VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB RECORD

Bio Inspired Systems (23CS5BSBIS)

Submitted by

Suhan S (1BM23CS344)

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
Aug-2025 to Jan-2026

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

CERTIFICATE

This is to certify that the Lab work entitled " Bio Inspired Systems (23CS5BSBIS)" carried out by **Suhan S (1BM23CS344),** 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.

Rohith Vaidya K Assistant Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE

Index

Github Link:
https://github.com/suhan-s255/SUHAN_S_1BM23CS344_BIS_LAB.git

Program 1

Genetic Algorithm for Optimization Problems

Algorithm:

PAGE:

LAB - 1

DATE: PAGE:

Genetic Algorithm

5 Main Phases - Initialization
Fitness Assignment
Selection
Crossover
Termination

Steps:

1) Selecting encoding techiques
   0 to 31

2) Select the initial population - 4

String No.	Initial Population	x Value	Fitness $f(x) = x^2$	Prob $\frac{f(x)}{\Sigma f(x)}$	% prob	Expected Count $\frac{f(x)}{Avg(\Sigma f(x))}$	Actual Count (floor value)
1	01100	12	144	0.1247	12.47	0.49	1
2	11001	25	625	0.5411	54.11	2.164	2
3	00101	5	25	0.0216	2.16	0.086	0
4	10011	19	361	0.3125	31.25	1.25	1
Sum			1155				
Avg			288.75				
Max			625				

3) Select Mating Pool

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					
Avg					
Max					

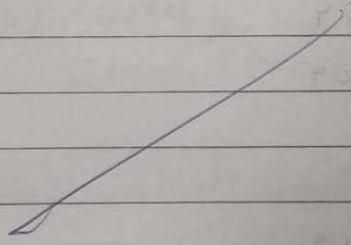
4) Crossover : Random 4 & 2

Max value - 729

~~5) Mutation~~

5) Mutation

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
Avg					630.5
Max					841



DATE: PAGE:

Code: Pseudo Code :-

```
import random
def fitness(x):
    return x**2

POPULATION_SIZE = 4
CHROMOSOME_LEN = 5
MUTATION_RATE = 0.1
GENERATIONS = 10

def binary_to_decimal(binary):
    return int(binary, 2)

def decimal_to_binary(n):
    return format(n, f'{CHROMOSE_LEN}')

def initialize_population():
    return [decimal_to_binary()]

def evaluate_population(population):
    return [fitness(binary_to_decimal(individual)) for
            individual in population]

def select_parents(population, fitness):
    parents = []
    for i in range(2):
        i,j = random.sample(range(len(population)), 2)
        if fitness[i] > fitness[j]:
            parents.append(population[i])
        else
            parents.append(population[j])
    return parents.
```

```
def crossover(parent1, parent2):
    point = random.randint(1, CHROMOSOME_LEN-1)
    child1 = parent1[:point] + parent2[point:]
    child2 = parent2[:point] + parent1[point:]
    return child1, child2

def mutate(individual):
    mutated = ""
    for bit in individual:
        if random.random() < MUTATION_RATE:
            mutated += '1' if bit == '0' else '0'
        else:
            mutated += bit
    return mutated

genetic algorithm()

final_x = express_gene(best_solution)
print(best_solution, final_x)
```

Select Mating pool
Mutation after Mating
Iterate
Note down the best value

Output

Gen 1 : x = 993 f(x) = 986049
Gen 2 : x = 1004 f(x) = 1008016
Gen 3 : x = 1022 f(x) = 1044484
Gen 4 : x = 1023 f(x) = 1046529
⋮
Gen 50 : x = 1023 f(x) = 1046529

Code:

```
import random

# Parameters
POP_SIZE = 1000
GENES = 5        # 5 bits to represent 0-31
GENERATIONS = 50
CROSSOVER_RATE = 0.7
MUTATION_RATE = 0.1

# Fitness function: f(x) = x^2
def fitness(binary_str):
    x = int(binary_str, 2)
    return x * x

# Create initial population of random 5-bit binary strings
def create_population():
    population = []
    for _ in range(POP_SIZE):
        individual = ''.join(random.choice('01') for _ in range(GENES))
        population.append(individual)
    return population

# Selection: Tournament Selection of size 2
def tournament_selection(pop):
    i1, i2 = random.sample(pop, 2)
    return i1 if fitness(i1) > fitness(i2) else i2

# Crossover: Single-point crossover
def crossover(parent1, parent2):
    if random.random() < CROSSOVER_RATE:
        point = random.randint(1, GENES - 1)
        child1 = parent1[:point] + parent2[point:]
        child2 = parent2[:point] + parent1[point:]
        return child1, child2
    else:
        return parent1, parent2

# Mutation: Bit flip mutation
def mutate(individual):
    new_ind = ''
    for bit in individual:
        if random.random() < MUTATION_RATE:
            new_ind += '1' if bit == '0' else '0'
        else:
            new_ind += bit
    return new_ind

# Main GA function
```

```
def genetic_algorithm():
    population = create_population()
    best_individual = None
    best_fitness = -1

    for gen in range(1, GENERATIONS + 1):
        new_population = []

        # Evaluate and keep track of best
        for ind in population:
            ind_fit = fitness(ind)
            if ind_fit > best_fitness:
                best_fitness = ind_fit
                best_individual = ind

        # Print best in current generation
        print(f"Generation {gen}: Best Individual = {best_individual} (x={int(best_individual, 2)}),
Fitness = {best_fitness}")

        # Create new generation
        while len(new_population) < POP_SIZE:
            parent1 = tournament_selection(population)
            parent2 = tournament_selection(population)
            child1, child2 = crossover(parent1, parent2)
            child1 = mutate(child1)
            child2 = mutate(child2)
            new_population.extend([child1, child2])

        population = new_population[:POP_SIZE]

    print(f"\nBest solution found: {best_individual} (x={int(best_individual, 2)}), Fitness =
{best_fitness}")

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

Program 2

Particle Swarm Optimization for Function Optimization

Algorithm:

Iteration 1

Part No.	Position (x,y)	Velocity (v_x, v_y)	pbest (x,y)	gbest (x,y)	Fitness values
P1	(1,1)	(0,0)	(1,1)	(1,1)	2
P2	(-1,1)	(0,0)	(-1,1)	(1,1)	2
P3	(0.5, -0.5)	(0,0)	(0.5, 0.5)	(1,1)	0.5
P4	(-1,-1)	(0,0)	(-1,-1)	(1,1)	2
P5	(0.25, 0.25)	(0,0)	(0.25, 0.25)	(1,1)	0.125

Best Fitness Value = 0.125 (P5)

So, gbest = (0.25, 0.25)

Iteration 2

Part No	Pos (x,y)	Velocity (v_x, v_y)	pbest (x,y)	gbest (x,y)	Fitness values
P1	(1,1)	(-0.75, -0.75)	1,1	(0.25, 0.25)	2
P2	(-1,1)	(1.25, -0.25)	-1,1	(0.25, 0.25)	2
P3	(0.5, 0.5)	(-0.5, 1.25)	0.5, -0.5	(0.25, 0.25)	0.5
P4	(-1,-1)	(-0.75, 0.75)	-1,-1	(0.25, 0.25)	2
P5	(0.25, 0.25)	(0,0)	0.25, 0.25	(0.25, 0.25)	0.125

gbest remains (0.25, 0.25)

Iteration 3

Part No	Pos (x,y)	Velocity (Vx, Vy)	pbest (x,y)	gbest (x,y)	Fitness values
P1	1,1	-0.75, -0.75	1,1	0.25, 0.25	2
P2	-1,1	1.25, -0.25	-1,1	0.25, 0.25	2
P3	0.5, -0.5	-0.5, 1.25	0.5, -0.5	0.25, 0.25	0.5
P4	-1,-1	-0.75, 0.75	-1,-1	0.25, 0.25	2
P5	0.25, 0.25	0,0	0.25, 0.25	0.25, 0.25	0.125

Best Position: (0.25, 0.25)

Best Fitness: 0.125

DATE: PAGE:

Particle Swarm Optimisation (PSO)

Pseudocode:

(1) p = particle.initialization()

(2) for i=1 to max:
    for each particle p in P do:
        $f_p = f(p)$

    if it is better than f pbest
        pbest = p
    end if
    end for
gbest = bestp in P
for each particle p in P do:

$$v_i^{t+1} = v_i^t + c_1 u_1^t (pb_i^t - p_i^t) + c_2 u_2^t (gb - p_i^t)$$

$$p_i^{t+1} = p_i^t + v_i^{t+1}$$

end for
end for

eg:- Iteration 1
F(x, y) = $x^2 + y^2$
Inertia weight (w) = 1
Cognitive constant (C_1) = 2
Social Constant (C_2) = 2
Initial solution set to (1000

Code:

```
import random

# Objective (fitness) function: De Jong function
def fitness_function(position):
    x, y = position
    return x*2 + y*2  # minimize this function

# PSO parameters
num_particles = 10
num_iterations = 50
W = 0.3       # inertia weight (from PDF)
C1 = 2        # cognitive coefficient
C2 = 2        # social coefficient

# Initialize particles and velocities
particles = [[random.uniform(-10, 10), random.uniform(-10, 10)] for _ in range(num_particles)]
velocities = [[0.0, 0.0] for _ in range(num_particles)]

# Initialize personal bests
pbest_positions = [p[:] for p in particles]
pbest_values = [fitness_function(p) for p in particles]

# Initialize global best
gbest_index = pbest_values.index(min(pbest_values))
gbest_position = pbest_positions[gbest_index][:]
gbest_value = pbest_values[gbest_index]

# PSO main loop
for iteration in range(num_iterations):
    for i in range(num_particles):
        r1, r2 = random.random(), random.random()

        # Update velocity
        velocities[i][0] = (W * velocities[i][0] +
                            C1 * r1 * (pbest_positions[i][0] - particles[i][0]) +
                            C2 * r2 * (gbest_position[0] - particles[i][0]))
        velocities[i][1] = (W * velocities[i][1] +
                            C1 * r1 * (pbest_positions[i][1] - particles[i][1]) +
                            C2 * r2 * (gbest_position[1] - particles[i][1]))

        # Update position
        particles[i][0] += velocities[i][0]
        particles[i][1] += velocities[i][1]

        # Evaluate fitness
        current_value = fitness_function(particles[i])

        # Update personal best
```

```
        if current_value < pbest_values[i]:
            pbest_positions[i] = particles[i][:]
            pbest_values[i] = current_value

            # Update global best
            if current_value < gbest_value:
                gbest_value = current_value
                gbest_position = particles[i][:]

    print(f"Iteration {iteration+1}/{num_iterations} | Best Value: {gbest_value:.6f} at
{gbest_position}")

print("\n☑ Optimal Solution Found:")
print(f"Best Position: {gbest_position}")
print(f"Minimum Value: {gbest_value}")
```

Program 3

Ant Colony Optimization for the Traveling Salesman Problem

Algorithm:

DATE: PAGE:

Complete tour by returning to starting city
Calculate total tour length
store tour and its length

Pheromone evaporation :
For all edges (i,j):
$T_{ij} = (1 - \text{evaporation\_rate}) * T_{ij}$

Pheromone update :
For each ant:
For each edge (i,j) in its tour:
$T_{ij} += Q / \text{tour\_length}$

Return the best-so-far tour as the optimal solution found

OUTPUT

It 1/100, best length : 22.3510
It 2/100, best length : 22.3510
It 3/100, best length : 22.3510
⋮

Best tour : [1, 0, 2, 4, 3, 1]
Best tour length : 22.3510

Input Matrix :

∞	2	4	5	7
2	∞	4	8	2
2	4	∞	1	3
5	8	1	∞	2
7	2	3	2	∞

∞

DATE: PAGE:

Ant Colony Optimization for Traveling Salesman Problem

Initialize:
- pheromone[i][j] for all edges (i,j) to a small constant to
- parameters: alpha (α), beta (β), evaporation_rate, Q (pheromone deposit constant)
- cities = list of coordinates or nodes

Repeat for max_iterations:

For each ant:
- Randomly choose a starting city
- Initialize tour with starting city
- Mark starting city as visited

While not all cities visited:
For each unvisited city j:
- Compute η_{ij} = 1/distance between current city i and city j
- compute probability to move to city j:

Use wheel selection to choose next city based on P_{ij}
Move to selected city
Add city to tour
Mark city as visited

Code:

```
import random
import math

def distance(city1, city2):
    return math.sqrt((city1[0] - city2[0])**2 + (city1[1] - city2[1])**2)

def initialize_pheromone(num_cities, tau0):
    return [[tau0 for _ in range(num_cities)] for _ in range(num_cities)]

def roulette_wheel_selection(probabilities):
    r = random.random()
    cumulative = 0.0
    for i, p in enumerate(probabilities):
        cumulative += p
        if r <= cumulative:
            return i
    return len(probabilities) - 1  # in case of rounding errors

def ant_colony_optimization(cities, alpha=1.0, beta=5.0, evaporation_rate=0.5, Q=100,
max_iterations=100, num_ants=10):
    num_cities = len(cities)
    pheromone = initialize_pheromone(num_cities, tau0=0.1)

    best_tour = None
    best_length = float('inf')

    for iteration in range(max_iterations):
        all_tours = []
        all_lengths = []

        for _ in range(num_ants):
            start_city = random.randint(0, num_cities - 1)
            visited = {start_city}
            tour = [start_city]
            current_city = start_city

            while len(visited) < num_cities:
                probabilities = []
                unvisited_cities = [city for city in range(num_cities) if city not in visited]

                denominator = 0
                for j in unvisited_cities:
                    dist = distance(cities[current_city], cities[j])
                    eta = 1.0 / dist if dist > 0 else 1e10  # avoid div by zero
                    denominator += (pheromone[current_city][j] ** alpha) * (eta ** beta)

                for j in unvisited_cities:
                    dist = distance(cities[current_city], cities[j])
```

```
                eta = 1.0 / dist if dist > 0 else 1e10
                numerator = (pheromone[current_city][j] ** alpha) * (eta ** beta)
                probabilities.append(numerator / denominator)

            next_city_index = roulette_wheel_selection(probabilities)
            next_city = unvisited_cities[next_city_index]
            tour.append(next_city)
            visited.add(next_city)
            current_city = next_city

        # Complete the tour by returning to start city
        tour.append(start_city)

        # Calculate total tour length
        length = 0
        for i in range(len(tour) - 1):
            length += distance(cities[tour[i]], cities[tour[i + 1]])

        all_tours.append(tour)
        all_lengths.append(length)

        if length < best_length:
            best_length = length
            best_tour = tour

    # Pheromone evaporation
    for i in range(num_cities):
        for j in range(num_cities):
            pheromone[i][j] *= (1 - evaporation_rate)

    # Pheromone update
    for tour, length in zip(all_tours, all_lengths):
        deposit = Q / length
        for i in range(len(tour) - 1):
            a, b = tour[i], tour[i + 1]
            pheromone[a][b] += deposit
            pheromone[b][a] += deposit  # if symmetric

    print(f"Iteration {iteration+1}/{max_iterations}, best length: {best_length:.4f}")

return best_tour, best_length

# Example usage:
if __name__ == "__main__":
    # Define some cities (x, y)
    cities = [(0, 0), (1, 5), (5, 2), (6, 6), (8, 3)]
    best_tour, best_length = ant_colony_optimization(cities)
    print("Best tour:", best_tour)
    print("Best tour length:", best_length
```

Program 4

Cuckoo Search (CS)

Algorithm:

DATE: PAGE:

Cuckoo Search Algorithm

BEGIN
Initialize N nests (random item selections)
For each nest:
Repair if overweight; compute fitness (total val)

REPEAT until MaxGen():
FOR each cuckoo():
Generate new solution by levy flight.
Repair if overweight: compute fitness
If better than random nest → replace it

Abandon Pa Fraction of worst nests
generate new random nests & repeat if needed
Keep best nest as current best.
END REPEAT

OUTPUT best solution and its total value
END

Input
values = [60, 100, 120]
weights = [10, 20, 30]
Capacity = 50

Output
Best Solution: [0, 1, 1]
Total Value: 220
Total Weight: 50

Code:

```
# --- Cuckoo Search Algorithm for 0/1 Knapsack Problem ---
import numpy as np
import math  # ☑ FIX: use the standard math library for gamma, sin, pi

# Problem Definition
values = np.array([60, 100, 120, 90, 30])     # value of items
weights = np.array([10, 20, 30, 25, 5])       # weight of items
capacity = 50                                 # maximum weight allowed
num_items = len(values)

# Fitness Function
def fitness(solution):
    total_weight = np.sum(solution * weights)
    total_value = np.sum(solution * values)
    if total_weight > capacity:
        return 0  # invalid solution
    return total_value

# Levy Flight function
def levy_flight(Lambda):
    # Mantegna's algorithm for Levy distribution
    sigma = (math.gamma(1 + Lambda) * math.sin(math.pi * Lambda / 2) /
             (math.gamma((1 + Lambda) / 2) * Lambda * 2 ** ((Lambda - 1) / 2))) ** (1 / Lambda)
    u = np.random.randn() * sigma
    v = np.random.randn()
    step = u / (abs(v) ** (1 / Lambda))
    return step

# --- Cuckoo Search Parameters ---
num_nests = 10
max_iter = 100
Pa = 0.25  # Discovery rate (probability a host bird discovers cuckoo's egg)
Lambda = 1.5  # Levy flight parameter

# Initialize nests (each nest = potential solution)
nests = np.random.randint(0, 2, (num_nests, num_items))
fitness_values = np.array([fitness(n) for n in nests])

# --- Main Loop ---
for t in range(max_iter):
    # Generate new solutions via Levy flight
    for i in range(num_nests):
        new_nest = nests[i].copy()
        step_size = levy_flight(Lambda)

        # Randomly flip bits depending on step size
        for j in range(num_items):
            if np.random.rand() < abs(step_size) % 1:  # probabilistic mutation
```

```
                new_nest[j] = 1 - new_nest[j]  # flip 0/1

        # Evaluate new solution
        new_fitness = fitness(new_nest)
        if new_fitness > fitness_values[i]:
            nests[i] = new_nest
            fitness_values[i] = new_fitness

    # Abandon a fraction of the worst nests (discovery)
    num_abandon = int(Pa * num_nests)
    worst_indices = np.argsort(fitness_values)[:num_abandon]
    for idx in worst_indices:
        nests[idx] = np.random.randint(0, 2, num_items)
        fitness_values[idx] = fitness(nests[idx])

    # Track best nest
    best_idx = np.argmax(fitness_values)
    best_nest = nests[best_idx]
    best_fit = fitness_values[best_idx]

    if t % 10 == 0:
        print(f"Iteration {t}: Best fitness = {best_fit}")

# --- Output Final Result ---
print("\nBest Solution Found:")
print("Items selected:", best_nest)
print("Total Value:", np.sum(best_nest * values))
print("Total Weight:", np.sum(best_nest * weights))
```

Program 5

Grey Wolf Optimizer (GWO)

Algorithm:

DATE: PAGE:

Grey Wolf Optimizer (GWO)

Pseudocode:

```
BEGIN
  Initialize number of wolves (N) and tasks
  Generate random schedules for all wolves
  Evaluate fitness (e.g., total time or cost)

  Identify α (best), β (second best), & δ (third best) wolves

  WHILE (not reached max iterations)
      FOR each wolf
          Update position using α, β, δ (follow the leaders)
          Repair schedule if invalid
          Recalculate fitness
      END FOR

      Update α, β, δ wolves
  END WHILE

  OUTPUT best schedule (α wolf)
END
```

Input

tasks = [3, 5, 2, 7, 4]

Output

Best Task Schedule: [2, 3, 4, 5, 7]

Total Completion Time: 21

Sen Roa 17/10

Code:

```
# --- Grey Wolf Optimizer for Task Scheduling ---
import numpy as np

# ----------------------------
# Problem Definition
# ----------------------------
num_tasks = 10
num_machines = 3

# Random task execution times (in arbitrary units)
task_times = np.random.randint(5, 20, num_tasks)

# Random machine speeds (higher = faster)
machine_speeds = np.random.uniform(0.8, 1.5, num_machines)

def calculate_makespan(schedule):
    """Calculate the total completion time for a schedule."""
    machine_loads = np.zeros(num_machines)
    for i, machine in enumerate(schedule):
        exec_time = task_times[i] / machine_speeds[machine]
        machine_loads[machine] += exec_time
    return np.max(machine_loads)  # Makespan = max machine load

# Fitness function: we minimize makespan, so return 1 / makespan
def fitness_function(schedule):
    return 1 / (1 + calculate_makespan(schedule))

# ----------------------------
# GWO Parameters
# ----------------------------
num_wolves = 15
max_iter = 100

# Each wolf = one possible task assignment [0..num_machines-1]
wolves = np.random.randint(0, num_machines, (num_wolves, num_tasks))
fitness = np.zeros(num_wolves)

# Initialize Alpha, Beta, Delta
alpha, beta, delta = np.zeros(num_tasks, dtype=int), np.zeros(num_tasks, dtype=int),
np.zeros(num_tasks, dtype=int)
alpha_score, beta_score, delta_score = -np.inf, -np.inf, -np.inf

# ----------------------------
# Main GWO Loop
# ----------------------------
for iteration in range(max_iter):
    for i in range(num_wolves):
        fitness[i] = fitness_function(wolves[i])
```

```
        # Update Alpha, Beta, Delta wolves
        if fitness[i] > alpha_score:
            delta_score, delta = beta_score, beta.copy()
            beta_score, beta = alpha_score, alpha.copy()
            alpha_score, alpha = fitness[i], wolves[i].copy()
        elif fitness[i] > beta_score:
            delta_score, delta = beta_score, beta.copy()
            beta_score, beta = fitness[i], wolves[i].copy()
        elif fitness[i] > delta_score:
            delta_score, delta = fitness[i], wolves[i].copy()

    a = 2 - iteration * (2 / max_iter)  # Decrease linearly from 2 to 0

    for i in range(num_wolves):
        for j in range(num_tasks):
            r1, r2 = np.random.rand(), np.random.rand()

            A1 = 2 * a * r1 - a
            C1 = 2 * r2
            D_alpha = abs(C1 * alpha[j] - wolves[i][j])
            X1 = alpha[j] - A1 * D_alpha

            r1, r2 = np.random.rand(), np.random.rand()
            A2 = 2 * a * r1 - a
            C2 = 2 * r2
            D_beta = abs(C2 * beta[j] - wolves[i][j])
            X2 = beta[j] - A2 * D_beta

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

            new_position = (X1 + X2 + X3) / 3

            # Discretize (assign to nearest machine)
            wolves[i][j] = int(np.clip(round(new_position), 0, num_machines - 1))

    if iteration % 10 == 0:
        print(f"Iteration {iteration}: Best makespan = {calculate_makespan(alpha):.3f}")

# ----------------------------
# Results
# ----------------------------
best_makespan = calculate_makespan(alpha)
print("\n☑ Best Schedule Found:")
for i, machine in enumerate(alpha):
```

```
    print(f"Task {i+1} → Machine {machine+1}")

print(f"\nTotal Makespan: {best_makespan:.3f}")
print(f"Machine Speeds: {np.round(machine_speeds, 2)}")
print(f"Task Times: {task_times}")
```

Program 6
Parallel Cellular Algorithms and Programs

Algorithm:

DATE: PAGE:

```
        if f(xi) < f(best_cell)
            best_cell = xi
        end if
    end for
end for
print("Best solution:", best_cell)
print("Best fitness:", f(best_cell))
```

output :-

Best solution found : [2.23000242e-05 3.00922293e-05]
Best fitness value : 2.78311034662692 7e-07

Parallel cellular Algorithm

Pseudocode:

Initialization:
Let $f(x)$
Let grid size
Let neighbourhood
Let max_iter
Let bounds
Let α, β

for $i \leftarrow 1$ to num_cells:
    initialize x_i randomly within the search
    evaluate fitness $f(x_i)$
end for

best_cell = cell with the lowest fitness

for $t \leftarrow 1$ to max_iter:
    for each cell i in parallel:
        N_i = Let of neighbouring cells of cell i
        x_i best neighbours = neighbours in N_i with best fitness

        r = random number in $[0,1]$
        x_i-new = $x_i + r \times (x\text{-best neighbour} - x_i)$
        f-new = $f(x_i\text{-new})$
        if f-new $< f(x_i)$
            $x_i = x_i$-new
        end if

Code:

```
import numpy as np
from multiprocessing import Pool

# --- Step 1: Define the problem (Objective function) ---
def objective_function(x):
    # Example: minimize f(x) = x^2 - 4x + 4
    return x**2 - 4*x + 4

# --- Step 2: Initialize parameters ---
GRID_SIZE = (10, 10)   # 10x10 grid = 100 cells
ITERATIONS = 100
VALUE_RANGE = (-10, 10)
NEIGHBORHOOD_RADIUS = 1  # 3x3 neighborhood

# --- Step 3: Initialize population (random values for cells) ---
grid = np.random.uniform(VALUE_RANGE[0], VALUE_RANGE[1], GRID_SIZE)

# --- Step 4: Fitness evaluation function ---
def evaluate_fitness(grid):
    return objective_function(grid)

# --- Step 5: Get neighborhood indices with wrapping ---
def get_neighborhood_indices(i, j, grid_shape, radius):
    rows, cols = grid_shape
    neighbors = []
    for dx in range(-radius, radius + 1):
        for dy in range(-radius, radius + 1):
            if dx == 0 and dy == 0:
                continue
            ni, nj = (i + dx) % rows, (j + dy) % cols
            neighbors.append((ni, nj))
    return neighbors

# --- Step 6: Update rule for each cell ---
def update_cell(args):
    i, j, grid = args
    neighbors = get_neighborhood_indices(i, j, grid.shape, NEIGHBORHOOD_RADIUS)
    # Find best neighbor (lowest fitness)
    best_neighbor = min(neighbors, key=lambda n: objective_function(grid[n]))
    # Example rule: average current cell and best neighbor values
    new_value = (grid[i, j] + grid[best_neighbor]) / 2
    return i, j, new_value

# --- Step 7: Parallel iteration ---
def run_parallel_cellular_algorithm():
    global grid
    best_value = None
    best_fitness = float("inf")
```

```
    for iteration in range(ITERATIONS):
        # Evaluate fitness of all cells
        fitness_grid = evaluate_fitness(grid)

        # Track best cell
        current_best_idx = np.unravel_index(np.argmin(fitness_grid), grid.shape)
        current_best_value = grid[current_best_idx]
        current_best_fitness = fitness_grid[current_best_idx]

        if current_best_fitness < best_fitness:
            best_value = current_best_value
            best_fitness = current_best_fitness

        # Parallel update of grid
        with Pool() as pool:
            updates = pool.map(update_cell, [(i, j, grid) for i in range(grid.shape[0]) for j in
range(grid.shape[1])])

        # Apply updates
        new_grid = np.copy(grid)
        for i, j, val in updates:
            new_grid[i, j] = val
        grid = new_grid

        # Print progress
        print(f"Iteration {iteration+1}: Best Fitness = {best_fitness:.6f}, Best Value = {best_value:.6f}")

    print("\n=== Final Result ===")
    print(f"Best Value Found: {best_value}")
    print(f"Best Fitness: {best_fitness:.6f}")

# --- Run the algorithm ---
if __name__ == "__main__":
    run_parallel_cellular_algorithm()
```

Program 7

Optimization via Gene Expression Algorithms

Algorithm:

LAB - 7

DATE: PAGE:

GENE EXPRESSION ALGORITHM

Step1: Fitness function: $f(x) = x^2$

Encoding technique : 0 to 31

use chromosome of fixed length (genotype)

Step2: Initial Population

S.no.	(Genotype) Initial chrosome	Phenotype (expression)	value	fitness	P
1	+xx	x^2	12	144	0.1247
2	+xx	2x	25	625	0.5411
3	x	x	5	25	0.0216
4	-x2	x-2	19	361	0.3125
Sum				1155	
avg				288.75	
max				625	

actual count	expected count
1	0.5
2	2.1
0	0.08
1	1.25

DATE: PAGE:

Step 3: Selection of mating pool

S.no.	Selected chromosome	Crossover point	Offspring	Phenotype
1	+xx	2	xx	x*(x+...)
2	+xx	1	+xx	2x
3	+xx	3	+x-	x+(x-...)
4	+x2	1	+x2	x+2

x value	fitness
13	169
24	576
27	729
17	289

Step 4: crossover: perform crossover randomly chosen gene position (not raw bits)

max fitness after crossover = 729

Step 5 mutation

S.no.	Offspring before mutation	mutation applied	Offspring after mutation	Phenotype
1	*x+	+ → -	*x-	x*(x-...)
2	+xx	None	+xx	2x
3	+x-	- → +	-x+	x+x*x
4	+x2	None	+x2	x+2

x value	fitness
29	841
24	576
27	729
20	400

Step6: Gene expression and evaluation

Decode each genotype → phenotype
calculate fitness

$\Sigma f(x) = 841 + 576 + 729 + 400 = 2546$

avg = 636.5
max = 841

Step7: Iterate until convergence

Repeat step 3 to 6 until fitness improvement is negligible or generator limit has reached.

Code: Pseudocode:-

```
POP_SIZE = 100
GENES = 5
GENERATIONS = 50
CROSSOVER_RATE = 0.7
MUTATION_RATE = 0.1

def fitness(x):
    return xxx

def create_population():
    population = []
    for i in range(POP_SIZE):
        population.append(genes)
    return population

def express_gene(gene_sequence): return
    return [fitness(express_gene(ind)) ]
def tournament_selection(population, fitness_value):
    i1, i2 = random.sample(range(len(population)), 2)
    return population

def crossover(p1, p2):
    if random.random() < CROSSOVER_RATE:
        point = random.randint(1, GENES-1)
        return child1, child2
    else
        return p1, p2
```

```
def mutate(individual):
    mutated = ''
    for bit in individual:
        if random.random() < MUTATION_RATE:
            mutated += '1'
        else:
            mutate += bit
    return mutated

def gen_expression_algorithm()

final_x = express_gene(best_solution)
print(best_solution, final_x)
```

OUTPUT:-

G1 : Best = 11111 (x = 31), Fitness = 961
G2 : Best = 11111 (x = 31), Fitness = 961
G3 : Best = 11111 (x = 31), Fitness = 961
:
G50 : Best = 11111 (x = 31), Fitness = 961

Best solution found : 11111 (x = 31), Fitness = 961

Code:

```
import random

# Step 2: Initialize Parameters
POP_SIZE = 1000            # Number of individuals in the population
GENES = 5              # Number of genes per chromosome (5 bits = 0 to 31)
GENERATIONS = 50        # Number of generations to run
CROSSOVER_RATE = 0.7     # Probability of crossover
MUTATION_RATE = 0.1      # Probability of mutation per gene

# Step 1: Define the Problem
# Optimization function: f(x) = x^2 (maximize)
def fitness(x):
    return x * x

# Step 3: Initialize Population (each gene is a bit string of length GENES)
def create_population():
    population = []
    for _ in range(POP_SIZE):
        genes = ''.join(random.choice('01') for _ in range(GENES))
        population.append(genes)
    return population

# Step 8: Gene Expression - Decode binary gene to integer x
def express_gene(gene_sequence):
    return int(gene_sequence, 2)

# Step 4: Evaluate Fitness
def evaluate_population(population):
    return [fitness(express_gene(ind)) for ind in population]

# Step 5: Selection - Tournament Selection
def tournament_selection(population, fitness_values):
    i1, i2 = random.sample(range(len(population)), 2)
    return population[i1] if fitness_values[i1] > fitness_values[i2] else population[i2]

# Step 6: Crossover - Single-point crossover
def crossover(parent1, parent2):
    if random.random() < CROSSOVER_RATE:
        point = random.randint(1, GENES - 1)
        child1 = parent1[:point] + parent2[point:]
        child2 = parent2[:point] + parent1[point:]
        return child1, child2
    else:
        return parent1, parent2

# Step 7: Mutation - Bit flip mutation
def mutate(individual):
    mutated = ''
```

```
    for bit in individual:
        if random.random() < MUTATION_RATE:
            mutated += '1' if bit == '0' else '0'
        else:
            mutated += bit
    return mutated

# Step 9: Main Loop
def gene_expression_algorithm():
    population = create_population()
    best_solution = None
    best_fitness = -1

    for gen in range(1, GENERATIONS + 1):
        fitness_values = evaluate_population(population)

        # Track best individual
        for i in range(len(population)):
            if fitness_values[i] > best_fitness:
                best_fitness = fitness_values[i]
                best_solution = population[i]

        # Print best of this generation
        x_val = express_gene(best_solution)
        print(f"Generation {gen}: Best = {best_solution} (x = {x_val}), Fitness = {best_fitness}")

        # Generate new population
        new_population = []
        while len(new_population) < POP_SIZE:
            parent1 = tournament_selection(population, fitness_values)
            parent2 = tournament_selection(population, fitness_values)
            child1, child2 = crossover(parent1, parent2)
            child1 = mutate(child1)
            child2 = mutate(child2)
            new_population.extend([child1, child2])

        population = new_population[:POP_SIZE]

    # Step 10: Output Best Solution
    final_x = express_gene(best_solution)
    print(f"\nBest solution found: {best_solution} (x = {final_x}), Fitness = {best_fitness}")

# Entry Point
if __name__ == "__main__":
    gene_expression_algorithm()
```