VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB RECORD

BIO INSPIRED SYSTEMS (23CS5BSBIS)

Submitted by

Manav Kumar (1BM22CS348)

in partial fulfilment 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 **Manav Kumar (1BM22CS348),** who is bonafide student of **B.M.S. College of Engineering.** It is in partial fulfilment 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.

Supreeth S
Assistant Professor
Department of CSE, BMSCE

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

Index

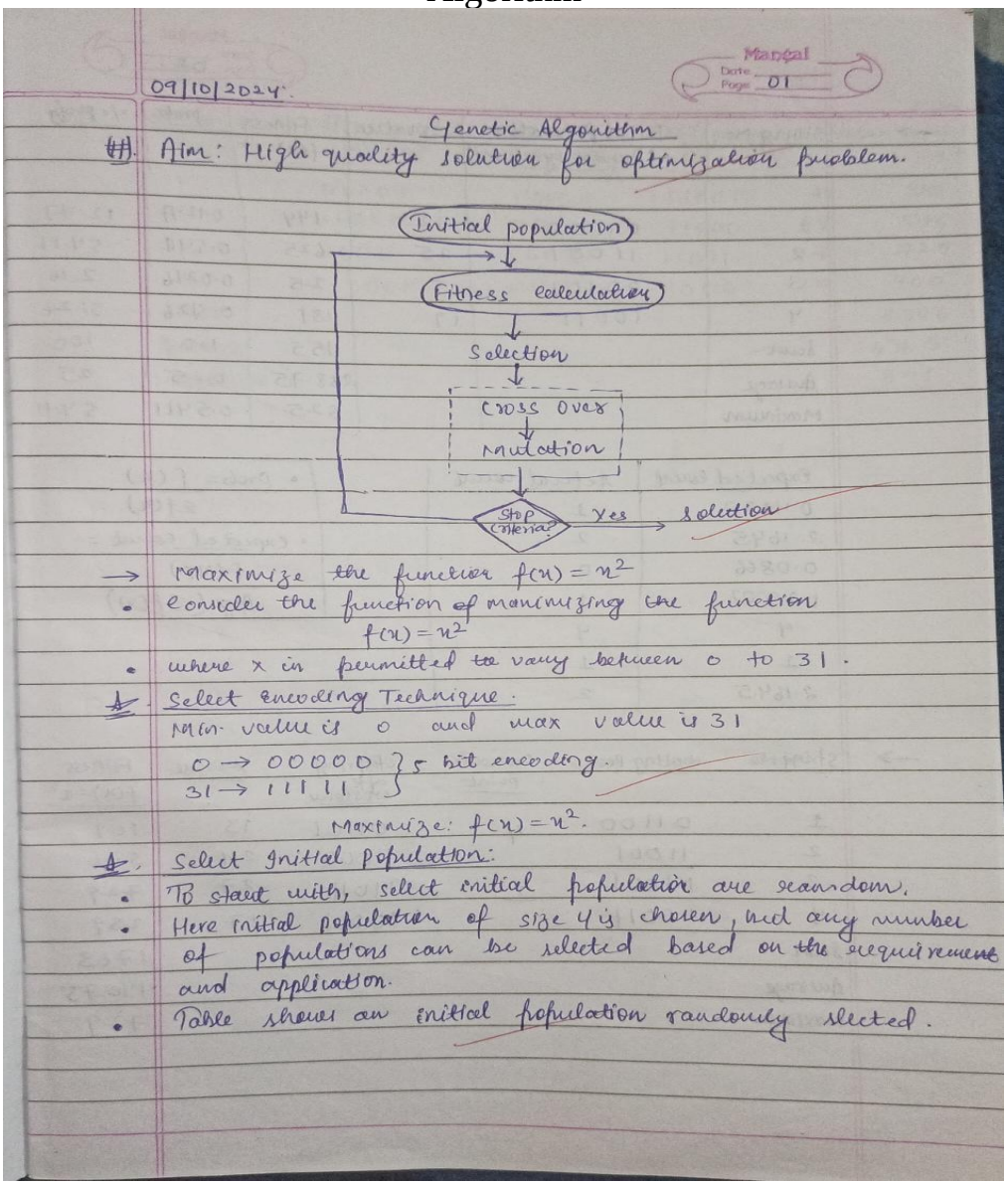
Github Link: https://github.com/Manav-Kumar123/MANAV_1BM22CS348_BIS

Laboratory Program - 1

Genetic Algorithm for Optimization Problems

Genetic Algorithms (GA) are inspired by the process of natural selection and genetics, where the fittest individuals are selected for reproduction to produce the next generation. GAs are widely used for solving optimization and search problems. Implement a Genetic Algorithm using Python to solve a basic optimization problem, such as finding the maximum value of a mathematical function.

Algorithm



→

String No.	Initial population (Randomly selected)	x value	Fitness $f(x)=x^2$	Prob	% prob
1	01100	12	144	0.1247	12.47
2	11001	25	625	0.5411	54.11
3	00101	5	25	0.0216	2.16
4	10011	19	181	0.3126	31.26
Sum			1155	1.0	100
Average			288.75	0.25	25
Maximum			625	0.5411	54.11

Expected count	Actual count
0.4987	1
2.1645	2
0.0866	0
1.2502	1
4	4
1	1
2.1645	2

- $\text{Prob} = \frac{f(x)}{\Sigma f(x)}$
- $\text{Expected count} = \frac{f(x_i)}{\text{Avg}(\Sigma f(x))}$

→

String No	Mating Pool	Crossover point	Offspring after crossover	x-value	Fitness $f(x)=x^2$
1	01100	4	01101	13	169
2	11001		11000	24	576
3	11001	2	11011	27	729
4	10011		10001	17	289
Sum					1763
Average					440.75
Maximum					729.

Manual
Date
Page 03.

String No.	offspring after crossover	mutation chromosome for flipping	offspring after mutation	X value	Fitness $f(x) = x^2$
1	01101	10000	11101	29	841
2	11000	00000	11000	24	576
3	11011	00000	11011	27	729
4	10001	00101	10100	20	400
Sum					2546
Average					636.5
Maximum					841

9/10/24

Code

```
import random

def fitness_function(x, y):
    return x ** 2

population_size = 100
mutation_rate = 0.1
crossover_rate = 0.8
num_generations = 50
variable_bounds = [-10, 10]

def initialize_population(population_size, bounds):
    population = []
    for _ in range(population_size):
        x = random.uniform(bounds[0], bounds[1])
        y = random.uniform(bounds[0], bounds[1])
        population.append([x, y])
    return population

def evaluate_population(population):
    fitness_scores = []
    for individual in population:
        fitness_scores.append(fitness_function(individual[0], individual[1]))
    return fitness_scores

def selection(population, fitness_scores):
    total_fitness = sum(fitness_scores)
    selected_population = []
    for _ in range(len(population)):
        pick = random.uniform(0, total_fitness)
        current = 0
        for individual, score in zip(population, fitness_scores):
            current += score
            if current > pick:
                selected_population.append(individual)
                break
    return selected_population

def crossover(parent1, parent2, crossover_rate):
    if random.random() < crossover_rate:
        crossover_point = random.randint(1, len(parent1) - 1)
        child1 = parent1[:crossover_point] + parent2[crossover_point:]
        child2 = parent2[:crossover_point] + parent1[crossover_point:]
    else:
        child1, child2 = parent1, parent2
```

```
    return child1, child2

def mutate(individual, mutation_rate, bounds):
    if random.random() < mutation_rate:
        mutation_position = random.randint(0, len(individual) - 1)
        mutation_value = random.uniform(bounds[0], bounds[1])
        individual[mutation_position] = mutation_value
    return individual

def genetic_algorithm():
    # Display student information at the start
    print("Student Name: Likhith M")
    print("USN: 1BM22CS135")
    print("-" * 40)

    population = initialize_population(population_size, variable_bounds)
    overall_best_solution = None
    overall_best_fitness = float('-inf')

    for generation in range(num_generations):
        fitness_scores = evaluate_population(population)

        generation_best_fitness = max(fitness_scores)
        generation_best_solution = population[fitness_scores.index(generation_best_fitness)]

        if generation_best_fitness > overall_best_fitness:
            overall_best_fitness = generation_best_fitness
            overall_best_solution = generation_best_solution

        # Displaying the output for generations that are multiples of 10
        if (generation + 1) % 10 == 0:
            print(f"Generation {generation + 1}:")
            print(f"  Best fitness   = {generation_best_fitness}")
            print(f"  Best solution  = {generation_best_solution}")
            print("-" * 40)

        selected_population = selection(population, fitness_scores)
        next_population = []

        for i in range(0, population_size, 2):
            parent1 = selected_population[i]
            parent2 = selected_population[i + 1]
            child1, child2 = crossover(parent1, parent2, crossover_rate)
            child1 = mutate(child1, mutation_rate, variable_bounds)
            child2 = mutate(child2, mutation_rate, variable_bounds)
            next_population.append(child1)
            next_population.append(child2)
```

```
    population = next_population

print(f"Best solution found after {num_generations} generations: {overall_best_solution}")
print(f"Best fitness value: {overall_best_fitness}")

genetic_algorithm()
```

Output:

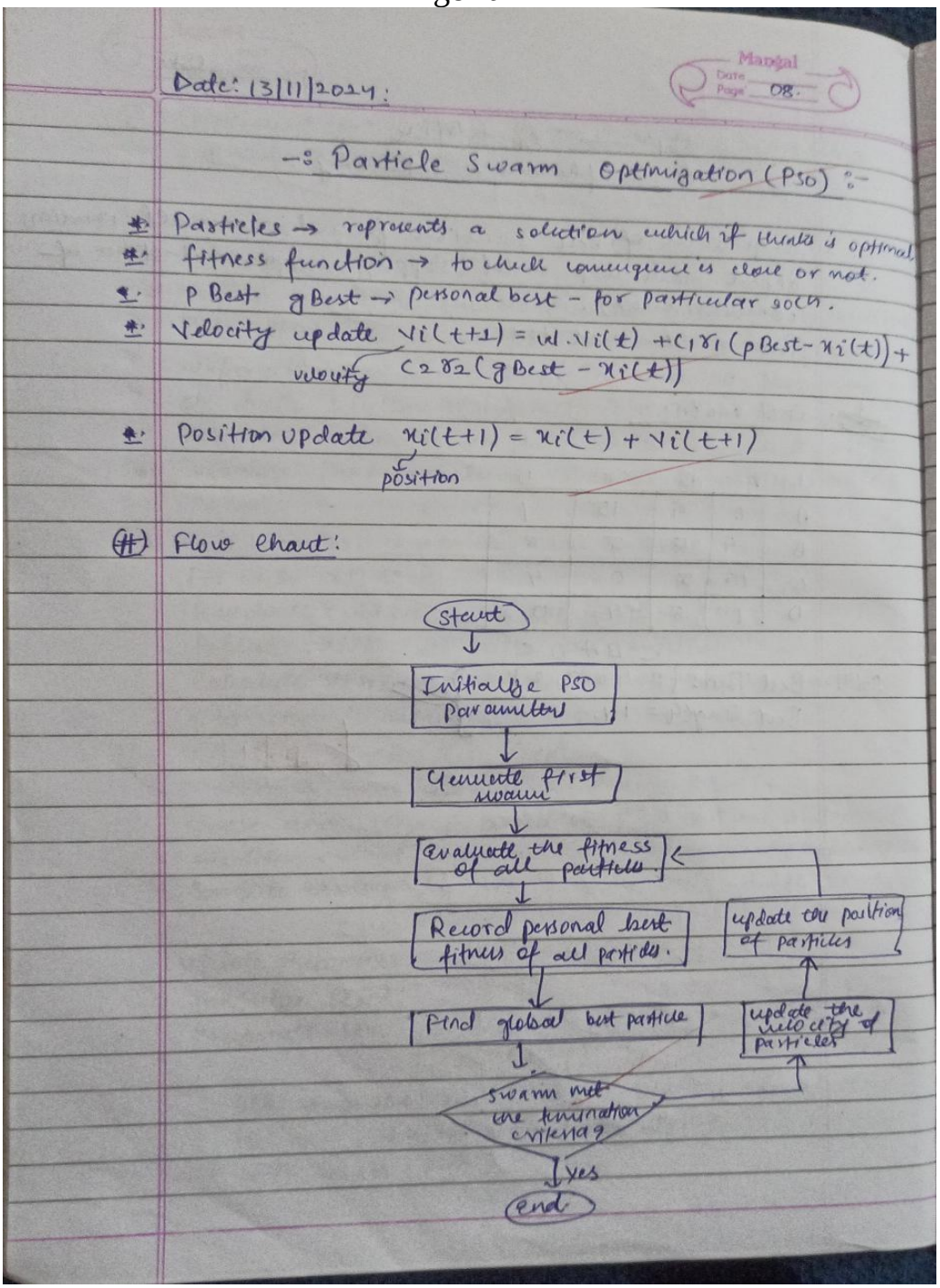
```
----------------------------------------
Generation 10:
  Best fitness   = 99.44798466551265
  Best solution  = [-9.972361037663681, -8.984613074594794]
----------------------------------------
Generation 20:
  Best fitness   = 99.44798466551265
  Best solution  = [-9.972361037663681, -9.634560295132298]
----------------------------------------
Generation 30:
  Best fitness   = 99.44798466551265
  Best solution  = [-9.972361037663681, -7.617025418591046]
----------------------------------------
Generation 40:
  Best fitness   = 99.13270771662218
  Best solution  = [-9.956540951385787, -1.8134161191052698]
----------------------------------------
Generation 50:
  Best fitness   = 99.33177287264611
  Best solution  = [-9.966532640424457, -7.824368824053634]
----------------------------------------
Best solution found after 50 generations: [-9.972361037663681, -5.769196750607994]
Best fitness value: 99.44798466551265
```

Laboratory Program - 2

Particle Swarm Optimization for Function Optimization

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

Algorithm



Manual
Date
Page 09.

(#) Pseudocode:

1. P = particle initialization();
2. For i = 1 to max
3. for each particle p in P do
   fP = f(P)
4. If fP is better than f(pbest)
   pbest = P;
5. end
6. end
7. gbest = best p in P.
8. for each particle p in P do.
9. $v_i^{t+1} = \underbrace{v_i^t}_{\text{inertia}} + \underbrace{c_1 u_1^t (pb_i^t - p_i^t)}_{\text{personal influence}} + \underbrace{c_2 u_2^t (gb^t - p_i^t)}_{\text{social influence}}$
10. $p_i^{t+1} = p_i^t + v_i^{t+1}$
11. end
12. end.

(#). Output:

Iteration 0, Global best value: 12.4801
" 10, " : 0.1506
" 20, " : 0.0011
" 30, " : 1.4744
" 40, " : 3.1113
" 50, " : 4.9150
" 60, " : 3.5974
" 70, " : 7.1054
" 80, " : 0.0
" 90, " : 0.0

Final global best position: [-4.77 -1.4692]
Final Global Best value: 0.0

0/1 sem
13/11/24

Code

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

# Objective function (Rastrigin function as an example)
def rastrigin(x):
    A = 10
    return A * len(x) + sum(x_i**2 - A * np.cos(2 * np.pi * x_i) for x_i in x)

# PSO Parameters
n_particles = 30     # Number of particles
n_dimensions = 2     # Number of dimensions (parameters to optimize)
n_iterations = 100   # Number of iterations

w = 0.5              # Inertia weight
c1 = 1.5             # Cognitive coefficient (particle's own best position)
c2 = 1.5             # Social coefficient (global best position)
v_max = 2.0          # Maximum velocity

# Initialize particles' positions and velocities
positions = np.random.uniform(-5.12, 5.12, (n_particles, n_dimensions))
velocities = np.random.uniform(-1, 1, (n_particles, n_dimensions))

# Initialize personal best positions and global best position
pbest_positions = positions.copy()
pbest_values = np.apply_along_axis(rastrigin, 1, pbest_positions)

gbest_position = pbest_positions[np.argmin(pbest_values)]
gbest_value = np.min(pbest_values)

# PSO Main Loop
for t in range(n_iterations):
    # Evaluate fitness
    fitness_values = np.apply_along_axis(rastrigin, 1, positions)

    # Update personal bests
    for i in range(n_particles):
        if fitness_values[i] < pbest_values[i]:
            pbest_positions[i] = positions[i]
            pbest_values[i] = fitness_values[i]

    # Update global best
    min_fitness_idx = np.argmin(pbest_values)
    if pbest_values[min_fitness_idx] < gbest_value:
        gbest_position = pbest_positions[min_fitness_idx]
        gbest_value = pbest_values[min_fitness_idx]
```

```
    # Update velocity and position for each particle
    r1 = np.random.rand(n_particles, n_dimensions)
    r2 = np.random.rand(n_particles, n_dimensions)

    velocities = (w * velocities +
                  c1 * r1 * (pbest_positions - positions) +
                  c2 * r2 * (gbest_position - positions))

    # Apply velocity limits (optional)
    velocities = np.clip(velocities, -v_max, v_max)

    # Update positions
    positions = positions + velocities

    # Print the current best solution only for multiples of 10
    if t % 10 == 0:
        print(f"Iteration {t}, Global Best Value: {gbest_value:.5f}")

# Final output
print(f"\nFinal Global Best Position: {gbest_position}")
print(f"Final Global Best Value: {gbest_value:.5f}")

# Plotting the optimization process (visualization for 2D case)
x_vals = np.linspace(-5.12, 5.12, 400)
y_vals = np.linspace(-5.12, 5.12, 400)
X, Y = np.meshgrid(x_vals, y_vals)
Z = rastrigin([X, Y])

plt.contour(X, Y, Z, levels=np.linspace(0, 500, 50), cmap='jet')
plt.scatter(gbest_position[0], gbest_position[1], color='red', label='Global Best')
plt.title("PSO Optimization (Rastrigin Function)")
plt.xlabel('x1')
plt.ylabel('x2')
plt.legend()
plt.show()
```

Output

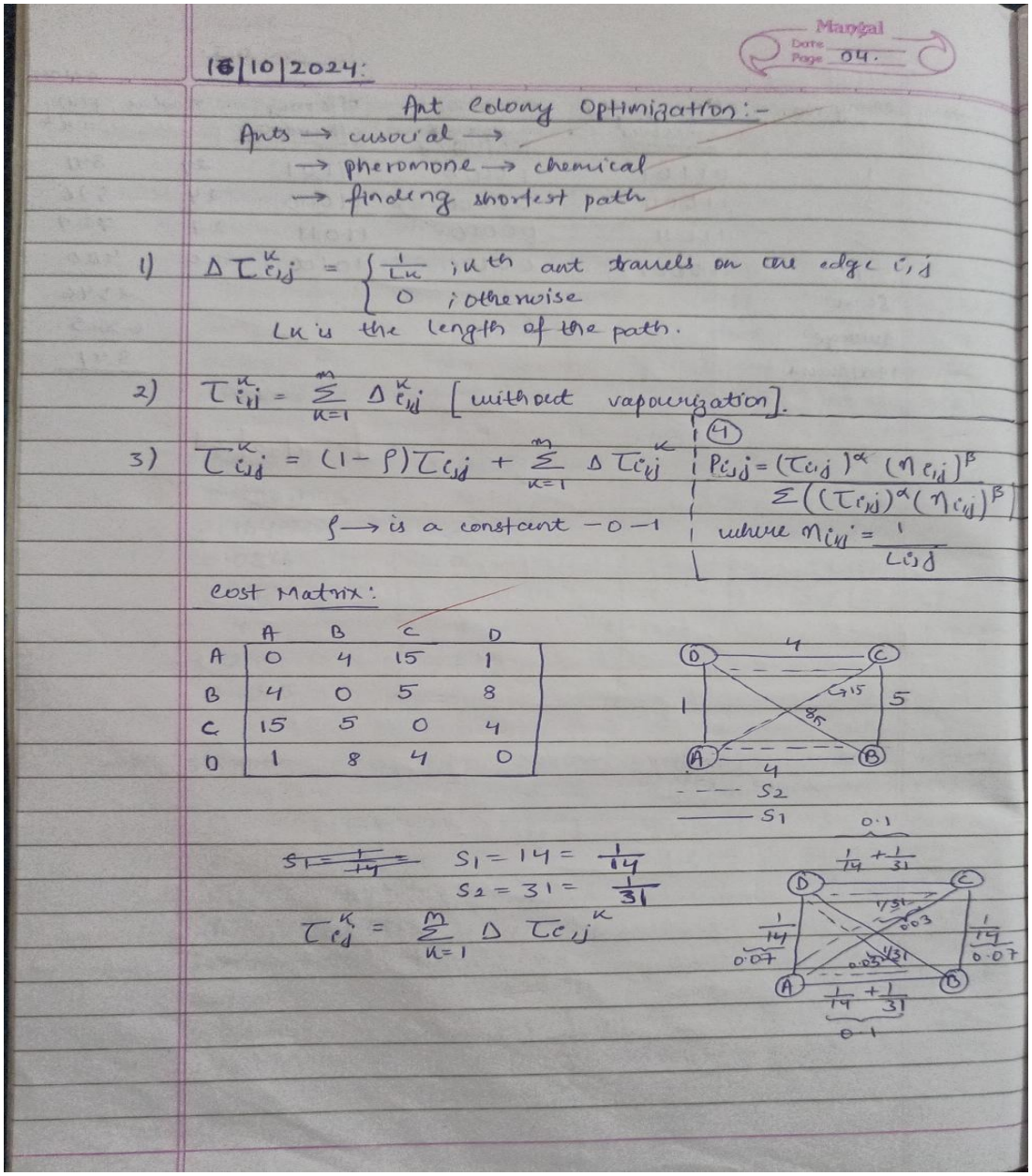
```
Iteration 0, Global Best Value: 7.29865
Iteration 10, Global Best Value: 0.01789
Iteration 20, Global Best Value: 0.00018
Iteration 30, Global Best Value: 0.00000
Iteration 40, Global Best Value: 0.00000
Iteration 50, Global Best Value: 0.00000
Iteration 60, Global Best Value: 0.00000
Iteration 70, Global Best Value: 0.00000
Iteration 80, Global Best Value: 0.00000
Iteration 90, Global Best Value: 0.00000

Final Global Best Position: [2.10172483e-09 6.30154843e-10]
Final Global Best Value: 0.00000
```

Laboratory Program - 3

Ant Colony Optimization for the Traveling Salesman Problem

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

Algorithm



Manipal
Date
Page 05

$\rho = 0.5$; let

→ From A to D:

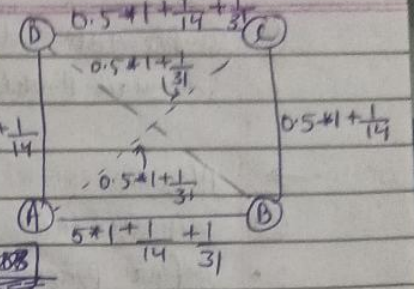
$$P_{ij} = \frac{1 * \frac{1}{1}}{\left(1*\frac{1}{1} + 1*\frac{1}{15} + 1*\frac{1}{4}\right)} = \boxed{0.75}$$

→ From A to B:

$$P = \frac{1 * \frac{1}{4}}{\left(1\times\frac{1}{1} + 1\times\frac{1}{15} + 1\times\frac{1}{4}\right)} = \boxed{0.25}$$

→ From A to C:

$$P = \frac{1 \times \frac{1}{15}}{\left(1\times\frac{1}{1} + 1\times\frac{1}{15} + 1\times\frac{1}{4}\right)} = \boxed{0.05}$$



Write Algorithm:

Before updating the pheromone:

$$P_1 = \frac{1*\frac{1}{1}}{1*\frac{1}{1} + 1*\frac{1}{15} + 1*\frac{1}{4}} = 0.15$$

$$P_2 = \frac{1*\frac{1}{4}}{1*\frac{1}{1} + 1*\frac{1}{15} + 1*\frac{1}{4}} = 0.18$$

$$P_3 = \frac{1*\frac{1}{15}}{1*\frac{1}{1} + 1*\frac{1}{15} + 1*\frac{1}{4}} = 0.05$$

After updating the pheromone:

$$P_1 = \frac{1*\frac{1}{1}}{\left(1*\frac{1}{1} + 1*\frac{1}{15} + 5*\frac{1}{4}\right)} = 0.43$$

$$P_2 = \frac{5*\frac{1}{4}}{\left(1*\frac{1}{1} + 1*\frac{1}{15} + 5*\frac{1}{4}\right)} = 0.53$$

$$P_3 = \frac{1*\frac{1}{15}}{1*\frac{1}{1} + 1*\frac{1}{15} + 5*\frac{1}{4}} = 0.02$$

Manjal
Date
Page 06.

23/10/24:

Q. Implement ant colony optimization for Travelling Salesman Problem:-

→ Algorithm:

1. Initialization:
   - Define parameters, Number of ants(K), Number of cities(n), T_0 (Initial pheromone level on all edges), P (Pheromone evaporation rate)$(0<P\leq 1)$, α (Influence of pheromone on path selection α (typically >0), β (Influence of heuristic information on path selection β (typically >0)).
   - Initialize pheromone levels $T_{ij} = T_0$ for all edges i,j

2. Main Loop:
   For each ant k:
   Randomly choose a starting city.
   Repeat until all cities are visited:
   Calculate the probability P_{ij} of moving from city i to city j using:

$$P_{ij} = \frac{T_{ij}^{\alpha} \cdot \eta_{ij}^{\beta}}{\sum_{l \in \text{allowed}} T_{il}^{\alpha} \cdot \eta_{il}^{\beta}}$$

   Choose next city j based on P_{ij} (can use a stochastic selection method, e.g. roulette wheel or greedy approach).
   Complete the tour by returning to the starting city.

Update pheromones:

For edge (i,j):
Evaporate the pheromone

$$T_{ij} \leftarrow (1-P) \cdot T_{ij}$$

Add pheromone based on the quality of solutions:
For each ant k, if it includes edge (i,j) in its tour with length L_k:

Mangal
Date
Page 07

$\tau_{ij} \leftarrow \tau_{ij} + Q/L_k$

Q is pheromone intensity.

Keep track of best solution found across all iterations
Repeat main loop until (stopping criteria met (no. of iterations, error).
Return the best tour.

A. cost matrix:

	A	B	C	D
A	0	4	15	1
B	4	0	5	8
C	15	5	0	4
D	1	8	4	0

C B A D C

Soln: Best Tour: [2, 1, 0, 3, 2]
Best length = 14.

23/10/21

Code

```
import numpy as np
np.random.seed(0)

# Problem Parameters
num_cities = 10
num_ants = 20
alpha = 1.0
beta = 5.0
rho = 0.5
initial_pheromone = 0.1
num_iterations = 100

# Generate Random Cities
cities = np.random.rand(num_cities, 2)

# Distance Matrix Calculation
def calculate_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):
            distance = np.linalg.norm(cities[i] - cities[j])
            distance_matrix[i][j] = distance
            distance_matrix[j][i] = distance
    return distance_matrix

distance_matrix = calculate_distance_matrix(cities)

# Initialize Pheromone Matrix
pheromone = np.ones((num_cities, num_cities)) * initial_pheromone

# City Selection
def select_next_city(probabilities):
    return np.random.choice(len(probabilities), p=probabilities)

# Probability Calculation
def calculate_probabilities(ant_path, pheromone, distance_matrix, alpha, beta):
    current_city = ant_path[-1]
    probabilities = np.zeros(len(distance_matrix))

    for city in range(len(distance_matrix)):
        if city not in ant_path:
            probabilities[city] = (pheromone[current_city][city] ** alpha) * ((1.0 /
distance_matrix[current_city][city]) ** beta)
    probabilities /= probabilities.sum()
    return probabilities
```

```
# Solution Construction
def construct_solution(pheromone, distance_matrix, alpha, beta):
    solution = []
    for _ in range(num_ants):
        ant_path = [np.random.randint(num_cities)]
        while len(ant_path) < num_cities:
            probabilities = calculate_probabilities(ant_path, pheromone, distance_matrix, alpha, beta)
            next_city = select_next_city(probabilities)
            ant_path.append(next_city)
        solution.append(ant_path + [ant_path[0]])
    return solution

# Pheromone Update
def update_pheromones(pheromone, solutions, distance_matrix, rho):
    pheromone *= (1 - rho)
    for solution in solutions:
        path_length = sum(distance_matrix[solution[i], solution[i+1]] for i in range(len(solution) - 1))
        pheromone_delta = 1.0 / path_length
        for i in range(len(solution) - 1):
            pheromone[solution[i]][solution[i + 1]] += pheromone_delta
            pheromone[solution[i + 1]][solution[i]] += pheromone_delta
    return pheromone

# Optimization Process
best_solution = None
best_path_length = float('inf')

for iteration in range(num_iterations):
    solutions = construct_solution(pheromone, distance_matrix, alpha, beta)
    pheromone = update_pheromones(pheromone, solutions, distance_matrix, rho)

    for solution in solutions:
        path_length = sum(distance_matrix[solution[i], solution[i + 1]] for i in range(len(solution) - 1))
        if path_length < best_path_length:
            best_path_length = path_length
            best_solution = solution

    # Display output only for multiples of 10
    if (iteration + 1) % 10 == 0:
        print(f"Iteration {iteration + 1}: Path length = {best_path_length:.5f}")

    # Check for convergence
    if iteration > 0 and best_path_length == previous_best_path_length:
        print(f"Convergence reached at iteration {iteration + 1}. Best solution found.")
        break
    previous_best_path_length = best_path_length
```

```
# Final Output
print("\nBest solution found:", best_solution)
print("Shortest path length:", best_path_length)
```

Output

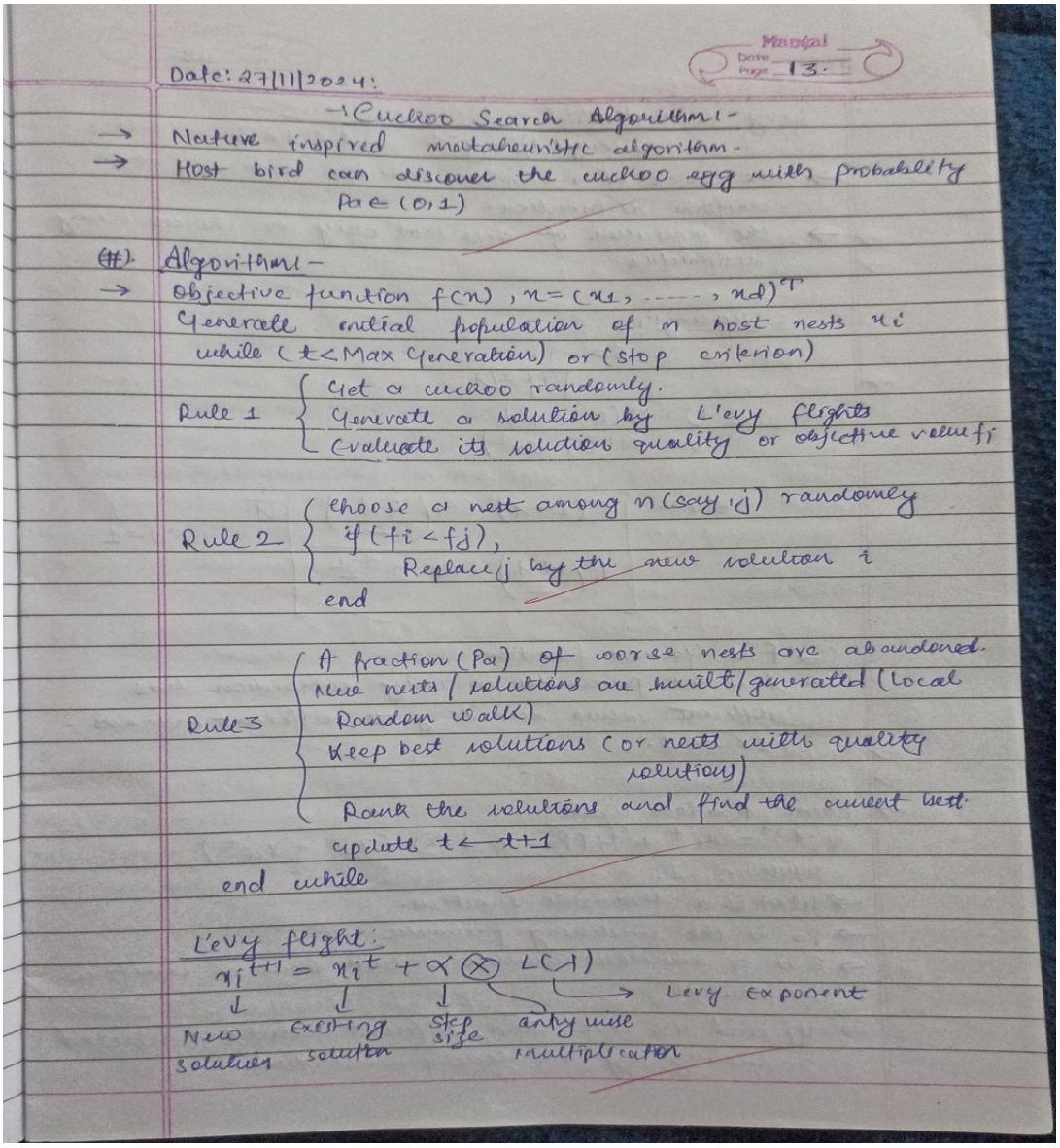
```
Convergence reached at iteration 4. Best solution found.

Best solution found: [2, 0, 1, 5, 4, 9, 6, 3, 8, 7, 2]
Shortest path length: 3.4388850126686448
```

Laboratory Program - 4

Cuckoo Search (CS)

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

Algorithm



Levy distribution:

$$\text{Levy}(\lambda) \approx g^{-\lambda} \quad ; 1 < \lambda \leq 3$$

→ the choice of a random direction according to uniform distribution

→ the generation of steps that obey the chosen Levy distribution.

Step length:

$$S = \frac{U}{|V|^{1/\lambda}}$$

$$U \approx N(0, \sigma_u^2)$$

$$V \approx N(0, \sigma_v^2)$$

$$\sigma_u^2 = \left[\frac{\Gamma(1+\lambda) \cdot \sin\left(\frac{\pi\lambda}{2}\right)}{\Gamma\left(\frac{1+\lambda}{2}\right) \times \lambda \times 2^{\frac{\lambda-1}{2}}} \right]^{\frac{1}{\lambda}} \quad ; \sigma_v^2 = 1$$

→ Γ is Gamma function.

→ λ is the parameter in Levy distribution has different values according to different instances. $1 < \lambda \leq 3$.

Local Random Walks

$$x_i^{t+1} = x_i^t + H(Pa - \varepsilon) \otimes (x_j^t - x_k^t)$$

where,

→ $H(u)$ is a Heaviside function.

→ Pa is the switching parameter.

→ ε is a random number drawn from a uniform distribution.

→ x_j and x_k are two different solutions selected randomly by random permutation.

##. Output:

Generation 1, Best Fitness: 5.0241
Generation 2, Best Fitness: 5.02412
⋮
Generation 42, Best Fitness: 2.3771
⋮
Generation 80, Best Fitness: 1.0821
⋮
Generation 100, Best fitness: 0.0955
Best solution: [-2.5848 -4.6116].
Best Fitness: 0.0955

Objective function used:

Rastrigin function:

$$f(x) = 10n + \sum_{i=1}^{n} [x_i^2 - 10\cos(2\pi x_i)].$$

n is the number of dimensions ($n=2$)
$x = [x_1, x_2]$ is input vector.

O/P seen
27/11/24

Code

```
import random
import math

# Objective Function
def objective_function(x):
    return sum(xi ** 2 for xi in x)

# Levy Flight Step
def levy_flight(Lambda, dim):
    sigma = (math.gamma(1 + Lambda) * math.sin(math.pi * Lambda / 2) /
             (math.gamma((1 + Lambda) / 2) * Lambda * 2 ** ((Lambda - 1) / 2))) ** (1 / Lambda)
    u = [random.gauss(0, sigma) for _ in range(dim)]
    v = [random.gauss(0, 1) for _ in range(dim)]
    step = [ui / abs(vi) ** (1 / Lambda) for ui, vi in zip(u, v)]
    return step

# Generate New Solution
def generate_new_solution(current_solution, alpha, Lambda, lower_bound, upper_bound):
    step = levy_flight(Lambda, len(current_solution))
    new_solution = [
        max(min(current_solution[i] + alpha * step[i], upper_bound), lower_bound)
        for i in range(len(current_solution))
    ]
    return new_solution

# Initialize Nests
def initialize_nests(n_nests, dim, lower_bound, upper_bound):
    return [[random.uniform(lower_bound, upper_bound) for _ in range(dim)] for _ in range(n_nests)]

# Cuckoo Search Algorithm
def cuckoo_search(objective, n_nests, max_iter, alpha, pa, lower_bound=-10, upper_bound=10,
Lambda=1.5):
    nests = initialize_nests(n_nests, 2, lower_bound, upper_bound)
    fitness = [objective(nest) for nest in nests]
    best_nest = min(nests, key=objective)
    best_fitness = objective(best_nest)

    for iteration in range(max_iter):
        random_index = random.randint(0, n_nests - 1)
```

```
        cuckoo_solution = generate_new_solution(nests[random_index], alpha, Lambda, lower_bound,
upper_bound)
        cuckoo_fitness = objective(cuckoo_solution)

        random_nest_index = random.randint(0, n_nests - 1)
        if cuckoo_fitness < fitness[random_nest_index]:
            nests[random_nest_index] = cuckoo_solution
            fitness[random_nest_index] = cuckoo_fitness

        num_to_abandon = int(pa * n_nests)
        worst_indices = sorted(range(n_nests), key=lambda i: fitness[i],
reverse=True)[:num_to_abandon]
        for idx in worst_indices:
            nests[idx] = [random.uniform(lower_bound, upper_bound) for _ in range(2)]
            fitness[idx] = objective(nests[idx])

        current_best_index = min(range(n_nests), key=lambda i: fitness[i])
        if fitness[current_best_index] < best_fitness:
            best_nest = nests[current_best_index]
            best_fitness = fitness[current_best_index]

        # Output every 100 iterations
        if iteration % 100 == 0 or iteration == max_iter - 1:
            print(f"Iteration {iteration}, Best Fitness: {best_fitness:.5f}, Best Solution: {best_nest}")

    return best_nest, best_fitness

# Main Function
if __name__ == "__main__":
    n_nests = 25    # Number of nests
    max_iter = 1000 # Maximum iterations
    alpha = 0.1     # Step size
    pa = 0.25       # Probability of abandoning worse nests

    best_solution, best_value = cuckoo_search(
        objective_function, n_nests=n_nests, max_iter=max_iter, alpha=alpha, pa=pa
    )

    print(f"\nBest solution found: x = {best_solution[0]:.5f}, y = {best_solution[1]:.5f}")
    print(f"Best objective function value: {best_value:.5f}")
```

Output

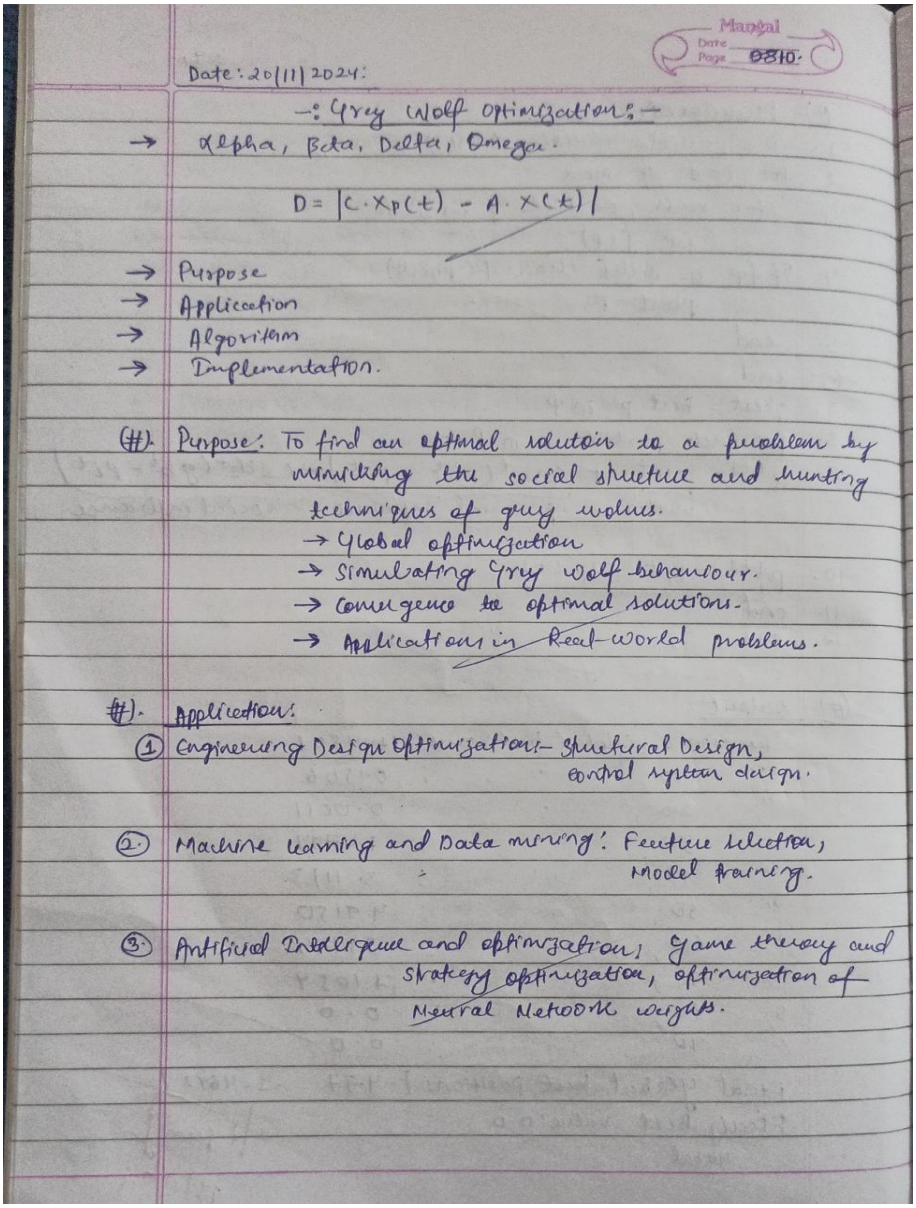
```
Iteration 0, Best Fitness: 0.22830, Best Solution: [-0.4664999684868931, 0.1033251221074103]
Iteration 100, Best Fitness: 0.05437, Best Solution: [-0.23267040694054952, 0.015338160420397762]
Iteration 200, Best Fitness: 0.00443, Best Solution: [0.06251808820158372, 0.022832902631498175]
Iteration 300, Best Fitness: 0.00010, Best Solution: [-0.009763824782933361, -0.0005021375169151215]
Iteration 400, Best Fitness: 0.00004, Best Solution: [-0.005905180653441653, -0.0016099886133596197]
Iteration 500, Best Fitness: 0.00004, Best Solution: [-0.005905180653441653, -0.0016099886133596197]
Iteration 600, Best Fitness: 0.00004, Best Solution: [-0.005905180653441653, -0.0016099886133596197]
Iteration 700, Best Fitness: 0.00004, Best Solution: [-0.005905180653441653, -0.0016099886133596197]
Iteration 800, Best Fitness: 0.00004, Best Solution: [-0.005905180653441653, -0.0016099886133596197]
Iteration 900, Best Fitness: 0.00004, Best Solution: [-0.005905180653441653, -0.0016099886133596197]
Iteration 999, Best Fitness: 0.00001, Best Solution: [0.003363734483260020 7, -0.0008645107976493899]

Best solution found: x = 0.00336, y = -0.00086
Best objective function value: 0.00001
```

Laboratory Program - 5

Grey Wolf Optimizer (GWO)

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

Algorithm



Manjal
Date
Page 0810

Date: 20/11/2024:

-: Grey Wolf Optimization :-

→ Alpha, Beta, Delta, Omega.

$$D = |C \cdot X_p(t) - A \cdot X(t)|$$

→ Purpose
→ Application
→ Algorithm
→ Implementation.

(#). Purpose: To find an optimal solution to a problem by mimicking the social structure and hunting techniques of grey wolves.
→ Global optimization
→ Simulating Grey Wolf behaviour.
→ Convergence to optimal solutions.
→ Applications in Real-world problems.

(#). Application:
(1) Engineering Design Optimization:- Structural Design, control system design.
(2.) Machine learning and Data mining: Feature selection, model training.
(3.) Artificial Intelligence and optimization; Game theory and strategy optimization, optimization of Neural Network weights.

(#). Algorithm:

1. Initialize the population:
   Population size = N
   Dimension D
   Each wolf X_i has a position vector in the D-dimensional space.

2. Set parameters:
   Max no of iterations: Tmax
   Population size: N
   Dimension of to problem: D
   convergence factor:

3. Evaluate the fitness:
   here,
   $$f(x) = \sum_{i=1}^{n} x_i^2.$$
   min at $x = 0$.

4. Sort wolves Based on fitness:
   X_α (Best fitness)
   X_β (Second Best)
   X_δ (Third best).

5. Update position of wolves:
   $$A = 2 \cdot a \cdot r_1 - a$$
   $$C = 2 \cdot r_2$$
   r_1, r_2 are random vectors $[0, 1]$.
   a is linearly decreasing parameter.
   $$a = 2 - 2 \cdot \frac{t}{T_{max}}.$$
   $t \rightarrow$ current iteration.

$X_1(t+1) = X_i(t) + A \cdot D_\alpha$.

→ A is a coefficient vector.

→ D_α is the distance of current wolf to target wolf.

$A = 2 \cdot rand() - 1$

$D_\alpha = |C \cdot X_\alpha(t) - X_i(t)|$

$C = 2 \cdot rand()$

$X_\alpha(t)$ is position of alpha wolf at time t.

$X_i(t)$ is the current position of wolf i.

#) Output:

Iteration 1/100, best score: 5.730170

Iteration 2/1000; Best score: 5.678430

⋮

Iteration 100/100, Best score: 0.0000112

Best position (solution): [0.00054732, 0.0002675]

Best score (objective value) = 0.000012

20/11/24

Code

```
import random

# Grey Wolf Optimizer Function
def grey_wolf_optimizer(objective_function, lower_bound, upper_bound, dim, num_wolves, max_iter):

    alpha_pos = [0] * dim
    beta_pos = [0] * dim
    delta_pos = [0] * dim

    alpha_score = float('inf')
    beta_score = float('inf')
    delta_score = float('inf')

    wolves = [[random.uniform(lower_bound, upper_bound) for _ in range(dim)] for _ in range(num_wolves)]

    for iteration in range(max_iter):
        for i in range(num_wolves):
            fitness = objective_function(wolves[i])

            if fitness < alpha_score:
                delta_score, delta_pos = beta_score, beta_pos[:]
                beta_score, beta_pos = alpha_score, alpha_pos[:]
                alpha_score, alpha_pos = fitness, wolves[i][:]
            elif fitness < beta_score:
                delta_score, delta_pos = beta_score, beta_pos[:]
                beta_score, beta_pos = fitness, wolves[i][:]
            elif fitness < delta_score:
                delta_score, delta_pos = fitness, wolves[i][:]

        a = 2 - iteration * (2 / max_iter)
        for i in range(num_wolves):
            for j in range(dim):
                r1 = random.random()
                r2 = random.random()
                A1 = a * (2 * r1 - 1)
                C1 = 2 * r2
                D_alpha = abs(C1 * alpha_pos[j] - wolves[i][j])
                X1 = alpha_pos[j] - A1 * D_alpha

                r1 = random.random()
                r2 = random.random()
                A2 = a * (2 * r1 - 1)
                C2 = 2 * r2
                D_beta = abs(C2 * beta_pos[j] - wolves[i][j])
```

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

            r1 = random.random()
            r2 = random.random()
            A3 = a * (2 * r1 - 1)
            C3 = 2 * r2
            D_delta = abs(C3 * delta_pos[j] - wolves[i][j])
            X3 = delta_pos[j] - A3 * D_delta

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

            if wolves[i][j] < lower_bound:
                wolves[i][j] = lower_bound
            elif wolves[i][j] > upper_bound:
                wolves[i][j] = upper_bound

        # Output every 10 iterations
        if iteration % 10 == 0 or iteration == max_iter - 1:
            print(f"Iteration {iteration}: Best Score = {alpha_score:.5f}, Best Position = {alpha_pos}")

    return alpha_pos, alpha_score

# Sphere Function (Objective Function)
def sphere_function(position):
    return sum(x ** 2 for x in position)

# Problem Parameters
lower_bound = -10
upper_bound = 10
dim = 3
num_wolves = 25
max_iter = 50

# Run Grey Wolf Optimizer
best_position, best_score = grey_wolf_optimizer(sphere_function, lower_bound, upper_bound, dim,
num_wolves, max_iter)

# Final Output
print("\nFinal Best Position:", best_position)
print("Final Best Score:", best_score)
```

Output

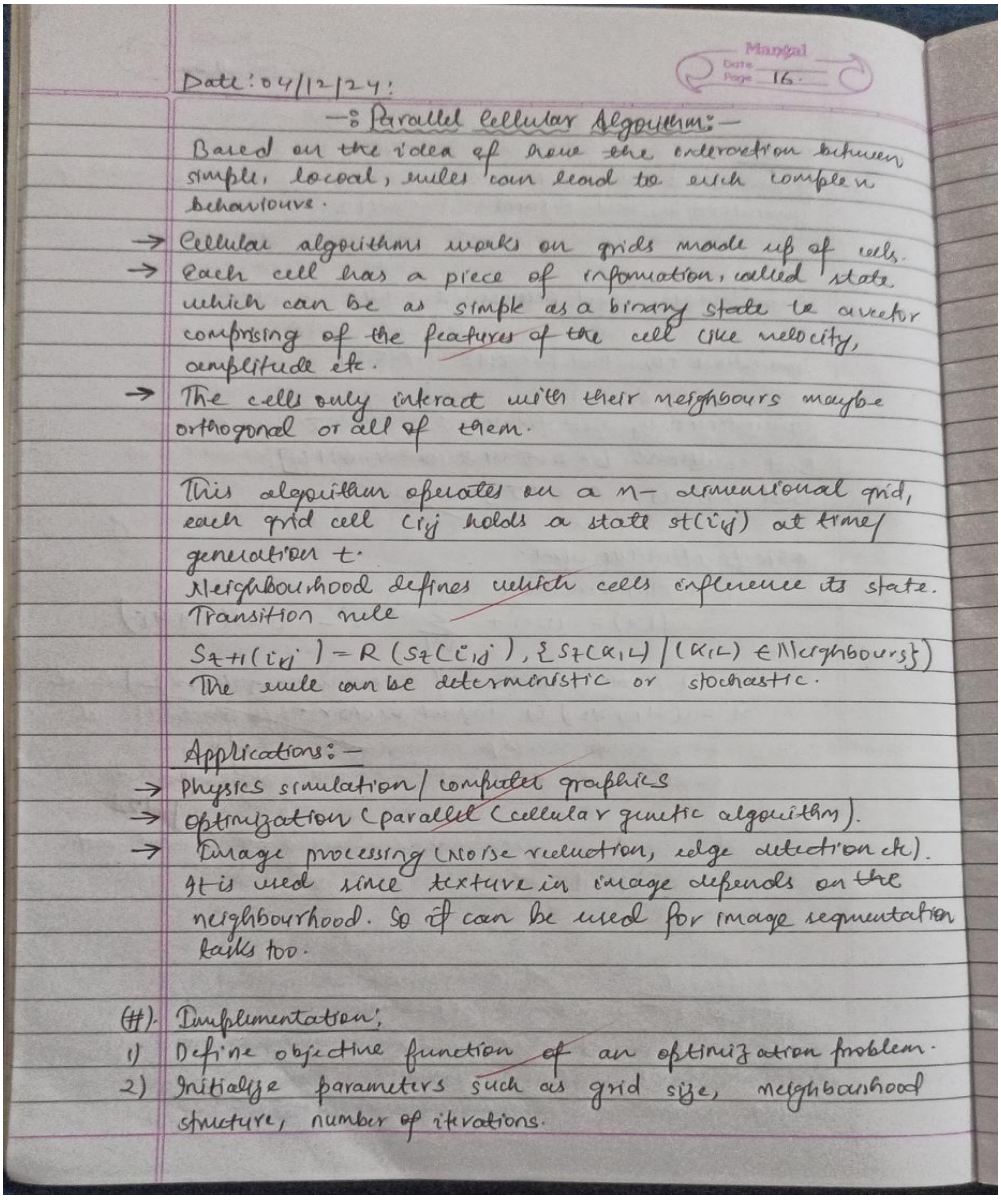
```
Iteration 0: Best Score = 8.87469, Best Position = [-1.3937875919991694, 2.6192794260861127, 0.2672459070460409]
Iteration 10: Best Score = 0.00006, Best Position = [-0.0022888418251507912, 0.005312156238702422, 0.0047323367161349284]
Iteration 20: Best Score = 0.00000, Best Position = [-5.509811050364098e-06, -4.388573147797282e-06, 3.946947280531768e-06]
Iteration 30: Best Score = 0.00000, Best Position = [-2.4489778136877548e-08, -8.658917449203874e-09, -1.4297757082884138e-08]
Iteration 40: Best Score = 0.00000, Best Position = [-4.301575426213075e-09, -4.106251702412561e-09, -4.374118169436466e-09]
Iteration 49: Best Score = 0.00000, Best Position = [-3.2476095209067505e-09, -2.7864689468237177e-09, -2.931715786065307e-09]

Final Best Position: [-3.2476095209067505e-09, -2.7864689468237177e-09, -2.931715786065307e-09]
Final Best Score: 2.690633424216157e-17
```

Laboratory Program - 6

Parallel Cellular Algorithms and Programs

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

Algorithm



Date: 04/12/24:

—: Parallel Cellular Algorithm:—

Based on the idea of how the interaction between simple, local, rules can lead to such complex behaviours.

→ Cellular algorithms works on grids made up of cells.
→ Each cell has a piece of information, called state. which can be as simple as a binary state to a vector comprising of the features of the cell like velocity, amplitude etc.
→ The cells only interact with their neighbours maybe orthogonal or all of them.

This algorithm operates on a n- dimensional grid, each grid cell (i,j) holds a state st(i,j) at time/ generation t.
Neighbourhood defines which cells influence its state.
Transition rule

$S_{t+1}(i,j) = R(S_t(i,j), \{S_t(k,l) \mid (k,l) \in \text{Neighbours}\})$

The rule can be deterministic or stochastic.

Applications:—

→ Physics simulation / computer graphics
→ Optimization (parallel cellular genetic algorithm).
→ Image processing (Noise reduction, edge detection etc). It is used since texture in image depends on the neighbourhood. So it can be used for image segmentation tasks too.

(#). Implementation:
1) Define objective function of an optimization problem.
2) Initialize parameters such as grid size, neighbourhood structure, number of iterations.

3) Initialize parameters such as grid size, neighbourhood structure, number of iterations.

3) Initialize population either randomly or if a start is known.

4) Evaluate fitness of each cell for optimization and maintain a global best.

5) Cells update based on transition rule (average, median etc.) can be for all cells simultaneously or each cell individually.

6) Repeat evaluation and updation for fixed number of iterations or convergence is met.

Since this algorithm considers only its neighbourhood and exits in a state grid, it has less exploration & more exploitation so it is best suited for problems that model state systems like growth, diffusion, physics simulations.

Ex: Conway's game of life, wave propagation etc.
Conway's game of life
Death if alive & overpopulated or lonely
Birth if population == 3
No change otherwise.

Output:-

Objective, minimize $f(x) = x^2 - 4x + 4$
Grid size: 10×10
Neighbourhood: 3×3
Number of iterations: 100
Initial state in range [-10, 10] for all cells.

Change in fitness

cu. solution	fitness
0.8005	1.4387
1.9853	0.00021
2.0020	4.3140e-06

Best solution: 2.002077

Best fitness: 4.3140e-06.

o/p seen 16/12/24

Code

```
import random

# Objective Function
def objective_function(x):
    return -(x ** 2) + 4 * x

# Initialize Parameters
def initialize_parameters():
    grid_size = 10  # Grid size
    num_iterations = 50  # Number of iterations
    lower_bound, upper_bound = -10, 10  # Bounds for the grid values
    return grid_size, num_iterations, lower_bound, upper_bound

# Initialize Population Grid
def initialize_population(grid_size, lower_bound, upper_bound):
    grid = [[random.uniform(lower_bound, upper_bound) for _ in range(grid_size)] for _ in
range(grid_size)]
    return grid

# Evaluate Fitness Grid
def evaluate_fitness(grid):
    fitness_grid = [[objective_function(cell) for cell in row] for row in grid]
    return fitness_grid

# Update Grid States Based on Neighbor Averages
def update_states(grid, fitness_grid):
    grid_size = len(grid)
    updated_grid = [[0] * grid_size for _ in range(grid_size)]

    for i in range(grid_size):
        for j in range(grid_size):
            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 < grid_size and 0 <= nj < grid_size:
                        neighbors.append(grid[ni][nj])

            if neighbors:
                updated_grid[i][j] = sum(neighbors) / len(neighbors)
            else:
                updated_grid[i][j] = grid[i][j]

    return updated_grid
```

```
# Print Grid State
def print_grid(grid, label="Grid"):
    print(f"{label}")
    for row in grid:
        print([f"{value: .4f}" for value in row])
    print()

# Parallel Cellular Algorithm
def parallel_cellular_algorithm():
    grid_size, num_iterations, lower_bound, upper_bound = initialize_parameters()
    grid = initialize_population(grid_size, lower_bound, upper_bound)

    print_grid(grid, label="Initial Grid")

    best_solution = None
    best_fitness = float('-inf')

    for iteration in range(num_iterations):
        fitness_grid = evaluate_fitness(grid)

        for i in range(grid_size):
            for j in range(grid_size):
                if fitness_grid[i][j] > best_fitness:
                    best_fitness = fitness_grid[i][j]
                    best_solution = grid[i][j]

        # Output progress at multiples of 10 iterations
        if iteration % 10 == 0 or iteration == num_iterations - 1:
            print(f"Iteration {iteration}: Best Solution = {best_solution:.4f}, Best Fitness =
{best_fitness:.4f}")

        grid = update_states(grid, fitness_grid)

    return best_solution, best_fitness

# Main Function
if __name__ == "__main__":
    best_solution, best_fitness = parallel_cellular_algorithm()
    print("\nFinal Results:")
    print(f"Best Solution: {best_solution:.4f}")
    print(f"Best Fitness: {best_fitness:.4f}")
```

Output

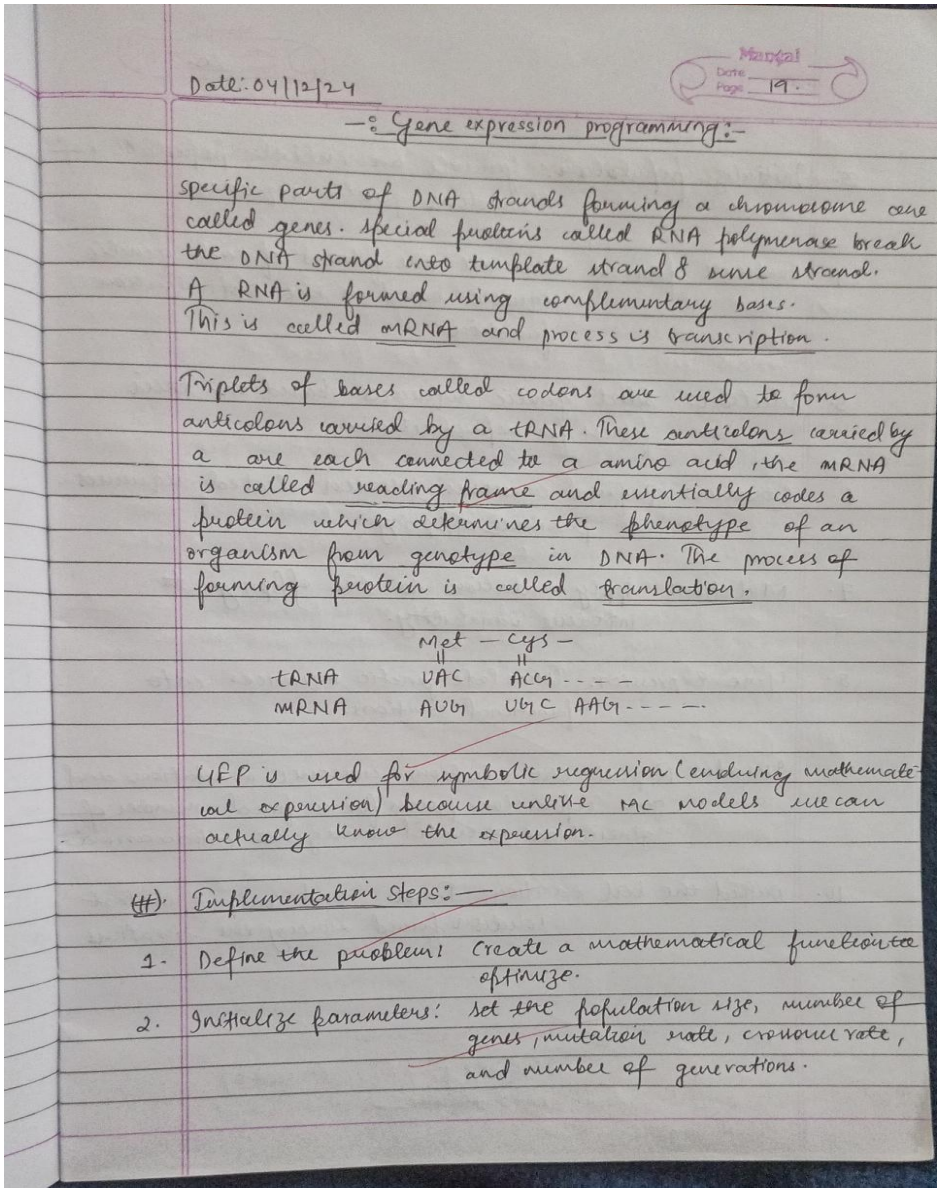
```
Initial Grid
[' 2.1002', '-8.5424', ' 0.5768', ' 1.0575', ' 8.1478', ' 5.6918', ' 0.0955', '-7.0900', ' 8.5333', '-5.1055']
['-8.9354', '-9.6884', '-5.2842', '-3.4578', '-4.9637', '-0.9836', '-3.5976', ' 8.3437', ' 6.9574', '-6.4682']
['-1.9309', '-7.2601', '-4.5101', ' 7.2017', '-8.3728', '-7.4899', '-8.2406', '-6.4359', '-0.4393', ' 1.8302']
[' 0.2574', ' 8.0121', '-4.9665', '-4.6416', ' 7.0306', ' 0.2532', ' 8.5815', ' 3.3289', ' 8.8667', '-5.1253']
[' 3.9054', '-6.9647', '-7.1257', '-5.9857', ' 8.9998', '-3.9828', ' 6.6955', '-7.3007', ' 9.8741', '-4.1903']
[' 5.8162', '-8.5069', ' 6.6198', ' 8.0581', '-7.6086', ' 7.3042', ' 8.2752', ' 1.8829', ' 3.0308', ' 3.5125']
['-1.9685', ' 7.2189', ' 7.0566', ' 5.4051', ' 0.6901', '-5.4429', '-6.6738', ' 9.6035', '-8.9977', ' 7.5862']
['-9.0879', '-0.7581', '-4.5743', ' 6.4059', '-6.0020', '-3.6441', ' 3.8070', '-7.6950', '-3.8934', '-3.1362']
['-3.2436', ' 4.7337', '-1.0044', '-9.3255', ' 1.6884', '-0.4363', ' 1.3384', ' 9.5544', ' 6.7410', '-2.7891']
[' 2.3002', '-6.1585', ' 3.3928', '-0.9514', '-3.0380', ' 2.2574', '-0.1634', ' 7.6562', '-8.3370', ' 9.5738']

Iteration 0: Best Solution = 2.1002, Best Fitness = 3.9900
Iteration 10: Best Solution = 1.9938, Best Fitness = 4.0000
Iteration 20: Best Solution = 1.9938, Best Fitness = 4.0000
Iteration 30: Best Solution = 1.9938, Best Fitness = 4.0000
Iteration 40: Best Solution = 1.9938, Best Fitness = 4.0000
Iteration 49: Best Solution = 1.9938, Best Fitness = 4.0000

Final Results:
Best Solution: 1.9938
Best Fitness: 4.0000
```

Laboratory Program - 7

Optimization via Gene Expression Algorithms

Gene Expression Algorithms (GEA) are inspired by the biological process of gene expression in living organisms. This process involves the translation of genetic information encoded in DNA into functional proteins. In GEA, solutions to optimizatsion 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



Date: 04/12/24

Page 19

-: Gene expression programming :-

Specific parts of DNA strands forming a chromosome are called genes. Special proteins called RNA polymerase break the DNA strand into template strand & sense strand. A RNA is formed using complementary bases. This is called mRNA and process is transcription.

Triplets of bases called codons are used to form anticolons carried by a tRNA. These anticolons carried by a are each connected to a amino acid, the mRNA is called reading frame and essentially codes a protein which determines the phenotype of an organism from genotype in DNA. The process of forming protein is called translation.

	Met	Cys
tRNA	UAC	ACG - - - -
MRNA	AUG	UGC AAG - - - -

GEP is used for symbolic regression (ensuring mathematical expression) because unlike ML models we can actually know the expression.

(#). Implementation steps:—

1. Define the problem: Create a mathematical function to optimize.
2. Initialize parameters: Set the population size, number of genes, mutation rate, crossover rate, and number of generations.

3. Initialize population: Generate an initial population of random genetic sequences.

4. Evaluate fitness: Evaluate the fitness of each genetic sequence based on the optimization function.

5. Selection: Select genetic sequences based on their fitness for reproduction.

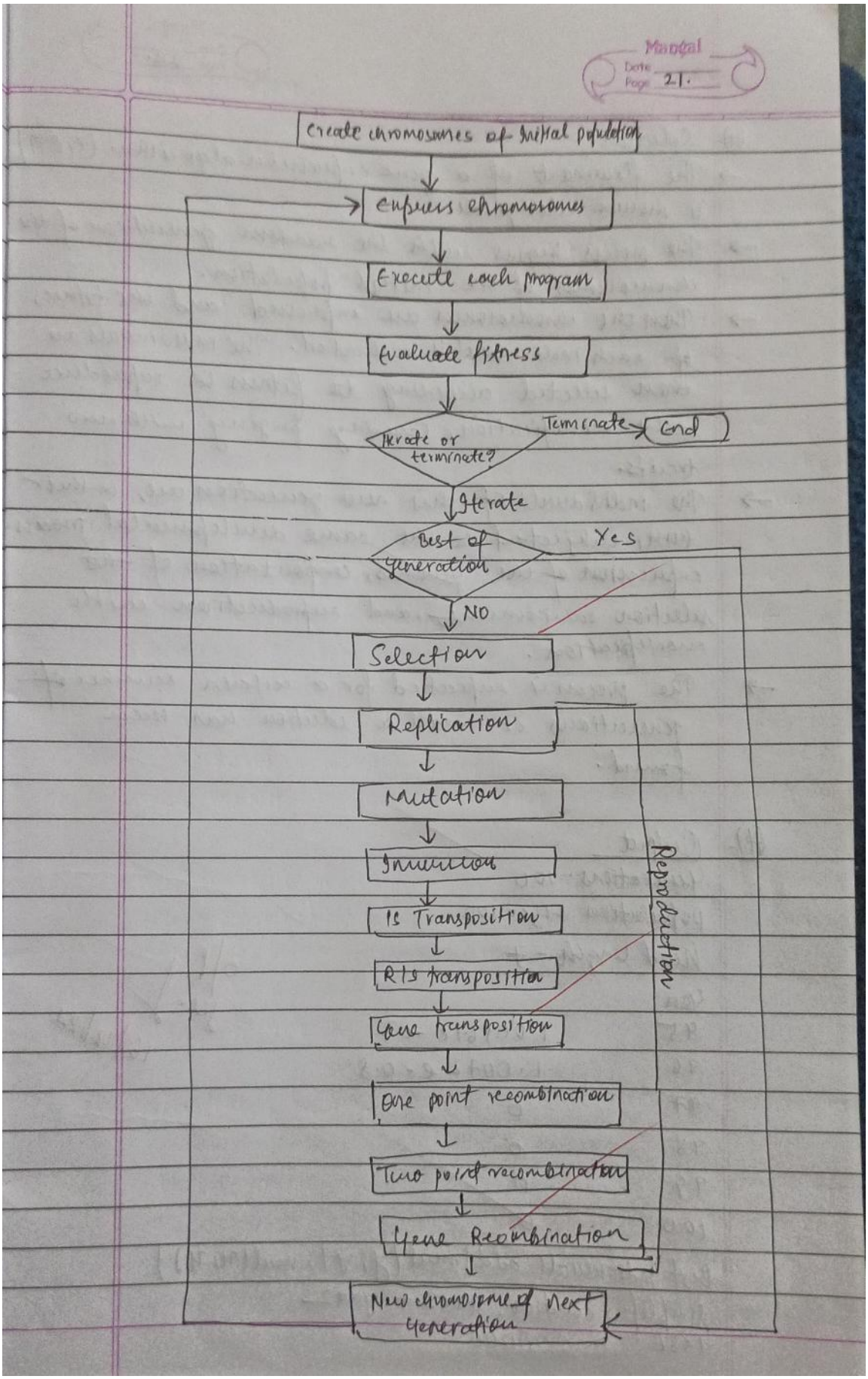
6. Crossover: Perform crossover between selected sequences to produce offspring.

7. Mutation: Apply mutation to the offspring to introduce variability.

8. Gene Expression: Translate genetic sequences into functional solutions.

9. Iterate: Repeat the selection, crossover, mutation and gene expression process for a fixed number of generations or until convergence criteria are met.

10. Output the best solution: Track and output the best solution found during the iterations


Manual
Date
Page 21.
Create chromosomes of initial population
Express chromosomes
Execute each program
Evaluate fitness
Iterate or terminate?
Terminate
End
Iterate
Best of generation
Yes
NO
Selection
Replication
Mutation
Inversion
Is Transposition
RIs transposition
Gene transposition
One point recombination
Two point recombination
Gene Recombination
New chromosome of next generation
Reproduction

#) Solution:

→ The flowchart of a gene expression algorithm (GEA) is shown alongside.

→ The process begins with the random generation of the chromosomes of the initial population.

→ Then the chromosomes are expressed and the fitness of each individual is evaluated. The individuals are then selected according to fitness to reproduce with modification, leaving progeny with new traits.

→ The individuals of this new generation are, in their turn, subjected to the same developmental process: expression of the genomes, confrontation of the selection environment, and reproduction with modification.

→ The process is repeated for a certain number of generations or until a solution has been found.

#) Output:

Generations = 100
population size = 100
head length = 7

Gen	
95	1.00567e-07
96	1.0072e-08
97	0
98	0
99	0
100	0

o/p seen 16/12/24

Best Individual: add(mul(y, y), mul(x, x))
Simplified solution: x**2 + y**2
MSE: 0.000000

Code

```
import operator
import numpy as np
import geppy as gep
from deap import creator, base, tools

# Step 1: Define the target function
def target_function(x, y):
    return x**2 + y**2

# Step 2: Define the dataset
x_data = np.linspace(-10, 10, 50)
y_data = np.linspace(-10, 10, 50)
X, Y = np.meshgrid(x_data, y_data)
Z = target_function(X, Y)  # Target outputs

# Flatten the data for evaluation
inputs = np.array([X.ravel(), Y.ravel()]).T
outputs = Z.ravel()

# Step 3: Define the GEP primitive set
pset = gep.PrimitiveSet('main', input_names=['x', 'y'])
pset.add_function(operator.add, 2)
pset.add_function(operator.mul, 2)
pset.add_constant_terminal(3)

# Step 4: Define the fitness and individual
if not hasattr(creator, "FitnessMax"):
    creator.create("FitnessMax", base.Fitness, weights=(1.0,))
if not hasattr(creator, "Individual"):
    creator.create('Individual', gep.Chromosome, fitness=creator.FitnessMax)

# Define head length and number of genes
h = 10  # Set head length to a suitable value
n_genes = 2  # Adjusted to ensure compatibility with the linker

# Step 5: Define the toolbox
toolbox = gep.Toolbox()

# Register chromosome, population, and compile function
toolbox.register('gene_gen', gep.Gene, pset=pset, head_length=h)
toolbox.register('individual', creator.Individual, gene_gen=toolbox.gene_gen, n_genes=n_genes,
linker=operator.add)
toolbox.register('population', tools.initRepeat, list, toolbox.individual)
toolbox.register('compile', gep.compile_, pset=pset)
```

```
# Define the fitness evaluation function
def evaluate(individual):
    func = toolbox.compile(individual)
    predictions = np.array([func(*input_pair) for input_pair in inputs])
    fitness = -np.mean((outputs - predictions)**2)  # Negative MSE
    return fitness,

toolbox.register('evaluate', evaluate)

# Register selection, mutation, and crossover operators
toolbox.register('select', tools.selRoulette)
toolbox.register('mut_uniform', gep.mutate_uniform, pset=pset, ind_pb=0.1)
toolbox.register('mut_invert', gep.invert, pb=0.1)
toolbox.register('mut_is_ts', gep.is_transpose, pb=0.1)
toolbox.register('mut_ris_ts', gep.ris_transpose, pb=0.1)
toolbox.register('mut_gene_ts', gep.gene_transpose, pb=0.1)
toolbox.register('cx_1p', gep.crossover_one_point, pb=0.4)
toolbox.register('cx_2p', gep.crossover_two_point, pb=0.2)
toolbox.register('cx_gene', gep.crossover_gene, pb=0.1)

# Explicitly set probabilities for the operators in Toolbox.pbs
toolbox.pbs['mut_uniform'] = 0.1  # Set the probability for mut_uniform

# Step 6: Define statistics and Hall of Fame
stats = tools.Statistics(key=lambda ind: ind.fitness.values[0])
stats.register("avg", np.mean)
stats.register("std", np.std)
stats.register("min", np.min)
stats.register("max", np.max)

hof = tools.HallOfFame(3)

# Step 7: Set population size and generations
n_pop = 100
n_gen = 5

pop = toolbox.population(n=n_pop)

print("Starting Genetic Programming Evolution...\n")

# Start evolution
pop, log = gep.gep_simple(pop, toolbox, n_generations=n_gen, n_elites=1,
                          stats=stats, hall_of_fame=hof, verbose=True)

# Step 8: Output the best individual
best_individual = hof[0]
simplified_solution = gep.simplify(best_individual)
```

```
print("\nBest Individual (Chromosome):")
print(best_individual)
print("\nSimplified Solution:")
print(simplified_solution)

# Evaluate the error of the solution
best_func = toolbox.compile(best_individual)
predictions = np.array([best_func(*input_pair) for input_pair in inputs])
mse = np.mean((outputs - predictions)**2)
print(f"\nMean Squared Error of the Best Solution: {mse:.6f}")

# Export the expression tree
rename_labels = {'add': '+', 'mul': '*'}
gep.export_expression_tree(best_individual, rename_labels, file='tree.png')
print("\nExpression tree exported to 'tree.png'.")
```

Output

```
Starting Genetic Programming Evolution...

gen     nevals  avg             std             min             max
0       100     -9.38893e+08    7.05021e+09     -6.96414e+10    -2512.38
1       75      -8326.55        22768.3         -153292         -1934.2
2       69      -7262.29        24797.9         -217036         -1621.95
3       64      -256400         1.53122e+06     -1.26277e+07    -0
4       78      -153100         1.3645e+06      -1.36496e+07    -0
5       76      -211820         1.56857e+06     -1.44685e+07    -0

Best Individual (Chromosome):
add(
        mul(x, x),
        mul(y, y)
)

Simplified Solution:
x**2 + y**2

Mean Squared Error of the Best Solution: 0.000000

Expression tree exported to 'tree.png'.
```