VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB RECORD

Bio Inspired Systems (23CS5BSBIS)

Submitted by

PRATHEEKSHA PAI (1BM23CS245)

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-2025 to Jan-2026

B.M.S. College of Engineering,
Bull Temple Road, Bangalore 560019
(Affiliated To Visvesvaraya Technological University, Belgaum)
Department of Computer Science and Engineering

CERTIFICATE

This is to certify that the Lab work entitled " Bio Inspired Systems (23CS5BSBIS)" carried out by **Pratheeksha Pai (1BM23CS245),** 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

Sl. No.	Date	Experiment Title	Page No.
1	28/08/2025	Genetic Algorithm for Optimization Problems	01-08
2	11/09/2025	Particle Swarm Optimization for Function Optimization	09-14
3	09/10/2025	Ant Colony Optimization for the Traveling Salesman Problem	15-23
4	16/10/2025	Cuckoo Search (CS)	24-28
5	23/10/2025	Grey Wolf Optimizer (GWO)	29-34
6	30/10/2025	Parallel Cellular Algorithms and Programs	35-41
7	04/09/2025	Optimization via Gene Expression Algorithms	42-54

Github Link:
https://github.com/Pratheeksha-pai/1BM23CS245_BIS

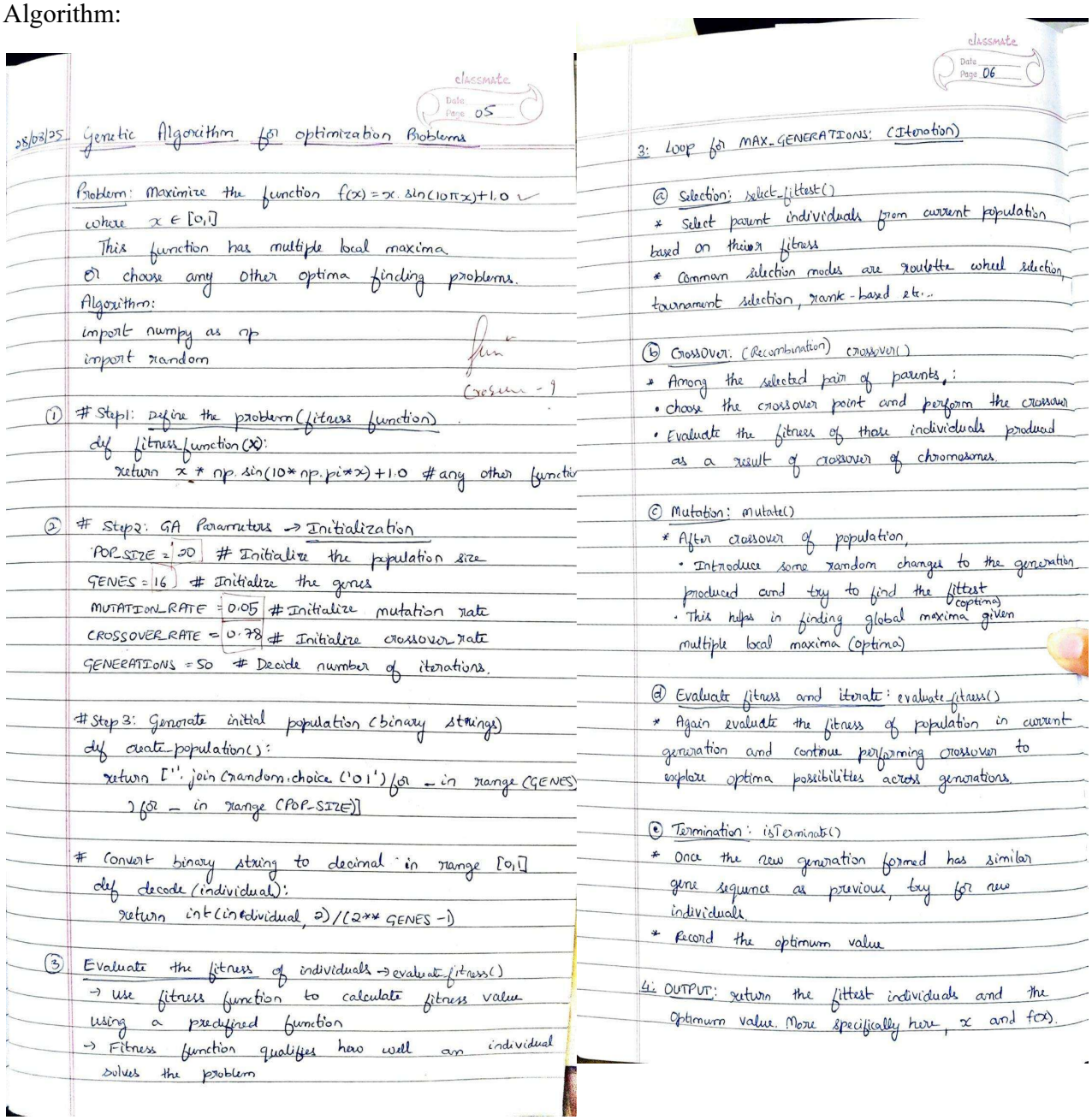
Program 1

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.

Application given: Star Constellation

Algorithm:



28/03/25 Genetic Algorithm for optimization Problems

Problem: Maximize the function $f(x) = x.\sin(10\pi x)+1.0$
where $x \in [0,1]$
This function has multiple local maxima
or choose any other optima finding problems.

Algorithm:
import numpy as np
import random

① # Step1: Define the problem (fitness function)
def fitness_function(x):
return x * np.sin(10*np.pi*x)+1.0 # any other functi

② # Step2: GA Parameters → Initialization
POP_SIZE = 20 # Initialize the population size
GENES = 16 # Initialize the genes
MUTATION_RATE = 0.05 # Initialize mutation rate
CROSSOVER_RATE = 0.78 # Initialize crossover rate
GENERATIONS = 50 # Decide number of iterations.

# Step 3: Generate initial population (binary strings)
def create_population():
return [''.join(random.choice('01') for _ in range(GENES)) for _ in range(POP_SIZE)]

# Convert binary string to decimal in range [0,1]
def decode(individual):
return int(individual, 2)/(2** GENES -1)

③ Evaluate the fitness of individuals → evaluate_fitness()
→ use fitness function to calculate fitness value using a predefined function
→ Fitness function qualifies how well an individual solves the problem

3: Loop for MAX_GENERATIONS: (Iteration)

ⓐ Selection: select_fittest()
* Select parent individuals from current population based on their fitness
* Common selection modes are roulette wheel selection, tournament selection, rank-based etc..

ⓑ Crossover: (Recombination) crossover()
* Among the selected pair of parents,:
• choose the crossover point and perform the crossover
• Evaluate the fitness of those individuals produced as a result of crossover of chromosomes.

ⓒ Mutation: mutate()
* After crossover of population,
• Introduce some random changes to the generation produced and try to find the fittest (optima)
• This helps in finding global maxima given multiple local maxima (optima)

ⓓ Evaluate fitness and iterate: evaluate_fitness()
* Again evaluate the fitness of population in current generation and continue performing crossover to explore optima possibilities across generations.

ⓔ Termination: isTerminate()
* Once the new generation formed has similar gene sequence as previous, try for new individuals
* Record the optimum value

4: OUTPUT: return the fittest individuals and the optimum value. More specifically here, x and f(x).

- Fun^n of Genetic Algorithm
- Execute WRT Star Constellation

Functions

① Fitness Function:

```
def ~~create~~ compute_fitness():
    # logic to evaluate individual based on optimization
      problem
    return fitness_score
```

② ~~def~~ Initialization function

```
def create_initial_population (population_size, solution_length):
    population = []
    # Generate random individual and add to
      population
    # create individual
      population.append (individual)
    return population
```

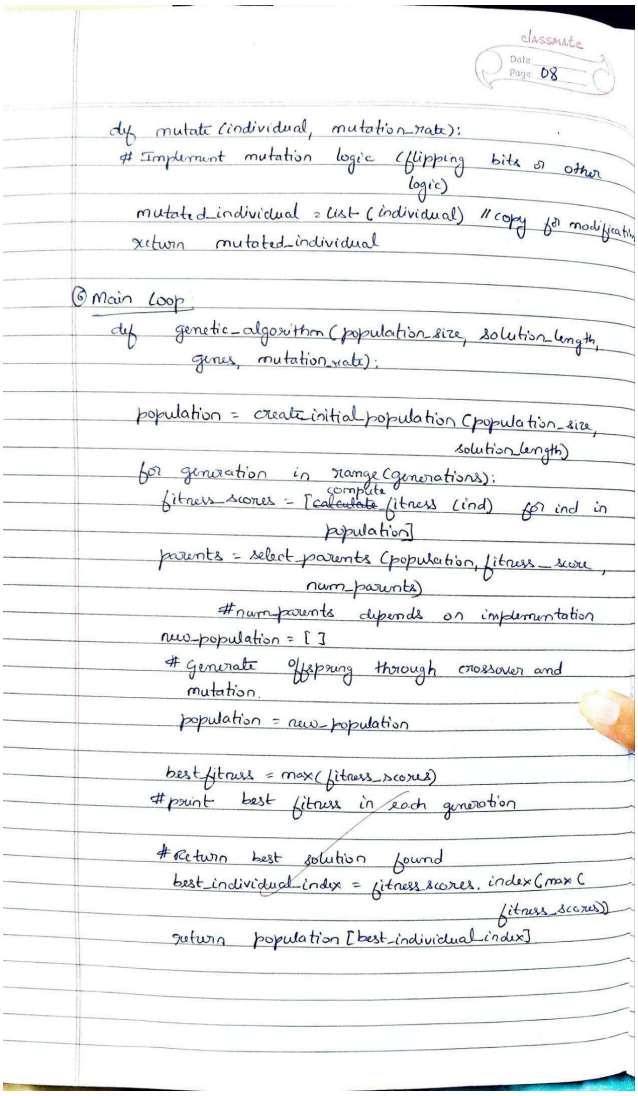
③ Selection function

```
def select_parents (population, fitness_score, num_parents):
    # implement selection logic (roulette wheel)
    parents = []
    # selection logic
    return parents.
```

④ Crossover function

```
def crossover (parent1, parent2):
    child1, child2 = [], []
    ~~return~~ # crossover logic
    return child1, child2
```

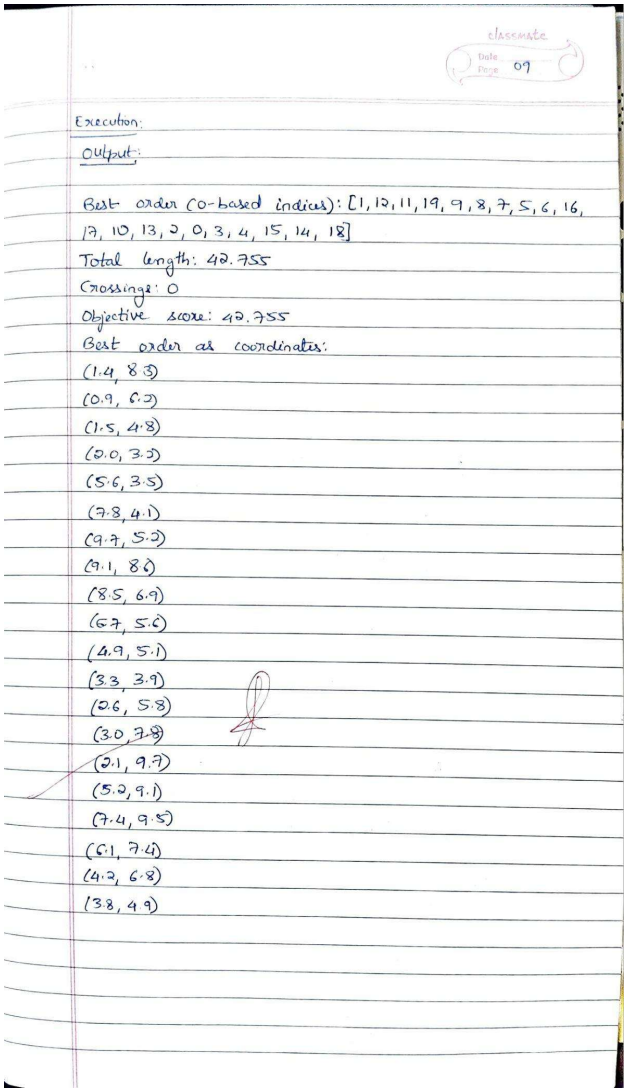
⑤ Mutation function



```
def mutate (individual, mutation_rate):
    # Implement mutation logic (flipping bits or other logic)
    mutated_individual = list (individual) # copy for modification
    return mutated_individual
```

⑥ Main Loop:

```
def genetic_algorithm (population_size, solution_length, genes, mutation_rate):

    population = create_initial_population (population_size, solution_length)
    for generation in range (generations):
        fitness_scores = [complete fitness (ind) for ind in population]
        parents = select_parents (population, fitness_score, num_parents)
            #num_parents depends on implementation
        new_population = [ ]
        # Generate offspring through crossover and mutation.
        population = new_population

        best_fitness = max(fitness_scores)
        #print best fitness in each generation

        #Return best solution found
        best_individual_index = fitness_scores. index(max(fitness_scores))
        return population [best_individual_index]
```



Execution:

Output:

Best order (0-based indices): [1, 12, 11, 19, 9, 8, 7, 5, 6, 16, 17, 10, 13, 2, 0, 3, 4, 15, 14, 18]
Total length: 42.755
Crossings: 0
Objective score: 42.755
Best order as coordinates:
(1.4, 8.3)
(0.9, 6.2)
(1.5, 4.8)
(2.0, 3.2)
(5.6, 3.5)
(7.8, 4.1)
(9.7, 5.2)
(9.1, 8.6)
(8.5, 6.9)
(6.7, 5.6)
(4.9, 5.1)
(3.3, 3.9)
(2.6, 5.8)
(3.0, 7.8)
(2.1, 9.7)
(5.2, 9.1)
(7.4, 9.5)
(6.1, 7.4)
(4.2, 6.8)
(3.8, 4.9)

Code:

```
# Star Constellation using genetic algorithm

import math
import random
from typing import List, Tuple
print("By: USN: 1BM23CS245, NAME: PRATHEEKSHA PAI")

Point = Tuple[float, float]
Chromosome = List[int]

# ---------- Geometry helpers ----------
def dist(a: Point, b: Point) -> float:
```

```
    return math.hypot(a[0]-b[0], a[1]-b[1])

def _orientation(p: Point, q: Point, r: Point) -> int:
    """Returns orientation of ordered triplet (p, q, r):
      0 -> colinear, 1 -> clockwise, 2 -> counterclockwise
    """
    val = (q[1]-p[1])*(r[0]-q[0]) - (q[0]-p[0])*(r[1]-q[1])
    if abs(val) < 1e-12:
        return 0
    return 1 if val > 0 else 2

def _on_segment(p: Point, q: Point, r: Point) -> bool:
    return (min(p[0], r[0]) <= q[0] <= max(p[0], r[0]) and
            min(p[1], r[1]) <= q[1] <= max(p[1], r[1]))

def segments_intersect(p1: Point, q1: Point, p2: Point, q2: Point) -> bool:
    """Proper/general intersection with colinear handling (excluding shared endpoints)."""
    # Exclude if sharing any endpoint exactly
    if p1 == p2 or p1 == q2 or q1 == p2 or q1 == q2:
        return False
    o1 = _orientation(p1, q1, p2)
    o2 = _orientation(p1, q1, q2)
    o3 = _orientation(p2, q2, p1)
    o4 = _orientation(p2, q2, q1)

    if o1 != o2 and o3 != o4:
        return True

    # Colinear special cases
    if o1 == 0 and _on_segment(p1, p2, q1): return True
    if o2 == 0 and _on_segment(p1, q2, q1): return True
    if o3 == 0 and _on_segment(p2, p1, q2): return True
    if o4 == 0 and _on_segment(p2, q1, q2): return True
    return False

# ---------- Fitness (length + crossings) ----------
def path_length(order: Chromosome, stars: List[Point], cycle: bool) -> float:
    total = 0.0
    for i in range(len(order)-1):
        total += dist(stars[order[i]], stars[order[i+1]])
    if cycle and len(order) > 1:
```

```
        total += dist(stars[order[-1]], stars[order[0]])
    return total

def count_crossings(order: Chromosome, stars: List[Point], cycle: bool) -> int:
    # Build the list of edges in this order
    edges = []
    for i in range(len(order)-1):
        edges.append((order[i], order[i+1]))
    if cycle and len(order) > 1:
        edges.append((order[-1], order[0]))

    crossings = 0
    for i in range(len(edges)):
        a1, a2 = edges[i]
        p1, q1 = stars[a1], stars[a2]
        for j in range(i+1, len(edges)):
            b1, b2 = edges[j]
            # Skip adjacent/overlapping edges (they share a vertex)
            if len({a1, a2, b1, b2}) < 4:
                continue
            p2, q2 = stars[b1], stars[b2]
            if segments_intersect(p1, q1, p2, q2):
                crossings += 1
    return crossings

def fitness(order: Chromosome, stars: List[Point], crossing_weight: float, cycle: bool) -> float:
    """Lower is better."""
    L = path_length(order, stars, cycle)
    X = count_crossings(order, stars, cycle)
    return L + crossing_weight * X

# ---------- GA operators ----------
def tournament_selection(pop: List[Chromosome], stars: List[Point], k: int, crossing_weight: float,
cycle: bool) -> Chromosome:
    """Pick best of k random individuals (min fitness)."""
    contenders = random.sample(pop, k)
    contenders.sort(key=lambda c: fitness(c, stars, crossing_weight, cycle))
    return contenders[0][:]

def order_crossover(parent1: Chromosome, parent2: Chromosome) -> Chromosome:
    """OX (Order Crossover) for permutations."""
```

```
    n = len(parent1)
    a, b = sorted(random.sample(range(n), 2))
    child = [-1] * n
    # Copy slice from parent1
    child[a:b+1] = parent1[a:b+1]
    # Fill remaining from parent2 preserving order
    p2_idx = 0
    for i in range(n):
        if child[i] == -1:
            while parent2[p2_idx] in child:
                p2_idx += 1
            child[i] = parent2[p2_idx]
            p2_idx += 1
    return child

def swap_mutation(ch: Chromosome, mutation_rate: float) -> None:
    n = len(ch)
    if random.random() < mutation_rate:
        i, j = random.sample(range(n), 2)
        ch[i], ch[j] = ch[j], ch[i]

# ---------- GA driver ----------
def run_ga(
    stars: List[Point],
    population_size: int = 200,
    generations: int = 500,
    tournament_k: int = 4,
    crossover_rate: float = 0.9,
    mutation_rate: float = 0.2,
    elitism: int = 4,
    crossing_weight: float = 50.0,
    cycle: bool = True,
    seed: int = 42
):
    """Returns best_order, best_fitness, history (list of best fitness by generation)."""
    random.seed(seed)
    n = len(stars)
    if n < 2:
        raise ValueError("Need at least 2 stars.")
    # Initialize population with random permutations
    base = list(range(n))
```

```
    population = [random.sample(base, n) for _ in range(population_size)]

    # Evaluate initial best
    def fit(ch): return fitness(ch, stars, crossing_weight, cycle)
    population.sort(key=fit)
    best = population[0][:]
    best_f = fit(best)
    history = [best_f]

    for gen in range(generations):
        new_pop: List[Chromosome] = []
        # Elitism
        elites = population[:elitism]
        new_pop.extend([e[:] for e in elites])

        # Fill rest with offspring
        while len(new_pop) < population_size:
            p1 = tournament_selection(population, stars, tournament_k, crossing_weight, cycle)
            p2 = tournament_selection(population, stars, tournament_k, crossing_weight, cycle)
            if random.random() < crossover_rate:
                child = order_crossover(p1, p2)
            else:
                child = p1[:]
            swap_mutation(child, mutation_rate)
            new_pop.append(child)

        population = new_pop
        population.sort(key=fit)

        # Track best
        if fit(population[0]) < best_f:
            best = population[0][:]
            best_f = fit(best)

        history.append(best_f)

    return best, best_f, history

# ---------- Example usage ----------
if __name__ == "__main__":
    # Example: a synthetic "sky" of 20 stars (you can replace with real data)
```

```
stars = [
    (2.1, 9.7), (1.4, 8.3), (3.0, 7.8), (5.2, 9.1), (7.4, 9.5),
    (9.1, 8.6), (8.5, 6.9), (9.7, 5.2), (7.8, 4.1), (5.6, 3.5),
    (3.3, 3.9), (1.5, 4.8), (0.9, 6.2), (2.6, 5.8), (4.2, 6.8),
    (6.1, 7.4), (6.7, 5.6), (4.9, 5.1), (3.8, 4.9), (2.0, 3.2),
]

# Tune crossing_weight:
#   higher -> prioritizes fewer crossings (good for "constellation-like" lines)
#   lower  -> prioritizes shorter total length
best_order, best_score, history = run_ga(
    stars,
    population_size=80,   # instead of 200
    generations=150,      # instead of 600
    crossing_weight=50.0,
    cycle=False,
    seed=123
)
# Report
total_len = path_length(best_order, stars, cycle=False)
crossings = count_crossings(best_order, stars, cycle=False)
print("Best order (0-based indices):", best_order)
print(f"Total length: {total_len:.3f}")
print(f"Crossings: {crossings}")
print(f"Objective score: {best_score:.3f}")
print("Best order as coordinates:")
for idx in best_order:
    print(stars[idx])
```

Output:

```
By: USN: 1BM23CS245, NAME: PRATHEEKSHA PAI
Best order (0-based indices): [1, 12, 11, 19, 9, 8, 7, 5, 6, 16, 17, 10, 13, 2, 0, 3, 4, 15, 14, 18]
Total length: 42.755
Crossings: 0
Objective score: 42.755
Best order as coordinates:
(1.4, 8.3)
(0.9, 6.2)
(1.5, 4.8)
(2.0, 3.2)
(5.6, 3.5)
(7.8, 4.1)
(9.7, 5.2)
(9.1, 8.6)
(8.5, 6.9)
(6.7, 5.6)
(4.9, 5.1)
(3.3, 3.9)
(2.6, 5.8)
(3.0, 7.8)
(2.1, 9.7)
(5.2, 9.1)
(7.4, 9.5)
(6.1, 7.4)
(4.2, 6.8)
(3.8, 4.9)
```

Program 2

Particle Swarm Optimization for Function Optimization:

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

Application given: Data Clustering, research paper analysis

Algorithm:

classmate
Date
Page 16

11/09/25 Particle Swarm optimization:

Pseudocode:

Function PSO (objective_function, max_iterations, num_of_particles, ~~swarm~~ search_space_min, search_space_max):

1. Initialization:

Inputs:
objective_function → function to be optimized (minimized / maximized)
max_iterations → maximum no. of iterations
num_of_particles → number of particles in swarm
search_space_min, search_space_max → bounds for finding solution.

Parameters:
w = Inertia weight (generally, 0.9→0.4, decreases for converging to optimal solution)
c1 = cognitive coefficient (self-confidence parameter of particles)
c2 = social coefficient (tendency to follow best particle of group)
Generally, c1 = c2 = 2.0

Initialization loop:
for i → 1 to num_of_particles do
particle[i] position = random_vector(search_space_min, search_space_max)
particle[i].velocity = random_vector(-|search_space_max - search_space_min|, |search_space_max - search_space_min|)
particle[i].personal_best_position = particle[i].position
particle[i].personal_best_fitness = objective_function(particle[i].position)
END for

global_best_position = best among all particle_best_positions.

global_best_fitness = objective_function(global_best_position)

2. Main loop

for iterations from 1 to max_iterations do
  for each particle i in swarm do:

    r1 = random(0,1)
    r2 = random(0,1)

    // update velocity
    particle[i].velocity = (w * particle[i].velocity) + (r1 * c1 * (particle[i].personal_best_position - particle[i].position) + (r2 * c2 * global_best_position - particle[i].position)

    // update position
    particle[i].position = particle[i].position + particle[i].velocity

    // ~~Evaluate~~ clamp within search space
    particle[i].position = clamp(particle[i].position, search_space_min, search_space_max)

    // Evaluate fitness
    curr_fitness = objective_function(particle[i].position)

    // update particle best position
    if (curr_fitness < particle[i].personal_best_fitness)
      particle[i].personal_best_fitness = curr_fitness
      particle[i].personal_best_position = particle[i].position

    End if

    // update global best position

    if (curr_fitness < global_best_fitness)
      global_best_fitness = curr_fitness
      global_best_position = particle[i].personal_best_position

    End if
  End for
End for

3. Output
  return global_best_position, global_best_fitness

End function

Applications
* Mathematical function optimization for non-linear functions since PSO doesn't need gradient
* Blackbox function optimizations in AI
* Routing applications → finding best path, like swarm of particles finding food

- Define gbest, pbest, fitness fn
- Execute in data clustering.

① gbest: Global best
Represents the best position (set of parameters) found by any particle in the entire swarm upto the current iteration. It acts like global guide, directing the swarm towards promising regions of the search space

pbest: personal best
Each individual particle in the swarm maintains a record of its best position.

Fitness function:
The function to evaluate ~~how~~ the quality of a potential solution (particle's position) in the search space). It might measure a performance or efficiency

(e) PSO in data clustring:

Output:
Global Best Fitness: 363.90915911
Final Centroids:
[[-8.80039661, 7.02920[
[-6.76269 -6.86916]
[-2.738664 8.955942]
[4.6214778 2.09540025]]

make_blobs() generates the dataset
- it creates 300 2D points grouped into 4 true clusters
- commonly used for testing clustering algorithms

Given random number of scattered dots on 2D plane, we want to group them into k (say 2) clusters. PSO finds the best position of 2 centroids so that:
- each point is assigned to its nearest centroid
- Total distance from points to their centroid is minimized

Research paper. data clustring. + PSO → (1) old (2) latest

Old research papers: 2011 - Springer
* Focused on combining PSO with classical methods like k-means to improve cluster quality and avoid local minima
* Showed PSO often gives better cluster accuracy than k-means but at the cost of higher runtime
* Limited scalability - usually tested on small to medium datasets

Recent: Application of PSO (2025)
* Introduced hybrid metaheuristics (Eg: PSO + Gray wolf optimization, fuzzy entropy) for better global search and noise handling
* Added adaptive PSO variants that tune parameters dynamically for better convergence and robustness
* Targeted large scale, high dimensional datasets

Code:
```
# Data Clustering application

import numpy as np
from sklearn.metrics import pairwise_distances_argmin_min

print("By: USN: 1BM23CS245, NAME: PRATHEEKSHA PAI")

class Particle:
    def __init__(self, n_clusters, n_features, data_min, data_max):
        # Position represents the cluster centroids
        self.position = np.random.uniform(data_min, data_max, (n_clusters, n_features))
        self.velocity = np.random.uniform(-1, 1, (n_clusters, n_features))
```

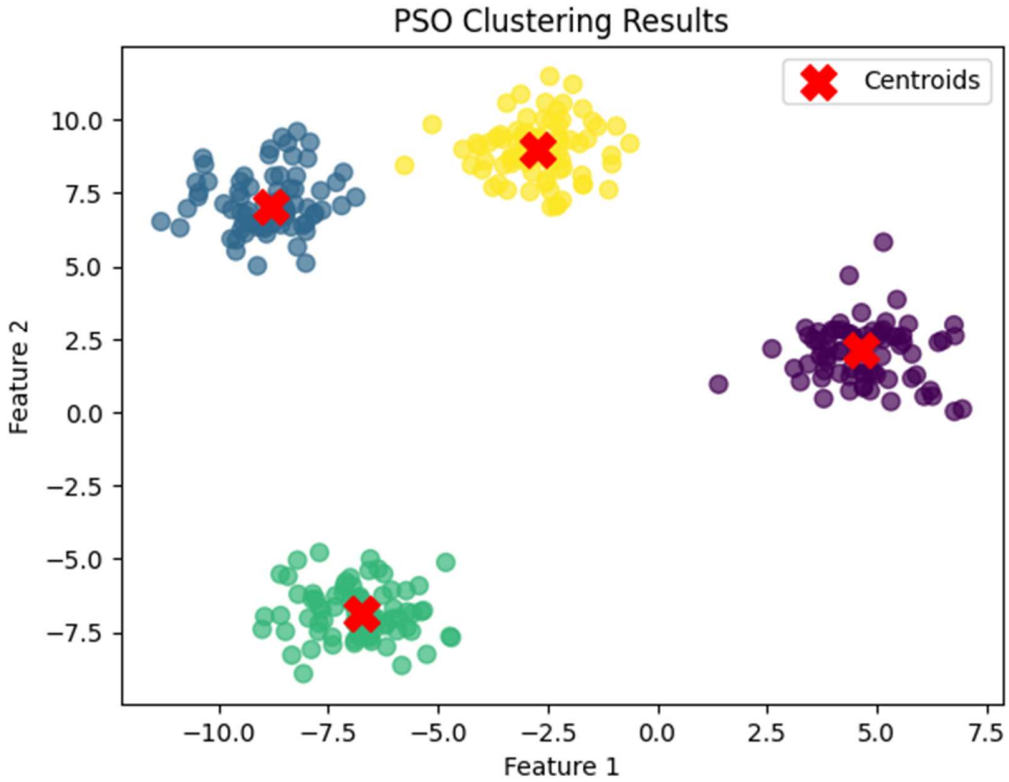
```
        self.best_position = np.copy(self.position)
        self.best_fitness = float('inf')

class PSOClustering:
    def __init__(self, data, n_clusters, n_particles=30, max_iter=100, w=0.7, c1=1.5, c2=1.5):
        self.data = data
        self.n_clusters = n_clusters
        self.n_particles = n_particles
        self.max_iter = max_iter
        self.w = w  # Inertia weight
        self.c1 = c1  # Cognitive component (personal best)
        self.c2 = c2  # Social component (global best)

        self.n_features = data.shape[1]
        self.data_min = np.min(data, axis=0)
        self.data_max = np.max(data, axis=0)

        self.particles = [Particle(n_clusters, self.n_features, self.data_min, self.data_max) for _ in
range(n_particles)]
        self.global_best_position = None
        self.global_best_fitness = float('inf')

    def _calculate_fitness(self, centroids):
        """Calculates the sum of squared distances for a given set of centroids."""
        labels, distances = pairwise_distances_argmin_min(self.data, centroids)
        return np.sum(distances)

    def run(self):
        for _ in range(self.max_iter):
            for particle in self.particles:
                # Calculate fitness for current position
                current_fitness = self._calculate_fitness(particle.position)

                # Update personal best
                if current_fitness < particle.best_fitness:
                    particle.best_fitness = current_fitness
                    particle.best_position = np.copy(particle.position)

                # Update global best
                if current_fitness < self.global_best_fitness:
                    self.global_best_fitness = current_fitness
```

```
                self.global_best_position = np.copy(particle.position)

            # Update particle velocities and positions
            for particle in self.particles:
                r1 = np.random.rand(self.n_clusters, self.n_features)
                r2 = np.random.rand(self.n_clusters, self.n_features)

                cognitive_component = self.c1 * r1 * (particle.best_position - particle.position)
                social_component = self.c2 * r2 * (self.global_best_position - particle.position)

                particle.velocity = self.w * particle.velocity + cognitive_component + social_component
                particle.position += particle.velocity

                # Ensure centroids stay within data bounds (optional but can help stability)
                particle.position = np.clip(particle.position, self.data_min, self.data_max)

        # Assign final clusters based on the global best centroids
        final_labels, _ = pairwise_distances_argmin_min(self.data, self.global_best_position)
        return final_labels, self.global_best_position, self.global_best_fitness

# Example Usage:
if __name__ == "__main__":
    from sklearn.datasets import make_blobs
    import matplotlib.pyplot as plt

    # Generate sample data
    X, _ = make_blobs(n_samples=300, centers=4, random_state=42)

    # Initialize and run PSO clustering
    pso_clustering = PSOClustering(data=X, n_clusters=4, n_particles=50, max_iter=200)
    labels, centroids, fitness = pso_clustering.run()

    print(f"Global Best Fitness: {fitness}")
    print(f"Final Centroids:\n{centroids}")

    # Visualize the results
    plt.scatter(X[:, 0], X[:, 1], c=labels, cmap='viridis', s=50, alpha=0.7)
    plt.scatter(centroids[:, 0], centroids[:, 1], c='red', marker='X', s=200, label='Centroids')
    plt.title("PSO Clustering Results")
    plt.xlabel("Feature 1")
    plt.ylabel("Feature 2")
```

```
plt.legend()
plt.show()
```

Output:

```
By: USN: 1BM23CS245, NAME: PRATHEEKSHA PAI
Global Best Fitness: 363.90915911838886
Final Centroids:
[[ 4.62147764  2.09540232]
 [-8.8003266   7.02920621]
 [-6.76269163 -6.8691614 ]
 [-2.73866472  8.95529407]]
```



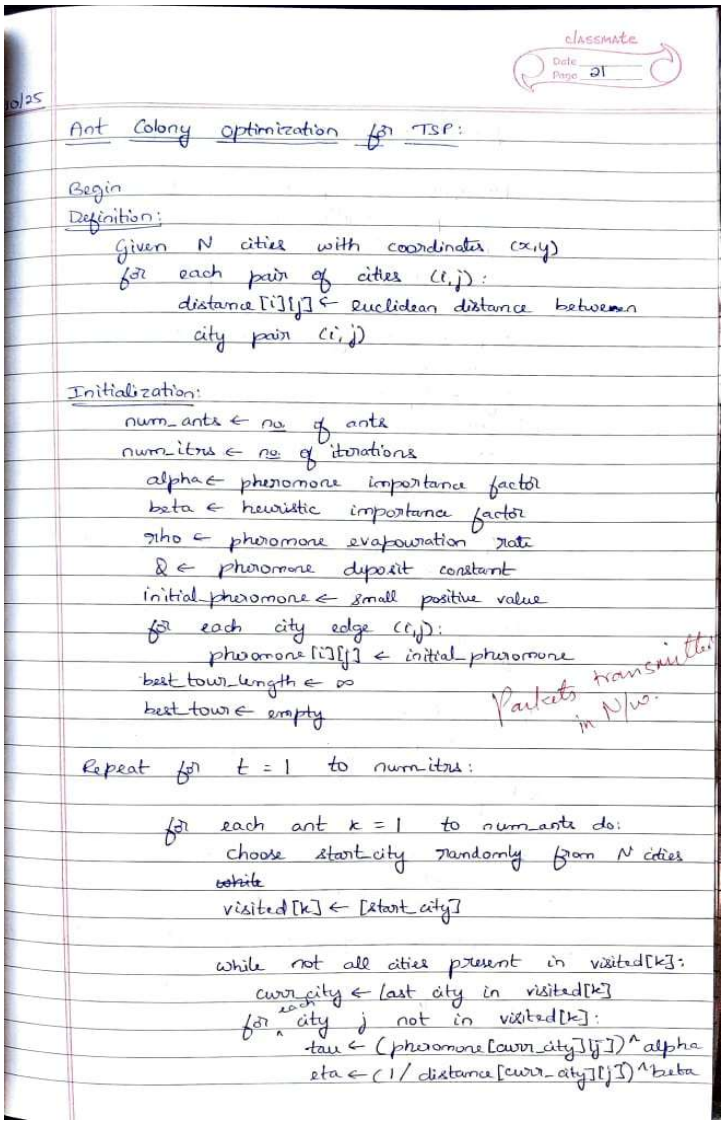
Program 3

Ant Colony Optimization for the Traveling Salesman Problem:

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

Application given: Packet transition in Network, Travelling Salesman Problem

Algorithm:



```
Ant Colony optimization for TSP:

Begin
Definition:
    Given N cities with coordinates (x,y)
    for each pair of cities (i,j):
        distance[i][j] ← euclidean distance between
        city pair (i,j)

Initialization:
    num_ants ← no. of ants
    num_itrs ← no. of iterations
    alpha ← pheromone importance factor
    beta ← heuristic importance factor
    rho ← pheromone evaporation rate
    Q ← pheromone deposit constant
    initial_pheromone ← small positive value
    for each city edge (i,j):
        pheromone[i][j] ← initial_pheromone
    best_tour_length ← ∞
    best_tour ← empty

Repeat for t = 1 to num_itrs:

    for each ant k = 1 to num_ants do:
        choose start_city randomly from N cities
        ~~while~~
        visited[k] ← [start_city]

        while not all cities present in visited[k]:
            curr_city ← last city in visited[k]
            for each city j not in visited[k]:
                tau ← (pheromone[curr_city][j])^alpha
                eta ← (1/distance[curr_city][j])^beta
```

Packets transmitted in N/w.

probability[j] ← eta * tau

Normalize all probabilities so that
sum(probabilities) = 1
choose next_city randomly from cities not
in visited[k], based on probability[j]

Append next_city to visited[k]
end ~~while~~ for
~~If best_path~~
end while
tour_length[k] ← distance between consecutive
cities in visited[k]

If tour_length[k] < best_tour_length then:
best_tour_length ← tour_length[k]
best_tour ← visited[k]
End If
End for

update pheromone
For each pair of edges (i,j) in cities:
pheromone[i][j] = (1 - rho) * pheromone[i][j]
End for

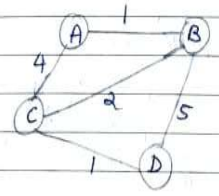
For each ant k ← 1 to num_ants:
for each city pair (i,j) visited during kth
ant's tour:
pheromone[i][j] = pheromone[i][j] + (Q/tour_length[k])

end for
end for
end repeat

output → best_tour, best_tour_length

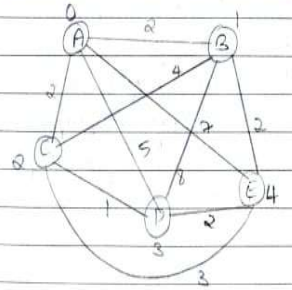
End

Application:
Packets transition in network: A to D



Iteration 1, Ant path: ['A', 'B', 'C', 'D']
Iteration 1, Ant path: ['A', 'B', 'C', 'D']
Iteration 1, Ant path: ['A', 'C', 'D']
Iteration 2, Ant path: ['A', 'B', 'C', 'D']
⋮
Iteration 10, Ant path: ['A', 'B', 'C', 'D']
Best path after 10 iterations: ['A', 'B', 'C', 'D']

Travelling salesman Problem using ACO:



Shortest path: ([(0,2), (2,3), (3,4), (4,1), (1,0)], 9.0)
distance = 2 + 1 + 2 + 2 + 2 = 9

Code:

```
# Applied to packet transition in network

import random

print("By: USN: 1BM23CS245, NAME: PRATHEEKSHA PAI")

class NetworkACO:
    def __init__(self, graph, alpha=1.0, beta=2.0, evaporation_rate=0.5, Q=100):
        """
        graph: dict of node: {neighbor: weight}
        alpha: pheromone importance
        beta: heuristic importance (inverse of weight)
        evaporation_rate: pheromone evaporation
        Q: pheromone deposit factor
        """
        self.graph = graph
        self.nodes = list(graph.keys())
        self.alpha = alpha
        self.beta = beta
        self.evaporation_rate = evaporation_rate
        self.Q = Q

        # Initialize pheromone levels on edges
        self.pheromone = {}
        for node in graph:
            for neighbor in graph[node]:
                self.pheromone[(node, neighbor)] = 1.0  # initial pheromone

    def _heuristic(self, node1, node2):
        # Inverse of the edge weight (distance or cost)
        return 1.0 / self.graph[node1][node2]

    def _probabilities(self, current_node, visited):
        neighbors = self.graph[current_node].keys()
        probs = []
        total = 0.0

        for neighbor in neighbors:
            if neighbor not in visited:
                pheromone = self.pheromone[(current_node, neighbor)] ** self.alpha
```

```
            heuristic = self._heuristic(current_node, neighbor) ** self.beta
            prob = pheromone * heuristic
            probs.append((neighbor, prob))
            total += prob

    # Normalize probabilities
    if total == 0:
        return []
    return [(node, prob / total) for node, prob in probs]

def _select_next_node(self, probabilities):
    r = random.random()
    cumulative = 0.0
    for node, prob in probabilities:
        cumulative += prob
        if r <= cumulative:
            return node
    return probabilities[-1][0]

def find_path(self, start, end):
    path = [start]
    visited = set(path)

    while path[-1] != end:
        current = path[-1]
        probabilities = self._probabilities(current, visited)
        if not probabilities:
            # Dead end, no unvisited neighbors
            return None
        next_node = self._select_next_node(probabilities)
        path.append(next_node)
        visited.add(next_node)

    return path

def _path_length(self, path):
    length = 0.0
    for i in range(len(path) - 1):
        length += self.graph[path[i]][path[i+1]]
    return length
```

```
    def update_pheromones(self, paths):
        # Evaporation
        for edge in self.pheromone:
            self.pheromone[edge] *= (1 - self.evaporation_rate)
            if self.pheromone[edge] < 0.1:
                self.pheromone[edge] = 0.1  # minimum pheromone

        # Deposit
        for path in paths:
            if path is None:
                continue
            length = self._path_length(path)
            deposit_amount = self.Q / length
            for i in range(len(path) - 1):
                edge = (path[i], path[i+1])
                self.pheromone[edge] += deposit_amount

def main():
    # Example network graph: node -> {neighbor: cost}
    graph = {
        'A': {'B': 1, 'C': 4},
        'B': {'A': 1, 'C': 2, 'D': 5},
        'C': {'A': 4, 'B': 2, 'D': 1},
        'D': {'B': 5, 'C': 1}
    }

    aco = NetworkACO(graph)

    start = 'A'
    end = 'D'
    iterations = 10
    ants_per_iteration = 5

    for i in range(iterations):
        paths = []
        for _ in range(ants_per_iteration):
            path = aco.find_path(start, end)
            if path:
                print(f"Iteration {i+1}, Ant path: {path}")
            paths.append(path)
        aco.update_pheromones(paths)
```

```
    # After iterations, find best path based on pheromones
    best_path = aco.find_path(start, end)
    print(f"Best path after {iterations} iterations: {best_path}")

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

Output:

```
By: USN: 1BM23CS245, NAME: PRATHEEKSHA PAI
Iteration 1, Ant path: ['A', 'B', 'C', 'D']
Iteration 1, Ant path: ['A', 'B', 'C', 'D']
Iteration 1, Ant path: ['A', 'B', 'C', 'D']
Iteration 1, Ant path: ['A', 'B', 'C', 'D']
Iteration 1, Ant path: ['A', 'B', 'C', 'D']
Iteration 2, Ant path: ['A', 'B', 'C', 'D']
Iteration 2, Ant path: ['A', 'B', 'C', 'D']
Iteration 2, Ant path: ['A', 'B', 'C', 'D']
Iteration 2, Ant path: ['A', 'B', 'C', 'D']
Iteration 2, Ant path: ['A', 'B', 'C', 'D']
Iteration 3, Ant path: ['A', 'B', 'C', 'D']
Iteration 3, Ant path: ['A', 'B', 'C', 'D']
Iteration 3, Ant path: ['A', 'B', 'C', 'D']
Iteration 3, Ant path: ['A', 'B', 'C', 'D']
Iteration 3, Ant path: ['A', 'B', 'C', 'D']
Iteration 4, Ant path: ['A', 'B', 'C', 'D']
Iteration 4, Ant path: ['A', 'B', 'C', 'D']
Iteration 4, Ant path: ['A', 'B', 'C', 'D']
Iteration 4, Ant path: ['A', 'B', 'C', 'D']
Iteration 4, Ant path: ['A', 'B', 'C', 'D']
Iteration 5, Ant path: ['A', 'B', 'C', 'D']
Iteration 5, Ant path: ['A', 'B', 'C', 'D']
Iteration 5, Ant path: ['A', 'B', 'C', 'D']
Iteration 5, Ant path: ['A', 'B', 'C', 'D']
Iteration 5, Ant path: ['A', 'B', 'C', 'D']
Iteration 6, Ant path: ['A', 'B', 'C', 'D']
Iteration 6, Ant path: ['A', 'B', 'C', 'D']
Iteration 6, Ant path: ['A', 'B', 'C', 'D']
Iteration 6, Ant path: ['A', 'B', 'C', 'D']
Iteration 6, Ant path: ['A', 'B', 'C', 'D']
Iteration 7, Ant path: ['A', 'B', 'C', 'D']
Iteration 7, Ant path: ['A', 'B', 'C', 'D']
```

```
Iteration 7, Ant path: ['A', 'B', 'C', 'D']
Iteration 7, Ant path: ['A', 'B', 'C', 'D']
Iteration 7, Ant path: ['A', 'B', 'C', 'D']
Iteration 8, Ant path: ['A', 'B', 'C', 'D']
Iteration 8, Ant path: ['A', 'B', 'C', 'D']
Iteration 8, Ant path: ['A', 'B', 'C', 'D']
Iteration 8, Ant path: ['A', 'B', 'C', 'D']
Iteration 8, Ant path: ['A', 'B', 'C', 'D']
Iteration 9, Ant path: ['A', 'B', 'C', 'D']
Iteration 9, Ant path: ['A', 'B', 'C', 'D']
Iteration 9, Ant path: ['A', 'B', 'C', 'D']
Iteration 9, Ant path: ['A', 'B', 'C', 'D']
Iteration 9, Ant path: ['A', 'B', 'C', 'D']
Iteration 10, Ant path: ['A', 'B', 'C', 'D']
Iteration 10, Ant path: ['A', 'B', 'C', 'D']
Iteration 10, Ant path: ['A', 'B', 'C', 'D']
Iteration 10, Ant path: ['A', 'B', 'C', 'D']
Iteration 10, Ant path: ['A', 'B', 'C', 'D']
Best path after 10 iterations: ['A', 'B', 'C', 'D']
```

```
# Applied to Travelling Salesman Problem:
import numpy as np

print('USN: 1BM23CS245, NAME: PRATHEEKSHA PAI')

class AntColony:
    def __init__(self, distances, n_ants, n_best, n_iterations, decay, alpha=1, beta=1):
        """
        distances: 2D numpy array of distances between nodes
        n_ants: number of ants per iteration
        n_best: number of best ants who deposit pheromone
        n_iterations: number of iterations to run
        decay: rate of pheromone evaporation
        alpha: pheromone importance
        beta: distance importance
        """
        self.distances = distances
        self.pheromone = np.ones(self.distances.shape) / len(distances)
        self.all_inds = range(len(distances))
        self.n_ants = n_ants
        self.n_best = n_best
        self.n_iterations = n_iterations
```

```
        self.decay = decay
        self.alpha = alpha
        self.beta = beta

    def run(self):
        shortest_path = None
        all_time_shortest_path = ("placeholder", np.inf)
        for _ in range(self.n_iterations):
            all_paths = self.gen_all_paths()
            self.spread_pheronome(all_paths, self.n_best, shortest_path=shortest_path)
            shortest_path = min(all_paths, key=lambda x: x[1])
            if shortest_path[1] < all_time_shortest_path[1]:
                all_time_shortest_path = shortest_path
            self.pheromone *= (1 - self.decay)

        # Convert NumPy data types to native Python types for readability
        clean_path = [(int(a), int(b)) for a, b in all_time_shortest_path[0]]
        clean_distance = float(all_time_shortest_path[1])
        return (clean_path, clean_distance)

    def spread_pheronome(self, all_paths, n_best, shortest_path):
        sorted_paths = sorted(all_paths, key=lambda x: x[1])
        for path, dist in sorted_paths[:n_best]:
            for move in path:
                self.pheromone[move] += 1.0 / self.distances[move]

    def gen_path_dist(self, path):
        total_dist = 0
        for move in path:
            total_dist += self.distances[move]
        return total_dist

    def gen_all_paths(self):
        all_paths = []
        for _ in range(self.n_ants):
            path = self.gen_path(0)  # start from node 0
            total_dist = self.gen_path_dist(path)
            all_paths.append((path, total_dist))
        return all_paths

    def gen_path(self, start):
```

```
        path = []
        visited = set([start])
        prev = start
        for _ in range(len(self.distances) - 1):
            move = self.pick_move(self.pheromone[prev], self.distances[prev], visited)
            path.append((prev, move))
            prev = move
            visited.add(move)
        path.append((prev, start))  # return to start
        return path

    def pick_move(self, pheromone, distances, visited):
        pheromone = np.copy(pheromone)
        pheromone[list(visited)] = 0
        row = pheromone ** self.alpha * ((1.0 / distances) ** self.beta)
        norm_row = row / row.sum()
        move = np.random.choice(self.all_inds, 1, p=norm_row)[0]
        return move


if __name__ == "__main__":
    # Example: 5 cities with distances
    distances = np.array([
        [np.inf, 2, 2, 5, 7],
        [2, np.inf, 4, 8, 2],
        [2, 4, np.inf, 1, 3],
        [5, 8, 1, np.inf, 2],
        [7, 2, 3, 2, np.inf]
    ])

    ant_colony = AntColony(distances, n_ants=10, n_best=3, n_iterations=100, decay=0.1, alpha=1,
beta=2)
    shortest_path = ant_colony.run()
    print("Shortest path:", shortest_path)
```

Output:

```
USN: 1BM23CS245, NAME: PRATHEEKSHA PAI
Shortest path: ([(0, 1), (1, 4), (4, 3), (3, 2), (2, 0)], 9.0)
```

Program 4

Cuckoo Search (CS):

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

Algorithm:

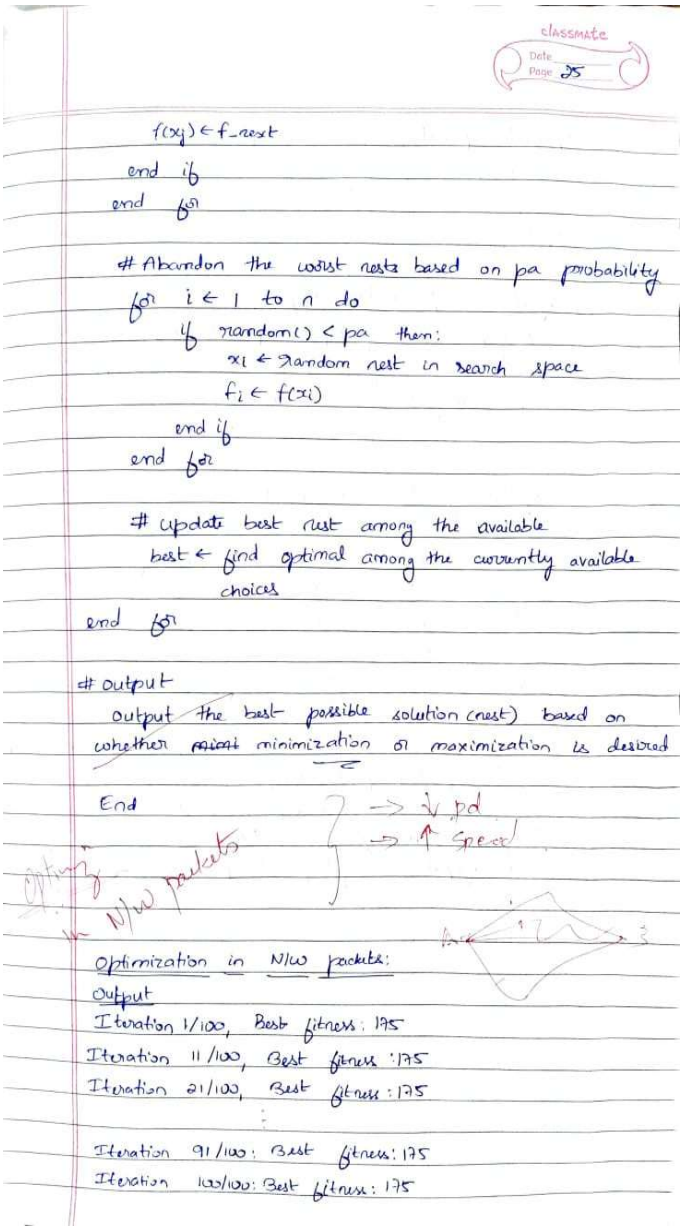
classmate

Date
Page 24

16/10/25

Cuckoo search:

```
Begin
# Define the optimization problem
    define the objective function f(x), x = (x1, x2, ..., xd)
    where d = no. of dimensions

# Initialize the algorithm parameters
  n ← no. of nests (population size)
  pa ← distribution probability (probability of host bird
                                 finding alien egg in its nest)
  maxitrs ← maximum no. of iterations

# Initialize the population
    for i ← 1 to n do
        xi ← random nest from within search space bound
        fi ← f(xi) //evaluate fitness
    end for
  #Find best solution
    best ← optimal nest among all possibilities (minimize/
           maximize)

# main loop
  for t = 1 to maxitrs:
        # find all best nests (explore)
        for i ← 1 to n do
          x_next = xi + α * Lévy(λ)  #α → scaling factor
                   # Lévy(λ) → jumps for exploring
                                global optimum
        f_next = f(x_next)

        # pick a random nest
        j ← random number from 1 to n
          if f_next < f(xj) then: (for minimization)
              xj ← x_next
```



```
        f(xj) ← f_nest
    end if
end for

# Abandon the worst nests based on pa probability
for i ← 1 to n do
    if random() < pa then:
        xi ← random nest in search space
        fi ← f(xi)
    end if
end for

# update best nest among the available
best ← find optimal among the currently available
        choices
end for

# output
output the best possible solution (nest) based on
whether minimization or maximization is desired

End
```

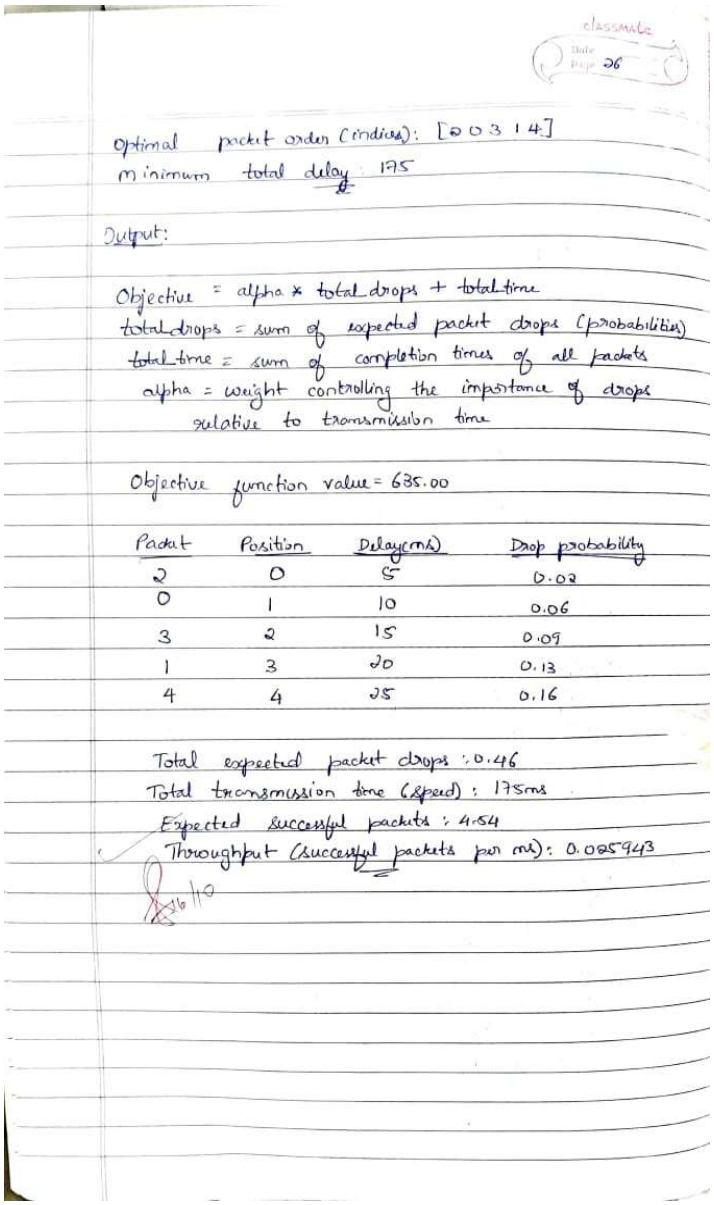
→ ↓ pd
→ ↑ speed

optimizing N/w packets

Optimization in N/w packets:

Output

```
Iteration 1/100, Best fitness: 175
Iteration 11/100, Best fitness: 175
Iteration 21/100, Best fitness: 175
:
Iteration 91/100: Best fitness: 175
Iteration 100/100: Best fitness: 175
```



Optimal packet order (indices): [2 0 3 1 4]
Minimum total delay: 175

Output:

Objective = alpha * total_drops + total_time
total_drops = sum of expected packet drops (probabilities)
total_time = sum of completion times of all packets
alpha = weight controlling the importance of drops relative to transmission time

Objective function value = 635.00

Packet	Position	Delay(ms)	Drop probability
2	0	5	0.03
0	1	10	0.06
3	2	15	0.09
1	3	20	0.13
4	4	25	0.16

Total expected packet drops : 0.46
Total transmission time (speed) : 175ms
Expected successful packets : 4.54
Throughput (successful packets per ms): 0.025943

16/10

Code:

```
# Network packet Transmission Optimization

import numpy as np
print('USN: 1BM23CS245, NAME: PRATHEEKSHA PAI')

# Packet delays in ms (transmission times)
packet_delays = [10, 20, 5, 15, 25]
```

```
# Base drop probability for packets (could be different per packet)
base_drop_probs = [0.05, 0.10, 0.02, 0.07, 0.12]

# Congestion factor (increases drop probability based on packet position)
congestion_factor = 0.01

# Objective function: combine total expected drops and total completion time
def objective(order, alpha=1000):
    total_time = 0
    current_time = 0
    total_drops = 0

    for position, pkt_idx in enumerate(order):
        current_time += packet_delays[pkt_idx]
        total_time += current_time

        # Calculate drop probability increasing with position in queue
        drop_prob = base_drop_probs[pkt_idx] + congestion_factor * position
        total_drops += drop_prob

    return alpha * total_drops + total_time

# Generate a random solution (permutation of packets)
def random_solution(n):
    solution = np.arange(n)
    np.random.shuffle(solution)
    return solution

# Levy flight step (swap two packets)
def levy_flight(solution):
    n = len(solution)
    i, j = np.random.choice(n, 2, replace=False)
    new_solution = solution.copy()
    new_solution[i], new_solution[j] = new_solution[j], new_solution[i]
    return new_solution

def cuckoo_search(n_packets, n_nests=15, n_iterations=100, pa=0.25):
    nests = [random_solution(n_packets) for _ in range(n_nests)]
    fitness = np.array([objective(nest) for nest in nests])

    best_idx = np.argmin(fitness)
```

```
    best_nest = nests[best_idx].copy()
    best_fitness = fitness[best_idx]

    for iter in range(n_iterations):
        for i in range(n_nests):
            new_nest = levy_flight(nests[i])
            new_fitness = objective(new_nest)

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

                if new_fitness < best_fitness:
                    best_fitness = new_fitness
                    best_nest = new_nest.copy()

        for i in range(n_nests):
            if np.random.rand() < pa:
                nests[i] = random_solution(n_packets)
                fitness[i] = objective(nests[i])

                if fitness[i] < best_fitness:
                    best_fitness = fitness[i]
                    best_nest = nests[i].copy()

        if iter % 10 == 0 or iter == n_iterations - 1:
            print(f"Iteration {iter+1}/{n_iterations}, Best fitness: {best_fitness:.4f}")

    return best_nest, best_fitness

def print_results(order):
    total_time = 0
    current_time = 0
    drop_probs = []

    print("\nPacket order (indices):", order)
    print("Packet | Position | Delay (ms) | Drop Probability")
    print("-----------------------------------------------")

    for position, pkt_idx in enumerate(order):
        current_time += packet_delays[pkt_idx]
```

```
        drop_prob = base_drop_probs[pkt_idx] + congestion_factor * position
        drop_probs.append(drop_prob)
        print(f"{pkt_idx:^6} | {position:^8} | {packet_delays[pkt_idx]:^10} | {drop_prob:.4f}")
    total_drops = sum(drop_probs)
    total_time = sum(np.cumsum([packet_delays[i] for i in order]))

    # Calculate throughput: expected successful packets / total_time
    expected_success_packets = sum(1 - dp for dp in drop_probs)
    throughput = expected_success_packets / total_time if total_time > 0 else 0

    print("----------------------------------------------")
    print(f"Total expected packet drops: {total_drops:.4f}")
    print(f"Total transmission time (sum of completion times): {total_time} ms")
    print(f"Expected successful packets: {expected_success_packets:.4f}")
    print(f"Throughput (successful packets per ms): {throughput:.6f}")

# Run optimization
best_order, best_cost = cuckoo_search(len(packet_delays))

print("\nOptimization complete.")
print(f"Objective function value (lower is better): {best_cost:.4f}")
print_results(best_order)
```

Output:

```
USN: 1BM23CS245, NAME: PRATHEEKSHA PAI
Iteration 1/100, Best fitness: 640.0000
Iteration 11/100, Best fitness: 635.0000
Iteration 21/100, Best fitness: 635.0000
Iteration 31/100, Best fitness: 635.0000
Iteration 41/100, Best fitness: 635.0000
Iteration 51/100, Best fitness: 635.0000
Iteration 61/100, Best fitness: 635.0000
Iteration 71/100, Best fitness: 635.0000
Iteration 81/100, Best fitness: 635.0000
Iteration 91/100, Best fitness: 635.0000
Iteration 100/100, Best fitness: 635.0000

Optimization complete.
Objective function value (lower is better): 635.0000

Packet order (indices): [2 0 3 1 4]
Packet | Position | Delay (ms) | Drop Probability
----------------------------------------------
  2    |    0     |     5      | 0.0200
  0    |    1     |     10     | 0.0600
  3    |    2     |     15     | 0.0900
  1    |    3     |     20     | 0.1300
  4    |    4     |     25     | 0.1600
----------------------------------------------
Total expected packet drops: 0.4600
Total transmission time (sum of completion times): 175 ms
Expected successful packets: 4.5400
Throughput (successful packets per ms): 0.025943
```

Program 5

Grey Wolf Optimizer (GWO):

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

Algorithm:

classmate

Date

Page 29

3/10/25

Grey wolf optimization

Begin GWO

1. Define the objective function $f(x)$, $x = (x_1, x_2, \ldots x_d)$
where $d \rightarrow$ maximum no of dimensions in the search space

2. Initialize the parameters
N = no of wolves (population size)
maxitrs = maximum no of iterations
a = 2 // control variable decreases linearly from 2 to 0
t = 0 // initial iteration count

3. Initialize the population
for each wolf i = 1 to N:
initialize the population position X_i randomly within the search space bounds

4. Evaluate fitness
For each wolf i = 1 to N:
evaluate $f(X_i)$ // fitness function

5. Identify the top three wolves
α (best fitness)
β (second best fitness)
δ (third best fitness)

6. while t < maxitrs:
for each wolf i = 1 to N:
for dimension j = 1 to d:
# compute coefficients
A1 = 2 * a * rand() − a
C1 = 2 * rand()
$D_\alpha = |C1 * X_{\alpha j} - X_{ij}|$

$X_1 = X_{\alpha j} - A_1 * D_\alpha$

$A_2 = 2 * a * rand() - a$
$C_2 = 2 * rand()$
$D_\beta = |C_2 * X_{\beta j} - X_{ij}|$
$X_2 = X_{\beta j} - A_2 * D_\beta$

$A_3 = 2 * a * rand() - a$
$C_3 = 2 * rand()$
$D_\delta = |C_3 * X_{\delta j} - X_{ij}|$
$X_3 = X_{\delta j} - A_3 * D_\delta$

$X_{ij} = (X_1 + X_2 + X_3)/3$

End for
# update fitness
for each wolf $i = 1$ to N:
compute $f(X_i)$
end for
// update control variable
$a = 2 - 2 * (t/maxitre)$
$t = t+1$
update wolves based on new fitness values
update alpha (α), β, δ wolves
7. # output
Output the best position X_i and best fitness found

End GWO

IP → color to BW / greyscale

Image processing

Original color image → Grayscale image → Black and white image

Optimal threshold found by GWO: 104

Code:

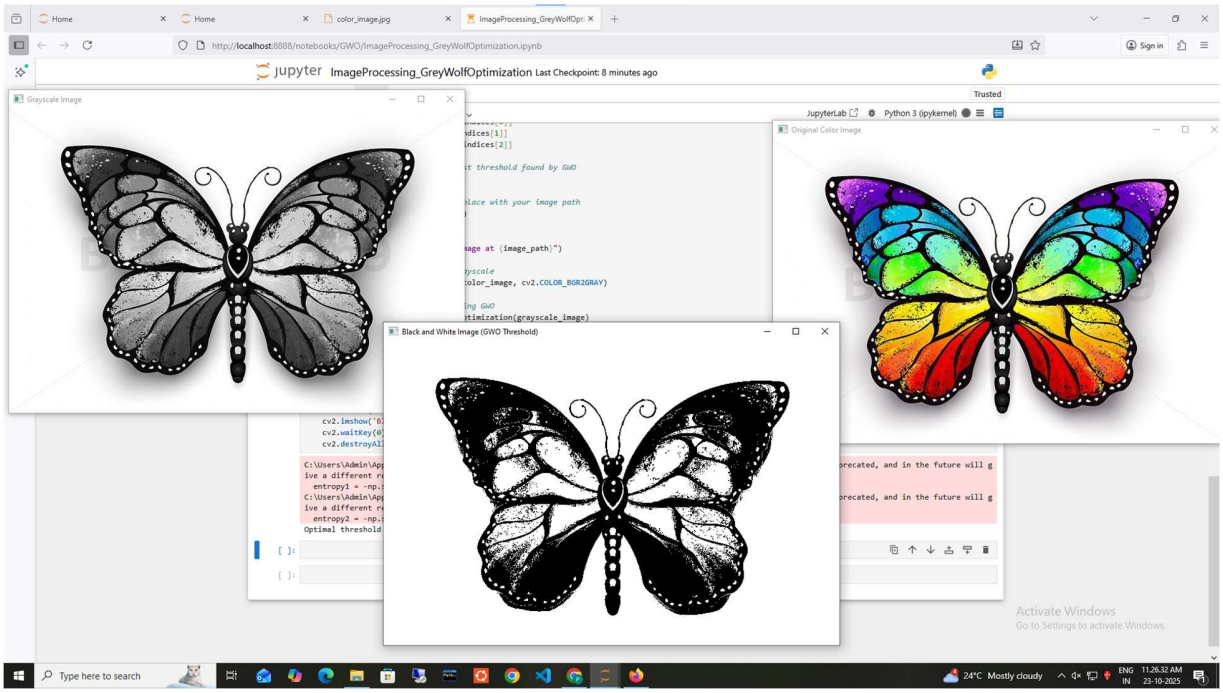
```
#  Image processing - converting image from color to Black and White depending on the  greyscale

import cv2
import numpy as np

print('USN: 1BM23CS245, NAME: PRATHEEKSHA PAI')

# Function to calculate the objective function (e.g., Kapur's entropy) for a given threshold
def objective_function(threshold, grayscale_image):
    # Ensure threshold is an integer
    threshold = int(threshold)
```

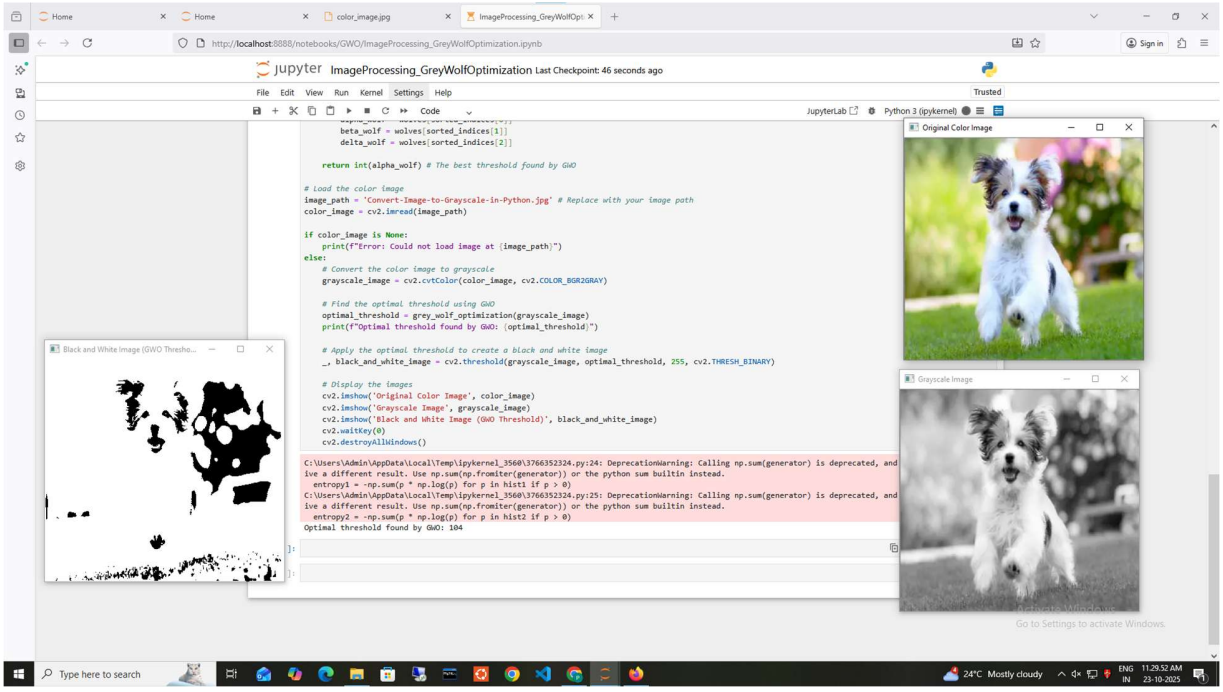
```
    # Separate pixels into two classes based on the threshold
    class1 = grayscale_image[grayscale_image <= threshold]
    class2 = grayscale_image[grayscale_image > threshold]

    # Calculate probabilities for each class
    prob1 = len(class1) / grayscale_image.size if grayscale_image.size > 0 else 0
    prob2 = len(class2) / grayscale_image.size if grayscale_image.size > 0 else 0

    # Calculate entropies (using Kapur's entropy as an example)
    if prob1 == 0 or prob2 == 0:
        return 0  # Avoid log(0)

    hist1, _ = np.histogram(class1, bins=256, range=(0, 256), density=True)
    hist2, _ = np.histogram(class2, bins=256, range=(0, 256), density=True)

    entropy1 = -np.sum(p * np.log(p) for p in hist1 if p > 0)
    entropy2 = -np.sum(p * np.log(p) for p in hist2 if p > 0)

    return entropy1 + entropy2

# Grey Wolf Optimization (GWO) implementation for thresholding
def grey_wolf_optimization(grayscale_image, num_wolves=10, max_iterations=50):
    # Initialize wolf positions (thresholds) randomly
    lower_bound = 0
    upper_bound = 255
    wolves = np.random.uniform(lower_bound, upper_bound, num_wolves)

    # Initialize alpha, beta, and delta wolves
    alpha_wolf = None
    beta_wolf = None
    delta_wolf = None

    # Calculate initial fitness for all wolves
    fitness = [objective_function(w, grayscale_image) for w in wolves]

    # Sort wolves by fitness to identify alpha, beta, delta
    sorted_indices = np.argsort(fitness)[::-1]  # Descending order for maximization
    alpha_wolf = wolves[sorted_indices[0]]
    beta_wolf = wolves[sorted_indices[1]]
    delta_wolf = wolves[sorted_indices[2]]
```

```
for iteration in range(max_iterations):
    a = 2 - iteration * (2 / max_iterations)  # 'a' decreases linearly from 2 to 0

    for i in range(num_wolves):
        # Calculate C1, C2, C3, and A1, A2, A3
        r1, r2 = np.random.rand(2)
        C1 = 2 * r2
        A1 = 2 * a * r1 - a

        r1, r2 = np.random.rand(2)
        C2 = 2 * r2
        A2 = 2 * a * r1 - a

        r1, r2 = np.random.rand(2)
        C3 = 2 * r2
        A3 = 2 * a * r1 - a

        # Calculate D_alpha, D_beta, D_delta
        D_alpha = np.abs(C1 * alpha_wolf - wolves[i])
        D_beta = np.abs(C2 * beta_wolf - wolves[i])
        D_delta = np.abs(C3 * delta_wolf - wolves[i])

        # Calculate X1, X2, X3
        X1 = alpha_wolf - A1 * D_alpha
        X2 = beta_wolf - A2 * D_beta
        X3 = delta_wolf - A3 * D_delta

        # Update wolf position
        wolves[i] = (X1 + X2 + X3) / 3

        # Keep wolves within bounds
        wolves[i] = np.clip(wolves[i], lower_bound, upper_bound)

    # Update alpha, beta, delta based on new positions
    fitness = [objective_function(w, grayscale_image) for w in wolves]
    sorted_indices = np.argsort(fitness)[::-1]
    alpha_wolf = wolves[sorted_indices[0]]
    beta_wolf = wolves[sorted_indices[1]]
    delta_wolf = wolves[sorted_indices[2]]
```

```
    return int(alpha_wolf) # The best threshold found by GWO

# Load the color image
image_path = 'color_image.jpg' # Replace with your image path
color_image = cv2.imread(image_path)

if color_image is None:
    print(f"Error: Could not load image at {image_path}")
else:
    # Convert the color image to grayscale
    grayscale_image = cv2.cvtColor(color_image, cv2.COLOR_BGR2GRAY)

    # Find the optimal threshold using GWO
    optimal_threshold = grey_wolf_optimization(grayscale_image)
    print(f"Optimal threshold found by GWO: {optimal_threshold}")

    # Apply the optimal threshold to create a black and white image
    _, black_and_white_image = cv2.threshold(grayscale_image, optimal_threshold, 255,
cv2.THRESH_BINARY)

    # Display the images
    cv2.imshow('Original Color Image', color_image)
    cv2.imshow('Grayscale Image', grayscale_image)
    cv2.imshow('Black and White Image (GWO Threshold)', black_and_white_image)
    cv2.waitKey(0)
    cv2.destroyAllWindows()
```

Output:


Home
Home
color_image.jpg
ImageProcessing_GreyWolfOpt
http://localhost:8888/notebooks/GWO/ImageProcessing_GreyWolfOptimization.ipynb
Sign in
jupyter ImageProcessing_GreyWolfOptimization Last Checkpoint: 8 minutes ago
Trusted
JupyterLab
Python 3 (ipykernel)
Grayscale Image
Original Color Image
Black and White Image (GWO Threshold)
Activate Windows
Go to Settings to activate Windows.
Type here to search
24°C Mostly cloudy
ENG IN
11:26:32 AM 23-10-2025


Home
Home
color_image.jpg
ImageProcessing_GreyWolfOpt
http://localhost:8888/notebooks/GWO/ImageProcessing_GreyWolfOptimization.ipynb
Sign in
jupyter ImageProcessing_GreyWolfOptimization Last Checkpoint: 46 seconds ago
File Edit View Run Kernel Settings Help
Trusted
JupyterLab
Python 3 (ipykernel)
Code
beta_wolf = wolves[sorted_indices[1]]
delta_wolf = wolves[sorted_indices[2]]
return int(alpha_wolf) # The best threshold found by GWO
# Load the color image
image_path = 'Convert-Image-to-Grayscale-in-Python.jpg' # Replace with your image path
color_image = cv2.imread(image_path)
if color_image is None:
print(f"Error: Could not load image at {image_path}")
else:
# Convert the color image to grayscale
grayscale_image = cv2.cvtColor(color_image, cv2.COLOR_BGR2GRAY)
# Find the optimal threshold using GWO
optimal_threshold = grey_wolf_optimization(grayscale_image)
print(f"Optimal threshold found by GWO: {optimal_threshold}")
# Apply the optimal threshold to create a black and white image
_, black_and_white_image = cv2.threshold(grayscale_image, optimal_threshold, 255, cv2.THRESH_BINARY)
# Display the images
cv2.imshow('Original Color Image', color_image)
cv2.imshow('Grayscale Image', grayscale_image)
cv2.imshow('Black and White Image (GWO Threshold)', black_and_white_image)
cv2.waitKey(0)
cv2.destroyAllWindows()
C:\Users\Admin\AppData\Local\Temp\ipykernel_3560\3766352324.py:24: DeprecationWarning: Calling np.sum(generator) is deprecated, and
ive a different result. Use np.sum(np.fromiter(generator)) or the python sum builtin instead.
entropy1 = -np.sum(p * np.log(p) for p in hist1 if p > 0)
C:\Users\Admin\AppData\Local\Temp\ipykernel_3560\3766352324.py:25: DeprecationWarning: Calling np.sum(generator) is deprecated, and
ive a different result. Use np.sum(np.fromiter(generator)) or the python sum builtin instead.
entropy2 = -np.sum(p * np.log(p) for p in hist2 if p > 0)
Optimal threshold found by GWO: 104
Black and White Image (GWO Thresho...
Original Color Image
Grayscale Image
Activate Windows
Go to Settings to activate Windows.
Type here to search
24°C Mostly cloudy
ENG IN
11:29:52 AM 23-10-2025

Program 6

Parallel Cellular Algorithms and Programs:

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

Algorithm:

classmate
Date
Page 30

30/10/25

Parallel Cellular Algorithm

```
FUNCTION ParallelCellularAlgorithm()
        problem ← DefineProblem()
        params ← InitializeParameters()
        cells ← InitializePopulation(params)

        best_solution ← NULL
        best_fitness ← -∞

        FOR itr from 1 to params.max_itrs DO
            fitness_values ← EvaluateFitness(cells, problem)

        FOR EACH cell in cells DO
            neighbors ← GetNeighbors(cell, cells, params.neighborhood)

            cell ← UpdateCellState(cell, neighbors, problem, params)
        END FOR

        (best_cell, best_cell_fitness) ← FindBestCell(cells, fitness)

        IF best_cell_fitness > best_fitness then
            best_fitness ← best_cell_fitness
            best_solution ← best_cell
        END IF
    END FOR

    RETURN best_solution, best_fitness
END FUNCTION

FUNCTION DefineProblem()
    DEFINE objective_function(x)
    RETURN
    Value of objective_function(x)
```

classmate
Date
Page 31

```
    END FUNCTION
    RETURN objective_function
END FUNCTION

FUNCTION InitializeParameters()
        params.num_cells ← N
        params.grid_size ← (rows, cols)
        params.neighborhood ← Moore or VonNeumann
        params.max_itrs ← max_itr
        params.mutation_rate ← μ
        RETURN params
END FUNCTION

FUNCTION InitializePopulation(params)
        cells ← []
        For i from 1 To params.num_cells DO
            cell.position ← RandomPositionInSolutionSpace()
            cell.state ← cell.position
            ADD cell to cells
        END FOR
        RETURN cells
END FUNCTION

FUNCTION EvaluateFitness(cells, problem)
        fitness_values ← [ ]
        FOR EACH cell in cells DO
            fitness ← problem(cell.state)
            ADD fitness TO fitness_values
        END FOR
        RETURN fitness_values
END FUNCTION

FUNCTION GetNeighbors(cell, cells, neighborhoodtype)
    // use grid topology
```

RETURN list_of_neighbor_cells
END FUNCTION

FUNCTION updateCellState (cell, neighbors, problem, cells)
best_neighbor ← NeighborWithBestFitness (neighbor, problem)
cell_state ← CombineStates (cells state, best_neighbor. state, params)
Return cell
END FUNCTION

FUNCTION FindBestCell(cells, fitness_values)
index ← argmax (fitness_values)
RETURN cells [index], fitness_values [index]
END FUNCTION

30/10/25

Mathematical function

Output:

1D: square rule (vectorized version) // $f(x)=x^2$
Initial: [0.5 1 -0.5 2 -1.5]
Step1: [0.25 1 0.25 4 2.25]
Step2: [0.0625 1 0.0625 16 5.0625]
Step 3: [3.9000e-03 1.0000e+00 3.9000e-03 2.5600e+02 2.5690e+01]
Here, each cell updates independent of neighbors
~~1D: Diffusion rule (uses)~~

Algorithm applies the function repeatedly to all elements in parallel, showing exponential growth/decay depending on value magnitude

1D: diffusion rule (uses neighbors, threaded parallel update)
Initial: [1 2 0 4 3 0.5]
Step1: [1.0833 1.5 1 3.1667 2.75 1]
Step 2: [1.1389 1.3472 1.4444 2.7361 2.5278 1.3056]
Step 3: [1.2014 1.3287 1.6435 2.4861 2.3588 1.4815]
Step 4: [1.2693 1.36 1.7315 2.3045 2.2338 1.581]

Rule used is diffusion
new[i] = $x_i + \alpha \times (\text{avg}(\text{neighbors}) - x_i)$
$\alpha = 0.5$, radius=1 (each cell sees left and right neighbor)

Each cell moves halfway toward the average of its neighbor. This models diffusion or smoothing - high values drop, low values rise

2D diffusion demo:
Initial:
[[10 10 10 10]
[10 0 0 10]
[10 0 0 10]
[10 10 10 10]]

Step1:
[[9.4444 8.8889 8.8889 9.4444]
[8.8889 2.7778 2.7778 8.8889]
[8.8889 2.7778 2.7778 8.8889]
[9.4444 8.8889 8.8889 9.4444]]

Step2:
[[8.9506 8.2716 8.2716 8.9506]
[8.2716 4.5062 4.5062 8.2716]
[8.2716 4.5062 4.5062 8.2716]
[8.9506 8.2716 8.2716 8.9506]]

Step 3:

[[8.5528 7.9287 7.9287 8.5528]
[7.9287 5.5898 5.5898 7.9287]
[7.9287 5.5898 5.5898 7.9287]
[8.5528 7.9287 7.9287 8.5528]]

Step 4:

[[8.2495 7.7381 7.7381 8.2495]
[7.7381 6.2742 6.2742 7.7381]
[7.7381 6.2742 6.2742 7.7381]
[8.2495 7.7381 7.7381 8.2495]]

Step 5:

[[8.0261 7.6323 7.6323 8.0261]
[7.6323 6.7093 6.7093 7.6323]
[7.6323 6.7093 6.7093 7.6323]
[8.0261 7.6323 7.6323 8.0261]]

Visually,
cold middle region heats up
hot boundary cools down
Eventually, grid reaches thermal equilibrium
(all values close to one another)

Research paper analysis:

Early research
John von Neumann (1950s), Toffoli & Margolus (1980s)
Focused on concept and theory - how local cell rules can produce complex global behavior
Rules: Fixed, hand-crafted, simple binary or discrete states
Performance: Theoretical massive speedup via custom hardware
limited in practice due to technology constraints

Application: Self-replication, theoretical computation

Modern research (2000s)
Focus on performance and learning using GPUs and neural networks for adaptive rules
learned / differentiable using deep learning
Implemented efficiently using GPUs
Performance is faster than CPU versions & scalable
used in AI, image processing, optimization
Practically implemented 100 times better

Code:

```
# Python code to demonstrate "mathematical function application using a parallel cellular algorithm".
# This code provides:
# 1) A vectorized (numpy) implementation (fast, recommended).
# 2) A ThreadPool-based "parallel" cellular update where every cell updates in parallel
#    based on the previous state (synchronous CA update). This mimics a parallel cellular algorithm.
# 3) 1D and 2D examples with sample functions (square, sin, average-with-neighbors).
# The ThreadPool version uses thread-workers so arrays are not repeatedly copied between processes.
# You can modify `cell_rule` to implement other mathematical mappings.

from concurrent.futures import ThreadPoolExecutor
```

```
import numpy as np
import math
from typing import Callable, Tuple

print("USN: 1BM23CS245 NAME: PRATHEEKSHA PAI")

def apply_rule_vectorized_1d(state: np.ndarray, rule: Callable[[float, np.ndarray, int], float],
radius:int=1) -> np.ndarray:
    """Apply rule to 1D state using vectorized operations when rule is simple.
      This helper expects that `rule` can be expressed as a simple elementwise call;
      if rule depends on neighbors, fall back to `apply_rule_threaded_1d`.
    """
    # If rule is a simple callable that accepts a scalar, we can call it on the whole array
    try:
        return np.vectorize(lambda x: rule(x, None, 0))(state)
    except Exception:
        # Fallback: use threaded version
        return apply_rule_threaded_1d(state, rule, radius)

def apply_rule_threaded_1d(state: np.ndarray, rule: Callable[[float, np.ndarray, int], float],
radius:int=1, workers:int=8) -> np.ndarray:
    """Apply rule to 1D state using ThreadPool to mimic parallel cellular update.
      rule(cell_value, neighborhood_array, i) -> new_value
      neighborhood_array is a view of previous state centered at i (with wrap-around).
    """
    n = len(state)
    prev = state.copy()
    def worker(i):
        # gather neighborhood (circular/wrap-around)
        if radius == 0:
            neigh = None
        else:
            idxs = [(i + d) % n for d in range(-radius, radius+1)]
            neigh = prev[idxs]
        return rule(prev[i], neigh, i)
    with ThreadPoolExecutor(max_workers=workers) as ex:
        results = list(ex.map(worker, range(n)))
    return np.array(results, dtype=float)

def apply_rule_threaded_2d(state: np.ndarray, rule: Callable[[float, np.ndarray, Tuple[int,int]], float],
radius:int=1, workers:int=8) -> np.ndarray:
```

```
    """Apply rule to 2D state using ThreadPool. Synchronous update (all cells read prev state, write
new state).
      rule(cell_value, neighborhood_block, (i,j)) -> new_value
      neighborhood_block is a (2*radius+1)x(2*radius+1) view with wrap-around.
    """
    rows, cols = state.shape
    prev = state.copy()
    indices = [(i,j) for i in range(rows) for j in range(cols)]
    def worker(idx):
        i,j = idx
        if radius == 0:
            neigh = None
        else:
            # build neighborhood with wrap-around
            ii = [ (i + di) % rows for di in range(-radius, radius+1) ]
            jj = [ (j + dj) % cols for dj in range(-radius, radius+1) ]
            block = prev[np.ix_(ii, jj)]
            neigh = block
        return rule(prev[i,j], neigh, (i,j))
    with ThreadPoolExecutor(max_workers=workers) as ex:
        results = list(ex.map(worker, indices))
    return np.array(results, dtype=float).reshape(state.shape)

# Example cell rules:
def square_rule(val, neigh, idx):
    return val * val

def sin_rule(val, neigh, idx):
    return math.sin(val)

def avg_neighbors_rule(val, neigh, idx):
    # if neigh is None, return val
    if neigh is None:
        return val
    return float(np.mean(neigh))

def diffusion_rule(val, neigh, idx):
    # simple diffusion: new = val + alpha*(mean(neigh)-val)
    if neigh is None:
        return val
    alpha = 0.5
```

```
    return float(val + alpha*(np.mean(neigh) - val))

# Runner functions: iterate steps
def run_1d_example(initial_state: np.ndarray, rule: Callable, steps:int=5, radius:int=1,
threaded:bool=True):
    s = initial_state.copy().astype(float)
    print("Initial:", s)
    for t in range(steps):
        if threaded:
            s = apply_rule_threaded_1d(s, rule, radius=radius)
        else:
            s = apply_rule_vectorized_1d(s, rule, radius=radius)
        print(f"Step {t+1}:", np.round(s, 4))
    return s

def run_2d_example(initial_state: np.ndarray, rule: Callable, steps:int=4, radius:int=1,
threaded:bool=True):
    s = initial_state.copy().astype(float)
    print("Initial:\n", np.round(s,4))
    for t in range(steps):
        s = apply_rule_threaded_2d(s, rule, radius=radius) if threaded else s  # vectorized 2D neighbor
rules are nontrivial here
        print(f"Step {t+1}:\n", np.round(s,4))
    return s

# --- Demonstrations ---
if __name__ == "__main__":
    # 1D demo: square each cell (no neighbors)
    arr1 = np.array([0.5, 1.0, -0.5, 2.0, -1.5])
    print("1D: square_rule (vectorized version recommended)")
    run_1d_example(arr1, square_rule, steps=3, radius=0, threaded=False)
    print("\n1D: diffusion_rule (uses neighbors, threaded parallel update)")
    arr2 = np.array([1.0, 2.0, 0.0, 4.0, 3.0, 0.5])
    run_1d_example(arr2, diffusion_rule, steps=4, radius=1, threaded=True)

    # 2D demo: diffusion on a small grid (mimics heat diffusion)
    grid = np.array([
        [10.0, 10.0, 10.0, 10.0],
        [10.0, 0.0, 0.0, 10.0],
        [10.0, 0.0, 0.0, 10.0],
        [10.0, 10.0, 10.0, 10.0],
```

```
])
print("\n2D diffusion demo:")
run_2d_example(grid, diffusion_rule, steps=5, radius=1, threaded=True)
```

Output:

```
USN: 1BM23CS245 NAME: PRATHEEKSHA PAI
1D: square_rule (vectorized version recommended)
Initial: [ 0.5  1.  -0.5  2.  -1.5]
Step 1: [0.25 1.   0.25 4.   2.25]
Step 2: [ 0.0625  1.      0.0625 16.      5.0625]
Step 3: [3.90000e-03 1.00000e+00 3.90000e-03 2.56000e+02 2.56289e+01]

1D: diffusion_rule (uses neighbors, threaded parallel update)
Initial: [1.  2.  0.  4.  3.  0.5]
Step 1: [1.0833 1.5    1.     3.1667 2.75   1.    ]
Step 2: [1.1389 1.3472 1.4444 2.7361 2.5278 1.3056]
Step 3: [1.2014 1.3287 1.6435 2.4861 2.3588 1.4815]
Step 4: [1.2693 1.36   1.7315 2.3245 2.2338 1.581 ]

2D diffusion demo:
Initial:
 [[10. 10. 10. 10.]
 [10.  0.  0. 10.]
 [10.  0.  0. 10.]
 [10. 10. 10. 10.]]
Step 1:
 [[9.4444 8.8889 8.8889 9.4444]
 [8.8889 2.7778 2.7778 8.8889]
 [8.8889 2.7778 2.7778 8.8889]
 [9.4444 8.8889 8.8889 9.4444]]
Step 2:
 [[8.9506 8.2716 8.2716 8.9506]
 [8.2716 4.5062 4.5062 8.2716]
 [8.2716 4.5062 4.5062 8.2716]
 [8.9506 8.2716 8.2716 8.9506]]
Step 3:
 [[8.5528 7.9287 7.9287 8.5528]
 [7.9287 5.5898 5.5898 7.9287]
 [7.9287 5.5898 5.5898 7.9287]
 [8.5528 7.9287 7.9287 8.5528]]
Step 4:
 [[8.2495 7.7381 7.7381 8.2495]
 [7.7381 6.2742 6.2742 7.7381]
 [7.7381 6.2742 6.2742 7.7381]
 [8.2495 7.7381 7.7381 8.2495]]
Step 5:
 [[8.0261 7.6323 7.6323 8.0261]
 [7.6323 6.7093 6.7093 7.6323]
 [7.6323 6.7093 6.7093 7.6323]
 [8.0261 7.6323 7.6323 8.0261]]
```

Program 7

Optimization via Gene Expression Algorithms:

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

Algorithm:

classmate
Date
Page 10

04/09/25 Gene Expression Algorithm

Gene expression algorithm

```
Input: population_size, max_generations, crossover_rate,
       mutation_rate, num_genes, problem_definition
Output: best_solution

1: Initialize all input variables
2: generations ← 0, population ← initialization(population_size, num_genes)
3: fitness_values ← evaluateFitness(population, problem_definition)
   best_solution ← getBestSolution(population, fitness_values)

4: while generations < max_generations and not converged(
      ~~for~~ fitness_values) do:
      selected_population ← Selection(population, fitness_values)
      offspring ← crossover(selected_population)
      mutated_offspring ← mutation(offspring)
      ~~gene expression~~ new_population ← GeneExpression(mutated_offspring)

      fitness_values ← evaluateFitness(new_population, problem_
                                       definition)
      candidate_solution ← GetBestSolution(new_population,
                                       fitness_values)

      if( Fitness(candidate_solution) > Fitness(best_solution)):
                  best_solution ← candidate_solution
      EndIf
      population ← new_population
      generation ← generation+1
   ~~EndIf~~
   EndWhile

   Return best_solution
End
```

Eg: Problem definition: x^2; range → population_size → [-5,5]
best_solution = 5

classmate
Date
Page 11

```
Function initialization(population_size, num_genes):
    population ← [ ]
    for i ← 1 to population_size do
        individual ← [ ]
        for j ← 1 to num_genes:
            gene ← random() # maybe random number,
                            # operator, etc...
            individual.add(Gene)
        EndFor
        population.add(individual)
    End for
    return population
End function

Function evaluateFitness(population, problem_definition):
    fitness_values = [ ]
    for each individual in population:
        fitness ← ComputeObjective(individual, problem_defi-
                                   -nition)
        fitness_values.add(fitness)
    EndFor
    return fitness_values.
EndFunction

Function GetBestSolution(population, fitness_values):
    # get best solution
    Index ← ArgMax(fitness_values)
    return population[Index]
EndFunction

Function Converged(fitness_values):
    # return Boolean indicating there is minimal / no
    # improvement in generations to select
    return Boolean
```

```
EndFunction

Function Selection(population, fitness_values):
    # Select best parents based on fitness
        parents = [ ]
        for each individual in population:
            # apply roulette selection or any other selection
            # techniques to select based on fitness values
            selected ← roulette_selection(fitness_values)
            parents.add(selected)
    EndFor
    return parents
EndFunction

Function crossover(selected_population, cross_rate):
        offspring = [ ]
        for i in size(selected_population)-1:
            ~~parent~~ parent1[i] ← selected_population[i]
            ~~child[i] ←~~
            parent2[i] ← selected_population[i+1]
        if Random() < size(selected_population):
            ~~child ← partition~~
            partition ← random()
            child1 ← parent1[0:partition] + parent2[partition:]
            child2 ← parent2[0:partition] + parent1[partition:]
        else
            child1 ← parent1
            child2 ← parent2
        EndIf
    EndFor
    return offspring
EndFunction
    # crossover function produces result from best selected parents
    # Fittest is selected and crossover performed
```



```
Function mutation(offspring, mutation_rate):
    # add random values to offspring to introduce
    # variety
    for each individual in offspring:
        gene ← random() # random changes
        ~~op~~ offspring.modify(gene)
    EndFor
    return offspring
EndFunction

Function GeneExpression(mutated_offspring):
    # Here each gene (chromosome) is not a solution,
it is expression or random explanation which needs
to be evaluated to find solutions.
        solution ← evaluate(mutated_offspring)
        return solution
EndFunction

Function Fitness(candidate):
    ~~Evaluate~~
    fitness is already evaluated
    find fitness of that individual candidate in
    population
    return fitness_value[candidate];
EndFunction,
```

Star + Sfic

GEA for star constellation:

Output:

Gen 0: Best fitness = 49.302
Gen 10: Best fitness = 47.905
Gen 20: Best fitness = 46.976

Gen 30: Best fitness : 43.415
Gen 40: Best fitness : 43.415
Gen 50: Best fitness : 43.415
Gen 60: Best fitness: 43.415
Gen 70: Best fitness: 43.415
Gen 80: Best fitness: 43.415
Gen 90: Best fitness: 43.290
Gen 99: Best fitness: 43.290

Best expression: (((-4.44+x)*(y-x))+((x*x)-(4.60*y)))
Best order: [1, 0, 12, 11, 2, 13, 19, 10, 18, 14, 17, 9, 3, 15, 16, 8, 7, 6, 4, 5]
Total length: 43.28976477
Crossings: 0
objective Score: 43.28976477

Specific GEA applications:

Symbolic Regression: Automatic discovery of mathematical equations from given data.

	Population Average		Best Individual		
Gen	Length	Fitness	Length	Fitness	OOB fitness (out-of-bag)
0	48.87	2.84e+07	63	4.0309	4.6985
1	29.52	80.3971	15	3.684	3.875
2	35.42	137.858	9	3.624	4.351
3	30.11	71.798	9	3.595	4.660
4	19.12	40.65	25	3.489	5.286
5	14.05	~~40~~ 57.768	9	3.462	5.814
6	14.09	35.268	11	3.511	5.375
7	14.34	26.240	25	3.488	5.550
8	14.34	31.734	33	3.509	5.317
9	14.45	56.791	21	3.443	5.890

Discovered Expression:

add(sub(mul(mul(mul(0.447, -0.248), sub(x0, -0.990)), -0.194), sub(0.224, x0)), add(mul(x0, x0), add(x0, 0.997)))

(graph)

4/5/25

Code:

```
# Star constellation

import math
import random
from typing import List, Tuple

Point = Tuple[float, float]
print("By: USN: 1BM23CS245, NAME: PRATHEEKSHA PAI")
```

```
# -------- Geometry helpers (from your original code) --------
def dist(a: Point, b: Point) -> float:
    return math.hypot(a[0] - b[0], a[1] - b[1])

def _orientation(p: Point, q: Point, r: Point) -> int:
    val = (q[1] - p[1]) * (r[0] - q[0]) - (q[0] - p[0]) * (r[1] - q[1])
    if abs(val) < 1e-12:
        return 0
    return 1 if val > 0 else 2

def _on_segment(p: Point, q: Point, r: Point) -> bool:
    return (min(p[0], r[0]) <= q[0] <= max(p[0], r[0]) and
            min(p[1], r[1]) <= q[1] <= max(p[1], r[1]))

def segments_intersect(p1: Point, q1: Point, p2: Point, q2: Point) -> bool:
    if p1 == p2 or p1 == q2 or q1 == p2 or q1 == q2:
        return False
    o1 = _orientation(p1, q1, p2)
    o2 = _orientation(p1, q1, q2)
    o3 = _orientation(p2, q2, p1)
    o4 = _orientation(p2, q2, q1)
    if o1 != o2 and o3 != o4:
        return True
    if o1 == 0 and _on_segment(p1, p2, q1): return True
    if o2 == 0 and _on_segment(p1, q2, q1): return True
    if o3 == 0 and _on_segment(p2, p1, q2): return True
    if o4 == 0 and _on_segment(p2, q1, q2): return True
    return False

def path_length(order: List[int], stars: List[Point], cycle: bool) -> float:
    total = 0.0
    for i in range(len(order) - 1):
        total += dist(stars[order[i]], stars[order[i + 1]])
    if cycle and len(order) > 1:
        total += dist(stars[order[-1]], stars[order[0]])
    return total

def count_crossings(order: List[int], stars: List[Point], cycle: bool) -> int:
    edges = []
    for i in range(len(order) - 1):
        edges.append((order[i], order[i + 1]))
```

```
    if cycle and len(order) > 1:
        edges.append((order[-1], order[0]))
    crossings = 0
    for i in range(len(edges)):
        a1, a2 = edges[i]
        p1, q1 = stars[a1], stars[a2]
        for j in range(i + 1, len(edges)):
            b1, b2 = edges[j]
            if len({a1, a2, b1, b2}) < 4:
                continue
            p2, q2 = stars[b1], stars[b2]
            if segments_intersect(p1, q1, p2, q2):
                crossings += 1
    return crossings

# -------- Protected division --------
def pdiv(a, b):
    try:
        return a / b if abs(b) > 1e-6 else 1.0
    except:
        return 1.0

# -------- Expression tree nodes --------
class Node:
    def evaluate(self, x, y):
        raise NotImplementedError
    def copy(self):
        raise NotImplementedError
    def size(self):
        raise NotImplementedError
    def get_subtree(self, index):
        raise NotImplementedError
    def replace_subtree(self, index, subtree):
        raise NotImplementedError

class Const(Node):
    def __init__(self, val):
        self.val = val
    def evaluate(self, x, y):
        return self.val
    def copy(self):
```

```
        return Const(self.val)
    def size(self):
        return 1
    def get_subtree(self, index):
        if index == 0:
            return self
        raise IndexError
    def replace_subtree(self, index, subtree):
        if index == 0:
            return subtree
        raise IndexError
    def __repr__(self):
        return f"{self.val:.2f}"

class VarX(Node):
    def evaluate(self, x, y):
        return x
    def copy(self):
        return VarX()
    def size(self):
        return 1
    def get_subtree(self, index):
        if index == 0:
            return self
        raise IndexError
    def replace_subtree(self, index, subtree):
        if index == 0:
            return subtree
        raise IndexError
    def __repr__(self):
        return "x"

class VarY(Node):
    def evaluate(self, x, y):
        return y
    def copy(self):
        return VarY()
    def size(self):
        return 1
    def get_subtree(self, index):
        if index == 0:
```

```
            return self
        raise IndexError
    def replace_subtree(self, index, subtree):
        if index == 0:
            return subtree
        raise IndexError
    def __repr__(self):
        return "y"

class OpNode(Node):
    def __init__(self, left, right):
        self.left = left
        self.right = right
    def size(self):
        return 1 + self.left.size() + self.right.size()
    def get_subtree(self, index):
        if index == 0:
            return self
        index -= 1
        left_size = self.left.size()
        if index < left_size:
            return self.left.get_subtree(index)
        else:
            return self.right.get_subtree(index - left_size)
    def replace_subtree(self, index, subtree):
        if index == 0:
            return subtree
        index -= 1
        left_size = self.left.size()
        if index < left_size:
            self.left = self.left.replace_subtree(index, subtree)
        else:
            self.right = self.right.replace_subtree(index - left_size, subtree)
        return self

class Add(OpNode):
    def evaluate(self, x, y):
        return self.left.evaluate(x, y) + self.right.evaluate(x, y)
    def copy(self):
        return Add(self.left.copy(), self.right.copy())
    def __repr__(self):
```

```
        return f"({self.left}+{self.right})"

class Sub(OpNode):
    def evaluate(self, x, y):
        return self.left.evaluate(x, y) - self.right.evaluate(x, y)
    def copy(self):
        return Sub(self.left.copy(), self.right.copy())
    def __repr__(self):
        return f"({self.left}-{self.right})"

class Mul(OpNode):
    def evaluate(self, x, y):
        return self.left.evaluate(x, y) * self.right.evaluate(x, y)
    def copy(self):
        return Mul(self.left.copy(), self.right.copy())
    def __repr__(self):
        return f"({self.left}*{self.right})"

class Div(OpNode):
    def evaluate(self, x, y):
        return pdiv(self.left.evaluate(x, y), self.right.evaluate(x, y))
    def copy(self):
        return Div(self.left.copy(), self.right.copy())
    def __repr__(self):
        return f"({self.left}/{self.right})"

# -------- Generate random expression --------
def random_expr(depth=3):
    if depth <= 0:
        terminal = random.choice([Const(random.uniform(-10, 10)), VarX(), VarY()])
        return terminal
    else:
        op = random.choice([Add, Sub, Mul, Div])
        return op(random_expr(depth - 1), random_expr(depth - 1))

# -------- Convert expression to star order --------
def expr_to_order(expr: Node, stars: List[Point]) -> List[int]:
    scored = [(i, expr.evaluate(x, y)) for i, (x, y) in enumerate(stars)]
    scored.sort(key=lambda x: x[1])
    order = [i for i, _ in scored]
    return order
```

```
# -------- Fitness --------
def fitness_expr(expr: Node, stars: List[Point], crossing_weight=50.0, cycle=False) -> float:
    order = expr_to_order(expr, stars)
    L = path_length(order, stars, cycle)
    X = count_crossings(order, stars, cycle)
    return L + crossing_weight * X

# -------- Genetic operators --------

def mutate_expr(expr: Node, prob_mut=0.1, max_depth=3) -> Node:
    if random.random() < prob_mut or max_depth <= 0:
        return random_expr(max_depth)
    else:
        if isinstance(expr, OpNode):
            expr.left = mutate_expr(expr.left, prob_mut, max(0, max_depth - 1))
            expr.right = mutate_expr(expr.right, prob_mut, max(0, max_depth - 1))
        return expr


def crossover_expr(expr1: Node, expr2: Node, prob_cross=0.7) -> Node:
    if random.random() > prob_cross:
        return expr1.copy()

    size1 = expr1.size()
    size2 = expr2.size()
    if size1 == 1 or size2 == 1:
        return expr2.copy()

    # Random subtree swap:
    cross_point1 = random.randint(1, size1 - 1)
    cross_point2 = random.randint(1, size2 - 1)

    subtree1 = expr1.get_subtree(cross_point1)
    subtree2 = expr2.get_subtree(cross_point2)

    new_expr = expr1.copy()
    new_expr = new_expr.replace_subtree(cross_point1, subtree2.copy())

    return new_expr
```

```
# -------- GEA driver --------
def run_gea(
    stars: List[Point],
    population_size=100,
    generations=100,
    crossing_weight=50.0,
    cycle=False,
    mutation_rate=0.2,
    crossover_rate=0.7,
    seed=42,
):
    random.seed(seed)
    pop = [random_expr(depth=3) for _ in range(population_size)]

    def fit(e): return fitness_expr(e, stars, crossing_weight, cycle)

    best = min(pop, key=fit)
    best_f = fit(best)

    for gen in range(generations):
        new_pop = []
        elite_n = max(1, population_size // 20)  # 5% elitism
        sorted_pop = sorted(pop, key=fit)
        elites = sorted_pop[:elite_n]
        new_pop.extend([e.copy() for e in elites])

        while len(new_pop) < population_size:
            # Tournament selection size=3
            contenders = random.sample(pop, 3)
            parent1 = min(contenders, key=fit)
            contenders = random.sample(pop, 3)
            parent2 = min(contenders, key=fit)

            if random.random() < crossover_rate:
                child = crossover_expr(parent1, parent2)
            else:
                child = parent1.copy()

            child = mutate_expr(child, mutation_rate)

            new_pop.append(child)
```

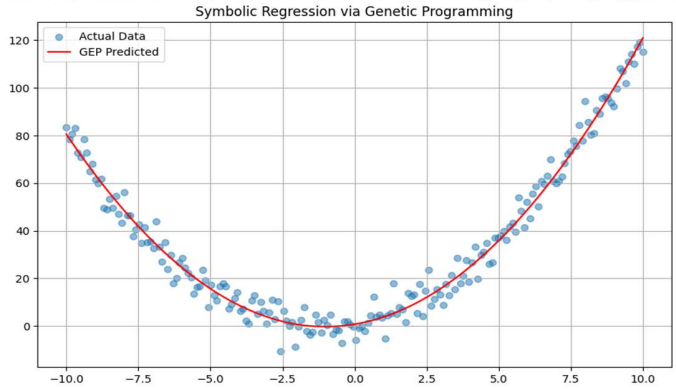
```
        pop = new_pop

        current_best = min(pop, key=fit)
        current_best_f = fit(current_best)
        if current_best_f < best_f:
            best = current_best
            best_f = current_best_f

        if gen % 10 == 0 or gen == generations - 1:
            print(f"Gen {gen}: Best fitness = {best_f:.3f}")

    best_order = expr_to_order(best, stars)
    return best, best_order, best_f

# -------- Main --------
if __name__ == "__main__":
    # Example star coordinates
    stars = [
        (2.1, 9.7), (1.4, 8.3), (3.0, 7.8), (5.2, 9.1), (7.4, 9.5),
        (9.1, 8.6), (8.5, 6.9), (9.7, 5.2), (7.8, 4.1), (5.6, 3.5),
        (3.3, 3.9), (1.5, 4.8), (0.9, 6.2), (2.6, 5.8), (4.2, 6.8),
        (6.1, 7.4), (6.7, 5.6), (4.9, 5.1), (3.8, 4.9), (2.0, 3.2),
    ]

    best_expr, best_order, best_score = run_gea(
        stars,
        population_size=100,
        generations=100,
        crossing_weight=50.0,
        cycle=False,
        mutation_rate=0.2,
        crossover_rate=0.7,
        seed=123,
    )

    print("\nBest expression:", best_expr)
    print("Best order:", best_order)
    print("Total length:", path_length(best_order, stars, cycle=False))
    print("Crossings:", count_crossings(best_order, stars, cycle=False))
    print("Objective score:", best_score)
```

Output:

```
By: USN: 1BM23CS245, NAME: PRATHEEKSHA PAI
Gen 0: Best fitness = 49.302
Gen 10: Best fitness = 47.905
Gen 20: Best fitness = 46.976
Gen 30: Best fitness = 43.415
Gen 40: Best fitness = 43.415
Gen 50: Best fitness = 43.415
Gen 60: Best fitness = 43.415
Gen 70: Best fitness = 43.415
Gen 80: Best fitness = 43.415
Gen 90: Best fitness = 43.290
Gen 99: Best fitness = 43.290

Best expression: (((-4.44+x)*(y-x))+((x*x)-(4.60*x)))
Best order: [1, 0, 12, 11, 2, 13, 19, 10, 18, 14, 17, 9, 3, 15, 16, 8, 7, 6, 4, 5]
Total length: 43.28976477797853
Crossings: 0
Objective score: 43.28976477797853
```

```
# Specific GEA application which is not possible(efficiently) by using Genetic algorithms
# Symbolic Regression:

import numpy as np
import matplotlib.pyplot as plt
from gplearn.genetic import SymbolicRegressor
from gplearn.functions import make_function

print("By: USN: 1BM23CS245, NAME: PRATHEEKSHA PAI")

# Step 1: Create dataset
np.random.seed(42)
X = np.linspace(-10, 10, 200).reshape(-1, 1)
y = X[:, 0]**2 + 2 * X[:, 0] + 1 + np.random.normal(0, 5, X.shape[0])  # add noise

# Step 2: Initialize the symbolic regressor
model = SymbolicRegressor(
    population_size=1000,
    generations=10,
    stopping_criteria=0.01,
    p_crossover=0.7,
    p_subtree_mutation=0.1,
    p_hoist_mutation=0.05,
    p_point_mutation=0.1,
    max_samples=0.9,
    verbose=1,
```

```
    parsimony_coefficient=0.001,
    random_state=42
)

# Step 3: Train the model
model.fit(X, y)

# Step 4: Predict
y_pred = model.predict(X)

# Step 5: Display the evolved formula
print("\nDiscovered Expression:")
print(model._program)

# Step 6: Plot actual vs predicted
plt.figure(figsize=(10, 6))
plt.scatter(X, y, label="Actual Data", alpha=0.5)
plt.plot(X, y_pred, color='red', label="GEP Predicted")
plt.title("Symbolic Regression via Genetic Programming")
plt.legend()
plt.grid(True)
plt.show()
```

Output:

```
By: USN: 1BM23CS245, NAME: PRATHEEKSHA PAI
    |   Population Average    |             Best Individual              |
---- ------------------------- ------------------------------------------ ----------
 Gen   Length          Fitness   Length          Fitness      OOB Fitness  Time Left
   0    48.87      2.84134e+07       63          4.03092          4.69852     22.24s
   1    29.52          80.3971       15          3.68648          3.87504      9.46s
   2    35.42          137.858        9          3.62487          4.35142      9.87s
   3    32.11          71.7987        9          3.59525          4.66022     10.09s
   4    19.12          40.6503       25          3.48941          5.28676      5.30s
   5    14.05          57.7685        9          3.46232          5.81429      3.99s
   6    14.09          35.2683       11          3.51113          5.37554      2.85s
   7    14.34          26.2405       25          3.48819          5.55017      1.83s
   8    14.34          31.7347       33          3.50924          5.31757      0.95s
   9    14.45          56.7714       21          3.44321          5.89005      0.00s

 Discovered Expression:
add(sub(mul(mul(mul(0.447, -0.248), sub(X0, -0.990)), -0.194), sub(0.224, X0)), add(mul(X0, X0), add(X0, 0.997)))
```