INDEX

1BM22CS353

NAME: Ankit STD.: SEC.: ROLL NO.: SUB.: BIS LAB

S. No.	Date	Title	Page No.	Teacher's Sign / Remarks
1.	26/9/24	Genetic Algorithm	10	3/10/24
2.	3/10/24	Particle Swarm optimization		
3.	24/10/24	Genetic Algorithm	10	24.10
4.	7/11/24	Particle Swarm optimization	10	07-11
5	14/11/24	Ant colony optimization	10	14.11
6.	21/11/24	Cuckoo search optimization	10	21.11
7	28/11/24	Grew wolf optimization	10	28.11
8.	19/12/24	Parallel cellular Algorithm		
9.	19/12/24	Optimization via Gene Expression Algorithm		

24/10/24

① IMPLEMENT A GENETIC ALGORITHM TO MAXIMIZE A FUNCTION ($f(x) = x^2$)

```
import numpy np

def objective_function(x):
        return    x**2

population_size = 100
mutation_rate = 0.01
crossover_rate = 0.7
num_generations = 50
x_min = -10
x_max = 10

def initialize(size):
      return np.random.uniform(x_min, x_max, size)

def  evaluate(population):
          return objective_function(population)

def  select(population, fitness):
     selected_indices = np.random.choice(len(population),
          size = 2, replace = False)
          return population[selected_indices[np.argmax(
          fitness[selected_indices])]]

def  crossover(parent1, parent2):
        if  np.random.rand() < crossover_rate:
              return (parent1 + parent2) / 2
        return parent1
```

Scanned with CamScanner

```
def mutate(individual):
    if np.random.rand() < mutation_rate:
        return individual + np.random.uniform
        (-1,1)
    return individual

def genetic_algorithm():

    population = initialize pop (population_size)
    best_soln = none
    best_fitness = -np.inf

    for generation in range(num_generation):
        fitness = evaluate_fitness(pop_size)

        curr_best_idx = np.argmax(fitness)
        if fitness[curr_best_idx] > best_fitness
            best_fitness = fitness[curr_best_idx]
            best_soln = population[curr_best_idx]

    return best_fitness, best_soln.

best_x, best_value = genetic_algorithm()
print("Best x: {best_x}, Max func. value: {best_
value}")
```

24.10.

OUTPUT:

Best x: 9.9620 3495
Best value: 99.2534684

classmate
Date
Page

PARTICLE SWARM OPTIMIZATION

7/11/24

```
import numpy as np
import matplotleb.pyplot as plt

def objective_function(x):
        return x**2 + 4* x + 4

num_particle = 30
dimensions = 1
iterations = 100
w = 0.5
c1 = 1.5
c2 = 1.5

#

for iter in range('iterations'):
        for p in range(num_particle):
                r1 = np.random.rand()
                r2 = np.random.rand()
                velocities [P] = w* velocities[P] + c1
                        * r1*(personal_best_pos+[P] - pos[P]
                        - pos[P]) + c2 * r2 * (global_best
                        - positions[P])

                pos[P] = pos[P] + velocities [P]
                current_score = objective_function(
                                positions [P])

                if current_score < personal_best_scores[P]:
                        personal best_scores [P] = current score
                        personal best_pos[P] = pos[P]

        if iteration % 10 = =0:
```

```
    print("Iteration {iteration}: Global Best
    score={global_best_score}")

print("Final Global Best Position: {global_best_pos}")
print('Final Global Best Score: {global_best_score}')

x = np.linspace(-10, 10, 400)
y = objective_function(x)

plt.plot(x, y, label="objective function:
        f(x) = x^2 + 4x + 4", color='blue')

plt.legend()
plt.xlabel("x")
plt.ylabel("f(x)")
plt.show()
```

output:

Iteration 10: Global Best Score = [9.52e-09]
Iteration 20: Global Best Score = [6.91e-12]
:
:
Iteration 50: Global Best Score = [1.51e-13]
Iteration 60: Global Best Score = [0.]

Final Global Best Position: [-2.]
Final Global Best Score: [0.].

02-11-24

ANT COLONY OPTIMIZATION :

```
import random
import math
import numpy as np

def distance (c1, c2)
    return math.sqrt((c1[0] - c2[0])** 2 +(c1[1]-
           c2[1]) **2)

class antColony :
    def _init__ (self, cities ,num_ants, alpha ) :
            self.cities = cities
            self. numcities = len (cities)
            self . alpha = alpha

        for i in range (self.numcities ):
            for j in range (i+ 1, self.num_cities ):
                self.distances[i][j] = distance (self.
                cities [i] , self.cities [j])
                self . distance[i][j] = self.distances[i]
                [j]

        def probability (self, ant , city, visited ) :

            pheromone = self. pheromone [city ]
            heuristic = np. array( [1.0 / self. distances
            [city][i] if i not in visited else 0
            for i in range (self.num cities )])
            return pheromone_heuristic / pheromone_
                   heuristic . sum () .
```

```
def run(self):

    best_distanu = float('inf')
    best_tour = None

        while len(visited) < self.numcities :
        city = visited[-1]
        prob = self.probablity(ant, city, visited)
        next_city = np.random.choice(range(
              self.numcities), p=prob)
        visited.append(next_city)

    total d += self.distances[visited[-1]][visited[0]]

    all_tours.append(tour)
    all_distances.append(total_distance)

    if total d < best_distance :
        best_distance = total d
        best_tour = tour

    return best_tour, best_distance

if __name__ == "__main__":

    cities = [(0,0), (1,3), (4,3), (6,1), (6,5)
        , (2,7), (7,4), (5,2)]

    num_ants = 10
    alpha = 1.0
    beta = 2.0
    rho = 0.1
```

classmate
Date
Page

print ("Best tour: ", best_tour)
print ("Best distance: ", best_distance)

output:

Best tour: [0, 1, 6, 5, 4, 2, 7, 3]
Best Distance: 24.772376022

21/11/24 CUCKOO SEARCH ALGORITHM

```
import numpy as np
import math

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

def levy_flight(lamba, d):

    sigma_u = (math.gamma(1 + lamba) * np.sin(
              np.pi * lambda/2) /
              (math.gamma((1+ Lambda)/2) *
                lambda * 2 ** ((lambda-1)/2)))
              ** (1/lambda)

    u = np.random.normal(0, sigma_u, d)
    v = np.random.normal(0, 1, d)
    step = u/np.abs(v) ** (1/lambda)
    return step

def cuckoo_search(func, n_nest, n_dim, max_iter,
                  pa = 0.25, lambd_levy = 1.5):

    nests = np.random.uniform(-5, 5, (n_nests, n_dim
    fitness = np.apply_along_axis(func, 1, nests)

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

    for iteration in range(max_iter):
        new_nests = nests + alpha * levy_flight
        (lamba_levy, n_dim)
```

```
new_nests = np.clip(new_nests, -5, 5)

new_fitness = np.apply_along_axis(func, 1,
              new_nests);

for i in range(n_nests):
    if np.random.rand() < pa:
        nests[i] = new_nests[i]
        fitness[i] = new_fitness[i]

n_nests = 50
n_dim = 10
max_iter = 100
pa = 0.25
alpha = 0.01
lambda_levy = 1.5
best_sol, best_val = cuckoo_search(n_nests, n_dim)
print("Best soln Found :", best_sol)
print("Best fitness value :", best_val)
```

Output:

Best Soln Found: [-1.0767, 2.064, -0.483, -1.408, -0.7945, 3.1574]

Best Fitness Value: 30.9679

classmate
Date
Page

28/11/24 GREY WOLF OPTIMIZATION

```
import numpy as np
def sphere(x):
    return np.sum(x**2)

class GWO:
    def __init__(self, obj_func, dim, lb, ub):
        self.obj_func = obj_func
        self.dim = dim
        ~~self.pop_si~~
        self.ub = ub
        self.lb = lb

    def update_position(self, alpha, beta, delta):

        r1 = np.random.random(self.dim)
        r2 = np.random.random(self.dim)

        D_alpha = abs(C[0] * r1 - position - alpha)
        D_beta = abs(C[1] * r1 - position - beta)
        D_delta = abs(C[2] * r1 - position - delta)

        ~~x1 = a1~~

    def optimize(self):

        for t in range(self.max_iter):
            a = 2 - t * (2 / self.max_iter)
            A = np.random.uniform(-a, a, 3)
            C = np.random.uniform(0, 2, 3
```

Scanned with CamScanner

```
for p in range (self.pop_size):
        fitness = self.obj_func(self.position
                                          [p])
    if fitness < self.alpha_score:
        self.alpha_score = fitness
        self.alpha_pos = self.position[p]

    elif fitness < self.beta_score:
        self.beta_score = fitness
        self.beta_pos = self.position[p]

    elif fitness < self.delta_score:
        self.delta_score = fitness
        self.delta_pos = self.position
                                    [p]

    return self.alpha_pos, self.alpha_score


dim = 10
pop_size = 50
max_iter = 100
lb = -5.12
ub = +5.12

gwo = GWO(obj_func = sphere, dim, lb=lb, ub=ub)
best_pos, best_score = gwo.optimize

print("Best soln & Best fitness:", best_pos, best_
                                        score)
```

Output: Best soln: [0.4866 0.2197 0.3349]
Best fitness: 0.0100

28.11.

19/12/24 PARALLEL CELLULAR ALGORITHMS:

Code:

```
import numpy as np
from multiprocessing import pool
def rastrigin_function(x):
        return 10* len(x) + sum([(xi ** 2 -
            10 * np.cos(2* np.pi * xi)) for xi
        in x ])

def initialize (grid_size, dimensions, lowerbound,
            upperbound):

    return np.random.uniform (lower_bound,
    upperbound, (gridd_size, grid_size, dimen
        -sions ))

def evaluate_fitness (grid):
        fitness = np.zeros ((grid.shape[0],
            gridshape[1] ))

            for i in range (grid.shape[0]):
                for j in range (grid.shape[1]):
                    fitness [i, j] = rastrigin_
        function (grid[i, j])
        return fitness.

def parallel_cellular_algorithm(grid_size, dimensions
        lower_bound, upper_bound):
    grid = initialize_population (grid_size, dimensi
        -ons, lower_bound, upper_bound)

        for i in range (iter):
            fitness = evaluate_fitness (grid)
```

```
grid= update_grid (grid, fitness)
print( f" Iteration {e+1} Best
     fitness : { fitness.min() " }

best_cell = np.unravel_index(fitness.argmin
   (), fitness . shape )
best_solution = grid[best_cell]
return best_solution , fitness .min()

if __name__ = "__main__":

   GRID_SIZE = 10
   DIMENSIONS = 2
   LOWER_BOUND = -5.12
   UPPER_BOUND = 5.12

sol, fit = parallel_cellular_algorithm)

print ( f"Best Solution : {~~best~~ sol}, Fitness : {fit}")
```

Output :

Best Solution = [-0.005 0.003],
fitness : 0.22137806183 .

19/12/24

OPTIMIZATION VIA GENE EXPRESSION ALGORITHMS

```
import numpy as np
import random
def objective_function(x):
    return x**2 + 4*x + 4

population_size = 30
num_genes = 1
mutation_rate = 0.05
crossover_rate = 0.7
num_generation = 100

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

def evaluate_fitness(population):
    return np.array([objective_function(individual[0])
        for individual in population])

def select(population, fitness):
    selected_indices = np.random.choice(len(population),
        size = 2, replace = False)
    return population[selected_indices]

def crossover(parent1, parent2):
    if random.random() < crossover_rate:
        return (parent1 + parent2)/2
    return parent1
```

```
def gene_expression_algorithm():

    population = initialize_population(pop_size, num_genes)

    best_soln = None
    best_fitness = float('inf')

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

        current_best_idx = np.argmin(fitness)
        if fitness[current_best_idx] < best_fitness:
            best_fitness = fitness[current_best_idx]
            best_soln = population[current_best_idx]

        newpop = []
        for i in range(pop_size):
            parent1 = select(population, fitness)
            parent2 = select(population, fitness)
            offspring = crossover(parent1, parent2)
            offspring = mutate(offspring)
            newpop.append(offspring)

        population = np.array(new_population)
        population = gene_expression(population)

    return bestsoln, best_fitness

bestsoln, bestfitness = gene_expression_algorithm()
print(f"Best Solution : {bestsoln}")
print(f"Best fitness : {bestfitness}")
```

OUTPUT:

Best Solution: [-2.00000]
Best Fitness: 0.0