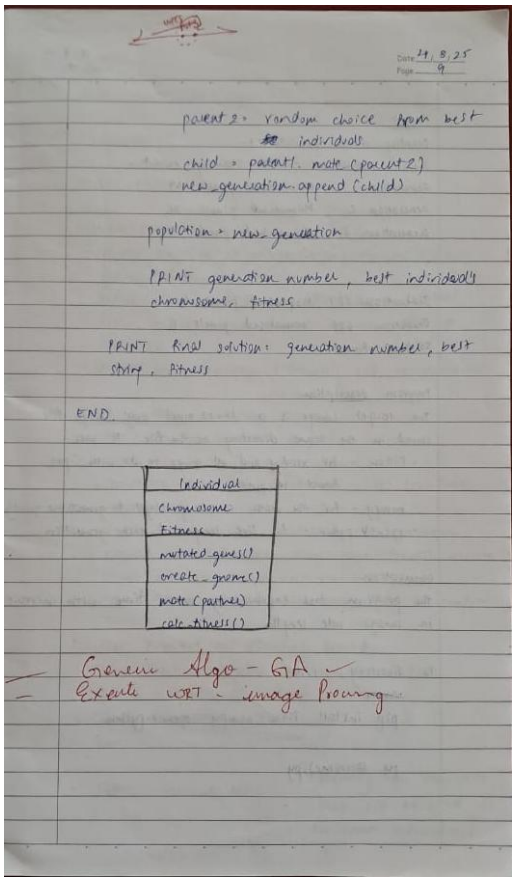
Genetic Algorithms (GA) are inspired by the process of natural selection and genetics, where the fittest individuals are selected for reproduction to produce the next generation. GAs are widely used for solving optimization and search problems. Implement a Genetic Algorithm using Python to solve a basic optimization problem, such as finding the maximum value of a mathematical function.

Implementation Steps:

1. Define the Problem: Create a mathematical function to optimize.
2. Initialize Parameters: Set the population size, mutation rate, crossover rate, and number of generations.
3. Create Initial Population: Generate an initial population of potential solutions.
4. Evaluate Fitness: Evaluate the fitness of each individual in the population.
5. Selection: Select individuals based on their fitness to reproduce.
6. Crossover: Perform crossover between selected individuals to produce offspring.
7. Mutation: Apply mutation to the offspring to maintain genetic diversity.
8. Iteration: Repeat the evaluation, selection, crossover, and mutation processes for a fixed number of generations or until convergence criteria are met.
9. Output the Best Solution: Track and output the best solution found during the generations.

Algorithm:





Code:

```
import random, os
import numpy as np
from PIL import Image
import cv2

POP, W, H = 100, 10, 10
T_PATH, OUT = 'target_image10.png', 'output'

# Load and process target
try:
    TGT = np.array(Image.open(T_PATH).resize((W, H)).convert('L'))
    TGT = np.where(TGT > 127, 255, 0).astype(np.uint8)
except: exit(f"Error: Missing {T_PATH}")

if not os.path.exists(OUT): os.makedirs(OUT)

class Ind:
    def __init__(self, dna):
        self.dna = dna
        self.fit = np.sum(np.abs(dna.astype(int) - TGT.astype(int)))

    @classmethod
    def gen(cls):
        return np.random.choice([0, 255], (H, W)).astype(np.uint8)

    def mate(self, p2):
        # Vectorized crossover: 45% self, 45% p2, 10% mutation
        rnd = np.random.random((H, W))
        mut = np.random.choice([0, 255], (H, W))

        child = np.where(rnd < 0.45, self.dna,
                np.where(rnd < 0.90, p2.dna, mut))
        return Ind(child.astype(np.uint8))

def main():
    gen, pop = 1, [Ind(Ind.gen()) for _ in range(POP)]
    print("Starting Evolution...")

    while True:
        pop.sort(key=lambda x: x.fit)
        best = pop[0]
        print(f"Gen: {gen}\t Err: {best.fit // 255}")

        cv2.imshow('Evo', best.dna)
        if cv2.waitKey(1) == ord('q') or best.fit == 0: break

        # Elitism (10%) + Mating (90%)
        new_pop = pop[:int(0.1 * POP)]
        parents = pop[:50] # Top 50% pool

        for _ in range(int(0.9 * POP)):
            new_pop.append(random.choice(parents).mate(random.choice(parents)))

        pop = new_pop
        gen += 1

    # Save and wait
    Image.fromarray(pop[0].dna).save(f"{OUT}/final.png")
    print("Done.")
    cv2.waitKey(0); cv2.destroyAllWindows()

if __name__ == '__main__': main()
```

Program 2

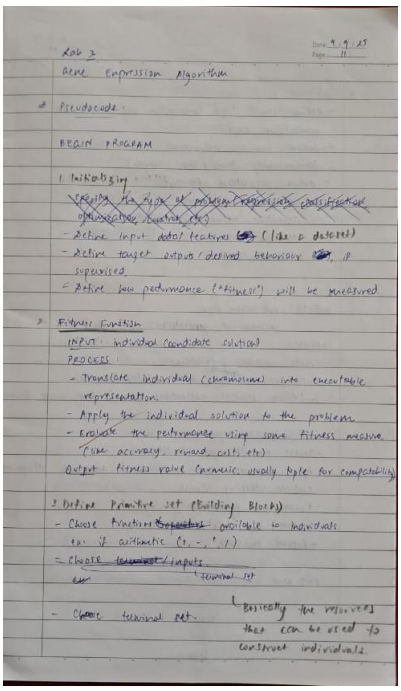
Program 2 Optimization via Gene Expression Algorithms:

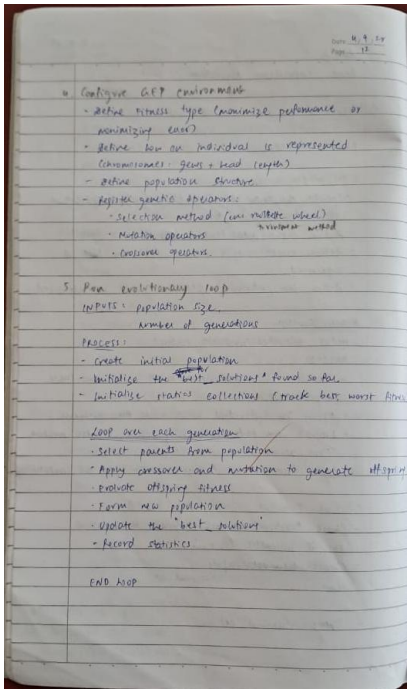
Gene Expression Algorithms (GEA) are inspired by the biological process of gene expression in living organisms. This process involves the translation of genetic information encoded in DNA into functional proteins. In GEA, solutions to optimization problems are encoded in a manner similar to genetic sequences. 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.

Implementation Steps:

1. Define the Problem: Create a mathematical function to optimize.
2. Initialize Parameters: Set the population size, number of genes, mutation rate, crossover rate, and number of generations.
3. Initialize Population: Generate an initial population of random genetic sequences.
4. Evaluate Fitness: Evaluate the fitness of each genetic sequence based on the optimization function.
5. Selection: Select genetic sequences based on their fitness for reproduction.
6. Crossover: Perform crossover between selected sequences to produce offspring.
7. Mutation: Apply mutation to the offspring to introduce variability.
8. Gene Expression: Translate genetic sequences into functional solutions.
9. Iterate: Repeat the selection, crossover, mutation, and gene expression processes for a fixed number of generations or until convergence criteria are met.
10. Output the Best Solution: Track and output the best solution found during the iterations

Algorithm:





Code:

```
import random, os
import numpy as np
from PIL import Image
import cv2

POP, W, H = 100, 10, 10
T_PATH, OUT = 'target_image10.png', 'output'

# Load and process target
try:
    TGT = np.array(Image.open(T_PATH).resize((W, H)).convert('L'))
    TGT = np.where(TGT > 127, 255, 0).astype(np.uint8)
except: exit(f"Error: Missing {T_PATH}")

if not os.path.exists(OUT): os.makedirs(OUT)

class Ind:
    def __init__(self, dna):
        self.dna = dna
        self.fit = np.sum(np.abs(dna.astype(int) - TGT.astype(int)))

    @classmethod
    def gen(cls):
        return np.random.choice([0, 255], (H, W)).astype(np.uint8)

    def mate(self, p2):
        # Vectorized crossover: 45% self, 45% p2, 10% mutation
        rnd = np.random.random((H, W))
        mut = np.random.choice([0, 255], (H, W))

        child = np.where(rnd < 0.45, self.dna,
                np.where(rnd < 0.90, p2.dna, mut))
        return Ind(child.astype(np.uint8))

def main():
    gen, pop = 1, [Ind(Ind.gen()) for _ in range(POP)]
    print("Starting Evolution...")

    while True:
        pop.sort(key=lambda x: x.fit)
        best = pop[0]
        print(f"Gen: {gen}\t Err: {best.fit // 255}")

        cv2.imshow('Evo', best.dna)
        if cv2.waitKey(1) == ord('q') or best.fit == 0: break

        # Elitism (10%) + Mating (90%)
        new_pop = pop[:int(0.1 * POP)]
        parents = pop[:50] # Top 50% pool

        for _ in range(int(0.9 * POP)):
            new_pop.append(random.choice(parents).mate(random.choice(parents)))

        pop = new_pop
        gen += 1

    # Save and wait
    Image.fromarray(pop[0].dna).save(f"{OUT}/final.png")
    print("Done.")
    cv2.waitKey(0); cv2.destroyAllWindows()

if __name__ == '__main__': main()
```

Program 3

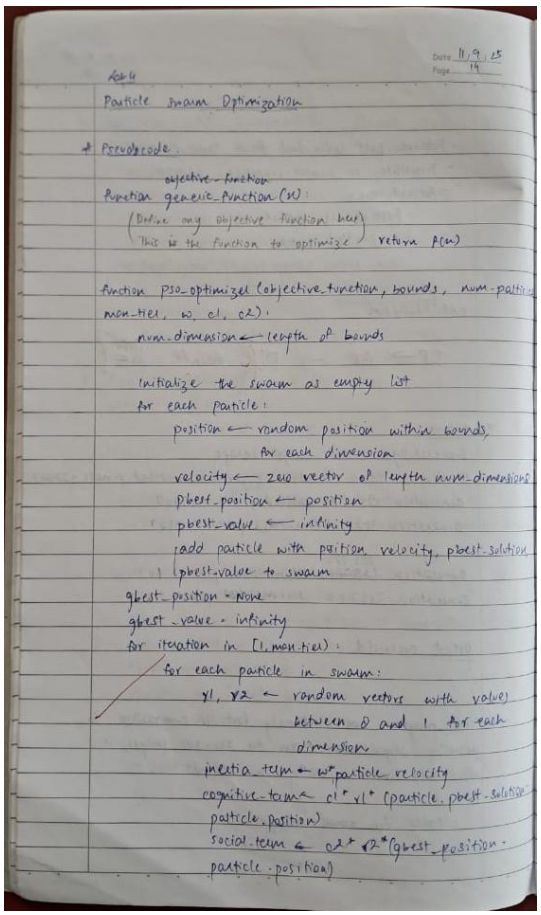
Particle Swarm Optimization for Function Optimization:

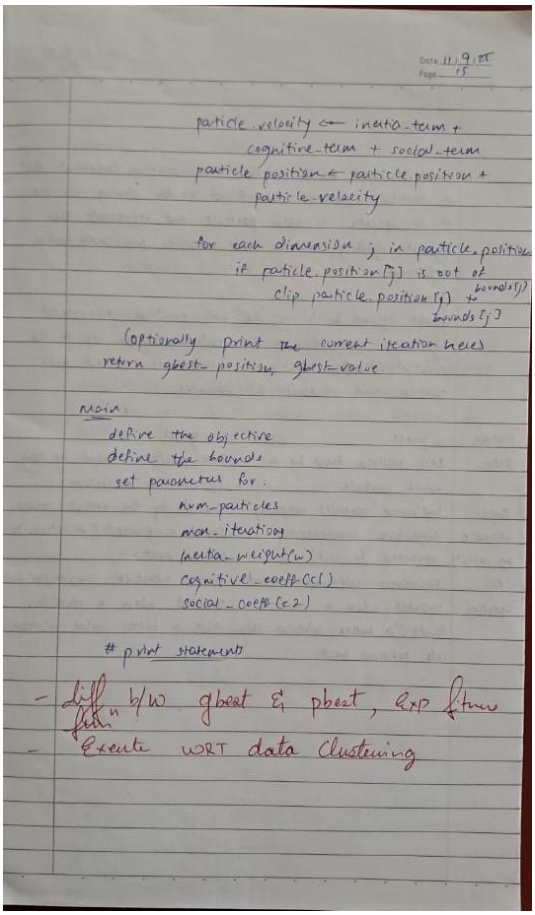
Particle Swarm Optimization (PSO) is inspired by the social behaviour of birds flocking or fish schooling. PSO is used to find optimal solutions by iteratively improving a candidate solution with regard to a given measure of quality. Implement the PSO algorithm using Python to optimize a mathematical function.

Implementation Steps:

1. Define the Problem: Create a mathematical function to optimize.
2. Initialize Parameters: Set the number of particles, inertia weight, cognitive and social coefficients.
3. Initialize Particles: Generate an initial population of particles with random positions and velocities.
4. Evaluate Fitness: Evaluate the fitness of each particle based on the optimization function.
5. Update Velocities and Positions: Update the velocity and position of each particle based on its own best position and the global best position.
6. Iterate: Repeat the evaluation, updating, and position adjustment for a fixed number of iterations or until convergence criteria are met.
7. Output the Best Solution: Track and output the best solution found during the iterations.

Algorithm:





Code:

```
import numpy as np
import matplotlib
matplotlib.use('Agg')
import matplotlib.pyplot as plt
from sklearn.datasets import make_blobs

class Part:
    def __init__(self, K, dim, lo, hi):
        self.pos = np.random.uniform(lo, hi, K * dim)
        self.vel = np.random.rand(K * dim) * 0.1
        self.best_pos, self.best_fit = self.pos.copy(), float('inf')

class PSO:
    def __init__(self, K, n_par, data, itr=100):
        self.K, self.n_par, self.data, self.itr = K, n_par, data, itr
        self.N, self.dim = data.shape

        lo, hi = np.min(data, 0), np.max(data, 0)
        self.lo, self.hi = np.tile(lo, K), np.tile(hi, K)
        self.swarm = [Part(K, self.dim, self.lo, self.hi) for _ in range(n_par)]
        self.g_pos, self.g_fit = None, float('inf')

    def get_fit(self, cents):
        # Vectorized distance: (N, 1, Dim) - (1, K, Dim) -> (N, K)
        dists = np.linalg.norm(self.data[:, None] - cents, axis=2)
        return np.sum(np.min(dists, axis=1)**2) / self.N

    def run(self):
        print("Starting PSO...")
        for i in range(self.itr):
            for p in self.swarm:
                fit = self.get_fit(p.pos.reshape(self.K, self.dim))

                if fit < p.best_fit: p.best_fit, p.best_pos = fit, p.pos.copy()
                if fit < self.g_fit: self.g_fit, self.g_pos = fit, p.pos.copy()

            for p in self.swarm:
                r1, r2 = np.random.rand(2, len(p.pos))
                # Update velocity: w=0.5, c1=1.5, c2=1.5
                p.vel = 0.5 * p.vel + 1.5 * r1 * (p.best_pos - p.pos) + \
                        1.5 * r2 * (self.g_pos - p.pos)
                p.pos = np.clip(p.pos + p.vel, self.lo, self.hi)

            if (i+1) % 10 == 0: print(f"Iter {i+1} SSE: {self.g_fit:.4f}")

        return self.g_pos.reshape(self.K, self.dim)

    def predict(self, cents):
        return np.argmin(np.linalg.norm(self.data[:, None] - cents, axis=2), axis=1)

if __name__ == "__main__":
    X, y = make_blobs(300, centers=4, cluster_std=0.8, random_state=42)

    pso = PSO(4, 30, X, 100)
    cents = pso.run()
    labels = pso.predict(cents)

    plt.figure(figsize=(8, 6))
    plt.scatter(X[:,0], X[:,1], c=labels, alpha=0.6)
    plt.scatter(cents[:,0], cents[:,1], c='r', marker='X', s=200)
    plt.savefig('pso_clusters.png')
    print("Plot saved to pso_clusters.png")
```

Program 4

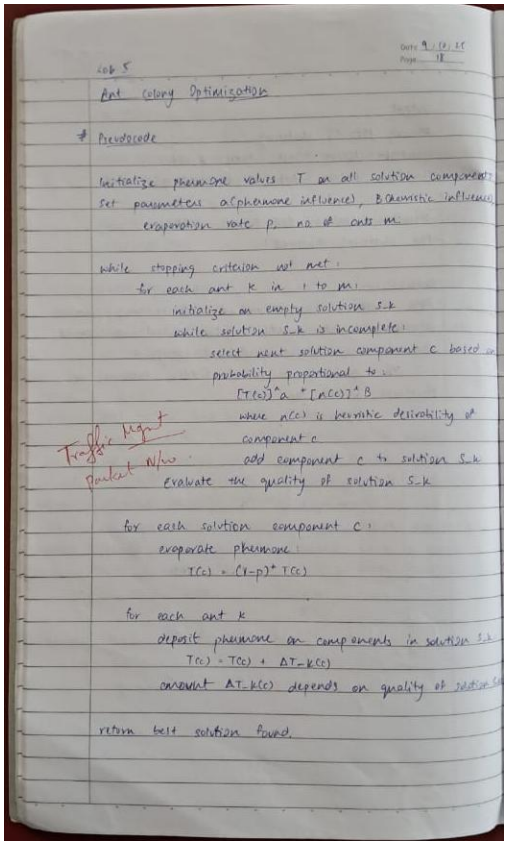
Ant Colony Optimization for the Traveling Salesman Problem:

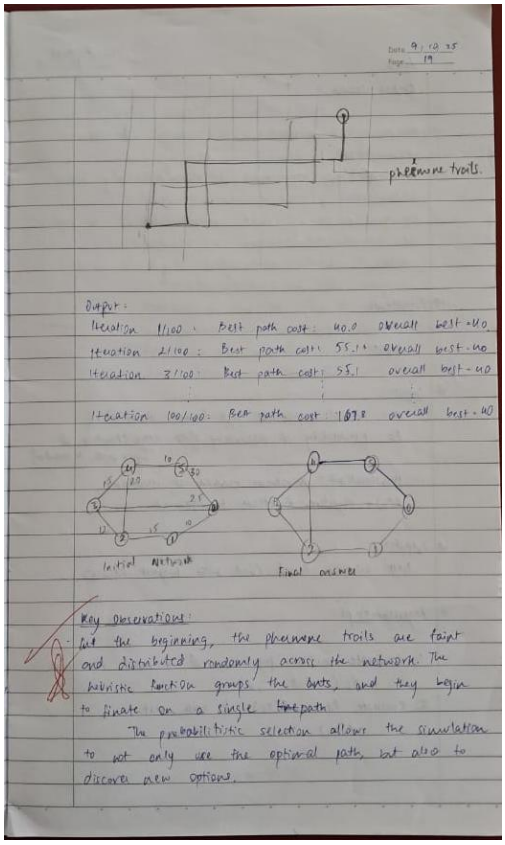
The foraging behaviour of ants has inspired the development of optimization algorithms that can solve complex problems such as the Traveling Salesman Problem (TSP). Ant Colony Optimization (ACO) simulates the way ants find the shortest path between food sources and their nest. Implement the ACO algorithm 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.

Implementation Steps:

1. Define the Problem: Create a set of cities with their coordinates.
2. Initialize Parameters: Set the number of ants, the importance of pheromone (alpha), the importance of heuristic information (beta), the evaporation rate (rho), and the initial pheromone value.
3. Construct Solutions: Each ant constructs a solution by probabilistically choosing the next city based on pheromone trails and heuristic information.
4. Update Pheromones: After all ants have constructed their solutions, update the pheromone trails based on the quality of the solutions found.
5. Iterate: Repeat the construction and updating process for a fixed number of iterations or until convergence criteria are met.
6. Output the Best Solution: Keep track of and output the best solution found during the iterations.

Algorithm:





Code:

```
import numpy as np
import random, cv2, math

class Vis:
    def __init__(self, G, N, title="ACO"):
        self.G, self.N, self.title = G, N, title
        self.SZ, self.R, self.F = (800, 800), 25, cv2.FONT_HERSHEY_SIMPLEX
        c, r = (400, 400), 320
        self.pos = {i: (int(c[0]+r*math.cos(2*math.pi*i/N)), int(c[1]+r*math.sin(2*math.pi*i/N))) for i in
range(N)}
    def base(self, img, w=False):
        for i in range(self.N):
            for j in range(i+1, self.N):
                if self.G[i,j] != float('inf'):
                    p1, p2 = self.pos[i], self.pos[j]
                    cv2.line(img, p1, p2, (200,200,200), 1)
                    if w:
                        m = ((p1[0]+p2[0])//2, (p1[1]+p2[1])//2)
                        cv2.putText(img, str(int(self.G[i,j])), (m[0]+5, m[1]-5), self.F, 0.6, (150,0,0), 2)
        for i in range(self.N):
            cv2.circle(img, self.pos[i], self.R, (255,150,0), -1)
            cv2.circle(img, self.pos[i], self.R, 0, 2)
            cv2.putText(img, str(i), (self.pos[i][0]-10, self.pos[i][1]+10), self.F, 0.8, (255,255,255), 2)

    def draw(self, phero, path, itr, cost, save_as=None):
        img = np.full((*self.SZ, 3), 255, dtype=np.uint8)
        self.base(img, w=False)
        mn, mx = phero.min(), phero.max()
        mx = mx if mx != mn else mx + 1e-9

        for i in range(self.N):
            for j in range(i+1, self.N):
                if self.G[i,j] != float('inf'):
                    lvl = (phero[i,j]-mn)/(mx-mn)
                    cv2.line(img, self.pos[i], self.pos[j], (255, int(100+lvl*155), 0), int(1+lvl*10))

        if path:
            for k in range(len(path)-1):
                cv2.line(img, self.pos[path[k]], self.pos[path[k+1]], (0,200,0), 4)

        cv2.putText(img, f"Iter: {itr} Cost: {cost:.2f}", (20,40), self.F, 1, 0, 2)

        if save_as:
            cv2.imwrite(save_as, img)
            print(f"Saved: {save_as}")
        else:
            cv2.imshow(self.title, img)
            return cv2.waitKey(50) != ord('q')
        return True

class ACO:
    def __init__(self, G, vis, ants, itr, a, b, rho, q):
        self.G, self.vis, self.ants, self.itr = G, vis, ants, itr
        self.a, self.b, self.rho, self.q = a, b, rho, q
        self.N = len(G)
        self.phero, self.cong = np.ones_like(G)/self.N, np.zeros_like(G)

    def run(self, s, e):
        best_p, best_c = None, float('inf')

        # Save initial state
        if self.vis:
            img = np.full((800,800,3), 255, np.uint8)
            self.vis.base(img, w=True)
            cv2.imwrite("initial_net.png", img)
```

```
        for i in range(self.itr):
            paths = [self.ant(s, e) for _ in range(self.ants)]
            self.upd_phero(paths)
            curr_p, curr_c = min((p for p in paths if p[0]), key=lambda x: x[1], default=(None,float('inf')))

            if curr_c < best_c: best_p, best_c = curr_p, curr_c
            print(f"Iter {i+1}: Best {curr_c:.2f} (All-time {best_c:.2f})")

            # Random Congestion
            u, v = random.randint(0, self.N-1), random.randint(0, self.N-1)
            if self.G[u,v] < float('inf'):
                self.cong[u,v] += random.uniform(2, 5); self.cong[v,u] += random.uniform(2, 5)

            if self.vis and not self.vis.draw(self.phero, best_p, i+1, best_c): break

        if self.vis: self.vis.draw(self.phero, best_p, "Final", best_c, "final_path.png")
        cv2.destroyAllWindows()
        return best_p, best_c

    def ant(self, s, e):
        path, cur = [s], s
        while cur != e:
            nxt = self.next_node(cur, path)
            if nxt is None: return None, float('inf')
            path.append(nxt); cur = nxt

        cost = sum(self.G[u,v] + self.cong[u,v] for u,v in zip(path[:-1], path[1:]))
        return path, cost

    def next_node(self, cur, vis):
        nbrs = [n for n in range(self.N) if self.G[cur,n] < float('inf') and n not in vis]
        if not nbrs: return None

        probs = []
        for n in nbrs:
            c = self.G[cur,n] + self.cong[cur,n]
            probs.append((self.phero[cur,n]**self.a) * ((1/(c+1e-10))**self.b))

        total = sum(probs)
        if total == 0: return random.choice(nbrs)
        return random.choices(nbrs, weights=[p/total for p in probs])[0]

    def upd_phero(self, paths):
        self.phero *= (1 - self.rho)
        for p, c in paths:
            if p:
                d = self.q / c
                for u, v in zip(p[:-1], p[1:]):
                    self.phero[u,v] += d; self.phero[v,u] += d

if __name__ == "__main__":
    inf = float('inf')
    G = np.array([
        [inf, 10, inf, 25, inf, 30],
        [10, inf, 15, inf, inf, inf],
        [inf, 15, inf, 12, 20, inf],
        [25, inf, 12, inf, 18, inf],
        [inf, inf, 20, 18, inf, 10],
        [30, inf, inf, inf, 10, inf]
    ])

    vis = Vis(G, 6)
    aco = ACO(G, vis, ants=20, itr=100, a=1.0, b=3.0, rho=0.3, q=100)

    print("Optimizing Path 0 -> 4...")
    path, cost = aco.run(0, 4)
    print(f"Result: {path} Cost: {cost:.2f}")
```

Program 5

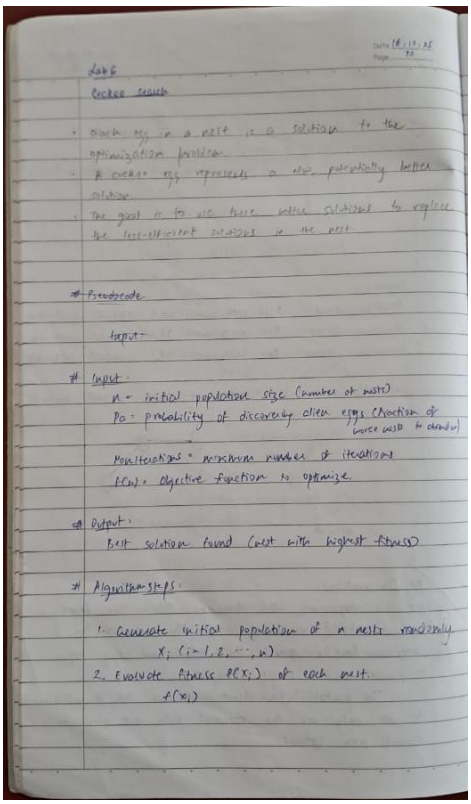
Cuckoo Search (CS):

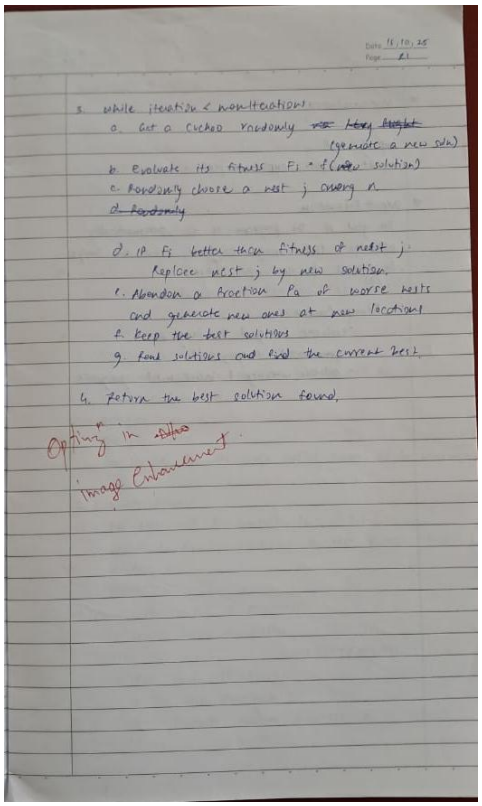
Cuckoo Search (CS) is a nature-inspired optimization algorithm based on the brood parasitism of some cuckoo species. This behaviour involves laying eggs in the nests of other birds, leading to the optimization of survival strategies. CS uses Lévy flights to generate new solutions, promoting global search 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.

Implementation Steps:

1. Define the Problem: Create a mathematical function to optimize.
2. Initialize Parameters: Set the number of nests, the probability of discovery, and the number of iterations.
3. Initialize Population: Generate an initial population of nests with random positions.
4. Evaluate Fitness: Evaluate the fitness of each nest based on the optimization function.
5. Generate New Solutions: Create new solutions via Lévy flights.
6. Abandon Worst Nests: Abandon a fraction of the worst nests and replace them with new random positions.
7. Iterate: Repeat the evaluation, updating, and replacement process for a fixed number of iterations or until convergence criteria are met.
8. Output the Best Solution: Track and output the best solution found during the iterations.

Algorithm:





Code:

```
import cv2, numpy as np, random, math
import matplotlib.pyplot as plt

N, PA, ITR = 10, 0.25, 30
A_R, B_R = (1.0, 3.0), (0, 80)
IN_IMG, OUT_IMG = 'input.jpg', 'enhanced_output.jpg'

img = cv2.imread(IN_IMG)
if img is None: exit(f"Error: {IN_IMG} not found")

def get_fit(im):
    gray = cv2.cvtColor(im, cv2.COLOR_BGR2GRAY)
    h = cv2.calcHist([gray], [0], None, [256], [0, 256]).flatten()
    h /= (h.sum() + 1e-7)
    return -np.sum(h * np.log2(h + 1e-7))

def levy(lam=1.5):
    num = math.gamma(1+lam) * math.sin(math.pi*lam/2)
    den = math.gamma((1+lam)/2) * lam * 2**((lam-1)/2)
    sigma = (num/den)**(1/lam)
    return (np.random.randn() * sigma) / (abs(np.random.randn())**(1/lam))

pop = np.array([[random.uniform(*A_R), random.uniform(*B_R)] for _ in range(N)])
fits = np.zeros(N)

for i in range(N):
    fits[i] = get_fit(cv2.convertScaleAbs(img, alpha=pop[i][0], beta=pop[i][1]))

for t in range(ITR):
    for i in range(N):
        step = levy() * np.random.randn(2)
        cand = np.clip(pop[i] + step, [A_R[0], B_R[0]], [A_R[1], B_R[1]])

        cand_img = cv2.convertScaleAbs(img, alpha=cand[0], beta=cand[1])
        cand_fit = get_fit(cand_img)

        if cand_fit > fits[i]: pop[i], fits[i] = cand, cand_fit

    mask = np.random.rand(N) < PA
    if mask.any():
        pop[mask] = [[random.uniform(*A_R), random.uniform(*B_R)] for _ in range(mask.sum())]
        for i in np.where(mask)[0]:
            fits[i] = get_fit(cv2.convertScaleAbs(img, alpha=pop[i][0], beta=pop[i][1]))

best = pop[np.argmax(fits)]
final = cv2.convertScaleAbs(img, alpha=best[0], beta=best[1])

k = np.array([[0,-1,0], [-1,5,-1], [0,-1,0]])
final = cv2.filter2D(final, -1, k)

cv2.imwrite(OUT_IMG, final)
print(f"Best: alpha={best[0]:.2f}, beta={best[1]:.2f}. Saved to {OUT_IMG}")

plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1); plt.title('Original'); plt.imshow(cv2.cvtColor(img, cv2.COLOR_BGR2RGB))
plt.subplot(1, 2, 2); plt.title('CS Enhanced'); plt.imshow(cv2.cvtColor(final, cv2.COLOR_BGR2RGB))
plt.savefig('comparison_plot.png')
```

Program 6

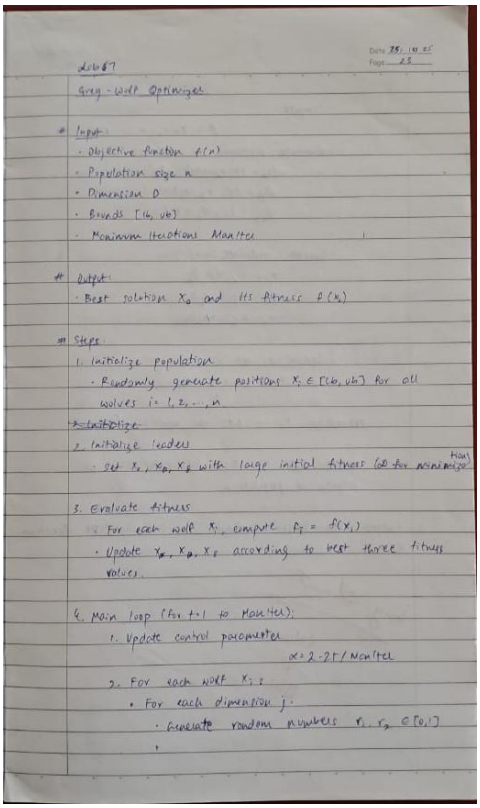
Grey Wolf Optimizer (GWO):

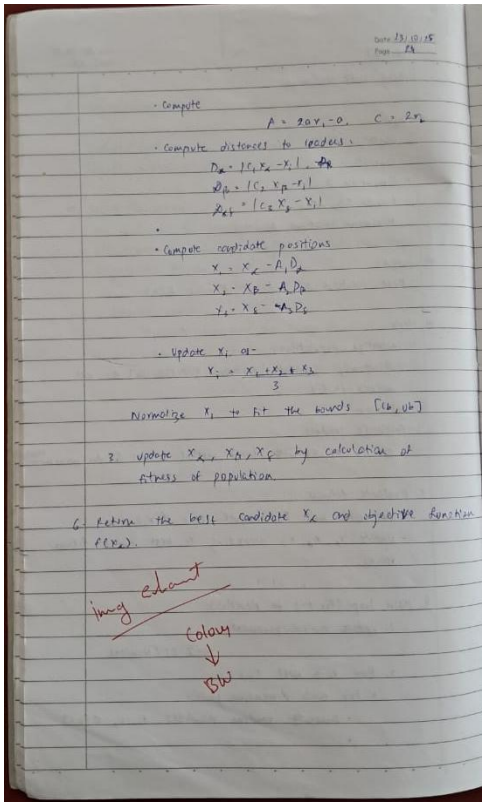
The Grey Wolf Optimizer (GWO) algorithm is a swarm intelligence algorithm inspired by the social hierarchy and hunting behaviour of grey wolves. It mimics the leadership structure of alpha, beta, delta, and omega wolves and their collaborative hunting strategies. The GWO algorithm uses these social hierarchies 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.

Implementation Steps:

1. Define the Problem: Create a mathematical function to optimize.
2. Initialize Parameters: Set the number of wolves and the number of iterations.
3. Initialize Population: Generate an initial population of wolves with random positions.
4. Evaluate Fitness: Evaluate the fitness of each wolf based on the optimization function.
5. Update Positions: Update the positions of the wolves based on the positions of alpha, beta, and delta wolves.
6. Iterate: Repeat the evaluation and position updating process for a fixed number of iterations or until convergence criteria are met.
7. Output the Best Solution: Track and output the best solution found during the iterations.

Algorithm:





Code:

```
import cv2, numpy as np, time
from skimage.measure import shannon_entropy as ent

IN_FILE, OUT_FILE = "input_image.png", "output_image.png"

def run_gwo(img, N=20, itr=50):
    img_f = img.astype(float)
    dim = 3
    pop = np.random.rand(N, dim)
    pop /= pop.sum(axis=1, keepdims=True)

    A, B, D = np.zeros(dim), np.zeros(dim), np.zeros(dim)
    As, Bs, Ds = -np.inf, -np.inf, -np.inf

    print("Starting GWO...")
    for t in range(itr):
        for i in range(N):
            # Fast weighted sum using dot product
            gray = np.clip(img_f @ pop[i], 0, 255).astype(np.uint8)
            fit = ent(gray)

            if fit > As: As, A = fit, pop[i].copy()
            elif fit > Bs: Bs, B = fit, pop[i].copy()
            elif fit > Ds: Ds, D = fit, pop[i].copy()

        a = 2 - t * (2 / itr)
        for i in range(N):
            X_new = []
            for Leader in [A, B, D]:
                r1, r2 = np.random.rand(), np.random.rand()
                A1, C1 = 2 * a * r1 - a, 2 * r2
                D_lead = np.abs(C1 * Leader - pop[i])
                X_new.append(Leader - A1 * D_lead)

            pop[i] = sum(X_new) / 3

        pop = np.clip(pop, 0, 1)
        pop /= pop.sum(axis=1, keepdims=True)

        if (t + 1) % 10 == 0: print(f"Iter {t+1}/{itr} Best Entropy: {As:.4f}")

    return A

if __name__ == "__main__":
    src = cv2.imread(IN_FILE)
    if src is None: exit(f"Err: {IN_FILE} missing")

    w = run_gwo(src)
    res = np.clip(src.astype(float) @ w, 0, 255).astype(np.uint8)

    cv2.imwrite(OUT_FILE, res)
    print(f"Saved {OUT_FILE}\nWeights [B,G,R]: {w}\nFinal Entropy: {ent(res):.4f}")
```

Program 7

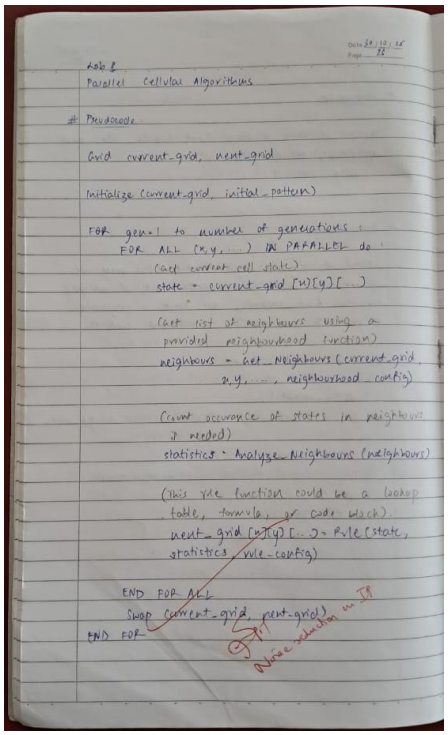
Parallel Cellular Algorithms and Programs:

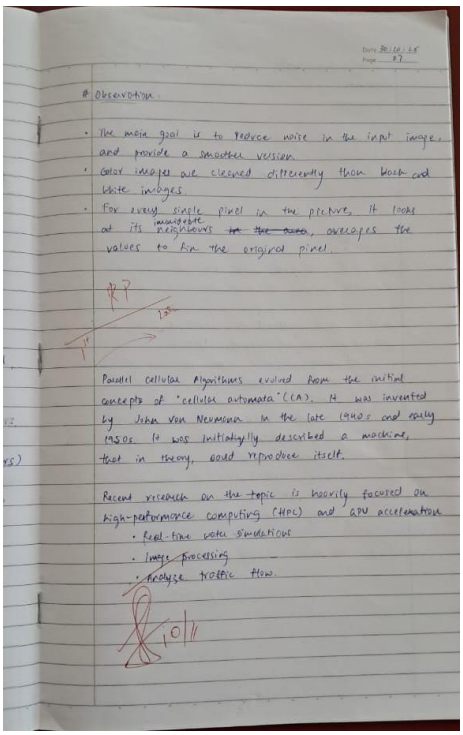
Parallel Cellular Algorithms are inspired by the functioning of biological cells that operate in a highly parallel and distributed manner. These algorithms leverage the principles of cellular automata and parallel computing to solve complex optimization problems efficiently. Each cell represents a potential solution and interacts with its neighbours 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.

Implementation Steps:

1. Define the Problem: Create a mathematical function to optimize.
2. Initialize Parameters: Set the number of cells, grid size, neighbourhood structure, and number of iterations.
3. Initialize Population: Generate an initial population of cells with random positions in the solution space.
4. Evaluate Fitness: Evaluate the fitness of each cell based on the optimization function.
5. Update States: Update the state of each cell based on the states of its neighbouring cells and predefined update rules.
6. Iterate: Repeat the evaluation and state updating process for a fixed number of iterations or until convergence criteria are met.
7. Output the Best Solution: Track and output the best solution found during the iterations.

Algorithm:





Code:

```
import cv2, numpy as np, os
import matplotlib.pyplot as plt

IN, OUT, P, K, ITR = "input_image.png", "output_denoised.png", 0.05, 3, 2

if not os.path.exists(IN): exit(f"Error: {IN} missing")
img = cv2.imread(IN)

def add_noise(im, amt):
    res = im.copy()
    h, w = res.shape[:2]
    n = int(amt * h * w / 2)
    for v in [255, 0]: # Salt then Pepper
        y, x = np.random.randint(0, h, n), np.random.randint(0, w, n)
        res[y, x] = v
    return res

noisy = add_noise(img, P)
clean = noisy.copy()

print(f"Applying Cellular Median Filter ({K}x{K}) for {ITR} iters...")
for i in range(ITR):
    # Cellular automata step: update state based on neighborhood median
    clean = cv2.medianBlur(clean, K)
    print(f"Iteration {i+1} complete")

cv2.imwrite("noisy_sample.png", noisy)
cv2.imwrite(OUT, clean)
print(f"Saved {OUT}")

fig, ax = plt.subplots(1, 3, figsize=(12, 4))
titles = [f"Original", f"Noisy ({P:.0%})", f"Restored ({ITR} iters)"]
for i, im in enumerate([img, noisy, clean]):
    ax[i].imshow(cv2.cvtColor(im, cv2.COLOR_BGR2RGB))
    ax[i].set_title(titles[i]); ax[i].axis('off')
plt.show()
```


Acknowledgement

I sincerely thank **Swathi Ma'am** for their guidance in **Bio-Inspired Systems** course and the **Computer Science and Engineering Department** for providing this opportunity to learn concepts that are directly associated with machine learning. I greatly appreciate the support.