VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB RECORD

Bio Inspired Systems (23CS5BSBIS)

Submitted by

SriKrishna Pejathaya P S(1BM22CS290)

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 **SriKrishna Pejathaya P S(1BM22CS290),** who is a bonafide student of **B.M.S. College of Engineering.** It is in partial fulfillment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the Visvesvaraya Technological University, Belgaum. The Lab report has been approved as it satisfies the academic requirements of the above mentioned subject and the work prescribed for the said degree.

Sheetal V A Assistant Professor Department of CSE, BMSCE	**Dr. Kavitha Sooda** Professor & HOD Department of CSE, BMSCE

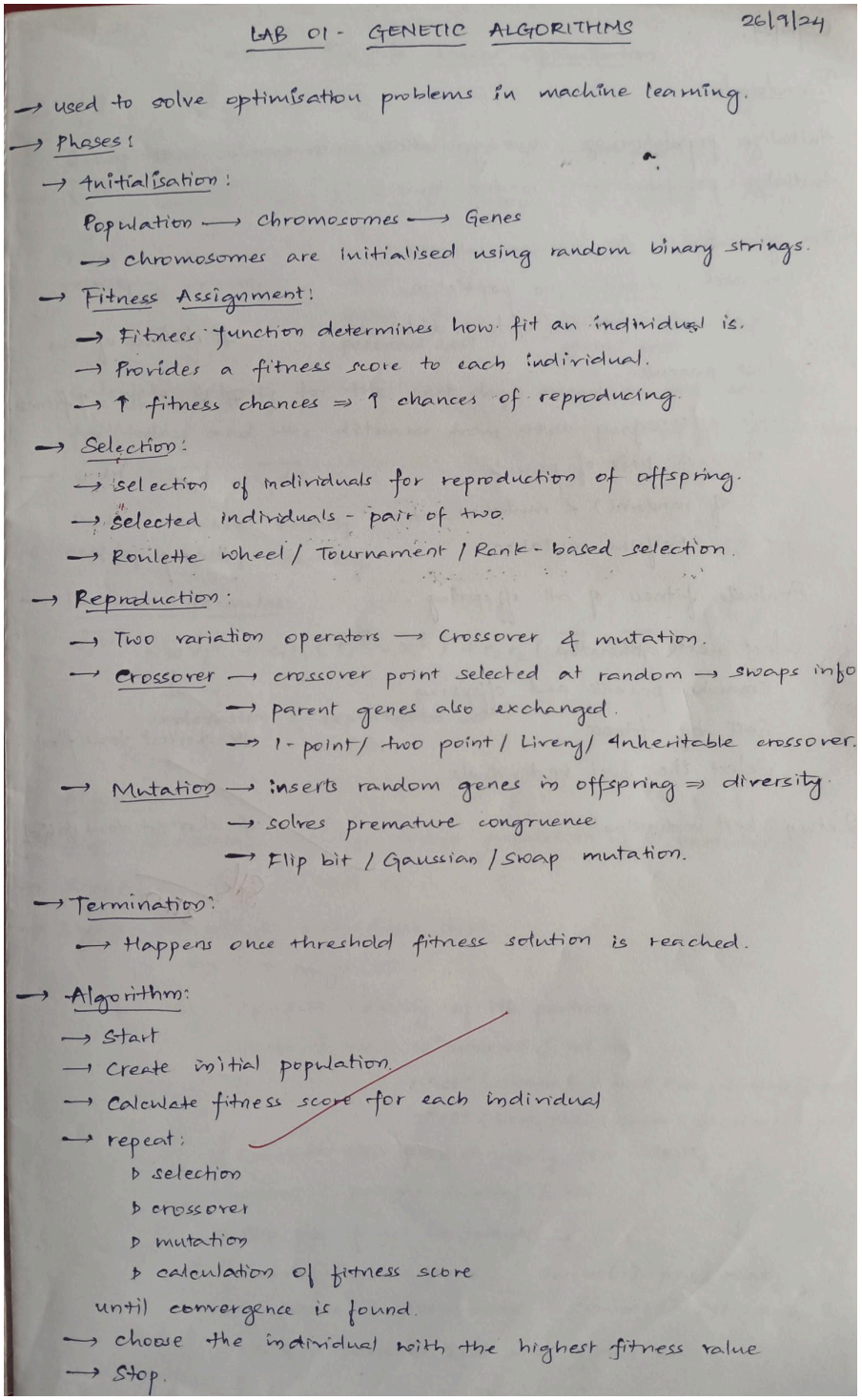
Index

Github Link :srikrishna-ps/BIS

Program 1

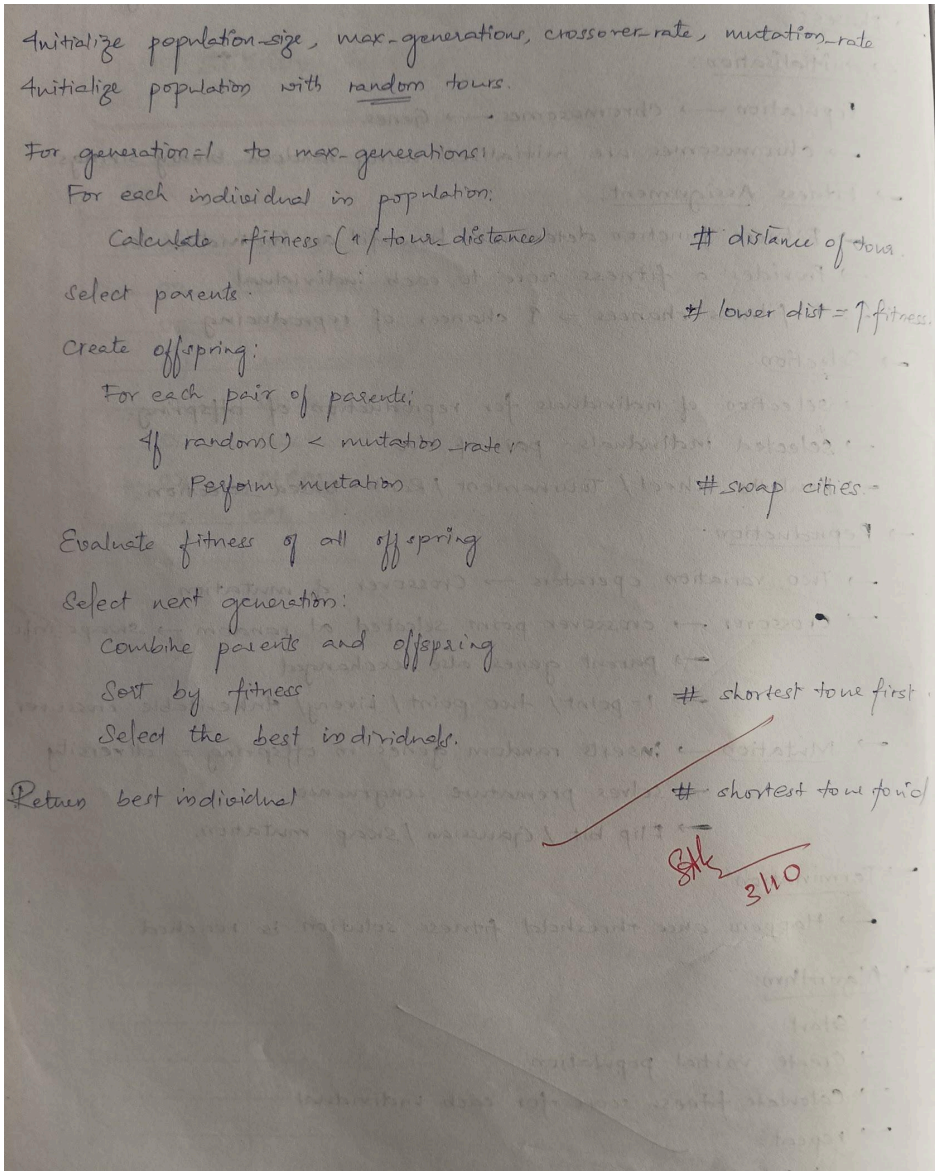
Problem Statement : Solve the Traveling Salesman Problem (TSP) using a Genetic Algorithm to find the shortest possible route that visits a set of cities exactly once and returns to the starting point.

Algorithm:



LAB 01 - GENETIC ALGORITHMS 26/9/24

→ used to solve optimisation problems in machine learning.
→ Phases:
  → Initialisation:
    Population → chromosomes → Genes
    → chromosomes are initialised using random binary strings.
  → Fitness Assignment:
    → Fitness function determines how fit an individual is.
    → Provides a fitness score to each individual.
    → ↑ fitness chances ⇒ ↑ chances of reproducing.
  → Selection:
    → selection of individuals for reproduction of offspring.
    → selected individuals - pair of two.
    → Roulette wheel / Tournament / Rank-based selection.
  → Reproduction:
    → Two variation operators → Crossover & mutation.
    → Crossover → crossover point selected at random → swaps info
      → parent genes also exchanged.
      → 1-point / two point / Livery / Inheritable crossover.
    → Mutation → inserts random genes in offspring ⇒ diversity
      → solves premature congruence
      → Flip bit / Gaussian / Swap mutation.
  → Termination:
    → Happens once threshold fitness solution is reached.
→ Algorithm:
  → Start
  → Create initial population.
  → Calculate fitness score for each individual
  → repeat:
    ▷ selection
    ▷ crossover
    ▷ mutation
    ▷ calculation of fitness score
  until convergence is found.
  → choose the individual with the highest fitness value
  → Stop.



Code:

```
#Travelling Salesman Problem

import random
import numpy as np

def get_cities():
    return [
        (0, 0), (10, 20), (20, 15), (30, 10), (40, 25),
        (50, 5), (60, 35), (70, 10), (80, 50), (90, 40),
        (10, 70), (20, 90), (30, 80), (40, 60), (50, 50),
        (60, 90), (70, 70), (80, 30), (90, 10), (100, 20)
    ]
```

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


def total_distance(path, cities):
    return sum(distance(cities[path[i]], cities[path[i + 1]]) for i in
range(len(path) - 1)) + distance(cities[path[-1]], cities[path[0]])


def initial_population(pop_size, num_cities):
    return [random.sample(range(num_cities), num_cities) for _ in range(pop_size)]

def fitness_function(path, cities):
    return 1 / total_distance(path, cities)

def select_population(population, cities, num_selected):
    fitness = [fitness_function(path, cities) for path in population]
    probabilities = [f / sum(fitness) for f in fitness]
    selected_indices = np.random.choice(len(population), size=num_selected,
replace=True, p=probabilities)
    return [population[i] for i in selected_indices]

def crossover(parent1, parent2):
    start, end = sorted(random.sample(range(len(parent1)), 2))
    child = [-1] * len(parent1)
    child[start:end] = parent1[start:end]
    pointer = end
    for city in parent2:
        if city not in child:
            if pointer >= len(child):
                pointer = 0
            child[pointer] = city
            pointer += 1
    return child

def mutate(path, mutation_rate):
    for i in range(len(path)):
        if random.random() < mutation_rate:
            j = random.randint(0, len(path) - 1)
            path[i], path[j] = path[j], path[i]
    return path

def genetic_algorithm(cities, pop_size=100, generations=2000, mutation_rate=0.01):
    num_cities = len(cities)
    population = initial_population(pop_size, num_cities)
    best_path, best_distance = None, float('inf')
```

```
    for generation in range(generations):
        population = select_population(population, cities, pop_size)
        next_generation = []

        for _ in range(len(population) // 2):
            parent1, parent2 = random.sample(population, 2)
            child1 = mutate(crossover(parent1, parent2), mutation_rate)
            child2 = mutate(crossover(parent2, parent1), mutation_rate)
            next_generation.extend([child1, child2])

        population = next_generation

        for path in population:
            dist = total_distance(path, cities)
            if dist < best_distance:
                best_path, best_distance = path, dist

        if generation % 50 == 0:
            print(f"Generation {generation}: Best Distance = {best_distance:.2f}")

    return best_path, best_distance

if __name__ == "__main__":
    cities = get_cities()
    best_path, best_distance = genetic_algorithm(cities)

    print("Best Path:", best_path)
    print("Best Distance:", best_distance)
```

Output:

```
Generation 0: Best Distance = 879.17
Generation 50: Best Distance = 729.58
Generation 100: Best Distance = 710.47
Generation 150: Best Distance = 683.69
Generation 200: Best Distance = 683.69
Generation 250: Best Distance = 683.69
Generation 300: Best Distance = 683.69
Generation 350: Best Distance = 683.69
Generation 400: Best Distance = 683.69
Generation 450: Best Distance = 683.69
Generation 500: Best Distance = 683.69
Generation 550: Best Distance = 683.69
Generation 600: Best Distance = 683.69
Generation 650: Best Distance = 683.69
Generation 700: Best Distance = 683.69
Generation 750: Best Distance = 659.36
Generation 800: Best Distance = 659.36
Generation 850: Best Distance = 659.36
Generation 900: Best Distance = 659.36
Generation 950: Best Distance = 659.36
Generation 1000: Best Distance = 659.36
Generation 1050: Best Distance = 659.36
Generation 1100: Best Distance = 659.36
Generation 1150: Best Distance = 659.36
Generation 1200: Best Distance = 659.36
...
Generation 1900: Best Distance = 659.36
Generation 1950: Best Distance = 632.79
Best Path: [11, 10, 8, 14, 13, 12, 15, 16, 9, 7, 17, 18, 19, 4, 0, 2, 1, 3, 5, 6]
Best Distance: 632.7871266164108
```

Program 2

Problem Statement : Solve a minimization problem using the Particle Swarm Optimization (PSO) algorithm to find the point in a multi-dimensional space that is closest to the origin.

Algorithm:

25/10/2024

Lab 03 - Particle Swarm Optimisation

→ $\vec{X_i^d} = [x_i^d, y_i^d, z_i^d \dots]$

Every particle record their 'x' and 'y' coordinates.

→ $\vec{v_i^{d+1}} = 2r_1 \vec{v_i^d} + 2r_2 (\vec{p_i^d} - \vec{x_i^d}) + 2r_3 (\vec{G^d} - \vec{x_i^d})$

- $\vec{v_i^{d+1}}$: Next velocity
- $\vec{v_i^d}$: current velocity
- $\vec{p_i^d}$: Personal best → Distance to personal best
- $\vec{G^d}$: Global best → Distance to global best

Calculate velocity for the next day based on the velocity for today and the distances from their personal & global best

→ $\vec{x_i^{d+1}} = \vec{x_i^d} + \vec{v_i^{d+1}}$

Position next day = Position today + velocity next day.

→ Parameter formulas:

$\vec{v_i^{t+1}} = w\vec{v_i^t} + c_1 r_1 (\vec{p_i^t} - \vec{x_i^t}) + c_2 r_2 (\vec{G^t} - \vec{x_i^t})$

↓

exploration & exploitation.

(↑) (↓).

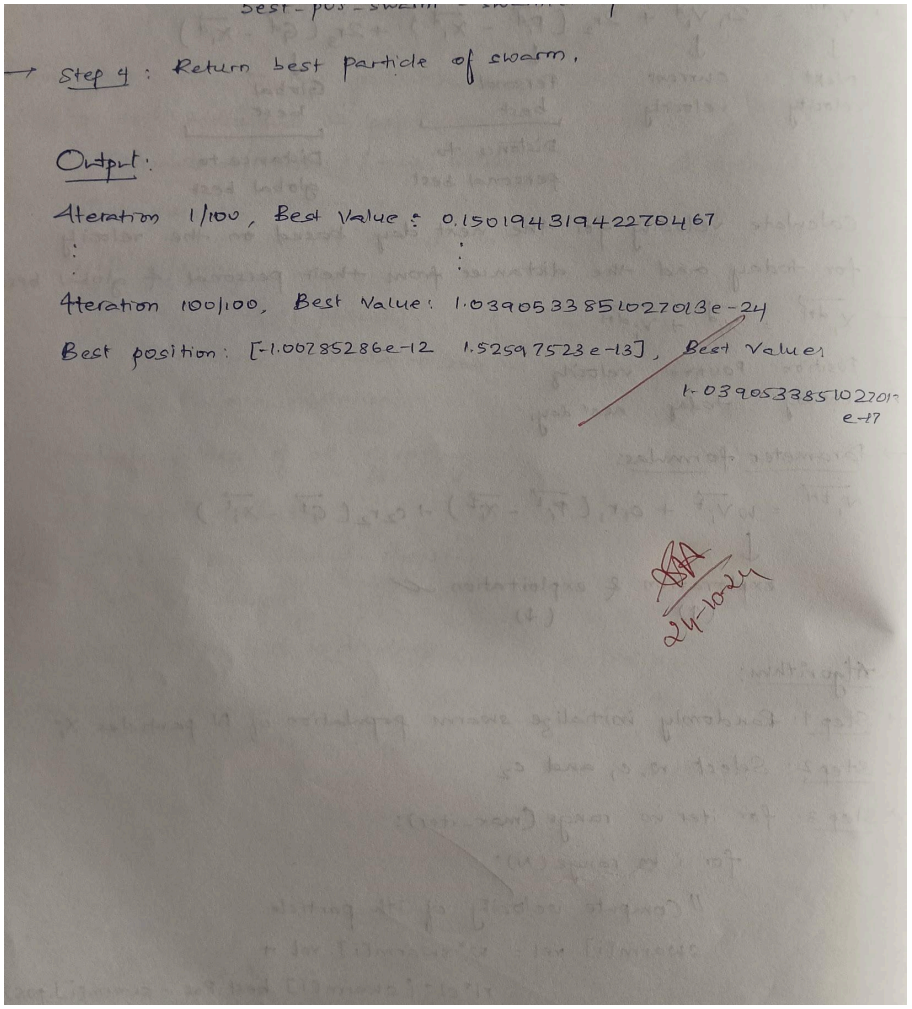
Algorithm:

→ Step 1: Randomly initialize swarm population of N particles X_i

→ Step 2: Select w, c_1 and c_2.

→ Step 3:

```
for iter in range(max_iter):
    for i in range(N):
        // Compute velocity of ith particle
        swarm[i].vel = w*swarm[i].vel +
                       r1*c1*(swarm[i].best Pos - swarm[i].pos)+
                       r2*c2*(best_pos_swarm - swarm[i].pos).
        // Compute new position using new velocity.
        swarm[i].pos += swarm[i].vel
        // clip pos if not in range
        if swarm[i].pos < minx :   swarm[i].pos = minx
        elif swarm[i].pos > maxx:  swarm[i].pos = maxx
```



Code:

```
#Minimization problem

import numpy as np

def distance_from_origin(x):
    return np.sqrt(np.sum(np.square(x)))

def particle_swarm_optimization(
    cost_function,
    dim,
    bounds,
    num_particles=20,
    max_iter=1000,
    w=0.8,
    c1=1.5,
    c2=1.2
):
    particles = np.random.uniform(bounds[0], bounds[1], (num_particles, dim))
    velocities = np.random.uniform(-1, 1, (num_particles, dim))
```

```
    personal_best_positions = np.copy(particles)
    personal_best_scores = np.array([cost_function(p) for p in particles])

    global_best_position = personal_best_positions[np.argmin(personal_best_scores)]
    global_best_score = np.min(personal_best_scores)

    for iteration in range(max_iter):
        for i in range(num_particles):
            r1, r2 = np.random.rand(dim), np.random.rand(dim)
            velocities[i] = (
                w * velocities[i] +
                c1 * r1 * (personal_best_positions[i] - particles[i]) +
                c2 * r2 * (global_best_position - particles[i])
            )

            particles[i] += velocities[i]

            particles[i] = np.clip(particles[i], bounds[0], bounds[1])

            cost = cost_function(particles[i])

            if cost < personal_best_scores[i]:
                personal_best_scores[i] = cost
                personal_best_positions[i] = particles[i]


        current_global_best_score = np.min(personal_best_scores)
        if current_global_best_score < global_best_score:
            global_best_score = current_global_best_score
            global_best_position =
personal_best_positions[np.argmin(personal_best_scores)]

        if iteration % 10 == 0 or iteration == max_iter - 1:
            print(f"Iteration {iteration}: Global Best Score =
{global_best_score:.4f}")

    return global_best_position, global_best_score


if __name__ == "__main__":

    dimensions = 4
    bounds = [-100, 100]


    best_position, best_score = particle_swarm_optimization(
```

```
        cost_function=distance_from_origin,
        dim=dimensions,
        bounds=bounds,
        num_particles=10,
        max_iter=100
    )

    print("\nOptimal Solution:")
    print(f"Best Position: {best_position}")
    print(f"Best Distance from Origin: {best_score:.4f}")
```

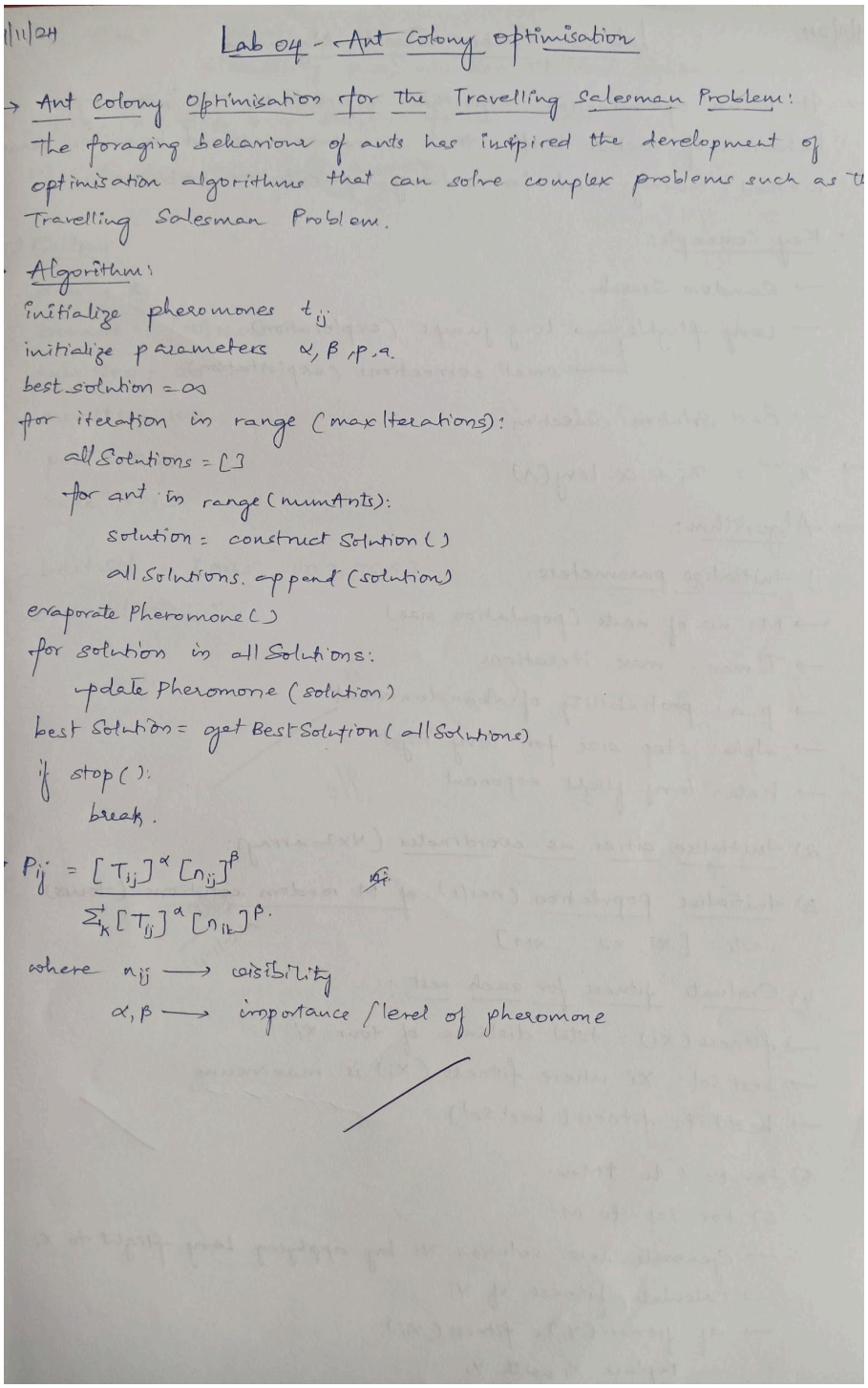
Output:

```
Iteration 0: Global Best Score = 69.5116
Iteration 10: Global Best Score = 13.9884
Iteration 20: Global Best Score = 5.9542
Iteration 30: Global Best Score = 1.3085
Iteration 40: Global Best Score = 0.9874
Iteration 50: Global Best Score = 0.8093
Iteration 60: Global Best Score = 0.2604
Iteration 70: Global Best Score = 0.2038
Iteration 80: Global Best Score = 0.0812
Iteration 90: Global Best Score = 0.0669
Iteration 99: Global Best Score = 0.0231

Optimal Solution:
Best Position: [-2.48618035e-03  1.66706220e-02 -2.85253096e-05 -1.58113410e-02]
Best Distance from Origin: 0.0231
```

Program 3

Problem Statement : Implement an Ant Colony Optimization (ACO) algorithm to solve the Traveling Salesman Problem (TSP), where the goal is to find the shortest possible path that visits all cities exactly once and returns to the starting city. The algorithm should utilize pheromone trails and heuristic information to guide the search efficiently.

Algorithm :



11/24

Lab 04 - Ant Colony optimisation

→ Ant Colony Optimisation for the Travelling Salesman Problem:
The foraging behaviour of ants has inspired the development of optimisation algorithms that can solve complex problems such as t
Travelling Salesman Problem.

Algorithm:
initialize pheromones t_{ij}
initialize parameters α, β, ρ, q.
best_solution = ∞
for iteration in range (maxIterations):
    allSolutions = []
    for ant in range (numAnts):
        solution = construct Solution ()
        allSolutions. append (solution)
evaporate Pheromone ()
for solution in allSolutions:
    update Pheromone (solution)
best Solution = get Best Solution (allSolutions)
if stop ():
    break.

$$P_{ij} = \frac{[T_{ij}]^{\alpha} [n_{ij}]^{\beta}}{\sum_k [T_{ij}]^{\alpha} [n_{ik}]^{\beta}}$$

where n_{ij} ⟶ visibility
α, β ⟶ importance / level of pheromone

Code:

```
#Traveling Salesman

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

NUM_ANTS = 50
ALPHA = 1.0
BETA = 2.0
RHO = 0.1
Q = 100
MAX_ITER = 100

def define_cities():
    return np.array([
        [10, 10],
        [20, 20],
        [50, 30],
        [40, 40],
        [80, 60],
        [15, 15],
        [125, 90],
        [100, 150],
        [200, 200],
        [180, 250],
        [250, 30],
        [300, 100],
        [220, 150],
        [60, 250],
        [120, 190],
    ])

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

def initialize_pheromone_matrix(num_cities):
    pheromone_matrix = np.ones((num_cities, num_cities))
    np.fill_diagonal(pheromone_matrix, 0)
```

```
    return pheromone_matrix

def calculate_tour_length(tour, dist_matrix):
    length = 0
    for i in range(len(tour) - 1):
        length += dist_matrix[tour[i], tour[i + 1]]
    length += dist_matrix[tour[-1], tour[0]]
    return length

def construct_solution(num_cities, pheromone_matrix, dist_matrix):
    tour = [random.randint(0, num_cities - 1)]
    visited = set(tour)
    while len(tour) < num_cities:
        current_city = tour[-1]
        probabilities = []
        for next_city in range(num_cities):
            if next_city not in visited:
                pheromone = pheromone_matrix[current_city, next_city] ** ALPHA
                distance = (1.0 / dist_matrix[current_city, next_city]) ** BETA
                probabilities.append(pheromone * distance)
            else:
                probabilities.append(0)
        total_prob = sum(probabilities)
        probabilities = [p / total_prob for p in probabilities]
        next_city = np.random.choice(range(num_cities), p=probabilities)
        tour.append(next_city)
        visited.add(next_city)
    return tour

def update_pheromone(pheromone_matrix, all_tours, dist_matrix, best_tour):
    pheromone_matrix *= (1 - RHO)
    for tour in all_tours:
        tour_length = calculate_tour_length(tour, dist_matrix)
        for i in range(len(tour) - 1):
            pheromone_matrix[tour[i], tour[i + 1]] += Q / tour_length
        pheromone_matrix[tour[-1], tour[0]] += Q / calculate_tour_length(tour,
dist_matrix)
    best_length = calculate_tour_length(best_tour, dist_matrix)
    for i in range(len(best_tour) - 1):
        pheromone_matrix[best_tour[i], best_tour[i + 1]] += Q / best_length
    pheromone_matrix[best_tour[-1], best_tour[0]] += Q / best_length

def ant_colony_optimization(cities, dist_matrix, pheromone_matrix, max_iter):
    best_tour = None
    best_tour_length = float('inf')
    for iteration in range(max_iter):
        all_tours = []
```

```
        for _ in range(NUM_ANTS):
            tour = construct_solution(len(cities), pheromone_matrix, dist_matrix)
            all_tours.append(tour)
            tour_length = calculate_tour_length(tour, dist_matrix)
            if tour_length < best_tour_length:
                best_tour = tour
                best_tour_length = tour_length
        update_pheromone(pheromone_matrix, all_tours, dist_matrix, best_tour)
    return best_tour, best_tour_length

def plot_tour(cities, best_tour):
    tour_cities = cities[best_tour]
    plt.plot(tour_cities[:, 0], tour_cities[:, 1], 'bo-', markersize=6)
    plt.scatter(cities[:, 0], cities[:, 1], color='red', marker='x')
    for i, city in enumerate(cities):
        plt.text(city[0], city[1], f'{i}', fontsize=12, ha='right')
    plt.title("ACO TSP Solution")
    plt.show()

if __name__ == "__main__":
    cities = define_cities()
    dist_matrix = compute_distance_matrix(cities)
    pheromone_matrix = initialize_pheromone_matrix(len(cities))
    best_tour, best_tour_length = ant_colony_optimization(cities, dist_matrix,
pheromone_matrix, MAX_ITER)
    print(f"Best tour length: {best_tour_length:.2f}")
    plot_tour(cities, best_tour)
```

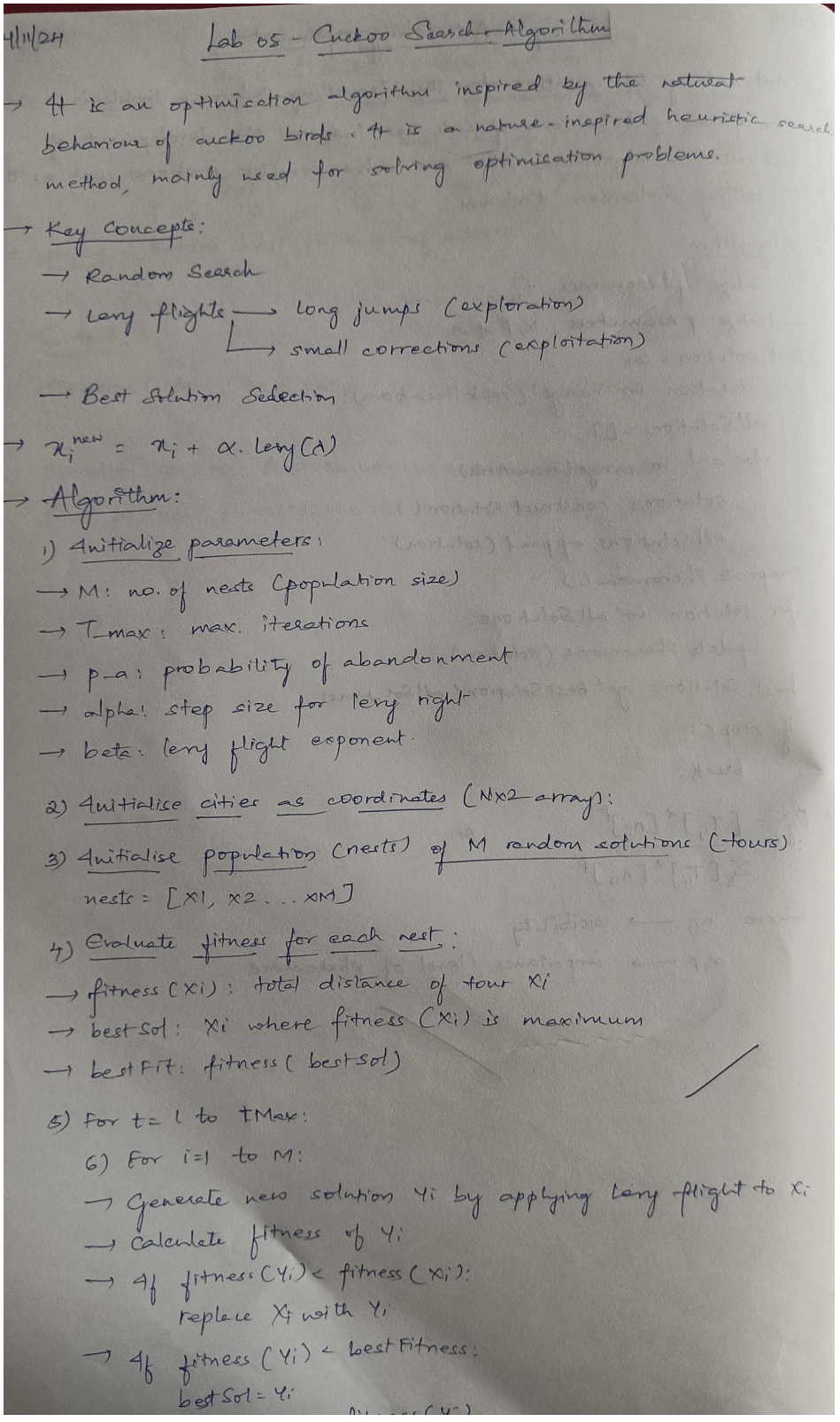
Output :

```
Best tour length: 985.68
```

Program 4

Problem Statement : Optimize traffic light timings at an intersection using the Cuckoo Search Algorithm to minimize total vehicle waiting time. The optimization ensures green light durations adhere to specified bounds for efficient solution exploration and refinement.

Algorithm :



4/11/24

Lab 05 - Cuckoo Search Algorithm

→ It is an optimisation algorithm inspired by the natural behaviour of cuckoo birds. It is a nature-inspired heuristic search method, mainly used for solving optimisation problems.

→ Key Concepts:
- → Random Search
- → Levy flights → long jumps (exploration)
  - → small corrections (exploitation)
- → Best Solution Selection

→ $x_i^{new} = x_i + \alpha \cdot Levy(\lambda)$

→ Algorithm:

1) Initialize parameters:
- → M: no. of nests (population size)
- → T_max: max. iterations
- → p_a: probability of abandonment
- → alpha: step size for levy right
- → beta: levy flight exponent

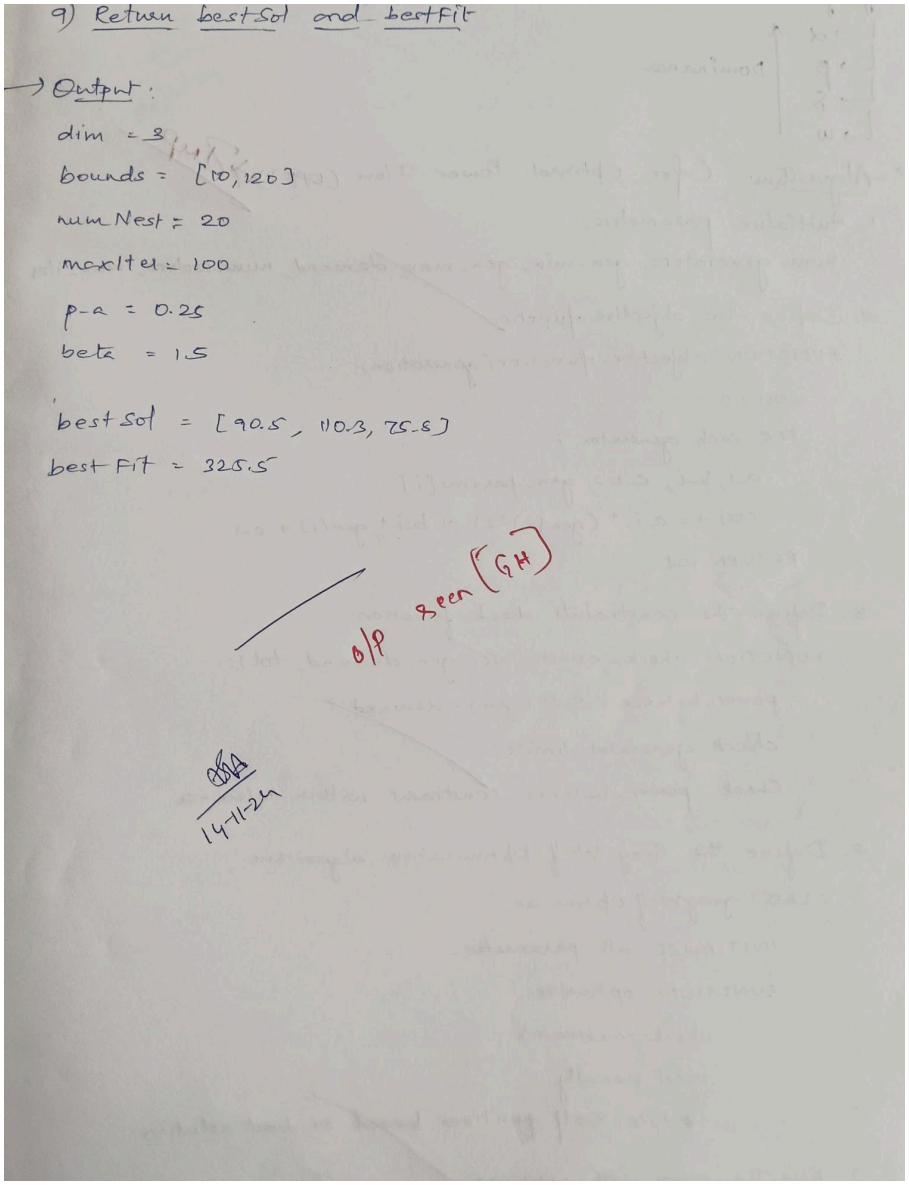
2) Initialise cities as coordinates (Nx2 array):

3) Initialise population (nests) of M random solutions (tours):
nests = [x1, x2 ... xM]

4) Evaluate fitness for each nest:
- → fitness(xi): total distance of tour xi
- → best sol: xi where fitness(xi) is maximum
- → best Fit: fitness(best sol)

5) For t = 1 to tMax:

6) For i=1 to M:
- → Generate new solution yi by applying levy flight to xi
- → Calculate fitness of yi
- → If fitness(yi) < fitness(xi):
  replace xi with yi
- → If fitness(yi) < best Fitness:
  best Sol = yi



Code:

```
#Traffic Light Optimisation

import numpy as np
from scipy.special import gamma

def fitness_function(x):
    waiting_times = np.array([10 + (x[i] ** 1.5) / 50 for i in range(len(x))])
    total_waiting_time = np.sum(waiting_times)
    return total_waiting_time
```

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


def cuckoo_search(dim, bounds, num_nests, max_iter, p_a=0.1, Lambda=1.5):
    nests = np.random.uniform(bounds[0], bounds[1], (num_nests, dim))
    fitness = np.array([fitness_function(nest) for nest in nests])

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

    for iter in range(max_iter):
        new_nests = np.copy(nests)

        for i in range(num_nests):
            step = levy_flight(dim, Lambda) * 0.1
            new_nests[i] = nests[i] + step
            new_nests[i] = np.clip(new_nests[i], bounds[0], bounds[1])

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

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

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

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

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

    return best_nest, best_fitness
```

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

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

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

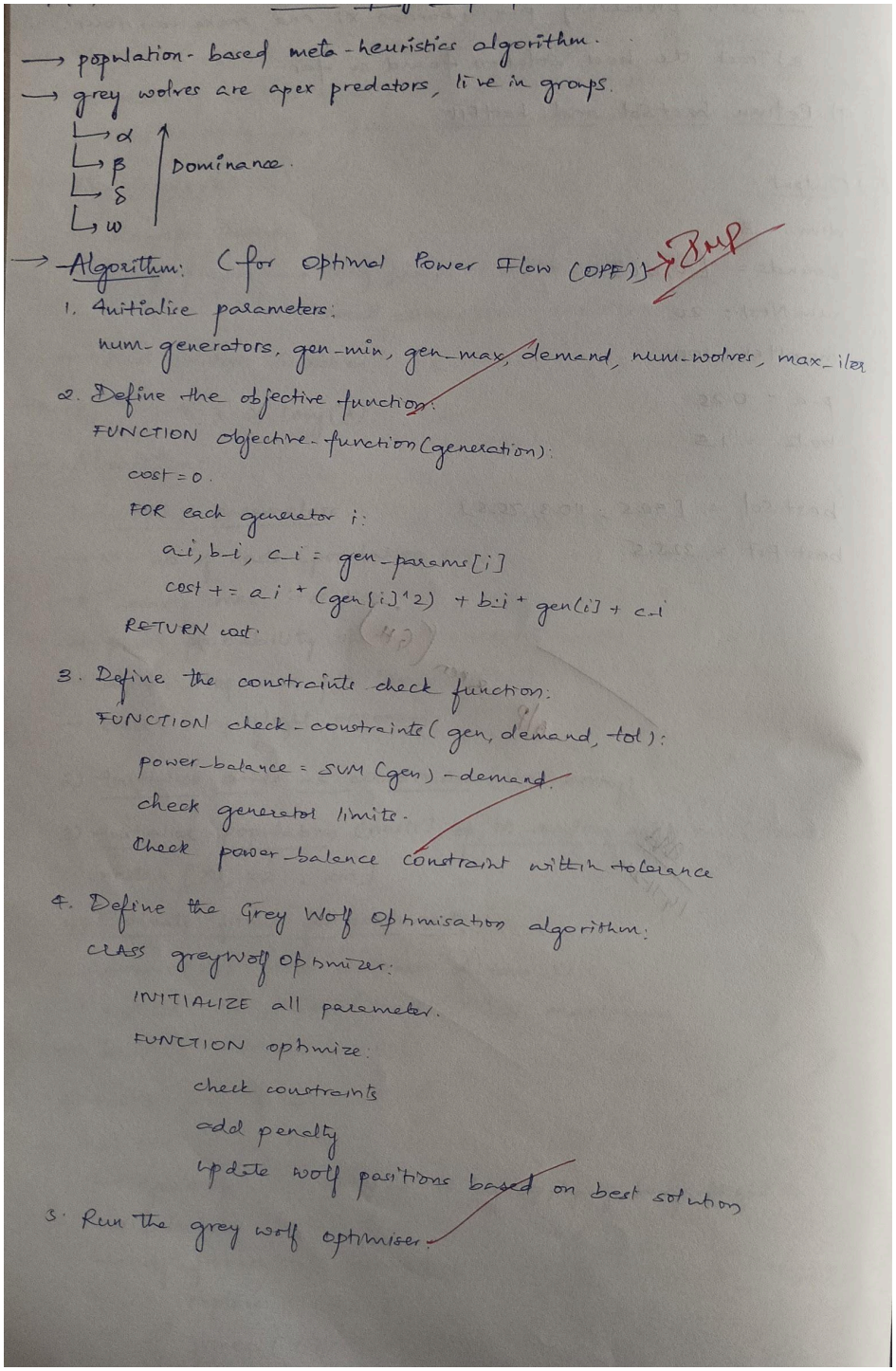
Output:

```
Green Light Timings (seconds): [14.04385594 13.38439428 33.44172735]
Best Fitness Value (Total Waiting Time): 35.89970997182197
```

Program 5

Problem Statement : Optimize the water distribution network using the Grey Wolf Optimization (GWO) algorithm to minimize the total system cost, including pipe and pump costs, while satisfying node pressure and flow rate demands.

Algorithm :



→ population-based meta-heuristics algorithm.
→ grey wolves are apex predators, live in groups.
  - α
  - β
  - δ
  - ω

  ↑ Dominance.

→ Algorithm: (for Optimal Power Flow (OPF)) → BMP

1. Initialize parameters:
   num_generators, gen_min, gen_max, demand, num_wolves, max_iter
2. Define the objective function:

```
FUNCTION objective_function(generation):
    cost = 0
    FOR each generator i:
        a_i, b_i, c_i = gen_params[i]
        cost += a_i * (gen[i]^2) + b_i * gen[i] + c_i
    RETURN cost
```

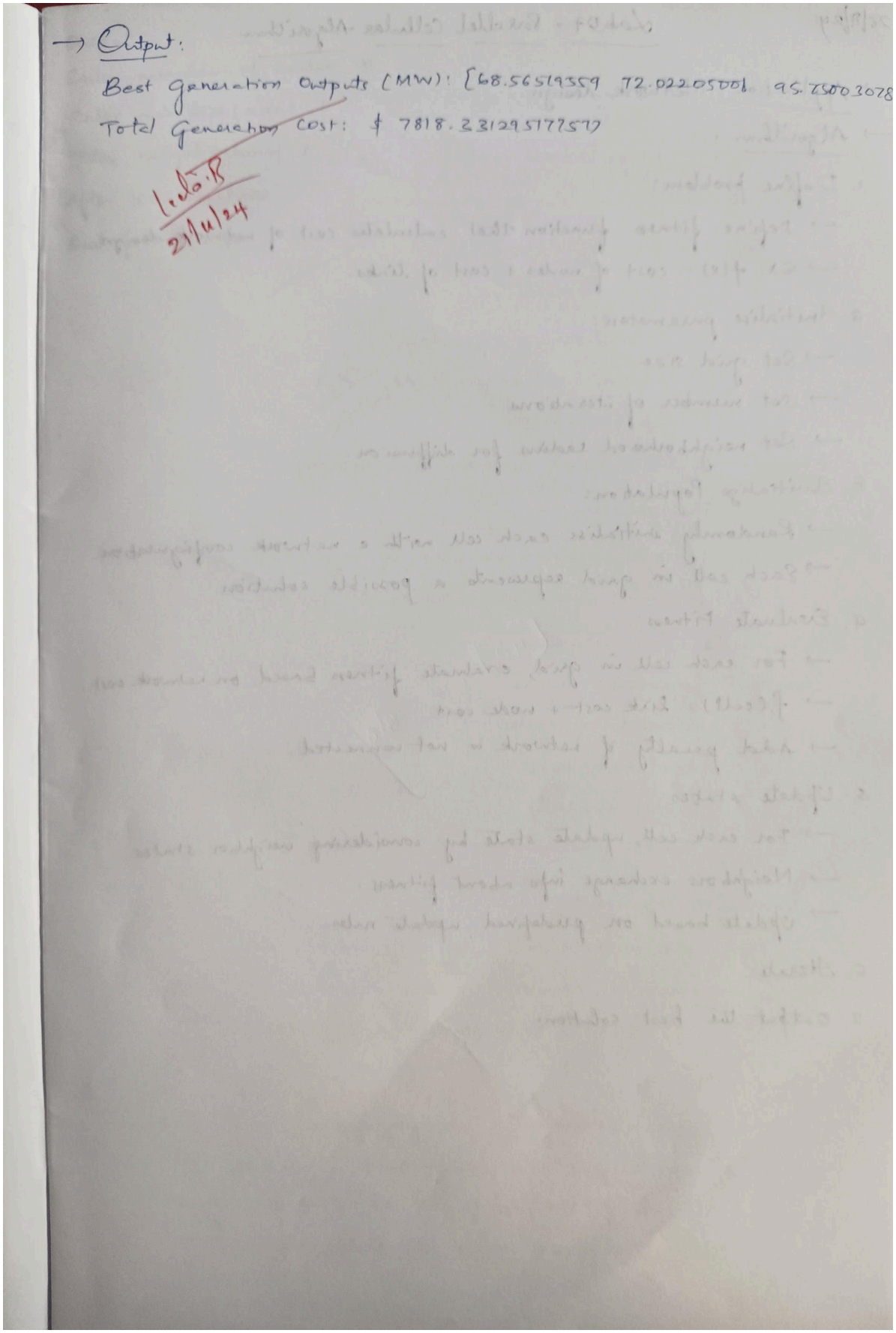
3. Define the constraints check function:

```
FUNCTION check_constraints(gen, demand, tol):
    power_balance = SUM(gen) - demand
    check generator limits.
    Check power_balance constraint within tolerance
```

4. Define the Grey Wolf Optimisation algorithm:

```
CLASS greyWolfOptimizer:
    INITIALIZE all parameter.
    FUNCTION optimize:
        check constraints
        add penalty
        update wolf positions based on best solution
```

5. Run the grey wolf optimiser.



Code :

```
#Waterflow optimization
import numpy as np
```

```
def objective_function(pipe_sizes, flow_rates, demand, node_pressure, pipe_costs,
pump_costs):
    pipe_cost = np.sum(pipe_sizes ** 2 * pipe_costs)
    pump_cost = np.sum(flow_rates * pump_costs)
    pressure_penalty = 0
    for i in range(len(node_pressure)):
        if node_pressure[i] < 20:
            pressure_penalty += (20 - node_pressure[i]) ** 2
    demand_penalty = 0
    for i in range(len(demand)):
        if flow_rates[i] < demand[i]:
            demand_penalty += (demand[i] - flow_rates[i]) ** 2

    total_cost = pipe_cost + pump_cost + pressure_penalty + demand_penalty
    return total_cost


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

        self.num_wolves = num_wolves
        self.max_iter = max_iter
        self.demand = demand
        self.pipe_costs = pipe_costs
        self.pump_costs = pump_costs
        self.num_nodes = num_nodes

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

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

    def fitness(self, wolf):
        pipe_sizes = wolf[:self.num_nodes]
        flow_rates = wolf[self.num_nodes:]
        node_pressure = np.random.rand(self.num_nodes) * 50
        return objective_function(pipe_sizes, flow_rates, self.demand,
node_pressure, self.pipe_costs, self.pump_costs)

    def update_positions(self):
```

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

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

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

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

    def optimize(self):
        for _ in range(self.max_iter):
            for i in range(self.num_wolves):
                fitness_value = self.fitness(self.wolves[i])

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

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

                elif fitness_value < self.delta_score:
                    self.delta_score = fitness_value
                    self.delta = self.wolves[i]


            self.update_positions()

        return self.alpha


num_wolves = 30
max_iter = 100
num_nodes = 5
demand = np.array([50, 40, 30, 60, 80])  #
pipe_costs = np.array([1, 2, 1.5, 3, 2])
pump_costs = np.array([0.1, 0.1, 0.1, 0.1, 0.1])
```

```
gwo = GreyWolfOptimization(num_wolves, max_iter, demand, pipe_costs, pump_costs,
num_nodes)
best_solution = gwo.optimize()


best_pipe_sizes = best_solution[:num_nodes]
best_flow_rates = best_solution[num_nodes:]

print("Best Pipe Sizes:", best_pipe_sizes)
print("Best Flow Rates:", best_flow_rates)
```

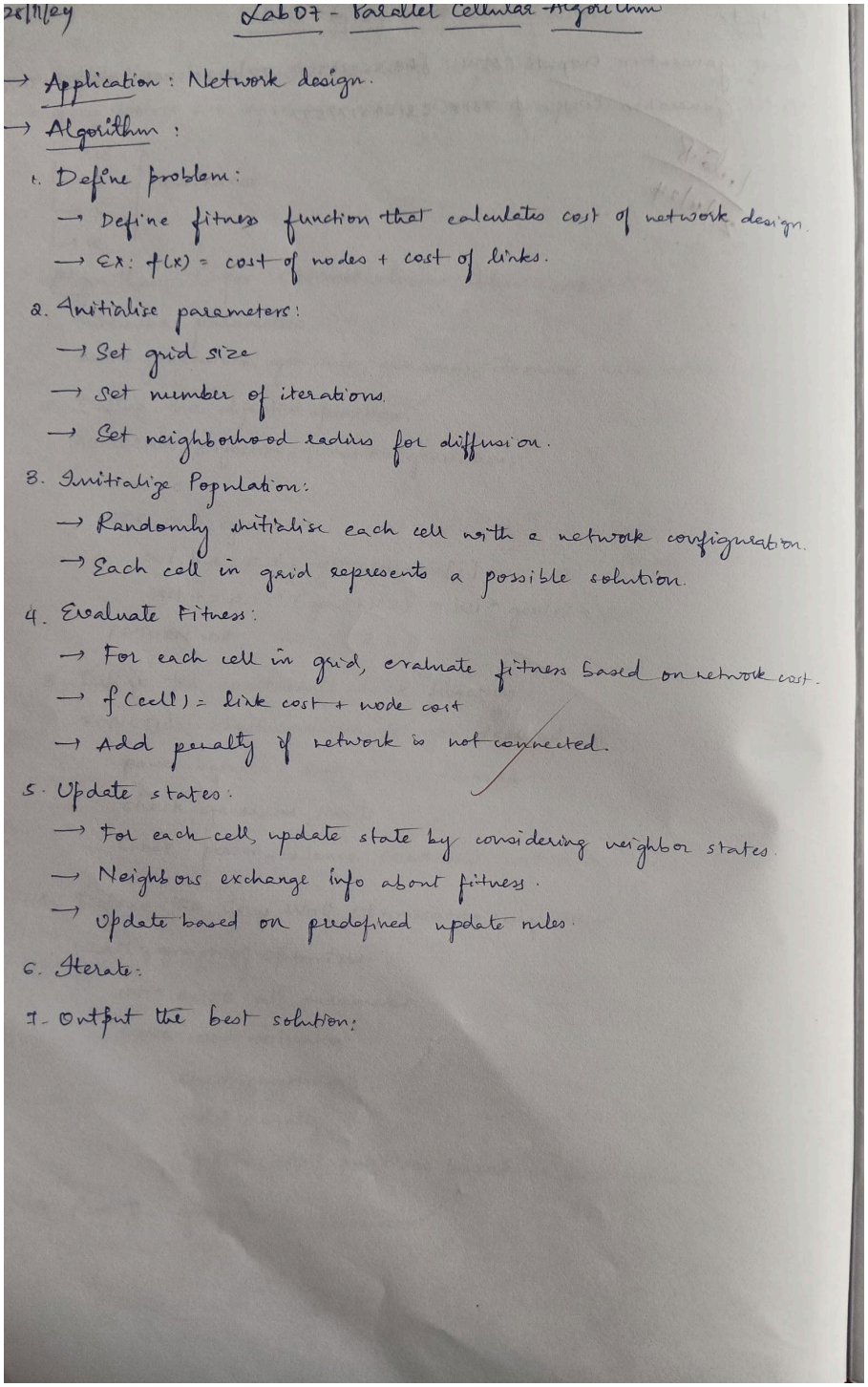
Output :

```
Best Pipe Sizes: [ 4.51303825  9.51306075  7.576117    8.00677675 10.0568968 ]
Best Flow Rates: [6.00439163 9.72371995 6.72743017 8.03606297 7.57520181]
```

Program 6

Problem Statement : Implement Conway's Game of Life using a parallel cellular algorithm on a 2D grid, where each cell updates simultaneously based on its neighbors' states, and the grid evolves for a specified number of generations, counting and displaying the live cells at each step.

Algorithm :

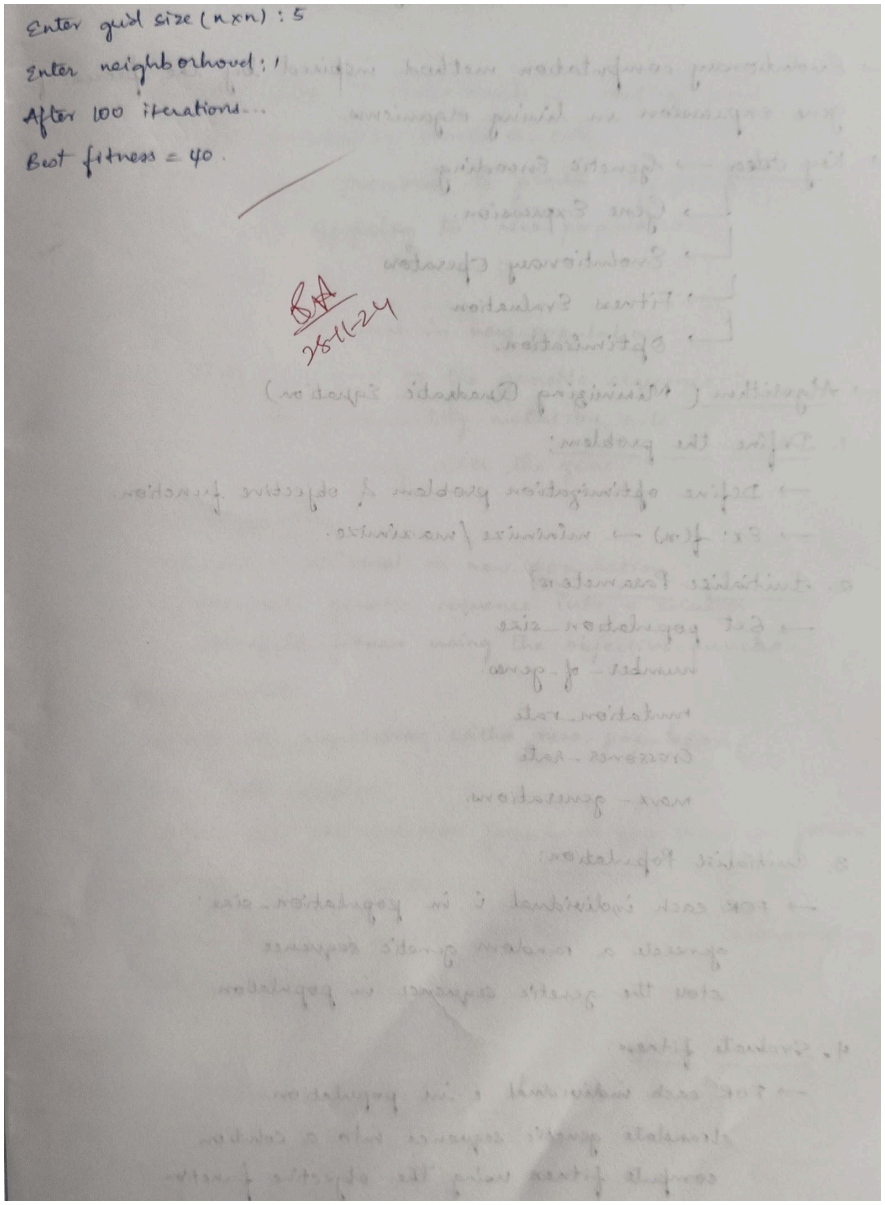


25/11/24 Lab 07 - Parallel Cellular Algorithm

→ Application : Network design.

→ Algorithm :

1. Define problem:
   → Define fitness function that calculates cost of network design.
   → Ex: f(x) = cost of nodes + cost of links.
2. Initialise parameters:
   → Set grid size
   → Set number of iterations.
   → Set neighborhood radius for diffusion.
3. Initialize Population:
   → Randomly initialise each cell with a network configuration.
   → Each cell in grid represents a possible solution.
4. Evaluate Fitness:
   → For each cell in grid, evaluate fitness based on network cost.
   → f(cell) = link cost + node cost
   → Add penalty if network is not connected.
5. Update states:
   → For each cell, update state by considering neighbor states.
   → Neighbors exchange info about fitness.
   → Update based on predefined update rules.
6. Iterate:
7. Output the best solution:



Code :

```
#Game of life
import random
import copy

GRID_WIDTH = 10
GRID_HEIGHT = 10
MAX_GENERATIONS = 20

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

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

```
                  ( 1, -1), ( 1, 0), ( 1, 1)]

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

    return live_neighbors

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

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

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

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

def game_of_life(grid_width, grid_height, max_generations):
    grid = initialize_grid(grid_width, grid_height)
    print("Initial Grid:")
    display_grid(grid)
    for generation in range(max_generations):
        print(f"Generation {generation + 1}:")
        grid = update_grid(grid)
        # display_grid(grid)
        alive_cells = count_alive_cells(grid)
        print(f"Number of alive cells: {alive_cells}")

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

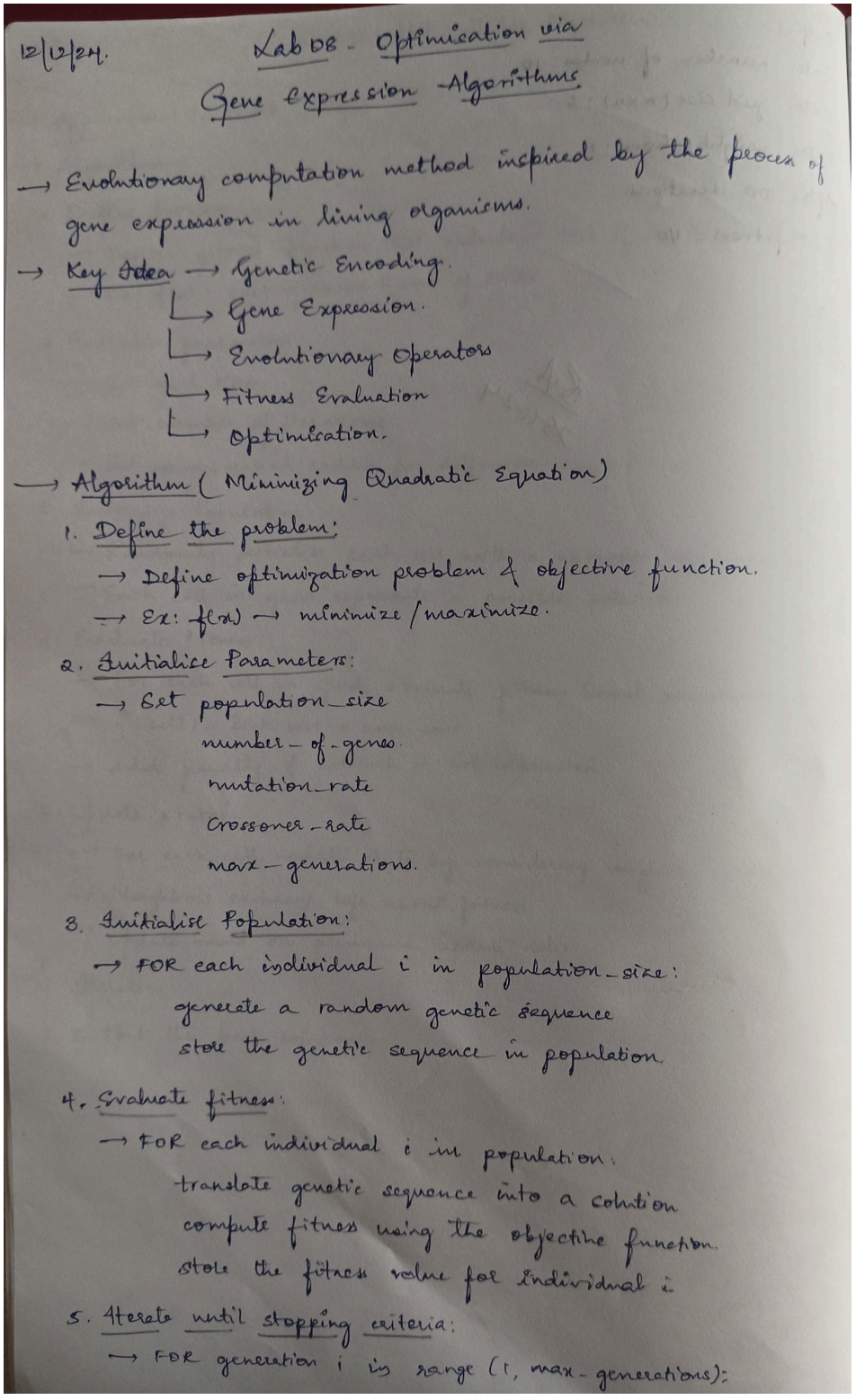
Output:

```
Initial Grid:
0 0 1 0 1 1 1 1 0 1
0 0 0 0 0 0 1 0 0 1
0 0 0 0 0 1 1 0 0 1
1 0 1 0 1 0 0 1 1 1
0 0 1 0 1 1 1 0 1 1
1 1 1 0 1 1 1 0 1 0
1 1 1 0 0 0 1 0 1 0
1 0 1 1 1 1 1 0 1 1
0 1 1 1 1 1 0 1 0 0
1 1 1 1 1 0 0 1 0 0

====================

Generation 1:
Number of alive cells: 20
Generation 2:
Number of alive cells: 26
Generation 3:
Number of alive cells: 24
Generation 4:
Number of alive cells: 27
Generation 5:
Number of alive cells: 28
Generation 6:
...
Generation 19:
Number of alive cells: 26
Generation 20:
Number of alive cells: 26
```

Program 7

Problem Statement : Implement Conway's Game of Life using a parallel cellular algorithm on a 2D grid, where each cell updates simultaneously based on its neighbors' states, and the grid evolves for a specified number of generations, counting and displaying the live cells at each step.

Algorithm :



12/12/24. Lab 08 - Optimization via Gene Expression Algorithms

→ Evolutionary computation method inspired by the process of gene expression in living organisms.

→ Key Idea → Genetic Encoding.
- Gene Expression.
- Evolutionary Operators
- Fitness Evaluation
- Optimization.

→ Algorithm (Minimizing Quadratic Equation)

1. Define the problem:
   → Define optimization problem & objective function.
   → Ex: f(x) → minimize / maximize.
2. Initialize Parameters:
   → Set population_size
   number_of_genes
   mutation_rate
   Crossover_rate
   max_generations.
3. Initialise Population:
   → FOR each individual i in population_size:
   generate a random genetic sequence
   store the genetic sequence in population
4. Evaluate fitness:
   → FOR each individual i in population:
   translate genetic sequence into a solution
   compute fitness using the objective function.
   store the fitness value for individual i
5. Iterate until stopping criteria:
   → FOR generation i in range (1, max_generations):

a) Selection:
Select individuals based on fitness.
Store selected individuals as mating-pool

b) Crossover:
FOR each pair of individuals in mating-pool:
with probability crossover_rate:
perform crossover to produce offspring
add offspring to new-population.

c) Mutation:
FOR each individual in new-population:
FOR each gene in the genetic sequence:
with probability mutation_rate:
randomly alter the gene.

d) Gene expression:
FOR each individual in new-population:
translate genetic sequence into a solution
compute fitness using the objective function.

e) Replacement:
Replace old population with new-population.

f) Track best solution:
Update the best solution found so far based on fitness

6. Output the best solution:
→ Return the best genetic solution and its corresponding fitness value

→ Output:
Input function: $x^2 - 2x + 10$
pop-size = 20
generations = 100
x_min, x_max = -10, 10
crossover_rate = 0.8
Best solution: x = 1.00000000, Minimum solution: f(x): 9

Code :

```
import random
```

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

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

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

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

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

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

# Gene Expression Algorithm
def gene_expression_algorithm(pop_size, generations, x_min, x_max):
    population = initialize_population(pop_size, x_min, x_max)
    best_solution = None
```

```
    best_fitness = float("inf")

    for gen in range(generations):
        fitness = evaluate_fitness(population)
        if min(fitness) < best_fitness:
            best_fitness = min(fitness)
            best_solution = population[fitness.index(best_fitness)]

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

    return best_solution, best_fitness

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

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

Output: