VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB RECORD

Bio Inspired Systems (23CS5BSBIS)

Submitted by

Agam Tiwari (1BM22CS023)

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 **Agam Tiwari (1BM22CS023),** 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.

Dr. Shashikala 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	16/10/24	Genetic Algorithm for Optimization Problems	4-6
2	23/10/24	Particle Swarm Optimization for Function Optimization	7-9
3	30/10/24	Ant Colony Optimization for the Traveling Salesman Problem	10-12
4	20/11/24	Cuckoo Search (CS)	13-15
5	27/11/24	Grey Wolf Optimizer (GWO)	16-17
6	27/11/24	Parallel Cellular Algorithms and Programs	18-20
7	03/11/24	Optimization via Gene Expression Algorithms	21-24

Github Link:

https://github.com/Agam1611/BIS-LAB-5Sem

Program 1
Genetic Algorithm for Optimization Problems

Algorithm:

9/10/24

Bafna Gold
Date: Page:

1) Genetic Algorithms

(i) Components, Structure & Terminology

→ Genetic algorithms simulate biological evolution using simplified components and terminology. The key components of these algorithms are:

- A fitness function for optimization
- A population of chromosomes (candidate solutions)
- Selection of chromosomes for reproduction
- Crossover to generate the next generation
- Mutation to introduce random changes

→ The fitness function measures how well each solution (chromosome) solves a given problem, and this is central to the algorithms effectiveness. Chromosomes are arrays of parameter values representing potential solutions.

→ The Algorithm starts with a randomly generated population and repeatedly selects, crosses over and mutates chromosomes to create new generations, aiming to optimize the fitness function.

→ The crossover operator swaps parts of chromosomes while the mutation operator randomly alter bits. These operations are repeated over several generations until the best solution stabilizes, indicating convergence. The effectiveness of the algorithm depends on factors like crossover and mutation probabilities, population size, and no of iterations.

2) Preliminary Examples

→ This section presents three examples of genetic algorithms in order of increasing complexity.

2.1) Example: maximizing a Function

The goal is to maximize the function
$f(x) = -\frac{x^2}{10} + 3x$ for x values between 0 and 31

→ Chromosomes are represented as 5 bit binary integers.
→ A population of 10 chromosomes is randomly generated, and fitness is evaluated
→ Chromosomes are selected for reproduction based on fitness using weighted probability
→ After one generation, both average and maximum fitness increase, showing genetic algorithms effectiveness in evolving better solutions.

2.2) Example: Maximizing the number of 1's in a String

→ This example uses a genetic algorithm to maximize the number of 1's in a bit string of length 20

→ The fitness function is the sum of 1's in each chromosome.
→ With a population of 100 chromosomes, a mutation rate of 0.001 and 200 iterations, three runs show varying results due to the probabilistic nature of the algorithm

Code:

```
import numpy as np

# Objective function to maximize
def fitness_function(x):
    return x**2

# Initialize parameters
population_size = 50
mutation_rate = 0.1
crossover_rate = 0.7
num_generations = 50
lower_bound = -10
upper_bound = 10

# Create initial population
def initialize_population(size, lower, upper):
    return np.random.uniform(lower, upper, size)

# Evaluate fitness for the population
def evaluate_fitness(population):
    return np.array([fitness_function(x) for x in population])

# Selection using roulette wheel selection
def select_parents(population, fitness):
    total_fitness = np.sum(fitness)
    selection_probs = fitness / total_fitness
    parents_indices = np.random.choice(len(population), size=2,
p=selection_probs)
    return population[parents_indices]

# Crossover to create offspring
def crossover(parent1, parent2):
    if np.random.rand() < crossover_rate:
        return (parent1 + parent2) / 2  # Linear crossover
    return parent1

# Mutation to introduce diversity
def mutate(offspring):
    if np.random.rand() < mutation_rate:
        return np.random.uniform(lower_bound, upper_bound)
    return offspring

# Genetic Algorithm main function
```

```python
def genetic_algorithm():
    # Initialize population
    population = initialize_population(population_size, lower_bound,
upper_bound)

    for generation in range(num_generations):
        # Evaluate fitness of the population
        fitness = evaluate_fitness(population)

        # Track the best solution
        best_fitness_idx = np.argmax(fitness)
        best_solution = population[best_fitness_idx]
        best_fitness_value = fitness[best_fitness_idx]

        print(f"Generation {generation}: Best Solution = {best_solution},
Fitness = {best_fitness_value}")

        # Create the next generation
        new_population = []
        for _ in range(population_size):
            parent1, parent2 = select_parents(population, fitness)
            offspring = crossover(parent1, parent2)
            offspring = mutate(offspring)
            new_population.append(offspring)

        population = np.array(new_population)

    # Final evaluation
    final_fitness = evaluate_fitness(population)
    best_fitness_idx = np.argmax(final_fitness)
    best_solution = population[best_fitness_idx]
    best_fitness_value = final_fitness[best_fitness_idx]

    return best_solution, best_fitness_value

# Run the genetic algorithm
best_solution, best_fitness_value = genetic_algorithm()
print(f"Best Solution Found: x = {best_solution}, f(x) =
{best_fitness_value}")

Output :
```

```
Generation 0: Best Solution = -9.967365011554792, Fitness = 99.34836527356666
Generation 1: Best Solution = -9.169251894044368, Fitness = 84.07518029643623
Generation 49: Best Solution = 9.123059138454053, Fitness = 83.23020804373002
Best Solution Found: x = 9.05670095588789, f(x) = 82.02383220438064
```

Program 2

Particle Swarm Optimization for Function Optimization

Algorithm:

23/10/24 Particle Swarm Optimization

```
import numpy as np
# Step 1 : Fitness Function

def fitness_function(position):
        return np.sum(position ** 2)

# Step 2 : Initialize parameters

def initialize_parameters():
    params = {
        'N' : 50,
        'dim' : 2,
        'max_iter' : 200,
        'minx' : -10,
        'maxx' : 10,
        'w' : 0.5,          # Inertia weight
        'c1' : 1.5,         # Cognitive coefficient
        'c2' : 1.5          # Social coefficient
    }
    return params

# Step 3 : Initialize Particles

class Particle:
    def __init__(self, position, velocity):
        self.position = position
        self.velocity = velocity
        self.bestPos = position.copy()
        self.bestFitness = float('inf')
```

```
def initialize_swarm(N, dim, minx, maxx):
    swarm = []
    for _ in range(N):
        position = np.random.uniform(minx, maxx, dim)
        velocity = np.random.uniform(-1, 1, dim)
        swarm.append(Particle(position, velocity))
    return swarm

# Step 4 : Evaluate Fitness
def evaluate_fitness(swarm):
    for particle in swarm:
        particle.fitness = fitness_function(particle.
                                            position)

# Step 5 : Update velocities and positions
def update_particles(swarm, best_pos_swarm, w, c1, c2,
                     minx, maxx):
    for particle in swarm:
        r1, r2 = np.random.rand(), np.random.rand()
        particle.velocity = (w * particle.velocity +
                             r1 * c1 * (particle.bestPos -
                                        particle.position)
                             + r2 * c2 * (best_pos_swarm
                                        - particle.position))

        particle.position += particle.velocity

        particle.position = np.clip(particle.position,
                                    minx, maxx)
```

Code:

```
import numpy as np

# Step 1: Define the Problem
def fitness_function(position):
    # Example: Minimize the Sphere function
    return np.sum(position**2)

# Step 2: Initialize Parameters
def initialize_parameters():
    params = {
        'N': 50,         # Number of particles
        'dim': 2,       # Dimensionality of the problem
        'max_iter': 200,   # Maximum number of iterations
        'minx': -10,     # Minimum bound for position
        'maxx': 10,      # Maximum bound for position
        'w': 0.5,        # Inertia weight
        'c1': 1.5,       # Cognitive coefficient
        'c2': 1.5        # Social coefficient
    }
    return params

# Step 3: Initialize Particles
class Particle:
    def __init__(self, position, velocity):
        self.position = position
        self.velocity = velocity
        self.bestPos = position.copy()
        self.bestFitness = float('inf')

def initialize_swarm(N, dim, minx, maxx):
    swarm = []
    for _ in range(N):
        position = np.random.uniform(minx, maxx, dim)
        velocity = np.random.uniform(-1, 1, dim)
        swarm.append(Particle(position, velocity))
    return swarm

# Step 4: Evaluate Fitness
def evaluate_fitness(swarm):
    for particle in swarm:
        particle.fitness = fitness_function(particle.position)

# Step 5: Update Velocities and Positions
def update_particles(swarm, best_pos_swarm, w, c1, c2, minx, maxx):
```

```
    for particle in swarm:
        r1, r2 = np.random.rand(), np.random.rand()
        particle.velocity = (w * particle.velocity +
                             r1 * c1 * (particle.bestPos - particle.position)
                             r2 * c2 * (best_pos_swarm - particle.position))
        particle.position += particle.velocity
        # Clip position to be within bounds
        particle.position = np.clip(particle.position, minx, maxx)

# Step 6: Iterate
def pso():
    params = initialize_parameters()
    swarm = initialize_swarm(params['N'], params['dim'], params['minx'],
params['maxx'])
    best_pos_swarm = swarm[0].position.copy()
    best_fitness_swarm = float('inf')

    for _ in range(params['max_iter']):
        evaluate_fitness(swarm)

        for particle in swarm:
            if particle.fitness < particle.bestFitness:
                particle.bestFitness = particle.fitness
                particle.bestPos = particle.position.copy()
            if particle.fitness < best_fitness_swarm:
                best_fitness_swarm = particle.fitness
                best_pos_swarm = particle.position.copy()

        update_particles(swarm, best_pos_swarm, params['w'], params['c1'],
params['c2'], params['minx'], params['maxx'])

    # Step 7: Output the Best Solution
    return best_pos_swarm, best_fitness_swarm

best_position, best_fitness = pso()

print("Best Position:", best_position)
print("Best Fitness:", best_fitness)

Output :

Best Position: [-9.19971249e-25  1.71937901e-24]
Best Fitness: 3.802611270068504e-48
```

Program 3

Ant Colony Optimization for the Traveling Salesman Problem

Algorithm:

30/10/24 Ant Colony Optimization for the TSP

```
import numpy as np
import random

class AntColony:
    def __init__(self, cities, num_ants=10, alpha=1.0,
                 beta=2.0, rho=0.5, iterations=100):

        self.cities = cities
        self.num_ants = num_ants
        self.alpha = alpha
        self.beta = beta
        self.rho = rho
        self.iterations = iterations
        self.num_cities = len(cities)
        self.pheromone = np.ones((self.num_cities,
                                  self.num_cities))
        self.distance = self.calculate_distance()

    def calculate_distance(self):
        distances = np.zeros((self.num_cities,
                              self.num_cities))
        for i in range(self.num_cities):
            for j in range(i+1, self.num_cities):
                distances[i][j] = distances[j][i]

        return distances
```



```
def select_next_city(self, current_city, visited):
    probabilities = []
    for next_city in range(self.num_cities):
        if next_city not in visited:
            pheromone = self.pheromone
            [current_city][next_city] ** self.alpha

            heuristic = (1 / self.distance
                         [current_city]
                         [next_city]) ** self.beta
            probabilities.append(pheromone *
                                 heuristic)
        else:
            probabilities.append(0)

    total = sum(probabilities)
    probabilities = [p / total for p in
                     probabilities]
    return np.random.choice(range(self.num_cities),
                            p=probabilities)

def construct_solution(self):
    for _ in range(self.num_ants):
        visited = [0]
        current_city = 0
        for _ in range(1, self.num_cities):
            next_city = self.select_next_city
                        (current_city, visited)
            visited.append(next_city)
            current_city = next_city
        visited.append(0)
        yield visited
```

Code:

```
import numpy as np
import random

class AntColony:
    def __init__(self, cities, num_ants=10, alpha=1.0, beta=2.0, rho=0.5,
iterations=100):
        self.cities = cities
        self.num_ants = num_ants
        self.alpha = alpha
        self.beta = beta
        self.rho = rho
        self.iterations = iterations
        self.num_cities = len(cities)
        self.pheromone = np.ones((self.num_cities, self.num_cities))
        self.distance = self.calculate_distances()
    def calculate_distances(self):
        distances = np.zeros((self.num_cities, self.num_cities))
        for i in range(self.num_cities):
            for j in range(i + 1, self.num_cities):
                distances[i][j] = distances[j][i] =
np.linalg.norm(np.array(self.cities[i]) - np.array(self.cities[j]))
        return distances
    def select_next_city(self, current_city, visited):
        probabilities = []
        for next_city in range(self.num_cities):
            if next_city not in visited:
                pheromone = self.pheromone[current_city][next_city] **
self.alpha
                heuristic = (1 / self.distance[current_city][next_city]) **
self.beta
                probabilities.append(pheromone * heuristic)
            else:
                probabilities.append(0)
        total = sum(probabilities)
        probabilities = [p / total for p in probabilities]
        return np.random.choice(range(self.num_cities), p=probabilities)
    def construct_solution(self):
        for _ in range(self.num_ants):
            visited = [0]
            current_city = 0
            for _ in range(1, self.num_cities):
                next_city = self.select_next_city(current_city, visited)
                visited.append(next_city)
                current_city = next_city
            visited.append(0)  # Return to starting city
```

```
            yield visited
    def update_pheromones(self, solutions):
        self.pheromone *= (1 - self.rho)  # Evaporation
        for solution in solutions:
            length = self.calculate_tour_length(solution)
            pheromone_deposit = 1 / length
            for i in range(len(solution) - 1):
                self.pheromone[solution[i]][solution[i + 1]] +=
pheromone_deposit
    def calculate_tour_length(self, solution):
        return sum(self.distance[solution[i]][solution[i + 1]] for i in
range(len(solution) - 1))

    def run(self):
        best_solution = None
        best_length = float('inf')
        for _ in range(self.iterations):
            solutions = list(self.construct_solution())
            self.update_pheromones(solutions)
            for solution in solutions:
                length = self.calculate_tour_length(solution)
                if length < best_length:
                    best_length = length
                    best_solution = solution
        return best_solution, best_length
cities = [(0, 0), (1, 2), (2, 1), (4, 4), (2, 4)]
aco = AntColony(cities)
best_route, best_distance = aco.run()
print("Best Route:", best_route)
print("Best Distance:", best_distance)

Output :

Best Route: [0, 1, 4, 3, 2, 0]
Best Distance: 12.313755207963359
```

Program 4
Cuckoo Search (CS)

Algorithm:

2/11/24 Cuckoo Search Algorithm

```
import numpy as np
import math
def objective_function(x):
    return np.sum(x**2)

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

def cuckoo_search(obj_fun, dim, lb, ub, num_nests
                  =25, max_iter=100, pa=0.25):
    nests = np.random.rand(num_nests, dim)
            * (ub-lb) + lb
    fitness = np.apply_along_axis(objective_
              function, 1, nests)

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

    for iteration in range(max_iter):
        for i in range(num_nests):
            step = levy_flight(size=dim)
            new_nest = nests[i] + 0.01 * step
            new_nest = np.clip(new_nest, lb, ub)
            new_fitness = obj_fun(new_nest)

            if new_fitness < fitness[i]:
                nests[i] = new_nest
                fitness[i] = new_fitness

        for i in range(num_nests):
            if np.random.rand() < pa:
                nests[i] = np.random.rand(dim)
                           *(ub, lb) + lb
                fitness[i] = obj_fun(nests[i])
    return best_nest, best_fitness

dim = 5
lb = -5
ub = 5
best_solution, best_fitness = cuckoo_search(obj_fun, dim,
                              lb, ub, num_nests=25,
                              max_iter=100, pa=0.25)

print(f"Best Solution: {best_solution}")
print(f"Best Fitness: {best_fitness}")
```

Output

Best Solution : [0.6498, 0.5496, 2.015, 0.9398, 0.3198]
Best Fitness : 5.7814

Code:

```
import numpy as np
import math
# Objective function to optimize (example: Sphere function)
def objective_function(x):
    return np.sum(x**2)

# Lévy Flight distribution
def levy_flight(beta=1.5, size=1):
    sigma_u = (math.gamma(1 + beta) * np.sin(np.pi * beta / 2) /
               math.gamma((1 + beta) / 2) * beta * (2 ** ((beta - 1) /
2)))**(1 / beta)
    u = np.random.normal(0, sigma_u, size)
    v = np.random.normal(0, 1, size)
    step = u / (np.abs(v) ** (1 / beta))
    return step

# Cuckoo Search Algorithm
def cuckoo_search(objective_function, dim, lower_bound, upper_bound,
num_nests=25, max_iter=100, pa=0.25):
    # Initialize nests with random solutions within bounds
    nests = np.random.rand(num_nests, dim) * (upper_bound - lower_bound) +
lower_bound
    fitness = np.apply_along_axis(objective_function, 1, nests)

    # Initialize the best solution
    best_nest_idx = np.argmin(fitness)
    best_nest = nests[best_nest_idx]
    best_fitness = fitness[best_nest_idx]

    # Iterate for a fixed number of generations or until convergence
    for iteration in range(max_iter):
        for i in range(num_nests):
            # Generate a new solution using Lévy flight
            step = levy_flight(size=dim)
            new_nest = nests[i] + 0.01 * step
            new_nest = np.clip(new_nest, lower_bound, upper_bound)

            # Evaluate the new solution
            new_fitness = objective_function(new_nest)

            # If the new solution is better, replace the old solution
            if new_fitness < fitness[i]:
                nests[i] = new_nest
                fitness[i] = new_fitness
```

```
        # Abandon the worst nests
        for i in range(num_nests):
            if np.random.rand() < pa:  # Probability to abandon
                nests[i] = np.random.rand(dim) * (upper_bound - lower_bound)
+ lower_bound
                fitness[i] = objective_function(nests[i])

        # Find the current best nest
        best_nest_idx = np.argmin(fitness)
        best_nest = nests[best_nest_idx]
        best_fitness = fitness[best_nest_idx]

        # print(f"Iteration {iteration+1}, Best Fitness: {best_fitness}")

    return best_nest, best_fitness

# Example usage of Cuckoo Search

# Define the problem dimensions and bounds
dim = 5  # Dimension of the solution space
lower_bound = -5  # Lower bound of the search space
upper_bound = 5  # Upper bound of the search space

# Run Cuckoo Search
best_solution, best_fitness = cuckoo_search(objective_function, dim,
lower_bound, upper_bound, num_nests=25, max_iter=100, pa=0.25)

print(f"Best Solution: {best_solution}")
print(f"Best Fitness: {best_fitness}")

Output :

Best Solution: [0.64982748 0.55961241 2.01501756 0.93987275 0.31984962]
Best Fitness: 5.78140211553397
```

Program 5
Grey Wolf Optimizer (GWO)

Algorithm:

Algorithm:

1) Start
2) Initialize parameters :
   N → number of wolves
   dim → dimensions , T → iterations
   upper-bound, lower-bound

3) Randomly initialize positions of wolves within search space

4) Initialize Best Positions : alpha, beta, delta position
5) For each wolf

   a) Evaluate its fitness using obj-function
   b) Update alpha, beta, delta

6) Check Boundary Constraints
7) Repeat for T iterations

8) Output the best solution which is alpha-pos with its fitness score

20/11/24

Code:

```
import numpy as np

# Objective function (example: Sphere function)
def objective_function(x):
    return np.sum(x**2)
N, dim, T = 30, 10, 100  # Number of wolves, dimensions, iterations
lower_bound, upper_bound = -10, 10

wolves = np.random.uniform(lower_bound, upper_bound, (N, dim))

alpha_pos, beta_pos, delta_pos = np.zeros(dim), np.zeros(dim), np.zeros(dim)
alpha_score, beta_score, delta_score = float('inf'), float('inf'),
float('inf')
for t in range(T):
    for i in range(N):
        fitness = objective_function(wolves[i])  # Evaluate fitness
        if fitness < alpha_score:
            delta_score, delta_pos = beta_score, beta_pos.copy()
            beta_score, beta_pos = alpha_score, alpha_pos.copy()
            alpha_score, alpha_pos = fitness, wolves[i].copy()
        elif fitness < beta_score:
            delta_score, delta_pos = beta_score, beta_pos.copy()
            beta_score, beta_pos = fitness, wolves[i].copy()
        elif fitness < delta_score:
            delta_score, delta_pos = fitness, wolves[i].copy()
    a = 2 - t * (2 / T)
    for i in range(N):
        r1, r2 = np.random.rand(dim), np.random.rand(dim)
        A, C = 2 * a * r1 - a, 2 * r2
        wolves[i] += A * (abs(C * alpha_pos - wolves[i]) +
                          abs(C * beta_pos - wolves[i]) +
                          abs(C * delta_pos - wolves[i]))

        wolves[i] = np.clip(wolves[i], lower_bound, upper_bound)
print("Best Solution:", alpha_pos)
print("Best Score:", alpha_score)

Output :
```

```
Best Solution: [-1.28434275  1.94786008  0.82301541 -1.85113457 -2.08806377
3.74582237
  0.84065243  0.8938704  -1.22271966 -0.29007149]
Best Score: 31.023829961456407
```

Program 6
Parallel Cellular Algorithms and Programs

Algorithm:

Bafna Gold
Date: Page:

parallel_cellular_algorithm ()

Output

Best Solution : [0.00129305, -0.00150246]
Best Fitness : 0.00078014 39

Algorithm

1) Define the Objective function → Rastrigin function

2) Initialize Parameters

3) Initialize Population , create grid of cells

4) Evaluate fitness based on Rastrigin function

5) Update Cell States , identify neighboring cells within the defined radius

6) Parallelize cell updates using parallel computing to simultaneously update the states of all cells.

7) Track Best Solution based on fitness at each iteration

27/11/24

Code:

```
import numpy as np
import random
import concurrent.futures

def rastrigin(x):
    A = 10
    return A * len(x) + sum([(xi ** 2 - A * np.cos(2 * np.pi * xi)) for xi in
x])

GRID_SIZE = (10, 10)
DIM = 2
RADIUS = 1
ITER = 100
BEST = None

def init_grid(size, dim):
    return [[np.random.uniform(-5.12, 5.12, size=(dim,)) for _ in
range(size[1])] for _ in range(size[0])]

def fitness(cell):
    return rastrigin(cell)

def update_state(grid, i, j, radius):
    curr = grid[i][j]
    fitness_curr = fitness(curr)
    neighbors = [grid[ni][nj] for dx in range(-radius, radius+1) for dy in
range(-radius, radius+1)
                 if 0 <= (ni := i+dx) < len(grid) and 0 <= (nj := j+dy) <
len(grid[0]) and (dx or dy)]
    if neighbors:
        best_neigh = min(neighbors, key=fitness)
        return curr + 0.1 * (best_neigh - curr)
    return curr

def run_iteration(grid, radius):
    new_grid = [[None for _ in range(len(grid[0]))] for _ in
range(len(grid))]
    with concurrent.futures.ThreadPoolExecutor() as ex:
        futures = [ex.submit(update_state, grid, i, j, radius) for i in
range(len(grid)) for j in range(len(grid[0]))]
        for idx, future in enumerate(futures):
            i, j = divmod(idx, len(grid[0]))
            new_grid[i][j] = future.result()
    return new_grid
```

```python
def track_best(grid):
    global BEST
    best_cell, best_fitness = None, float('inf')
    for row in grid:
        for cell in row:
            f = fitness(cell)
            if f < best_fitness:
                best_fitness = f
                best_cell = cell
    if BEST is None or best_fitness < fitness(BEST):
        BEST = best_cell

def parallel_cellular_algorithm():
    global BEST
    grid = init_grid(GRID_SIZE, DIM)
    for _ in range(ITER):
        grid = run_iteration(grid, RADIUS)
        track_best(grid)
        print(f"Best Fitness: {fitness(BEST)}")
    print("Best Solution:", BEST)
    print("Best Fitness:", fitness(BEST))

parallel_cellular_algorithm()

Output :
```

```
Best Fitness: 2.4309484366586602
Best Fitness: 2.4309484366586602
Best Fitness: 0.0007801439196555293
Best Fitness: 0.0007801439196555293
Best Fitness: 0.0007801439196555293
Best Solution: [ 0.00129305 -0.00150346]
Best Fitness: 0.0007801439196555293
```

Program 7

Optimization via Gene Expression Algorithms

Algorithm:

02/11/24

Optimization via Gene Expression Algorithms

```
import numpy as np

def optimization_function(x):
    return np.sum(x**2)

POPULATION_SIZE = 50
GENE_LENGTH = 5
MUTATION_RATE = 0.1
CROSSOVER_RATE = 0.7
GENERATIONS = 100
SEARCH_SPACE = (-10, 10)

def initialize_population():
    return np.random.uniform(SEARCH_SPACE[0],
                             (POPULATION, GENE))

def evaluate_fitness(population):
    fitness = np.array([optimization_function(ind)
                        for ind in population])
    return fitness

def select_parents(population, fitness):
    inverted_fitness = 1/(fitness + 1e-6)
    selection_prob = inverted_fitness / np.sum(inverted)
    selected_ind = np.random.choice(np.arange(POPO_SIZE
                   size = POP_SIZE, p = selection_prob)

    return population[selected_indices]
```

Code:

```
import numpy as np

# Define the mathematical function to optimize (example: minimize f(x) = x^2)
def optimization_function(x):
    return np.sum(x**2)  # Modify this for other functions to optimize

# Parameters
POPULATION_SIZE = 50  # Number of individuals
GENE_LENGTH = 5       # Number of genes (dimensions of the problem)
MUTATION_RATE = 0.1   # Probability of mutation
CROSSOVER_RATE = 0.7  # Probability of crossover
GENERATIONS = 100     # Number of generations
SEARCH_SPACE = (-10, 10)  # Range of values for genes

# Initialize Population
def initialize_population():
    return np.random.uniform(SEARCH_SPACE[0], SEARCH_SPACE[1],
(POPULATION_SIZE, GENE_LENGTH))

# Evaluate Fitness (lower is better for minimization)
def evaluate_fitness(population):
    fitness = np.array([optimization_function(ind) for ind in population])
    return fitness

# Selection (Roulette Wheel Selection)
def select_parents(population, fitness):
    # Convert fitness to probabilities (lower fitness is better)
    inverted_fitness = 1 / (fitness + 1e-6)  # Avoid division by zero
    selection_prob = inverted_fitness / np.sum(inverted_fitness)
    selected_indices = np.random.choice(np.arange(POPULATION_SIZE),
size=POPULATION_SIZE, p=selection_prob)
    return population[selected_indices]

# Crossover (Blend Crossover)
def crossover(parents):
    offspring = np.empty_like(parents)
    for i in range(0, POPULATION_SIZE, 2):
        p1, p2 = parents[i], parents[i+1]
        if np.random.rand() < CROSSOVER_RATE:
            alpha = np.random.rand()  # Blending factor
            offspring[i] = alpha * p1 + (1 - alpha) * p2
            offspring[i+1] = alpha * p2 + (1 - alpha) * p1
        else:
            offspring[i], offspring[i+1] = p1, p2
```

```
    return offspring

# Mutation (Random Perturbation)
def mutate(offspring):
    for i in range(POPULATION_SIZE):
        if np.random.rand() < MUTATION_RATE:
            mutation_point = np.random.randint(0, GENE_LENGTH)
            offspring[i][mutation_point] += np.random.uniform(-1, 1)
            # Keep within search space
            offspring[i][mutation_point] =
np.clip(offspring[i][mutation_point], SEARCH_SPACE[0], SEARCH_SPACE[1])
    return offspring

# Gene Expression (Translate Genetic Code into Solutions)
def gene_expression(genes):
    # In this simple example, the genes directly represent the solution
    return genes

# Main Algorithm
def gene_expression_algorithm():
    # Initialize population
    population = initialize_population()
    best_solution = None
    best_fitness = float('inf')

    # Iterate through generations
    for generation in range(GENERATIONS):
        # Evaluate fitness
        fitness = evaluate_fitness(population)

        # Track the best solution
        current_best_idx = np.argmin(fitness)
        if fitness[current_best_idx] < best_fitness:
            best_fitness = fitness[current_best_idx]
            best_solution = population[current_best_idx]

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

        # Selection
        parents = select_parents(population, fitness)

        # Crossover
        offspring = crossover(parents)

        # Mutation
        offspring = mutate(offspring)
```

```
        # Gene Expression (not needed explicitly as genes represent solutions)
        population = gene_expression(offspring)

    print("\nOptimal Solution Found:")
    print("Best Solution:", best_solution)
    print("Best Fitness:", best_fitness)

# Run the algorithm
if __name__ == "__main__":
    gene_expression_algorithm()

Output :

Generation 1: Best Fitness = 16.545885126119284
Generation 2: Best Fitness = 11.641082640808637
…
Generation 99: Best Fitness = 0.02233046748484963
Generation 100: Best Fitness = 0.02233046748484963

Optimal Solution Found:
Best Solution: [ 0.07226226 -0.11854791  0.03245473 -0.01236219  0.04299877]
Best Fitness: 0.02233046748484963
```