

VISVESVARAYA TECHNOLOGICAL UNIVERSITY

“JnanaSangama”, Belgaum -590014, Karnataka.



LAB RECORD

Bio Inspired Systems (23CS5BSBIS)

Submitted by

SYED FARHAN (1BM23CS424)

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 **Syed Farhan (1BM23CS424)**, 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.

Syed Akram Assistant Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE
---	--

Index

Sl. No.	Date	Experiment Title	Page No.
1	03/10/24	GENETIC ALGORITHM	4-9
2	24/10/24	PARTICLE SWARM OPTIMIZATION	10-16
3	07/11/24	ANT COLONY OPTIMIZATION	17-23
4	21/11/24	CUCKOO SEARCH ALGORITHM	24-26
5	28/11/24	GREY WOLF OPTIMIZATION	27-31
6	12/12/24	PARALLEL CELLULAR ALGORITHM	32-36
7	12/12/24	GENE EXPRESSION ALGORITHM	37-41

Github Link: <https://github.com/Syed-Farhan-bmsce/BIS.git>

Program 1

Genetic Algorithm for Optimization Problems

Algorithm:

Lab-1

Manjal
Date: 31/01/24
Page: 01

```
class GeneticAlgorithm:
    def __init__(self):
        self.population_size = 100
        self.genomes = ["abcde fghij klmnopqrstuvwxyz ABCDEFGHIJKLMNOPQRSTUVWXYZ"]
        self.mutation_rate = 0.01
        self.target = "BESTI"
        self.fitness_fn = None

    def create_individual(self):
        genome = []
        for _ in range(len(self.genomes[0])):
            gene = random.choice(self.genomes)
            genome.append(gene)
        return Individual("".join(genome))

    def evaluate_population(self):
        for individual in self.population:
            fitness = self.fitness_fn(individual)
            individual.fitness = fitness

    def select_parents(self):
        parents = []
        for _ in range(len(self.population) // 2):
            parent1 = self.select_random()
            parent2 = self.select_random()
            parents.append((parent1, parent2))
        return parents

    def crossover(self, parents):
        children = []
        for parent1, parent2 in parents:
            child = self.create_individual()
            child.chromosome = self.mutate(self.create_crossover(parent1.chromosome, parent2.chromosome))
            children.append(child)
        return children

    def mutate(self, chromosome):
        mutated_chromosome = []
        for gene in chromosome:
            if random.random() < self.mutation_rate:
                mutated_chromosome.append(random.choice(self.genomes))
            else:
                mutated_chromosome.append(gene)
        return mutated_chromosome

    def create_crossover(self, parent1, parent2):
        crossover_point = random.randint(0, len(parent1) - 1)
        child1 = parent1[:crossover_point] + parent2[crossover_point:]
        child2 = parent2[:crossover_point] + parent1[crossover_point:]
        return (child1, child2)

    def select_random(self):
        index = random.randint(0, len(self.population) - 1)
        return self.population[index]

    def find_best(self):
        best_fitness = float("-inf")
        best_genome = None
        for individual in self.population:
            if individual.fitness > best_fitness:
                best_fitness = individual.fitness
                best_genome = individual
        return best_genome
```

```

child = chromosome.append(gp1)
elif prob < 0.90:
    child = chromosome.append(gp2)
else:
    child = chromosome.append(gp1)
    child = chromosome.append(gp2)
return Individual(child, chromosome)

```

```

def cal_fitness(self):
    global TARGET
    fitness = 0
    for ge, gt in zip(self.chromosome, TARGET):
        if ge != gt: fitness += 1
    return fitness

```

```
def mainer:
```

```

    global POPULATION_SIZE
    generation = 1
    found = False
    population = []
    for i in range(POPULATION_SIZE):
        genome = Individual.create_genome()
        population.append(Individual(genome))

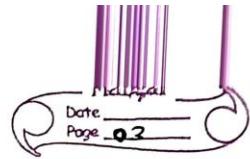
```

```
while not found:
```

```

    population = sorted(population, key=lambda
x: x.fitness)
    if population[0].fitness == 0:
        found = True
        break

```



new_generation = []

s = int((10 * population_size) / 100)

new_generation.extend(population[:s])

s = int((90 * population_size) / 100)

for i in range(s):

parent1 = random.choice(population[1:s])

parent2 = random.choice(population[1:s])

child = parent1.mate(parent2)

new_generation.append(child)

new_population = new_generation

new_population.sort(key=lambda x: x.fitness, reverse=True)

new_population = new_population[:int(len(new_population) * 0.1)]

new_population.sort(key=lambda x: x.fitness)

generation += 1

print("Generation: ", generation, " fitness: ", new_population[0].fitness)

point('Generation: ' + str(generation) + ' fitness: ' + str(new_population[0].fitness))

new_population.sort(key=lambda x: x.fitness, reverse=True)

new_population = new_population[:int(len(new_population) * 0.1)]

new_population.sort(key=lambda x: x.fitness)

if name == '1-methylimidazole': mutation

name()

output: gen: 1 s1n: BR6CP fitness: 3.1

gen: 2 s1n: BR6CP fitness: 3.1

gen: 3 s1n: BR6CP fitness: 3

gen: 4 s1n: 6n9SCF fitness: 2

gen: 5 s1n: 6n9SCF fitness: 2

gen: 6 s1n: B#SCF fitness: 1.1

gen: 7 s1n: B#SCF fitness: 1

gen: 8 s1n: BHSCF fitness: 1

gen: 9 s1n: BMSCF fitness: 0

Code:

```
import random

POPULATION_SIZE = 100

GENES = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ1234567890,-;:_!#%&/()=?@${[]}""

TARGET = "I love GeeksforGeeks"

class Individual(object):
    """
    Class representing individual in population
    """

    def __init__(self, chromosome):
        self.chromosome = chromosome
        self.fitness = self.cal_fitness()

    @classmethod
    def mutated_genes(self):
        """
        create random genes for mutation
        """
        global GENES
        gene = random.choice(GENES)
        return gene

    @classmethod
    def create_gnome(self):
        """
        create chromosome or string of genes
        """
        global TARGET
        gnome_len = len(TARGET)
        return [self.mutated_genes() for _ in range(gnome_len)]

    def mate(self, par2):
        """
        Perform mating and produce new offspring
        """

        child_chromosome = []
        for gp1, gp2 in zip(self.chromosome, par2.chromosome):

            prob = random.random()
            if prob < 0.45:
                child_chromosome.append(gp1)
            else:
                child_chromosome.append(gp2)

        return Individual(child_chromosome)
```

```

        child_chromosome.append(gp1)

    elif prob < 0.90:
        child_chromosome.append(gp2)

    else:
        child_chromosome.append(self.mutated_genes())

    return Individual(child_chromosome)

def cal_fitness(self):
    """
    Calculate fitness score, it is the number of
    characters in string which differ from target
    string.
    """
    global TARGET
    fitness = 0
    for gs, gt in zip(self.chromosome, TARGET):
        if gs != gt: fitness+= 1
    return fitness

def main():
    global POPULATION_SIZE

    generation = 1

    found = False
    population = []

    for _ in range(POPULATION_SIZE):
        gnome = Individual.create_gnome()
        population.append(Individual(gnome))

    while not found:

        population = sorted(population, key = lambda x:x.fitness)

        if population[0].fitness <= 0:
            found = True
            break

        new_generation = []

        s = int((10*POPULATION_SIZE)/100)
        new_generation.extend(population[:s])

```

```
s = int((90*POPULATION_SIZE)/100)
for _ in range(s):
    parent1 = random.choice(population[:50])
    parent2 = random.choice(population[:50])
    child = parent1.mate(parent2)
    new_generation.append(child)

population = new_generation

print("Generation: {} \tString: {} \tFitness: {}".format(generation,
    ''.join(population[0].chromosome),
    population[0].fitness))

generation += 1

print("Generation: {} \tString: {} \tFitness: {}".format(generation,
    ''.join(population[0].chromosome),
    population[0].fitness))

if __name__ == '__main__':
    main()
```

Program 2

PARTICLE SWARM OPTIMIZATION

Algorithm:

Lab - 2	Manal Date 24/10/24 Page 4
Procedure for Swarm optimization for function	
Initial Optimization:	
Import random module	
Import math module	
Import copy module	
Import sys module	
def fitness_mountain(position):	
fitnessVal = 0.0	
for i in range(len(position)):	
xi = position[i]	
fitnessVal += (xi**2) - (10 * math.cos(2 *	
math.pi * xi)) + 10	
return fitnessVal	
def fitnessSphere(position):	
fitnessVal = 0.0	
for i in range(len(position)):	
xi = position[i]	
fitnessVal += (xi**2)	
return fitnessVal	
class particle:	
def __init__(self, fitness, dim, minx, maxx, seed):	
self.dim = random.Random(seed)	
self.position = [0.0 for i in range(dim)]	
self.velocity = [0.0 for i in range(dim)]	
self.bestPosition = [0.0 for i in range(dim)]	
for i in range(len(self.position)):	
self.position[i] = (maxx - minx) *	
random.uniform(minx, maxx)	

self. velocity[i] = (maxx - minx) * self. rand. random() + minx).

Self. fitness = fitness(slf. position).

self. begit. part - pos = copy. copy (self. position)

$$\text{Self. best_point_fitness}[\text{val}] = \text{self. fitness}$$

```
def psn(fitres, max_iter, n, dim, minx, maxx):
```

$$M = 0.729$$

$$c_1 = 1.49445$$

$$L_2 = 1.49445$$

mind = Random(0, 1)

Swarm = [Particle (fitness); dim, minx, maxx, i)
for i in range (n)]:

beat. swimm^t pos. = [0.0 fm, 9m orange(dim)]

heat - swarm fitness vol. = sys. filnat. info. max.

for i in range(n): if i < n - 1:

?f swarm [?]. fitness < best_swarm_fitnessval:

~~and it is a - final best - Swami - pos = copy, copy(Swami T)~~

position) | found it was

$$T_{\text{tor}} = 0$$

which her most Peter: at the same time, 18

9f) $\lim_{n \rightarrow \infty} \frac{1}{n} \ln(1 + \frac{1}{n}) = 0$, and $\lim_{n \rightarrow \infty} \frac{1}{n} > 1$:

point $(f, 12\pi) = (12\pi)$; best fitness = best swarm-fitnessVol: (87%)

from ? to orange(n): i-1-i+1

from K in orange (dim):

$\sigma_1 = \text{rand. random}()$

$\sigma_2 = \text{rand. random}()$

$\text{swarm}[i]. velocity[k] =$

$w * \text{swarm}[i]. velocity[k] +$

$c_1 * \sigma_1 * (\text{swarm}[i]. best_pos[k]) +$

$- (\text{swarm}[i]. position[k]) * \sigma_2$

$c_2 * \sigma_2 * (\text{best_swarm_pos}[k] - \text{swarm}[i].$

$.position[k])$

If $\text{swarm}[i]. velocity[k] < \min$:

$\text{swarm}[i]. velocity[k] = \min$

Else if $\text{swarm}[i]. velocity[k] > \max$:

$\text{swarm}. velocity[i] = \max$.

End loop for all individuals (dpm):

Final: $\text{swarm}[i] = \text{position}[i] + \text{swarm}[i]. velocity$

$\text{swarm}[i]. fitness = \text{fitness}(\text{swarm}[i]. position)$

If $\text{swarm}[i]. fitness < \text{swarm}[i]. best_fitness$:

$\text{swarm}[i]. best_pos = \text{swarm}[i]. fitness$

$\text{swarm}[i]. best_pos[i] = \text{copy_copy}(\text{swarm}[i].$

$.position)$

If $\text{swarm}[i]. fitness < \text{best_swarm_fitness}$:

$\text{best_swarm_fitness} = \text{swarm}[i]. fitness$

$\text{best_swarm_pos} = \text{copy_copy}(\text{swarm}[i].$

$.position)$

$i + 1 = 1$

$\text{return best_swarm_pos}$



point ("In Begin of particle swarm optimization on Repertoir function")
dim = 3
fitness_{ps} = fitness - max origin.
point (f "Goal is to minimize Repertoir's function in (dim variable)")
point ("function has known min = 0.0 at (0, end = "")
point (".", ".join ([0] * (dim - 1)) + ", 0)")
num_particles = 50
max_iter = 100
point ("Setting num_particles = %d from particle")
point ("Setting max_iter = %d" % max_iter)
point ("In Starting PSO algorithm")

best_position = pso(fitness, max_iter, num_particles,
dim, -10.0, 10.0)
point ("In PSO completed")
point ("In Best solution found :")
point ("f = best_position [k]. gf y" for k in range(dim))
fitness_{goal} = fitness(best_position)
point ("Fitness of best solution = %f" % fitness_{goal})
point ("In End particle swarm for Repertoir function")

point (f "Goal is to minimize Sphere function in (dim).
variables")
point ("function has known min = 0.0 at ", end = "")
point (".", ".join ([0] * (dim - 1)) + ", 0")
num_particles = 50
max_iter = 100

point(f"Setting num. particles to 5 from particle")
point(f"Setting max_iter = {max_iter}")
point("In Starting PSO algorithm\n")

best_position = pso(fitneess, max_iter, num_particle, dim, -10.0, 10.0)

point("In PSO completed\n")
point("In Best Solution found :")
point(f"Best position [k] : .6f 3" for x in range(dim))

fitneess_val = fitness(best_position)
point(f"Fitness of best solution = {fitneess_val:.6f}\")

point("In End of procedure Current Best Sphere function value")

O/P ("Calculated value 0.0001 current min 0.0001")

Output: Best Solution found [0.0001, -0.0001, 0.0001]

Best solution: 0.0001

(Best solution with 3 digits)

(The current solution is 0.0001)

Current best sphere function value = 0.0001

Condition: best's function value <= 0.0001

Best position found by current best sphere function value

Best position with min function value = 0.0001

Current best sphere function value = 0.0001

Best position found by current best sphere function value

(Current best sphere function value = 0.0001)

Condition: best's function value <= 0.0001

Best position found by current best sphere function value

(Current best sphere function value = 0.0001)

Condition: best's function value <= 0.0001

Best position found by current best sphere function value

(Current best sphere function value = 0.0001)

Code:

```
import random
import math
import copy
import sys

def fitness_rastrigin(position):
    return sum((xi * xi) - (10 * math.cos(2 * math.pi * xi)) + 10 for xi in position)

def fitness_sphere(position):
    return sum(xi * xi for xi in position)

class Particle:
    def __init__(self, fitness, dim, minx, maxx, seed):
        self.rnd = random.Random(seed)
        self.position = [(maxx - minx) * self.rnd.random() + minx for _ in range(dim)]
        self.velocity = [(maxx - minx) * self.rnd.random() + minx for _ in range(dim)]
        self.best_part_pos = self.position[:]
        self.fitness = fitness(self.position)
        self.best_part_fitnessVal = self.fitness

def pso(fitness, max_iter, n, dim, minx, maxx):
    w, c1, c2 = 0.729, 1.49445, 1.49445
    rnd = random.Random(0)
    swarm = [Particle(fitness, dim, minx, maxx, i) for i in range(n)]

    best_swarm_pos, best_swarm_fitnessVal = [0.0] * dim, sys.float_info.max
    for p in swarm:
        if p.fitness < best_swarm_fitnessVal:
            best_swarm_fitnessVal = p.fitness
            best_swarm_pos = p.position[:]

    for Iter in range(max_iter):
        if Iter % 10 == 0 and Iter > 1:
            print(f"Iter = {Iter} best fitness = {best_swarm_fitnessVal:.3f}")

        for p in swarm:
            for k in range(dim):
                r1, r2 = rnd.random(), rnd.random()
                p.velocity[k] = w * p.velocity[k] + c1 * r1 * (p.best_part_pos[k] - p.position[k]) + c2 * r2 * (best_swarm_pos[k] - p.position[k])
                p.velocity[k] = max(min(p.velocity[k], maxx), minx)

            p.position = [p.position[k] + p.velocity[k] for k in range(dim)]
            p.fitness = fitness(p.position)

            if p.fitness < p.best_part_fitnessVal:
```

```

p.best_part_fitnessVal = p.fitness
p.best_part_pos = p.position[:]

if p.fitness < best_swarm_fitnessVal:
    best_swarm_fitnessVal = p.fitness
    best_swarm_pos = p.position[:]

return best_swarm_pos

def run_pso(fitness, dim, minx, maxx):
    print(f"Goal is to minimize the function in {dim} variables")
    print(f"Function has known min = 0.0 at ({', '.join(['0'] * (dim - 1))}, 0)")

    num_particles, max_iter = 50, 100
    best_position = pso(fitness, max_iter, num_particles, dim, minx, maxx)

    print(f"Best solution found: {', '.join([f'{x:.6f}' for x in best_position])}")
    print(f"Fitness of best solution = {fitness(best_position):.6f}\n")

print("\nBegin PSO for Rastrigin function\n")
run_pso(fitness_rastrigin, 3, -10.0, 10.0)

print("\nBegin PSO for Sphere function\n")
run_pso(fitness_sphere, 3, -10.0, 10.0)

```

PROGRAM 3

ANT COLONY OPTIMIZATION

Algorithm:

Ant Colony Optimization for the Travelling Salesman Problem.

```
Import random
Import numpy as np
Import math
# Import ACO TSP function
class City:
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def distance(self, city):
        return math.sqrt((self.x - city.x)**2 + (self.y - city.y)**2)
class ACO_TSP:
    def __init__(self, cities, num_ants, num_iterations, alpha = 1.0, beta = 2.0, rho = 0.9):
        self.cities = cities
        self.num_ants = num_ants
        self.num_iterations = num_iterations
        self.alpha = alpha
        self.beta = beta
        self.rho = rho
        self.Q = 100
        self.pheromone = np.zeros(len(cities) * len(cities))
    def pheromone_update(self, num_cities, self.num_ants, self.num_iterations):
        for i in range(len(self.cities)):
            for j in range(i+1, len(self.cities)):
                if self.pheromone[i * len(self.cities) + j] < self.Q:
                    self.pheromone[i * len(self.cities) + j] += self.Q
    def heuristic(self, i, j):
        if self.cities[i].x == self.cities[j].x and self.cities[i].y == self.cities[j].y:
            return 0
        else:
            return 1 / self.cities[i].distance(self.cities[j])
    def tour(self, ant):
        tour = []
        for i in range(len(self.cities)):
            if i not in ant:
                tour.append(ant[i])
        tour.append(tour[0])
        return tour
    def calculate_fitness(self, tour):
        fitness = 0
        for i in range(len(tour) - 1):
            fitness += self.cities[tour[i]].distance(self.cities[tour[i+1]])
        return fitness
    def select_next_city(self, ant, current, fitness):
        probabilities = np.zeros(len(self.cities))
        for i in range(len(self.cities)):
            if i != current:
                probabilities[i] = ((self.Q / self.cities[current].distance(self.cities[i])) / fitness) ** self.alpha * ((1 / self.heuristic(current, i)) ** self.beta)
        total_prob = np.sum(probabilities)
        if total_prob == 0:
            return current
        else:
            return np.random.choice(len(self.cities), p=probabilities / total_prob)
    def run(self):
        best_tour = None
        best_fitness = float('inf')
        for iteration in range(self.num_iterations):
            tours = []
            for ant in range(self.num_ants):
                tour = [0]
                for i in range(len(self.cities) - 1):
                    next_city = self.select_next_city(ant, tour[-1], self.calculate_fitness(tour))
                    tour.append(next_city)
                tours.append(tour)
            fitnesses = np.array([self.calculate_fitness(tour) for tour in tours])
            best_index = np.argmin(fitnesses)
            best_tour = tours[best_index]
            best_fitness = fitnesses[best_index]
            self.pheromone_update(best_tour, self.num_ants, self.num_iterations)
        return best_tour, best_fitness
```

`self.beet_total_length = float('inf')`

`for i in range(self.num_cities):`

`for j in range(i+1, self.num_cities):`

`dist = cities[i].distance(cities[j])`

`self.heuristic[i][j] = 1.0 / dist if dist > 0`

`edge =`

`self.heuristic[i][j] = self.heuristic[i][j]`

`def select_next_city(self, current_city, visited_cities):`

`probabilities = np.zeros(self.num_cities)`

`total_phenomone = 0.0`

`for city in range(self.num_cities):`

`if city not in visited_cities:`

`phenomone = self.phenomone[current_city][city]`

`alpha = self.alpha`

`heuristic = self.heuristic[current_city][city]`

`beta =`

`probability_p[city] = phenomone + heuristic`

`total_phenomone += probability_p[city]`

`If`

`total_phenomone == 0.0`

`autumn_random_choice = random.choice([city for city in range(self.num_cities) if city not in visited_cities])`

`probability_p[autumn_random_choice] = 1.0`

`if random.random() < self.g:`

`next_city = np.argmax(probability_p)`

`else:`

`next_city = np.random.choice(self.num_cities, p=probability_p)`

def update_phenomone(self, ontg):

$$\text{self.phenomone}^i = (1 - \text{self.who})$$

(1) Initialization function

for ont in ontg: ~~when a city found~~

$$\text{phenomone-deposit} = 1.0 / \text{ont.tour.length}$$

for i in range (self.num_cities):

current_tour.city[i] = self.ont[i]

$$\text{next_tour} = \text{ont.tour}[i+1 : self.num_cities]$$

$$\text{self.phenomone}[current_city][next_city] +=$$

self.who * deposit * phenomone-deposit

$$\text{self.phenomone}[next_city][current_city] +=$$

(self.who * phenomone-deposit)

def iteration(self): ~~initialization and 1st iteration~~

for iteration in range (self.num_iterations):

$$\text{self.ont} = [\text{And}(self.ont[i].city, self.ont[i].tour) for i in range (self.num_onts)]$$

for ont in ontg:

ont.computant_solutions()

def (self, ontg): ~~(self, ontg, self.update_phenomone(ontg))~~

self.ont = [ont for ont in ontg if ont.tour.length == 1]

(e.g. if ont.tour.length == self.belt.tour.length):

$$\text{current_ont} = \text{And}(\text{self.belt.tour}, \text{ont.tour}) = \text{ont.tour.length}$$

$$\text{self.belt.tour} = \text{ont.tour}$$

print(f"iteration {iteration + 1}/{self.num_iterations}:

Iteration 7: Belt tour length = {self.belt.tour.length})

return self.belt.tour, self.belt.tour.length

~~... 0.0000000000000002~~

Código Ant:

def __init__(self, num_cities, alpha_beta):

self.alpha_beta = alpha_beta

self.tour = []

109

`self.down = length = 0; q = cities[0];`
`def congradient_solution(self):`
 `start_city = random.randint(0, self.num_cities - 1);`
 `self.tour = [start_city];`
 `self.down_length = 0; q = cities[0];`
 `visited_cities = set([self.tour[0]]);`
 `current_city = start_city;`
 `while len(self.tour) < self.num_cities:`
 `next_city = self.qso_dsp.select_next_city(cities, self.tour, current_city);`
 `self.tour.append(next_city);`
 `visited_cities.add(next_city);`
 `self.down_length += self.qso_dsp.cities[current_city].distance(cities[current_city], next_city);`
 `print("Visited cities: ", visited_cities);`
 `print("Tour length: ", self.down_length);`
`if __name__ == "__main__":`
 `cities = City((0, 0), city(1, 3), city(4, 3), city(6, 1));`
 `aeo = ArcTSP(cities, num_cities=10, num_iterations=100,`
 `alpha = 0.0, beta = 2.0, rho = 0.5, q0 = 0.9);`
 ~~`tour, best_tour, best_tour_length = aeo.run();`~~
 ~~`print("Tour: ", tour);`~~
 ~~`print("Best tour: ", best_tour);`~~
 ~~`print("Best tour length: ", best_tour_length);`~~
 ~~`print("Best tour length: ", best_tour_length);`~~

Code:

```
import random
import numpy as np
import math

class City:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def distance(self, city):
        return math.sqrt((self.x - city.x)**2 + (self.y - city.y)**2)

class ACO_TSP:
    def __init__(self, cities, num_ants, num_iterations, alpha=1.0, beta=2.0, rho=0.5,
                 q0=0.9):
        self.cities = cities
        self.num_ants = num_ants
        self.num_iterations = num_iterations
        self.alpha = alpha
        self.beta = beta
        self.rho = rho
        self.q0 = q0
        self.num_cities = len(cities)
        self.pheromone = np.ones((self.num_cities, self.num_cities))
        self.heuristic = np.zeros((self.num_cities, self.num_cities)) (inverse of distance)
        self.best_tour = None
        self.best_tour_length = float('inf')

    for i in range(self.num_cities):
        for j in range(i + 1, self.num_cities):
            dist = cities[i].distance(cities[j])
            self.heuristic[i][j] = 1.0 / dist if dist != 0 else 0
            self.heuristic[j][i] = self.heuristic[i][j]

    def select_next_city(self, current_city, visited_cities):
        probabilities = np.zeros(self.num_cities)
        total_pheromone = 0.0

        for city in range(self.num_cities):
```

```

if city not in visited_cities:
    pheromone = self.pheromone[current_city][city] ** self.alpha
    heuristic = self.heuristic[current_city][city] ** self.beta
    probabilities[city] = pheromone * heuristic
    total_pheromone += probabilities[city]

if total_pheromone == 0:
    return random.choice([city for city in range(self.num_cities) if city not in
visited_cities])

probabilities /= total_pheromone

if random.random() < self.q0:
    next_city = np.argmax(probabilities)
else:
    next_city = np.random.choice(self.num_cities, p=probabilities)

return next_city

def update_pheromone(self, ants):
    self.pheromone *= (1 - self.rho)

for ant in ants:
    pheromone_deposit = 1.0 / ant.tour_length
    for i in range(self.num_cities):
        current_city = ant.tour[i]
        next_city = ant.tour[(i + 1) % self.num_cities]
        self.pheromone[current_city][next_city] += pheromone_deposit
        self.pheromone[next_city][current_city] += pheromone_deposit

def run(self):
    for iteration in range(self.num_iterations):
        ants = [Ant(self.num_cities, self) for _ in range(self.num_ants)]
        for ant in ants:
            ant.construct_solution()

        self.update_pheromone(ants)

    for ant in ants:

```

```

        if ant.tour_length < self.best_tour_length:
            self.best_tour_length = ant.tour_length
            self.best_tour = ant.tour

        print(f"Iteration {iteration + 1}/{self.num_iterations}: Best Tour Length =
{self.best_tour_length}")

    return self.best_tour, self.best_tour_length

class Ant:
    def __init__(self, num_cities, aco_tsp):
        self.num_cities = num_cities
        self.aco_tsp = aco_tsp
        self.tour = []
        self.tour_length = 0.0

    def construct_solution(self):
        start_city = random.randint(0, self.num_cities - 1)
        self.tour = [start_city]
        self.tour_length = 0.0
        visited_cities = set(self.tour)
        current_city = start_city
        while len(self.tour) < self.num_cities:
            next_city = self.aco_tsp.select_next_city(current_city, visited_cities)
            self.tour.append(next_city)
            visited_cities.add(next_city)
            self.tour_length +=
                self.aco_tsp.cities[current_city].distance(self.aco_tsp.cities[next_city])
            current_city = next_city
            self.tour_length += self.aco_tsp.cities[self.tour[-1]].distance(self.aco_tsp.cities[self.tour[0]])

    if __name__ == "__main__":
        cities = [City(0, 0), City(1, 3), City(4, 3), City(6, 1), City(3, 0)]
        aco = ACO_TSP(cities=cities, num_ants=10, num_iterations=100, alpha=1.0,
beta=2.0, rho=0.5, q0=0.9)
        best_tour, best_tour_length = aco.run()

        print("\nBest tour found:", best_tour)
        print("Best tour length:", best_tour_length)

```

PROGRAM 4

CUCKOO SEARCH ALGORITHM

Algorithm:

Lab - 4

Mangal
Date 21/11/24
Page 12

Cuckoo Search (CS):

```

import numpy as np
from scipy.special import gamma
def objective(x):
    return np.sum(x**2)
def levy_flight(beta, dim):
    sigma = (gamma(1+beta) + np.sin(np.pi * beta / 2)) / (gamma((1+beta)/2) * beta * np.power(2, (beta/2))) * np.power(1/beta, 1/dim)
    w = np.random.normal(0, sigma, dim)
    v = np.random.normal(0, 1, dim)
    return w * np.abs(v)**(1/(1/beta))
def cuckoo_search(obj_func, dim, bounds, N=20, p=0.2, max_iter=100):
    ngs = np.random.uniform(bounds[0], bounds[1], (N, dim))
    fitness = np.array([obj_func(ne) for ne in ngs])
    best_ngt = ngs[np.argmax(fitness)]
    best_fitness = np.mean(fitness)

    for i in range(max_iter):
        new_ngt = ngs.copy()
        for i in range(N):
            step = levy_flight(1.5, dim)
            new_ngt[i] = ngs[i] + 0.01 * step
            new_ngt[i] = np.clip(new_ngt[i], bounds[0], bounds[1])
        new_fitness = np.array([obj_func(ne) for ne in new_ngt])
        if np.any(new_fitness < fitness):
            fitness = new_fitness
            best_ngt = new_ngt[np.argmin(fitness)]
            best_fitness = np.mean(fitness)
    return best_ngt, best_fitness

```

for i in range(n):

if np.random.rand() < p: rand.new_fitness[i]

fitness[i] = min(np.sum((rand.new_fitness[i] -

negt[i]) * new_negt[i])

fitness[i] = new_fitness[i] + negt[i]

for i in range(n):

best = negt[i] = np.argmin(fitness)

best = negt[i] = np.argmax(best + negt[i] * 1)

best = fitness[best] = fitness[best + negt[i]]

best = np.argmax(best * C) + best + 1) % n + 1

return best - negt, best + fitness[best]

dim = 10.0 * np.ones(10) * boundary_min * np.pi = r

boundary[BS, S] = boundary_min * np.pi = r

best - negt[best] = best + fitness[best] = min(r - objective.dim.

boundary),

best, r = best with sum of dim = constant. best

o/p : best = 1.0000000000000002, r = 6.34

(2) Abundant numbers, mathematics, gcd & phasor

Best, negt [2.519, 0.8100, 2.396, 1.6168, -1.860]

Best, fitness [32.124400000000004, 1.0000000000000002]

(Casting,

possible) minimum, gcd & phasor & boundary condition

(possible) min(r - objective.dim) & best

possible) (min(r - objective.dim)) & best

(min(r - objective.dim)) & best

: (min(r - objective.dim)) & best

(min(r - objective.dim)) & best

r = 3.141592653589793 * best + 6.34

best = 1.0000000000000002, min(r - objective.dim) = 6.34

min(r - objective.dim) = 6.34

best = 1.0000000000000002, min(r - objective.dim) = 6.34

best = 1.0000000000000002

Code:

```
import numpy as np
from scipy.special import gamma

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

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

def cuckoo_search(obj_func, dim, bounds, N=20, pa=0.25, max_iter=100):
    nests = np.random.uniform(bounds[0], bounds[1], (N, dim))
    fitness = np.array([obj_func(nest) for nest in nests])
    best_nest = nests[np.argmin(fitness)]
    best_fitness = np.min(fitness)
    for _ in range(max_iter):
        new_nests = np.copy(nests)
        for i in range(N):
            step = levy_flight(1.5, dim) # Lévy exponent 1.5
            new_nests[i] = nests[i] + 0.01 * step
            new_nests[i] = np.clip(new_nests[i], bounds[0], bounds[1])
        new_fitness = np.array([obj_func(nest) for nest in new_nests])
        for i in range(N):
            if np.random.rand() < pa and new_fitness[i] < fitness[i]:
                nests[i] = new_nests[i]
                fitness[i] = new_fitness[i]
        best_nest_idx = np.argmin(fitness)
        best_nest = nests[best_nest_idx]
        best_fitness = fitness[best_nest_idx]
    return best_nest, best_fitness

dim = 10
bounds = [-5, 5]
best_nest, best_fitness = cuckoo_search(objective, dim, bounds)

print(f"Best Nest: {best_nest}")
print(f"Best Fitness: {best_fitness}")
```

PROGRAM 5

GREY WOLF OPTIMIZATION

Algorithm:

Lab - 5

Manjal
Date: 28/10/2021
Page: 15

```
Grey wolf optimization (code)
Import numpy as np
def gwo(obj=fitness, dim=7, Search_agents=50, max_iter=1000):
    Alpha_pos = np.zeros(dim)
    Beta_pos = np.zeros(dim)
    Delta_pos = np.zeros(dim)
    Alpha_score = float("inf")
    Beta_score = float("inf")
    Delta_score = float("inf")
    position = np.random.uniform(lb, ub, (Search_agents), dim)
    for Iteration in range(max_iter):
        for i in range(Search_agents):
            position[i] = np.clip(position[i], lb, ub)
            fitness[i] = obj.function(position[i])
        if fitness[i] < Alpha_score:
            Alpha_pos = position[i].copy()
            Alpha_score = fitness[i]
        elif fitness[i] < Beta_score:
            Beta_pos = position[i].copy()
            Beta_score = fitness[i]
        elif fitness[i] < Delta_score:
            Delta_pos = position[i].copy()
            Delta_score = fitness[i]
        print(f"Iteration: {Iteration + 1}/{max_iter}, Best pos: {Alpha_pos} with score: {Alpha_score} at {Iteration + 1}/{max_iter} iteration, {i}th agent")
        for i in range(Search_agents):
            position[i] = np.clip(position[i], lb, ub)
            fitness[i] = obj.function(position[i])
        if fitness[i] < Alpha_score:
            Alpha_pos = position[i].copy()
            Alpha_score = fitness[i]
        elif fitness[i] < Beta_score:
            Beta_pos = position[i].copy()
            Beta_score = fitness[i]
        elif fitness[i] < Delta_score:
            Delta_pos = position[i].copy()
            Delta_score = fitness[i]
        print(f"Iteration: {Iteration + 1}/{max_iter}, Best pos: {Alpha_pos} with score: {Alpha_score} at {Iteration + 1}/{max_iter} iteration, {i}th agent")
```

for i, j in range(dim):
 m1, m2 = np.random.rand(), np.random.rand()
 A1, c1 = 2 * a * p1 + 0, 2 * m1 + p1
 D_alpha = abs(c1 * Alpha.pop[i]) - position
 x1 = Alpha.pop[i] - A1 * D_alpha, int.

m1, m2 = np.random.rand(), np.random.rand()
 A2, c2 = 2 * b * p2 + 0, 2 * m2 + p2
 D_beta = abs(c2 * Beta.pop[i]) - position
 x2 = Beta.pop[i] - A2 * D_beta
 (x1 + x2) / 2
 m1, m2 = np.random.rand(), np.random.rand()
 A3, c3 = 2 * d * p3 + 0, 2 * m3 + p3
 D_delta = abs(c3 * Delta.pop[i]) - position
 x3 = Delta.pop[i] - A3 * D_delta
 (x1 + x2 + x3) / 3
 position[i, j] = (x1 + x2 + x3) / 3

def sphere_function(x):

return np.sum((x * x))

Search-agent = 30 (no. of agents)

max_iter = 50 (no. of iterations)

lb, ub = -10, 10

best_pos, best_score = gwo(sphere_function, dim)

Search-agent, max_iter, lb, ub)

point("Best Position": 4, "Best Score": best_score)

point("Best Score": best_score)

o/p

~~Rept position : [-3.1825e-05, 3.4555e-05, 3.125e-05, 3.312e-05]~~

~~Rept : Scale : 6.07266e-09, height : 1.0~~

~~rotation of 3D object~~

Code:

```
import numpy as np

def gwo(obj_function, dim, search_agents, max_iter, lb, ub):
    Alpha_pos = np.zeros(dim)
    Beta_pos = np.zeros(dim)
    Delta_pos = np.zeros(dim)

    Alpha_score = float("inf")
    Beta_score = float("inf")
    Delta_score = float("inf")

    positions = np.random.uniform(lb, ub, (search_agents, dim))

    for iteration in range(max_iter):
        for i in range(search_agents):
            positions[i] = np.clip(positions[i], lb, ub)

            fitness = obj_function(positions[i])

            if fitness < Alpha_score:
                Alpha_score, Alpha_pos = fitness, positions[i].copy()
            elif fitness < Beta_score:
                Beta_score, Beta_pos = fitness, positions[i].copy()
            elif fitness < Delta_score:
                Delta_score, Delta_pos = fitness, positions[i].copy()

        print(f"Iteration {iteration + 1}/{max_iter}, Best Score: {Alpha_score:.6f}")

    a = 2 - iteration * (2 / max_iter) # Linearly decreases from 2 to 0

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

            A1, C1 = 2 * a * r1 - a, 2 * r2
            D_alpha = abs(C1 * Alpha_pos[j] - positions[i, j])
            X1 = Alpha_pos[j] - A1 * D_alpha

            r1, r2 = np.random.rand(), np.random.rand()
```

```

A2, C2 = 2 * a * r1 - a, 2 * r2
D_beta = abs(C2 * Beta_pos[j] - positions[i, j])
X2 = Beta_pos[j] - A2 * D_beta

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

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

return Alpha_pos, Alpha_score

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

dim = 5
search_agents = 30
max_iter = 50
lb, ub = -10, 10

best_position, best_score = gwo(sphere_function, dim, search_agents, max_iter, lb, ub)
print("Best Position:", best_position)
print("Best Score:", best_score)

```

PROGRAM 6

PARALLEL CELLULAR ALGORITHM

Algorithm:

Mangal
Date _____
Page 18

Parallel cellular Algorithms and programming.

Import numpy as np. It is written in line 1

```
def optimization_function(position):
    return position[0]**2 + position[1]**2
```

```
def initialize_parameters():
    grid_size = (10, 10)
    num_iterations = 100
    neighbourhood_size = 1
    return grid_size, num_iterations, neighbourhood_size
```

```
def initialize_population(grid_size):
    population = np.random.uniform(-10, 10, grid_size[1]*2)
    population = population.reshape(grid_size[1], 2)
    return population
```

```
def evaluate_fitness(population):
    fitness = np.zeros((population.shape[0], population.shape[1]))
    for i in range(population.shape[0]):
        for j in range(population.shape[1]):
            fitness[i, j] = optimization_function(population[i, j])
    return fitness
```

```
def update_state(population, fitness, neighbourhood_size):
    updated_population = np.copy(population)
    for i in range(population.shape[0]):
        for j in range(population.shape[1]):
            x_min = max(i - neighbourhood_size, 0)
            x_max = min(i + neighbourhood_size, population.shape[0] - 1)
            x = np.random.randint(x_min, x_max)
            if fitness[i, j] > fitness[x, j]:
                updated_population[i, j] = population[x, j]
    return updated_population
```

```

 $y_{\max} = \min(y + \text{neighborhood\_size} +), \text{population}.shape[1]$ 
best_neighbour = population[[y, :]]
best_fitness = fitness[y, :]
for x in range(x_min, x_max):
    for y in range(y_min, y_max):
        if fitness[x, y] < best_fitness:
            best_neighbour = population[x, :]
            best_fitness = fitness[x, :]
updated_population[y, :] = (population[y, :] + best_neighbour) / 2
return updated_population

```

```

def parallel_cellular_algorithm():
    grid_size, num_iterations, neighborhood_size =
    initialize_parameters()
    population = initialize_population(grid_size)
    best_solution = None
    best_fitness = float('inf')
    for iteration in range(num_iterations):
        fitness = evaluate_fitness(population)
        min_fitness < np.min(fitness)
        best_fitness = min_fitness
        best_solution = population[np.unravel_index(
            np.argmin(fitness), fitness.shape)]
        population = update_state(population, fitness,
        neighborhood_size)

```

point (f'' Bézout solution: \langle Bézout solution 3, Bézout Fitnug: (bezout_fitnug^*)).

return Bézout solution, Bézout fitnug;

If name == "main":
parallel + cellular algorithm;

Output: multihop cellular Bézout

Bézout Solution: [-8.02149781 -7.04216626].

Bézout Fitnug: [-30.12732754, 0.60046].

~~Fitnug~~

multihop, bivariate point

multihop, bivariate

multihop, bivariate, following path

multihop, Bézout solution, Bézout fitnug

multihop, Bézout solution

multihop, Bézout solution, Bézout fitnug

multihop, Bézout solution

multihop, Bézout solution

multihop, Bézout solution, Bézout fitnug

Code:

```
import numpy as np

def optimization_function(position):
    return position[0]**2 + position[1]**2

def initialize_parameters():
    grid_size = (10, 10)
    num_iterations = 100
    neighborhood_size = 1
    return grid_size, num_iterations, neighborhood_size

def initialize_population(grid_size):
    population = np.random.uniform(-10, 10, (grid_size[0], grid_size[1], 2))
    return population

def evaluate_fitness(population):
    fitness = np.zeros((population.shape[0], population.shape[1]))
    for i in range(population.shape[0]):
        for j in range(population.shape[1]):
            fitness[i, j] = optimization_function(population[i, j])
    return fitness

def update_states(population, fitness, neighborhood_size):
    updated_population = np.copy(population)
    for i in range(population.shape[0]):
        for j in range(population.shape[1]):
            x_min = max(i - neighborhood_size, 0)
            x_max = min(i + neighborhood_size + 1, population.shape[0])
            y_min = max(j - neighborhood_size, 0)
            y_max = min(j + neighborhood_size + 1, population.shape[1])

            best_neighbor = population[i, j]
            best_fitness = fitness[i, j]

            for x in range(x_min, x_max):
                for y in range(y_min, y_max):
                    if fitness[x, y] < best_fitness:
                        best_neighbor = population[x, y]
                        best_fitness = fitness[x, y]
```

```

updated_population[i, j] = (population[i, j] + best_neighbor) / 2

return updated_population

def parallel_cellular_algorithm():
    grid_size, num_iterations, neighborhood_size = initialize_parameters()

    population = initialize_population(grid_size)

    best_solution = None
    best_fitness = float('inf')

    for iteration in range(num_iterations):
        fitness = evaluate_fitness(population)

        min_fitness = np.min(fitness)
        if min_fitness < best_fitness:
            best_fitness = min_fitness
            best_solution = population[np.unravel_index(np.argmin(fitness),
                fitness.shape)]

        population = update_states(population, fitness, neighborhood_size)

        print(f"Iteration {iteration + 1}: Best Fitness = {best_fitness}")
        print(f"Best Solution: {best_solution}, Best Fitness: {best_fitness}")
    return best_solution, best_fitness

if __name__ == "__main__":
    parallel_cellular_algorithm()

```

PROGRAM 7

GENE EXPRESSION ALGORITHM

Algorithm:

Manjal
Date _____
Page 21

Optimization via Gene Expression Algorithm

```
Import numpy as np

def optimization_function(solution):
    return solution[0]**2 + solution[1]**2

def initialize_parameters():
    population_size = 50
    num_genes = 2
    mutation_rate = 0.1
    crossover_rate = 0.8
    num_generations = 100
    return population_size, num_genes, mutation_rate, crossover_rate, num_generations

def initialize_population(population_size, num_genes):
    return np.random.uniform(-10, 10, (population_size, num_genes))

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

def select_parents(population, fitness):
    probabilities = 1 / (fitness + 1)
    probabilities /= probabilities.sum()
    indices = np.random.choice(len(population), size=bn, p=probabilities)
    return population[indices]

# Main loop
for generation in range(num_generations):
    population = initialize_population(population_size, num_genes)
    fitness = evaluate_fitness(population)
    parents = select_parents(population, fitness)
    # Crossover and mutation steps
    # ...
    # Selection for next generation
    # ...
    # Print progress
    print(f'Generation {generation}: Mean fitness = {np.mean(fitness)}')
```

def create_over (parents, crossover_rate):

offspring = []

for i in range (0, len (parents), 2):

~~off~~ for i+1 < len (parents) and np.random.rand

< crossover_rate: i, i+1, parents[i], parents[i+1]

point = np.random.randint (0, len (parents), 1)

offspring[0] = np.concatenate ([parents[i][0:point],

parents[i+1][point:]])

offspring[1] = np.concatenate ([parents[i+1][0:point],

parents[i][point:]])

offspring = extend ([offspring[0], offspring[1]])

edge:

offspring = extend ([parents[i], parents[i+1]])

if i+1 < len (parents): edge = parents[i+1]

return np.array (offspring)

def mutate (offspring, mutation_rate):

for individual in offspring:

if np.random.randint (0, 1) < mutation_rate:

gene = np.random.randint (individual.size)

if random.randint (0, 1) <= np.random.uniform (0, 1):

individual[gene] = np.random.randint (0, 1)

return offspring

def genetic_exploration_algorithm ():

population_size, num_genes, mutation_rate,

initialize_population (population_size,

num_genes)

best_solution = initialize_population (population_size,

num_genes)

best_fitness = float ('inf')

for generation in range (num_generations):

$\text{fitness} = \text{evaluate_fitness}(\text{population})$

$\min \text{fitness}, \text{idx} = \text{np.argmin}(\text{fitness})$

If $\text{fitness}[\text{min_fitness_idx}] < \text{best_fitness}$:

$\text{best_fitness} = \text{fitness}[\text{min_fitness_idx}]$

$\text{best_solution} = \text{population}[\text{min_fitness_idx}]$,

$\text{parents} = \text{select_parents}(\text{population}, \text{fitness})$

$\text{offspring} = \text{crossover}(\text{parents}, \text{crossover_rate})$

$\text{population} = \text{mutate}(\text{offspring}, \text{mutation_rate})$

$\text{population} = \text{gene_expression}(\text{population})$

$\text{point}(\text{f" generation}, \text{generation} + 1) : \text{Best}$

$\text{Fitness} = \{\text{best_fitness}\})$

$\text{points} (" \text{Best Solution} : \text{best_solution} \text{, Best}$

$\text{Fitness} : \{\text{best_fitness}\})$

return best solution, best fitness.

~~def~~

$\text{f_name} = " \text{main}"$

$\text{gene_expression_algorithm}()$

~~output:~~

Best solution: 70.00434829 0.00143412,

Best Fitness: 2.096871137428934e-05.

Code:

```
import numpy as np

def optimization_function(solution):
    return solution[0]**2 + solution[1]**2

def initialize_parameters():
    population_size = 50
    num_genes = 2
    mutation_rate = 0.1
    crossover_rate = 0.8
    num_generations = 100
    return population_size, num_genes, mutation_rate, crossover_rate, num_generations

def initialize_population(population_size, num_genes):
    return np.random.uniform(-10, 10, (population_size, num_genes))

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

def select_parents(population, fitness):
    probabilities = 1 / (fitness + 1e-6)
    probabilities /= probabilities.sum()
    indices = np.random.choice(len(population), size=len(population), p=probabilities)
    return population[indices]

def crossover(parents, crossover_rate):
    offspring = []
    for i in range(0, len(parents), 2):
        if i + 1 < len(parents) and np.random.rand() < crossover_rate:
            point = np.random.randint(1, parents.shape[1])
            offspring1 = np.concatenate((parents[i, :point], parents[i + 1, point:]))
            offspring2 = np.concatenate((parents[i + 1, :point], parents[i, point:]))
            offspring.extend([offspring1, offspring2])
        else:
            offspring.extend([parents[i], parents[i + 1] if i + 1 < len(parents) else parents[i]])
    return np.array(offspring)

def mutate(offspring, mutation_rate):
```

```

for individual in offspring:
    if np.random.rand() < mutation_rate:
        gene = np.random.randint(individual.size)
        individual[gene] += np.random.normal(0, 1)

def gene_expression(population):
    return population

def gene_expression_algorithm():
    population_size, num_genes, mutation_rate, crossover_rate, num_generations =
    initialize_parameters()

    population = initialize_population(population_size, num_genes)

    best_solution = None
    best_fitness = float('inf')

    for generation in range(num_generations):
        fitness = evaluate_fitness(population)

        min_fitness_idx = np.argmin(fitness)
        if fitness[min_fitness_idx] < best_fitness:
            best_fitness = fitness[min_fitness_idx]
            best_solution = population[min_fitness_idx]

        parents = select_parents(population, fitness)
        offspring = crossover(parents, crossover_rate)
        population = mutate(offspring, mutation_rate)
        population = gene_expression(population)
        print(f"Generation {generation + 1}: Best Fitness = {best_fitness}")
    print(f"Best Solution: {best_solution}, Best Fitness: {best_fitness}")
    return best_solution, best_fitness

if __name__ == "__main__":
    gene_expression_algorithm()

```