VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB RECORD

Bio Inspired Systems (23CS5BSBIS)

Submitted by

B Vatsal (1BM23CS061)

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 **B Vatsal (1BM23CS061),** 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.

Raghavendra C K Assistant Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE

Index

GitHub Link:
https://github.com/Vatsalshetty/Bio_Inspired_Systems

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:

Bafna Gold
Date: Page:

Genetic Optimization Algorithm

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

$$Prob = \frac{f(x)}{\Sigma f(x_i)} \qquad \text{Expected Output} = \frac{f(x_i)}{Avg(\Sigma f(x))}$$

eg: ① $x \rightarrow 0-31$

② Starting

No	Initial Population	x value	fitness $f(x) = x^2$	Prob	% Prob
1	01100	12	144	0.1247	12.47
2	11001	25	625	0.541	54.11
3	00101	5	25	0.0216	2.16
4	10011	19	181	0.3126	31.26
Sum			1155	1.0	100
Avg			288.75	0.25	25
Max			625	0.541	54.11

Expected Output	Actual count
0.49	1
2.16	2
0.08	0
1.25	1
4	
1	
2.16	

③ Selecting mating pool.

String No.	Mating pool	Crossover pt	Offspring after crossover	x value	f(x)
1	01101	4	01101	13	16
2	11001		11000	24	576
3	11001	2	11011	27	729
4	10011		10001	17	289
Sum					1763
Avg					440.7
Max					729

Crossover:

Crossover point is chosen randomly

Mutation:

String No.	Offspring after crossover	Mutation Chromosome	Offspring after mutation	x value	f(x)
1	01101	10000	11101	29	5
2	11000	00000	11000	24	5
3	11011	00000	11011	27	7
4	10001	00101	10100	20	400
Sum					2544
Avg					636.5
Max					841

```
best = population [best_idx]
best_fitness = fitness_scores [best_idx]

* Check for convergence
    if is_converged (population, prev_population):
        unchanged_gen += 1

    else:
        unchanged_gen = 0.

    if unchanged_gen >= patience:
        print the generation.

return best, best_fitness.
```

Output :

Generation 1 : Best fitness = 0.9683, Selected features = [...]
Generation 2 : Best fitness = 0.9683, Selected features = [...]

⋮

Generation 23 : Best fitness = 0.97192..., Selected features = [...]

⋮

Generation 50 : Best fitness = 0.971929..., Selected features = [...]

Optimal features : [1, 3, 4, 7, 8, 9, 13, 14, 16, 17, 21, 23, 24, 27, 29]

Best accuracy : 0.9719

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:

Bafna Gold
Date: Page:

Gene Expression Algorithm for Travelling Salesman Problem

Algorithm:

1→ Initialize parameters
- Population size (pop_size)
- Number of generations (max_generations)
- Crossover rate (crossover_rate)
- Mutation rate (mutation_rate)
- Distance Matrix (distance).

2→ Generate Initial Population.
- Create a random initial population of solutions (tours)
- Each individual in population is possible solution (tour)

3→ Evaluate Fitness
- for each individual:
  - calculate distance
  - shorter distance = higher fitness.

4→ Repeat for max generations:
- for each generation:
  - selection:
    - select the subset of individuals via a selection model.
  - crossover
    - for each pair perform crossover via crossover method and create new offspring by combing both parents.

Mutation :

for each individual apply mutation with prob mutation rate randomly swap two cities in Tour (2-opt mutation)

Evaluate fitness of new population

calculate fitness for each individual

Survival selection

Combine parent and offspring pop. select best individuals to form next generation.

Output the best solution.

Output :

Generation 0 :

Tour : [1, 3, 2, 0] Dis = 80

Tour : [1 2 3 0] Dis = 95

Tour : [1 3 2 0] Dis = 80

Tour : [1 0 2 3] Dis = 80

Generation 1 :

Tour : [2 3 0 1] Dis = 95

Tour : [3, 1 2 0] Dis = 95

Tour : [1 3 2 0] Dis = 80

Tour [1 2 3 0] Dis = 80

For 2 generations the min dist is 80

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:

Bafna Gold
Date: Page:

Particle Swarm Optimization

Algorithm:

Initialize swarm with num particles:
  For each particle:
    Randomly initialize position
    Randomly initialize velocity
    set personal best position = current position
    evaluate fitness = f(position)
    set personal best fitness = fitness.

Initialize global best position (gbest) as zero vector

Initialize global best fitness as ~~inf~~ infinity

for iteration = 1 to max iterations:

  for each particle in swarm:
    Evaluate current fitness = f(current position)

    If current fitness < personal best fitness:
      update personal best fitness
      update personal best position.

    If current fitness < global best fitness:
      update global best fitness
      update global best position.

  for each particle in swarm:
    generate two random numbers:
      rand1 and rand2 in [0,1]

update velocity

velocity = (w * velocity) +
(c1 * rand1 * (personal best position
- current position)
+ (c2 * rand2 * (global best position
- current position)

update position

position = position + velocity.

return

global best position and fitness as
solution.

Output:

Position : [0.00203 0.000783]

Fitness : 4.7709e-06

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 (alpha), the importance of heuristic information (beta), the evaporation rate (rho), 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:

Bafna Gold
Date: Page:

Ant colony Optimization for TSP

function ACO_Algorithm

Initialize_Pheromones()
// Set a small equal pheromone value on all paths.

while (termination condition)
for each ant in colony
tour = Construct_Solution for Ant
add tour to the tours iteration
end for
update pheromones (all_tours)
update_best_tour_found()
end while
return Best best_tour_found
end function

```
function construct_solution_for_Ant(ant)
    start with empty tour and place
    the ant at random city.

    while (tour is not complete):
        current_city = ant.current_location
        next_city = select_next_city(current
                                     unvisited
                                     city)
        tour.add(next_city)
        ant.move(next city)

    end while
    return tour

end function.

function update_pheromones(all_tours):

    // part A: Evaporation

    for each path (i,j) on map:
        path(i,j).pheromone = (1-rho) * pa
                                        (
                                        .phe

    end for.
```

// Part B: Reinforcement

```
for each tour in all tours:
    pheromone to add = calculate tour length
    for each path (i, j) in tour
        path(i,j).pheromone += pheromone_to_add
    end for
end for
end function
```

Output:

Best tour city indices: [

distance = [
[0, 29, 20, 21, 16],
[29, 0, 15, 17, 28],
[20, 15, 0, 28, 23],
[21, 17, 28, 0, 12],
[16, 28, 23, 12, 0]
]

Iteration 1: Best tour length = 80.012
Iteration 2: Best tour length = 80.01
:
Iteration 10: Best tour length = 80.00.

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:

Cuckoo Search Algorithm

Algo:

1. Initialize n nests (solutions) randomly.
2. Evaluate the fitness of each nest

   if fitness (new) is better than fitness (nest)
   replace the nest with new nest.

3. while (t < MaxIter):
   for each cuckoo i (i = 1 to n):
   Generate new solution by
   Levy flight from
   nest

$$x_i^{t+1} = x_i^t + \alpha \oplus \text{Levy}(\lambda)$$

   Select a random nest j
   If f(x_new) is better than
   f(nest_j):
   replace nest_j with x_new

   Abandon a fraction pa of worst nests
   and replace with new
   random

   Evaluate all nest
   update the ~~best~~ current best nest

   return x_best and f(x_best).

Input
NUM
NUM
P

Output :
Initia
fina
fina

Bafna Gold
Date: Page:

Input

NUM_NEST_TSP = 30.

NUM_ITERATIONS = 500

P_ABANDON = 0.2

$\alpha = 0.6$.

Output :

Initializing Cuckoo Search Algorithm.

final best tour = [1 0 5 2 4 3 7]

final best fitness = 15.47

fitness : Distance travelled.

Sol
17/9

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:

Grey Wolf Optimization

Pseudocode:

1. Initialize population of wolves (x_i) randomly
   - n → number of wolves
   - d → number of dimension
2. Set initial parameters
   - $a = 2$ (exploration factor)
   - Max iterations = max num of iteration
   - initialize random co-eff A and C.
3. Evaluate fitness of each wolf and identify
   - x_α = best wolf
   - x_β = second best
   - x_δ = third best.
4. For each iteration $t = 1$ to Max Iteration

   Update $a = a_2 * (1 - t / \text{Max Iterations})$

   For each wolf i in population

   Calculate random values A and C

   $A = 2 * r_1 - a$

   $C = 2 * r_2$

   Calculate the distance between wolf and 3 best wolves:

   $D_\alpha = |C_1 * x_\alpha - x_i|$

   $D_\beta = |C_2 * x_\beta - x_i|$

Update the pos of wolf

$x_1 = x_\alpha - A * D_\alpha$

$x_2 = x_\beta - A * D_\beta$

$x_3 = x_\delta - A_3 * D_\delta$

$x_i = (x_1 + x_2 + x_3)/3$

average pos.

update the fitness of all wolves after position change

Reassign x_α, x_β, x_δ based on new fitness values

After all iterations value x_α as best solution found

Output:

Fitness fn : x^2

Bound : -10 to +10

Dimensions : 1

Number of wolves : 30

Iterations : 100

Best solutions obtained is found at 0

Output

0 1: Best distance : 10

2: Best distance : 9.8

⋮

100 : 8

Best route : [0, 1, 2, 3]

Best Distance : 8

Sen

29/9

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:

Parallel Cellular Algorithms

Algo:

1. create func to optimize
2. Initialize parameters - set grid size, no of cells, iterations.
3. Initialize population randomly place cells in solution space.
4. Evaluate fitness: assess how well each cell performs
5. Update states: cells update based on neighborhood iterations
6. Iterate: Repeat process with convergence
7. Track best solutions.

Pseudocode:

```
init_pop(pop_size, solution_space):
    pop = empty list
    for i from 1 to pop_size:
        sol = gen_random_sol(solution_space)
        add sol to pop
    return pop

evaluate_f(sol):
    return f(sol)

update(sol, neig, best):
    new = sol + rc_sol - sum(neighbors)
    if f(new) < f(sol):
        return new
```

else:
return sol

num_optim(it, pop_size, sol_space):
pop = int_pop(pop_size, sol_space)
best_sol = get_best(pop)
for iteration from 1 to i+c:
for each sol in pop:
sol_fitness = eva_fitness(sol)
for each sol in pop:
neigh = get_neig(sol, pop)
sol = update_sol(sol, neigh, best_sol)

best_sol = get_best_sol(pop)

return best_sol.

O/p: ~~Image noise reduction~~
10 tags [3 7 2 6 4 5 9 8 2 1]

Best scheduling: [0, 2, 1, 1, 0, 1, 2, 0]
Best fitness : 15.

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()
```