observation

INDEX

Name Nithya Lakshmi. V Std Sec 5D

Roll No. Subject Bio Inspired LAB School/College

School/College Tel. No. Parents Tel. No.

Sl. No.	Date	Title	Page No.	Teacher Sign / Remarks
1.	8/10	LAB-1 : Genetic Algo.		✓ 10
2.	18/10	LAB-2 : Particle Swam opt		✓ 8
3.	25/10	LAB-3 : Ant colony		✓ 8
4.	28/10	LAB-4 : cuckoo search		✓ 8
5.	4/12	LAB-5 : Grey wolf		✓ 8
6.	4/12	LAB-6 : Parallel cellular		✓ 8
7.	4/12	LAB-7 : GEA		✓ 8

Date __/__/__
Page _____

Genetic Algorithm for Optimigation

Algorithm

1) Randomly initialize populations p
2) Determine fitness of population
3) Until convergence repeat:
   a) Select parents from population
   b) Crossover and generate new population
   c) perform mutation on new popula-tion
   d) Calculate fitness for new population

```
// Initialize population
population = initialize_population(N)
calculate_fitness(population)

while not converged:
    // select parents based on fitness
    parents = select_parents(population, fitness)
    // create new population through crossover
    new_population = []
    while size(new_population) < size(population):
        parent1, parent2 = random_selection(parents)
```

Date __/__/__
Page _____

offspring1, offspring2 = crossover (parent1, parent2)
new_population.append(offspring1)
new_population.append(offspring2)

// Mutate new population
for individual in new_population:
    if random() < mutation_rate (probability):
        mutate(individual)

// calculate fitness for new population
new_fitness = calculate_fitness (new_population)

// Optionally replace old population with new population
population = new_population

Code:

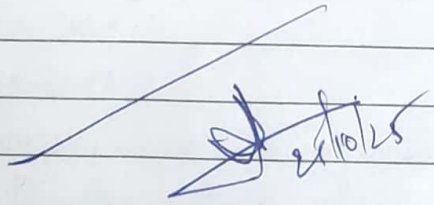
```
import random
def fitness_function(x):
    return x**2
population_size = 500
mutation_rate = 0.01
crossover_rate = 0.7
generations = 5
lower_bound = -10
upper_bound = 10
def create_population():
    population = [random.uniform(lower_bound, upper_bound)
```

Date __/__/__
Page _____

```
for _ in range(population_size)]
return population
def evaluate_fitness(population):
    fitness_scores = [fitness_function(
                        individual)
def select(population, fitness_scores):
total_fitness = sum(fitness_scores)
selection_probs = [score / totalfitness
                   for score in fitness_
                                scores]
return selected
def crossover(parent1, parent2):
  if random.random() < crossover_rate:
    child1 = (parent1 + parent2) / 2
    child2 = (parent1 + parent2) / 2
  else:
  child1, child2 = parent1, parent2
  return child1, child2
```

output:

Generation 1: Best fitness = 99.8289...
Best Individual = 9.99144



Date __/__/__
Page _____

particle Swarm Optimization

Algorithm

1) Initialize parameters:
- Define the no. of particles N
- Set the max. no. of iterations T
- choose the ~~~~ problem dimensions D
- Initialize the positions X and velocity V
- Initialize pBest for each particle as its initial position
- Initialize gBest as the best position among all pBest.

2) Evaluate Fitness:
- for each particle i (1 to N):
  calculate the fitness f(xi) using fitness fn

3) Update pBest & gBest:
- for each particle i:
  if f(xi) < f(pBesti) update pBesti = Xi
- Update gBest:
  if f(pBesti) < f(gBesti) set gBest = pBest; for the particle with the best fitness

4) Update Velocities & positions
for each particle i:

$$V_i = wV_i + c_1 s_1 (pBest_i - x_i) + c_2 s_2 (gBest$$

Date __/__/__
Page _____

$x_i = x_i + v_i$

5) If max no. of iterations T is reached, stop else go Back to ~~de~~ step 2

Code:

```
import numpy as np

class Particle:
    self.__init__(self, bounds):
        self.position = np.random.uniform
                        (bounds[0], [i]
        self.velocity = np.random.uniform(-1,1)
        self.

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

def particle = S=0 (dim, bounds,
                    n_pa, n_itel):
    particle = np.random.uniform
               (bound[0], bound[1],
                (n_particles, dim))

    velocities = np.random.uniform
                 (1, 1, (n_particles
                         dim))

    personal_best_pos = np.array
                        (sphere_fun(x)
                         for p in particles))
```

Date __/__/__
Page _____

```
global_best_pos = personal_b_s[np.argmin(personal_b, v)]
for in range (n_iterations)
    r1, r2 = np.random.rand(dim), np.random.rand(dim)

velocities = c0.T * velocities + 1.0 *
                * c1 *
                (personal_b_p - particle) +
                1 * c2 * (global_b_p - particle))
particle += velocities
particle = np.clip(particles, bounds[0])
values = np.array([[sphere_fun(p) for p in particles]
betterment = values < personal_b_v
personal_best_p = [better_values] = particles
global_b_pos = personal_best_p[np.argmin(personal_b_v)]
return global_best_pos

dim = 2
bounds = (-10, 10)
n_particle = 30
n_iteration = 30
best_pos, best_value = particle_
```

Date __/__/__
Page _____

```
Crop (dim, bounds, n_particles,
                    n_iterations)
print("Best position:", best_pos)
print("Best value =", best_val)
```

Output

Best position = [0.002 -0.001]
Best value = 0.000004

Date __/__/__
Page _____

Ant colony optimization for the Traveling Salesperson Problem

Algorithm

1) initialize parameters:
no. of ants, pheromone, heuristic, evaporation rate, pheromone level

2) Construct Solution:
- each ant starts at a random city.
- Select the next city based on pheromone level, and heuristic
- Repeat until all cities are visited forming a completed tour

3) Calculate the total distance for each ant tour, Track the best tour found so far.

4) Update pheromones:
Evaporate pheromones globally
Deposite new pheromones on edges proportionally to the quality of each ant's tour.
Repeat for fixed iterations

5) output : The Best tour.

Date __/__/__
Page

Code:-

```
import numpy as np
import random

def aco_tsp(d, n, i, a, b, r):
    c = np.ones((n, n))/n
    t = None
    d = float('inf')
    for _ in range(i):
        s = []
        for _ in range(n):
            v = set()
            u = [random.randint(
                0, n-1)]

            v.add(u[0])
            while len(u) < n:
                x = u[-1]
                p = []
                t = 0
                for y in range(n):
                    if y not in v:

p.append((c[x,y]**a)*
         (1/d[x,y]**b))

                t += p[-1]
                p = [z/t for z in
                     p]

            u.append(random.choices
                     (range(n),
                      weights=p,
                      k=1)[0])
```

```
        v.add(u[-1])
        s.append(u)
    c* = (1-8)
    for u in s:
        d_u = sum(d[u[i]], u[i+1]]
    for i in range(n-1)) + d[u[-1],
                            u[0]]
        if l_u < l:
            l = l_u
            t = u
        for i in range(n-1):
            c[u[i], u[i+1]] += 1/l_u
        c[u[-1], u[0]] += 1/l_u
    return t, l

d = [[0,2,9,5], [2,0,6,3],
     [9,6,0,8], [5,3,8,0]]
```

distance Matrix

```
t, l = aco_tsp(d, 10, 100, 1.2, 0.1)
print("best tour:", t)
print("Best distance:", l)
```

output:

Best tour : [0, 1, 3, 2, 0]
Best Distance : 16.0

Date __/__/__
Page

Cuckoo Search

Algorithm

1) Initidization :

State with a set of initial solutions (nests)

2) cuckoo egg laying:
generate new solutions by slightly modifying existing solutions through a random walk called levy flight.

3) Solution Evalution
Evaluate the quality of all solutions (cuckoo eggs and host nests)

4) Selection :

Keep the best solution (nests) for the next generation

5) iteration
Repeat steps 2 - 5 for a specified number of iterations.

6) Termination
The algorithm stops when a stopping criterion is met

Date __/__/__
Page _____

code

```
import numpy as np

def l(b=1.5):
    s = (gamma(1+b)*np.sin(
        np.pi*b/2)/
        gamma((1+s)/2)*
        np.sin(np.pi*b
        /2))**
        (1/b)
    u,v = np.random.randn(2)
    return s*u/abs(v)**(1/b)

def c(f, b, n, p, i):
    d = len(b[0])
    s = np.random.rand(n,d)*(b[:,
    1]-b[:,0])+b[:,0]
    t = s[np.argmin([f(x) for x
        in s])]
    t_f = f(t)
    for _ in range(i):
        for j in range(n):
            x = s[j] + l()*(s[j]-t)
            x = np.clip(x, b[:,0],
                b[:,1])
            x_f = f(x)
            if x_f < f(s[j]):
                s[j] = x
            if np.random.rand()<p:
                s[j] = np.random.
                rand(d)*
                (b[:,1]-b
                [:,0])
```

```
for j in range(n):
    if f(S[j]) < t_f:
        t = S[j]
        t_f = f(t)
return t, t_f
print(f"Best Solution: {best_solution}")
print(f"Best fitness: {best_fitness}")
```

output:

Best Solution: [1.95581557,
-1.27762143,
0.59020481,
3.72688103,
0.79926703,
-0.5427445,
-0.96002714,
-0.36067229,
3.18591725,
1.31872175]

Best Fitness: 33.452598 04-
-839685

Date __/__/__
Page _____

GREY WOLF OPTIMIZER

Algorithm

1. input : Define parameters
num, wolves, dimension, bounds, max_iter, objective_function

2. initialization :

$X_{ij} = min\_bound + rand() * (max\_bound - min\_bound)$

Compute the fitness values

$f(x) = \sum x_i^2$

alpha_pos : Best wolf
bet_pos : Second Best
delta_pos : Third Best

3. for t = 1 to max_iter :
update $a = 2 - (2 * t) / max\_iter$

for each wolf :
$A = 2 * a * rand() - a$
$C = 2 * rand()$

Compute the distance b/w the wolf and the three leaders

4. Evaluate fitness : compute fitness of updated pop.

5. Output : alpha_pos and alpha_score

Date __/__/__
Page

code:-

```
import numpy as np

def gwo(f, lb, ub, n=5, i=500):
    d = len(lb)
    a = lambda l : 2 - l * (2 / i)
    X = np.random.rand(n, d) *
        (ub - lb) + lb

    a, b, d = X[np.argmin([f(x) for
        x in X])], X[np.argmin([f(x)
        for x in X])], X[np.argmin
                [f(x)
                for x in X]]

    for _ in range(i):
        for j in range(n):
            if f(X[j]) < f(a) and
(a, f(a)) := (X[j], f(X[j]))
            f(X[i]) > f(a) and
            f(X[i]) > f(b) and f(X[i]) <
                        f(d)
    for _ in range(max_ite)
        a = 2 - t * (2 / max_ite)
        population = update_pos
                (pop, obj_pos),
        fitness = eval - fitness(pop,
                        obj_func)

        for i in range(num_wolves):
            if fitness[i] < a_score:
                a_score = fitness[i]
```

```
        a_pos = pop[i]
    if fitness[i] < b_score
        b_score = fitness[i]
        b_pos = pop[i]

return a_pos, a_score
num_wolves = 30
dim = 5
bound = (-10, 10)
max_iter = 50

print(f" Best Solution = {best solution}")

print(f" Best score : {best score}")
```

output:

iteration 1150, best fitness : 46.87190

.
.
.

Best solution : [2.51916187 -1.83354 1]

Best score : 2.0959372052314 e-09

Date __/__/__
Page _____

Parallel Cellular

Algorithm

1. initialize parameters:
num_cells, grid_size, iterations, neighbor_size, convergence

2. Initialize population:
generate a population of num_cells with random position within the grid space each position is between -10 & 10

3. evaluate fitness:
for each cell, calculate fitness using optimization function $f(x) = x^2$

4. Update state:
for each cell, find the neighboring cells, and update the cells position based on the best neighbour

5. iterate:
for i = 1 to max_iter

6. output:
Best solution and Best value

Date __/__/____
Page _____

Code:

```
import numpy as np

def opti(n):
    return x**2

num_cells = 100
grid_size = 10
iter = 100
neig_size = 3
conve_thres = 0.0001
np.random.seed(42)
popn = np.random.uniform(-10, 10,
                         num_cells)

def eval_f(popn):
    return np.array([opti(cell) for
                     each cell in pop])

fitness_val = eval_f(pop)

for i in range(iter)
    fitness_val = eval_f(pop)
    min_idx = np.argmin(fitness.
                        val)
    min_fitness = fitness.val[min_idx]
    best_sol = pop[min_idx]

if iter > 0:
    improvement = prev_f - min_fitness
pop = update(pop, fitness_val)
prev_f = min_fitness
```

Date
Page

trap.

if improvement < convergence:
    print(f"Converged at iteration {i+1}")
    break

print(f"Best solution: {best_sol}")
print(f"Best fitness: {min_fitness}")

output

Iteration 1: Best fitness: 0.000980
0487
Best solution: -0.0964179 7777
.
.
.
.
Converged at iteration 20 with
fitness, 2.2052116220 5187e-08
Best solution: 0.0069596808
Best fitness: 2.2052116220 51e-8

Date __/__/__
Page _____

Gene Expression Algorithm

Algorithm

1. initialize parameter:
pop_size, num_genes, mutation_rate, cross_rate, num_gen

2. generate an initial population of individual with genes within bound

3. Evaluate fitness of each individual using objective function $f(x) = 10 \times n + \sum (x_i^2 - 10 \times \cos(2\pi x_i))$

4. Use roulette wheel selection to choose parent

5. crossover: apply 2 point crossover offspring

6. Apply mutation: Randomly change one of the gene by adding random value

7. iterate for set no. of generations

8. Return Best solution & fitness value.

Date __/__/__
Page

Code:

```
import numpy as np

def obj(x):
    return 10 * len(x) + sum(x**2 -
                10 * np.
                cos(2 * np.
                pi * x)
                for x in x)

pop_size = 100
mut_g = 10
c_r = 10.7
n_gen = 100
m_r = 0.01
popn = np.random.uniform
        (-5.12, 5.12)

def eval(popn):
    return np.array([obj(ind) for
                ind in popn])

def sel(popn, fit):
    f_sum = np.sum(fit)
    prob /= np.sum(prob)
    s_indices = np.random.
                choice(np.
                arange(len
                (popn)))
    return popn[sel_indices]
```

```
def mutation(popn):
    for i in range(pop_size):
        if np.random.rand() < m_r:
            m_p = np.random.randint(0, num_gene)
            popn[i, m_p] += np.random.uniform(-0.5, 0.5)
    return population

for generation in range(num_gen):
    fit = eval(popn)
    pop = selection(popn, fit)
    pop = crossover(popn)
    pop = mutation(popn)
    best_sol = popn[np.argmin(fit)]

    print("{}^ generation {} generation {}
                                    +{}

Best solution : {best_solution}
    fitness = {min(fit)}

final_f = eval(pop)
Best_sol = popn[np.argmin
                    (final_f)]
print("{}^ find Best Solution
                    {Best_sol}
    fitness : {min(fit)}")
```

Date __/__/__
Page _____

output

Generation 1: Best solution:
[-0.01980865
-4.61178376
-3.83458543
3.54614127
1.04550891]

Final Best solution: [0.8997071
2.74312455
1.84377788 -2.0993
2.34789789
2.64422209]