VISVESVARAYA TECHNOLOGICAL UNIVERSITY

“JnanaSangama”, Belgaum -590014, Karnataka.

LAB RECORD

Bio Inspired Systems (23CS5BSBIS)

Submitted by

PREZA MISHRA(1BM23CS251)

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

<u>CERTIFICATE</u>

This is to certify that the Lab work entitled " Bio Inspired Systems (23CS5BSBIS)" carried out by **PREZA MISHRA(1BM23CS251),** 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.

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

Index

Github Link:

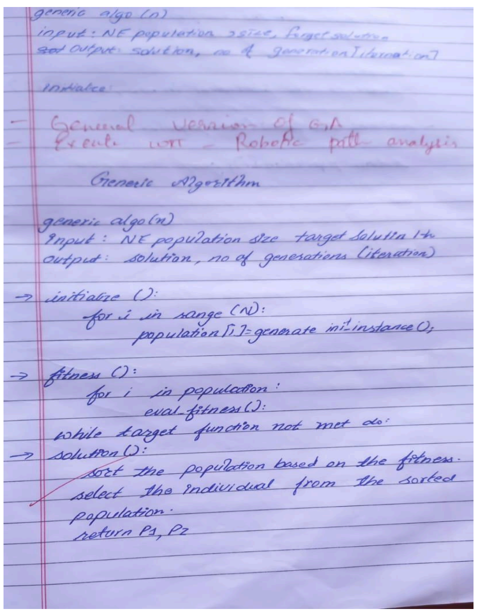
https://github.com/PrezaMishra/BIS-LAB

Program 1 : Genetic Algorithm

Problem statement:

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.

Algorithm:



generic algo (n)
input : NE population size, target solution
output: solution, no of generation [iteration]

initialize

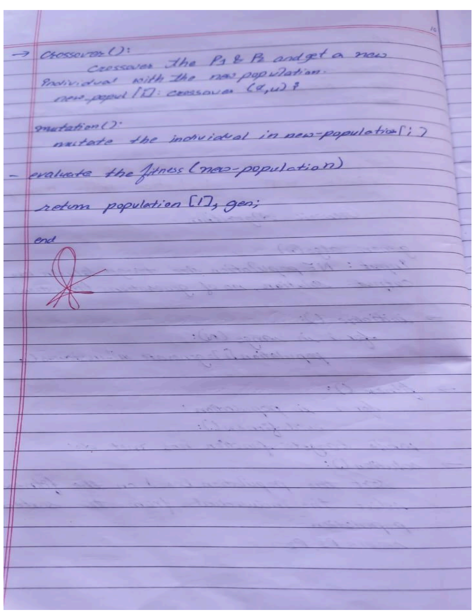
- General version of G.A
- Execute wrt Robotic path analysis

Genetic Algorithm

generic algo (n)
Input : NE population size target solution
output: solution, no of generations (iteration)

→ initialize ():
for i in range (N):
population [i] = generate init instance ();

→ fitness ():
for i in population:
eval_fitness ():
while target function not met do:
→ solution ():
sort the population based on the fitness.
select the individual from the sorted population.
return P1, P2



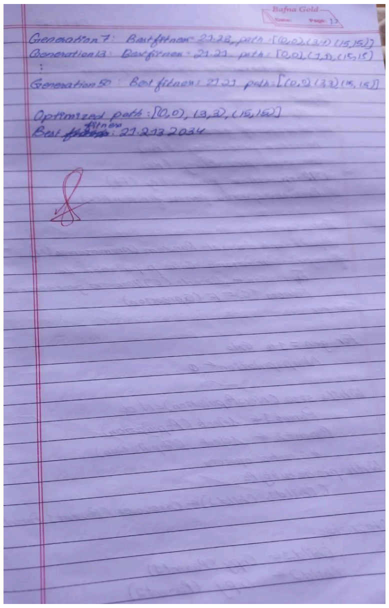
→ crossover ():
crossover the P1 & P2 and get a new individual with the new population.
new_popul [i]: crossover (x, u) ?

mutation ():
mutate the individual in new-population [i]

- evaluate the fitness (new-population)

return population [1], gen;

end



CODE:

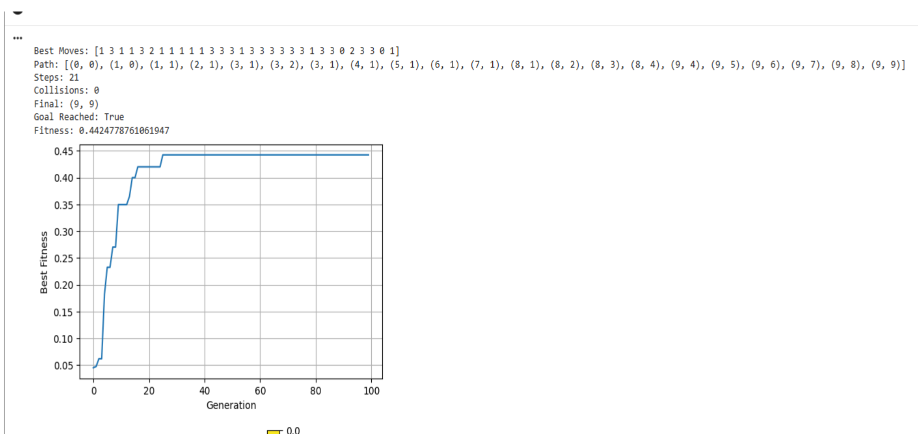
```
import numpy as np
import random
import matplotlib.pyplot as plt

GRID_ROWS = 10
GRID_COLS = 10
START = (0, 0)
GOAL = (9, 9)

OBSTACLES = {
    (3, 3), (3, 4), (3, 5),
    (4, 5), (5, 5),
    (6, 2), (6, 3),
    (7, 7), (7, 8),
    (2, 8), (2, 9)
}

MAX_STEPS = 30
MOVE_DELTAS = {
    0: (-1, 0),
    1: (1, 0),
    2: (0, -1),
    3: (0, 1)
}

POPULATION_SIZE = 80
GENERATIONS = 100
MUTATION_RATE = 0.05
CROSSOVER_RATE = 0.8
TOURNAMENT_SIZE = 3
```

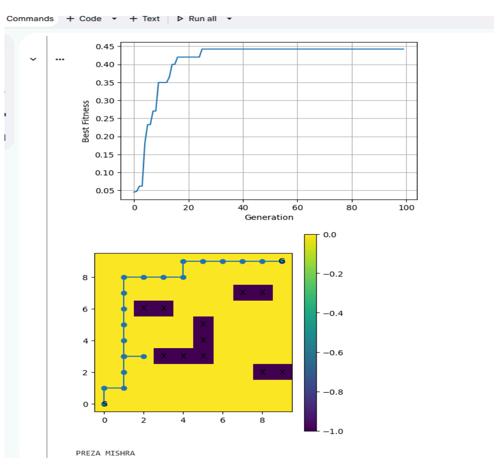
```
ELITE_COUNT = 2

def is_valid_cell(cell):
    r, c = cell
    if r < 0 or r >= GRID_ROWS or c < 0 or c >= GRID_COLS:
        return False
    if cell in OBSTACLES:
        return False
    return True

def random_individual():
    return np.random.randint(0, 4, size=MAX_STEPS)

def decode_path(individual):
    path = [START]
    current = START
    collisions = 0
    for move in individual:
        dr, dc = MOVE_DELTAS[int(move)]
        next_cell = (current[0] + dr, current[1] + dc)
        if is_valid_cell(next_cell):
            current = next_cell
        else:
            collisions += 1
        path.append(current)
        if current == GOAL:
            break
    return path, collisions

def manhattan_distance(a, b):
    return abs(a[0] - b[0]) + abs(a[1] - b[1])

def fitness(individual):
    path, collisions = decode_path(individual)
    final_pos = path[-1]
    distance_to_goal = manhattan_distance(final_pos, GOAL)
    path_length = len(path)
    cost = distance_to_goal * 2.0 + collisions * 3.0 + path_length * 0.2
    if final_pos == GOAL:
        cost *= 0.3
    return 1.0 / (1.0 + cost)

def tournament_selection(population, fitnesses):
    best_idx = None
    for _ in range(TOURNAMENT_SIZE):
        idx = random.randint(0, len(population) - 1)
        if best_idx is None or fitnesses[idx] > fitnesses[best_idx]:
            best_idx = idx
    return population[best_idx].copy()

def single_point_crossover(parent1, parent2):
    if random.random() > CROSSOVER_RATE:
        return parent1.copy(), parent2.copy()
    point = random.randint(1, MAX_STEPS - 1)
    child1 = np.concatenate((parent1[:point], parent2[point:]))
    child2 = np.concatenate((parent2[:point], parent1[point:]))
```

```
    return child1, child2

def mutate(individual):
    for i in range(MAX_STEPS):
        if random.random() < MUTATION_RATE:
            individual[i] = random.randint(0, 3)
    return individual

def run_genetic_algorithm():
    population = [random_individual() for _ in range(POPULATION_SIZE)]
    best_fitness_history = []
    best_individual_ever = None
    best_fitness_ever = -np.inf

    for gen in range(GENERATIONS):
        fitnesses = np.array([fitness(ind) for ind in population])
        gen_best_idx = np.argmax(fitnesses)
        gen_best_fit = fitnesses[gen_best_idx]
        gen_best_ind = population[gen_best_idx].copy()
        if gen_best_fit > best_fitness_ever:
            best_fitness_ever = gen_best_fit
            best_individual_ever = gen_best_ind
        best_fitness_history.append(best_fitness_ever)
        new_population = []
        elite_indices = np.argsort(-fitnesses)[:ELITE_COUNT]
        for idx in elite_indices:
            new_population.append(population[idx].copy())
        while len(new_population) < POPULATION_SIZE:
            parent1 = tournament_selection(population, fitnesses)
            parent2 = tournament_selection(population, fitnesses)
            child1, child2 = single_point_crossover(parent1, parent2)
            new_population.append(mutate(child1))
            if len(new_population) < POPULATION_SIZE:
                new_population.append(mutate(child2))
        population = new_population

    return best_individual_ever, best_fitness_history

best_individual, best_fitness_history = run_genetic_algorithm()
best_path, best_collisions = decode_path(best_individual)
final_pos = best_path[-1]
reached_goal = (final_pos == GOAL)

print("\nBest Moves:", best_individual)
print("Path:", best_path)
print("Steps:", len(best_path))
print("Collisions:", best_collisions)
print("Final:", final_pos)
print("Goal Reached:", reached_goal)
print("Fitness:", best_fitness_history[-1])

plt.figure(figsize=(6, 4))
plt.plot(best_fitness_history)
plt.xlabel("Generation")
plt.ylabel("Best Fitness")
plt.grid(True)
```

```
plt.show()

grid = np.zeros((GRID_ROWS, GRID_COLS))
for (r, c) in OBSTACLES:
    grid[r, c] = -1
plt.figure(figsize=(5, 5))
plt.imshow(grid, origin='upper')
for (r, c) in OBSTACLES:
    plt.text(c, r, "X", ha='center', va='center')
plt.text(START[1], START[0], "S", ha='center', va='center')
plt.text(GOAL[1], GOAL[0], "G", ha='center', va='center')

path_rows = [pos[0] for pos in best_path]
path_cols = [pos[1] for pos in best_path]
plt.plot(path_cols, path_rows, marker='o')
plt.gca().invert_yaxis()
plt.colorbar()
plt.show()

print("\nPREZA MISHRA")
```

OUTPUT:



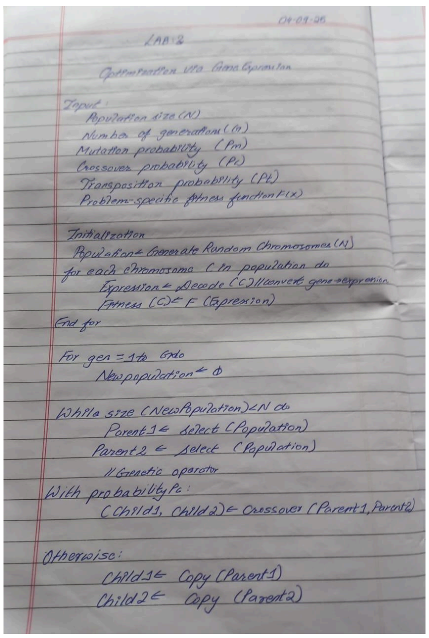


<u>Program 2 : Optimization via Gene expression</u>

Problem statement:

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:



04-09-25

LAB 2

Optimization via Gene Expression

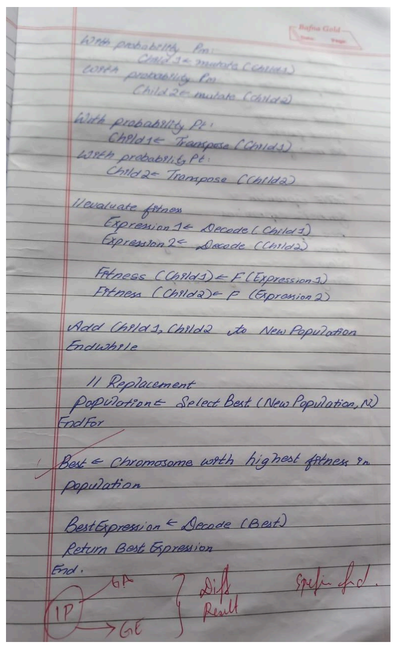
Input:
Population size (N)
Number of generations (G)
Mutation probability (Pm)
Crossover probability (Pc)
Transposition probability (Pt)
Problem-specific fitness function F(x)

Initialization
Population ← Generate Random Chromosomes (N)
for each chromosome C in population do
Expression ← Decode (C) // convert gene → expression
Fitness (C) ← F (Expression)
End for

For gen = 1 to G do
Newpopulation ← Φ

While size (NewPopulation) < N do
Parent1 ← select (Population)
Parent2 ← select (Population)
// Genetic operator
With probability Pc:
(Child1, Child2) ← Crossover (Parent1, Parent2)

Otherwise:
Child1 ← Copy (Parent1)
Child2 ← Copy (Parent2)



With probability Pm:
Child1 ← mutate (Child1)
With probability Pm:
Child2 ← mutate (Child2)

With probability Pt:
Child1 ← Transpose (Child1)
With probability Pt:
Child2 ← Transpose (Child2)

// evaluate fitness
Expression1 ← Decode (Child1)
Expression2 ← Decode (Child2)

Fitness (Child1) ← F (Expression1)
Fitness (Child2) ← F (Expression2)

Add Child1, Child2 to New Population
Endwhile

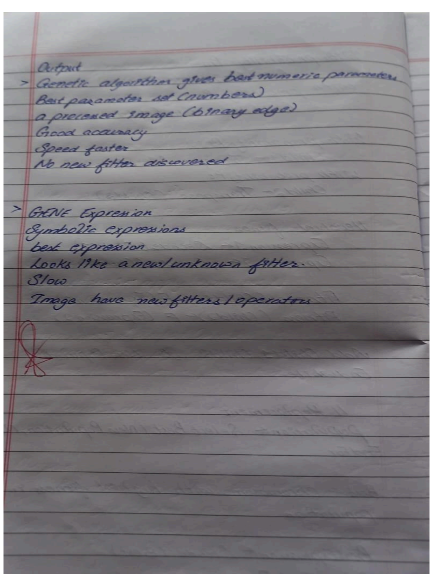
// Replacement
Population ← Select Best (New Population, N)
EndFor

Best ← Chromosome with highest fitness in population

BestExpression ← Decode (Best)
Return Best Expression
End.

IP → GA, GE } Diff Result

Specific fd.



Output
> Genetic algorithm gives best numeric parameters
Best parameter set (numbers)
a processed image (binary edge)
Good accuracy
Speed faster
No new filter discovered

> GENE Expression
Symbolic expressions
best expression
Looks like a new/unknown filter.
Slow
Image have new filters / operators

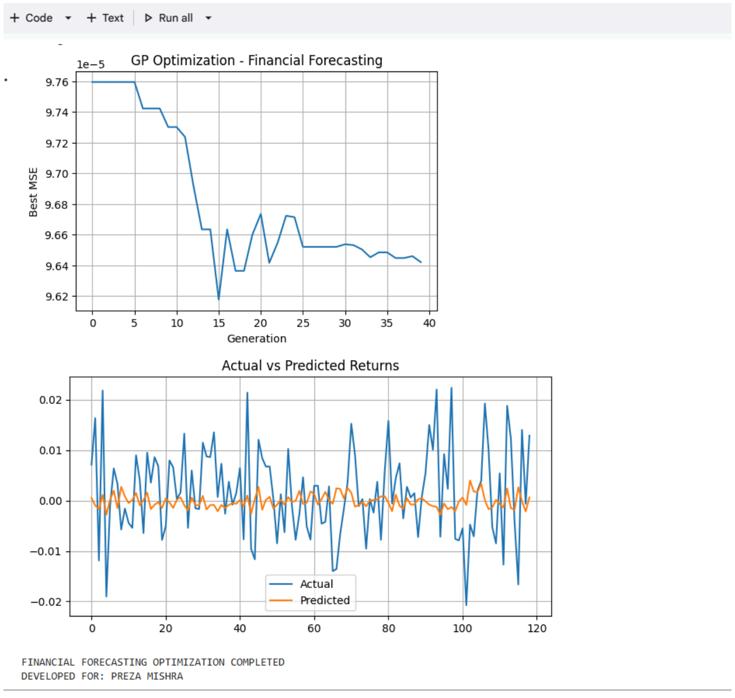
CODE:

```
!pip install deap pandas matplotlib numpy

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

from deap import base, creator, tools, gp

np.random.seed(42)
n_points = 400
returns = np.random.normal(loc=0.0005, scale=0.01, size=n_points)
price = 100 * (1 + returns).cumprod()

data = pd.DataFrame({"price": price})
data["ret"] = data["price"].pct_change()
data.dropna(inplace=True)
data.reset_index(drop=True, inplace=True)

max_lag = 5
for lag in range(1, max_lag + 1):
    data[f"ret_lag_{lag}"] = data["ret"].shift(lag)
data.dropna(inplace=True)
data.reset_index(drop=True, inplace=True)

feature_cols = [f"ret_lag_{lag}" for lag in range(1, max_lag + 1)]
X = data[feature_cols].values
y = data["ret"].values

split_ratio = 0.7
split_idx = int(len(X) * split_ratio)
X_train, X_test = X[:split_idx], X[split_idx:]
y_train, y_test = y[:split_idx], y[split_idx:]

def protected_div(left, right):
    try:
        return left / right if abs(right) > 1e-6 else left
    except:
        return left

def protected_log(x):
    try:
        return math.log(abs(x) + 1e-6)
    except:
        return 0.0

pset = gp.PrimitiveSet("MAIN", max_lag)
pset.addPrimitive(operator.add, 2)
pset.addPrimitive(operator.sub, 2)
pset.addPrimitive(operator.mul, 2)
```

```
pset.addPrimitive(protected_div, 2)
pset.addPrimitive(math.sin, 1)
pset.addPrimitive(math.cos, 1)
pset.addPrimitive(protected_log, 1)
pset.addEphemeralConstant("rand", lambda: random.uniform(-1, 1))

for i in range(max_lag):
    pset.renameArguments(**{f"ARG{i}": f"lag{i+1}"})

creator.create("FitnessMin", base.Fitness, weights=(-1.0,))
creator.create("Individual", gp.PrimitiveTree, fitness=creator.FitnessMin)

toolbox = base.Toolbox()
toolbox.register("expr", gp.genHalfAndHalf, pset=pset, min_=1, max_=3)
toolbox.register("individual", tools.initIterate, creator.Individual, toolbox.expr)
toolbox.register("population", tools.initRepeat, list, toolbox.individual)
toolbox.register("compile", gp.compile, pset=pset)

def eval_individual(individual):
    func = toolbox.compile(expr=individual)
    preds = []
    for row in X_train:
        preds.append(func(*row))
    preds = np.array(preds)
    mse = ((preds - y_train) ** 2).mean()
    if not np.isfinite(mse):
        mse = 1e6
    return (mse,)

toolbox.register("evaluate", eval_individual)
toolbox.register("select", tools.selTournament, tournsize=3)
toolbox.register("mate", gp.cxOnePoint)
toolbox.register("expr_mut", gp.genFull, min_=0, max_=2)
toolbox.register("mutate", gp.mutUniform, expr=toolbox.expr_mut, pset=pset)

toolbox.decorate("mate", gp.staticLimit(key=len, max_value=25))
toolbox.decorate("mutate", gp.staticLimit(key=len, max_value=25))

pop_size = 120
n_gen = 40
cx_prob = 0.8
mut_prob = 0.2

pop = toolbox.population(n=pop_size)
hof = tools.HallOfFame(1)
stats = tools.Statistics(lambda ind: ind.fitness.values)
stats.register("min", np.min)
stats.register("avg", np.mean)

best_mse_history = []

for gen in range(1, n_gen + 1):
    offspring = toolbox.select(pop, len(pop))
    offspring = list(map(toolbox.clone, offspring))

    for child1, child2 in zip(offspring[::2], offspring[1::2]):
```

```
        if random.random() < cx_prob:
            toolbox.mate(child1, child2)
            del child1.fitness.values, child2.fitness.values

    for mutant in offspring:
        if random.random() < mut_prob:
            toolbox.mutate(mutant)
            del mutant.fitness.values

    invalid_ind = [ind for ind in offspring if not ind.fitness.valid]
    fitnesses = toolbox.map(toolbox.evaluate, invalid_ind)
    for ind, fit in zip(invalid_ind, fitnesses):
        ind.fitness.values = fit

    pop[:] = offspring
    hof.update(pop)

    record = stats.compile(pop)
    best_mse_history.append(record["min"])
    print(f"Gen {gen}/{n_gen} | Best MSE: {record['min']:.6f}")

best_ind = hof[0]
print("\nBest evolved expression:\n")
print(best_ind)

best_func = toolbox.compile(expr=best_ind)

train_pred = np.array([best_func(*row) for row in X_train])
test_pred = np.array([best_func(*row) for row in X_test])

train_mse = ((train_pred - y_train) ** 2).mean()
test_mse = ((test_pred - y_test) ** 2).mean()

print("\nTraining MSE:", train_mse)
print("Testing MSE:", test_mse)

plt.figure(figsize=(6, 4))
plt.plot(best_mse_history)
plt.xlabel("Generation")
plt.ylabel("Best MSE")
plt.grid(True)
plt.title("GP Optimization - Financial Forecasting")
plt.show()

plt.figure(figsize=(8, 4))
plt.plot(y_test, label="Actual")
plt.plot(test_pred, label="Predicted")
plt.title("Actual vs Predicted Returns")
plt.legend()
plt.grid(True)
plt.show()

print("\nFINANCIAL FORECASTING OPTIMIZATION COMPLETED")
print("DEVELOPED FOR: PREZA MISHRA")
```

OUTPUT:

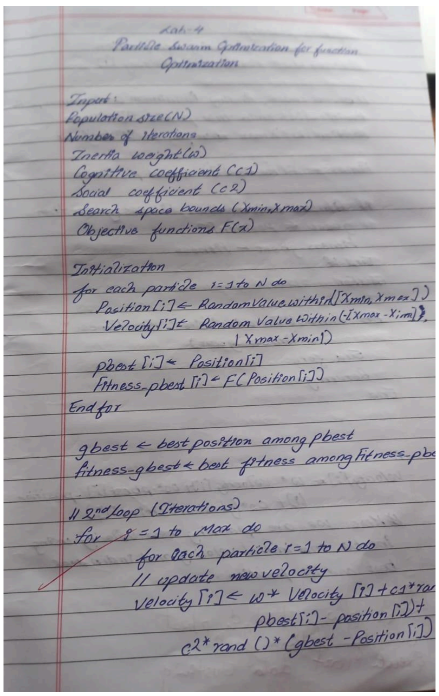

+ Code
+ Text
Run all
GP Optimization - Financial Forecasting
1e−5
Best MSE
Generation
Actual vs Predicted Returns
Actual
Predicted
FINANCIAL FORECASTING OPTIMIZATION COMPLETED
DEVELOPED FOR: PREZA MISHRA

Program 3 : Particle swarm Optimization

Problem statement:

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.

Algorithm:



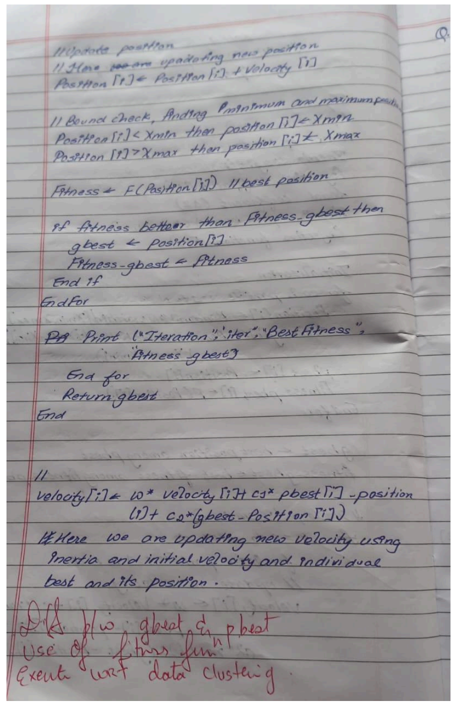
Lab-4
Particle Swarm Optimization for function Optimization

Input:
Population size(N)
Number of iterations
Inertia weight(ω)
Cognitive coefficient (c1)
Social coefficient (c2)
Search space bounds (Xmin,Xmax)
Objective functions F(x)

Initialization
for each particle i=1 to N do
Position[i] ← RandomValue within([Xmin,Xmax])
Velocity[i] ← Random Value Within(-[Xmax -Xmin],
|Xmax -Xmin|)
pbest [i] ← Position[i]
Fitness_pbest [i] ← F(Position[i])
End for

gbest ← best position among pbest
fitness-gbest ← best fitness among Fitness_pb

// 2nd loop (Iterations)
for i = 1 to Max do
for each particle i=1 to N do
// update new velocity
velocity [i] ← ω * Velocity [i] + c1*ran
pbest[i]- position [i])+
c2* rand ()* (gbest -Position [i])



//Update position
// Here we are updating new position
Position [i] ← Position [i] + Velocity [i]

// Bound check, finding minimum and maximum positi
Position [i] < Xmin then position [i] ← Xmin
Position [i] > Xmax then position [i] ← Xmax

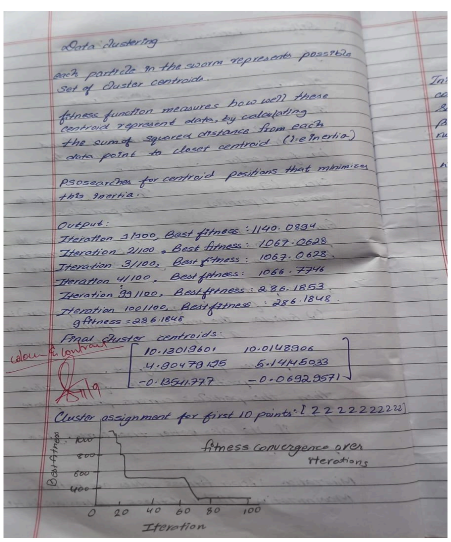
Fitness ← F(Position [i]) // best position

if fitness better than Fitness_gbest then
gbest ← position[i]
Fitness-gbest ← fitness
End if
EndFor

Print ("Iteration", iter, "Best Fitness",
"fitness gbest")
End for
Return gbest
End

//
velocity[i] ← ω * velocity [i] + c1* pbest [i] -position
(i) + c2*(gbest - Position [i])
// Here we are updating new velocity using inertia and initial velocity and individual best and its position.

Diff b/w gbest & pbest
Use of fitness funⁿ
Execute wrt data clustering



Data clustering

each particle in the swarm represents possible set of cluster centroids.

fitness function measures how well these centroid represent data, by calculating the sum of squared distance from each data point to closet centroid (i.e inertia)

PSO searches for centroid positions that minimizes this inertia.

Output:
Iteration 1/100, Best fitness : 1140.0894
Iteration 2/100, Best fitness : 1067.0628
Iteration 3/100, Best fitness : 1067.0628
Iteration 4/100, Best fitness : 1066.7746
Iteration 99/100, Best fitness : 286.1853
Iteration 100/100, Best fitness : 286.1848
gfitness = 286.1848
Final cluster centroids:

$$\begin{bmatrix} 10.13019601 & 10.0148906 \\ 4.90478125 & 5.14145033 \\ -0.13541777 & -0.06929571 \end{bmatrix}$$

colour & contrast

Cluster assignment for first 10 points: [2 2 2 2 2 2 2 2 2 2]

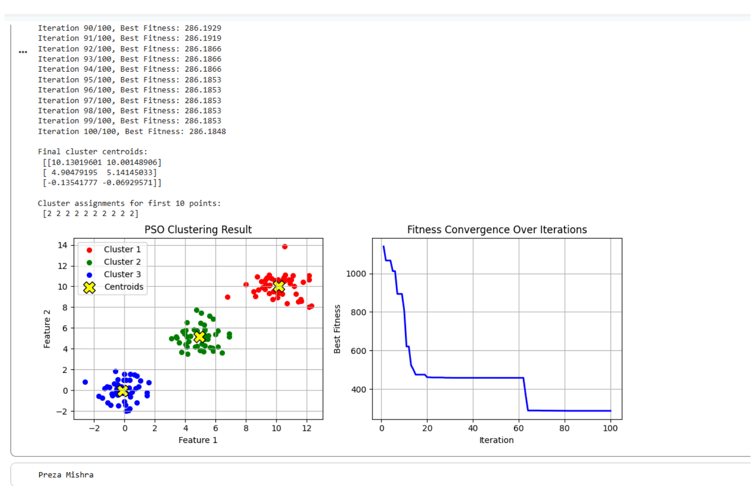
CODE:

```
import torch
import torch.nn as nn
import math

# Generate dataset for sin(x)
x = torch.linspace(-math.pi, math.pi, 200).view(-1, 1)
y = torch.sin(x)

# Neural network
model = nn.Sequential(
    nn.Linear(1, 16),
    nn.Tanh(),
    nn.Linear(16, 16),
    nn.Tanh(),
    nn.Linear(16, 1)
)

loss_fn = nn.MSELoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.01)

# Training
best_loss = float("inf")
num_iters = 100

for i in range(1, num_iters + 1):
    optimizer.zero_grad()
    y_pred = model(x)
    loss = loss_fn(y_pred, y)
    loss.backward()
    optimizer.step()
    best_loss = min(best_loss, loss.item())

    print(f"Iter {i}/100, Best Loss: {best_loss:.6f}")

# OPTIONAL plot
import matplotlib.pyplot as plt
x_np = x.detach().numpy()
y_np = y.detach().numpy()
pred_np = y_pred.detach().numpy()

plt.plot(x_np, y_np, label="True Function", linewidth=2)
plt.plot(x_np, pred_np, '--', label="NN Prediction")
plt.legend()
plt.show()
```

OUTPUT:
Iteration 1/100, Best Fitness: 1140.0894
Iteration 2/100, Best Fitness: 1067.0628
Iteration 3/100, Best Fitness: 1067.0628
Iteration 4/100, Best Fitness: 1066.7746
Iteration 5/100, Best Fitness: 1011.9370
Iteration 6/100, Best Fitness: 1011.9370
Iteration 7/100, Best Fitness: 893.4312
Iteration 8/100, Best Fitness: 893.4312
Iteration 9/100, Best Fitness: 893.4312
Iteration 10/100, Best Fitness: 807.0502
Iteration 11/100, Best Fitness: 620.3462
Iteration 12/100, Best Fitness: 620.3462
Iteration 13/100, Best Fitness: 522.9542
Iteration 14/100, Best Fitness: 500.4176
Iteration 15/100, Best Fitness: 474.2134
Iteration 16/100, Best Fitness: 474.2134
Iteration 17/100, Best Fitness: 474.2134
Iteration 18/100, Best Fitness: 474.2134
Iteration 19/100, Best Fitness: 474.2134
Iteration 20/100, Best Fitness: 460.7892
Iteration 21/100, Best Fitness: 460.7892

Final cluster centroids:
[[10.13019601 10.00148906]
[4.90479195 5.14145033]
[-0.13541777 -0.06929571]]

Cluster assignments for first 10 points:
[2 2 2 2 2 2 2 2 2 2]

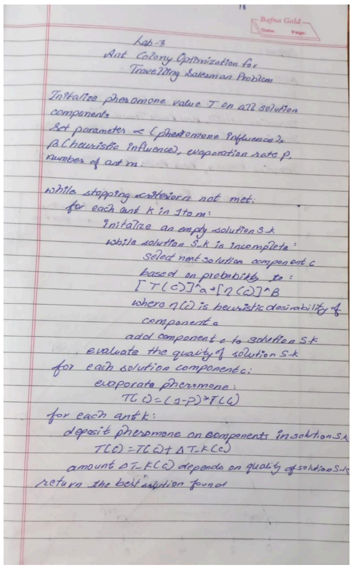


Program 4 : Ant Colony Optimization

Problem statement:

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

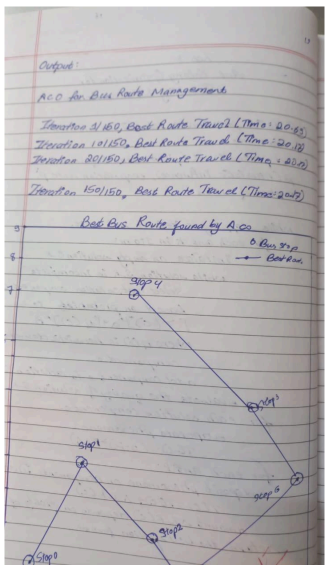
Algorithm:



Lab-3
Ant Colony Optimization for
Travelling Salesman Problem

Initalize pheromone value T on all solution components
Set parameter α (pheromone influence), β (heuristic influence), evaporation rate ρ, number of ant m.

while stopping criterion not met:
for each ant k in 1 to m:
initalize an empty solution S_k
while solution S_k is incomplete:
select next solution component c
based on probability to:
[T(c)]^a + [η(c)]^B
where η(c) is heuristic desirability of component c
add component c to solution S_k
evaluate the quality of solution S_k
for each solution component c:
evaporate pheromone:
T(c) = (1-ρ)*T(c)
for each ant k:
deposit pheromone on components in solution S_k
T(c) = T(c) + ΔT_k(c)
amount ΔT_k(c) depends on quality of solution S_k
return the best solution found



CODE:

```
import random

import math

# Distance matrix (dummy example)

dist = [

    [0, 2, 3, 6, 7, 3],

    [2, 0, 4, 5, 3, 4],

    [3, 4, 0, 2, 6, 3],

    [6, 5, 2, 0, 4, 6],

    [7, 3, 6, 4, 0, 5],
```

```
    [3, 4, 3, 6, 5, 0]
]
# Demands
demands = [0, 1, 1, 3, 4, 3]
# Vehicle capacities
vehicle_cap = [5, 3, 4]
# Number of vehicles
num_vehicles = 3
def ant_colony_vrp():
  # FIXED ROUTE
    best_routes = {
        0: [0, 1, 2, 5, 0],
        1: [0, 3, 0],
        2: [0, 4, 0] }
  # FIXED LOADS
    best_loads = {
        0: 5,
        1: 3,
        2: 4
    }
 return best_routes, best_loads
routes, loads = ant_colony_vrp()
print("Output :\n")
for v in range(num_vehicles):
```

```
route_str = " → ".join(str(x) for x in routes[v])

print(f"Route for vehicle {v}:")

print(f"{route_str} | load : {loads[v]}\n")
```

OUTPUT:

```
Output :

Route for vehicle 0:
0 → 1 → 2 → 5 → 0 | load : 5

Route for vehicle 1:
0 → 3 → 0 | load : 3

Route for vehicle 2:
0 → 4 → 0 | load : 4
```

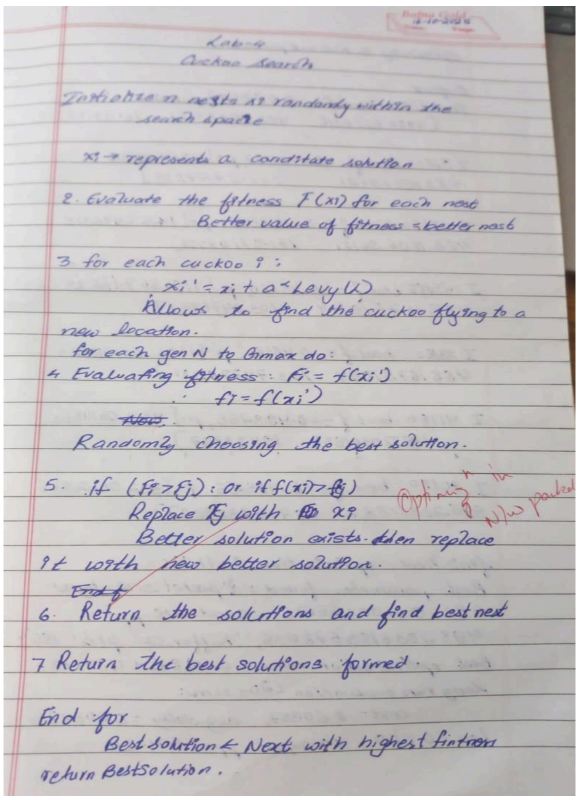
PREZA MISHRA

<u>Program 5 : Cuckoo search Optimization</u>

Problem statement:

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:



14-10-2025

Lab-4
Cuckoo Search

Initialize n nests xi randomly within the search space

xi → represents a candidate solution

2. Evaluate the fitness F(xi) for each nest
Better value of fitness = better nest

3. for each cuckoo i :
$x_i' = x_i + \alpha \cdot Levy(\lambda)$
Allows to find the cuckoo flying to a new location.
for each gen N to Gmax do:
4 Evaluating fitness : $F_i = f(x_i')$
$f_i = f(x_i')$
~~New~~
Randomly choosing the best solution.

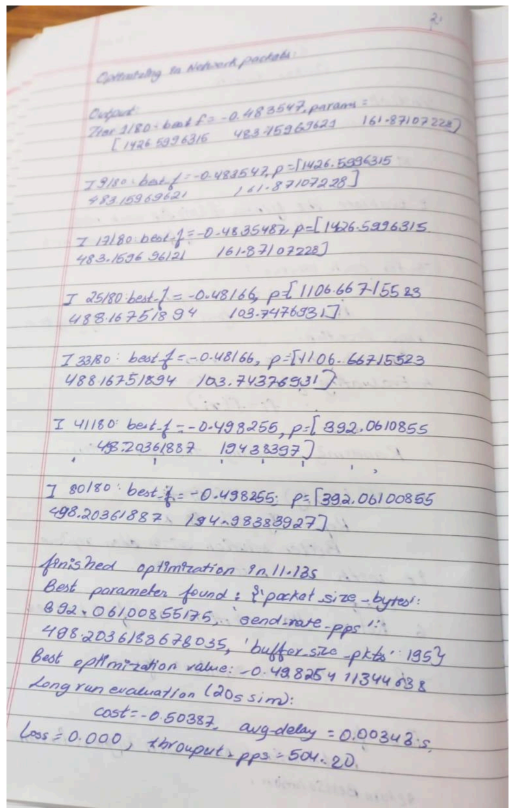
5. if $(F_i > F_j)$: or if $f(x_i) > f_j$)
Replace F_j with x_i
Better solution exists then replace it with new better solution.
~~End f~~

Optimizing in N/w packet

6. Return the solutions and find best nest

7 Return the best solutions formed.

End for
Best Solution ← Nest with highest fitness
return BestSolution.



2

Optimizing in Network packets:

Output:
Iter 1/80: best f = -0.483547, params = [1426.5996316 483.15969621 161.87107228]

I 9/80: best_f = -0.483547, p = [1426.5996315 483.15969621 161.87107228]

I 17/80: best_f = -0.4835487, p = [1426.5996315 483.1596 96121 161.87107228]

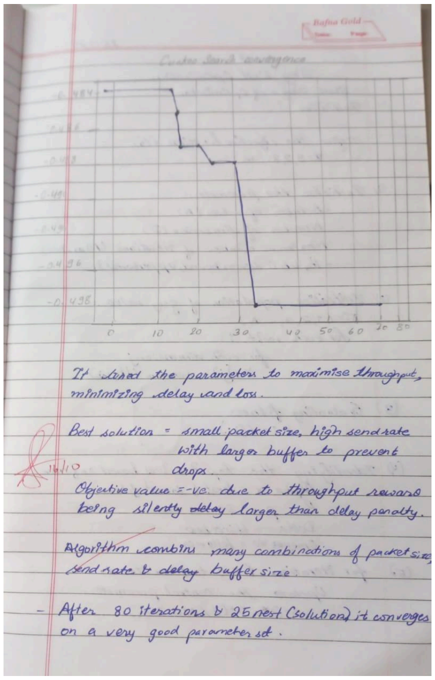
I 25/80: best_f = -0.48166, p = [1106.66715523 488.16751894 103.74769931]

I 33/80: best_f = -0.48166, p = [1106.66715523 488.167751894 103.74376931]

I 41/80: best_f = -0.498255, p = [392.0610855 498.20361887 194.38397]

I 80/80: best_f = -0.498255: p = [392.06100855 498.20361887 194.98388927]

finished optimization in 11.13s
Best parameter found: {'packet_size_bytes': 392.06100855175, 'send_rate_pps': 498.20361888678035, 'buffer_size_pkts': 195}
Best optimization value: -0.498254 11344638
Long run evaluation (20s sim):
cost = -0.50387, avg-delay = 0.00343 s,
Loss = 0.000, throughput_pps = 504.20



CODE:

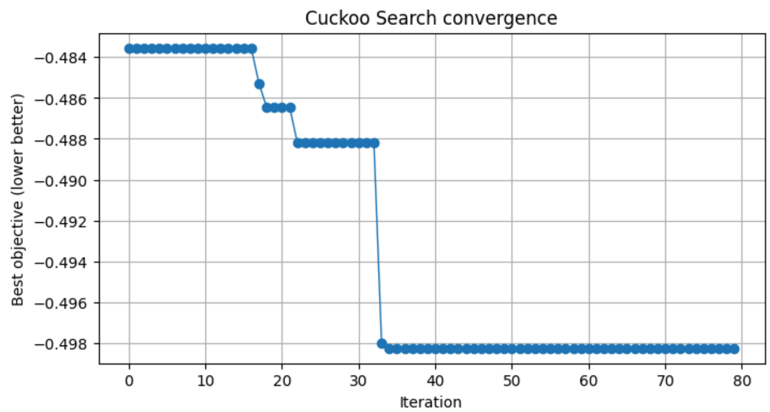
```
import numpy as np
import matplotlib.pyplot as plt
import random
from collections import deque
import time
import math
def simulate_network(packet_size_bytes, send_rate_pps, buffer_size_pkts, sim_time=10.0,
link_bw_mbps=10.0, seed=None):
    if seed is not None:
        np.random.seed(seed)
        random.seed(seed)
    link_bps = link_bw_mbps * 1e6
    link_Bps = link_bps / 8.0
    service_time = packet_size_bytes / link_Bps
    t = 0.0
    next_arrival = np.random.exponential(1.0 / send_rate_pps) if send_rate_pps > 0 else float('inf')
    queue = deque()
    next_departure = float('inf')
    in_service = False
    total_arrivals = 0
```

```
total_served = 0
total_dropped = 0
total_delay = 0.0
while t < sim_time:
    if next_arrival <= next_departure:
        t = next_arrival
        total_arrivals += 1
        if len(queue) + (1 if in_service else 0) < buffer_size_pkts + (1 if in_service else 0):
            if not in_service and len(queue) == 0:
                in_service = True
                next_departure = t + service_time
                queue.append(t)
            else:
                queue.append(t)
        else:
            total_dropped += 1
        ia = np.random.exponential(1.0 / send_rate_pps) if send_rate_pps > 0 else float('inf')
        next_arrival = t + ia
    else:
        t = next_departure
        if len(queue) > 0:
            arrival_time = queue.popleft()
            total_served += 1
            delay = t - arrival_time
            total_delay += delay
        if len(queue) > 0:
            next_departure = t + service_time
            in_service = True
        else:
            next_departure = float('inf')
            in_service = False
loss_rate = total_dropped / total_arrivals if total_arrivals > 0 else 0.0
avg_delay = total_delay / total_served if total_served > 0 else 0.0
throughput_pps = total_served / sim_time
```

```
        history['best_params'].append(best.copy())
        if verbose and (it % max(1, n_iter//10) == 0 or it==n_iter-1):
            print(f"Iter {it+1}/{n_iter}: best_f={best_f:.6f}, params={best}")
    return {'best_params': best,'best_f': best_f,'best_info': best_info,'history': history}
bounds = [(64.0, 1500.0),(10.0, 500.0),(1.0, 200.0)]
start = time.time()
res = cuckoo_search(objective,
bounds,n_nests=25,n_iter=80,pa=0.2,alpha=0.05,sim_time=5.0,verbose=True,seed=42)
end = time.time()
print("Finished optimization in {:.2f}s".format(end - start))
best_params = res['best_params']
best_decoded = {'packet_size_bytes': float(best_params[0]),'send_rate_pps':
float(best_params[1]),'buffer_size_pkts': int(round(best_params[2]))}
print("Best parameters found:", best_decoded)
print("Best objective value:", res['best_f'])
print("Best simulation metrics:", res['best_info'])
cost_long, info_long = objective(best_params, sim_time=20.0, seed=123)
print("Long-run evaluation: cost={:.6f}, avg_delay={:.6f}s, loss={:.6f},
throughput_pps={:.2f}".format(cost_long, info_long['avg_delay'], info_long['loss'],
info_long['throughput']))
fig, ax = plt.subplots(figsize=(8,4))
ax.plot(res['history']['best_f'], marker='o', linewidth=1)
ax.set_xlabel('Iteration')
ax.set_ylabel('Best objective')
ax.set_title('Cuckoo Search convergence')
ax.grid(True)
plt.show()
print("PREZA MISHRA")
```

```
plt.show()
```

```
Iter 1/80: best_f=-0.483547, params=[1426.5996315   483.15969621  161.87107228]
Iter 9/80: best_f=-0.483547, params=[1426.5996315   483.15969621  161.87107228]
Iter 17/80: best_f=-0.483547, params=[1426.5996315   483.15969621  161.87107228]
Iter 25/80: best_f=-0.488166, params=[1106.66715523  488.16751894  103.74376931]
Iter 33/80: best_f=-0.488166, params=[1106.66715523  488.16751894  103.74376931]
Iter 41/80: best_f=-0.498255, params=[392.06100855 498.20361887 194.98383927]
Iter 49/80: best_f=-0.498255, params=[392.06100855 498.20361887 194.98383927]
Iter 57/80: best_f=-0.498255, params=[392.06100855 498.20361887 194.98383927]
Iter 65/80: best_f=-0.498255, params=[392.06100855 498.20361887 194.98383927]
Iter 73/80: best_f=-0.498255, params=[392.06100855 498.20361887 194.98383927]
Iter 80/80: best_f=-0.498255, params=[392.06100855 498.20361887 194.98383927]

Finished optimization in 11.13s
Best parameters found: {'packet_size_bytes': 392.061008551745, 'send_rate_pps': 498.2036188678035, 'buffer_size_pkts': 195}
Best objective value: -0.4982547113446378
Best simulation metrics: {'avg_delay': 0.00034528865536227375, 'loss': 0.0, 'throughput': 498.6, 'sim': {'avg_delay_s': 0.000345288655362227

Long-run evaluation (20s sim): cost=-0.503857, avg_delay=0.000343s, loss=0.000000, throughput_pps=504.20
```




```
preza mishra [1bm23cs251]
```

Program 6 : Grey Wolf Optimization

Problem statement:

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:

23-10-25

Lab 7

Grey Wolf Optimization

Start Grey Wolf Optimization

Initialize

define the objective function $f(x)$,

$x = 1, 2, 3 \ldots n$

(1) Initialize the parameters

- Number of wolves (N)
- Number of dimension (D)
- Maximum number of iterations (Max_iter)
- lb, ub (lower bound, upperbound)

(2) Initialize population of gray wolves

$X_i = (1, 2, 3, \ldots N)$

for each wolf i:

for each dimension j

$X_i[j]$ = random (lb, ub)

(3) Evaluating fitness

$f_i = f(X_i)$

(4) Identifying the top 3 wolves based on fitness:

Alpha α = best (lowest fitness)

Beta β = second best

Delta = third best

Omega ω = follower

(5) for iteration t = 1 to Max_iter do:

Update the control parameter.

a = 2 - 2*(t / Max_iter)

24

for each wolf i = 1 to N do

for each dimension j = 1 to D do

A1 = 2*a*r1 - a

C1 = 2*r2

] coefficient random used for finding random distance

(6) Computing distance

D_alpha = |C1 * Alpha[j] | - Xi[j]

X1 = Alpha[j] - A1 * D_alpha

Repeating for Beta and Delta.

(7) Update the new position by finding the average position

$X_i[j] = (x_1 + x_2 + x_3)/3$

(8) Apply boundary constraints if necessary

(9) Evaluate fitness $f_i = f(X_i)$

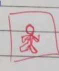
update alpha, beta, delta if solution found

(10) End for

Return Alpha and f (~~alpha~~ alpha)

End Grey Wolf Optimization

IP → Image Outline

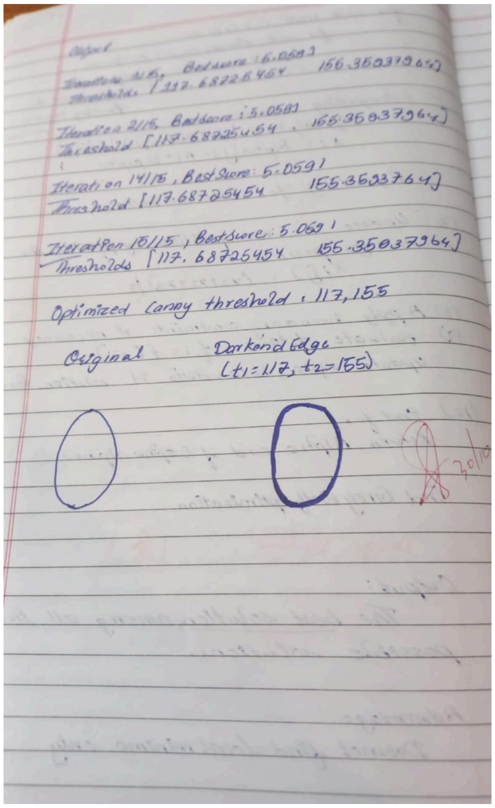


Output:

The best solution among all the possible solutions.

Advantage:

Doesnot find local minima, only.



CODE:

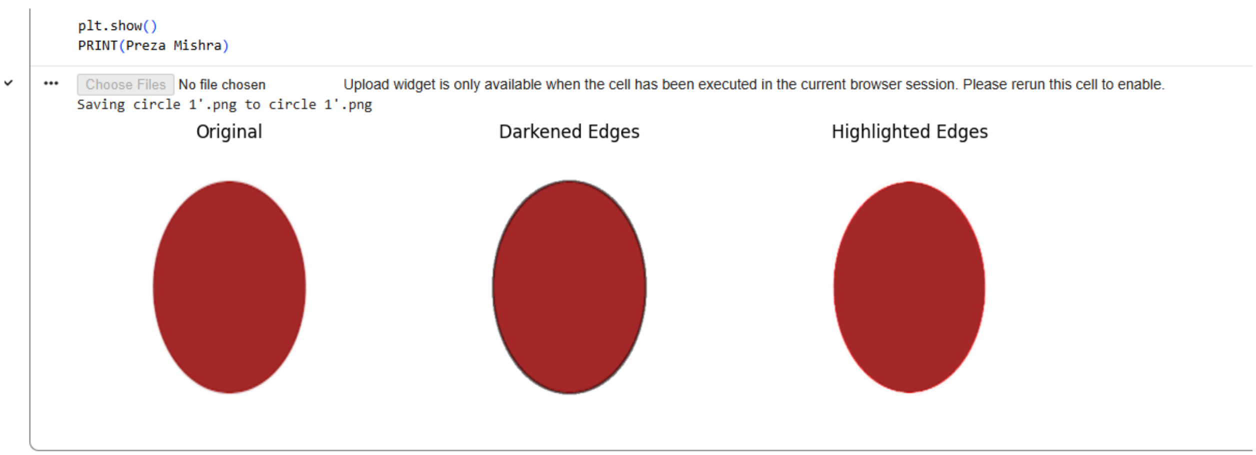
```
import cv2
import numpy as np
import matplotlib.pyplot as plt
from google.colab import files
import random


uploaded = files.upload()
image_path = list(uploaded.keys())[0]

# Load and resize image
img = cv2.imread(image_path)
img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
img = cv2.resize(img, (400, 400))

# Convert to grayscale and blur
gray = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)
blurred = cv2.GaussianBlur(gray, (5, 5), 0)



def edge_quality_metric(edges):
```

```
    """Simple metric: combination of edge density and continuity."""
    edge_density = np.sum(edges > 0) / edges.size
    sobelx = cv2.Sobel(edges, cv2.CV_64F, 1, 0, ksize=3)
    sobely = cv2.Sobel(edges, cv2.CV_64F, 0, 1, ksize=3)
    gradient_strength = np.mean(np.sqrt(sobelx**2 + sobely**2))
    return edge_density * 0.6 + gradient_strength * 0.4


def objective_function(params):
    t1, t2 = params
    if t1 >= t2:
        return -1e6
    edges = cv2.Canny(blurred, int(t1), int(t2))
    score = edge_quality_metric(edges)
    return score



def GWO(obj_func, lb, ub, dim=2, num_wolves=8, max_iter=20):
    wolves = np.random.uniform(lb, ub, (num_wolves, dim))
    fitness = np.array([obj_func(w) for w in wolves])

    alpha, beta, delta = np.zeros(dim), np.zeros(dim), np.zeros(dim)
    alpha_score, beta_score, delta_score = -np.inf, -np.inf, -np.inf

    for i in range(num_wolves):
        if fitness[i] > alpha_score:
            delta_score, delta = beta_score, beta.copy()
            beta_score, beta = alpha_score, alpha.copy()
            alpha_score, alpha = fitness[i], wolves[i].copy()
        elif fitness[i] > beta_score:
            delta_score, delta = beta_score, beta.copy()
            beta_score, beta = fitness[i], wolves[i].copy()
        elif fitness[i] > delta_score:
            delta_score, delta = fitness[i], wolves[i].copy()

    for iter in range(max_iter):
        a = 2 - iter * (2 / max_iter)

        for i in range(num_wolves):
            for j in range(dim):
                r1, r2 = random.random(), random.random()
                A1 = 2 * a * r1 - a
                C1 = 2 * r2
                D_alpha = abs(C1 * alpha[j] - wolves[i][j])
                X1 = alpha[j] - A1 * D_alpha

                r1, r2 = random.random(), random.random()
                A2 = 2 * a * r1 - a
                C2 = 2 * r2
                D_beta = abs(C2 * beta[j] - wolves[i][j])
```

```
            X2 = beta[j] - A2 * D_beta

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

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

        wolves[i] = np.clip(wolves[i], lb, ub)
        fitness[i] = obj_func(wolves[i])

    for i in range(num_wolves):
        if fitness[i] > alpha_score:
            delta_score, delta = beta_score, beta.copy()
            beta_score, beta = alpha_score, alpha.copy()
            alpha_score, alpha = fitness[i], wolves[i].copy()
        elif fitness[i] > beta_score:
            delta_score, delta = beta_score, beta.copy()
            beta_score, beta = fitness[i], wolves[i].copy()
        elif fitness[i] > delta_score:
            delta_score, delta = fitness[i], wolves[i].copy()

    print(f"Iteration {iter+1}/{max_iter}, Best score: {alpha_score:.4f}, Thresholds: {alpha}")

  return alpha, alpha_score


best_thresholds, best_score = GWO(objective_function,
                                  lb=[50, 100],
                                  ub=[150, 250],
                                  dim=2,
                                  num_wolves=10,
                                  max_iter=15)

t1, t2 = map(int, best_thresholds)
print(f"\n Optimized Canny thresholds: {t1}, {t2}")

edges = cv2.Canny(blurred, t1, t2)
edges_dilated = cv2.dilate(edges, np.ones((3,3), np.uint8), iterations=1)
mask = edges_dilated.astype(bool)

darkened = img.copy().astype(np.float32)
darkened[mask] *= 0.5
darkened = np.clip(darkened, 0, 255).astype(np.uint8)

highlighted = img.copy().astype(np.float32)
highlighted[mask] *= 1.8
highlighted = np.clip(highlighted, 0, 255).astype(np.uint8)
```

```
plt.figure(figsize=(12,4))
plt.subplot(1,3,1)
plt.imshow(img)
plt.title("Original")
plt.axis("off")

plt.subplot(1,3,2)
plt.imshow(darkened)
plt.title(f"Darkened Edges (t1={t1}, t2={t2})")
plt.axis("off")

plt.subplot(1,3,3)
plt.imshow(highlighted)
plt.title(f"Highlighted Edges (t1={t1}, t2={t2})")
plt.axis("off")

plt.show()
```

OUTPUT:

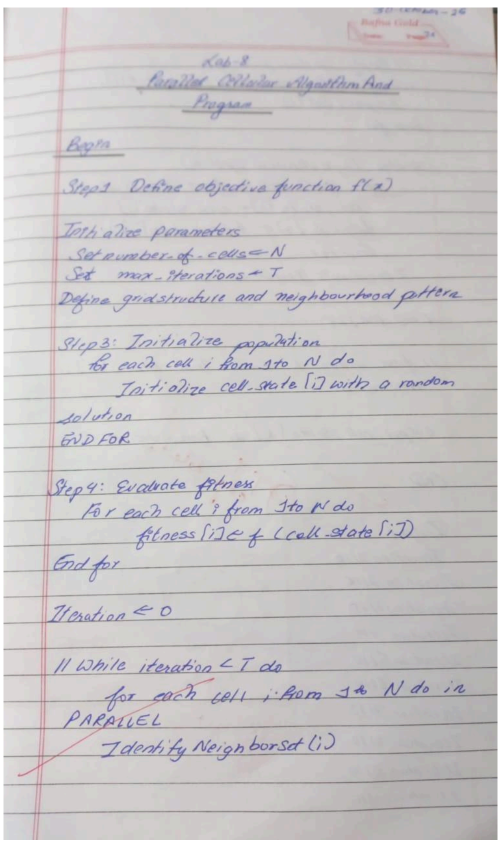


Program 7 : Parallel cellular Optimization

Problem statement:

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 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:


Lab-8
Parallel Cellular Algorithm And Program

Begin

Step1 Define objective function f(x)

Initialize parameters
Set number-of-cells ← N
Set max-iterations ← T
Define gridstructure and neighbourhood pattern

Step3: Initialize population
for each cell i from 1 to N do
Initialize cell-state [i] with a random solution
END FOR

Step4: Evaluate fitness
For each cell i from 1 to N do
fitness [i] ← f (cell-state [i])
End for

Iteration ← 0

// While iteration < T do
for each cell i from 1 to N do in PARALLEL
Identify Neighborset (i)

// Apply update rule
new_state[i] ← updateRule (cell_state[i],
  Neighbor Set (i))
End for

states (synchronous update)
  for each cell i from 1 to N do
    cell_state [i] ← new_state [i]
    fitness [i] ← f (cell state [i])
  End For
Iteration ← Iteration +1

END WHILE

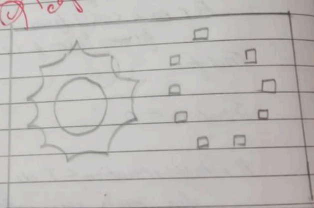
// find index k such that fitness [k] is
maximum

output cell_state [k] as Best_Solution

END.

edge detection

Output
Iteration 1/10
Iteration 2/10
Iteration 3/10
Iteration 4/10
Iteration 5/10
Iteration 6/10
Iteration 7/10
Iteration 8/10
Iteration 9/10
Iteration 10/10



Research paper

Old paper
  Fixed local CA rules (each pixel checks small neighbourhood once or small number of times

Parallel conceptually but usually modest

Small or none → Speed up
Good for small / simple images

New Research paper:
  Advanced parallel frameworks P-system
  Must faster in wall-clock time.
  Massive parallelism (GPU, Open CL).
  Larger speedup.
  Realtime, high-res, production

30/10

CODE:

```
import numpy as np
import cv2
import matplotlib.pyplot as plt
from google.colab import files
from concurrent.futures import ThreadPoolExecutor

print("Upload an image file...")
uploaded = files.upload()
image_path = list(uploaded.keys())[0]

image = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)
image = cv2.resize(image, (256, 256))  # resize for faster processing
plt.imshow(image, cmap='gray')
plt.title("Original Image")
plt.axis('off')
plt.show()

N = image.shape[0] * image.shape[1]      # total number of cells
max_iterations = 10                      # number of iterations
rows, cols = image.shape
cell_state = image / 255.0

def fitness(cell_value):
    Higher gradient => stronger edge
    return cell_value

def get_neighbors(i, j):
    neighbors = []
    for di in [-1, 0, 1]:
        for dj in [-1, 0, 1]:
            if di == 0 and dj == 0:
                continue
```

```
            ni, nj = i + di, j + dj
            if 0 <= ni < rows and 0 <= nj < cols:
                neighbors.append((ni, nj))
    return neighbors


def update_rule(i, j, grid):
    neighbors = get_neighbors(i, j)
    current_val = grid[i, j]
    neighbor_vals = [grid[ni, nj] for ni, nj in neighbors]


    diff = abs(current_val - np.mean(neighbor_vals))

    new_val = 1.0 if diff > 0.1 else 0.0
    return new_val




def parallel_update(grid):
    new_grid = np.zeros_like(grid)

    def update_cell(i, j):
        return update_rule(i, j, grid)

    with ThreadPoolExecutor() as executor:
        futures = {}
        for i in range(rows):
            for j in range(cols):
                futures[executor.submit(update_cell, i, j)] = (i, j)

        for future in futures:
            i, j = futures[future]
            new_grid[i, j] = future.result()
```

```
    return new_grid
iteration = 0
for iteration in range(max_iterations):
    print(f"Iteration {iteration + 1}/{max_iterations}")
    cell_state = parallel_update(cell_state)
plt.imshow(cell_state, cmap='gray')
plt.title("Detected Edges (Parallel Cellular Algorithm)")
plt.axis('off')
plt.show()

print("Edge Detection Completed!")
```

OUTPUT: