VISVESVARAYA TECHNOLOGICAL UNIVERSITY

“JnanaSangama”, Belgaum -590014, Karnataka.

LAB RECORD

Bio Inspired Systems (23CS5BSBIS)

Submitted by

SWAPNIL SAHIL(1BM22CS300)

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 **SWAPNIL SAHIL (1BM22CS300)**, 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.

SHEETAL V A Assistant Professor Department of CSE, BMSCE	Dr. Joythi S Nayak Professor & HOD Department of CSE, BMSCE

Index

Sl. No.	Date	Experiment Title	Page No.
1	03/10/24	Genetic Algorithm	04-10
2	24/10/24	Particle Swarm Optimization	11-14
3	07/11/24	Ant Colony Optimization	15-21
4	14/11/24	Cuckoo Search	22-25
5	21/11/24	Grey Wolf Optimization	26-30
6	28/11/24	Parallel Cellular Algorithms and Programs	31-34
7	19/12/24	Optimization via Gene Expression Algorithm	35-40

Github Link:
https://github.com/Swapnilsahil/BIS-Lab

Program 1:

Genetic Algorithm

Algorithm:

Date 03/10/24

Page

LAB - 01

Genetic Algorithm for Optimization Problems:

⇒ Introduction:

→ A genetic algorithm (GA) is a type of optimization and search algorithm ~~a~~ inspired by the process of natural selection and genetics. It belongs to a broader class of algorithms known as evolutionary algorithms. GAs are particularly useful for solving complex problems where traditional methods may not be effective.

⇒ Key concepts:

1. Population : A set of candidate solutions to the problem.
2. Chromosome : A representation of a solution, often encoded as a string of bits or numbers.
3. Fitness Function : A function that evaluates how good a solution is at solving the problem.
4. Selection : The process of choosing the fittest individuals to reproduce.
5. Crossover : Combining parts of two parent solutions to create new offspring.
6. Mutation : Randomly altering parts of a solution to ~~creat~~ maintain genetic diversity.

Date __/__/__
Page ____

⇒ Applications of GAs

1. Optimization Problems:
GAs are widely used to find optimal solutions in complex spaces. such as:
   - Scheduling
   - Resource allocation
   - TSP (Traveling Salesman Problem)
   - Knapsack Problem.

2. Machine learning: GAs can optimize hyperparameters, select features, and improve model performance.

3. Engineering Design: Used for optimizing design parameters in fields like structural engineering, aerospace and automotive design.

4. Financial Modeling: GAs help in optimizing investment portfolios & predicting market trends.

5. ~~Fram~~ Robotics: Applied in path planning, control systems & robot design.

Date __/__/__
Page ____

* Travelling salesman problem (TSP)
=> Pseudocode

```
Function calculate_fitness(tour, distance_matrix):
        total distance : 0
        For i from 0 to length(tours) - 1:
            total_distance += distance_matrix[tour[i]]
                              [tour[i+1] mod length(tour)]]

        Return total distance.

Function tournament selection (population, fitness_
                               scores, tournament_size):
    selected = Random_sample(population, fitness_scores,
                              tournaments_size)
    selected = sort (selected by fitness score)
    Return selected [0]

Function order_crossover(parent1, parent2):
    size = length(parent1)
    start, end = Randomly select two indices
    child = Array of size filled with -1
    copy segment from parent1[start] to
     parent1[end] into child[start] to child[end]

    current pos = (end+1) mod size
    For each city in parent2:
        If city not in child:
            child [current_pos] = city
            current_pos = (current_pos +1) % size
    Return child
```

Date ___/___/___
Page ______

Function swap_mutation(tour, mutation_rate):
    If random() < mutation_rate:
        idx1, idx2 = Randomly select two indices from tour
        SWAP tour[idx1] & tour[idx2]

Function genetic_algorithm(cities, population (cities, population_size, generations, mutation_rate):
    distance matrix = calculate_distance_matrix (cities)

    population = Initialize random tours.
    For generation from 1 to generations:
        fitness_scores = [1/calculate_fitness

// input
coords = np.random.rand(num_cities, 2) * 100

city coordinates:
city 0 : (23.45, 67.89)
city 1 : (12.34, 45.67)
city 2 : (78.90, 12.34)
city 3 : (56.78, 90.12)
city 4 : (34.56, 23.45)

// Output:
Best Tour: [0, 4, 1, 3, 2]
Best Distance: 215.67

Code:

```
import random
import numpy as np
import math

def generate_cities(num_cities):
    return [(random.randint(0, 100), random.randint(0, 100)) for _ in range(num_cities)]

def compute_distance_matrix(cities):
    num_cities = len(cities)
    distances = [[0] * num_cities for _ in range(num_cities)]
    for i in range(num_cities):
        for j in range(num_cities):
            if i != j:
                distances[i][j] = math.sqrt(
                    (cities[i][0] - cities[j][0])**2 + (cities[i][1] - cities[j][1])**2
                )
    return distances

class TSP:
    def __init__(self, distances):
        self.distances = distances
        self.num_cities = len(distances)

    def fitness(self, route):

        total_distance = sum(
            self.distances[route[i]][route[i + 1]] for i in range(len(route) - 1)
        )
        total_distance += self.distances[route[-1]][route[0]]

class GeneticAlgorithm:
    def __init__(self, tsp, population_size=100, generations=500, mutation_rate=0.1):
        self.tsp = tsp
        self.population_size = population_size
        self.generations = generations
        self.mutation_rate = mutation_rate
        self.population = self._initialize_population()
```

```
    def _initialize_population(self):
        return [random.sample(range(self.tsp.num_cities), self.tsp.num_cities) for _ in
range(self.population_size)]

    def _select_parents(self):

        fitnesses = [self.tsp.fitness(route) for route in self.population]
        total_fitness = sum(fitnesses)
        probabilities = [f / total_fitness for f in fitnesses]
        return random.choices(self.population, probabilities, k=2)

    def _crossover(self, parent1, parent2):

        size = len(parent1)
        start, end = sorted(random.sample(range(size), 2))
        child = [-1] * size
        child[start:end] = parent1[start:end]

        p2_idx = 0
        for i in range(size):
            if child[i] == -1:
                while parent2[p2_idx] in child:
                    p2_idx += 1
                child[i] = parent2[p2_idx]
        return child

    def _mutate(self, route):

        if random.random() < self.mutation_rate:
            i, j = random.sample(range(len(route)), 2)
            route[i], route[j] = route[j], route[i]

    def evolve(self):
        for _ in range(self.generations):
            new_population = []
            for _ in range(self.population_size):
                parent1, parent2 = self._select_parents()
```

```
            child = self._crossover(parent1, parent2)
            self._mutate(child)
            new_population.append(child)
        self.population = new_population

    def get_best_solution(self):

        best_route = min(self.population, key=lambda route: 1 / self.tsp.fitness(route))
        best_distance = 1 / self.tsp.fitness(best_route)
        return best_route, best_distance

if __name__ == "__main__":
    num_cities = 5
    cities = generate_cities(num_cities)
    distances = compute_distance_matrix(cities)

    print("City Coordinates:")
    for i, city in enumerate(cities):
        print(f"City {i}: {city}")

    tsp = TSP(distances)
    ga = GeneticAlgorithm(tsp, population_size=50, generations=100, mutation_rate=0.2)
    ga.evolve()
    best_route, best_distance = ga.get_best_solution()
    print("\nBest route:", best_route)
    print("Best distance:", best_distance)
```

Program 2:
Particle Swarm Optimization

Algorithm:

Date 24/10/24
Page

LAB-02

* Practicle Swarm Optimization for function Optimation:

→ Algorithm:

step-1: Randomly initialize swarm population of N particles Xi (i=1, 2 ... n)

step 2: select hyperparameter values w, c1 & c2

step3: For Iter in range(max_iter):
    For i in range(N):
        a. Compute new velocity of ith particle
            swarm[i].velocity =
                w*swarm[i].velocity +
                r1*c1*(swarm[i].bestPos -
                    swarm[i].position) +
                r2*c2*(best_pos_swarm -
                    swarm[i].position)

        b. Compute new position of ith particle using its new velocity ~~swarm[i]~~
            swarm[i].position += swarm[i].velocity

        c. If position is not in range[minx, maxx] then clip it
            if swarm[i].position < minx:
                swarm[i].position = minx
            elif swarm[i].position > maxx:
                swarm[i].position = maxx

        d. update new best of this particle

and new best of swarm
if swaInsensitive to scaling of design variables.rm[i]. fitness< swarm[i]
best fitness:
swarm[i].bestfitness = swarm[i].fitness
swarm[i]. bestPos= swarm[i].position

if swarm[i].fitness < best-fitness swarm
best-fitness_swarm = swarm[i].fitness
best pos-swarm= swarm[i]. position
End-for
End-for

Step4: Return best particle of swarm

Program:
// input:
- objective function : $x^2 + y^2$
- dimensions : 2
- iterations : 100
- population size : 50
- w = 0.7 (Inertia weight)
- cognitive parameter = 1.4
- social parameter = 1.4

24-10-24

// O/p

Best position found : [4.0479, -2.23363]
Best value found : 2.1374591

Code:

```
import random
import numpy as np

def objective_function(position):
    """The function to be minimized."""
    x, y = position
    return x**2 + y**2

def pso(objective_function, dimensions, iterations, population_size, w=0.7, c1=1.4, c2=1.4):

    particles = []
    for _ in range(population_size):
        position = np.random.uniform(-10, 10, dimensions)
        velocity = np.random.uniform(-1, 1, dimensions)
        particles.append({
            'position': position,
            'velocity': velocity,
            'best_position': position.copy(),
            'best_value': objective_function(position)
        })

    global_best_position = particles[0]['best_position'].copy()
    global_best_value = particles[0]['best_value']

    for _ in range(iterations):
        for particle in particles:

            r1 = random.random()
            r2 = random.random()
            particle['velocity'] = (w * particle['velocity'] +
                                    c1 * r1 * (particle['best_position'] - particle['position']) +
                                    c2 * r2 * (global_best_position - particle['position']))

            particle['position'] = particle['position'] + particle['velocity']

            particle['position'] = np.clip(particle['position'], -10, 10)
```

```
            value = objective_function(particle['position'])

            if value < particle['best_value']:
                particle['best_value'] = value
                particle['best_position'] = particle['position'].copy()

            if value < global_best_value:
                global_best_value = value
                global_best_position = particle['position'].copy()

    return global_best_position, global_best_value

dimensions = 2
iterations = 100
population_size = 50

best_position, best_value = pso(objective_function, dimensions, iterations, population_size)
print(f"Best position found: {best_position}")
print(f"Best value found: {best_value}")
```

Program 3:
Ant Colony Optimization

Algorithm:

Date 07/11/24
Page

LAB-03

* Ant colony Optimization for Traveling salesman problem.

→ Algorithm:

initialize pheromone values $\forall i, j \in [1,n] : \tau_{ij} \rightarrow \tau_0$

repeat

for each ant $l \in [1, \ldots m]$ do

initialize selection set $S \rightarrow \{1, \ldots, n\}$

randomly choose starting city $i_0 \in S$ for ant l

move to starting city $i \rightarrow i_0$

while $S \neq 0$ do

remove current city from selection set $S \rightarrow S \setminus \{i\}$

choose next city j in tour with probability $p_{ij} = \dfrac{\tau_{ij}^{\alpha} \cdot \eta_{ij}^{\beta}}{\sum_{n \in S} \tau_{in}^{\alpha} \cdot \eta_{in}^{\beta}}$

update solution vector $\pi_l(i) \rightarrow j$

move to new city $i \rightarrow j$

end while

finalize solution vector $\pi_l(i) \rightarrow i_0$

end for

for each solution π_l, $l \in \{1, \ldots m\}$ do

calculate tourlength $f(\pi_l) \rightarrow \sum_{i=1} d_{i\pi_l(i)}$

end for

for all (i,j) do

evaporate pheromone $\tau_{ij} \rightarrow (1-\rho) \cdot \tau_{ij}$

end for

Date __/__/__
Page ____

determine best solution of iteration
$\pi^{+} = \arg\min_{i \in [1,m]} f(\pi_i)$

if π^{+} better than current best π^{*}, i.e.
$f(\pi^{+}) < f(\pi^{*})$ then
set $\pi^{*} \rightarrow \pi^{+}$
end if
for all $(i,j) \in \pi^{+}$ do
reinforce $\tau_{ij} \rightarrow \tau_{ij} + \Delta/2$
end for
for all $(i,j) \in \pi^{*}$ do
reinforce $\tau_{ij} \rightarrow \tau_{ij} + \Delta/2$
end for
until condition for termination met

//input:
Number of ants = 20
Number of cities = 20
alpha = 1.0
Beta = 2.0
RHO = 0.1
Q = 100 (pheromone deposit constant)
max interation = 100

//output:
Iteration 0: Best length: 98.92
Iteration 10: Best length: 79.23
Iteration 20: Best length: 79.23
Iteration 30: Best length: 79.23
Iteration 40: Best length: 77.88
Iteration 50: Best length: 77.88

Date ___/___/___
Page ________

Iteration 60 : Best length : 77.88
Iteration 70 : Best length : 77.88
Iteration 80 : Best length : 77.88
Iteration 90 : Best length : 77.88

7-11-24

Code:

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

# Define constants for the algorithm
NUM_ANTS = 50
NUM_CITIES = 20  # Now we have 20 cities
ALPHA = 1.0  # Influence of pheromone
BETA = 2.0   # Influence of distance
RHO = 0.1    # Pheromone evaporation rate
Q = 100      # Pheromone deposit constant
MAX_ITER = 100  # Maximum number of iterations

# Predefined 20 cities (coordinates in 2D space)
def generate_cities():
    cities = np.array([
        [5, 10], [11, 5], [14, 9], [12, 15], [8, 13],  # Cities 0-4
        [10, 10], [13, 7], [16, 5], [14, 3], [18, 6],  # Cities 5-9
        [4, 2], [7, 1], [8, 5], [6, 7], [4, 10],  # Cities 10-14
        [15, 18], [12, 17], [3, 18], [17, 12], [19, 8]  # Cities 15-19
    ])
    return cities

# Compute the distance matrix
def compute_distance_matrix(cities):
    num_cities = len(cities)
    distance_matrix = np.zeros((num_cities, num_cities))
    for i in range(num_cities):
        for j in range(i + 1, num_cities):
            dist = np.linalg.norm(cities[i] - cities[j])
            distance_matrix[i, j] = dist
            distance_matrix[j, i] = dist
    return distance_matrix

# Initialize pheromone matrix
def initialize_pheromone_matrix(num_cities):
```

```
    pheromone_matrix = np.ones((num_cities, num_cities))  # Pheromone starts as 1 for all
edges
    np.fill_diagonal(pheromone_matrix, 0)  # No pheromone on the diagonal (self-loops)
    return pheromone_matrix

# Calculate the total length of a tour
def calculate_tour_length(tour, dist_matrix):
    length = 0
    for i in range(len(tour) - 1):
        length += dist_matrix[tour[i], tour[i + 1]]
    length += dist_matrix[tour[-1], tour[0]]  # Returning to the start
    return length

# Ant solution construction (probabilistic decision on next city)
def construct_solution(num_cities, pheromone_matrix, dist_matrix):
    tour = [random.randint(0, num_cities - 1)]  # Start from a random city
    visited = set(tour)

    while len(tour) < num_cities:
        current_city = tour[-1]
        probabilities = []
        for next_city in range(num_cities):
            if next_city not in visited:
                pheromone = pheromone_matrix[current_city, next_city] ** ALPHA
                distance = (1.0 / dist_matrix[current_city, next_city]) ** BETA
                probabilities.append(pheromone * distance)
            else:
                probabilities.append(0)

        total_prob = sum(probabilities)
        probabilities = [p / total_prob for p in probabilities]

        # Choose the next city based on the probabilities
        next_city = np.random.choice(range(num_cities), p=probabilities)
        tour.append(next_city)
        visited.add(next_city)

    return tour
```

```python
# Update the pheromone matrix based on the solutions found by ants
def update_pheromone(pheromone_matrix, all_tours, dist_matrix, best_tour):
    # Evaporate pheromone
    pheromone_matrix *= (1 - RHO)

    # Add pheromone for all ants
    for tour in all_tours:
        tour_length = calculate_tour_length(tour, dist_matrix)
        for i in range(len(tour) - 1):
            pheromone_matrix[tour[i], tour[i + 1]] += Q / tour_length
        pheromone_matrix[tour[-1], tour[0]] += Q / calculate_tour_length(tour, dist_matrix)

    # Add pheromone for the best tour
    best_length = calculate_tour_length(best_tour, dist_matrix)
    for i in range(len(best_tour) - 1):
        pheromone_matrix[best_tour[i], best_tour[i + 1]] += Q / best_length
    pheromone_matrix[best_tour[-1], best_tour[0]] += Q / best_length

# Main ACO algorithm for solving TSP
def ant_colony_optimization(num_cities, dist_matrix, pheromone_matrix, max_iter):
    best_tour = None
    best_tour_length = float('inf')

    # Main loop
    for iteration in range(max_iter):
        all_tours = []

        # Step 1: All ants construct their solutions
        for _ in range(NUM_ANTS):
            tour = construct_solution(num_cities, pheromone_matrix, dist_matrix)
            all_tours.append(tour)
            tour_length = calculate_tour_length(tour, dist_matrix)

            # Step 2: Update the best tour if necessary
            if tour_length < best_tour_length:
                best_tour = tour
                best_tour_length = tour_length
```

```
        # Step 3: Update pheromone matrix
        update_pheromone(pheromone_matrix, all_tours, dist_matrix, best_tour)

        # Optional: print progress every 10 iterations
        if iteration % 10 == 0:
            print(f"Iteration {iteration}: Best tour length = {best_tour_length:.2f}")

    return best_tour, best_tour_length


# Main Execution
if __name__ == "__main__":
    # Step 1: Generate predefined cities and distance matrix
    cities = generate_cities()
    dist_matrix = compute_distance_matrix(cities)

    # Step 2: Initialize pheromone matrix
    pheromone_matrix = initialize_pheromone_matrix(NUM_CITIES)

    # Step 3: Run ACO algorithm
    best_tour, best_tour_length = ant_colony_optimization(NUM_CITIES, dist_matrix,
pheromone_matrix, MAX_ITER)

    # Step 4: Output the best tour and visualize it
    print(f"Best tour length: {best_tour_length:.2f}")
```

Program 4:
Cuckoo Search Algorithm

Algorithm:

Date 14/11/24
Page

LAB-04

* Cuckoo search Algorithm

Algorithm:
Initialize population of n cuckoos.
Evaluate fitness of each cukoo
// Input
// n = initial population size
// Pa: fraction of worse nests to be abandoned and replaced
// Max iterations: the maximum no. of iteration
// f : the objective f^n to optimize ()

// output
// The best solution found.

Generate the initial population of n host nests x_i (i=1, 2, ---n)

while t < Max iterations:
Get a cuckoo randomly by levy flights
Evaluate its quality fitness F_i

j ← choose a nest among n randomly

if $F_i > F_j$:
Replace j by the new solution

if A fraction (Pa) of worse nests are abandoned and new ones are built;
Keep the best solutions
Rank the solutions and find the current best
Postprocess results and visualization.

Date ___/___/___
Page ________

Application Name: Traffic signal Optimization

Inputs:

dimensions (number of interactions) : dim = 3

bounds (Range of green light time) : [10,120]

num-nests (No. of candidate solutions) : 20

max_iter (the max^m no. of iterations) : 100

p_a (probability of abandoning the worst nests) : 0.25

lambda (The parameter of levy flight step size) : 1.5

// output:

best solution : [90.5, 110.3, 75.8]

best-fitness : 325.5 (total waiting time)

o/p seen (G.H)

14-11-24

Code:

```
#cuckoo search(Traffic Signal Optimization)
import numpy as np
from scipy.special import gamma

def fitness_function(x):
    waiting_times = np.array([10 + (x[i] ** 2) / 100 for i in range(len(x))])
    total_waiting_time = np.sum(waiting_times)
    return total_waiting_time

def levy_flight(dim, beta=1.5):
    sigma_u = np.power((gamma(1 + beta) * np.sin(np.pi * beta / 2) /
                    gamma((1 + beta) / 2) * beta * (2 ** (beta - 1))), 1 / beta)
    u = np.random.normal(0, sigma_u, dim)
    v = np.random.normal(0, 1, dim)
    step = u / np.power(np.abs(v), 1 / beta)
    return step

def cuckoo_search(dim, bounds, num_nests, max_iter, p_a=0.25, Lambda=1.5):
    nests = np.random.uniform(bounds[0], bounds[1], (num_nests, dim))
    fitness = np.array([fitness_function(nest) for nest in nests])

    best_idx = np.argmin(fitness)
    best_nest = nests[best_idx]
    best_fitness = fitness[best_idx]

    for iter in range(max_iter):
        new_nests = np.copy(nests)
        for i in range(num_nests):
            step = levy_flight(dim, Lambda)
            new_nests[i] = nests[i] + step
            new_nests[i] = np.clip(new_nests[i], bounds[0], bounds[1])
```

```
        new_fitness = np.array([fitness_function(nest) for nest in new_nests])

        for i in range(num_nests):
            if new_fitness[i] < fitness[i]:
                nests[i] = new_nests[i]
                fitness[i] = new_fitness[i]

        if np.random.rand() < p_a:
            random_idx = np.random.randint(num_nests)
            nests[random_idx] = np.random.uniform(bounds[0], bounds[1], dim)
            fitness[random_idx] = fitness_function(nests[random_idx])

        current_best_idx = np.argmin(fitness)
        current_best_fitness = fitness[current_best_idx]

        if current_best_fitness < best_fitness:
            best_fitness = current_best_fitness
            best_nest = nests[current_best_idx]



    return best_nest, best_fitness

dim = 3
bounds = [10, 120]
num_nests = 20
max_iter = 100

best_solution, best_value = cuckoo_search(dim, bounds, num_nests, max_iter)

print("\n--- Best Solution ---")
print("Green Light Timings (seconds):", best_solution)
print("Best Fitness Value (Total Waiting Time):", best_value)
```

Program 5:

Grey Wolf Optimization

Algorithm:

Date 21/11/24
Page

LAB-05

* Grey wolf optimization:

- Algorithm

Initialize population of wolves with random position.
Initialize α, β, δ position & ~~their~~ their fitness

for each iteration from 1 to max
    for each wolf i in population
        if fitness(i) > fitness(α)
            update α position
        else if fitness(i) > fitness(β)
            update β position
        else if fitness(i) > fitness(δ)
            update δ position

    for each wolf i in population
        calculate α, β, δ distance
        update position of wolf using:
            $X_i = X_i + A \cdot D(\alpha), A \cdot D(\beta), A \cdot D(\delta)$

        if position is outside bounds
            Adjust position to be within bounds

return best solution found (α position & fitness)

21/11

- α → best fitness
- β → 2nd best fitness
- δ → third best fitness

$X_i = X_i + A \cdot D$

Date ___ / ___ / ___
Page ______

Application: Water distribution optimization

Inputs:

- Number of wolves : 30
- Maximum iterations : 100
- Number of nodes : 5
- Demand at each node : [50, 40, 30, 60, 80]
- Pipe costs : [1, 2, 1.5, 3, 2]
- pump costs : [0.1, 0.1, 0.1, 0.1, 0.1]

// Output:

Best Pipe Sizes: [0.703, 0.734, 0.586, 0.483, 1.89]
Best Flow Rates: [0.884, 1.045, 1.117, 0.810, 1.385]

Seen

Code:

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

def objective_function(pipe_sizes, flow_rates, demand, node_pressure, pipe_costs,
pump_costs):
    pipe_cost = np.sum(pipe_sizes ** 2 * pipe_costs)
    pump_cost = np.sum(flow_rates * pump_costs)

    pressure_penalty = 0
    for i in range(len(node_pressure)):
        if node_pressure[i] < 20:
            pressure_penalty += (20 - node_pressure[i]) ** 2

    demand_penalty = 0
    for i in range(len(demand)):
        if flow_rates[i] < demand[i]:
            demand_penalty += (demand[i] - flow_rates[i]) ** 2

    total_cost = pipe_cost + pump_cost + pressure_penalty + demand_penalty
    return total_cost

class GreyWolfOptimization:
    def __init__(self, num_wolves, max_iter, demand, pipe_costs, pump_costs,
num_nodes):
        self.num_wolves = num_wolves
        self.max_iter = max_iter
        self.demand = demand
        self.pipe_costs = pipe_costs
        self.pump_costs = pump_costs
        self.num_nodes = num_nodes

        self.wolves = np.random.rand(self.num_wolves, 2 * self.num_nodes)
```

```
        self.alpha = None
        self.beta = None
        self.delta = None
        self.alpha_score = float('inf')
        self.beta_score = float('inf')
        self.delta_score = float('inf')

    def fitness(self, wolf):
        pipe_sizes = wolf[:self.num_nodes]
        flow_rates = wolf[self.num_nodes:]

        node_pressure = np.random.rand(self.num_nodes) * 50

        return objective_function(pipe_sizes, flow_rates, self.demand, node_pressure,
self.pipe_costs, self.pump_costs)

    def update_positions(self):
        for i in range(self.num_wolves):
            A = 2 * np.random.rand(1) - 1
            C = 2 * np.random.rand(1)
            D_alpha = np.abs(C * self.alpha - self.wolves[i])
            X1 = self.alpha - A * D_alpha

            A = 2 * np.random.rand(1) - 1
            C = 2 * np.random.rand(1)
            D_beta = np.abs(C * self.beta - self.wolves[i])
            X2 = self.beta - A * D_beta

            A = 2 * np.random.rand(1) - 1
            C = 2 * np.random.rand(1)
            D_delta = np.abs(C * self.delta - self.wolves[i])
            X3 = self.delta - A * D_delta
```

```
            self.wolves[i] = (X1 + X2 + X3) / 3
    def optimize(self):
        for _ in range(self.max_iter):
            for i in range(self.num_wolves):
                fitness_value = self.fitness(self.wolves[i])

                if fitness_value < self.alpha_score:
                    self.alpha_score = fitness_value
                    self.alpha = self.wolves[i]

                elif fitness_value < self.beta_score:
                    self.beta_score = fitness_value
                    self.beta = self.wolves[i]

                elif fitness_value < self.delta_score:
                    self.delta_score = fitness_value
                    self.delta = self.wolves[i]
            self.update_positions()
        return self.alpha
num_wolves = 30
max_iter = 100
num_nodes = 5
demand = np.array([50, 40, 30, 60, 80])
pipe_costs = np.array([1, 2, 1.5, 3, 2])
pump_costs = np.array([0.1, 0.1, 0.1, 0.1, 0.1])

gwo = GreyWolfOptimization(num_wolves, max_iter, demand, pipe_costs, pump_costs,
num_nodes)
best_solution = gwo.optimize()
best_pipe_sizes = best_solution[:num_nodes]
best_flow_rates = best_solution[num_nodes:]
print("Best Pipe Sizes:", best_pipe_sizes)
print("Best Flow Rates:", best_flow_rates)
```

Program:6

Parallel Cellular Algorithms and Programs

Algorithm:

Date 28/11/24

LAB-06

★ Parallel Cellular Algorithms and Programs

• Pseudocode:

```
# constants
Grid_Width = N
Grid_Height = M
Max_Generations = T

# Initialize grid (random 0 or 1)
grid = InitializeGrid(Grid_Width, Grid_Height)

# Count live neighbours
function CountLiveNeighbours(grid, i, j):
        live_neighbours = 0
        for each neighbour (x, y) of (i, j):
                if grid[x][y] == 1:
                        live_neighbours += 1
        return live_neighbours

# Apply Game of Life rules
function ApplyRules(grid, i, j):
        live_neighbors = CountLiveNeighbors(grid, i, j)
        if grid[i][j] == 1:
                return 1 if live_neighbors == 2 or
                  live_neighbors == 3 else 0
        else:
                return 1 if live_neighbors == 3 else 0

# update grid (parallel update for all cells)
function UpdateGrid(grid):
        new_grid = CopyGrid(grid)
```

for i in range(0, Grid-height):
    for j in range(0, Grid-width):
        new_grid[i][j] = ApplyRules(grid, i, j)

return new_grid

* Implementation (Game of Life rules)
// Input:
Grid-width : 10
Grid-Height : 10
Max-generation: 20

Initial generation / Grid:

1	0	1	0	0	1	1	1	0	1
0	1	1	1	0	1	0	1	1	0
0	0	1	0	1	0	0	0	0	0
1	0	1	0	0	0	1	0	0	0
1	0	0	0	0	0	1	1	0	0
1	0	1	0	0	1	1	0	0	0
0	0	1	0	1	1	0	0	1	0
1	0	1	1	0	0	0	0	1	0
1	0	0	1	0	1	0	0	0	0
0	0	1	1	1	1	1	0	0	0

No. of alive cells = 35

28-11-24 Generation 20:
Number of alive cells : 39

Code:

```
#Parallel Cellular Algorithm n programs
import random
import copy

GRID_WIDTH = 10
GRID_HEIGHT = 10
MAX_GENERATIONS = 20

def initialize_grid(width, height):
    return [[random.randint(0, 1) for _ in range(width)] for _ in range(height)]

def count_live_neighbors(grid, i, j):
    live_neighbors = 0
    directions = [(-1, -1), (-1, 0), (-1, 1),
                  ( 0, -1),          ( 0, 1),
                  ( 1, -1), ( 1, 0), ( 1, 1)]

    for dx, dy in directions:
        x = (i + dx) % len(grid)
        y = (j + dy) % len(grid[0])
        live_neighbors += grid[x][y]

    return live_neighbors



def apply_rules(grid, i, j):
    live_neighbors = count_live_neighbors(grid, i, j)
    if grid[i][j] == 1:
        return 1 if live_neighbors == 2 or live_neighbors == 3 else 0
    else:
        return 1 if live_neighbors == 3 else 0
def update_grid(grid):
```

```
    new_grid = copy.deepcopy(grid)
    for i in range(len(grid)):
        for j in range(len(grid[0])):
            new_grid[i][j] = apply_rules(grid, i, j)
    return new_grid


def display_grid(grid):
    for row in grid:
        print(' '.join(str(cell) for cell in row))
    print("\n" + "="*20 + "\n")

def count_alive_cells(grid):
    return sum(sum(row) for row in grid)


def game_of_life(grid_width, grid_height, max_generations):
    grid = initialize_grid(grid_width, grid_height)
    print("Initial Grid:")
    display_grid(grid)

    for generation in range(max_generations):
        print(f"Generation {generation + 1}:")
        grid = update_grid(grid)
        # display_grid(grid)
        alive_cells = count_alive_cells(grid)
        print(f"Number of alive cells: {alive_cells}")


if __name__ == "__main__":
    game_of_life(GRID_WIDTH, GRID_HEIGHT, MAX_GENERATIONS)
```

Program 7:

Optimization via Gene Expression Algorithm

Algorithm:

Date 19 / 12 / 24
Page

LAB-07

★ Gene Expression Algorithms (GEA):

Algorithm:

1. Define the Problem
- Define the optimization problem and objective function:
  Example: f(x) → minimize or Maximize

2. Initialize Parameters
- Set Population_Size
- Set Number_of_Genes (Length of Genetic sequence)
- Set Mutation_Rate
- Set Crossover_Rate
- Set Max_Generations (Number of iterations)

3. Initialize Population
- For each individual i in Population_Size:
  Generate a random genetic sequence (List of genes)
  Store the genetic sequence in Population.

4. Evaluate Fitness
- For each individual i in Population:
  Translate the genetic sequence into a solution (Gene Expression)
  Compute fitness using the objective function
  Store the fitness value for individual i

5. Iterate Until Stopping Criteria
- For generation in range(1, Max_Generations):
  a. selection:
  - Select individuals based on fitness (e.g. Roulette wheel or Tournament selection)
  - Store selected individuals as Mating Pool

Date / /
Page

b. Crossover:
- For each pair of individuals in Mating_Pool:
  with probability Crossover_Rate:
    perform crossover to produce offspring
    Add offspring to New_Population

c. Mutation:
- For each individual in New_Population:
  For each gene in the genetic sequence:
    with probability Mutation_Rate:
      Randomly alter the gene (introduce variation)

d. Gene Expression:
- For each individual in New_Population:
  Translate the genetic sequence into a solution
  compute the fitness using the objective fitness

e. Replacement:
- Replace the old population with New_Population

f. Track Best Solution:
- Update the best solution found so far based on fitness

6. Output the Best Solution
- Return the best genetic sequence and its corresponding fitness value.

Date / /
Page

Implementation (Application - Minimizing quad. equ)

// Input

function: $x^2 - 2x + 10$

pop_size = 20

generation = 100

x_min, x_max = -10, 10

crossover_rate = 0.8

// Output:

Best solution: x = 1.00000, Minimum value: f(x) = 9.00000

Code:

```
import random

# Objective function
def objective_function(x):
    return x**2 - 2*x + 10

# Initialize population
def initialize_population(pop_size, x_min, x_max):
    return [random.uniform(x_min, x_max) for _ in range(pop_size)]

# Evaluate fitness
def evaluate_fitness(population):
    return [objective_function(x) for x in population]

# Selection: Tournament Selection
def select_parents(population, fitness, tournament_size=3):
    selected = []
    for _ in range(len(population)):
        competitors = random.sample(list(zip(population, fitness)), tournament_size)
        winner = min(competitors, key=lambda x: x[1])
        selected.append(winner[0])
    return selected

# Crossover: Arithmetic crossover
def crossover(parents, crossover_rate=0.8):
    offspring = []
    for i in range(0, len(parents), 2):
        p1, p2 = parents[i], parents[(i+1) % len(parents)]
        if random.random() < crossover_rate:
            alpha = random.random()
            child1 = alpha * p1 + (1 - alpha) * p2
            child2 = alpha * p2 + (1 - alpha) * p1
```

```
        else:
            child1, child2 = p1, p2
        offspring.extend([child1, child2])
    return offspring

# Mutation: Add small random noise
def mutate(offspring, mutation_rate=0.1, mutation_step=0.5):
    for i in range(len(offspring)):
        if random.random() < mutation_rate:
            offspring[i] += random.uniform(-mutation_step, mutation_step)
    return offspring

# Gene Expression Algorithm
def gene_expression_algorithm(pop_size, generations, x_min, x_max):
    population = initialize_population(pop_size, x_min, x_max)
    best_solution = None
    best_fitness = float("inf")
    for gen in range(generations):
        fitness = evaluate_fitness(population)
        if min(fitness) < best_fitness:
            best_fitness = min(fitness)
            best_solution = population[fitness.index(best_fitness)]

        parents = select_parents(population, fitness)
        offspring = crossover(parents)
        population = mutate(offspring)

    return best_solution, best_fitness

pop_size = 20
generations = 100
x_min, x_max = -10, 10
```

```
best_x, best_fitness = gene_expression_algorithm(pop_size, generations, x_min, x_max)
print(f"Best solution: x = {best_x:.5f}, Minimum value: f(x) = {best_fitness:.5f}")
```