

VISVESVARAYA TECHNOLOGICAL UNIVERSITY

“JnanaSangama”, Belgaum -590014, Karnataka.



LAB RECORD

Bio Inspired Systems (23CS5BSBIS)

Submitted by

Abdul Ahad(1BM23CS353)

in partial fulfillment for the award of the degree of

BACHELOR OF ENGINEERING
in
COMPUTER SCIENCE AND ENGINEERING



B.M.S. COLLEGE OF ENGINEERING

(Autonomous Institution under VTU)

BENGALURU-560019

Aug-2025 to Dec-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 **ABDUL AHAD (1BM23CS353)**, 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.

| | |
|---|--|
| Sowmya T Assistant Professor Department of CSE, BMSCE | Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE |
|---|--|

Index

| Sl. No. | Date | Experiment Title | Page No. |
|---------|------------|---|----------|
| 1 | 18/08/2025 | Genetic Algorithm for Optimization Problems | 1-7 |
| 2 | 25/08/2025 | Optimization via Gene Expression Algorithms | 8-13 |
| 3 | 01/09/2025 | Particle Swarm Optimization for Function Optimization | 14-18 |
| 4 | 08/09/2025 | Ant Colony Optimization for the TSP | 19-25 |
| 5 | 15/09/2-05 | Cuckoo Search (CS): | 26-30 |
| 6 | 29/09/2025 | Grey Wolf Optimizer (GWO): | 31-35 |
| 7 | 13/10/2025 | Parallel Cellular Algorithms and Programs: | 36-40 |

GitHub Link:

https://github.com/Abdul31347/BIS_LAB

Program 1

Genetic Algorithm for Optimization Problems:

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

Implementation Steps:

1. Define the Problem: Create a mathematical function to optimize.
2. Initialize Parameters: Set the population size, mutation rate, crossover rate, and number of generations.
3. Create Initial Population: Generate an initial population of potential solutions.
4. Evaluate Fitness: Evaluate the fitness of each individual in the population.
5. Selection: Select individuals based on their fitness to reproduce.
6. Crossover: Perform crossover between selected individuals to produce offspring.
7. Mutation: Apply mutation to the offspring to maintain genetic diversity.
8. Iteration: Repeat the evaluation, selection, crossover, and mutation processes for a fixed number of generations or until convergence criteria are met.
9. Output the Best Solution: Track and output the best solution found during the generations.

Algorithm:

Implementation Genetic Algorithm

1. Select initial population
 2. Calculate the fitness
 3. Selecting the mating pool
 4. Crossover
 5. Mutation

Prob = $f(x)$
 $E f(x)$
 $= \frac{144}{1155}$
 $= 0.1247$

Expected output = 144
 Avg $E f(x_i)$ = 288.75

1. $x \rightarrow 0-31$ = 0.49

2. Fitness

| String No | Initial Population | X value | $f(x) = x^2$ | Prob | % Prob | Expected output | Actual count |
|-----------|--------------------|---------|--------------|--------|--------|-----------------|--------------|
| 1 | 01100 | 12 | 144 | 0.1247 | 12.47 | 0.49 | 1 |
| 2 | 11001 | 25 | 625 | 0.5411 | 54.11 | 2.16 | 2 |
| 3 | 00101 | 5 | 25 | 0.0216 | 2.16 | 0.08 | 0 |
| 4 | 10011 | 19 | 361 | 0.3126 | 31.26 | 1.25 | 1 |
| Sum | | | 1155 | 1.0 | 100 | 4 | |
| Average | | | 288.75 | 0.25 | 25 | 1 | |
| Maximum | | | 625 | 0.5411 | 54.11 | 2.16 | |

3. Selecting Mating Pool

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

4. Crossover: Crossover point is chosen randomly

| String no | Mutation offspring after crossover | Mutation Chromosome | Offspring after mutation | X-value | Fitness |
|-----------|--|------------------------|-----------------------------|---------|---------|
| 1 | 0 1 1 0 1 | 1 0 0 0 0 | 1 1 1 0 1 | 29 | 841 |
| 2 | 1 1 0 0 0 | 0 0 0 0 0 | 1 1 0 0 0 | 24 | 576 |
| 3 | 1 1 0 1 1 | 0 0 0 0 0 | 1 1 0 1 1 | 27 | 729 |
| 4 | 1 0 0 0 1 | 0 0 1 0 1 | 1 0 1 0 0 | 20 | 400 |
| Sum | | | | | 2546 |
| Average | | | | | 636.5 |
| Max | | | | | 841 |

Genetic Algorithm for scheduling

Goal: Assign tasks to resources to minimize time/cost.

Why GA: Handles NP-hard scheduling efficiently with near-optimal solutions

Key steps

- (1) Encode schedule as chromosome
- (2) Generate random initial population
- (3) Fitness = $1/\text{makespan}$
- (4) Apply selection \rightarrow crossover \rightarrow mutation
- (5) Repeat for generations.

Used in: Airline crew scheduling

Best Job order: [2, 1, 4, 5, 3, 0]

Job times: [7, 2, 9, 4, 5, 3]

total completion time (Makespan): 30

```

Code:
import random
import math

# -----
# CONFIG
NUM_CITIES = 10
POP_SIZE = 100
GENERATIONS = 20
MUTATION_RATE = 0.02

# -----
# Generate random cities
cities = [(random.uniform(0, 100), random.uniform(0, 100)) for _ in range(NUM_CITIES)]

# -----
# Distance between two cities
def distance(city1, city2):
    return math.hypot(city1[0] - city2[0], city1[1] - city2[1])

# Total path length (fitness is inverse)
def total_distance(tour):
    return sum(distance(cities[tour[i]], cities[tour[(i+1) % NUM_CITIES]]) for i in
range(NUM_CITIES))

# -----
# Create random individual (a tour)
def create_individual():
    tour = list(range(NUM_CITIES))
    random.shuffle(tour)
    return tour

# Crossover: Order Crossover (OX)
def crossover(parent1, parent2):
    start, end = sorted(random.sample(range(NUM_CITIES), 2))
    child = [None] * NUM_CITIES

    # Copy slice from first parent
    child[start:end+1] = parent1[start:end+1]

    # Fill in the rest from second parent
    ptr = 0
    for city in parent2:
        if city not in child:

```

```

        while child[ptr] is not None:
            ptr += 1
            child[ptr] = city

    return child

# Mutation: Swap two cities
def mutate(tour):
    if random.random() < MUTATION_RATE:
        i, j = random.sample(range(NUM_CITIES), 2)
        tour[i], tour[j] = tour[j], tour[i]
    return tour

# -----
# Initial population
population = [create_individual() for _ in range(POP_SIZE)]

# -----
# Main GA loop
for gen in range(GENERATIONS):
    # Evaluate fitness
    scored = [(ind, total_distance(ind)) for ind in population]
    scored.sort(key=lambda x: x[1]) # lower distance is better
    best = scored[0]

    print(f'Generation {gen+1}: Best distance = {best[1]:.2f}')

    # Selection: keep top 50%
    selected = [ind for ind, _ in scored[:POP_SIZE // 2]]

    # Reproduce
    children = []
    while len(children) < POP_SIZE:
        p1, p2 = random.sample(selected, 2)
        child = crossover(p1, p2)
        child = mutate(child)
        children.append(child)

    population = children

# Final best route
best_tour = scored[0][0]
print("\nBest tour found:", best_tour)

```

Program 2

Optimization via Gene Expression Algorithms:

Gene Expression Algorithms (GEA) are inspired by the biological process of gene expression in living organisms. This process involves the translation of genetic information encoded in DNA into functional proteins. In GEA, solutions to optimization problems are encoded in a manner similar to genetic sequences. The algorithm evolves these solutions through selection, crossover, mutation, and gene expression to find optimal or near-optimal solutions. GEA is effective for solving complex optimization problems in various domains, including engineering, data analysis, and machine learning.

Implementation Steps:

1. Define the Problem: Create a mathematical function to optimize.
2. Initialize Parameters: Set the population size, number of genes, mutation rate, crossover rate, and number of generations.
3. Initialize Population: Generate an initial population of random genetic sequences.
4. Evaluate Fitness: Evaluate the fitness of each genetic sequence based on the optimization function.
5. Selection: Select genetic sequences based on their fitness for reproduction.
6. Crossover: Perform crossover between selected sequences to produce offspring.
7. Mutation: Apply mutation to the offspring to introduce variability.
8. Gene Expression: Translate genetic sequences into functional solutions.
9. Iterate: Repeat the selection, crossover, mutation, and gene expression processes for a fixed number of generations or until convergence criteria are met.
10. Output the Best Solution: Track and output the best solution found during the iterations.

Algorithm:

Lab-2

Gene Expression Algorithm (GEA) for Travelling Salesman Problem

Algorithm Steps

- (1) Define Problem
 - TSP: Find the shortest route visiting all cities once and returning to start
- (2) Initialize Parameters
 - Population size (POP size)
 - Crossover rate (CR), Mutation rate (MR)
 - Number of generations (MAX_GEN)
- (3) Initialize Population
 - Each chromosome = a permutation of cities (route)
- (4) Evaluate Fitness
 - Complete route distance:
$$\text{Distance} = \sum \text{dist}(\text{city}[i], \text{city}[i+1])$$
 - Fitness = $1 / \text{distance}$
- (5) Selection
 - use tournament or roulette wheel based on fitness
- (6) Crossover (Order based crossover)
 - Select two parents
 - Choose random segment from parent 1 and preserve order from parent 2
- (7) Mutation
 - Swap two cities in the route with a small probability
- (8) Gene Expression
 - The chromosome directly represents a route → compute its distance

9/8/20

9 Iteration

Repeat steps 4-8 for MAX-GEN or until convergence

10 Output Best Solution

Track the route with the minimum distance

Output

Gen 1 : Best Distance = 18.30

Gen 5 : Best Distance = 15.72

Gen 11 : Best Distance = 14.48

Gen 16 : Best Distance = 14.48

Gen 20 : Best Distance = 14.48

Best Route : [0, 1, 2, 3, 4]

Best Distance : 14.48

for 20/20

Code:

```
import numpy as np
import random
import operator

# Generate noisy data from a nonlinear function
def target_function(x):
    return 2 * x**3 - x + 1

def generate_data(n_points=30):
    xs = np.linspace(-2, 2, n_points)
    ys = np.array([target_function(x) for x in xs]) + np.random.normal(0, 1, n_points)
    return xs, ys

X_data, y_data = generate_data()

# Parameters
POP_SIZE = 50
GENE_LENGTH = 20
NUM_GENERATIONS = 10
MUTATION_RATE = 0.1
CROSSOVER_RATE = 0.7

# Function and terminal sets
FUNCTIONS = [('+', operator.add), ('-', operator.sub), ('*', operator.mul)]
TERMINALS = ['x', '1']
func_dict = {f[0]: f[1] for f in FUNCTIONS}

# Convert gene (list of symbols) into executable function
def express_gene(gene):
    stack = []
    for symbol in gene:
        if symbol in func_dict:
            try:
                b = stack.pop()
                a = stack.pop()
                func = func_dict[symbol]
                stack.append(lambda x, a=a, b=b, func=func: func(a(x), b(x)))
            except IndexError:
                stack.append(lambda x: 1)
        elif symbol == 'x':
            stack.append(lambda x: x)
        elif symbol == '1':
            stack.append(lambda x: 1)
        else:
            stack.append(lambda x: 1)
```

```

    return stack[-1] if stack else lambda x: 1

# Fitness: negative MSE
def fitness(gene):
    func = express_gene(gene)
    try:
        ys_pred = np.array([func(x) for x in X_data])
        mse = np.mean((ys_pred - y_data) ** 2)
        if np.isnan(mse) or np.isinf(mse):
            return -float('inf')
        return -mse
    except Exception:
        return -float('inf')

# Tournament selection
def selection(pop, k=3):
    selected = random.sample(pop, k)
    return max(selected, key=fitness)

# Crossover
def crossover(parent1, parent2):
    if len(parent1) != len(parent2):
        return parent1[:]
    point = random.randint(1, len(parent1) - 2)
    return parent1[:point] + parent2[point:]

# Mutation
def mutate(gene, mutation_rate=MUTATION_RATE):
    gene = gene[:]
    symbols = [f[0] for f in FUNCTIONS] + TERMINALS
    for i in range(len(gene)):
        if random.random() < mutation_rate:
            gene[i] = random.choice(symbols)
    return gene

# Initialize population
population = [[random.choice([f[0] for f in FUNCTIONS] + TERMINALS) for _ in
range(GENE_LENGTH)] for _ in range(POP_SIZE)]

best_gene = None
best_fit = -float('inf')

# Main loop
for gen in range(NUM_GENERATIONS):
    new_population = []
    for _ in range(POP_SIZE):
        parent1 = selection(population)

```

```

    parent2 = selection(population)
    child = crossover(parent1, parent2) if random.random() < CROSSOVER_RATE else parent1[:]
    child = mutate(child)
    new_population.append(child)

population = new_population

generation_best = max(population, key=fitness)
generation_best_fit = fitness(generation_best)

if generation_best_fit > best_fit:
    best_fit = generation_best_fit
    best_gene = generation_best

if gen % 10 == 0 or gen == NUM_GENERATIONS - 1:
    print(f'Generation {gen}: Best fitness = {best_fit:.6f}')

# Output best gene and prediction results
print("Best gene (symbolic expression):", best_gene)
best_func = express_gene(best_gene)
print("Sample predictions vs actual values:")
for x, y_true in zip(X_data[:10], y_data[:10]):
    try:
        y_pred = best_func(x)
        print(f'x={x:.3f}, Predicted={y_pred:.3f}, Actual={y_true:.3f}')
    except Exception:
        print(f'x={x:.3f}, Predicted=Error, Actual={y_true:.3f}')

```

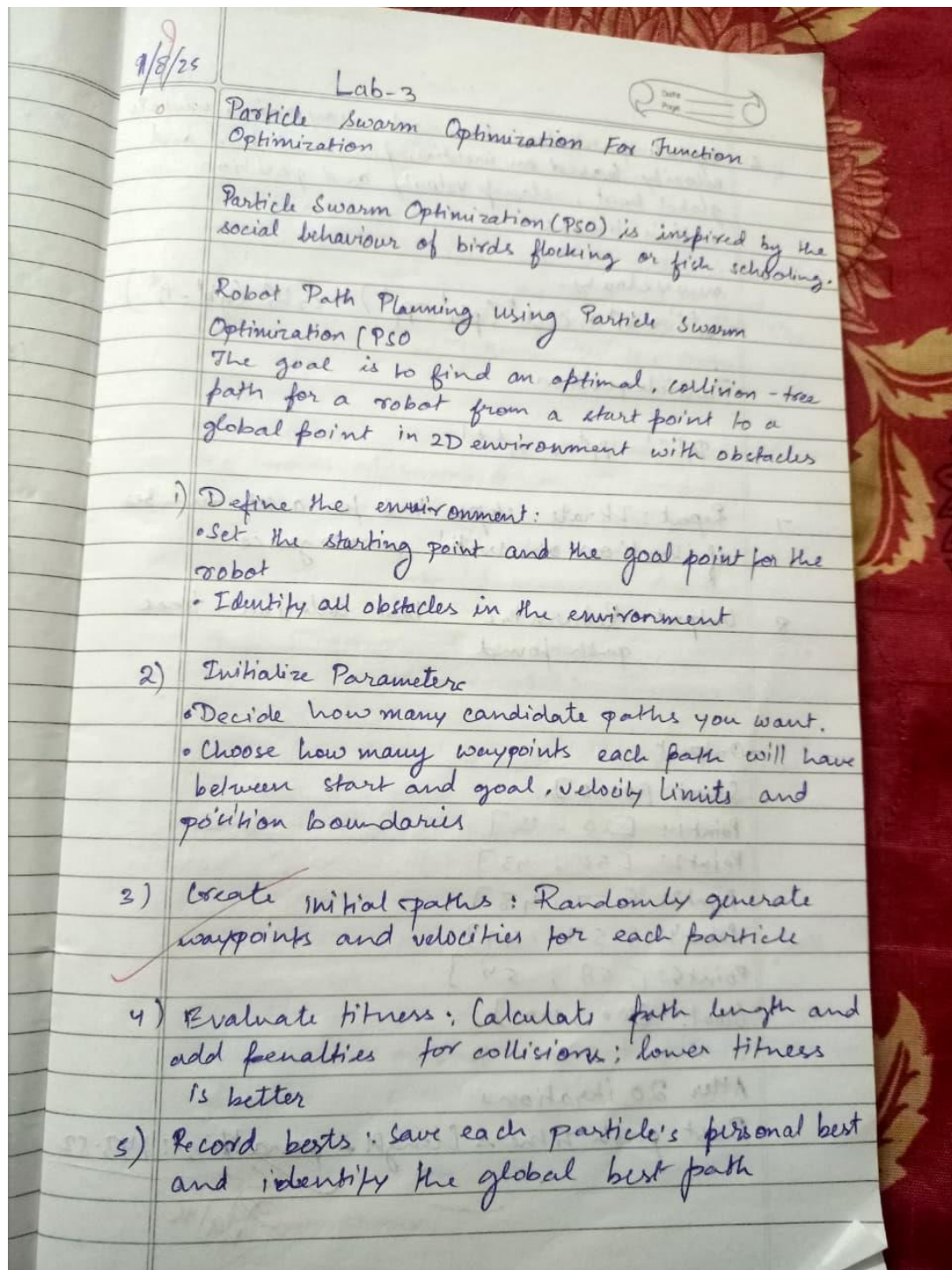
Program 3

Particle Swarm Optimization for Function Optimization:

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

1. Define the Problem: Create a mathematical function to optimize.
2. Initialize Parameters: Set the number of particles, inertia weight, cognitive and social coefficients.
3. Initialize Particles: Generate an initial population of particles with random positions and velocities.
4. Evaluate Fitness: Evaluate the fitness of each particle based on the optimization function.
5. Update Velocities and Positions: Update the velocity and position of each particle based on its own best position and the global best position.
6. Iterate: Repeat the evaluation, updating, and position adjustment for a fixed number of iterations or until convergence criteria are met.
7. Output the Best Solution: Track and output the best solution found during the iterations.

Algorithm:



$$v_i^{t+1} = v_i^t + C_1 \cdot U_1^t (p_{bi}^t - p_i^t) + C_2 \cdot U_2^t (g_{bt}^t - p_i^t)$$

- 6 Update particles: For each waypoint, update velocity based on inertia, personal best and global best, clamp velocity and position within limits

new velocity

$$v_i^{t+1} = v_i^t + C_1 U_1^t (p_{bi}^t - p_i^t) + C_2 U_2^t (g_{bt}^t - p_i^t)$$

Particle position

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

- 7 Repeat: Iterate steps 4-6 for a set number of iteration or until convergence

- 8 Output: Return the best collision-free path found

Output

Start: [0, 0]

Point 1: [20, 15]

Point 2: [50, 43]

Point 3: [57, 48]

Point 4: [57, 49]

Point 5: [58, 54]

Goal: [100, 100]

After 20 iterations

Best path fitness (length + penalties): 143.52

8/9/25

Code:

```
import numpy as np

def de_jong(position):
    x, y = position
    return x**2 + y**2

num_particles = 30
dimensions = 2
iterations = 10
w = 0.5
c1 = 1.5
c2 = 1.5
bounds = (-5.12, 5.12)

positions = np.random.uniform(bounds[0], bounds[1], (num_particles, dimensions))
velocities = np.random.uniform(-1, 1, (num_particles, dimensions))
pbest_positions = np.copy(positions)
pbest_scores = np.array([de_jong(p) for p in positions])

gbest_index = np.argmin(pbest_scores)
gbest_position = pbest_positions[gbest_index]
gbest_score = pbest_scores[gbest_index]

for t in range(iterations):
    for i in range(num_particles):
        r1 = np.random.rand(dimensions)
        r2 = np.random.rand(dimensions)

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

        positions[i] += velocities[i]
        positions[i] = np.clip(positions[i], bounds[0], bounds[1])

        fitness = de_jong(positions[i])

        if fitness < pbest_scores[i]:
            pbest_positions[i] = positions[i]
            pbest_scores[i] = fitness

        if fitness < gbest_score:
            gbest_position = positions[i]
            gbest_score = fitness

    print(f'Iteration {t+1:3d} | Best Score: {gbest_score:.6f}')
```

```
print("\nBest Solution Found:")  
print(f'Position: x = {gbest_position[0]:.6f}, y = {gbest_position[1]:.6f}')  
print(f'Score: {gbest_score:.10f}')
```

Program 4

Ant Colony Optimization for the Traveling Salesman Problem:

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

1. Define the Problem: Create a set of cities with their coordinates.
2. Initialize Parameters: Set the number of ants, the importance of pheromone (α), the importance of heuristic information (β), the evaporation rate (ρ), and the initial pheromone value.
3. Construct Solutions: Each ant constructs a solution by probabilistically choosing the next city based on pheromone trails and heuristic information.
4. Update Pheromones: After all ants have constructed their solutions, update the pheromone trails based on the quality of the solutions found.
5. Iterate: Repeat the construction and updating process for a fixed number of iterations or until convergence criteria are met.
6. Output the Best Solution: Keep track of and output the best solution found during the iterations.

Algorithm:

8/9/25 Lab-4

Ant Colony Optimization for the Travelling Salesman Problem

Problem

Given a list of cities and the distances between each pair of cities, the task is to find shortest possible route that visits each city exactly once and returns to the origin city

Goal

To minimize the total travel distance representing most efficient path

Ants in nature find the shortest path to food by leaving pheromone trails

At first, ants walk randomly. Some find food and return, leaving pheromones on their path. Other ants follow stronger pheromone paths reinforcing them. Over time the shortest path has strongest pheromone, so most ants follow it

1) Initialization

- Set a small amount of pheromone on every path between cities
- Choose the number of ants, pheromone evaporation rate and how strongly ants follow pheromone vs distance
- Place ants randomly on cities

Mutation → Bit Flip

File Summation

$V_i + C_i$
Ants

2) Construct solutions

- Each ant starts at a city
- One by one, ants pick the next city to visit
 - They prefer shorter paths
 - They also prefer paths with stronger pheromone
- Each ant keeps going until it has visited all cities and returns to the starting city (completes a tour)

3) Evaluate tours

- Measure the total distance of each ant's tour
- Keep track of shortest tour found so far

4) Update Pheromones

- Reduce pheromone on all paths a little (evaporation)
- Add new pheromone to the paths ant travelled giving more pheromone to shorter tours

5) Repeat

- Keep repeating the process of ants exploring, evaluating and updating pheromones
- Over time stronger pheromone gathers on best path

6) Output best tour

- When process stops, output the shortest tour found and its distance

Example

- At start
- Suppose
- distance
- Another
- More
- Last
- Next
- 18-
- After
- The

Best

to

Due

E

E

E

E

E

Example with 4 cities (A, B, C, D)

- At start: ants randomly travel
- Suppose one ant finds route $A \rightarrow B \rightarrow C \rightarrow D \rightarrow A$ with distance 22
- Another find $A \rightarrow C \rightarrow D \rightarrow B \rightarrow A$ with distance 18
- More pheromone is added to the shorter path (distance 18).
- Next round more ants are likely to follow 18-distance path
- After several rounds, almost all ants follow the best (shortest) path

Best route found: $A \rightarrow C \rightarrow D \rightarrow B \rightarrow A$

total distance : 18

Output

Enter the number of cities: 4

Enter coordinates for city 1 (x, y): 1 2

Enter coordinates for city 2 (x, y): 3 5

Enter coordinates for city 3 (x, y): 4 3

Enter coordinates for city 4 (x, y): 7 5

Best tour [2, 3, 1, 4]

Best distance : 5.40

Sgt
9/15

Code:

```
import numpy as np
import random
```

```
class ACO_TSP:
```

```
    def __init__(self, distances, n_ants=10, n_iterations=100, alpha=1, beta=5, rho=0.5, Q=100):
        self.distances = distances
        self.n_cities = distances.shape[0]
        self.n_ants = n_ants
        self.n_iterations = n_iterations
        self.alpha = alpha
        self.beta = beta
        self.rho = rho
        self.Q = Q
        self.pheromone = np.ones((self.n_cities, self.n_cities))
```

```
    def run(self):
```

```
        best_length = float("inf")
        best_path = None
```

```
        for iteration in range(self.n_iterations):
```

```
            paths = []
            lengths = []
```

```
            for ant in range(self.n_ants):
```

```
                path = self.construct_solution()
                length = self.calculate_length(path)
                paths.append(path)
                lengths.append(length)
```

```
                if length < best_length:
                    best_length = length
                    best_path = path
```

```
            self.update_pheromones(paths, lengths)
            print(f'Iteration {iteration+1}: Best Length = {best_length}')
```

```
        return best_path, best_length
```

```
    def construct_solution(self):
```

```
        start = random.randint(0, self.n_cities - 1)
        path = [start]
        visited = {start}
```

```
        for _ in range(self.n_cities - 1):
```

```
            current = path[-1]
            next_city = self.choose_next_city(current, visited)
```

```

        path.append(next_city)
        visited.add(next_city)

    return path

def choose_next_city(self, current, visited):
    probabilities = []
    for city in range(self.n_cities):
        if city not in visited:
            tau = self.pheromone[current][city] ** self.alpha
            eta = (1 / self.distances[current][city]) ** self.beta
            probabilities.append((city, tau * eta))
        else:
            probabilities.append((city, 0))

    total = sum(prob for _, prob in probabilities)
    r = random.random() * total
    cumulative = 0
    for city, prob in probabilities:
        cumulative += prob
        if cumulative >= r:
            return city

def calculate_length(self, path):
    length = 0
    for i in range(len(path) - 1):
        length += self.distances[path[i]][path[i+1]]
    length += self.distances[path[-1]][path[0]]
    return length

def update_pheromones(self, paths, lengths):
    self.pheromone *= (1 - self.rho) # evaporation
    for path, length in zip(paths, lengths):
        deposit = self.Q / length
        for i in range(len(path) - 1):
            self.pheromone[path[i]][path[i+1]] += deposit
            self.pheromone[path[i+1]][path[i]] += deposit

if __name__ == "__main__":
    distances = np.array([
        [0, 2, 9, 10, 1],
        [2, 0, 6, 4, 3],
        [9, 6, 0, 8, 5],
        [10, 4, 8, 0, 7],
        [1, 3, 5, 7, 0]
    ])

```

```
aco = ACO_TSP(distances, n_ants=10, n_iterations=10)
best_path, best_length = aco.run()
print("\nBest Path:", best_path)
print("Best Length:", best_length)
```

Program 5

Cuckoo Search (CS):

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

1. Define the Problem: Create a mathematical function to optimize.
2. Initialize Parameters: Set the number of nests, the probability of discovery, and the number of iterations.
3. Initialize Population: Generate an initial population of nests with random positions.
4. Evaluate Fitness: Evaluate the fitness of each nest based on the optimization function.
5. Generate New Solutions: Create new solutions via Lévy flights.
6. Abandon Worst Nests: Abandon a fraction of the worst nests and replace them with new random positions.
7. Iterate: Repeat the evaluation, updating, and replacement process for a fixed number of iterations or until convergence criteria are met.
8. Output the Best Solution: Track and output the best solution found during the iterations.

Algorithm:

Reproduction → crossover → two parent → uniform
Mutation → Bit Flip
Swap mutation

Particle swarm

15/9/25

Cuckoo Search Algorithm

It is an optimization method inspired by breeding behaviour of cuckoo birds, which lay their eggs in the nest of other birds.

Key principles

- Cuckoo lays one egg in randomly selected nest
- The best nests, with high quality eggs are the one that goes in next generation
- There is a probability P_a that host bird will discover cuckoo eggs. If then host bird will leave and make new nest leaving cuckoo egg's behind

Cuckoo Search for Travelling Salesman Algorithm

- 1) Initialize Parameters:
First decide how many solution (nests), set the probability for discovering cuckoo eggs (P_a) and decide how many iterations the algorithm will run
- 2) Generate Initial population
Create random routes by shuffling the list of cities. These routes are your starting guesses for the best path
- 3) Evaluate fitness of each nest
Calculate the total total distance for each route. Shorter distances means better routes.

4) Create new nests
Make swap possible

5) Evaluate
If the replacement

6) If a new nest is better than the old one

7) Try to keep each

8) Terminate

- Date _____
 Page _____
- 4) Create new solutions (wicker eggs)
 Make small changes to each route like swapping or reversing parts to explore new possible routes
 - 5) Evaluate new solutions (eggs)
 If the new route is shorter than the old one, replace the old route with new one
 - 6) Abandon worst solution
 Remove a fraction of worst routes and replace them with new random routes to keep diversity
 - 7) Track the best solution
 Keep track of shortest route found so far after each iteration
 - 8) Termination
 Stop after the maximum iterations and return the best route and its distance as the final solution

Output

Initial Best Route: [1, 3, 0, 4, 2] | Distance: 271.82

Iteration 2: New Best Route | Distance: 263.55

⋮

Iteration 5: New Best Route | Distance: 259.07

Final Best Route: [3, 1, 0, 4, 2]

Final Distance: 259.07

Sept 11/15

Code:

```
import numpy as np
import random

values = [60, 100, 120]
weights = [10, 20, 30]
capacity = 50
n_items = len(values)
n_nests = 15
Pa = 0.25
max_iter = 100

def fitness(x):
    total_weight = np.dot(x, weights)
    if total_weight > capacity:
        return 0
    return np.dot(x, values)

def repair(x):
    while np.dot(x, weights) > capacity:
        idx = random.choice([i for i in range(n_items) if x[i]])
        x[idx] = 0
    return x

def levy_flight(Lambda=1.5):
    return np.random.normal(0, Lambda, n_items)

nests = [np.random.randint(0, 2, n_items) for _ in range(n_nests)]
nest_scores = [fitness(repair(x.copy())) for x in nests]

for generation in range(max_iter):
    for i in range(n_nests):
        new_nest = nests[i].copy()
        step = levy_flight()
        new_nest = np.abs(new_nest + step) > 0.5
        new_nest = new_nest.astype(int)
        new_nest = repair(new_nest)
        f_new = fitness(new_nest)
        if f_new > nest_scores[i]:
            nests[i] = new_nest
            nest_scores[i] = f_new

indices = np.argsort(nest_scores)[:int(Pa * n_nests)]
for idx in indices:
    nests[idx] = np.random.randint(0, 2, n_items)
    nests[idx] = repair(nests[idx])
```

```
    nest_scores[idx] = fitness(nests[idx])
best_idx = np.argmax(nest_scores)
print("Best solution:", nests[best_idx], "Value:", nest_scores[best_idx])
```

Program 6

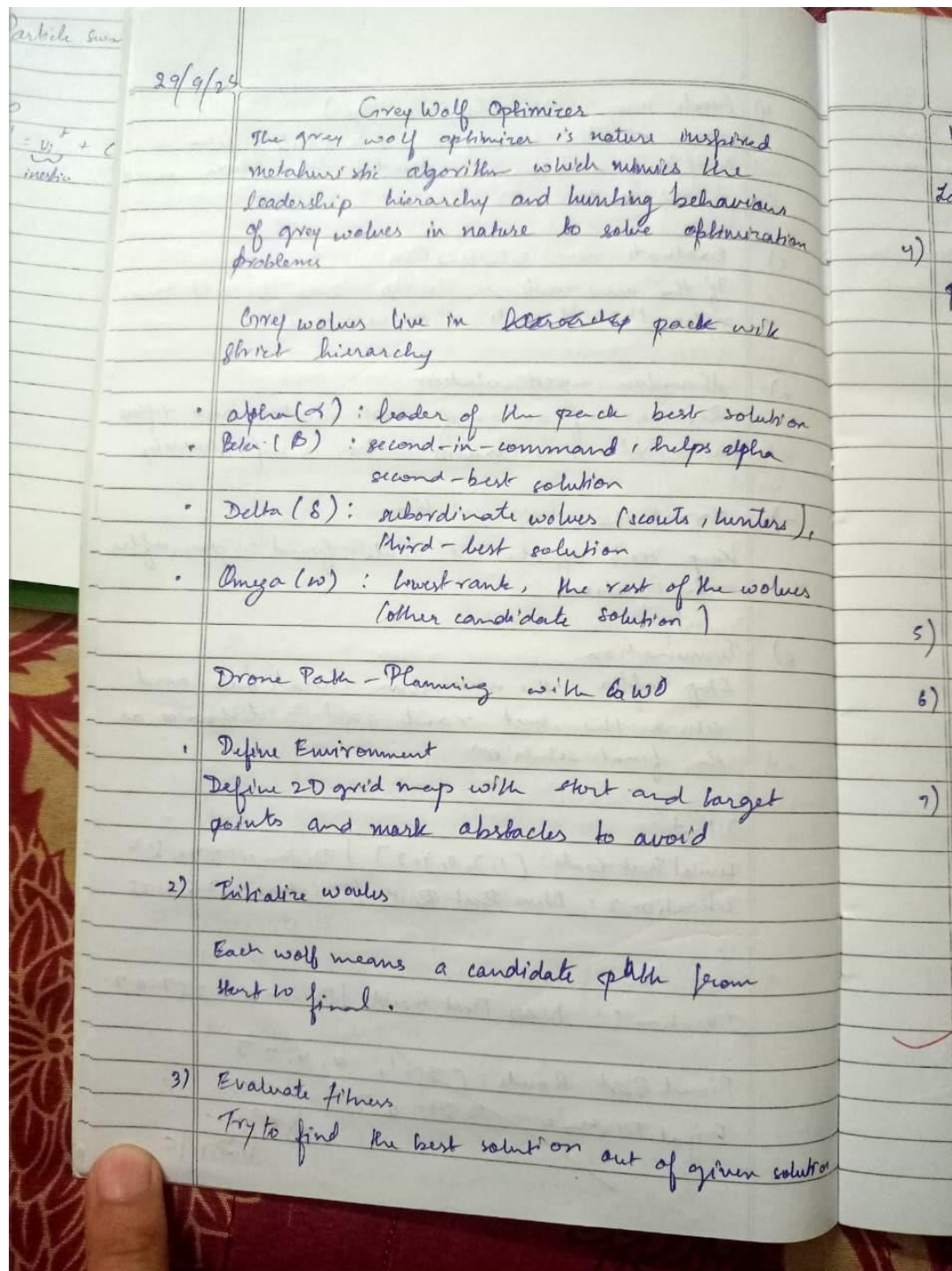
Grey Wolf Optimizer (GWO):

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

Implementation Steps:

1. Define the Problem: Create a mathematical function to optimize.
2. Initialize Parameters: Set the number of wolves and the number of iterations.
3. Initialize Population: Generate an initial population of wolves with random positions.
4. Evaluate Fitness: Evaluate the fitness of each wolf based on the optimization function.
5. Update Positions: Update the positions of the wolves based on the positions of alpha, beta, and delta wolves.
6. Iterate: Repeat the evaluation and position updating process for a fixed number of iterations or until convergence criteria are met.
7. Output the Best Solution: Track and output the best solution found during the iterations.

Algorithm:



inspired
the
behaviors
optimization

the wolf

the solution
the alpha

(hunters),

the wolves

target
1

from

the solution

by fitness value

fitness = path length + obstacle length
Lower the fitness means better path

- 4) Update position as wolf moves tracking the prey

$$D = |C \cdot X_p - X|$$

$$X(t+1) = X_p - A \cdot D$$

where X_p = prey's position (best solution)

X = wolf's position (candidate solution)

A, C = coefficient vector controlling
exploration and exploitation

- 5) Keep all the points inside valid area
- 6) Repeat evaluation and update ~~for~~ until convergence
- 7) Return alpha wolf which is optimal drone path

Input

Start coordinate (x, y) : 0 0

Target coordinate (x, y) : 10 10

No of candidate path (wolves): 10

No of iterations: 20

Number of waypoints: 5

Number of obstacle: 2

Obstacle 1 (x, y) : (5, 5)

Obstacle 2 (x, y) : (7, 7)

→ uniform
 Mutation → Bit Flip
 → Swap mutation
 → Gaussian Mutation

Particle Swarm Optimization

Output

waypoint 1: (0, 0)
 waypoint 2: (2.2, 2.1)
 waypoint 3: (4, 3.9)
 waypoint 4: (7.5, 7.9)
 waypoint 5: (10, 10)

Total fitness: 14.80

12/10/25

12/10/25

PCA
 where
 given
 inter
 solu
 win

PCA

1) Sp

2) S

3) P

4)

5)

Code:

```
import numpy as np
```

```
class GreyWolfOptimizer:
```

```
    def __init__(self, func, lb, ub, dim, pop_size=30, max_iter=100):
```

```
        """
```

```
        :param func: Objective function to optimize
```

```
        :param lb: Lower bounds of the search space
```

```
        :param ub: Upper bounds of the search space
```

```
        :param dim: Number of dimensions (variables) in the search space
```

```
        :param pop_size: Number of wolves (population size)
```

```
        :param max_iter: Maximum number of iterations
```

```
        """
```

```
        self.func = func
```

```
        self.lb = lb
```

```
        self.ub = ub
```

```
        self.dim = dim
```

```
        self.pop_size = pop_size
```

```
        self.max_iter = max_iter
```

```
        self.position = np.random.uniform(low=self.lb, high=self.ub, size=(self.pop_size, self.dim))
```

```
        self.fitness = np.apply_along_axis(self.func, 1, self.position)
```

```
        self.alpha_pos = np.zeros(self.dim)
```

```
        self.alpha_score = float("inf")
```

```
        self.beta_pos = np.zeros(self.dim)
```

```
        self.beta_score = float("inf")
```

```
        self.delta_pos = np.zeros(self.dim)
```

```
        self.delta_score = float("inf")
```

```
    def optimize(self):
```

```
        for t in range(self.max_iter):
```

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

```
                fitness_val = self.func(self.position[i])
```

```
                if fitness_val < self.alpha_score:
```

```
                    self.alpha_score = fitness_val
```

```
                    self.alpha_pos = self.position[i]
```

```
                elif fitness_val < self.beta_score:
```

```
                    self.beta_score = fitness_val
```

```
                    self.beta_pos = self.position[i]
```

```
                elif fitness_val < self.delta_score:
```

```
                    self.delta_score = fitness_val
```

```
                    self.delta_pos = self.position[i]
```

```

a = 2 - t * (2 / self.max_iter)
for i in range(self.pop_size):
    A1 = 2 * a * np.random.random(self.dim) - a
    C1 = 2 * np.random.random(self.dim)
    D_alpha = np.abs(C1 * self.alpha_pos - self.position[i])
    X1 = self.alpha_pos - A1 * D_alpha

    A2 = 2 * a * np.random.random(self.dim) - a
    C2 = 2 * np.random.random(self.dim)
    D_beta = np.abs(C2 * self.beta_pos - self.position[i])
    X2 = self.beta_pos - A2 * D_beta

    A3 = 2 * a * np.random.random(self.dim) - a
    C3 = 2 * np.random.random(self.dim)
    D_delta = np.abs(C3 * self.delta_pos - self.position[i])
    X3 = self.delta_pos - A3 * D_delta

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

    self.position[i] = np.clip(self.position[i], self.lb, self.ub)

    print(f'Iteration {t + 1}/{self.max_iter}, Best Score: {self.alpha_score}')

return self.alpha_pos, self.alpha_score

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

lower_bound = -5.0
upper_bound = 5.0
dim = 30
pop_size = 50
max_iter = 10

gwo = GreyWolfOptimizer(func=sphere_function, lb=lower_bound, ub=upper_bound, dim=dim,
pop_size=pop_size, max_iter=max_iter)

best_position, best_score = gwo.optimize()

print(f'Best Position: {best_position}')
print(f'Best Score: {best_score}')

```

Program 7

Parallel Cellular Algorithms and Programs:

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

Implementation Steps:

1. Define the Problem: Create a mathematical function to optimize.
2. Initialize Parameters: Set the number of cells, grid size, neighborhood structure, and number of iterations.
3. Initialize Population: Generate an initial population of cells with random positions in the solution space.
4. Evaluate Fitness: Evaluate the fitness of each cell based on the optimization function.
5. Update States: Update the state of each cell based on the states of its neighboring cells and predefined update rules.
6. Iterate: Repeat the evaluation and state updating process for a fixed number of iterations or until convergence criteria are met.
7. Output the Best Solution: Track and output the best solution found during the iterations.

Algorithm:

15/11/25

PCA (Parallel Circular Algorithms)

PCA are inspired by cellular automata - systems where many simple cells update their states simultaneously based on local rules and neighbour interactions. In PCA each cell represents a candidate solution and cells update their states simultaneously using local information from neighbouring cells.

PCA for Optimization problems

- 1) Specify the objective function $f(x)$ that you want to optimize (minimize or maximize)
- 2) Set the size of grid, total number of cells, neighbourhood structure (3x3) and the number of times algorithm needs to run
- 3) Randomly generate the initial candidate solution for each cell and solution should be inside the defined search space
- 4) For each cell, compute the fitness by applying the objective function $f(x)$ to its current solution
- 5) For each cell:
 - Gather the solutions and fitness values of its neighbouring cells
 - Identify the neighbour(s) with the best fitness
 - Update the cell's solution based on neighbour, for example by moving toward's the average or best neighbour's solution

Particle swarm Optimization

Pos

PSO

$$v_{i,t+1} = v_{i,t} + \underbrace{c_1 r_1}_{\text{cognitive}} + \underbrace{c_2 r_2}_{\text{social}} + \dots$$

- 6) Repeat the fitness evaluation and state update steps for the given number of iterations or until convergence criterion is met
- 7) Throughout the process, track and output the cell with the best fitness value found

Input

grid-rows = 10

grid-columns = 10

neighbourhood-size = 3

num-iterations = 20

dim = 1

search-space bound (-10, 10)

Objective function (n) $f(x) = x^2 - 8x + 5$

Output

$f(x) = x^2 - 8x + 5$

Goal: Minimize $f(x)$

Iteration 1: Best Fitness = -10.15

Iteration 2: Best Fitness = -10.98

Iteration 3: Best Fitness = -10.99

⋮

Iteration 19: Best Fitness = -11.00

Iteration 20: Best Fitness = -11.00

Best solution found $x = 4.00$

$f(x) = -11.00$

Sp. P.
27/10/23

Code:

```
import numpy as np
from multiprocessing import Pool

def edge_rule(subgrid):
    out = np.zeros_like(subgrid)
    rows, cols = subgrid.shape
    for i in range(1, rows-1):
        for j in range(1, cols-1):
            gx = (
                subgrid[i-1, j+1] + 2*subgrid[i, j+1] + subgrid[i+1, j+1] -
                subgrid[i-1, j-1] - 2*subgrid[i, j-1] - subgrid[i+1, j-1]
            )
            gy = (
                subgrid[i-1, j-1] + 2*subgrid[i-1, j] + subgrid[i-1, j+1] -
                subgrid[i+1, j-1] - 2*subgrid[i+1, j] - subgrid[i+1, j+1]
            )

            grad = np.sqrt(gx**2 + gy**2)
            out[i, j] = 255 if grad > 13 else 0

    return out

def get_chunks_with_overlap(img, num_workers):
    height = img.shape[0]
    chunk_size = height // num_workers
    chunks = []
    for i in range(num_workers):
        start = i * chunk_size
        end = (i + 1) * chunk_size if i < num_workers - 1 else height

        if i > 0:
            start -= 1
        if i < num_workers - 1:
            end += 1

        chunks.append(img[start:end, :])
    return chunks

def parallel_edge_detection(img, num_workers=4):
    chunks = get_chunks_with_overlap(img, num_workers)
    with Pool(num_workers) as p:
        processed_chunks = p.map(edge_rule, chunks)

    results = []
```

```

for i, chunk in enumerate(processed_chunks):
    if i > 0:
        chunk = chunk[1:]
    if i < num_workers - 1:
        chunk = chunk[:-1]
    results.append(chunk)

return np.vstack(results)

if __name__ == "__main__":
    import imageio
    import matplotlib.pyplot as plt

    img = imageio.imread('path_to_image.jpg', mode='L')

    edges = parallel_edge_detection(img)

    plt.imshow(edges, cmap='gray')
    plt.show()

```

Output:

