VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB RECORD

Bio Inspired Systems (23CS5BSBIS)

Submitted by

PRANAV GAJANAN KAMATE (1BM23CS241)

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 **PRANAV GAJANAN KAMATE (1BM22CS241),** 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. Joythi S Nayak Professor & HOD Department of CSE, BMSCE

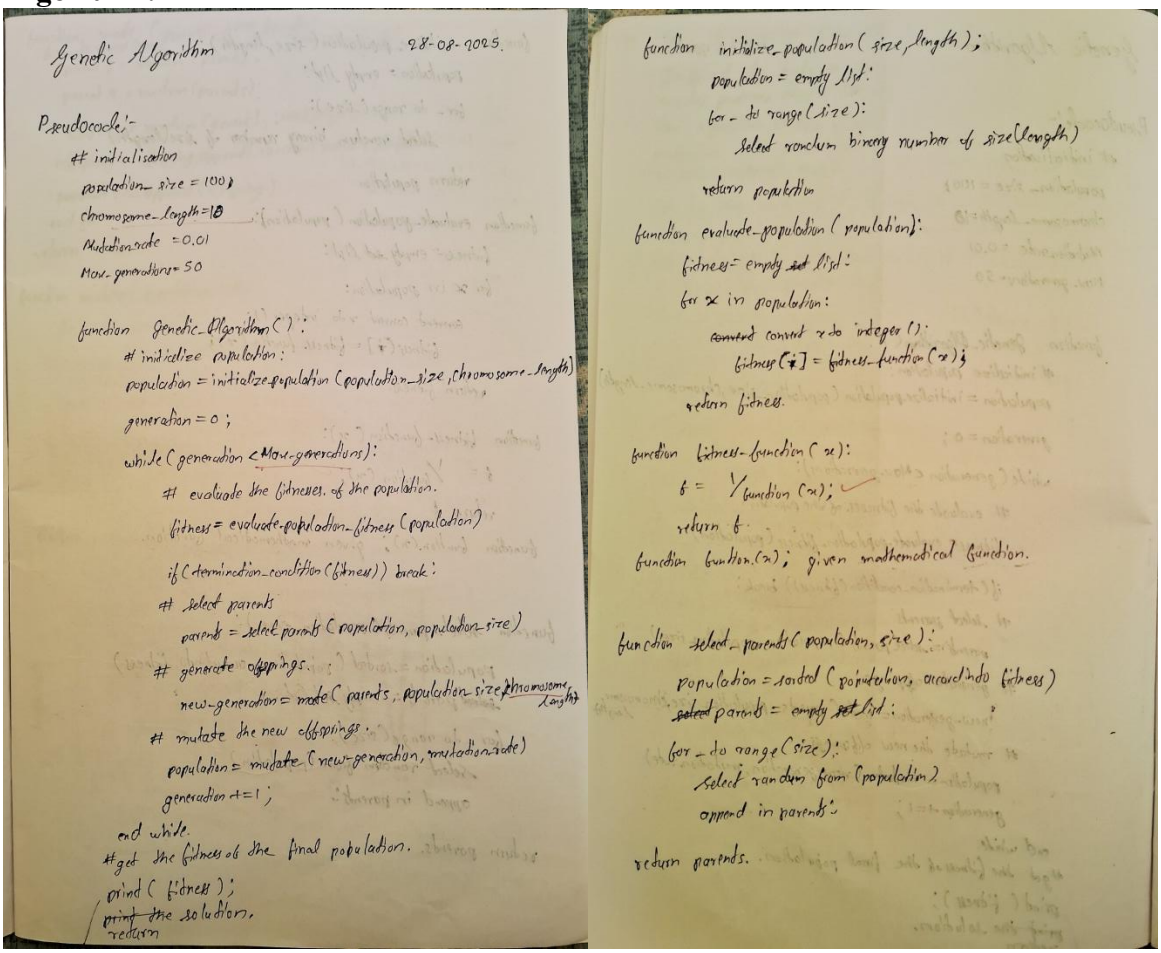
Index

Github Link:
https://github.com/PranavKamate-cs/1BM23CS241_BIO_SYS/tree/main

Program 1

Problem statemen: Use the Genetic Algorithm to analyze astronomical data (Light curve fit).

Algorithm:



Genetic Algorithm 28-08-2025.

Pseudocode:-

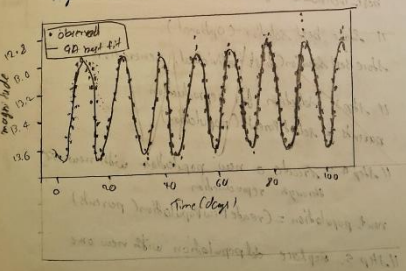
```
# initialisation
population_size = 100
chromosome_length = 10
Mutation_rate = 0.01
Max_generations = 50

function Genetic_Algorithm():
    # initialize population:
    population = initialize_population(population_size, chromosome_length)
    generation = 0;
    while(generation < Max_generations):
        # evaluate the fitnesses of the population.
        fitness = evaluate_population_fitness(population)
        if(termination_condition(fitness)) break;
        # select parents
        parents = select_parents(population, population_size)
        # generate offsprings.
        new_generation = make(parents, population_size, chromosome_length)
        # mutate the new offsprings.
        population = mutate(new_generation, mutation_rate)
        generation += 1;
    end while
    #get the fitness of the final population.
    print(fitness);
    print the solution.
    return
```

```
function initialize_population(size, length);
    population = empty list;
    for _ do range(size):
        Select random binary number of size(length)
    return population

function evaluate_population(population):
    fitness = empty list;
    for x in population:
        convert convert x to integer();
        fitness[x] = fitness_function(x);
    return fitness.

function fitness_function(x):
    f = 1/function(x);
    return f
function function(x); given mathematical function.

function select_parents(population, size):
    population = sorted(population, according to fitness)
    parents = empty list;
    for _ do range(size):
        select random from (population)
        append in parents;
return parents.
```

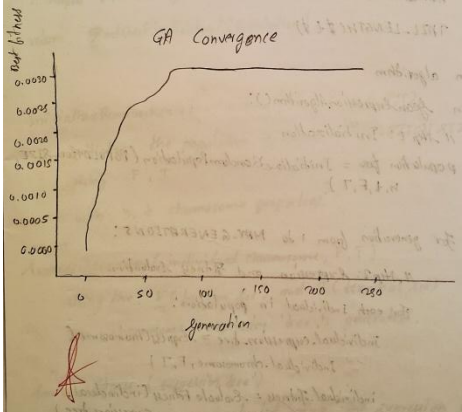
```
function mate (parents, size, length):
    new = empty list.
    for _ do range (size):
        parent 1 = random (parents);
        parent 2 = random (parents);
        offspring = random (parent 1 : 50) length/2 );
        offspring = offspring.append( random (parent 2 : length/2)
        ~~population~~ new.append (offspring)
    end for
    return new.

function mutate (population, rate):
    for _ in population:
        mutate the individual at the rate)
    return population:
```

- Write Generic Version of GA
- Execute WRT - Astronomical data

Output for Light Curve Fit (Astronomical data).




```
Generic version of GA

// Global parameters
Define POPULATION_SIZE
Define MAX_GENERATIONS
Define CROSSOVER_RATE
Define MUTATION_RATE

// Main algorithm

function GeneticAlgorithm():
    // Step1: Initialization
    population = InitializeRandomPopulation (POPULATION_SIZE)

    // Main loop for evolution
    for generation from 1 to MAX_GENERATIONS:
        // Step 2: Fitness evaluation
        EvaluateFitness (population)

        // Find the best individual in the current population
        best_individual = FindBestIndividual (population)

        // Store best solution (optional)
        StoreBestSolution (best_individual, generation)

        // Step3: Selection for reproduction
        parents = SelectParents (population)

        // Step 4: Create a new population with ~~new one~~ through reproduction
        next_population = CreateNewPopulation (parents)

        // Step 5: Replace old population with new one
        population = next_population.

        // Step 6: Termination check.
        if Termination condition met:
            break.

    // Output final best solution
    Return FindBestIndividual (population).
```



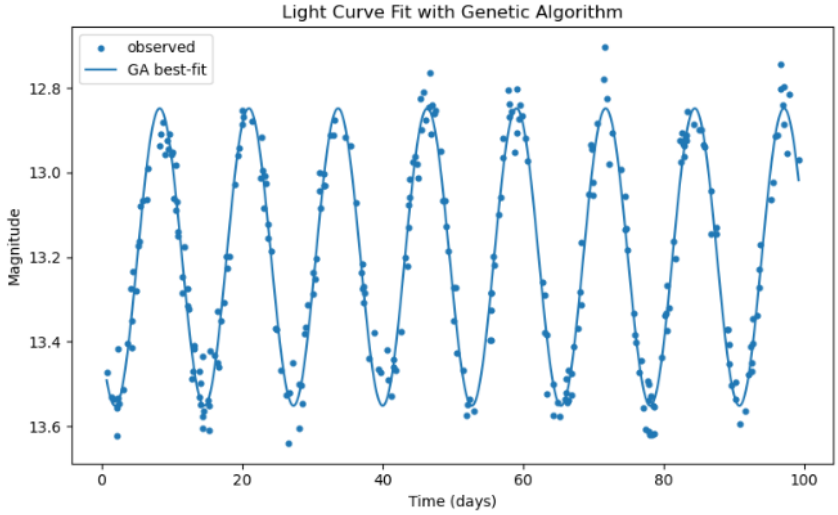
Code:

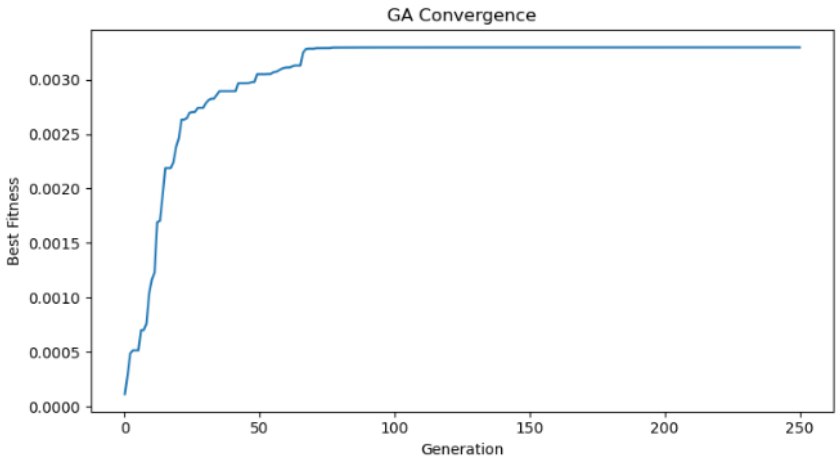
```
import numpy as np
import matplotlib.pyplot as plt

rng = np.random.default_rng(42)

def light_curve_model(t, A, P, phi, m0):
    return m0 + A * np.sin(2 * np.pi * t / P + phi)

def chi2(params, t, y, yerr):
    A, P, phi, m0 = params
    y_model = light_curve_model(t, A, P, phi, m0)
    r = (y - y_model) / yerr
    return float(np.sum(r * r))

def fitness(params, t, y, yerr):
    c2 = chi2(params, t, y, yerr)
    return 1.0 / (1.0 + c2)

def run_ga(t, y, yerr, bounds, cfg):
    pop_size, generations, elite_frac, tournament_k, mutation_rate,
mutation_scale, crossover_rate, rng = cfg
    def clip_to_bounds(ind):
        for i, (lo, hi) in enumerate(bounds):
            ind[i] = np.clip(ind[i], lo, hi)
        return ind

    def random_individual():
        return np.array([rng.uniform(lo, hi) for lo, hi in bounds],
dtype=float)

    def tournament_select(pop, fitnesses, k):
        idx = rng.integers(0, len(pop), size=k)
        best = idx[np.argmax(fitnesses[idx])]
        return pop[best].copy()

    def blend_crossover(p1, p2):
        alpha = rng.uniform(0.0, 1.0, size=p1.shape)
        c1 = alpha * p1 + (1 - alpha) * p2
        c2 = alpha * p2 + (1 - alpha) * p1
        return c1, c2

    def mutate(ind):
        ind = ind.copy()
        for i, (lo, hi) in enumerate(bounds):
            if rng.random() < mutation_rate:
                span = (hi - lo)
```

```
                ind[i] += rng.normal(0.0, mutation_scale * span)
        return clip_to_bounds(ind)

    pop = np.array([random_individual() for _ in range(pop_size)])
    fit = np.array([fitness(ind, t, y, yerr) for ind in pop])
    best_curve = [float(np.max(fit))]
    n_elite = max(1, int(elite_frac * pop_size))

    for _ in range(generations):
        elite_idx = np.argsort(fit)[-n_elite:]
        elites = pop[elite_idx].copy()
        next_pop = []
        while len(next_pop) < pop_size - n_elite:
            p1 = tournament_select(pop, fit, tournament_k)
            p2 = tournament_select(pop, fit, tournament_k)
            if rng.random() < crossover_rate:
                c1, c2 = blend_crossover(p1, p2)
            else:
                c1, c2 = p1.copy(), p2.copy()
            c1 = mutate(c1)
            c2 = mutate(c2)
            next_pop.append(c1)
            if len(next_pop) < pop_size - n_elite:
                next_pop.append(c2)
        pop = np.vstack([elites, np.array(next_pop)])
        fit = np.array([fitness(ind, t, y, yerr) for ind in pop])
        best_curve.append(float(np.max(fit)))

    best_idx = int(np.argmax(fit))
    best_params = pop[best_idx].copy()
    return best_params, best_curve

if __name__ == "__main__":
    # Generate synthetic dataset
    n_points = 300
    t = np.sort(rng.uniform(0.0, 100.0, n_points))
    A_true, P_true, phi_true, m0_true = 0.35, 12.7, 0.6, 13.2
    noise_sigma = 0.05
    y = m0_true + A_true * np.sin(2 * np.pi * t / P_true +
phi_true) + rng.normal(0, noise_sigma, size=n_points)
    yerr = np.full_like(y, noise_sigma)

    # Bounds: (A, P, phi, m0)
    bounds = [(0.0, 1.0), (5.0, 20.0), (0.0, 2 * np.pi), (12.0,
14.0)]
    cfg = (200, 250, 0.05, 3, 0.15, 0.07, 0.9,
np.random.default_rng(2025))
```

```
    best_params, best_curve = run_ga(t, y, yerr, bounds, cfg)
    A_hat, P_hat, phi_hat, m0_hat = best_params

    # Plot results
    plt.figure(figsize=(8, 5))
    plt.scatter(t, y, s=12, label="observed")
    tt = np.linspace(t.min(), t.max(), 1000)
    plt.plot(tt, m0_hat + A_hat * np.sin(2 * np.pi * tt / P_hat +
phi_hat), label="GA best-fit")
    plt.gca().invert_yaxis()
    plt.xlabel("Time (days)")
    plt.ylabel("Magnitude")
    plt.title("Light Curve Fit with Genetic Algorithm")
    plt.legend()
    plt.tight_layout()
    plt.show()

    # Convergence
    plt.figure(figsize=(8, 4.5))
    plt.plot(best_curve)
    plt.xlabel("Generation")
    plt.ylabel("Best Fitness")
    plt.title("GA Convergence")
    plt.tight_layout()
    plt.show()
```

Output:

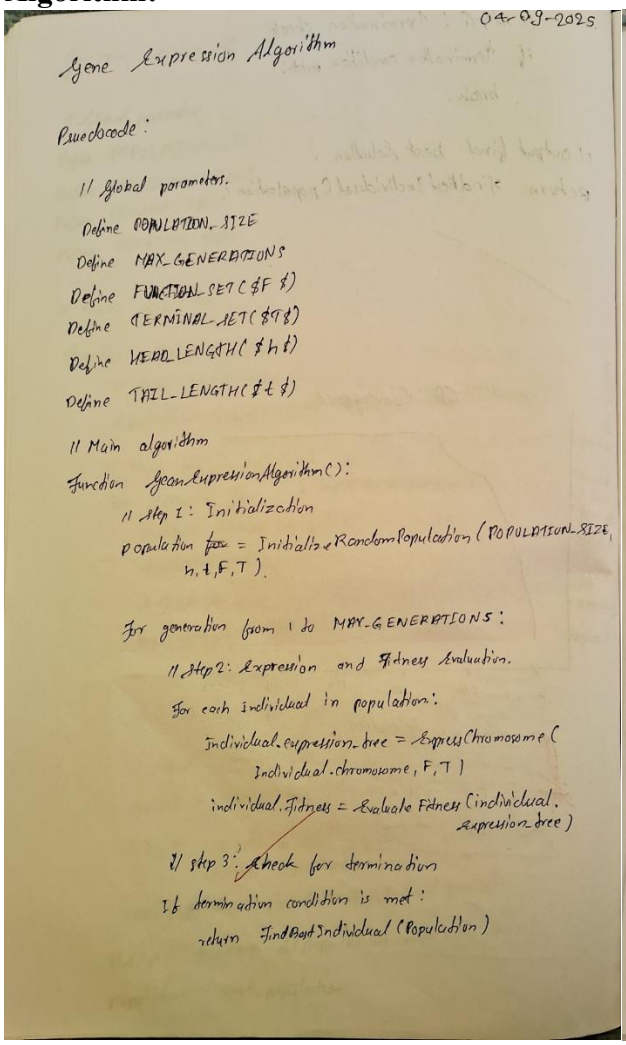

Light Curve Fit with Genetic Algorithm
observed
GA best-fit
Magnitude
Time (days)
12.8
13.0
13.2
13.4
13.6
0
20
40
60
80
100


GA Convergence
Best Fitness
Generation
0.0030
0.0025
0.0020
0.0015
0.0010
0.0005
0.0000
0
50
100
150
200
250

Program 2

Problem statemen: Use the Gene Expression Algorithm to analyze astronomical data (Light curve fit). And compare with genetic algorithm.

Algorithm:

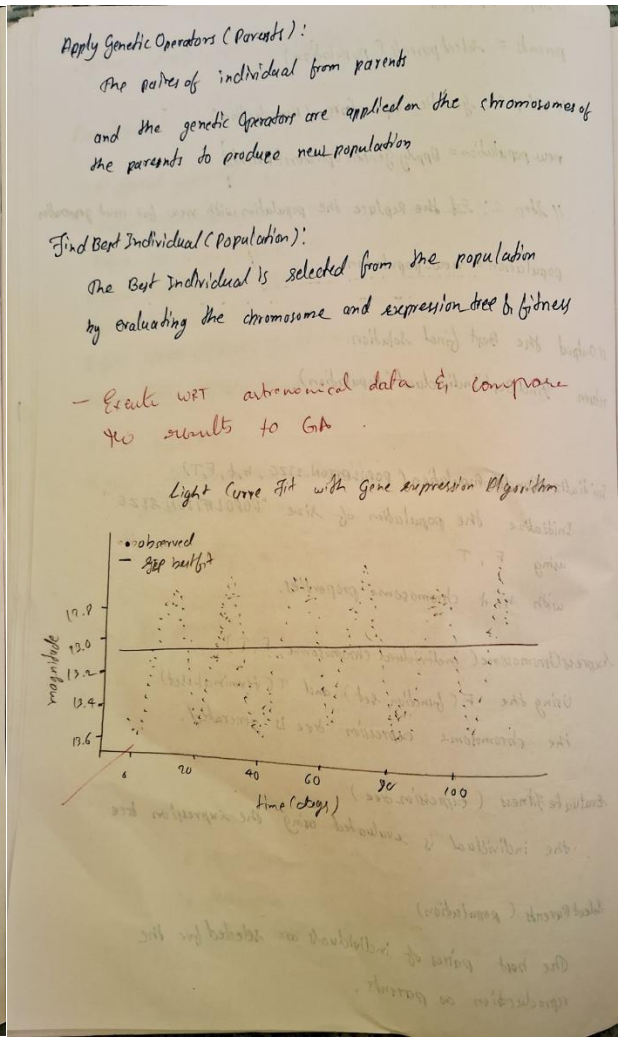


04-09-2025

Gene Expression Algorithm

Pseudocode:

```
// Global parameters.
Define POPULATION_SIZE
Define MAX_GENERATIONS
Define FUNCTION_SET ($F $)
Define TERMINAL_SET ($T $)
Define HEAD_LENGTH ( $h $)
Define TAIL_LENGTH ( $t $)

// Main algorithm
Function GeneExpressionAlgorithm():
    // Step 1: Initialization
    population = InitializeRandomPopulation (POPULATION_SIZE,
        h, t, F, T ).

    For generation from 1 to MAX_GENERATIONS:
        // Step 2: Expression and fitness Evaluation.
        For each individual in population:
            Individual.expression_tree = ExpressChromosome (
                Individual.chromosome, F, T )
            individual.fitness = EvaluateFitness (individual.
                                                  expression_tree)
        // Step 3: Check for termination
        If termination condition is met:
            return FindBestIndividual (Population)
```



Apply Genetic Operators (Parents):

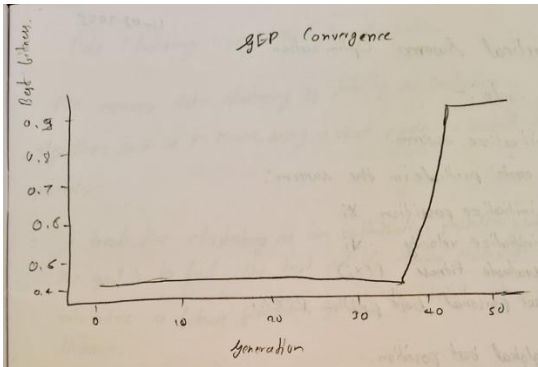
The pairs of individual from parents and the genetic operators are applied on the chromosomes of the parents to produce new population

FindBestIndividual (Population):

The Best Individual is selected from the population by evaluating the chromosome and expression tree & fitness

- Create WRT astronomical data & compare the results to GA

Light Curve fit with Gene expression Algorithm



- Based on the plot of the light curve and the convergence of each GA & GEP.
  The Genetic algorithm is best suited for the light curve optimization.
- GA produces a physically meaningful and accurate model for light curve. And shows a clear improvement over generations.
- GEP failed to find a meaningful fit and the convergence shows ~~that~~ unusual and and ineffective optimization process.

```
// Step 4: Selection
parents = selectParents(population)

// Step 5: Genetic Operators (Reproduction)
new_population = ApplyGeneticOperators(parents)

// Step 6: ~~Set the~~ Replace the population with new for next generation
population = new_population

// Output the best final solution
return FindBestIndividual(Population).


InitializeRandomPopulation(POPULATION_SIZE, h, t, F, T)
    Initialize the population of size "POPULATION_SIZE"
    using F, T
    with h, t chromosome properties.

ExpressChromosome(~~individual~~ chromosome, F, T)
    Using the F(function_set) and T(terminal set)
    the chromosome expression tree is generated.

EvaluateFitness(expression_tree)
    the individual is evaluated using the expression tree

SelectParents(population)
    The best pairs of individuals are selected for the
    reproduction as parents.
```

Code:

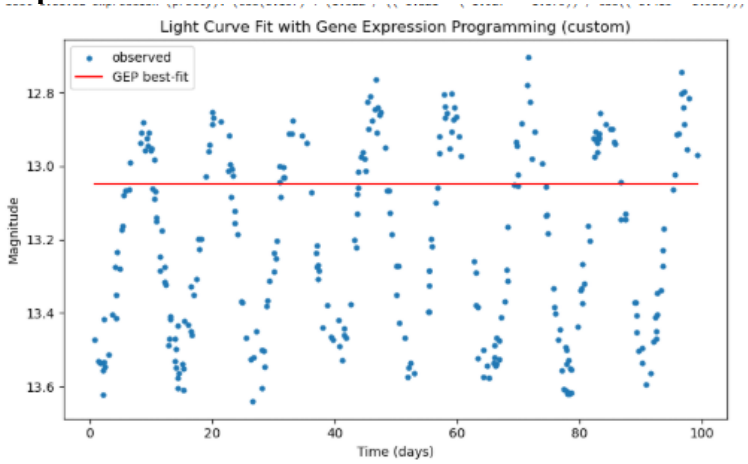
```
import numpy as np
import matplotlib.pyplot as plt

rng = np.random.default_rng(42)

# ================================
# Dataset (same as GA example)
# ================================
n_points = 300
t = np.sort(rng.uniform(0.0, 100.0, n_points))
A_true, P_true, phi_true, m0_true = 0.35, 12.7, 0.6, 13.2
noise_sigma = 0.05

y = m0_true + A_true * np.sin(2 * np.pi * t / P_true + phi_true) \
    + rng.normal(0, noise_sigma, size=n_points)
yerr = np.full_like(y, noise_sigma)

# ================================
# GEP Building Blocks
# ================================
functions = {
    "add": (2, lambda a, b: a + b),
    "sub": (2, lambda a, b: a - b),
    "mul": (2, lambda a, b: a * b),
    # safe division
    "div": (2, lambda a, b: np.where(np.abs(b) > 1e-8, a / b,
1.0)),
    "sin": (1, np.sin),
    "cos": (1, np.cos)
}
terminals = ["t", "const"]

def random_expression(max_depth=3):
    """ Recursively build a random expression tree. """
    if max_depth == 0 or rng.random() < 0.3:
        if rng.random() < 0.5:
            return ("t",)  # variable
        else:
            return ("const", rng.uniform(-2, 2))  # random constant
    func = rng.choice(list(functions.keys()))
    arity, _ = functions[func]
    return (func,) + tuple(random_expression(max_depth - 1) for _
in range(arity))

def evaluate(expr, t_val):
    """ Evaluate expression tree for given t values (vectorized).
"""
```

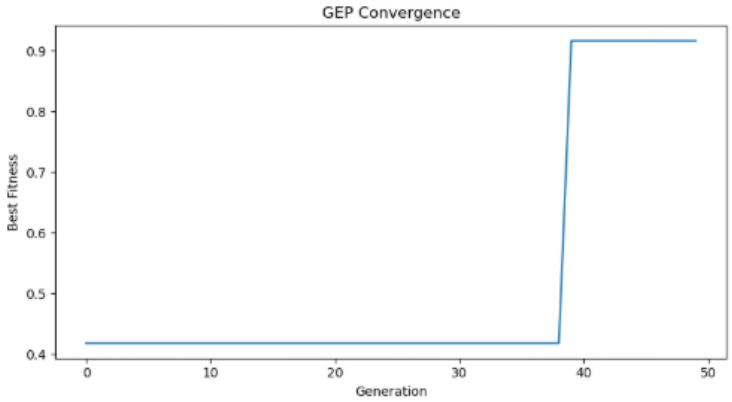
```
    op = expr[0]
    if op == "t":
        return t_val
    elif op == "const":
        return np.full_like(t_val, expr[1], dtype=float)
    else:
        arity, func = functions[op]
        args = [evaluate(arg, t_val) for arg in expr[1:1+arity]]
        return func(*args)

def fitness(expr, t, y):
    """ Mean squared error fitness. """
    try:
        y_pred = evaluate(expr, t)
        mse = np.mean((y - y_pred) ** 2)
        return 1 / (1 + mse)  # higher is better
    except Exception:
        return 1e-6

def mutate(expr, prob=0.1, max_depth=3):
    """ Randomly mutate an expression. """
    if rng.random() < prob:
        return random_expression(max_depth)
    op = expr[0]
    if op in ("t", "const"):
        return expr
    arity, _ = functions[op]
    return (op,) + tuple(mutate(arg, prob, max_depth) for arg in
expr[1:1+arity])

def crossover(e1, e2, prob=0.7):
    """ Random subtree crossover. """
    if rng.random() > prob:
        return e1
    if e1[0] in ("t", "const"):
        return e2
    arity, _ = functions[e1[0]]
    i = rng.integers(arity)
    new_args = list(e1[1:])
    new_args[i] = crossover(new_args[i], e2, prob)
    return (e1[0],) + tuple(new_args)

# Pretty printer
def expr_to_str(expr):
    """ Convert an expression tree to a human-readable string. """
    op = expr[0]
    if op == "t":
        return "t"
```

```
    elif op == "const":
        return f"{expr[1]:.3f}"
    elif op in ("add", "sub", "mul", "div"):
        a, b = expr[1], expr[2]
        a_str, b_str = expr_to_str(a), expr_to_str(b)
        if op == "add":
            return f"({a_str} + {b_str})"
        elif op == "sub":
            return f"({a_str} - {b_str})"
        elif op == "mul":
            return f"({a_str} * {b_str})"
        elif op == "div":
            return f"({a_str} / {b_str})"
    elif op in ("sin", "cos"):
        return f"{op}({expr_to_str(expr[1])})"
    else:
        return str(expr)  # fallback

# ================================
# Run GEP
# ================================
pop_size = 100
generations = 50

# Initialize population
population = [random_expression(max_depth=4) for _ in
range(pop_size)]
fitnesses = [fitness(ind, t, y) for ind in population]

best_curve = []

for gen in range(generations):
    # Elitism
    elite_idx = np.argmax(fitnesses)
    elite = population[elite_idx]

    next_pop = [elite]  # carry best

    while len(next_pop) < pop_size:
        # Tournament selection for parent1
        idx1 = rng.choice(len(population), size=3, replace=False)
        parent1 = max((population[i] for i in idx1), key=lambda
ind: fitness(ind, t, y))

        # Tournament selection for parent2
        idx2 = rng.choice(len(population), size=3, replace=False)
        parent2 = max((population[i] for i in idx2), key=lambda
ind: fitness(ind, t, y))
```

```
            # Crossover + mutation
            child = crossover(parent1, parent2)
            child = mutate(child, prob=0.2)
            next_pop.append(child)

        population = next_pop
        fitnesses = [fitness(ind, t, y) for ind in population]
        best_fit = max(fitnesses)
        best_curve.append(best_fit)

# Best solution
best_idx = np.argmax(fitnesses)
best_expr = population[best_idx]
print("Best evolved expression (raw):", best_expr)
print("Best evolved expression (pretty):", expr_to_str(best_expr))

# ================================
# Plot results
# ================================
tt = np.linspace(t.min(), t.max(), 1000)
y_pred = evaluate(best_expr, tt)

plt.figure(figsize=(8, 5))
plt.scatter(t, y, s=12, label="observed")
plt.plot(tt, y_pred, label="GEP best-fit", color="red")
plt.gca().invert_yaxis()
plt.xlabel("Time (days)")
plt.ylabel("Magnitude")
plt.title("Light Curve Fit with Gene Expression Programming
(custom)")
plt.legend()
plt.tight_layout()
plt.show()

# Convergence
plt.figure(figsize=(8, 4.5))
plt.plot(best_curve)
plt.xlabel("Generation")
plt.ylabel("Best Fitness")
plt.title("GEP Convergence")
plt.tight_layout()
plt.show()
```

Output:

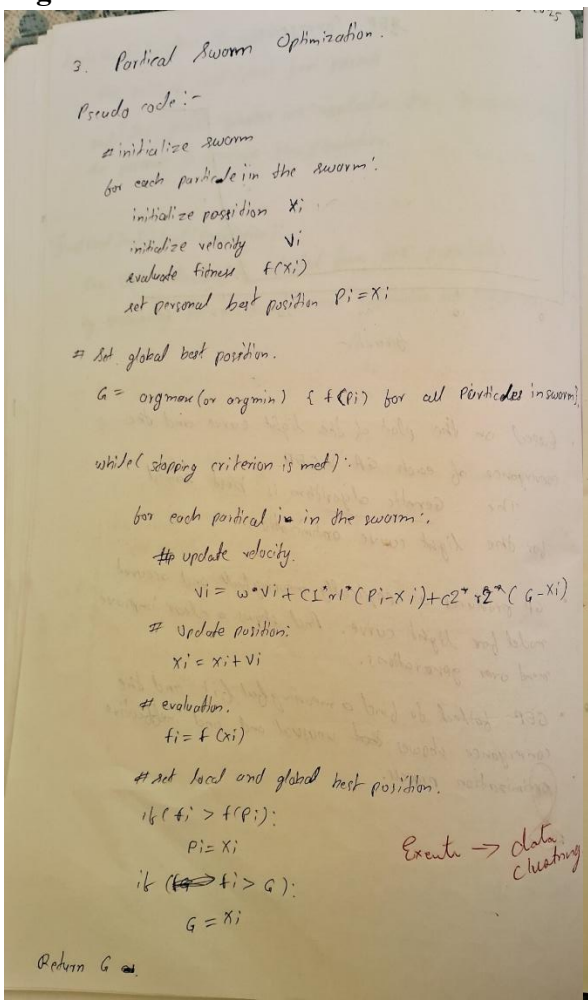

Light Curve Fit with Gene Expression Programming (custom)
observed
GEP best-fit
Magnitude
Time (days)
12.8
13.0
13.2
13.4
13.6
0
20
40
60
80
100


GEP Convergence
Best Fitness
Generation
0.4
0.5
0.6
0.7
0.8
0.9
0
10
20
30
40
50

Program 3

Problem statemen: Use Particle Swarm Optimization Algorithm to solve the Data Clustering problem.

Algorithm:



3. Partical Swarm Optimization.

Pseudo code :-

# initialize sworm
for each particle in the sworm:
initialize position Xi
initialize velocity Vi
evaluate fitness f(Xi)
set personal best position Pi = Xi

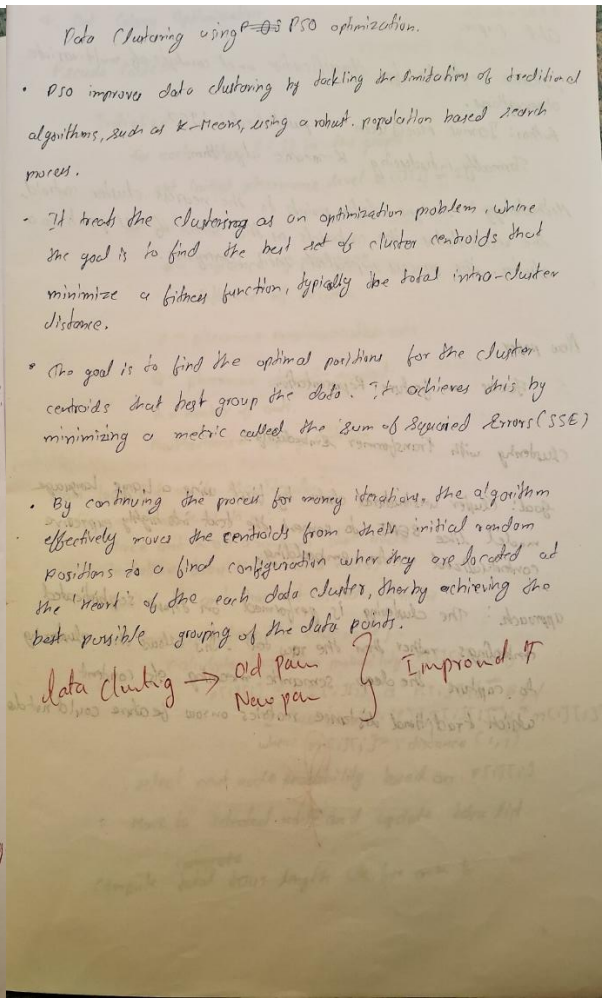
# Set global best position.
G = argmax(or argmin) { f(Pi) for all Particles in sworm}

while(stopping criterion is met):

for each paritical in the sworm:
update velocity.
Vi = w*Vi + c1*r1*(Pi-Xi) + c2*r2*(G-Xi)
# Update position:
Xi = Xi + Vi
# evaluation.
fi = f(Xi)
# set local and global best position.
if(fi > f(Pi)):
Pi = Xi
if (fi > G):
G = Xi

Return G

Execute → data clustering



Data Clustering using PSO optimization.

- PSO improves data clustering by tackling the limitations of traditional algorithms, such as K-Means, using a robust, population based search process.
- It treats the clustering as an optimization problem, where the goal is to find the best set of cluster centroids that minimize a fitness function, typically the total intra-cluster distance.
- The goal is to find the optimal positions for the cluster centroids that best group the data. It achieves this by minimizing a metric called the sum of Squared Errors (SSE)
- By continuing the process for many iterations, the algorithm effectively moves the centroids from their initial random positions to a final configuration where they are located at the 'heart' of the each data cluster, thereby achieving the best possible grouping of the data points.

data clustering → Old pair, New pair } Improved ✓

Old Paper.

Some methods for classification and analysis of multivariate observations.

Author: James MacQueen published: 1967.

Formally introducing K-means algorithm.

Methodology: Assigns data points to the nearest cluster centroid, the recalculates the centroids as the mean of all points in a cluster. the process repeated until convergance.

New paper

Focus on feature Representation

Clustering with Transformer Embeddings.

goal: Cluster unstructured data by first using a Large Language model like BERT to convert the text into highly expressive contextualized vector embedding.

approach: The clustering is performed on these sophisticated embeddings rather than the raw text. This allows the clustering to capture the deep semantic meaning of content. which traditional distance metrics on raw feature could not do

Code:

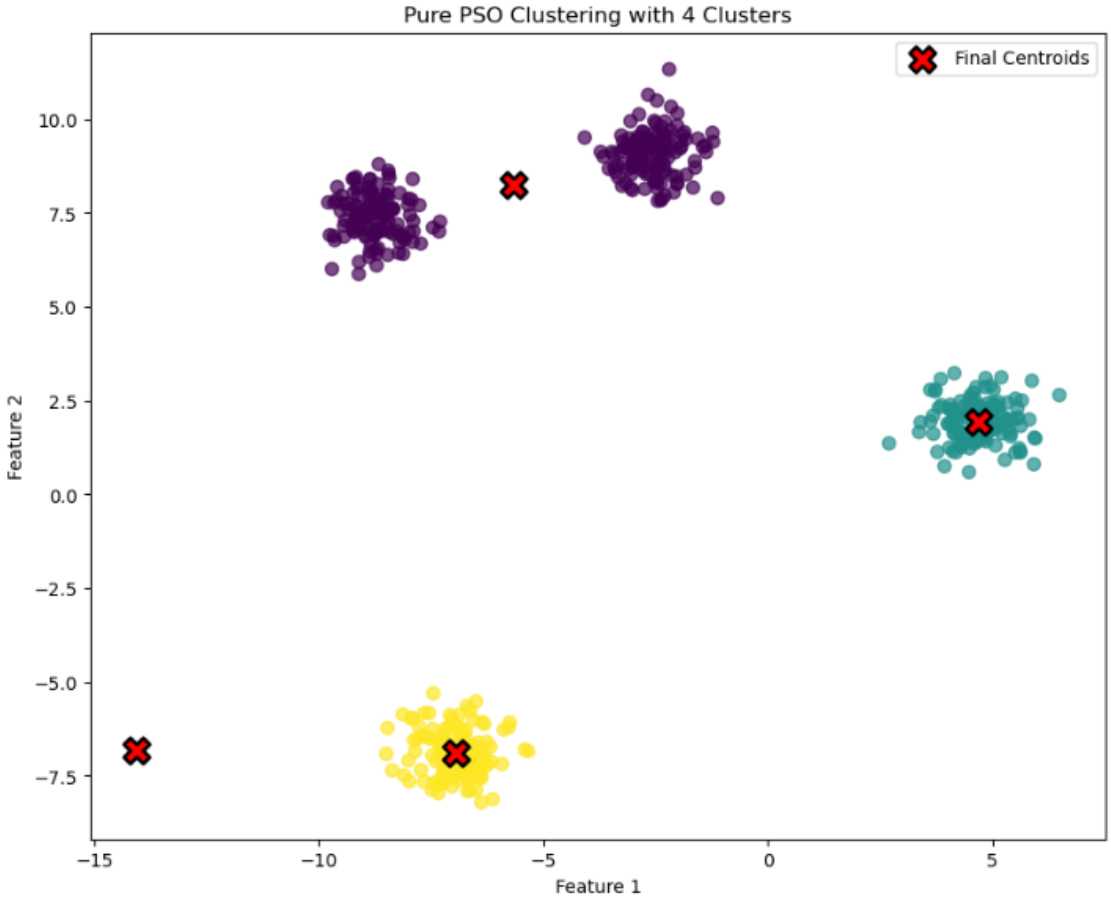
```
import numpy as np
from sklearn.datasets import make_blobs
import matplotlib.pyplot as plt

class PSOClusteringPure:
    """
    Pure Particle Swarm Optimization (PSO) for data clustering.
    Each particle represents a set of 'k' cluster centroids.
    """
    def __init__(self, n_clusters, n_particles, data, w=0.7,
c1=2.0, c2=2.0, max_iter=100):
        self.n_clusters = n_clusters
        self.n_particles = n_particles
        self.data = data
        self.n_features = data.shape[1]
        self.w = w  # Inertia weight
        self.c1 = c1  # Cognitive factor
        self.c2 = c2  # Social factor
        self.max_iter = max_iter

        # Initialize particles randomly across the data range
        min_vals = np.min(data, axis=0)
        max_vals = np.max(data, axis=0)
        self.particles = np.random.rand(n_particles, n_clusters,
self.n_features) * (max_vals - min_vals) + min_vals
        self.velocities = np.zeros_like(self.particles)

        # Initialize personal best positions and scores
        self.pbest_positions = self.particles.copy()
        self.pbest_scores = np.full(n_particles, np.inf)

        # Initialize global best position and score
        self.gbest_position = None
        self.gbest_score = np.inf

    def fitness_function(self, centroids):
        """
        Calculates the fitness (Sum of Squared Errors) of a
particle.
        Lower SSE means a better clustering.
        """
        # Assign each data point to the nearest centroid
        distances = np.sqrt(((self.data - centroids[:,
np.newaxis])**2).sum(axis=2))
        cluster_assignments = np.argmin(distances, axis=0)

        # Calculate the SSE for the current centroids
```

```
        sse = 0
        for i in range(self.n_clusters):
            cluster_points = self.data[cluster_assignments == i]
            if len(cluster_points) > 0:
                sse += np.sum((cluster_points - centroids[i])**2)

        return sse

    def optimize(self):
        """
        The main optimization loop for pure PSO clustering.
        """
        for i in range(self.max_iter):
            for j in range(self.n_particles):
                # Calculate fitness for the current particle
                current_fitness =
self.fitness_function(self.particles[j])

                # Update personal best if current position is
better
                if current_fitness < self.pbest_scores[j]:
                    self.pbest_scores[j] = current_fitness
                    self.pbest_positions[j] =
self.particles[j].copy()

                # Update global best if personal best is better
than current global best
                if current_fitness < self.gbest_score:
                    self.gbest_score = current_fitness
                    self.gbest_position = self.particles[j].copy()

            # Update velocity and position for each particle
            for j in range(self.n_particles):
                r1, r2 = np.random.rand(2)

                # Update velocity based on inertia, cognitive, and
social components
                cognitive_velocity = self.c1 * r1 *
(self.pbest_positions[j] - self.particles[j])
                social_velocity = self.c2 * r2 *
(self.gbest_position - self.particles[j])
                self.velocities[j] = self.w * self.velocities[j] +
cognitive_velocity + social_velocity

                # Update position based on new velocity
                self.particles[j] += self.velocities[j]

        return self.gbest_position
```

```
    def get_labels(self):
        """
        Assigns data points to the final clusters based on the
global best centroids.
        """
        if self.gbest_position is None:
            raise ValueError("Optimization has not been run yet.
Call optimize() first.")

        distances = np.sqrt(((self.data - self.gbest_position[:,
np.newaxis])**2).sum(axis=2))
        return np.argmin(distances, axis=0)

# --- Main execution block ---
if __name__ == '__main__':
    # 1. Generate synthetic data
    X, _ = make_blobs(n_samples=500, centers=4, cluster_std=0.60,
random_state=42)
    n_clusters = 4

    # 2. Run pure PSO to find the clusters
    print("Running pure PSO clustering...")
    pso = PSOClusteringPure(n_clusters=n_clusters, n_particles=50,
data=X, max_iter=200)
    final_centroids = pso.optimize()
    final_labels = pso.get_labels()
    print("PSO clustering complete.")

    # 3. Visualize the results
    plt.figure(figsize=(10, 8))
    plt.scatter(X[:, 0], X[:, 1], c=final_labels, cmap='viridis',
s=50, alpha=0.7)
    plt.scatter(final_centroids[:, 0], final_centroids[:, 1],
marker='X', s=200, c='red', edgecolor='black', linewidth=2,
label='Final Centroids')
    plt.title(f'Pure PSO Clustering with {n_clusters} Clusters')
    plt.xlabel('Feature 1')
    plt.ylabel('Feature 2')
    plt.legend()
    plt.show()
```

Output:

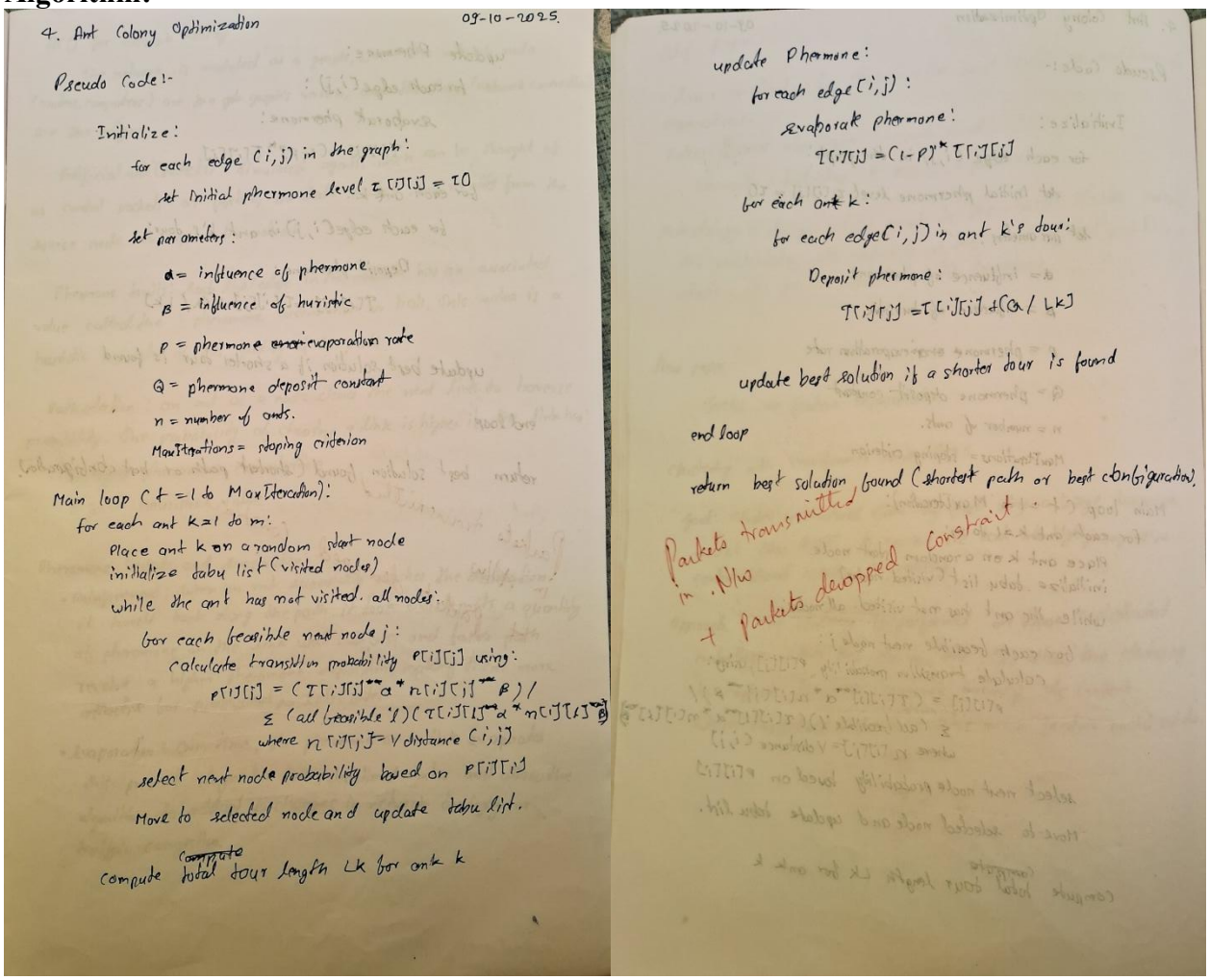
```
Running pure PSO clustering...
PSO clustering complete.
```



Program 4

Problem statemen: Use Ant Colony Optimization Algorithm to solve the network routing problem.

Algorithm:



4. Ant Colony Optimization 09-10-2025.

Pseudo Code:-

Initialize:
for each edge (i,j) in the graph:
set initial pheromone level τ[i][j] = τ0
set parameters:
α = influence of pheromone
β = influence of huristic
ρ = pheromone ~~evaporation~~ evaporation rate
Q = pheromone deposit constant
n = number of ants.
MaxIterations = stoping criterion

Main loop (t = 1 to MaxIteration):
for each ant k=1 to m:
Place ant k on a random start node
initialize tabu list (visited nodes)
while the ant has not visited all nodes:
for each feasible next node j:
calculate transition probability P[i][j] using:
P[i][j] = (τ[i,j]^α * n[i,j]^β) /
Σ (all feasible l)(τ[i,j][l]^α * n[i,j][l]^β)
where n[i][j] = 1/distance(i,j)
select next node probability based on P[i][j]
Move to selected node and update tabu list.

Compute total tour length Lk for ant k

update Pheromone:
for each edge(i,j):
evaporate pheromone:
τ[i][j] = (1-ρ)* τ[i][j]

for each ant k:
for each edge(i,j) in ant k's tour:
Deposit pheromone:
τ[i][j] = τ[i][j] + (Q / Lk)

update best solution if a shorter tour is found

end loop

return best solution found (shortest path or best configuration)

Packets transmitted in NIw
+ Packets dewopped Constraint.

ACO for network Routing

The network is modeled as a graph, where the network node (routers, computers) are the graph's vertices, and the links (network connections) are the edges.

Artificial ants (packets): Simulated agents, which can be thought of as control packets or parts of the data packet, are released from the source node to explore the network.

Pheromone trails: Each link/edge in the network has an associated value called the pheromone concentration or trail. This value is a heuristic indicator of the quality of the path.

Path selection: an ant at a node chooses the next link to traverse probability. The probability of choosing a link is higher if the link has:

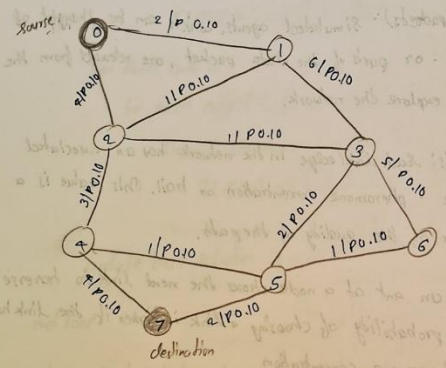
- A higher pheromone concentration
- A good heuristic value

Pheromone Update and Evaporation:

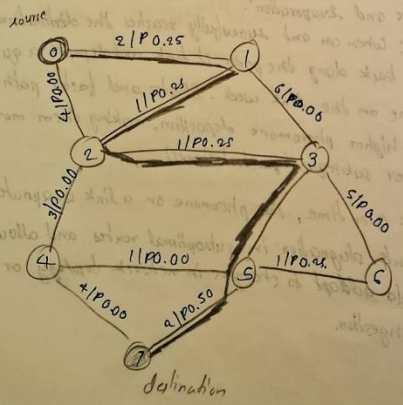
- Reinforcement: When an ant successfully reaches the destination it travels back along the path it took and deposits a quantity of pheromone on the link used. Shorter and faster path receive a higher pheromone deposition, making them more attractive for subsequent packets (ants)
- Evaporation: Over time, the pheromone on a link evaporates this prevents stagnation in suboptimal routes and allow the algorithm to adapt to changes in network topology or traffic congestion.

ACO simulation. (8 node) network.

initially.



Final Routing path Best distance : 8.



optimal path : 0 → 1 → 2 → 3 → 5 → 7.
distance : 8.

In case of packed droping.

ACO routing is inherently adaptive to packet droping, particularly when droping is due to.

congestion

link failures

It is achived by the core concepts of positive feedback and pheromone evaporation.

- in case of packet droping due to failure to reinforce the 'Bad' path using the positive feedback.
- The pheromone evaporation ensures that a blocked or congested path is quickly "forgotten"

TSP

TSP Nearest Neighbor Tour (Path 0 -> 0).

0 -> 1 -> 2 -> 3 -> 5 -> 4 -> 7 -> 6 -> 0

total distance :- 18.

visited nodes 0, 1, 2, 3, 4, 5, 6, 7.

Longer path then the Aco

Highly constrained.

Code:

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

# --- More Complex Network Configuration (Graph) ---
# Nodes 0 to 7. Distances are non-zero for connected links.
DISTANCE_MATRIX = np.array([
#   0  1  2  3  4  5  6  7
    [0, 2, 4, 0, 0, 0, 0, 0], # 0 (Source)
    [2, 0, 1, 6, 0, 0, 0, 0], # 1
    [4, 1, 0, 1, 3, 0, 0, 0], # 2
    [0, 6, 1, 0, 0, 2, 5, 0], # 3
    [0, 0, 3, 0, 0, 1, 0, 4], # 4
    [0, 0, 0, 2, 1, 0, 1, 2], # 5
    [0, 0, 0, 5, 0, 1, 0, 1], # 6
    [0, 0, 0, 0, 4, 2, 1, 0]  # 7 (Destination)
])
# Note: The shortest path is likely 0 -> 1 -> 2 -> 3 -> 5 -> 7
(2+1+1+2+2 = 8)
# or 0 -> 2 -> 4 -> 5 -> 7 (4+3+1+2 = 10) or 0 -> 2 -> 3 -> 6 -> 7
(4+1+5+1 = 11)

NUM_NODES = len(DISTANCE_MATRIX)
START_NODE = 0
END_NODE = 7

# --- ACO Parameters ---
NUM_ANTS = 20         # Increased ants for faster exploration
NUM_ITERATIONS = 75   # Increased iterations to show clear
convergence
EVAPORATION_RATE = 0.4  # Slightly lower evaporation
PHEROMONE_DEPOSIT = 1.0
ALPHA = 1.0
BETA = 3.0            # Increased BETA to make distance (heuristic)
a stronger factor

# Initialize Pheromone Matrix
PHEROMONE_MATRIX = np.ones((NUM_NODES, NUM_NODES)) * 0.1

# --- Graph Visualization Setup ---

# Create a NetworkX graph object from the distance matrix
G = nx.Graph()
for i in range(NUM_NODES):
    for j in range(i + 1, NUM_NODES):
        if DISTANCE_MATRIX[i, j] > 0:
```

```
            G.add_edge(i, j, weight=DISTANCE_MATRIX[i, j])

# Set a fixed layout for consistent plotting
pos = nx.spring_layout(G, seed=42)

def draw_network(G, pos, best_path, iteration, shortest_distance):
    """Draws the network graph, highlighting the best path and link
pheromones."""
    plt.figure(figsize=(12, 8))
    plt.title(f"ACO Routing - Iteration {iteration} | Best
Distance: {shortest_distance}")

    # 1. Update edge attributes for drawing
    # Avoid errors if all pheromones are zero by checking
max_pheromone
    max_pheromone = np.max(PHEROMONE_MATRIX) if
np.max(PHEROMONE_MATRIX) > 0 else 1.0

    edge_weights = nx.get_edge_attributes(G, 'weight')
    edge_labels = {}
    edge_color_map = []
    edge_width_map = []

    for u, v in G.edges():
        pher = PHEROMONE_MATRIX[u, v]

        # Pheromone for label (rounded)
        edge_labels[(u, v)] = f"{edge_weights[(u, v)]} |
P:{pher:.2f}"

        # Color: Map pheromone intensity (green for high)
        color_intensity = pher / max_pheromone
        edge_color_map.append((1 - color_intensity,
color_intensity, 0)) # Red (low) to Green (high)

        # Width: Thicker lines for higher pheromone (min width 1,
max 6)
        edge_width_map.append(1.5 + (pher / max_pheromone) * 4.5)

    # 2. Draw all edges with color/width based on Pheromone
    nx.draw_networkx_edges(
        G, pos,
        edge_color=edge_color_map,
        width=edge_width_map,
        alpha=0.7
    )

    # 3. Draw nodes
```

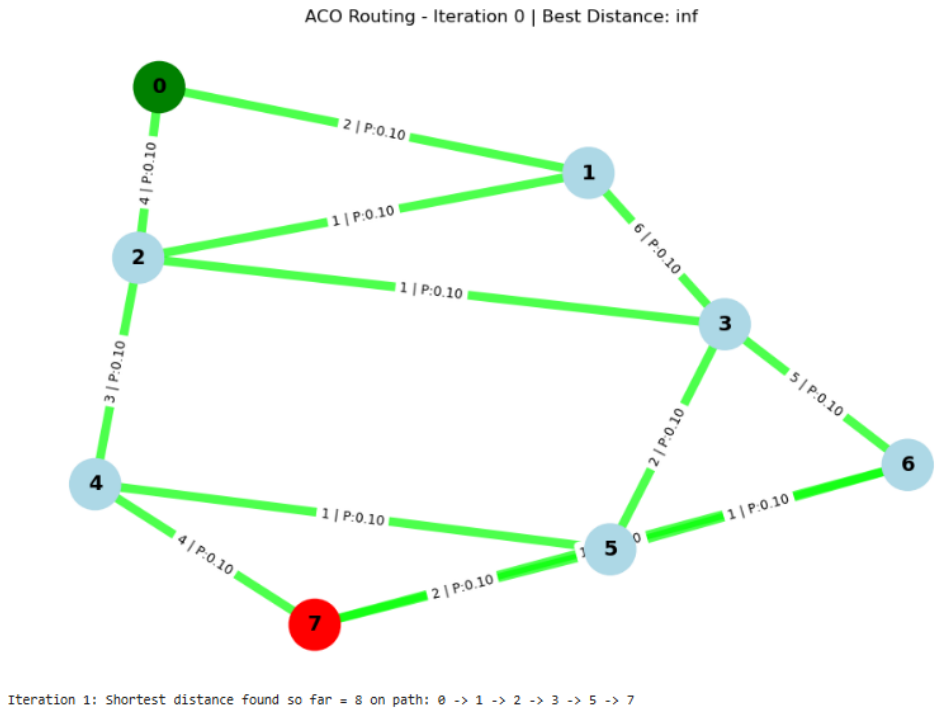
```
    node_colors = ['#ADD8E6'] * NUM_NODES # Light blue default
    node_colors[START_NODE] = '#008000' # Green source
    node_colors[END_NODE] = '#FF0000'   # Red destination

    nx.draw_networkx_nodes(G, pos, node_color=node_colors,
node_size=1200)

    # 4. Draw node labels
    nx.draw_networkx_labels(G, pos, font_size=14,
font_weight='bold', font_color='black')

    # 5. Draw edge labels (Distance | Pheromone)
    nx.draw_networkx_edge_labels(G, pos, edge_labels=edge_labels,
font_size=9, label_pos=0.5)

    # 6. Highlight the current Best Path
    if best_path:
        path_edges = list(zip(best_path, best_path[1:]))
        nx.draw_networkx_edges(
            G, pos,
            edgelist=path_edges,
            edge_color='blue',
            width=5,
            alpha=1.0
        )

    plt.axis('off')
    plt.show()

# ------------------------------------------------------------
# --- Core ACO Functions (Rely on PHEROMONE_MATRIX and
DISTANCE_MATRIX) ---
# ------------------------------------------------------------

def get_allowed_neighbors(current_node, visited_nodes):
    neighbors = []
    for neighbor in range(NUM_NODES):
        if DISTANCE_MATRIX[current_node, neighbor] > 0 and neighbor
not in visited_nodes:
            neighbors.append(neighbor)
    return neighbors

def calculate_transition_probability(current_node, neighbors):
    probabilities = {}
    total_attractiveness = 0

    for next_node in neighbors:
        tau = PHEROMONE_MATRIX[current_node, next_node] ** ALPHA
```

```
        distance = DISTANCE_MATRIX[current_node, next_node]
        # Heuristic (eta): Inverse of distance, strongly favors
short links
        eta = (1.0 / distance) ** BETA

        attractiveness = tau * eta
        total_attractiveness += attractiveness
        probabilities[next_node] = attractiveness

    if total_attractiveness == 0:
        return {node: 1.0 / len(neighbors) for node in neighbors}
    else:
        return {node: prob / total_attractiveness for node, prob in
probabilities.items()}

def run_ant_simulation():
    current_node = START_NODE
    path = [START_NODE]
    path_distance = 0
    pheromone_path = []

    while current_node != END_NODE:
        visited_nodes = set(path)
        neighbors = get_allowed_neighbors(current_node,
visited_nodes)

        if not neighbors:
            # Trapped: failed path (simulates a dropped packet/link
failure)
            return None, float('inf'), None

        probabilities =
calculate_transition_probability(current_node, neighbors)
        next_node = random.choices(
            list(probabilities.keys()),
            weights=list(probabilities.values()),
            k=1
        )[0]

        pheromone_path.append((current_node, next_node))
        path_distance += DISTANCE_MATRIX[current_node, next_node]
        path.append(next_node)
        current_node = next_node

    return path, path_distance, pheromone_path

def update_pheromones(paths_found):
    global PHEROMONE_MATRIX
```

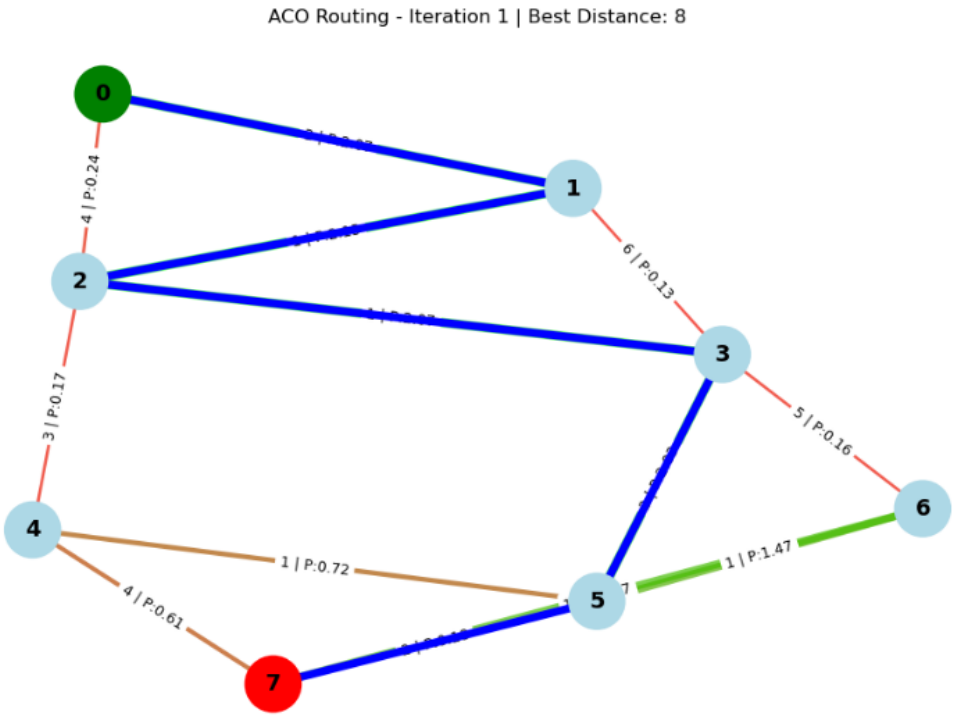
```
    # 1. Evaporation: Fades all old trails
    PHEROMONE_MATRIX = PHEROMONE_MATRIX * (1.0 - EVAPORATION_RATE)

    # 2. Deposit (Reinforcement): Strengthens successful trails
    for path_data in paths_found:
        path_distance = path_data['distance']
        pheromone_path = path_data['pheromone_path']

        if path_distance > 0:
            # The shorter the distance, the larger the deposit
            pheromone_delta = PHEROMONE_DEPOSIT / path_distance

            for i, j in pheromone_path:
                PHEROMONE_MATRIX[i, j] += pheromone_delta
                PHEROMONE_MATRIX[j, i] += pheromone_delta

# --- Main ACO Execution ---

best_path = None
shortest_distance = float('inf')
print("Starting ACO Simulation with a Complex Network (8
nodes)...")

# Visualize the initial state
draw_network(G, pos, best_path, 0, shortest_distance)
#

for iteration in range(NUM_ITERATIONS):
    all_ant_results = []

    # Run all ants and gather results
    for ant in range(NUM_ANTS):
        path, distance, pheromone_path = run_ant_simulation()
        if path:
            all_ant_results.append({
                'path': path,
                'distance': distance,
                'pheromone_path': pheromone_path
            })

            # Update best path found so far
            if distance < shortest_distance:
                shortest_distance = distance
                best_path = path

    # Update pheromone trails
    update_pheromones(all_ant_results)
```

```
    # Visualize the graph every 25 iterations and at the end
    if (iteration + 1) % 25 == 0 or iteration == NUM_ITERATIONS - 1
or iteration == 0:
        print(f"Iteration {iteration + 1}: Shortest distance found
so far = {shortest_distance} on path: {' -> '.join(map(str,
best_path)) if best_path else 'None'}")
        draw_network(G, pos, best_path, iteration + 1,
shortest_distance)


# --- Final Results ---
print("\n--- Simulation Complete ---")
print(f"Final Shortest Distance: {shortest_distance}")
print(f"Optimal Path: {' -> '.join(map(str, best_path))}")
```

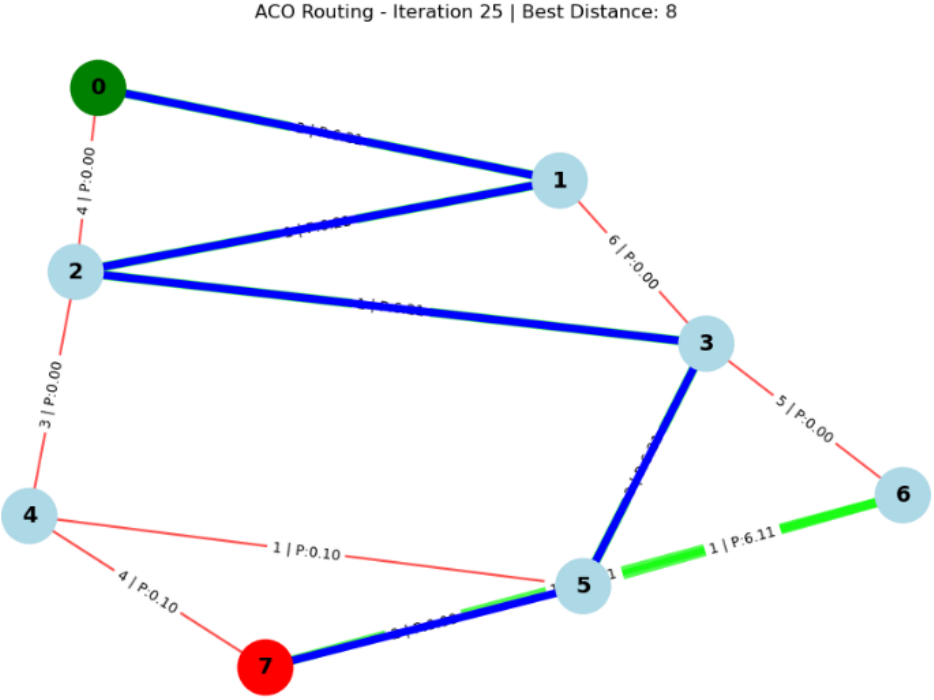
Output:



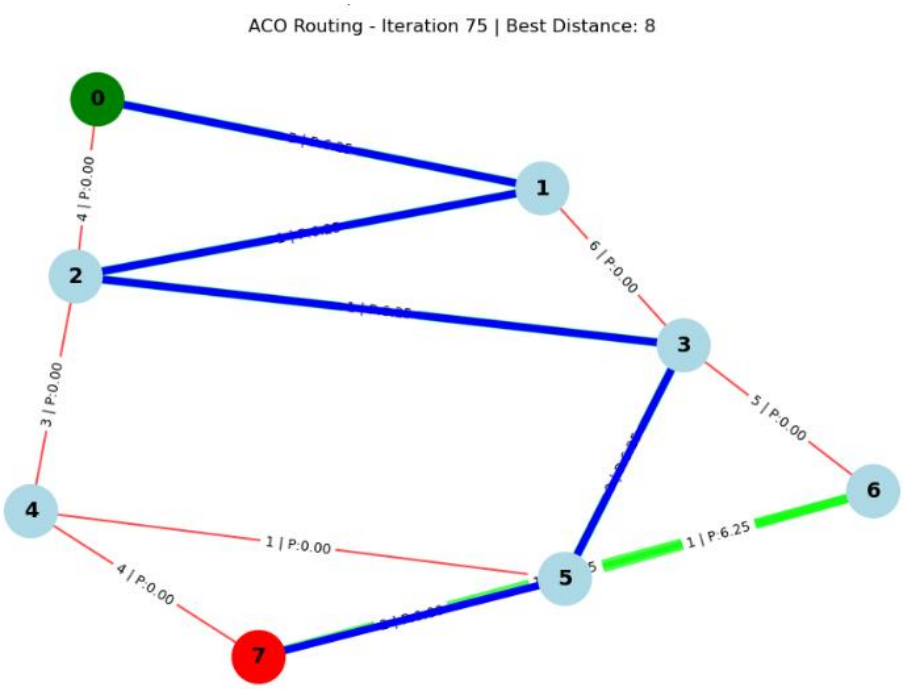
Iteration 1: Shortest distance found so far = 8 on path: 0 -> 1 -> 2 -> 3 -> 5 -> 7



Iteration 25: Shortest distance found so far = 8 on path: 0 -> 1 -> 2 -> 3 -> 5 -> 7



Iteration 50: Shortest distance found so far = 8 on path: 0 -> 1 -> 2 -> 3 -> 5 -> 7

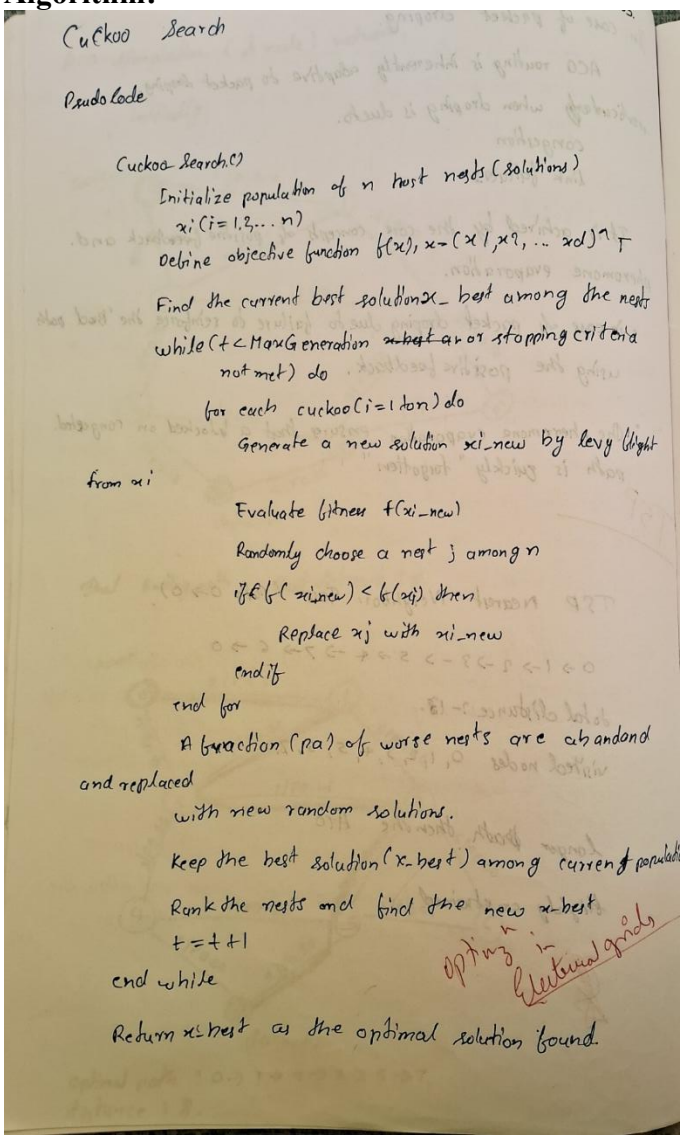



```
--- Simulation Complete ---
Final Shortest Distance: 8
```

Program 5

Problem statemen: Use Cuckoo Search Algorithm to solve the Electric Grid (simple economic dispatch problem) problem.

Algorithm:



Cuckoo Search

Pseudo Code

Cuckoo_Search()
Initialize population of n host nests (solutions)
x_i (i = 1, 2, ... n)
Define objective function f(x), $x = (x_1, x_2, \ldots x_d)^T$
Find the current best solution x_best among the nests
while (t < MaxGeneration ~~x_best~~ or stopping criteria not met) do
for each cuckoo (i = 1 to n) do
Generate a new solution x_{i_new} by levy flight from x_i
Evaluate fitness $f(x_{i\_new})$
Randomly choose a nest j among n
if ($f(x_{i\_new}) < f(x_j)$) then
Replace x_j with x_{i_new}
end if
end for
A fraction (pa) of worse nests are abandond and replaced
with new random solutions.
Keep the best solution (x_best) among current populati
Rank the nests and find the new x_best
t = t + 1
end while
Return x_best as the optimal solution found.

optimization in Electrical grids

Cuckoo Search in Electric Grids.

Electric grids are complex, nonlinear, and often involve mixed-integer variables.

CSA works well here because it uses Levy flights for global exploration and selection for intensification.

Implementing CSA for a Simple Economic Dispatch problem. which is one of the most common optimization problems in electric grids.

We have 3 generators supplying power to meet a total load demand $P_d = 500$ MW

quadratic cost function

$$C_i(P_i) = a_i + b_i P_i + c_i P_i^2$$

$$P_i^{min} \le P_i \le P_i^{max}$$

Generator	a	b	c	Pmin	Pmax
1	500	5.3	0.004	100	400
2	400	5.5	0.006	100	356
3	200	5.8	0.009	50	300

Output.

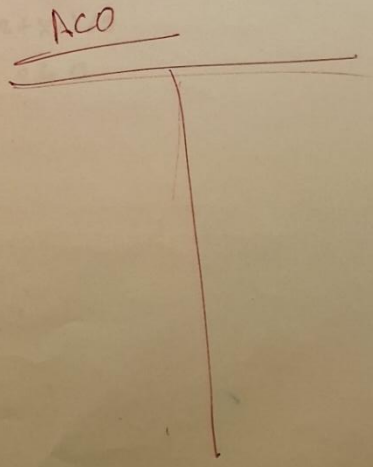
Generator 1: 256.40 MW

Generator 2: 156.67 MW

Generator 3: 86.93 MW

Total power: 500.00 MW

Total Cost: ₹4303.05.



	ACO	
Inspiration	Foraging behavior of ants using pheromones	Brood parasitism of cuckoo birds + Levy flights.
Search Mechanism	Construct solution step by step based on pheromone trails and heuristic info	Global random search using Levy flights and elitism.
Convergence speed	Slower initially due to pheromone learning phase	Usually faster due to Levy flight
Best suited for	discrete / combinational	continuous / hybrid problems.
Parameters	Many (α, β, ρ, ants)	Few (pa, population size).

Code:

```
import numpy as np
print("Name: PRANAV GAJANAN KAMATE\nUSN: 1BM23CS241")
# ==========================
# 1️⃣ Problem Data
# ==========================
demand = 500  # MW

# Generator data: [a, b, c, Pmin, Pmax]
generators = np.array([
    [500, 5.3, 0.004, 100, 400],
    [400, 5.5, 0.006, 100, 350],
    [200, 5.8, 0.009,  50, 300]
])

num_gens = len(generators)

# ==========================
# 2️⃣ Helper Functions
# ==========================

def cost_function(P):
    """Calculate total generation cost + penalty for power balance
violation."""
    total_cost = 0
    for i in range(num_gens):
        a, b, c, *_ = generators[i]
        total_cost += a + b*P[i] + c*(P[i]**2)

    # Penalty for power mismatch
    penalty = 1e5 * (np.sum(P) - demand)**2
    return total_cost + penalty

def random_solution():
    """Generate a random feasible power allocation that satisfies
limits approximately."""
    P = np.array([np.random.uniform(gen[3], gen[4]) for gen in
generators])
    # Adjust proportionally to match total demand
    P = P * demand / np.sum(P)
    return P

def levy_flight(beta=1.5):
    """Generate step size using Lévy distribution."""
    sigma = (np.math.gamma(1+beta) * np.sin(np.pi*beta/2) /
             (np.math.gamma((1+beta)/2)*beta*2**((beta-
1)/2)))**(1/beta)
    u = np.random.normal(0, sigma, size=num_gens)
```

```
    v = np.random.normal(0, 1, size=num_gens)
    step = u / np.abs(v)**(1/beta)
    return step

def apply_bounds(P):
    """Ensure generator limits are respected."""
    for i in range(num_gens):
        P[i] = np.clip(P[i], generators[i][3], generators[i][4])
    # Adjust to meet demand approximately
    P = P * demand / np.sum(P)
    return P

# ==========================
# 3️⃣ Cuckoo Search Parameters
# ==========================
n = 20              # Number of nests
pa = 0.25           # Discovery probability
max_iter = 200

# ==========================
# 4️⃣ Initialization
# ==========================
nests = np.array([random_solution() for _ in range(n)])
fitness = np.array([cost_function(P) for P in nests])
best_idx = np.argmin(fitness)
best = nests[best_idx].copy()

# ==========================
# 5️⃣ Main Loop
# ==========================
for t in range(max_iter):
    for i in range(n):
        # Lévy flight from current nest
        step = levy_flight()
        new_nest = nests[i] + step * (nests[i] - best) * 0.01
        new_nest = apply_bounds(new_nest)

        new_fitness = cost_function(new_nest)
        j = np.random.randint(n)

        if new_fitness < fitness[j]:
            nests[j] = new_nest
            fitness[j] = new_fitness

    # Abandon a fraction of worst nests
    num_abandon = int(pa * n)
    worst_idx = np.argsort(fitness)[-num_abandon:]
```

```
    for idx in worst_idx:
        nests[idx] = random_solution()
        fitness[idx] = cost_function(nests[idx])

    # Update global best
    current_best_idx = np.argmin(fitness)
    if fitness[current_best_idx] < cost_function(best):
        best = nests[current_best_idx].copy()

# ==========================
# 6️⃣ Results
# ==========================
print("=== Cuckoo Search Economic Dispatch ===")
for i in range(num_gens):
    print(f"Generator {i+1}: {best[i]:.2f} MW")

print(f"Total Power: {np.sum(best):.2f} MW")
print(f"Total Cost: ₹ {cost_function(best):.2f}")
```

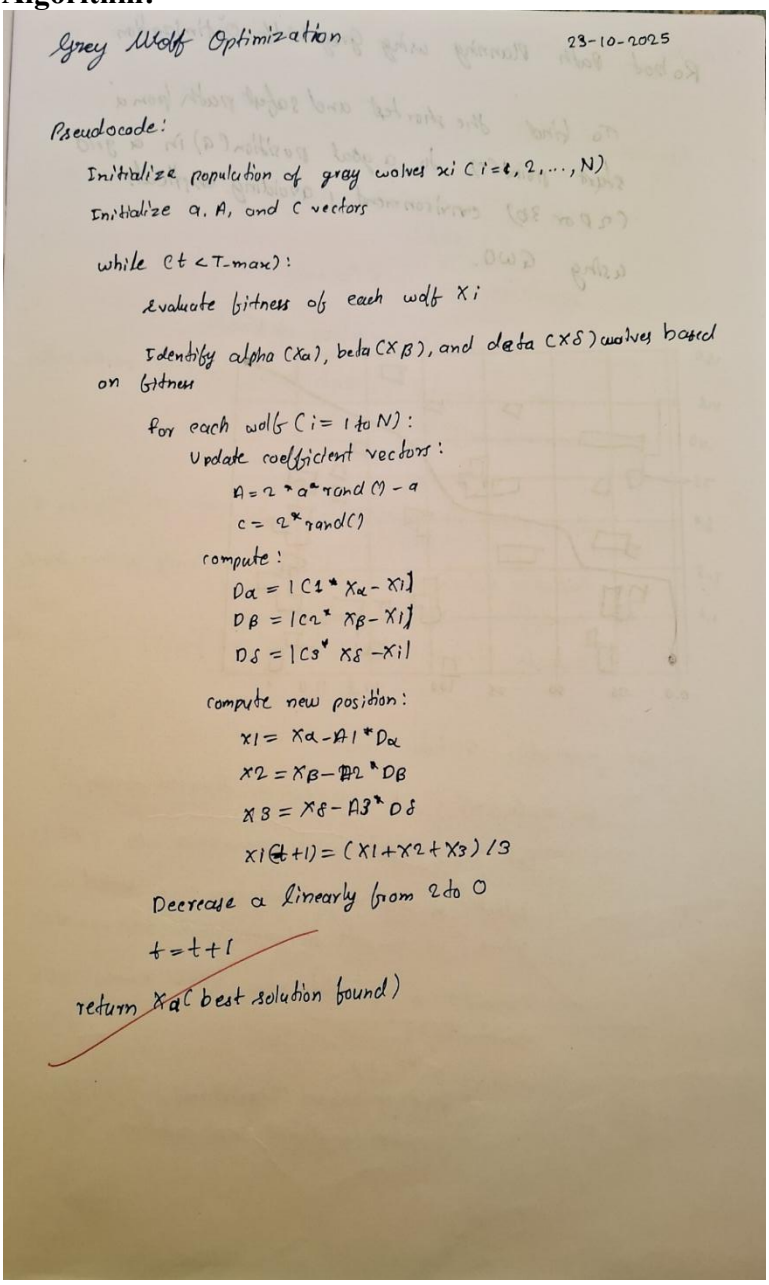
Output:

```
Name: PRANAV GAJANAN KAMATE
USN: 1BM23CS241
=== Cuckoo Search Economic Dispatch ===
Generator 1: 255.76 MW
Generator 2: 155.78 MW
Generator 3: 88.46 MW
Total Power: 500.00 MW
Total Cost: ₹ 4303.07
```

Program 6

Problem statemen: Use Grey Wolf Optimization Algorithm to solve the shortest and safest path in a grid.

Algorithm:



Grey Wolf Optimization 23-10-2025

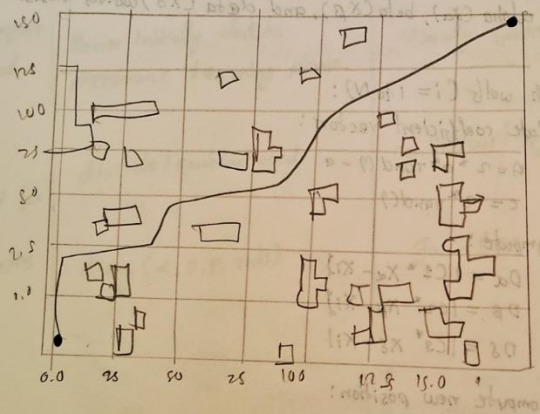
Pseudocode:

Initialize population of gray wolves xi (i=1, 2, ..., N)
Initialize a, A, and C vectors

while (t < T_max):
    Evaluate fitness of each wolf Xi
    Identify alpha (Xα), beta (Xβ), and delta (Xδ) wolves based on fitness
    for each wolf (i = 1 to N):
        Update coefficient vectors:
            A = 2 * a * rand() - a
            C = 2 * rand()
        compute:
            Dα = |C1 * Xα - Xi|
            Dβ = |C2 * Xβ - Xi|
            Dδ = |C3 * Xδ - Xi|
        compute new position:
            X1 = Xα - A1 * Dα
            X2 = Xβ - A2 * Dβ
            X3 = Xδ - A3 * Dδ
            Xi(t+1) = (X1 + X2 + X3) / 3
    Decrease a linearly from 2 to 0
    t = t + 1
return Xα (best solution found)

Robot Path Planning using Grey Wolf

To find the shortest and safest path from a start point(s) to a goal position(s) in a gold (2D or 3D) environment, avoiding obstacles using GWO.



Code:

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

# ==============================
# Parameters & Environment
# ==============================
GRID_SIZE = 20
START = np.array([0, 0])    # [row, col]
GOAL  = np.array([19, 19])  # [row, col]
OBSTACLE_DENSITY = 0.2  # 20% obstacles

np.random.seed(0)
obstacle_map = (np.random.rand(GRID_SIZE, GRID_SIZE) <
OBSTACLE_DENSITY).astype(int)
obstacle_map[START[0], START[1]] = 0
obstacle_map[GOAL[0], GOAL[1]] = 0

# ==============================
# Helpers: validity, repair, collisions
# ==============================
def is_valid(point):
    r, c = int(round(point[0])), int(round(point[1]))
    return 0 <= r < GRID_SIZE and 0 <= c < GRID_SIZE and
obstacle_map[r, c] == 0

def find_nearest_free_cell(r, c, max_radius=6):
    r0, c0 = int(round(r)), int(round(c))
    if 0 <= r0 < GRID_SIZE and 0 <= c0 < GRID_SIZE and
obstacle_map[r0, c0] == 0:
        return r0, c0
    for radius in range(1, max_radius + 1):
        for dr in range(-radius, radius + 1):
            for dc in range(-radius, radius + 1):
                nr, nc = r0 + dr, c0 + dc
                if 0 <= nr < GRID_SIZE and 0 <= nc < GRID_SIZE and
obstacle_map[nr, nc] == 0:
                    return nr, nc
    # fallback: return START or any free cell
    for nr in range(GRID_SIZE):
        for nc in range(GRID_SIZE):
            if obstacle_map[nr, nc] == 0:
                return nr, nc
    return r0, c0

def repair_path(path):
    repaired = path.copy()
```

```
    for i in range(len(repaired)):
        repaired[i] = np.clip(repaired[i], 0, GRID_SIZE - 1)
        r, c = repaired[i]
        if obstacle_map[int(round(r)), int(round(c))] == 1:
            nr, nc = find_nearest_free_cell(r, c)
            repaired[i] = np.array([nr, nc], dtype=float)
    repaired[0] = START.astype(float)
    repaired[-1] = GOAL.astype(float)
    return repaired

def segment_collision(p1, p2, samples=12):
    for t in np.linspace(0, 1, samples):
        pt = p1 * (1 - t) + p2 * t
        if not is_valid(pt):
            return True
    return False

def fitness(path):
    length = 0.0
    collisions = 0
    for i in range(len(path) - 1):
        length += np.linalg.norm(path[i + 1] - path[i])
        if segment_collision(path[i], path[i + 1], samples=12):
            collisions += 1
    # stronger penalty so optimizer avoids collisions
    return length + 100000.0 * collisions

# ==============================
# Initialize population (same idea as before)
# ==============================
def initialize_population(num_wolves, num_points):
    population = []
    for _ in range(num_wolves):
        xs = np.linspace(START[0], GOAL[0], num_points) +
np.random.uniform(-1.0, 1.0, num_points)
        ys = np.linspace(START[1], GOAL[1], num_points) +
np.random.uniform(-1.0, 1.0, num_points)
        path = np.stack([xs, ys], axis=1).astype(float)
        path[0] = START.astype(float)
        path[-1] = GOAL.astype(float)
        path = np.clip(path, 0, GRID_SIZE - 1)
        path = repair_path(path)
        population.append(path)
    return np.array(population)

# ==============================
# Grey Wolf Optimizer (as before)
# ==============================
```

```
def gwo_path_planning(num_wolves=30, num_points=20, max_iter=120):
    wolves = initialize_population(num_wolves, num_points)
    fitness_values = np.array([fitness(w) for w in wolves])
    sorted_idx = np.argsort(fitness_values)
    wolves = wolves[sorted_idx]
    fitness_values = fitness_values[sorted_idx]
    alpha, beta, delta = wolves[0].copy(), wolves[1].copy(),
wolves[2].copy()
    best_scores = []

    for t in range(max_iter):
        a = 2 - t * (2 / max_iter)
        for i in range(len(wolves)):
            X = wolves[i].copy()
            for j in range(1, num_points - 1):
                # alpha
                r1, r2 = np.random.rand(), np.random.rand()
                A1 = 2 * a * r1 - a
                C1 = 2 * r2
                D_alpha = np.abs(C1 * alpha[j] - X[j])
                X1 = alpha[j] - A1 * D_alpha
                # beta
                r1, r2 = np.random.rand(), np.random.rand()
                A2 = 2 * a * r1 - a
                C2 = 2 * r2
                D_beta = np.abs(C2 * beta[j] - X[j])
                X2 = beta[j] - A2 * D_beta
                # delta
                r1, r2 = np.random.rand(), np.random.rand()
                A3 = 2 * a * r1 - a
                C3 = 2 * r2
                D_delta = np.abs(C3 * delta[j] - X[j])
                X3 = delta[j] - A3 * D_delta

                new_point = (X1 + X2 + X3) / 3.0
                X[j] = np.clip(new_point, 0, GRID_SIZE - 1)

            X = repair_path(X)
            wolves[i] = X
            fitness_values[i] = fitness(X)

        sorted_idx = np.argsort(fitness_values)
        wolves = wolves[sorted_idx]
        fitness_values = fitness_values[sorted_idx]
        alpha, beta, delta = wolves[0].copy(), wolves[1].copy(),
wolves[2].copy()
        best_scores.append(fitness_values[0])
        if t % 10 == 0:
```

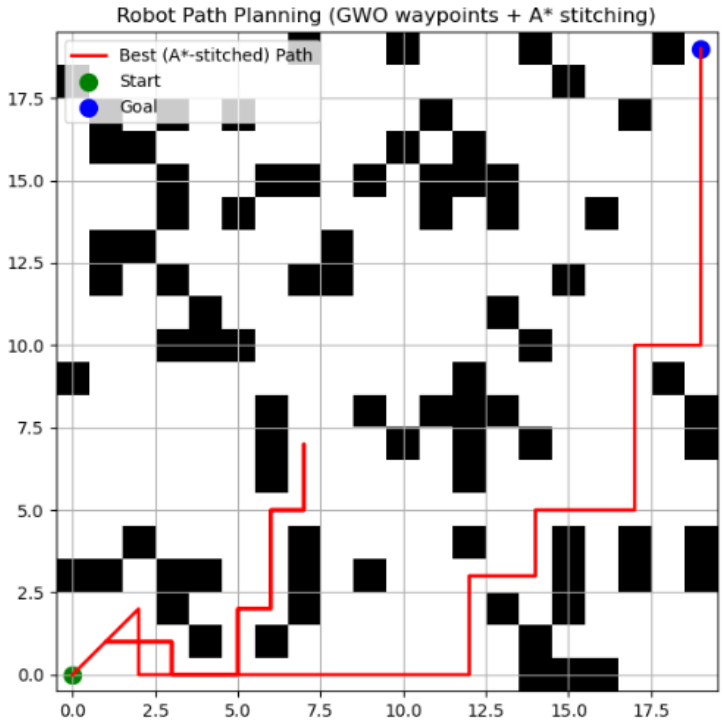
```
            print(f"Iter {t:03d}: Best fitness =
{fitness_values[0]:.2f}")

    return alpha, best_scores

# ==============================
# A* on the grid (4-neighbors)
# ==============================
def astar(start_cell, goal_cell, grid):
    """start_cell and goal_cell are (r,c) integers. Returns list of
cells from start to goal (inclusive) or None."""
    R, C = grid.shape
    def heuristic(a, b):
        return abs(a[0]-b[0]) + abs(a[1]-b[1])

    open_heap = []
    heapq.heappush(open_heap, (0 + heuristic(start_cell,
goal_cell), 0, start_cell, None))
    came_from = {}
    gscore = {start_cell: 0}
    closed = set()

    while open_heap:
        f, g, current, parent = heapq.heappop(open_heap)
        if current in closed:
            continue
        came_from[current] = parent
        if current == goal_cell:
            # reconstruct
            path = []
            cur = current
            while cur is not None:
                path.append(cur)
                cur = came_from[cur]
            return path[::-1]
        closed.add(current)

        for dr, dc in [(1,0),(-1,0),(0,1),(0,-1)]:
            nr, nc = current[0]+dr, current[1]+dc
            nb = (nr, nc)
            if 0 <= nr < R and 0 <= nc < C and grid[nr, nc] == 0:
                tentative_g = g + 1
                if nb in closed:
                    continue
                prev_g = gscore.get(nb, 1e9)
                if tentative_g < prev_g:
                    gscore[nb] = tentative_g
                    heapq.heappush(open_heap, (tentative_g +
```

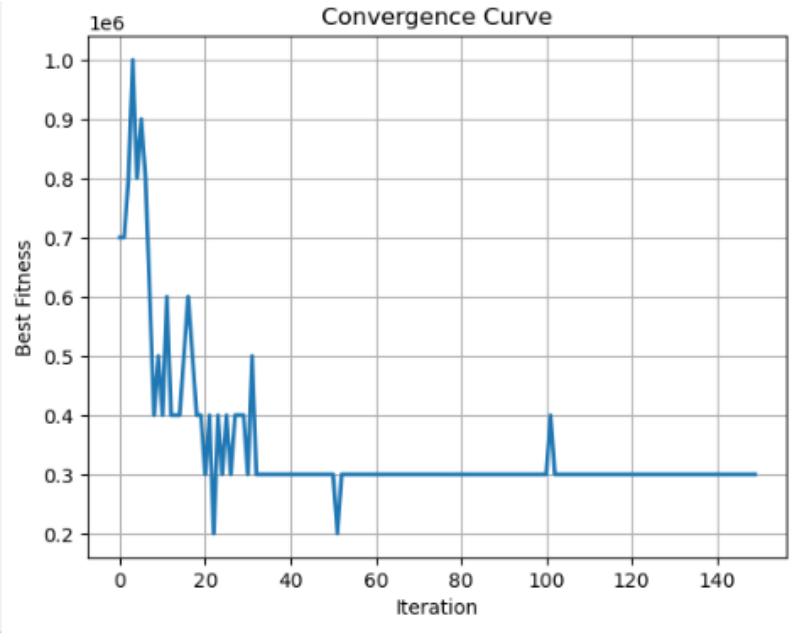
```
heuristic(nb, goal_cell), tentative_g, nb, current))
    return None

# ==============================
# Post-process best GWO path with A*: replace colliding segments
# ==============================
def postprocess_with_astar(best_path):
    """best_path: array of floats shape (N,2) [row,col].
       Returns discrete grid path (list of (r,c) ints) connected by
A* where needed."""
    discrete_path = []
    for i in range(len(best_path)-1):
        a = np.round(best_path[i]).astype(int)
        b = np.round(best_path[i+1]).astype(int)
        if segment_collision(best_path[i], best_path[i+1],
samples=20):
            astar_segment = astar(tuple(a), tuple(b), obstacle_map)
            if astar_segment is None:
                # fallback: include the endpoints at least
                if not discrete_path or discrete_path[-1] !=
tuple(a):
                    discrete_path.append(tuple(a))
                discrete_path.append(tuple(b))
            else:
                # append astar path but avoid duplicating the first
cell if already in discrete_path
                if discrete_path and discrete_path[-1] ==
astar_segment[0]:
                    discrete_path.extend(astar_segment[1:])
                else:
                    discrete_path.extend(astar_segment)
        else:
            # straight segment is free: just ensure endpoints are
present
            if not discrete_path or discrete_path[-1] != tuple(a):
                discrete_path.append(tuple(a))
            # ensure next appended later (no duplication)
            # will append b in next iteration or after loop
    # append final goal explicitly
    if not discrete_path or discrete_path[-1] !=
tuple(np.round(best_path[-1]).astype(int)):
        discrete_path.append(tuple(np.round(best_path[-
1]).astype(int)))
    return discrete_path

# ==============================
# Run GWO, postprocess, and plot
# ==============================
```

```
import numpy as np
print("Name: PRANAV GAJANAN KAMATE\nUSN: 1BM23CS241")
best_path, convergence = gwo_path_planning(num_wolves=30,
num_points=25, max_iter=150)

# Post-process: replace colliding straight segments with grid A*
paths
discrete = postprocess_with_astar(best_path)  # list of (r,c)
discrete = np.array(discrete)  # shape (M,2)

print("Final discrete path length (cells):", len(discrete))

# Plot
plt.figure(figsize=(7,7))
plt.imshow(obstacle_map, cmap='gray_r', origin='lower')
# plot discrete path: cols->x, rows->y
plt.plot(discrete[:,1], discrete[:,0], 'r-', lw=2, label='Best (A*-
stitched) Path')
plt.scatter(START[1], START[0], c='green', s=100, label='Start')
plt.scatter(GOAL[1],  GOAL[0],  c='blue',  s=100, label='Goal')
plt.title("Robot Path Planning (GWO waypoints + A* stitching)")
plt.legend()
plt.grid(True)
plt.xlim(-0.5, GRID_SIZE-0.5)
plt.ylim(-0.5, GRID_SIZE-0.5)
plt.gca().set_aspect('equal', adjustable='box')
plt.show()

plt.figure()
plt.plot(convergence, '-', lw=2)
plt.title("Convergence Curve")
plt.xlabel("Iteration")
plt.ylabel("Best Fitness")
plt.grid(True)
plt.show()
```

Output:

```
Name: PRANAV GAJANAN KAMATE
USN: 1BM23CS241
Iter 000: Best fitness = 700072.71
Iter 010: Best fitness = 400081.82
Iter 020: Best fitness = 300076.97
Iter 030: Best fitness = 300099.10
Iter 040: Best fitness = 300081.45
Iter 050: Best fitness = 300068.43
Iter 060: Best fitness = 300058.87
Iter 070: Best fitness = 300055.20
Iter 080: Best fitness = 300052.11
Iter 090: Best fitness = 300053.70
Iter 100: Best fitness = 300048.93
Iter 110: Best fitness = 300048.87
Iter 120: Best fitness = 300045.40
Iter 130: Best fitness = 300045.57
Iter 140: Best fitness = 300044.29
Final discrete path length (cells): 69
```

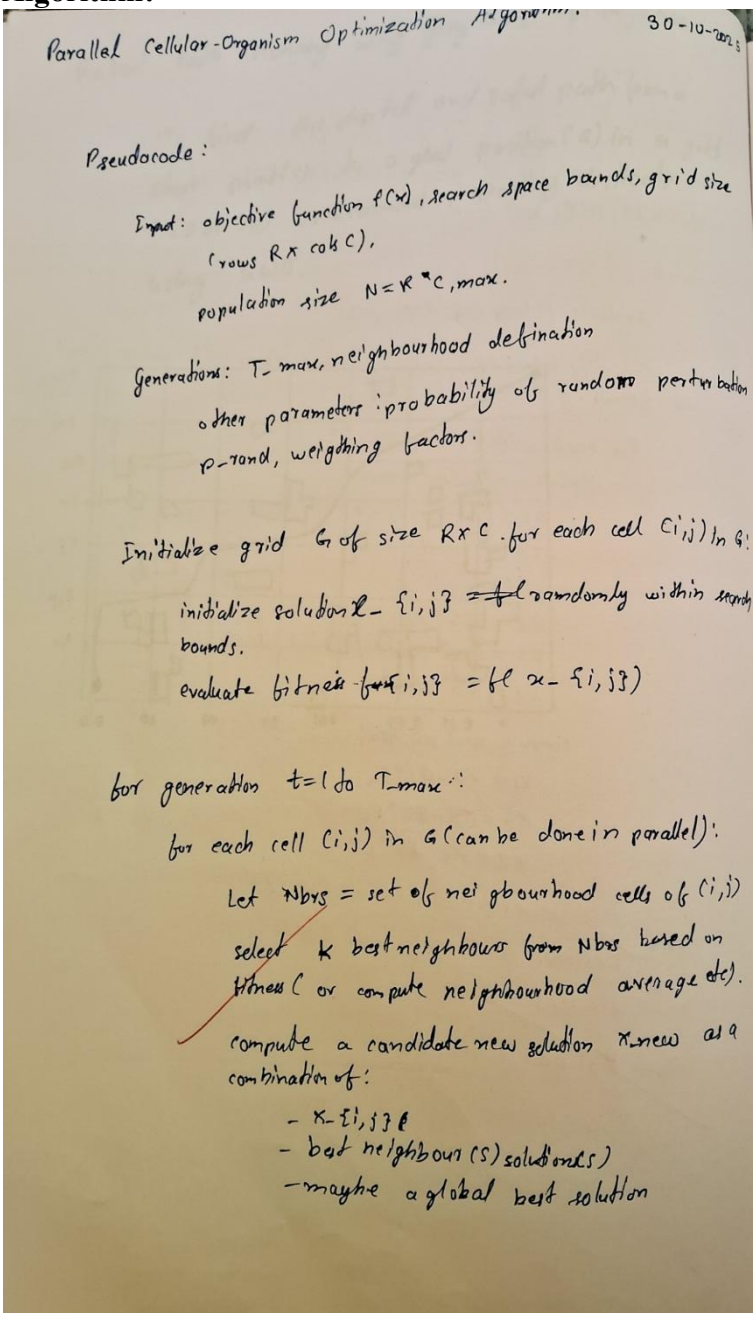



Convergence Curve
1e6
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
Best Fitness
0
20
40
60
80
100
120
140
Iteration

Program 7

Problem statemen: Use Parallel cellular-organism Optimization Algorithm to solve the scalable multi-agent coordination in gaming.

Algorithm:



Parallel Cellular-Organism Optimization Algorithm

30-10-2025

Pseudocode:

Input: objective function f(x), search space bounds, grid size (rows R x cols C),

population size N = R * C, max.

Generations: T_max, neighbourhood defination

other parameters: probability of random perturbation p_rand, weighting factors.

Initialize grid G of size R x C. for each cell (i,j) in G:

initialize solution x_{i,j} randomly within search bounds.

evaluate fitness for f(x_{i,j})

for generation t=1 do T_max:

for each cell (i,j) in G (can be done in parallel):

Let Nbrs = set of neighbourhood cells of (i,j)

select k best neighbours from Nbrs based on fitness (or compute neighbourhood average etc).

compute a candidate new solution x_new as a combination of:

- x_{i,j}
- best neighbour (s) solutions)
- maybe a global best solution

x_gbest (optional)

for example:

x_new = w0 * x_[i,j]

+ w1 * x_best_neighbour

+ w2 * x_gbest

+ rand perturbation * step_size

with probability p_rand do a random perturbation:

x_new = random solution within bounds.

Evaluate f_new = f(x_new).

if f_new is better than f_[i,j], then update:

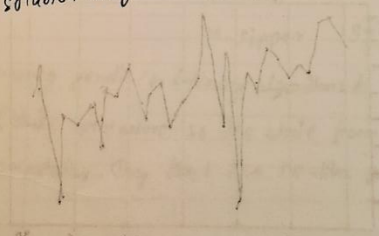
x_[i,j] := x_new

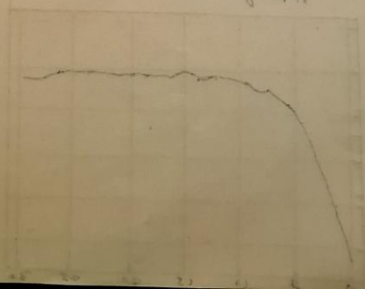
f_[i,j] := f_new

optionally update global best solution x_gbest (overall cells)

return best found solution x_gbest (or best x_[i,j]).

Sgif.





PCOO in gaming

Scalable multi-agent coordination:

Use PCOO to tune parameters for groups.

we want to coordinate multiple agents in a shared space.

such as:

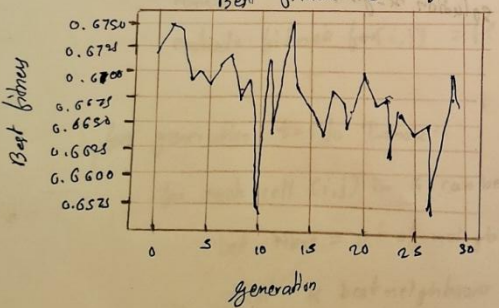
- a battlefield.
- a resource collection arena
- or a swarm navigation field.

Each agent must act autonomously but also cooperatively adapting its behavior to both:

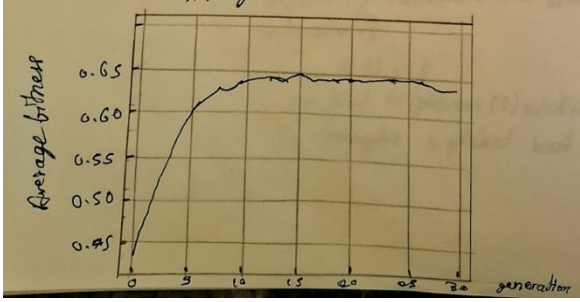
local neighbors and global goals.

Output

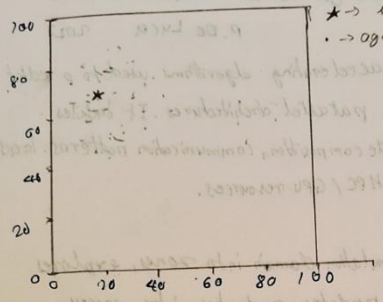
Best fitness over generations.



Average fitness over generations.



Final agent positions and shared goal



RT

Old paper

Evolution of Parallel Cellular Machines

M. Sipper 1997

cellular programming - using genetic / evolutionary algorithms to evolve local rules for arrays of simple processors so the whole parallel array performs useful global computations. They treat the CA-like grid itself as the evolving machine.

Method: Genetic algorithms applied to populations of candidate rule-sets; fitness evaluated on global tasks run on a grid of cells experiments across several canonical CA benchmarks.

Code:

```
"""
PCOO-based scalable multi-agent coordination demo (single-file).
- Grid of organisms; each organism encodes boid-like behavior
weights.
- Each generation runs a joint simulation to evaluate all agents'
fitness.
- PCOO local updates (neighborhood crossover/mutation/selection)
with double-buffering.
- Produces plots and saves top genomes to CSV.

Dependencies:
    pip install numpy matplotlib pandas

Optional (Jupyter interactive table):
    caas_jupyter_tools (only used in the original run; not
required)
"""

import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import math

np.random.seed(42)

# ----- Parameters -----
GRID_W, GRID_H = 8, 8            # grid of organisms
NUM_AGENTS = GRID_W * GRID_H
GENOME_DIM = 4                   # [w_align, w_cohere, w_sep,
w_goal]
P_SWAP = 0.02
P_MUT = 0.15
MAX_GEN = 30
SIM_STEPS = 120                  # steps per simulation used for
fitness evaluation
ARENA_SIZE = 100.0               # square arena 0..ARENA_SIZE in
both axes
NEIGH_RADIUS = 8.0               # sensing radius for agent
behaviors
DT = 1.0                         # time step for movement update
MAX_SPEED = 2.0                  # cap agent speed

# ----- Utility helpers -----
def limit_vec(v, maxnorm):
    norm = np.linalg.norm(v)
    if norm > maxnorm and norm > 0:
        return v / norm * maxnorm
```

```
    return v

# ----- Genome -----
def random_genome():
    return np.random.uniform(-1, 1, size=(GENOME_DIM,))

# ----- Initialize population grid -----
population = np.array([[random_genome() for _ in range(GRID_W)] for
_ in range(GRID_H)])  # (H, W, GENOME_DIM)

# ----- Simulation to evaluate fitness of all agents together -----
def evaluate_population(pop):
    H, W, _ = pop.shape
    N = H * W
    positions = np.random.uniform(0, ARENA_SIZE, size=(N, 2))
    velocities = np.random.normal(0, 1, size=(N, 2))
    speeds = np.linalg.norm(velocities, axis=1, keepdims=True)
    speeds[speeds == 0] = 1.0
    velocities = (velocities / speeds) * 0.5

    genomes = pop.reshape((N, GENOME_DIM))
    goal = np.random.uniform(ARENA_SIZE*0.2, ARENA_SIZE*0.8,
size=(2,))

    dist_to_goal_history = np.zeros((N, SIM_STEPS))
    collision_counts = np.zeros(N)
    grid_bins = 20
    bin_size = ARENA_SIZE / grid_bins
    visited_bins = [set() for _ in range(N)]

    for t in range(SIM_STEPS):
        diffs = positions[:, None, :] - positions[None, :, :]
        dists = np.linalg.norm(diffs, axis=2) + np.eye(N) * 1e6
        neighbors_mask = dists < NEIGH_RADIUS

        align = np.zeros((N,2))
        cohesion = np.zeros((N,2))
        separation = np.zeros((N,2))
        to_goal = (goal - positions)

        for i in range(N):
            neigh_idx = np.where(neighbors_mask[i])[0]
            if neigh_idx.size > 0:
                align[i] = np.mean(velocities[neigh_idx], axis=0) -
velocities[i]
                cohesion[i] = (np.mean(positions[neigh_idx],
axis=0) - positions[i])
                close = np.where(dists[i] < (NEIGH_RADIUS *
```

```
0.5))[0]
                if close.size > 0:
                    sep = np.sum((positions[i] - positions[close])
/ (dists[i, close][:,None] + 1e-6), axis=0)
                    separation[i] = sep

        steer = (genomes[:,0:1] * align +
                 genomes[:,1:2] * cohesion +
                 genomes[:,2:3] * separation +
                 genomes[:,3:4] * to_goal)

        velocities += steer * DT * 0.05
        speeds = np.linalg.norm(velocities, axis=1, keepdims=True)
        too_fast = speeds[:,0] > MAX_SPEED
        if np.any(too_fast):
            velocities[too_fast] = (velocities[too_fast] /
speeds[too_fast]) * MAX_SPEED
        positions += velocities * DT
        positions = np.mod(positions, ARENA_SIZE)

        dist_to_goal = np.linalg.norm(positions - goal, axis=1)
        dist_to_goal_history[:, t] = dist_to_goal

        coll_pairs = np.where(dists < 1.0)
        for a, b in zip(*coll_pairs):
            if a < b:
                collision_counts[a] += 1
                collision_counts[b] += 1

        bins = np.floor(positions / bin_size).astype(int)
        bins = np.clip(bins, 0, grid_bins-1)
        for i in range(N):
            visited_bins[i].add((int(bins[i,0]), int(bins[i,1])))

    mean_dist = np.mean(dist_to_goal_history, axis=1)
    unique_bins = np.array([len(s) for s in visited_bins])

    max_possible = ARENA_SIZE * math.sqrt(2)
    norm_dist = 1.0 - (mean_dist / max_possible)            #
higher better
    norm_coll = 1.0 - (collision_counts / (SIM_STEPS + 1)) # higher
better
    norm_cov = unique_bins / (grid_bins * grid_bins)       # 0..1

    fitness = 0.5 * norm_dist + 0.3 * norm_cov + 0.2 * norm_coll
    fitness = np.clip(fitness, 0.0, 1.0)

    fitness_grid = fitness.reshape((H, W))
```

```
    diagnostics = {
        "mean_dist": mean_dist.reshape((H, W)),
        "collisions": collision_counts.reshape((H, W)),
        "coverage_bins": unique_bins.reshape((H, W)),
        "final_positions": positions.reshape((H, W, 2)),
        "goal": goal
    }
    return fitness_grid, diagnostics

# ----- Genetic operators -----
def mutate(genome, p_mut=P_MUT):
    g = genome.copy()
    for i in range(len(g)):
        if np.random.rand() < p_mut:
            g[i] += np.random.normal(0, 0.2)
    return np.clip(g, -1, 1)

def crossover(a, b):
    alpha = np.random.rand()
    child = alpha * a + (1-alpha) * b
    return np.clip(child, -1, 1)

def get_neighbors_coords(x, y, W=GRID_W, H=GRID_H):
    coords = []
    for dy in [-1,0,1]:
        for dx in [-1,0,1]:
            nx, ny = x + dx, y + dy
            if 0 <= nx < W and 0 <= ny < H and not (dx==0 and
dy==0):
                coords.append((nx, ny))
    return coords

# ----- Main PCOO loop -----
print("Name: PRANAV GAJANAN KAMATE\nUSN: 1BM23CS241")
best_over_time = []
avg_over_time = []

population_next = population.copy()

for gen in range(1, MAX_GEN+1):
    fitness_grid, diag = evaluate_population(population)
    avg_fit = float(np.mean(fitness_grid))
    best_fit = float(np.max(fitness_grid))
    best_over_time.append(best_fit)
    avg_over_time.append(avg_fit)
    print(f"Generation {gen}/{MAX_GEN}  avg_fit={avg_fit:.4f}
best_fit={best_fit:.4f}")
```

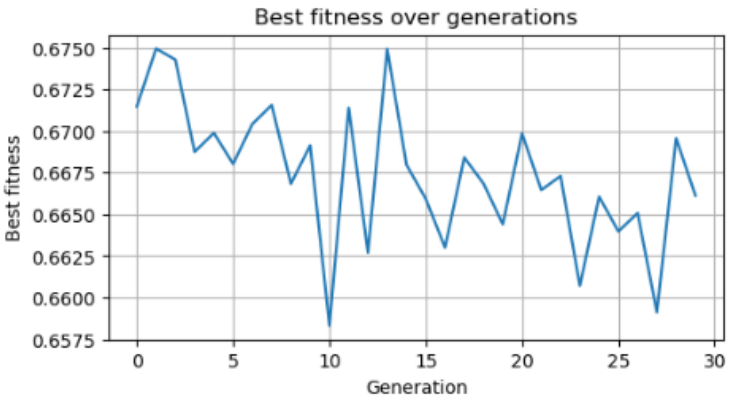
```
    H, W, _ = population.shape
    population_next = population.copy()

    for y in range(H):
        for x in range(W):
            Xi = population[y, x]
            neigh = get_neighbors_coords(x, y, W, H)
            if len(neigh) > 0:
                px, py = neigh[np.random.randint(len(neigh))]
                partner = population[py, px]
            else:
                partner = Xi.copy()

            if np.random.rand() < 0.6:
                child = crossover(Xi, partner)
            else:
                child = mutate(Xi, p_mut=P_MUT)

            # selection heuristic: move toward best neighbor if it
helps
            best_ng = Xi.copy()
            best_ng_fit = fitness_grid[y, x]
            for (nx, ny) in neigh:
                if fitness_grid[ny, nx] > best_ng_fit:
                    best_ng_fit = fitness_grid[ny, nx]
                    best_ng = population[ny, nx]

            dist_child_to_best = np.linalg.norm(child - best_ng)
            dist_x_to_best = np.linalg.norm(Xi - best_ng)
            if best_ng_fit > fitness_grid[y, x] and
dist_child_to_best < dist_x_to_best:
                population_next[y, x] = mutate(child, p_mut=0.05)
            else:
                if np.random.rand() < 0.02:
                    population_next[y, x] = mutate(Xi, p_mut=0.2)
                else:
                    population_next[y, x] = Xi

    population = population_next.copy()

# Final evaluation and results
final_fitness, final_diag = evaluate_population(population)
best_idx = np.unravel_index(np.argmax(final_fitness),
final_fitness.shape)
best_genome = population[best_idx]
best_fitness_val = final_fitness[best_idx]

print("\nBest organism found at grid position (row, col):",
```

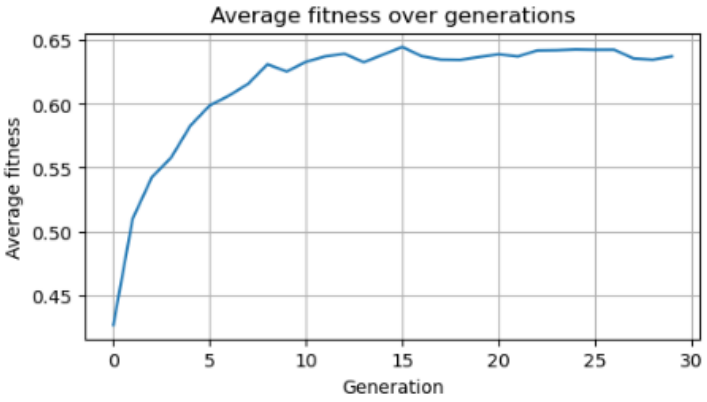
```
best_idx)
print("Best genome weights [align, cohesion, separation, goal]:",
np.round(best_genome, 3))
print("Best fitness:", float(best_fitness_val))

H, W, _ = population.shape
all_genomes = population.reshape((H*W, GENOME_DIM))
all_fitness = final_fitness.reshape((H*W,))
top_k = 8
top_idx = np.argsort(-all_fitness)[:top_k]
top_table = pd.DataFrame({
    "grid_pos": [f"{i//W},{i%W}" for i in top_idx],
    "fitness": np.round(all_fitness[top_idx], 4),
    "align": np.round(all_genomes[top_idx,0], 3),
    "cohere": np.round(all_genomes[top_idx,1], 3),
    "sep": np.round(all_genomes[top_idx,2], 3),
    "goal": np.round(all_genomes[top_idx,3], 3)
})

# Plotting
plt.figure(figsize=(6,3))
plt.plot(best_over_time)
plt.title("Best fitness over generations")
plt.xlabel("Generation")
plt.ylabel("Best fitness")
plt.grid(True)
plt.show()

plt.figure(figsize=(6,3))
plt.plot(avg_over_time)
plt.title("Average fitness over generations")
plt.xlabel("Generation")
plt.ylabel("Average fitness")
plt.grid(True)
plt.show()

final_positions = final_diag["final_positions"].reshape((-1,2))
goal = final_diag["goal"]
plt.figure(figsize=(5,5))
plt.scatter(final_positions[:,0], final_positions[:,1], s=12)
plt.scatter([goal[0]], [goal[1]], s=80, marker='*')
plt.title("Final agent positions and shared goal")
plt.xlim(0, ARENA_SIZE)
plt.ylim(0, ARENA_SIZE)
plt.gca().set_aspect('equal', adjustable='box')
plt.show()

# Save top genomes CSV
```

```
out_path = "top_genomes.csv"
top_table.to_csv(out_path, index=False)
print(f"\nTop genomes saved to {out_path}")
```

Output:

```
Name: PRANAV GAJANAN KAMATE
USN: 1BM23CS241
Generation 1/30  avg_fit=0.4264  best_fit=0.6715
Generation 2/30  avg_fit=0.5099  best_fit=0.6750
Generation 3/30  avg_fit=0.5426  best_fit=0.6743
Generation 4/30  avg_fit=0.5580  best_fit=0.6688
Generation 5/30  avg_fit=0.5831  best_fit=0.6699
Generation 6/30  avg_fit=0.5989  best_fit=0.6680
Generation 7/30  avg_fit=0.6067  best_fit=0.6704
Generation 8/30  avg_fit=0.6160  best_fit=0.6716
Generation 9/30  avg_fit=0.6313  best_fit=0.6668
Generation 10/30  avg_fit=0.6256  best_fit=0.6691
Generation 11/30  avg_fit=0.6333  best_fit=0.6583
Generation 12/30  avg_fit=0.6375  best_fit=0.6714
Generation 13/30  avg_fit=0.6395  best_fit=0.6627
Generation 14/30  avg_fit=0.6330  best_fit=0.6749
Generation 15/30  avg_fit=0.6389  best_fit=0.6680
Generation 16/30  avg_fit=0.6449  best_fit=0.6659
Generation 17/30  avg_fit=0.6377  best_fit=0.6630
Generation 18/30  avg_fit=0.6350  best_fit=0.6684
Generation 19/30  avg_fit=0.6348  best_fit=0.6668
Generation 20/30  avg_fit=0.6371  best_fit=0.6644
Generation 21/30  avg_fit=0.6392  best_fit=0.6699
Generation 22/30  avg_fit=0.6375  best_fit=0.6665
Generation 23/30  avg_fit=0.6422  best_fit=0.6673
Generation 24/30  avg_fit=0.6424  best_fit=0.6607
Generation 25/30  avg_fit=0.6430  best_fit=0.6661
Generation 26/30  avg_fit=0.6427  best_fit=0.6640
Generation 27/30  avg_fit=0.6427  best_fit=0.6651
Generation 28/30  avg_fit=0.6359  best_fit=0.6591
Generation 29/30  avg_fit=0.6349  best_fit=0.6696
Generation 30/30  avg_fit=0.6375  best_fit=0.6661

Best organism found at grid position (row, col): (0, 2)
Best genome weights [align, cohesion, separation, goal]: [-0.681 -0.128 -0.088  0.27 ]
Best fitness: 0.6624057447688273
```






```
Top genomes saved to top_genomes.csv
```