INDEX

Name Shilpa K. M

Standard Section E Roll No.

Subject

SL. No.	Date	Title	Page No.	Teacher Sign / Remarks
01	26/9/24	Genetic Algorithm	1	10
	3/10/24	Genetic Algorithm	2-3	
	24/10/24	Genetic Algo. (Implementation)	4	10
02	7/11/24	Partical Swarm Optimization	5	10
03	14/11/24	Ant colony Optimization	6	10
04	21/11/24	Cuckoo Search Algorithm	7	10
05	28/11/24	Grey wolf Optimizer (GWO)	8	10
06	19/11/24	Parallel cellular Algo	9-10	
07	19/11/24	Gene expression Algo	11-12	

Lab-01

Genetic Algorithm for Optimization Problems:

→ A Genetic Algorithm is an adaptive heuristic search Algorithm inspired by principle of natural selection and genetics. GAs are widely used for solving optimization and search problems.

Population size, mutation rate, crossover rate and ~~numb~~

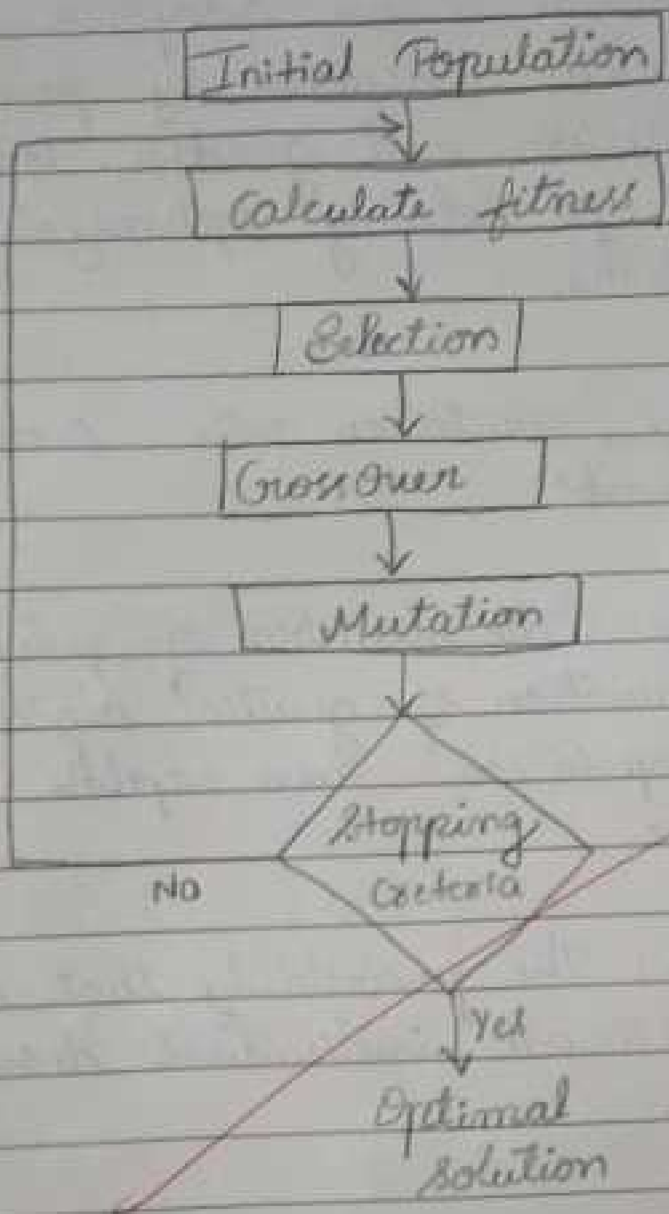
Population Size : The number of potential solution in each generation of genetic algorithm. A larger population can explore a broader solution space

Mutation rate : The probability that a mutation will occur in an individual during reproduction.

Crossover Rate : The probability that two parents solution will combine to produce offspring.

The best performing solution are selected to reproduce combining their attributes through crossover and mutation crossover mixes part of two parents solution to create offsprings, while mutation introduce random random changes to promote diversity. This cycle of selection, reproduction and evaluation continues for multiple generations as generations pro

The population tends to evolve towards better solution



20/9/24

[g-what] Algorithm:

Step 1: Intialize Parameter: Set the population size, mutation rate, crossover rate, & number of generation.

Step 2: Generate Initial Population: Create a random population of potential solutions within given bounds.

Step 3: Evaluate Fitness: Calculate the fitness of each individual in the population by evaluating objective foundation.

Step 4: Select Parents: Select the fittest individ-uals from population to reproduce, based on their fitness.

Step 5: CrossOver: Perform crossover between the selected parents to create new offspring, with a probability equal to the crossover rate.

Step 6: Mutate: Apply mutation to offspring, with a probability equal to the mutation rate, to introduce ~~…~~ new traits.

Step 7: Replace Least Fit: Replace the least fit individuals in population with new offspring.

Step 8 Repeat: Repeat step 3-7 for a fixed no. of generations & until convergence criteria are met

Step 9: Output Best Solution: Return the best solution found during the generations, which is the individual with highest fitness.

Applications of Genetic Algorithm:

(i) Optimization problems: Finding the best solution to a problem under certain constraints ex. maximizing profit, minimum cost

(ii) Function maximization/minimization: GAs can be used to find maxima & minima of complex mathematical functions

(iii) Scheduling problems: GAs are useful in resource allocation, task scheduling problems.

(iv) Machine learning & AI: GAs are sometimes used for optimizing hyperparameters in machine learning models

(vi) Engineering design: GAs help in optimization design for efficiency such as structures, ~~to~~ circuits & aerodynamics

(v) Robotics & control system: GAs can help in designing robots & optimizing control parameters for efficient performance

Optimizing Techniques

* Selection: Selecting the fittest individuals from population to reproduce

* Crossover: Combining the genetic infromation of two parents to create a new offsprings

* Mutation: Randomly changing genetic information of an individuals to introduce new traits

* Elitism: Preserving the best solution from previous generation to ensure that best solutions are not lost.

* Tournament Selection: Selecting the fittest ~~indivicals~~ individuals ~~having a higher chance of being selected~~ from a subset of population to reproduce.

* Roulette wheel station: Selecting individuals based on their fitness, with higher fitness individual having a higher chance of being selected.

* Simulated ~~bn~~ binary crossover: A crossover technique that simulates the process of binary crossover to create new offspring

S 3/10/24

Lab-03

Implement the Genetic Algorithm 24/10/25

```
import random
import numpy as np

def objective_function(x):
    return x**2 + 2*x + 1

def generate_initial_population(population_size,
                bounds):
    population = []
    for i in range(population_size):
        x = random.uniform(bounds[0], bounds[1])
        population.append(x)
    return population

def evaluate_fitness(population):
    fitness = []
    for x in population:
        fitness.append(objective_function(x))
    return fitness

def selection(population, fitness, num_parents):
    parents = []
    for _ in range(num_parents):
        max_fitness_idx = np.argmax(fitness)
        parents.append(population[max_fitness_idx])
        fitness[max_fitness_idx] = -float('inf')
    return parents

def crossover(parents, crossover_rate):
    offspring = []
    for _ in range(len(parents) // 2):
        parent1, parent2 = random.sample(parents, 2)
```

```
def mutation(offspring, mutation rate, bounds):
  for i in range(len(offspring)):
      if random.random() < mutation rate:
          offspring[i] += random.uniform(-0.1,
          0.1) * (bounds[1] - bounds[0])
          offspring[i] = max(bounds[0], min(offsp
-ring[i], bounds[1]))
  return offspring

def genetic algorithm(population size, mutation size,
crossover rate, num generations, bounds):
    population = generate initial population(popul
     -ation size, bounds)
    for generation in range(num generations):
       fitness = evaluate fitness(population):
       parent = selection(population, fitness,
                 population size //2)
       offspring = mutation(offspring, mutation
                             rate, bounds)
       offspring = crossover(parents, crossover rate)
       ~~offspring~~ = population = offspring + parents
    best_solution = min(population, key-objective
                 function)
    return best solution.

population size = 100
mutation rate = 0.01
crossover rate = 0.5
num generation = 100
bounds = (-10, 10)
```

best solution = genetic algorithm (population size, bounds, mutation rate, crossover rate, num generation)
print("Best solution:", best solution)

Output :-

Best solution = 9.9469111342383 6

Lab-02 7/11/24

Page 5

Partical Swarm for Optimization

~~PSO~~ it an Optimi

```
import numpy as np

def Sphere-function(x):
    return np.sum(x**2)

class PSO:
    def init (self, func, dim, numparticles=30,
    max_iter=100, w=0.5, c1=1.5, c2=1.5, bound=
            (-5.12, 5.12)):
        self.func = fun
        self.dim = dim
        self.numparticles = numparticles
        self.max_iter = max_iter
        self.w = w
        self.c1 = c1
        self.c2 = c2
        self.bound = bounds
        self.position = np.random.uniform(self.bounds[0],
            self.bounds[1], (self.numparticles, self.dim))
        self.velocities = np.random.uniform(-1, 1), (self.
            numparticles , self.dim))
        self.personalbest positions = np.copy (self.position)
        self.personal best score = np.array ([func(p)
            for p in self.positions])
        self.globalbest position = self.personal best_position
            [np.argmin(self.personal best score)]
        self.global best score = np.min(self.
            personalbest score)
```

```
def update velocity( self, i):
    r1, r2 = np. random. rand (2)
    congnitive velocity = self. c1 * r1 * (self. personal
            best position[i] - self. positions[i])
    social velocity = self. c2 * r2 * (self. global best
            position - self. positions[i])
    inertia. velocity = self. w * self. velocity[i]
    return interia velocity + congnitive velocity
            + social velocity

def update. position( self, i)
    Self. position[i] += self. velocities[i]
    Self. position[i] = np. clip( self. positions[i],
            Self. bounds[0], self. bounds [1])

def upd-optimize( self ):
    for iteration in range( self. max iter):
        for i in range( self. num particles):
            fitness = self. func( self. position[i])
            if fitness < self. personal best [i]
                Self. personal best [i] = fitness
                self. personal best position[i] = self. pos[i]

        for i in range (self. num particles)
            self. velocities [i] = self. update velocity(i)
            self. update position(i)

        print (f" Iteration {iteration + 1} / self. max iter,
            BestScore : {self. global best score}")

best position , best score = pso.optimize()
```

Bafna Gold

Date: Page:

```
print(f"\n optimal sol": {best_position}")
print(f"\n optimal score (objective value): {best_score}
                    {best_sites}")
```

output:-

optimal solution: [-0.001013 , -0.0010166]
optimal score : 2.0590321646

07-11-2

Lab-03

14/11/23

Ant Colony Optimization for Travelling Salesman problem

```
import numpy as np
import random
import math

~~cities~~ cities = [(0,0), (1,2)(2,4), (5,6), (7,8), (8,0)]

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

num cities = len(cities)
distance matrix = np.zeros((num cities, num cities
for i in range(num cities):
    for j in range(i+1, num cities):
        dist = euclidean distance(cities[i], cities[j])
        distance matrix[i][j] = dist
        distance matrix[j][i] = dist

num ants = 10
iterations = ~~10~~ 3
alpha = 1.0
beta = 2.0
rho = 0.5
tau0 = 10-4

pheromone matrix = np.ones((numcities,
          numcities)) * tau0
```

```
def construct sol ():
    path = [random.randint(0, numcities-1)]
    visited = set(path)

    while len(path) < numcities:
        current city = path[-1]
        probabilities = []

        for next city in range(numcities):
            if next city not in visited:
                pheromone = pheromonematrix [current
city][next city] ** alpha
                heuristic = (1.0 / distance matrix [current
 city][next city]) ** beta
                probabilities.append(pheromone * heuristic)
            else:
                probabilities.append(0)

        total_prob = sum(probabilities)
        probabilities = [p / total_prob for p in probabili
                                                   ties]
        next city = np.random.choice(range(numcities),
              p= probabilities)
        path.append(next city)
        visited.add(next city)

    return path

def update ~~phero~~ pheromones(allpaths, alllengths
    global pheromone matrix
    pheromone_matrix *= (1-rho)
```

```
for path, length in zip(all paths, all lengths)
    pheromone deposite = 1.0 / length

best path = None
best loop = float('inf')

for iteration in range(iteration)
    all paths = []
    all lengths = []

for _ in range(num ants)
    path = construct_sol()

    all paths.append(path)
    all length.append(length)

    if length < best length
      best_length = length
      best_path = path

  update_pheromones(all paths, all lengths)

  print(f"Iteration {iteration + 1}/{iterations}:
            best length = {best length}")

print("\nBest path found:", best path)
print("Best length:", best_length)
```

o/p

Iteration 1/3 : Best length = 26.96837...
Iteration 2/3 : Best length = 26.96837...
Iteration 3/3 : Best length = 26.96837...
Best path found : [0, 1, 2, 3, 4, 5]
Best length : 26.96837

14.11

Cuckoo Search Algorithm

```
import numpy as np

def objective funt(x):
    return np.sum(x**2)

def levy flight (lambda, dim):
    Step = np.random.randn(dim) * np.power
           (np.abs(np.random.randn(dim)), 1/
                lambda)
    return step

def cuckoo search (objective funt, num nest = 25,
             max iter = 100, pa = 0.25, lambda = 1.5, dim = 5):
    nests = np.random.uniform(-10, 10(num nests,
                    dim))
    fitness = np.apply-along (objective funt, 1, nests)

    bestnest = nests[np.argmin(fitness)]
    best fitness = np.min(fitness)

    for iterations in range(max iter)
      new nests = np.copy(nests)
      for i in range(numnests):
          step = levy flight (lambda, dim)
          new-nests[i] = nests[i] + step
          new-nests[i] = np.clip(new-nests[i], -10, 10)
    new fitness = np.apply-along(objective funt, 1,
                    new-nests)
      for i in range(num nests):
```

```
        if (new_fitness[i] < fitness[i]):
            nests[i] = new_nests[i]
            fitness[i] = new_fitness[i]

    for i in range(int(pa * num_nests)):
        random_index = np.random.randint(0,
                       num_nests)
        nests[random_index] = np.random.uniform
                       (-10, 10, dim)
        fitness[random_index] = objective_funt(nests
                       [random_index])

    current_best = np.min(fitness)
    if current_best < best_fitness:
        best_fitness = current_best
        best_nest = nests[np.argmin(fitness)]

    printf(f"Iteration {iteration+1}: Best fitness =
               {best_fitness}")

    return best_nest, best_fitness

best_sol, best_fitness = cuckoo_search(objective_funt)
print("\n Best solution found:", best_soln)
print("Best fitness value:", best_fitness)
```

O/P.

```
Iteration 1 : Best fitness : 58.03326
Iteration 2 : Best  -"-    : 44.467649
Iteration 3 :  -"-         : 44.497991
Iteration 4 :  -"-         : 36.590403

Best solution found : [-1.26757  -0.24178  1.64976  4.69124
                       3.17159070]

Best fitness value : 36.59040340 66528
```

28/10/24

Lab - 05

Grey Wolf Optimizer (GWO):

GWO is nature - inspired at optimization algorithm based on hunting behaviour and Social hierarchy of grey wolves. The algorithm mimics the roles of wolves in a peak: alpha, beta, delta & Omega.

```
import numpy as np

def obj-fun (x):
    return np.sum(x**2).

def gwo(no-wolves, no-itera, dim):
    wolves = np.random.uniform(-10, 10, (num
                no-wolves, dim))

alpha pos = np.zeros(dim)
beta-pos = np.zeros(dim)
delta-pos = np.zeros(dim)
alphascore = float('inf')
betascore = float('inf
deltascore = float('inf')

for in range (no-itera):
    for i in range (no-wolves):
        fitness = obj-fun(wolves [i])

        if fitness < alphascore:
            deltascore = betascore
            deltapos = beta pos
            beta-score = alphascore
```

```
        betapos = alphapos
        alphascore = fitness
        alphapos = wolves[i]

    elif fitness < beta score:
        delta score = beta score
        delta pos = beta pos
        beta score = beta pos
        beta pos = fitness

    elif fitness < delta score:
        delta score = fitness
        delta pos = wolves[i]

    a = 2-2 * (_ / no_itera)
    for i in range (no_wolves):
    r1, r2 = np. random. rand (2)
    A = 2*a * r1 - a
    C = 2 * r2
    D alpha = abs(C * alphapos - wolves[i])
    D beta = abs(C* betapos - wolves[i])
    D delta = abs(C * deltapos - wolves[i])

    wolves[i] = wolves[i] - A * D alpha - A*D_beta
                    A * D delta
    return alphapos, alphascore

best pos, best score = gwo (30, 100, 2)
print ("Best position:", best pos)
print ("Best score:", best_score)
```

O/P: Best position: [3.19199700[illegible]30 5.778007[illegible]]

Best score: 0.99391371132000192

Parallel cellular Algorithms and Programs

Parallel cellular algorithm are inspired by functioning of biological cells that operate in highly parallel and distributed manner. These algorithms leverage the principles of cellular automata and parallel computing to solve complex optimization problems efficiently.

```
import numpy as np

def fitness function (position):
    return np. sum (position **2)

num cells = 100
gride size = (10,10)
solution dim = 2
iteration = 100
neighbor radius = 1
search space bounds = [-5.0, 5.0]

def evaluate fitness (population):
    fitness = np. zeros ((grid size[0], grid size[0]))
    for i in range(grid size[0]):
        for j in range (grid size[1]):
            fitness [i, j] = fitness function (popu
                            - lation [i, j])
    return false fitness
```

```
def get neighbors(grid size, i, j, radius):
    neighbours = []
    for di in range(-radius, radius+1):
        for dj in range(-radius, radius+1):
            ni, nj = (i+di) % gridsize[0], (j+dj) %
                    gridsize[1]
            if (di != 0 or dj != 0):
                neighbors.append((ni, nj))
    return neighbors

def update population(population, fitness, gridsize,
                radius):
    newpopulation = np.copy(population)
    for i in range(grid size[0]):
        for j in range(grid size[1]):
            neighbors = get neighbors(grid size, i, j,
                                      radius)
            best neighbor = population[i, j]
            best fitness = fitness[i, j]
            for ni, nj, in neighbors:
                if fitness[ni, nj] < best fitness:
                    best neighbor = population[ni, nj]
                    best fitness = fitness[ni, nj]
            new population[i, j] = (population[i, j] +
                    bestneighbor) / 2.0
    return newpopulation

def parallel cellular():
    population = initialize population(gridsize,
            solution dim, search space bounds)
    best solution = None
    best fitness = float('inf')
```

```
for iteration in range(iteration):
    fitness = evaluate_fitness(population)
  if min_fitness < best_fitness:
    best_fitness = min_fitness
    best_indices = np.unravel_index(np.argmin
        (fitness), fitness.shape)
   best_solution = population[bestindices]

  population = update_population(population,
     fitness, gridsize, neighborhood_radius)

  print(f"Iteration {iteration+1}/{iterations},
   Best fitness: {bestfitness:.6f}")

 ~~print~~
print("\Best Solution found :", best_solution)
print("Best fitness value :", best_fitness)
```

Output:-

Best Solution found : [4.52424603e-06 -3.90935704e-06]
Best fitness value : 3.5751874622488412e-11

Optimization via Gene Expression Algorithm

GEA are inspired by biological process of gene expression in living organisms. This process involves the translation of generic information encoded in DNA into functional proteins

```
import numpy as np
import random

def fitness(solution):
    return np.sum(np.array(solution)**2)

population = 20
num gene = 10
mutation = 0.1
crossover = 0.7
generation = 25
Gene bound = [-5.0 , 5.0]

def initialize population(population size, num genes
                 bounds):
    population = []
    for _ in range(population size):
        individual = [random.uniform(bounds[0],
            bounds[1]) for _ in range(num genes)]
        population.append(individual)
    return population
```

```
def evaluate_population(population):
    fitness_scores = [fitness_func(individual)
        for individual in population]
    return fitness_scores

def tournament_se(population, fitness_scores,
            tournament_size=3):
    selected = []
    for _ in range(len(population)):
        participants = random.sample(list(enum
            -erate(fitness_scores)), tournment_sz)
        winner = min(participants, key=lambda
            x: x[1])
        selected.append(population[winner[0]])
    return selected

def crossover(parent1, parent2):
    if random.random() < crossover_rate:
        point = random.randint(1, len(parent1)-1)
        offspring1 = parent1[:point] + parent2[point:]
        offspring2 = parent2[:point] + parent1[point:]
        return offspring1, offspring2
    return parent1, parent2

def mutate(individual, bounds, mutation_rate):
    for i in range(len(individual)):
        if random.random() < mutation_rate
            individual[i] = random.uniform
                (bound[0], bound[1])
    return individual
```

```
def geneexpression():
    population = initialize_population(populationsize,
        numgenes, genebounds)

    bestsolution = None
    bestfitness = float('inf')

    for generation in range(generations):
        fitness_scores = evaluate_population(population)

        currbest = np.argmin(fitnessscores)
        if fitnessscores[currbest] < bestfitness:
            bestfitness = fitnessscores[currbest]
            bestsol = population[currbest]

        selected_population = tournament(population,
                        fitnessscores)
        nextgeneration = []
        for i in range(0, populationsize, 2):
            parent1 : selected_population[i]
            parent2 : selected_population[i+1]
            offspring1, offspring2 = crossover(parent1,
                        parent2)
            nextgeneration.append(mutate(offspring1,
                    genebounds, mutation_rate)
            nextgeneration.append(mutate(offspring2,
                    genebounds, mutationrate)

        population = nextgeneration

        print(f"Generation{generation+1}/{generations},
```

Best fitness : {bestfitness:.6f}")

```
print("\Best solution found :", bestsolution)
print("Best fitness value :", bestfitness)

if __name__ == "__main__":
    geneexpression()
```

Output:-

Best Solution found : [0.1956637320022, 1.22774355884, -0.8043843675228429, -0.7894773943883185, 0.15407105 -0.2711541235533, -0.1175746543334 3202, -0.8708068, -0.2980.51041 ..]

Best fitness value : 4.37919322230016 71