VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB RECORD

Bio Inspired Systems (23CS5BSBIS)

Submitted by

Suhas H A(1BM22CS293)

in partial fulfillment for the award of the degree of

BACHELOR OF ENGINEERING
in
COMPUTER SCIENCE AND ENGINEERING

B.M.S. COLLEGE OF ENGINEERING
(Autonomous Institution under VTU)
BENGALURU-560019
Sep-2024 to Jan-2025

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

CERTIFICATE

This is to certify that the Lab work entitled " Bio Inspired Systems (23CS5BSBIS)" carried out by **Suhas H A(1BM22CS293),** who is a 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.

Sheetal Assistant Professor Department of CSE, BMSCE	Dr. Joythi S Nayak Professor & HOD Department of CSE, BMSCE

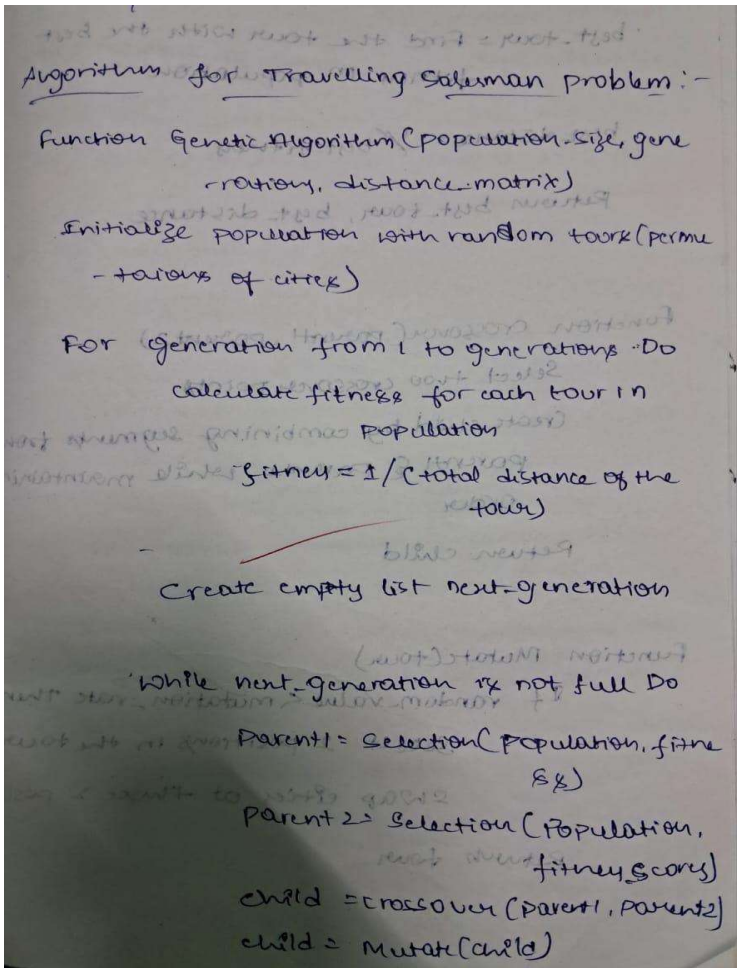
Index

Github Link : https://github.com/suhas1504/BIS_LAB_suhas_H_A

Program 1

Problem Statement : Optimize the allocation of a portfolio using a Genetic Algorithm to maximize the Sharpe Ratio, balancing expected returns and risk. Ensure the total asset allocation adheres to a fixed budget constraint.

Algorithm:



Algorithm for Travelling Saleman problem :-

Function Genetic Algorithm (population_size, gene -rations, distance_matrix)

Initialize population with random tours (permu -tations of cities)

For generation from 1 to generations Do

Calculate fitness for each tour in population

fitness = 1/(total distance of the tour)

Create empty list next_generation

While next_generation is not full Do

Parent1 = Selection(population, fitness)

Parent2 = Selection(population, fitness_scores)

child = Crossover(parent1, parent2)

child = Mutate(child)

Add child to next_generation

SET population = next_generation

best_tour = Find the tour with the best fitness in population

best_distance = 1 / best_fitness

Return best_tour, best_distance

Function Crossover(parent1, parent2)

Select two crossover points

Create child by combining segments from parent1 & parent2 while maintaining order

Return child

Function Mutate(tour)

If random_value < mutation_rate then

Select 2 positions in the tour

Swap cities at those 2 positions

Return tour

Function Selection(population, fitness_scores)

Select a candidate based on fitness scores

Return selected candidate

Enter number of cities:

Code:

```
# Generic algorithm -Portfolio Optimization
import numpy as np
import random

# Example data: expected returns and covariance matrix for assets
EXPECTED_RETURNS = [0.12, 0.18, 0.10, 0.07, 0.15]  # Annual returns
COVARIANCE_MATRIX = [
    [0.04, 0.02, 0.01, 0.03, 0.02],
    [0.02, 0.05, 0.02, 0.01, 0.03],
    [0.01, 0.02, 0.03, 0.02, 0.01],
    [0.03, 0.01, 0.02, 0.06, 0.04],
    [0.02, 0.03, 0.01, 0.04, 0.07]
]
BUDGET = 1.0  # Total budget (e.g., 100% allocation)
NUM_ASSETS = len(EXPECTED_RETURNS)
POPULATION_SIZE = 50
NUM_GENERATIONS = 100
MUTATION_RATE = 0.1

# Generate a random portfolio (chromosome)
def generate_chromosome():
    allocations = np.random.dirichlet(np.ones(NUM_ASSETS), size=1)[0]
    return allocations

# Fitness function: Sharpe Ratio = (Return - Risk-Free Rate) / Risk
def fitness(chromosome, risk_free_rate=0.03):
    portfolio_return = np.dot(chromosome, EXPECTED_RETURNS)
    portfolio_variance = np.dot(chromosome, np.dot(COVARIANCE_MATRIX, chromosome))
    portfolio_risk = np.sqrt(portfolio_variance)
    sharpe_ratio = (portfolio_return - risk_free_rate) / portfolio_risk
    return sharpe_ratio

# Selection: Roulette wheel selection
def select_population(population, fitness_scores):
    total_fitness = sum(fitness_scores)
    probabilities = [score / total_fitness for score in fitness_scores]
    selected = random.choices(population, probabilities, k=POPULATION_SIZE)
    return selected

# Crossover: Single-point crossover
def crossover(parent1, parent2):
    point = random.randint(1, NUM_ASSETS - 1)
    child = np.concatenate((parent1[:point], parent2[point:]))
    return child / sum(child)  # Normalize to ensure budget constraint

# Mutation: Small random change in allocation
```

```
def mutate(chromosome):
    if random.random() < MUTATION_RATE:
        idx = random.randint(0, NUM_ASSETS - 1)
        change = random.uniform(-0.05, 0.05)
        chromosome[idx] += change
        chromosome = np.clip(chromosome, 0, 1)  # Ensure allocations are between 0
and 1
        chromosome /= sum(chromosome)  # Normalize
    return chromosome

# Genetic Algorithm
def genetic_algorithm():
    population = [generate_chromosome() for _ in range(POPULATION_SIZE)]
    for generation in range(NUM_GENERATIONS):
        fitness_scores = [fitness(chromo) for chromo in population]
        print(f"Generation {generation}, Best fitness: {max(fitness_scores)}")
        if max(fitness_scores) > 2:  # Stop if high Sharpe Ratio is found
            break
        population = select_population(population, fitness_scores)
        next_generation = []
        for i in range(0, POPULATION_SIZE, 2):
            parent1, parent2 = population[i], population[i + 1]
            child1, child2 = crossover(parent1, parent2), crossover(parent2,
parent1)
            next_generation.extend([mutate(child1), mutate(child2)])
        population = next_generation
    return max(population, key=fitness)

# Run the Genetic Algorithm
best_portfolio = genetic_algorithm()
print("Best portfolio allocations:", best_portfolio)
print("Expected portfolio return:", np.dot(best_portfolio, EXPECTED_RETURNS))
print("Portfolio risk:", np.sqrt(np.dot(best_portfolio, np.dot(COVARIANCE_MATRIX,
best_portfolio))))


# Chromosome: Represents the proportion of budget allocated to each asset.
# Fitness Function: Uses the Sharpe Ratio to measure the trade-off between risk and
return.
# Constraints: Allocations are normalized to ensure the total budget is respected.
# Crossover: Merges two parent portfolios to create a new portfolio.
# Mutation: Randomly tweaks allocations to explore more solutions.
```

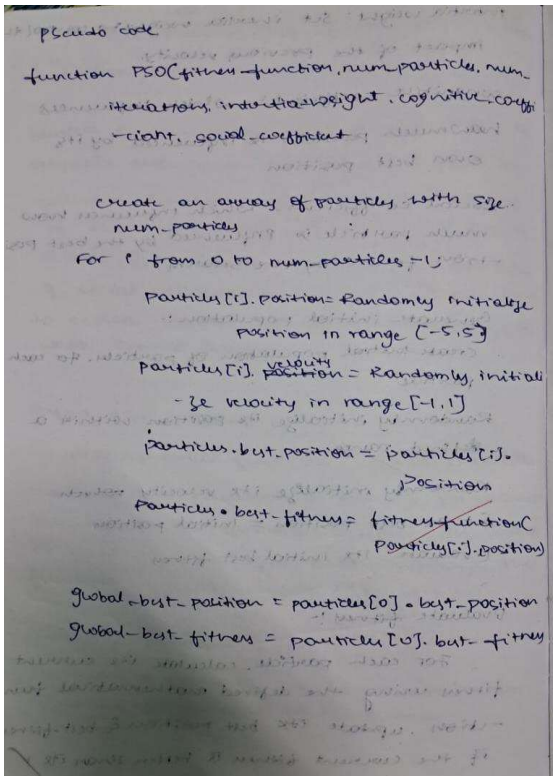
Output:

```
Generation 0, Best fitness: 0.695064742965254
Generation 1, Best fitness: 0.6909220321773359
Generation 2, Best fitness: 0.6869376258171495
Generation 3, Best fitness: 0.6738831283852085
Generation 4, Best fitness: 0.6723149241032417
Generation 5, Best fitness: 0.6688945471252489
Generation 6, Best fitness: 0.6666654608880371
Generation 7, Best fitness: 0.6686823287696594
Generation 8, Best fitness: 0.6417309204825173
Generation 9, Best fitness: 0.6444722817686085
Generation 10, Best fitness: 0.6440652631329423
Generation 11, Best fitness: 0.6626060356056972
Generation 12, Best fitness: 0.6574389737159604
Generation 13, Best fitness: 0.6748664629169678
Generation 14, Best fitness: 0.6760368964928318
Generation 15, Best fitness: 0.6569329742829026
Generation 16, Best fitness: 0.6423107748026713
Generation 17, Best fitness: 0.6419976296739576
Generation 18, Best fitness: 0.6444968459222964
Generation 19, Best fitness: 0.6470081800257046
Generation 20, Best fitness: 0.6469881151382244
Generation 21, Best fitness: 0.6422388297247027
Generation 22, Best fitness: 0.6513931195049387
Generation 23, Best fitness: 0.6390177148822977
Generation 24, Best fitness: 0.6388354988653773
Generation 25, Best fitness: 0.6149186634037511
Generation 26, Best fitness: 0.6170569086594664
Generation 27, Best fitness: 0.6169526137298017
Generation 28, Best fitness: 0.6170558495855434
Generation 29, Best fitness: 0.6253054198164103
Generation 30, Best fitness: 0.627473888192279
Generation 31, Best fitness: 0.6298654677679763
Generation 32, Best fitness: 0.6312002177680891
Generation 33, Best fitness: 0.631471993633289
Generation 34, Best fitness: 0.6378201596349676
Generation 35, Best fitness: 0.6426551636161763
Generation 36, Best fitness: 0.6425195892345336
Generation 37, Best fitness: 0.6425234872953706
Generation 38, Best fitness: 0.6435783344806354
Generation 39, Best fitness: 0.6365725938602643
Generation 40, Best fitness: 0.6337079566084003
Generation 41, Best fitness: 0.6336340335805741
Generation 42, Best fitness: 0.6248431198367109
Generation 43, Best fitness: 0.6177674056162533
Generation 44, Best fitness: 0.6117945182727395
Generation 45, Best fitness: 0.613754626632173
Generation 46, Best fitness: 0.6132264772900665
Generation 47, Best fitness: 0.6213604206998655
Generation 48, Best fitness: 0.6223340255622521
Generation 49, Best fitness: 0.6213812691829578
Generation 50, Best fitness: 0.6141759428483451
Generation 51, Best fitness: 0.6184540542910166
Generation 52, Best fitness: 0.6170986365834434
Generation 53, Best fitness: 0.6173504529884307
Generation 54, Best fitness: 0.617467688210354
Generation 55, Best fitness: 0.6119591856152575
Generation 56, Best fitness: 0.6103657876398093
Generation 57, Best fitness: 0.6144348817281562
Generation 58, Best fitness: 0.6110490056459755
Generation 59, Best fitness: 0.6133139022718151
Generation 60, Best fitness: 0.6140628528280487
Generation 61, Best fitness: 0.6235628104347493
Generation 62, Best fitness: 0.6153360177530592
Generation 63, Best fitness: 0.6189109870484316
Generation 64, Best fitness: 0.6128236518799614
Generation 65, Best fitness: 0.6095474381016422
Generation 66, Best fitness: 0.6189465516444184
Generation 67, Best fitness: 0.6158234800434268
Generation 68, Best fitness: 0.616192795831215
Generation 69, Best fitness: 0.6212622425947059
Generation 70, Best fitness: 0.6175388935984585
Generation 71, Best fitness: 0.6179718173733627
Generation 72, Best fitness: 0.6062755789306569
Generation 73, Best fitness: 0.6125186709320615
Generation 74, Best fitness: 0.6130094206573721
Generation 75, Best fitness: 0.6128761026343555
Generation 76, Best fitness: 0.6123181601081817
Generation 77, Best fitness: 0.6143219806761443
Generation 78, Best fitness: 0.6129578935114196
Generation 79, Best fitness: 0.6128414730504712
Generation 80, Best fitness: 0.6180706446542509
Generation 81, Best fitness: 0.6181748229594893
Generation 82, Best fitness: 0.6183453845901472
Generation 83, Best fitness: 0.6199808810492905
Generation 84, Best fitness: 0.6159897168541965
Generation 85, Best fitness: 0.6152723230634146
Generation 86, Best fitness: 0.610768786883817
Generation 87, Best fitness: 0.6107818551791
Generation 88, Best fitness: 0.6109803257039085
Generation 89, Best fitness: 0.6090702565516154
Generation 90, Best fitness: 0.6079481208919458
Generation 91, Best fitness: 0.6081901418750741
Generation 92, Best fitness: 0.6061818624656854
Generation 93, Best fitness: 0.6202208175441253
Generation 94, Best fitness: 0.6052548173475841
Generation 95, Best fitness: 0.6090917264043236
Generation 96, Best fitness: 0.6094681761511236
Generation 97, Best fitness: 0.6095915470845986
Generation 98, Best fitness: 0.6094841339237812
Generation 99, Best fitness: 0.6112895866618347
Best portfolio allocations: [0.20490548 0.20615414 0.25788912 0.10232249 0.22872877]
Expected portfolio return: 0.12895720418208578
Portfolio risk: 0.1595446718656624
```

Program 2

Problem Statement : Implement a Particle Swarm Optimization (PSO) algorithm to minimize benchmark functions, such as the Rastrigin and Sphere functions, by optimizing their input parameters. The goal is to find the global minimum while efficiently exploring the solution space using swarm intelligence.

Algorithm :



Pseudo code

function PSO(fitness_function, num_particles, num_iterations, inertia_weight, cognitive_coefficient, social_coefficient):

create an array of particles with size num_particles

For i from 0 to num_particles - 1:

particles[i].position = Randomly initialize position in range [-5,5]

particles[i].velocity = Randomly initialize velocity in range [-1,1]

particles.best_position = particles[i].position

particles.best_fitness = fitness_function(particles[i].position)

global_best_position = particles[0].best_position

global_best_fitness = particles[0].best_fitness

for iteration from 1 to num_iterations:

for each particle in particles:

current_fitness = fitness_function(particle.position)

if current_fitness < particle.best_fitness:

particle.best_position = particle.position

particle.best_fitness = current_fitness

if current_fitness < global_best_fitness

global_best_position = particle.position

global_best_fitness = current_fitness

r1 = random value in range [0,1]

r2 = random value in range [0,1]

particle.velocity = (inertia_weight * particle.velocity + cognitive_coeff * r1 * (particle.best_position - particle.position) + social co-efficient * r2 * (global_best_position - particle.position))

particle.position = particle.position + particle.velocity.

Code:

```
import random
import math  # cos() for Rastrigin
import copy  # array-copying convenience
import sys   # max float

#-------fitness functions---------

# Rastrigin function
def fitness_rastrigin(position):
    fitnessVal = 0.0
    for i in range(len(position)):
        xi = position[i]
        fitnessVal += (xi * xi) - (10 * math.cos(2 * math.pi * xi)) + 10
    return fitnessVal

# Sphere function
def fitness_sphere(position):
    fitnessVal = 0.0
    for i in range(len(position)):
        xi = position[i]
        fitnessVal += (xi * xi)
    return fitnessVal
#-------------------------

# Particle class
class Particle:
    def __init__(self, fitness, dim, minx, maxx, seed):
        self.rnd = random.Random(seed)

        # Initialize position of the particle with 0.0 value
        self.position = [0.0 for i in range(dim)]

        # Initialize velocity of the particle with 0.0 value
        self.velocity = [0.0 for i in range(dim)]

        # Initialize best particle position of the particle with 0.0 value
        self.best_part_pos = [0.0 for i in range(dim)]

        # Loop dim times to calculate random position and velocity
        # Range of position and velocity is [minx, max]
        for i in range(dim):
            self.position[i] = ((maxx - minx) *
                                self.rnd.random() + minx)
            self.velocity[i] = ((maxx - minx) *
                                self.rnd.random() + minx)
```

```
        # Compute fitness of particle
        self.fitness = fitness(self.position)   # Current fitness

        # Initialize best position and fitness of this particle
        self.best_part_pos = copy.copy(self.position)
        self.best_part_fitnessVal = self.fitness   # Best fitness

# Particle swarm optimization function
def pso(fitness, max_iter, n, dim, minx, maxx):
    # Hyper parameters
    w = 0.729  # Inertia
    c1 = 1.49445  # Cognitive (particle)
    c2 = 1.49445  # Social (swarm)

    rnd = random.Random(0)

    # Create n random particles
    swarm = [Particle(fitness, dim, minx, maxx, i) for i in range(n)]

    # Compute the value of best_position and best_fitness in swarm
    best_swarm_pos = [0.0 for i in range(dim)]
    best_swarm_fitnessVal = sys.float_info.max  # Swarm best

    # Compute best particle of swarm and its fitness
    for i in range(n):  # Check each particle
        if swarm[i].fitness < best_swarm_fitnessVal:
            best_swarm_fitnessVal = swarm[i].fitness
            best_swarm_pos = copy.copy(swarm[i].position)

      Iter = 0
    while Iter < max_iter:
        if Iter % 10 == 0 and Iter > 1:
            print("Iter = " + str(Iter) + " best fitness = %.3f" %
best_swarm_fitnessVal)

        for i in range(n):  # Process each particle


            for k in range(dim):
                r1 = rnd.random()  # Randomizations
                r2 = rnd.random()

                swarm[i].velocity[k] = (
                    (w * swarm[i].velocity[k]) +
                    (c1 * r1 * (swarm[i].best_part_pos[k] - swarm[i].position[k]))
+
                    (c2 * r2 * (best_swarm_pos[k] - swarm[i].position[k]))
```

```
                )

                if swarm[i].velocity[k] < minx:
                    swarm[i].velocity[k] = minx
                elif swarm[i].velocity[k] > maxx:
                    swarm[i].velocity[k] = maxx

                        for k in range(dim):
                swarm[i].position[k] += swarm[i].velocity[k]


            swarm[i].fitness = fitness(swarm[i].position)

                        if swarm[i].fitness < swarm[i].best_part_fitnessVal:
                swarm[i].best_part_fitnessVal = swarm[i].fitness
                swarm[i].best_part_pos = copy.copy(swarm[i].position)


            if swarm[i].fitness < best_swarm_fitnessVal:
                best_swarm_fitnessVal = swarm[i].fitness
                best_swarm_pos = copy.copy(swarm[i].position)

        Iter += 1

    return best_swarm_pos
# End PSO

#----------------------------
# Driver code for Rastrigin function
print("\nBegin particle swarm optimization on Rastrigin function\n")
dim = 3
fitness = fitness_rastrigin

print("Goal is to minimize Rastrigin's function in " + str(dim) + " variables")
print("Function has known min = 0.0 at (", end="")
for i in range(dim-1):
    print("0, ", end="")
print("0)")

num_particles = 50
max_iter = 100

print("Setting num_particles = " + str(num_particles))
print("Setting max_iter = " + str(max_iter))
print("\nStarting PSO algorithm\n")

best_position = pso(fitness, max_iter, num_particles, dim, -10.0, 10.0)
```

```
print("\nPSO completed\n")
print("\nBest solution found:")
print(["%.6f" % best_position[k] for k in range(dim)])
fitnessVal = fitness(best_position)
print("fitness of best solution = %.6f" % fitnessVal)

print("\nEnd particle swarm for Rastrigin function\n")


print()
print()


# Driver code for Sphere function
print("\nBegin particle swarm optimization on Sphere function\n")
dim = 3
fitness = fitness_sphere

print("Goal is to minimize sphere function in " + str(dim) + " variables")
print("Function has known min = 0.0 at (", end="")
for i in range(dim-1):
    print("0, ", end="")
print("0)")

num_particles = 50
max_iter = 100

print("Setting num_particles = " + str(num_particles))
print("Setting max_iter = " + str(max_iter))
print("\nStarting PSO algorithm\n")

best_position = pso(fitness, max_iter, num_particles, dim, -10.0, 10.0)
print("\nPSO completed\n")
print("\nBest solution found:")
print(["%.6f" % best_position[k] for k in range(dim)])
fitnessVal = fitness(best_position)
print("fitness of best solution = %.6f" % fitnessVal)
print("\nEnd particle swarm for Sphere function\n")
```

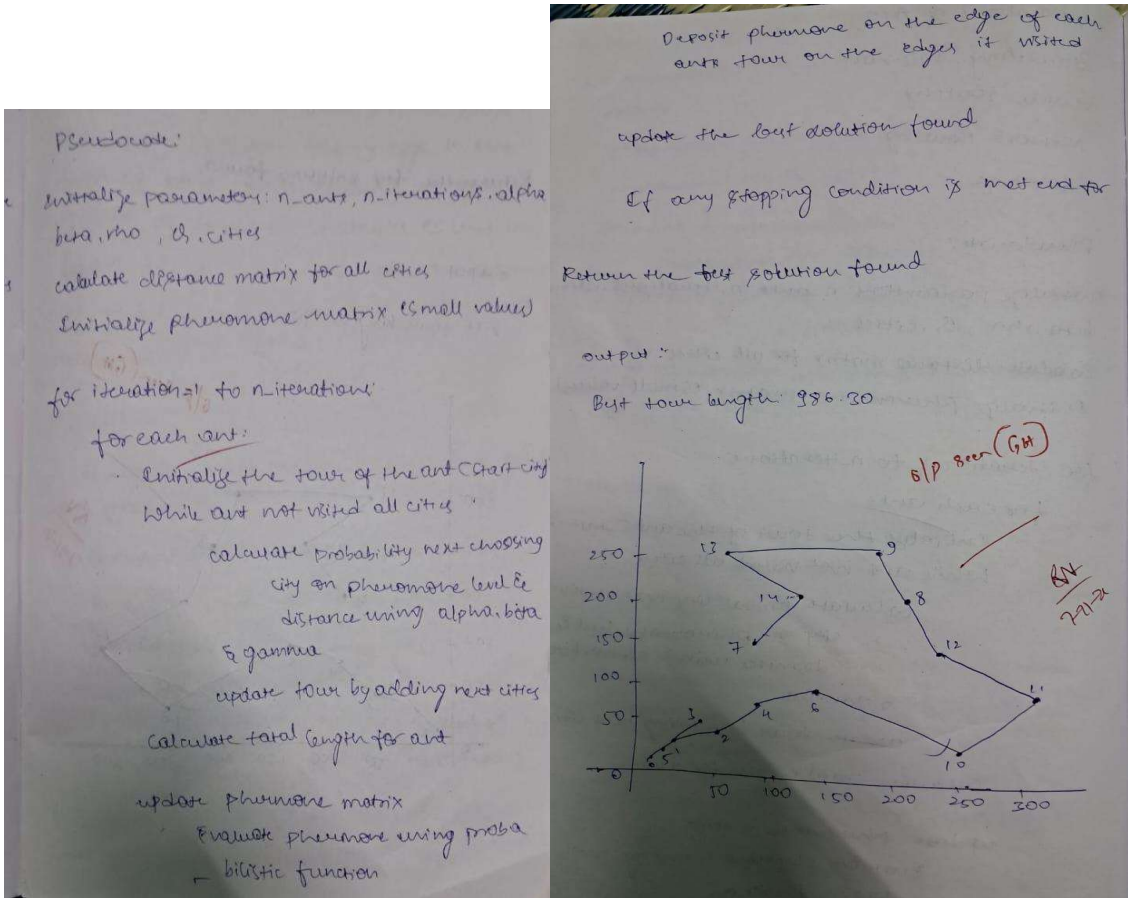
Output:

```
Begin particle swarm optimization on Rastrigin function

Goal is to minimize Rastrigin's function in 3 variables
Function has known min = 0.0 at (0, 0, 0)
Setting num_particles = 50
Setting max_iter = 100

Starting PSO algorithm

Iter = 10 best fitness = 8.463
Iter = 20 best fitness = 4.792
Iter = 30 best fitness = 2.223
Iter = 40 best fitness = 0.251
Iter = 50 best fitness = 0.251
Iter = 60 best fitness = 0.061
Iter = 70 best fitness = 0.007
Iter = 80 best fitness = 0.005
Iter = 90 best fitness = 0.000

PSO completed


Best solution found:
['0.000618', '0.000013', '0.000616']
fitness of best solution = 0.000151

End particle swarm for Rastrigin function


Begin particle swarm optimization on Sphere function

Goal is to minimize sphere function in 3 variables
Function has known min = 0.0 at (0, 0, 0)
Setting num_particles = 50
Setting max_iter = 100

Starting PSO algorithm

Iter = 10 best fitness = 0.189
Iter = 20 best fitness = 0.012
Iter = 30 best fitness = 0.001
Iter = 40 best fitness = 0.000
Iter = 50 best fitness = 0.000
Iter = 60 best fitness = 0.000
Iter = 70 best fitness = 0.000
Iter = 80 best fitness = 0.000
Iter = 90 best fitness = 0.000

PSO completed


Best solution found:
['0.000004', '-0.000001', '0.000007']
fitness of best solution = 0.000000

End particle swarm for Sphere function
```

Program 3

Problem Statement : Implement an Ant Colony Optimization (ACO) algorithm to solve the Traveling Salesman Problem (TSP), where the goal is to find the shortest possible path that visits all cities exactly once and returns to the starting city. The algorithm should utilize pheromone trails and heuristic information to guide the search efficiently.

Algorithm :



Code:

```
import numpy as np

# Initialize distance matrix with integers
dist_matrix = np.array([[0, 2, 2, 1, 1],
                        [2, 0, 2, 1, 3],
                        [2, 2, 0, 1, 1],
                        [1, 1, 1, 0, 2],
                        [1, 3, 1, 2, 0]])

# Convert dist_matrix to float64 to prevent type mismatch when adding np.eye()
dist_matrix = dist_matrix.astype(np.float64)
# Add a small value to the diagonal to ensure no zero distances
dist_matrix += np.eye(dist_matrix.shape[0])  # Set diagonal to a non-zero value
# Initialize pheromone matrix with lower initial values
pheromone = np.ones(dist_matrix.shape) * 0.1  # Starting with a smaller pheromone
value
# Parameters for the ACO (Ant Colony Optimization)
num_ants = 5  # Number of ants
iterations = 100  # Number of iterations
alpha = 1.0  # Influence of pheromone
beta = 2.0  # Influence of distance (heuristic information)
evaporation_rate = 0.3  # Moderate evaporation rate to keep pheromones for a while
Q = 10  # Moderate constant for pheromone update
# Function to compute the probability of moving to a particular city
def calculate_probability(ant_position, pheromone, dist_matrix, alpha, beta):
    pheromone_values = pheromone[ant_position]
    distance_values = dist_matrix[ant_position]
    pheromone_factor = pheromone_values ** alpha
    distance_factor = (1.0 / distance_values) ** beta
    probability = pheromone_factor * distance_factor
    probability /= probability.sum()  # Normalize to make sure the probabilities
sum to 1
    return probability
# Function to update pheromones after each iteration
def update_pheromones(pheromone, ants_paths, dist_matrix, evaporation_rate, Q):
    pheromone *= (1 - evaporation_rate)  # Apply pheromone evaporation
    for path in ants_paths:
        # Update pheromone on all edges in the path (including returning to start)
        for i in range(len(path) - 1):
            pheromone[path[i], path[i + 1]] += Q / dist_matrix[path[i], path[i +
1]]
        pheromone[path[-1], path[0]] += Q / dist_matrix[path[-1], path[0]]  # Add
pheromone for returning to the start
# ACO algorithm
```

```
def ant_colony_optimization(dist_matrix, pheromone, num_ants, iterations, alpha,
beta, evaporation_rate, Q):


    best_path = None
    best_path_length = float('inf')
    for iteration in range(iterations):
        ants_paths = []
        for ant in range(num_ants):
            path = [0]  # Start from the first city (0-indexed)
            visited = set(path)
            while len(path) < len(dist_matrix):
                current_position = path[-1]
                probability = calculate_probability(current_position, pheromone,
dist_matrix, alpha, beta)

                # Choose next city based on probabilities
                next_city = np.random.choice(len(dist_matrix), p=probability)
                if next_city not in visited:
                    path.append(next_city)
                    visited.add(next_city)

            path.append(path[0])  # Ensure the ant returns to the starting city
            ants_paths.append(path)
        # Update pheromones after ants' paths are determined
        update_pheromones(pheromone, ants_paths, dist_matrix, evaporation_rate, Q)
        # Find the best path in the current iteration
        for path in ants_paths:
            path_length = sum(dist_matrix[path[i], path[i + 1]] for i in
range(len(path) - 1))
            if path_length < best_path_length:
                best_path_length = path_length
                best_path = path

    return best_path, best_path_length

# Run the ACO algorithm
best_path, best_path_length = ant_colony_optimization(dist_matrix, pheromone,
num_ants, iterations, alpha, beta, evaporation_rate, Q)

# Output the final results
print(f"\nBest path found: {best_path}")
print(f"Length of best path: {best_path_length}")
```

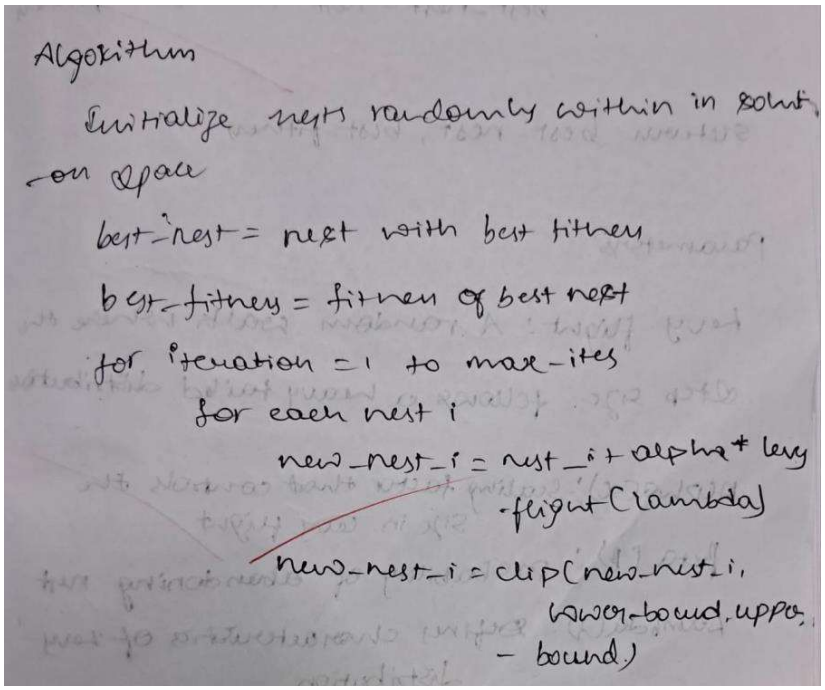
Output :

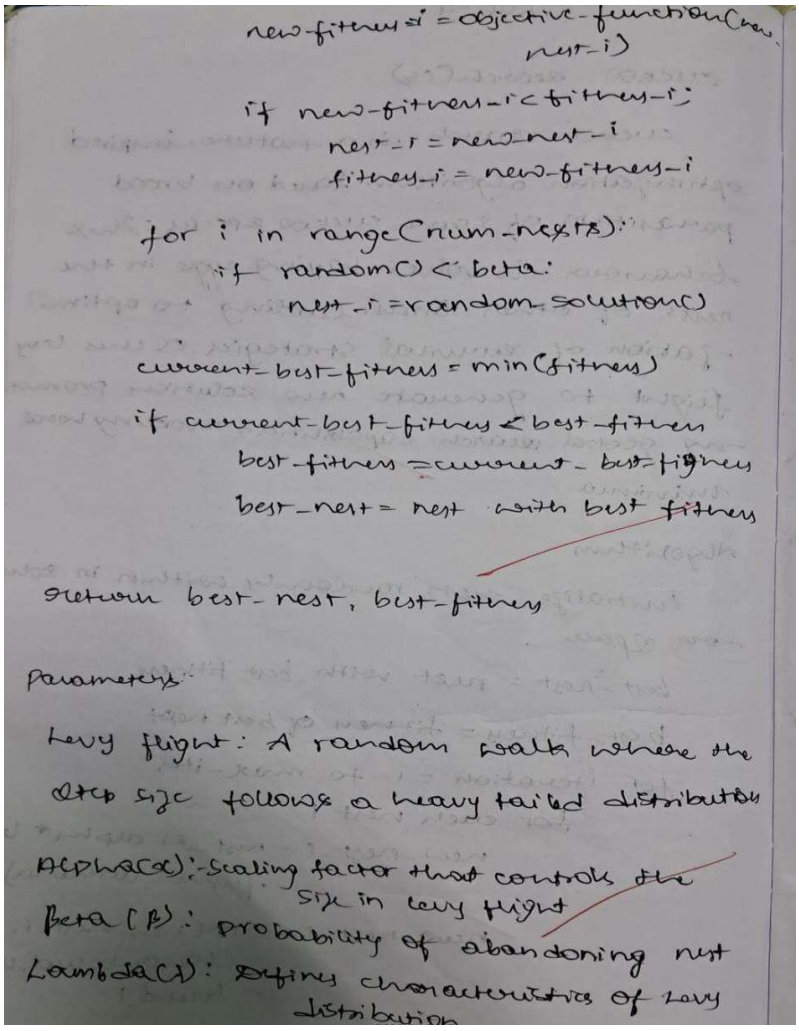
```
Best path found: [0, 3, 1, 2, 4, 0]
Length of best path: 6.0
```

Program 4

Problem Statement : Implement the Cuckoo Search Algorithm for feature selection to identify an optimal subset of features that maximizes the classification accuracy of a Support Vector Machine (SVM) on a given dataset.

Algorithm :





Code:

```
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score

# Define the fitness function (classification accuracy)
def fitness_function(features, X_train, y_train, X_test, y_test):
    # Subset the features: Convert the binary array into the actual feature subset
    X_train_subset = X_train[:, features.astype(bool)] if np.any(features) else
X_train[:, :1]  # Default to 1 feature if no feature is selected
    X_test_subset = X_test[:, features.astype(bool)] if np.any(features) else
X_test[:, :1]

    # Train a classifier (e.g., SVM)
    clf = SVC()
    clf.fit(X_train_subset, y_train)

    # Make predictions and calculate accuracy
    y_pred = clf.predict(X_test_subset)
    accuracy = accuracy_score(y_test, y_pred)
    return accuracy

# Cuckoo Search Algorithm
def cuckoo_search(X, y, num_nests=10, max_iter=100, pa=0.25):
    n_features = X.shape[1]

    # Initialize nests (random binary feature subsets)
    nests = np.random.randint(2, size=(num_nests, n_features))

    # Split data into train and test sets
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
random_state=42)

    # Evaluate fitness of each nest
    fitness = np.array([fitness_function(nest, X_train, y_train, X_test, y_test)
for nest in nests])

    # Best solution (initially the best nest)
    best_nest = nests[np.argmax(fitness)]
    best_fitness = np.max(fitness)

    # Main Cuckoo Search Loop
    for iteration in range(max_iter):
        # Generate new solutions using Levy flight
  new_nests = nests + np.random.normal(0, 1, nests.shape) * (best_nest - nests)
```

```
        new_nests = np.clip(new_nests, 0, 1)  # Ensure binary values (0 or 1)

        # Evaluate the fitness of the new nests
        new_fitness = np.array([fitness_function(nest, X_train, y_train, X_test,
y_test) for nest in new_nests])

        # Select nests to replace (based on fitness)
        replace_mask = new_fitness > fitness
        nests[replace_mask] = new_nests[replace_mask]
        fitness[replace_mask] = new_fitness[replace_mask]

        # Abandon worst nests based on probability
        abandon_mask = np.random.rand(num_nests) < pa
        nests[abandon_mask] = np.random.randint(2, size=(np.sum(abandon_mask),
n_features))

        # Update the best solution
        best_nest = nests[np.argmax(fitness)]
        best_fitness = np.max(fitness)

        # Print progress (optional)


    return best_nest, best_fitness

# Example usage
if __name__ == "__main__":
    # Load your dataset (e.g., Iris dataset)
    from sklearn.datasets import load_iris
    X, y = load_iris(return_X_y=True)

    # Run Cuckoo Search for feature selection
    best_features, best_accuracy = cuckoo_search(X, y, num_nests=10, max_iter=100)

    print("Best feature subset:", best_features)
    print("Best classification accuracy:", best_accuracy)
```

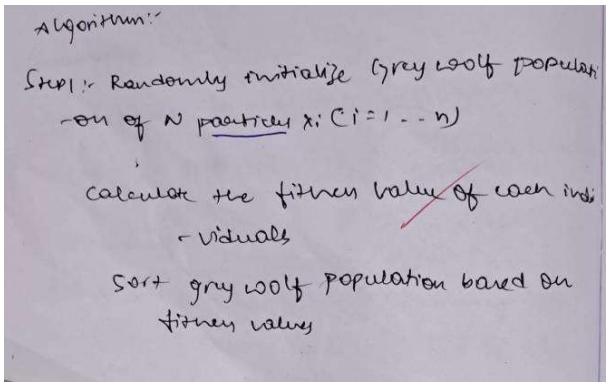
Output:

```
Best feature subset: [1 0 1 0]
Best classification accuracy: 1.0
```

Program 5

Problem Statement : Implement the Grey Wolf Optimizer (GWO) to optimize the hyperparameters (C and gamma) of a Support Vector Machine (SVM) classifier for achieving the best classification accuracy on the Iris dataset.

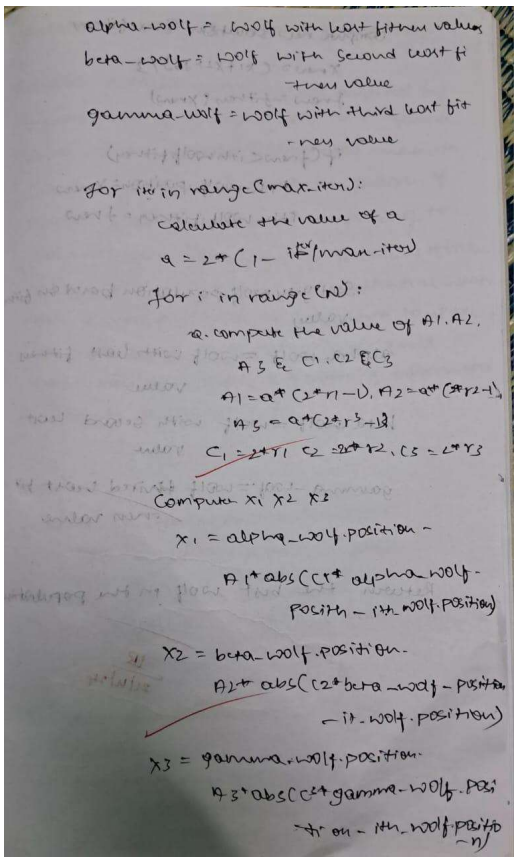
Algorithm :


Algorithm:-

Step 1:- Randomly initialize Grey wolf Popula-
-on of N particles x_i (i = 1 - - n)

Calculate the fitness value of each indi-
-viduals

Sort grey wolf population based on
fitness values


alpha_wolf = wolf with least fitness value
beta_wolf = wolf with second least fi-
-tness value
gamma_wolf = wolf with third least fit-
-ness value

for it in range(max_iter):
calculate the value of a
a = 2*(1 - it/max_iter)
for i in range(N):
compute the value of A1, A2,
A3 & C1, C2 & C3
A1 = a*(2*r1 - 1), A2 = a*(2*r2 - 1),
A3 = a*(2*r3 - 1)
C1 = 2*r1, C2 = 2*r2, C3 = 2*r3
Compute x1 x2 x3
x1 = alpha_wolf.position -
A1*abs(C1*alpha_wolf.
posith - ith_wolf.position)
x2 = beta_wolf.position -
A2*abs(C2*beta_wolf - positi-
-on - ith_wolf.position)
x3 = gamma_wolf.position -
A3*abs(C3*gamma_wolf.Posi
-tion - ith_wolf.positio
-n)

compute new solution & its fitness

$x_{new} = (x_1 + x_2 + x_3)/3$

$f_{new} = fitness(x_{new})$

if (f_{new} < ith wolf fitness)

ith-wolf.position = x_{new}

ith-wolf.fitness = f_{new}

- Sort grey wolf population based on fitness value

alpha_wolf = wolf with least fitness value

beta_wolf = wolf with second least value

gamma_wolf = wolf ~~third least~~ fitness value

Return the best wolf in the population

IB
21/11/24

Code :

```
import numpy as np
from sklearn.svm import SVC
from sklearn.model_selection import train_test_split, cross_val_score
from sklearn.datasets import load_iris
#APPLICATION

# Objective function: Minimize 1 - accuracy of SVM
def svm_fitness(params, X, y):
    C, gamma = params
    C = max(0.01, C)  # Ensure C is positive
    gamma = max(0.01, gamma)  # Ensure gamma is positive
    model = SVC(C=C, gamma=gamma)
    scores = cross_val_score(model, X, y, cv=5)  # 5-fold cross-validation
    return 1 - scores.mean()  # Minimize (1 - accuracy)

# GWO Algorithm
class GreyWolfOptimizer:
    def __init__(self, func, dim, bounds, n_wolves=10, max_iter=50):
        self.func = func
        self.dim = dim
        self.bounds = bounds
        self.n_wolves = n_wolves
        self.max_iter = max_iter
        self.alpha_pos = np.zeros(dim)
        self.beta_pos = np.zeros(dim)
        self.delta_pos = np.zeros(dim)
        self.alpha_score = float("inf")
        self.beta_score = float("inf")
        self.delta_score = float("inf")
        self.positions = np.random.uniform(bounds[:, 0], bounds[:, 1], (n_wolves,
dim))

    def optimize(self, *args):
        for iter in range(self.max_iter):
            for i in range(self.n_wolves):
                # Fitness evaluation
                fitness = self.func(self.positions[i], *args)

                # Update alpha, beta, delta
                if fitness < self.alpha_score:
                    self.alpha_score, self.alpha_pos = fitness,
self.positions[i].copy()
                elif fitness < self.beta_score:
                    self.beta_score, self.beta_pos = fitness,
self.positions[i].copy()
                elif fitness < self.delta_score:
```

```
                    self.delta_score, self.positions[i] = fitness,
self.positions[i].copy()

            # Update positions
            a = 2 - iter * (2 / self.max_iter)
            for i in range(self.n_wolves):
                for d in range(self.dim):
                    r1, r2 = np.random.rand(), np.random.rand()
                    A1, C1 = 2 * a * r1 - a, 2 * r2
                    D_alpha = abs(C1 * self.alpha_pos[d] - self.positions[i][d])
                    X1 = self.alpha_pos[d] - A1 * D_alpha

                    r1, r2 = np.random.rand(), np.random.rand()
                    A2, C2 = 2 * a * r1 - a, 2 * r2
                    D_beta = abs(C2 * self.beta_pos[d] - self.positions[i][d])
                    X2 = self.beta_pos[d] - A2 * D_beta

                    r1, r2 = np.random.rand(), np.random.rand()
                    A3, C3 = 2 * a * r1 - a, 2 * r2
                    D_delta = abs(C3 * self.delta_pos[d] - self.positions[i][d])
                    X3 = self.delta_pos[d] - A3 * D_delta

                    self.positions[i][d] = (X1 + X2 + X3) / 3

                # Ensure positions are within bounds
                self.positions[i] = np.clip(self.positions[i], self.bounds[:, 0],
self.bounds[:, 1])

            print(f"Iteration {iter + 1}: Best Fitness = {self.alpha_score}")

        return self.alpha_score, self.alpha_pos

# Load dataset
data = load_iris()
X, y = data.data, data.target
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# Define bounds for C and gamma
bounds = np.array([[0.01, 100], [0.01, 1]])  # C and gamma bounds

# Run GWO
optimizer = GreyWolfOptimizer(func=svm_fitness, dim=2, bounds=bounds, max_iter=30)
best_score, best_params = optimizer.optimize(X_train, y_train)

# Train and test the final model
best_C, best_gamma = best_params
```

```
final_model = SVC(C=best_C, gamma=best_gamma)
final_model.fit(X_train, y_train)
accuracy = final_model.score(X_test, y_test)

print(f"Best Parameters: C = {best_C}, Gamma = {best_gamma}")
print(f"Test Accuracy: {accuracy}")
```

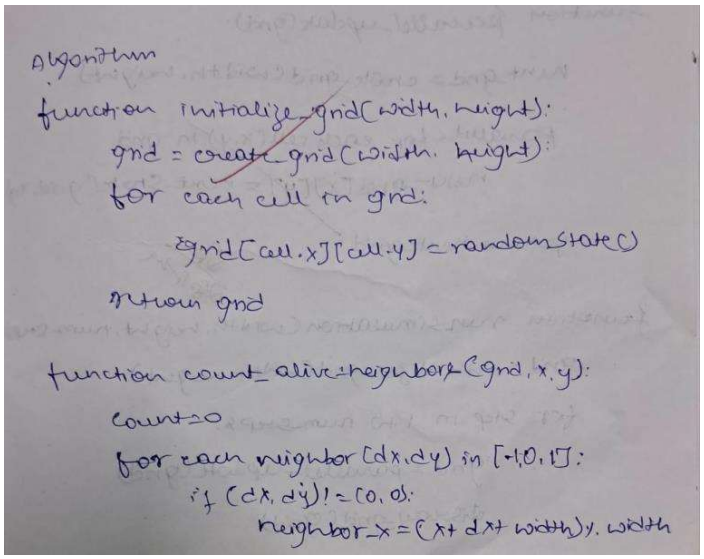
Output :

```
Iteration 1: Best Fitness = 0.050000000000000044
Iteration 2: Best Fitness = 0.04166666666666663
Iteration 3: Best Fitness = 0.04166666666666663
Iteration 4: Best Fitness = 0.04166666666666663
Iteration 5: Best Fitness = 0.04166666666666663
Iteration 6: Best Fitness = 0.04166666666666663
Iteration 7: Best Fitness = 0.04166666666666663
Iteration 8: Best Fitness = 0.04166666666666663
Iteration 9: Best Fitness = 0.03333333333333344
Iteration 10: Best Fitness = 0.03333333333333344
Iteration 11: Best Fitness = 0.03333333333333344
Iteration 12: Best Fitness = 0.03333333333333344
Iteration 13: Best Fitness = 0.03333333333333344
Iteration 14: Best Fitness = 0.03333333333333344
Iteration 15: Best Fitness = 0.03333333333333344
Iteration 16: Best Fitness = 0.03333333333333344
Iteration 17: Best Fitness = 0.03333333333333344
Iteration 18: Best Fitness = 0.03333333333333344
Iteration 19: Best Fitness = 0.03333333333333344
Iteration 20: Best Fitness = 0.03333333333333344
Iteration 21: Best Fitness = 0.03333333333333344
Iteration 22: Best Fitness = 0.03333333333333344
Iteration 23: Best Fitness = 0.03333333333333344
Iteration 24: Best Fitness = 0.03333333333333344
Iteration 25: Best Fitness = 0.03333333333333344
Iteration 26: Best Fitness = 0.03333333333333344
Iteration 27: Best Fitness = 0.03333333333333344
Iteration 28: Best Fitness = 0.03333333333333344
Iteration 29: Best Fitness = 0.03333333333333344
Iteration 30: Best Fitness = 0.03333333333333344
Best Parameters: C = 39.84983388283063, Gamma = 0.01
Test Accuracy: 1.0
```

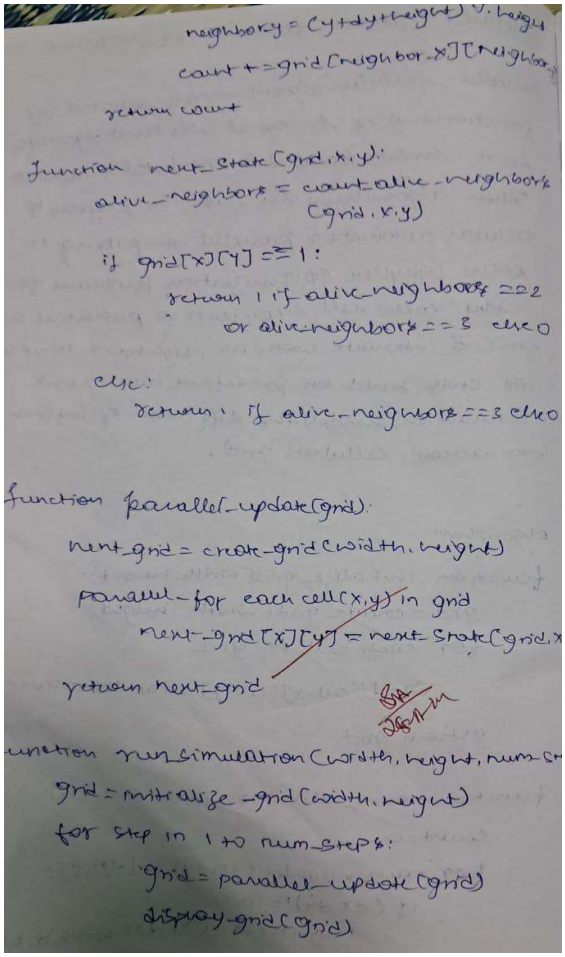
Program 6

Problem Statement : Develop a parallel cellular automaton-based algorithm for optimal robot route planning in a grid-based environment, ensuring collision-free navigation while minimizing travel distance and computational time.

Algorithm :



```
Algorithm
function initialize_grid(width, height):
    grid = create_grid(width, height)
    for each cell in grid:
        grid[cell.x][cell.y] = randomState()
    return grid

function count_alive_neighbors(grid, x, y):
    count=0
    for each neighbor (dx, dy) in [-1,0,1]:
        if (dx, dy) != (0, 0):
            neighbor_x = (x + dx + width) % width
```



```
        neighbor_y = (y+dy+height) % height
        count += grid[neighbor_x][neighbor_y]
    return count

function next_state(grid, x, y):
    alive_neighbors = count_alive_neighbors
                      (grid, x, y)
    if grid[x][y] == 1:
        return 1 if alive_neighbors == 2
               or alive_neighbors == 3 else 0
    else:
        return 1 if alive_neighbors == 3 else 0

function parallel_update(grid):
    next_grid = create_grid(width, height)
    parallel_for each cell(x,y) in grid
        next_grid[x][y] = next_state(grid,x
    return next_grid

function run_simulation(width, height, num_s
    grid = initialize_grid(width, height)
    for step in 1 to num_steps:
        grid = parallel_update(grid)
        display_grid(grid)
```

Code :

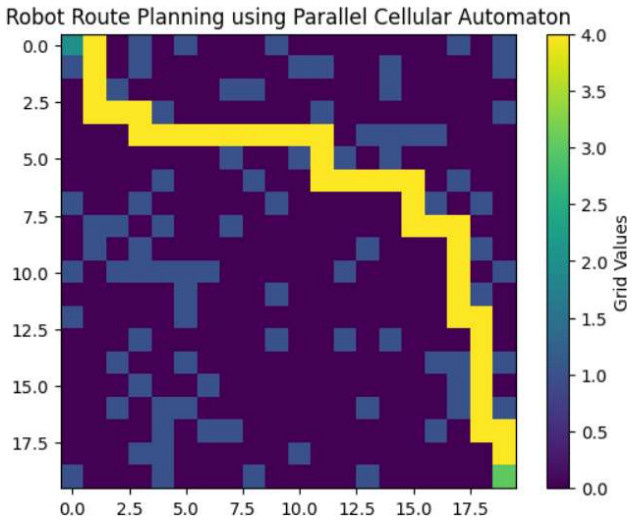
```
import numpy as np
import multiprocessing as mp
import matplotlib.pyplot as plt

# Define grid environment
FREE = 0      # Free space
OBSTACLE = 1 # Obstacle
START = 2     # Start point
TARGET = 3    # Target point
PATH = 4      # Path found

# Generate grid environment
def create_grid(rows, cols, obstacle_ratio=0.2):
    grid = np.random.choice([FREE, OBSTACLE], size=(rows, cols),
p=[1-obstacle_ratio, obstacle_ratio])
    grid[0, 0] = START          # Start point
    grid[-1, -1] = TARGET       # Target point
    return grid

# Define the cellular automaton rules
def update_cell(grid, cost_grid, row, col):
    if grid[row, col] == OBSTACLE or grid[row, col] == START:
        return cost_grid[row, col]   # No update for obstacles and start

    neighbors = [
        cost_grid[row-1, col] if row > 0 else np.inf,            # Up
        cost_grid[row+1, col] if row < grid.shape[0]-1 else np.inf,   # Down
        cost_grid[row, col-1] if col > 0 else np.inf,            # Left
        cost_grid[row, col+1] if col < grid.shape[1]-1 else np.inf  # Right
    ]
    return min(neighbors) + 1 if grid[row, col] != OBSTACLE else np.inf
# Parallel function for cost update
def parallel_update(grid, cost_grid):
    rows, cols = grid.shape
    new_cost_grid = cost_grid.copy()
    pool = mp.Pool(mp.cpu_count())

    tasks = [(grid, cost_grid, r, c) for r in range(rows) for c in range(cols)]
    results = pool.starmap(update_cell, tasks)

    # Update cost grid
    for idx, (r, c) in enumerate([(r, c) for r in range(rows) for c in
range(cols)]):
        new_cost_grid[r, c] = results[idx]

    pool.close()
```

```
    pool.join()
    return new_cost_grid
# Backtrack to find the path
def backtrack_path(grid, cost_grid, start, target):
    path = []
    current = target
    while current != start:
        path.append(current)
        row, col = current
        neighbors = [
            ((row-1, col), cost_grid[row-1, col]) if row > 0 else (None, np.inf),
            ((row+1, col), cost_grid[row+1, col]) if row < grid.shape[0]-1 else
(None, np.inf),
            ((row, col-1), cost_grid[row, col-1]) if col > 0 else (None, np.inf),
            ((row, col+1), cost_grid[row, col+1]) if col < grid.shape[1]-1 else
(None, np.inf)
        ]
        next_cell = min(neighbors, key=lambda x: x[1])
        if next_cell[0] is None or next_cell[1] == np.inf:
            raise ValueError("No path found!")
        current = next_cell[0]

    path.append(start)
    return path[::-1]
# Main function to execute the route planning
def main():
    rows, cols = 20, 20  # Grid size
    grid = create_grid(rows, cols)

    # Initialize cost grid
    cost_grid = np.full_like(grid, np.inf, dtype=float)
    cost_grid[0, 0] = 0  # Cost at start is 0

    start = (0, 0)
    target = (rows-1, cols-1)

    # Display initial grid
    print("Initial Grid:")
    print(grid)
    # Run cellular automaton for route planning
    iterations = rows + cols
    for i in range(iterations):
        new_cost_grid = parallel_update(grid, cost_grid)
        if np.array_equal(new_cost_grid, cost_grid):
            break  # Stop if no update
        cost_grid = new_cost_grid
```

```
    # Backtrack to find the path
    try:
        path = backtrack_path(grid, cost_grid, start, target)
        print("\nPath Found:")
        for r, c in path:
            if grid[r, c] == FREE:
                grid[r, c] = PATH
        print(grid)
    except ValueError as e:
        print("\nError:", e)

    # Visualize the grid
    plt.imshow(grid, cmap='viridis', origin='upper')
    plt.title("Robot Route Planning using Parallel Cellular Automaton")
    plt.colorbar(label="Grid Values")
    plt.show()
if __name__ == "__main__":
    main()
```

Output :

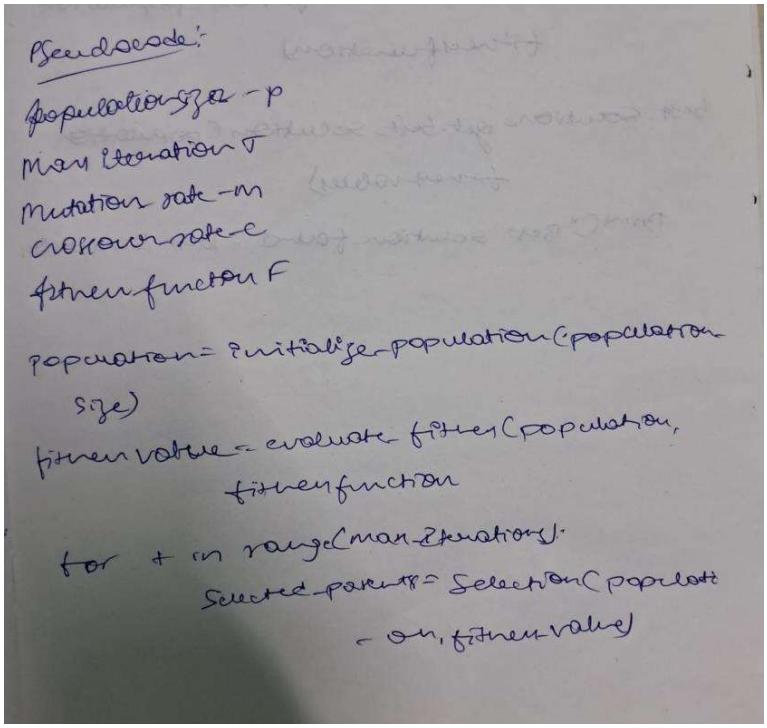
```
Initial Grid:
[[2 0 0 1 0 1 0 0 0 1 0 0 0 0 0 0 0 1 0 1]
 [1 0 0 1 0 0 0 0 0 0 1 1 0 0 1 0 0 0 0 1]
 [0 0 1 0 0 0 0 1 1 0 0 0 0 0 1 0 0 0 0 0]
 [0 0 0 0 1 0 0 0 0 0 0 1 0 0 0 0 0 0 0 1]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 0 0 0]
 [0 0 0 0 0 0 0 1 0 0 1 0 1 0 1 0 0 0 0 0]
 [0 0 0 0 1 0 0 0 1 0 0 0 0 0 0 0 1 0 0 0]
 [1 0 0 1 0 0 0 0 0 1 0 0 0 0 0 0 1 0 1 0]
 [0 1 1 0 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 1 0 1 0 0 0 0 0 0 0 0 0 1 0 0 0 0 1 0]
 [1 0 1 1 1 1 1 0 0 0 0 0 1 0 0 0 0 0 0 1]
 [0 0 0 0 0 1 0 0 0 1 0 0 0 0 0 0 0 0 1 0]
 [1 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 1 0 0 0 0 0 1 0 0 1 0 1 0 0 0 0 0]
 [0 0 1 0 0 1 0 0 0 0 0 0 0 0 0 1 1 0 1 0]
 [0 0 0 1 0 0 1 0 0 0 0 0 0 0 0 0 0 1 0 0]
 [0 0 1 0 1 1 0 0 0 0 0 0 0 1 0 0 0 1 0 1]
 [0 0 0 0 1 0 1 1 0 0 0 0 0 0 0 1 0 0 0 0]
 [0 0 0 1 1 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0]
 [1 0 0 0 1 0 0 0 1 0 0 0 0 1 0 0 0 0 0 3]]

Path Found:
[[2 4 0 1 0 1 0 0 0 1 0 0 0 0 0 0 0 1 0 1]
 [1 4 0 1 0 0 0 0 0 0 1 1 0 0 1 0 0 0 0 1]
 [0 4 1 0 0 0 0 1 1 0 0 0 0 0 1 0 0 0 0 0]
 [0 4 4 4 1 0 0 0 0 0 0 1 0 0 0 0 0 0 0 1]
 [0 0 0 4 4 4 4 4 4 4 4 4 0 1 1 1 1 0 0 0]
 [0 0 0 0 0 0 0 1 0 0 1 4 1 0 1 0 0 0 0 0]
 [0 0 0 0 1 0 0 0 1 0 0 4 4 4 4 4 0 1 0 0]
 [1 0 0 1 0 0 0 0 0 1 0 0 0 0 0 4 1 0 1 0]
 [0 1 1 0 1 0 0 1 0 0 0 0 0 0 0 4 4 4 0 0]
 [0 1 0 1 0 0 0 0 0 0 0 0 0 1 0 0 0 4 1 0]
 [1 0 1 1 1 1 1 0 0 0 0 0 1 0 0 0 0 4 0 1]
 [0 0 0 0 0 1 0 0 0 1 0 0 0 0 0 0 0 4 1 0]
 [1 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 4 4 0]
 [0 0 0 1 0 0 0 0 0 1 0 0 1 0 1 0 0 0 4 0]
 [0 0 1 0 0 1 0 0 0 0 0 0 0 0 0 0 1 1 4 1]
 [0 0 0 1 0 0 1 0 0 0 0 0 0 0 0 0 0 1 4 0]
 [0 0 1 0 1 1 0 0 0 0 0 0 0 1 0 0 0 1 4 1]
 [0 0 0 0 1 0 1 1 0 0 0 0 0 0 0 1 0 4 4 4]
 [0 0 0 1 1 0 0 0 0 0 1 0 0 0 0 0 0 0 0 4]
 [1 0 0 0 1 0 0 0 1 0 0 0 0 1 0 0 0 0 0 3]]
```



Program 7

Problem Statement : Solve the 0/1 Knapsack Problem using the Gene Expression Algorithm (GEA) to maximize the total value of selected items without exceeding the given weight capacity.

Algorithm :



Pseudocode:

```
population_size = p
max_iteration = T
mutation_rate = m
crossover_rate = c
fitness_function F

population = initialize_population(population_size)
fitness_value = evaluate_fitness(population, fitness_function)

for t in range(max_iterations):
    selected_parents = selection(population, fitness_value)
```

```
offspring = crossover(selected_parents,
                      crossover_rate)
offspring = mutation(offspring,
                     mutation_rate)
offspring_fitness = evaluate_fitness
                    (offspring, fitness
                     function)
population = combine_&_select(population,
             offspring, fitness_values, offspring_fitness)
fitness_values = evaluate_fitness(population,
                 fitness_function)
best_solution = get_best_solution(population,
                fitness_values)
print("Best solution found:", best_solution)
```

Code :

```
import random


# Define the Knapsack Problem (Objective Function)
def knapsack_fitness(items, capacity, solution):
    total_weight = sum([items[i][0] for i in range(len(solution)) if solution[i] ==
1])
    total_value = sum([items[i][1] for i in range(len(solution)) if solution[i] ==
1])

    # If total weight exceeds the capacity, return 0 (invalid solution)
    if total_weight > capacity:
        return 0
    return total_value


# Gene Expression Algorithm (GEA)
class GeneExpressionAlgorithm:
    def __init__(self, population_size, num_items, mutation_rate, crossover_rate,
generations, capacity, items):
        self.population_size = population_size
        self.num_items = num_items
        self.mutation_rate = mutation_rate
        self.crossover_rate = crossover_rate
        self.generations = generations
        self.capacity = capacity
        self.items = items
        self.population = []

    # Initialize population with random solutions (binary representation)
    def initialize_population(self):
        self.population = [[random.randint(0, 1) for _ in range(self.num_items)]
for _ in range(self.population_size)]

    # Evaluate fitness of the population
    def evaluate_fitness(self):
        return [knapsack_fitness(self.items, self.capacity, individual) for
individual in self.population]

    # Select individuals based on fitness (roulette wheel selection)
    def selection(self):
        fitness_values = self.evaluate_fitness()
        total_fitness = sum(fitness_values)
        if total_fitness == 0:  # Avoid division by zero
            return random.choices(self.population, k=self.population_size)
        selected = random.choices(self.population, weights=[f / total_fitness for f
```

```
in fitness_values], k=self.population_size)
        return selected

    # Crossover (single-point) between two individuals
    def crossover(self, parent1, parent2):
        if random.random() < self.crossover_rate:
            crossover_point = random.randint(1, self.num_items - 1)
            return parent1[:crossover_point] + parent2[crossover_point:]
        return parent1

    # Mutation (random flip of a gene) of an individual
    def mutation(self, individual):
        if random.random() < self.mutation_rate:
            mutation_point = random.randint(0, self.num_items - 1)
            individual[mutation_point] = 1 - individual[mutation_point]
        return individual

    # Evolve population over generations
    def evolve(self):
        self.initialize_population()
        best_solution = None
        best_fitness = 0

        for gen in range(self.generations):
            # Selection
            selected = self.selection()

            # Crossover and Mutation
            new_population = []
            for i in range(0, self.population_size, 2):
                parent1 = selected[i]
                parent2 = selected[i+1] if i+1 < self.population_size else
selected[i]

                offspring1 = self.crossover(parent1, parent2)
                offspring2 = self.crossover(parent2, parent1)

                new_population.append(self.mutation(offspring1))
                new_population.append(self.mutation(offspring2))

            self.population = new_population

            # Evaluate fitness and track the best solution
            fitness_values = self.evaluate_fitness()
            max_fitness = max(fitness_values)
            if max_fitness > best_fitness:
                best_fitness = max_fitness
```

```
                best_solution = self.population[fitness_values.index(max_fitness)]

            print(f"Generation {gen + 1}: Best Fitness = {best_fitness}")

        return best_solution, best_fitness


# Get user input for the knapsack problem
def get_user_input():
    print("Enter the number of items:")
    num_items = int(input())
    items = []
    print("Enter the weight and value of each item (space-separated):")
    for i in range(num_items):
        weight, value = map(int, input(f"Item {i + 1}: ").split())
        items.append((weight, value))

    print("Enter the knapsack capacity:")
    capacity = int(input())

    return items, capacity, num_items


# Get user input for GEA parameters
def get_algorithm_parameters():
    print("Enter the population size:")
    population_size = int(input())
    print("Enter the mutation rate (e.g., 0.1 for 10%):")
    mutation_rate = float(input())
    print("Enter the crossover rate (e.g., 0.8 for 80%):")
    crossover_rate = float(input())
    print("Enter the number of generations:")
    generations = int(input())

    return population_size, mutation_rate, crossover_rate, generations


# Main function
if __name__ == "__main__":
    # Get user input
    items, capacity, num_items = get_user_input()
    population_size, mutation_rate, crossover_rate, generations =
get_algorithm_parameters()

    # Run GEA
    gea = GeneExpressionAlgorithm(population_size, num_items, mutation_rate,
crossover_rate, generations, capacity, items)
```

```
    best_solution, best_fitness = gea.evolve()

    # Output the best solution and fitness
    print("\nBest solution (items selected):", best_solution)
    print("Best fitness (total value):", best_fitness)
```

Output :

```
Enter the number of items:
5
Enter the weight and value of each item (space-separated):
Item 1: 10 20
Item 2: 30 40
Item 3: 50 60
Item 4: 70 80
Item 5: 90 100
Enter the knapsack capacity:
90
Enter the population size:
100
Enter the mutation rate (e.g., 0.1 for 10%):
0.2
Enter the crossover rate (e.g., 0.8 for 80%):
0.9
Enter the number of generations:
10
Generation 1: Best Fitness = 120
Generation 2: Best Fitness = 120
Generation 3: Best Fitness = 120
Generation 4: Best Fitness = 120
Generation 5: Best Fitness = 120
Generation 6: Best Fitness = 120
Generation 7: Best Fitness = 120
Generation 8: Best Fitness = 120
Generation 9: Best Fitness = 120
Generation 10: Best Fitness = 120

Best solution (items selected): [1, 1, 1, 0, 0]
Best fitness (total value): 120
```