INDEX

NAME: Shlok Iyer STD.: _____ SEC.: 5E ROLL NO.: IBM22CS260 SUB.: BIS-LAB

S. No.	Date	Title	Page No.	Teacher's Sign / Remarks
1.	3/10/24	Summary of 7 Algo		8
2.	24/10/24	Lab 1: Generative Algorithm for optimization problem		10
3.	7/11/24	Lab 2: Particle Swarm Optimization		10
4.	14/11/24	Ant Colony Optimization		10
5.	21/11/24	Cuckoo Search		9
6.	28/11/24	Grey Wolf Optimization		10
7	17/12/24	Parallel Cellular Algo		10
7	17/12/24	Optimization Gene Exp		

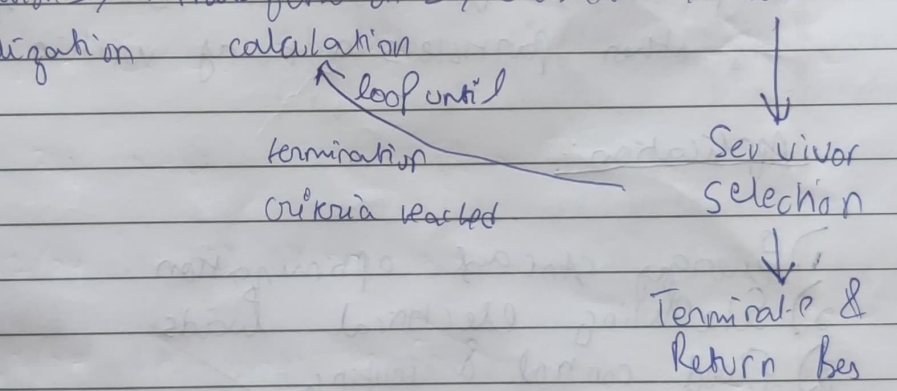
Algorithm - 1 -

Genetic Algorithm for optimization problems

GA (Genetic Algorithm) is a method for solving both constrained & uncountained optimization problem

GA is a natural selection process that mimics biological evolution

GA generates a population of points at ~~each~~ each iteration. The best point in the population approaches an optimal solution. Over successive generation, the population evolves to an optimal solutions.



Applications:

1) Task scheduling - GA algorithm is used to determine optimal solution based on certain constraint

2) Financial markets - finding an optimal combination of parameters that can affect trader or market rules.

3) Robotic - Generating optimal routes for a robot so that it uses least amount of resources to get to the derived position

Algo 2

Particle swarm optimization Algorithm

PSO is a population - based algorithm for search

It is a simulation to discover the pattern in which birds fly & their formations and grouping during flying activity

In PSO, every individual is considered to be a particle in some high-dimensional signal search space PSO is inspired by psychological tendencies of people to tend to copy from other people's success

A particle's behaviour in search space is influenced by other particle's activity within the swarm

Application:

1) Energy storage optimization
2) Scheduling electrical loads
3) Flood control & routing
4) Water quality monitoring
5) Diseases detection & classification

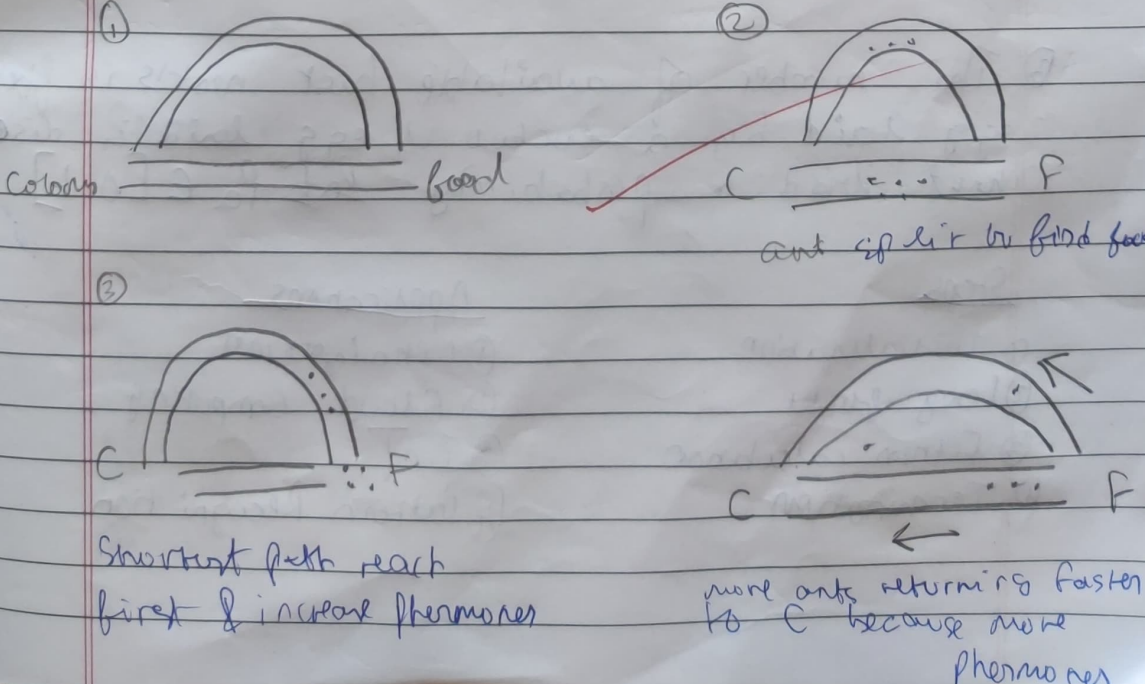
Algo 3

Ant Colony optimization for Traveling Salesman problem

Protocols used by ants to communicate and plan rotes. They do so by coordinating through Pheromon message (Chemical trails)

Procedure Ant Colony Optimization:

Initialize necessary parameters and pheromone trails
while
do:
Generate ant population
Calculation of fitness of each ant
find best solution through selection method
Update pheromone trail
end while
end procedure
To this, TSP can be applied



Applications:
1. Scheduling problems
2. Image processing
3. Data mining

Algorithm-4 - Cuckoo search (CS)

Introduction:
- CS is an evolutionary optimization algorithm & was inspired by species of bird - cuckoo & their aggressive reproduction strategy

- They lay their eggs in nests of other host birds if host bird recognizes eggs as not their own, they will throw it away or build a new one

- Rules
1. Each cuckoo lays 1 egg at a time and randomly crosses nest
2. Best nest with highest quality eggs will carry over to next generation
3. The number of available host nests is fixed & egg laid by a cuckoo & egg laid is discovered host bird a probability $P_a \in [0, 1]$

Steps
1. Initialization
2. Levy flight
3. Fitness calculation
4. Termination

Applications
1. Optimization
2. Cloud computing
3. IoT
4. Pattern Recognition

Algorithm 5 : Grey wolf optimizer (GWO)

Introduction :

- Nature inspired metaheuristic based on the behaviour of pack of wolves to find optimal solution
- Hunt in large packs & rely on cooperation among individual wolves
  (1) Social hierarchy (2) hunting mechanism
- Complex social hierarchy includes delta, gamma, beta & alpha values representing best solution candidates at each iteration
- To find best answer, the hunting behaviour of grey wolves are picked. The beta & gamma wolves follow the alpha wolve's lead when they with their meal, delta wolf attacks first.

Workflow :

Initialization → Fitness Evaluation → Encircling Prey → Attack → Update position → Check fitness → Stopping Criterion → Output

Application :

(1) Data Mining
(2) Image & signal processing
(3) Energy management
(4) Machine Learning

Algorithm 6: Parallel Cellular Algorithms:

Introduction:

- Parallel cellular algorithm are computation model based on the idea of dividing a problem into smaller units or cells that operate simultaneously & often independently
- Inspired by cellular automata, where the cells interact with their neighbours based local rules
- Parallelization in cellular algorithm is achived by distributing these cells across multiple processing units allowing for concurrent execution of many operations

Two main concepts - ① Cellular automata
② Parallel processing

Applications:

① Image Processing
② Fluid dynamics
③ Biological System
④ Physics simulation

Algorithm 7: Optimization via Gene Expression Algorithms

Introduction:

- GEA is inspired by biological processes, particularly,
- GEA's optimize problems by mimicking the way genes in DNA express the metric to develop specific traits in organizations.

Optimization Process:

Population initialisation → Expression → Fitness Evaluation

↓

Next Generation ← Crossover ← Selection

Mutation

Application:

1. Machine learning
2. Data mining
3. Resource allocations
4. Engineering optimization

Algorithm for Genetic Algorithm:

function objective(x):
    return x[0]**2 + 2*x[0] + 1 **2

function initialize_population(bounds, n)
    return random uniform from bound[0], bound[1]

function evaluate_fitness(population)
    return objective(index) for index in population

function roulette_wheel_selection(population, scores)
    total_fitness ← sum of (scores)
    probabilities ← 1 - score/total_fitness for each score
    select_ind ← random choice p/sum(p)
    return population[select_ind]

function crossover(p1, p2, α)
    if
    if random <
    return
    return offspring = α * p1[0] + (1-α)p2[0]

function mutation(individual, bounds, rate)
    if random() < rate:
        return random(bound[0], bound[1])
    return individual

Function genetic_algorithm(bounds, n_iter, n_pop, r_rate)

population ← initialize_population(bounds, n_pop)
best, best_eval ← pop[0], objective(pop[0])

for gen in n_iter:
    scores ← evaluate_fitness(population)

    for each n_pop:
        if score[index] < best_eval:
            best, best_eval = population[i], scores[i]
            print new best func[index]

    create children
    for each n_pop:
        p1 ← roulette_wheel_selection(population, scores)
        p2 ← roulette_wheel_selection(population, scores)

        offspring ← crossover(p1, p2)

        offspring ← mutation(offspring, bound, rate)

        append offspring to children

    population ← children

Applications

1. Optimization Problems: TSP, Scheduling Problem
2. ML: Hyperparameter tuning, feature selection
3. Engineering design: Structural optimization

classmate
Date 7-11-24
Page

Algorithm for Particle swarm -

Function rastrigin_function(x)
    n ← length x)
    A ← constant
    return A * n + sum (x² − A cos(2 * π * x))

Particle class:

    class constructor to initialize particle pos, vel
        (self, dimension
        min, max)

        self.position ← position random min, ma
        di

        self.velocity ← random velocity (-1, 1, dim
        self.best_pos ← copy of position

    ~~def~~ evaluate
    Function evaluate (self, fitness rastrigin_functio

        self.fitness ← fitness_func (self.pos)

        if self.fitness < self.best_fitness
            exchange the fitnesses

Function pso(fitness_func, dim, population_size, max_ite
        min_x, max_x, w=0.
        c1=1.5, c2=1.5)

    for _ in range(pop_size):
        swarm = [Particle(dim, min_x, max_x)

for gen in range (maximum_iteration):

avg_best_particle fitness ← 0

for i in range (population size):

swarm[i].velocity ← compute velocity using the constants

swarm[i].pos ← updated position

swarm[i].evaluate (fitness_fine)

if swarm[i].fitness < best_fitness_swarm:

update global best position

avg_best_particle ← swarm[i].best_fitness

avg_best_particle /= N

return best_pos swarm, best_fitness_swarm,

avg_particle best_fitness,

num_iter

Applications:

- Energy storage Optimization
- Scheduling Electrical loads.
- Flood control & routing

Ant Colony Optimization

Algorithm:

function Input_City_coordinates():

Input: number of cities, x and y coordinates
output: 2D array of city coordinates

function calculate_Euclidian_distance(point1, point2):

Input: point1, point2
Output: distance bw point1 and point 2

distance ← sq root of (point1 - point2)^2 for each coordi...

function construct_solution(n_points, pheromone, points, alpha, beta):

visited ← list of false values of length n_points
current_point ← random starting city index
Set visited[current_point] ← True
path ← list containing current_point
path_length ← 0

while there are unvisited cities:
unvisited ← indices of cities not visited
probabilities ← array of zeroes

For each unvisited city:
pheromone_value ← pheromone[current_point, unvisited_city] squared to alpha
distance_value ← distance(points[current_point], points[unvisited_point]) ** beta

probabilities [i] ← pheromone val / distance val

probabilities ← sum of probabilities / probabilities

next_point ← random (unvisited, probabilities)

append next_point

path_length ← path_length + distance

set visited of next point to true and

update current point ← next point

return path, path_length

def function update_pheromone (pheromone, paths, path_lengths, evaporation_rate, Q):

Evaporate all pheromones by setting pheromone ← pheromone * evaporation_rate

for each path in paths:

for each pair of consecutive cities in the path:

increment pheromone

increment pheromone for the last to first city in the path to close the loop

function ant_colony_optimizing func (points, n_ants, n_iterations, alpha, beta, evaporation_rate, Q):

n_points ← length of points
initialize_pheromone ← matrix of ones with shape (n_points, n_points)

best_path ← None
best_path_length ← infinite

For each iteration in range (n_iterations):

paths ← empty list
path_lengths ← empty list

for each ant in range (n_ants):
Call construct_solution() to get path and path_length
Append path to paths and path_length to path_lengths
if path_length < best_path_length:
best_path ← path
best_path_length ← path_length

Call update_pheromones() with pheromone, evaporation_rate

Print current iteration and best_path_length

Applications:

- Logistics & Supply chain
- Vehicle Routing Problem
- Telecommunications & Network scanning

classmate
Date 21-11-24
Page

Algo - 4 - Cuckoo Search

Algorithm

```
objective_function(x):
    return x[0]**2

levy_flight_function(num, β=1.5):

    sigma_u ← (gamma(1+β) * sin(π * β/2)) / (gamma((1+β)/2) * β * (2^((β-1)/2)))^(1/β)

    u ← random.normal(0, sigma_u, num)
    v ← random.normal(0, 1, num)

    return u / |v|^(1/β)

cuckoo_search(iter, num_nests, pa=0.25):

    num_dim ← 1
    num_nests ← random(num_nests, num_dim) * 10 - 5

    fitness ← objective_function(num_nests)

    best_nest ← nests[minimum(fitness)]
    best_fitness ← minimum[fitness]

    for _ in range(iter):
        for i in range(num_nests):
            new_nest ← nests[i] + levy_flight(num_dim)

            new_fitness ← objective_function_1d(new_nest)

            if new_fitness < fitness[i]:
                nests[i] ← new_nest
                fitness[i] ← new_fitness
```

```
worst_nests ← sort(fitness)[-int(pa * num_n
for j in worst_nests:
    nests[j] ← random(num_dim) * 10 - 5
    fitness[j] ← objective_function.id(nests[
current_best_idx ← minimum(fitness)
current_best_fitness ← fitness[current_best_idx
return best_nest, best_fitness
```

Lab 8
21/11/24

Applications:

- Wifi Network
- Cloud Computing
- IoT

classmate
Date 28-11-2024
Page

GREY WOLF OPTIMIZATION

```
function Initialize_wolves (search_space, num_wolves):
    dimensions ← length (search_space)
    wolves ← zeroes((num_wolves, dimensions))
    for i in range(num_wolves):
        wolves[i] = random(search_space[:, 0],
                           search_space[:, 1])
    return wolves

function fitness_function(x):
    return sum(x**2)

function gwo_algorithm (search_space, num_wolves,
                        max_iter):
    dimension ← length(search_space)
    wolves ← initialize_wolves(search_space, num_wolves)

    alpha_wolf ← np.zeroes(dimensions)
    beta_wolf ← np.zeroes(dimensions)
    gamma_wolf ← zeroes(dimensions)

    alpha_fitness ← float('infinity')
    alpha_fitness ← float('infinity')
    gamma_fitness ← float('infinity')

    beta_fitness ← float('infinity')

    for iteration in range(max_iterations):

        a ← 2-(iteration/max_iteration)*2
```

print(f"Iteration {t+1}/{max_iterations}")

for i in range(num_wolves):

fitness = fitness_function(wolves[i])

if fitness < alpha_fitness:
gamma_wolf = beta_wolf.copy()
gamma_fitness = beta_fitness
beta_wolf = alpha_wolf.copy()
beta_fitness = alpha_fitness
alpha_wolf = wolves[i].copy()
alpha_fitness = fitness

elif fitness < beta_fitness:
gamma_wolf = beta_wolf.copy()
gamma_fitness = beta_fitness
beta_wolf = wolves[i].copy()
beta_fitness = fitness

print("Best {alpha_fitness}")

if alpha_fitness < best_fitness:
best_fitness = alpha_fitness

for i in range(num_wolves):
for j in range(dimensions):
r1 = np.random.random()
r2 = np.random.random()

A1 = 2 * a * r1 - a
C1 = 2 * r2
D_alpha = np.abs(C1 * alpha_wolf[j] - wolves[i,j])

X1 ← alpha_wolf[j] - A1 * D_alpha

r1 ← np.random.random()
r2 ← np.random.random()

A2 ← 2 * a * r1 - a
C2 ← 2 * r2

D_beta ← NP.abs(C2 * beta_wolf[j] - wolves[i,j])
X2 ← beta_wolf[j] - A2 * D_beta

r1 ← NP.random.random()
r2 ← NP.random.random()

A3 ← 2 * a * r1 - a
C3 ← 2 * r2

D_gamma ← NP.abs(C3 * gamma_wolf[j] - wolves[i,j])

X3 ← gamma_wolf[j] - A3 * D_gamma

wolves[i,j] ← (X1 + X2 + X3) / 3

wolves[i,j] ← np.clip(wolves[i,j], search_space[j,0], search_space[j,1])

print(f"alpha_wolves")
print(f"best fitness")

seach_space ← AP.array([[-5,5],[-5,5]])

num_wolves ← 10
max_iterations ← 100

optimal_solution ← gwo_algorithm(search_space, num_wolves, max_iterations)

Print(optimal_solution)

28/11/24

Applications:

- Signal Processing
- Energy Design Optimization
- Control Systems
- Vehicle Routing

PARALLEL CELLULAR ALGORITHMS

Step 1: function fitness_func

Define fitness function

$$f(x) = \sum_{i,j} x_i^2$$

Step 2: Initialize Parameters

Set grid_size = (rows, cols)
Set dimensionality of the search space
Define search space bounds : minx and maxx
Set number of iterations : max_iterations

Step 3: Initialize Population

Create a 3D grid population of size (rows, cols, dim) to store positions of cells

For each (i,j) in grid:
    rand initialize [minx, maxx]

Step 4: Evaluate Fitness

Create a fitness_grid of size (rows, cols)

For each cell (i,j) in grid:
- compute fitness value using fitness func f(position)

Step 5: Update States

Define a function to get the neighbours of a cell (i,j)

For each (i,j):
identify best neighbour
Update cell position to move towards pos
of best neighbour
-Add a small random perturbation to avoid
Stagnation
• Clip the position to ensure it stays within bound
Cmin, max

Step 6: Iterate

1. Repeat the following for max_iterations:
. Evaluate fitness of current position
For each (i,j):
Update cell position based on neighbour

Replace old population with new updated Popula
print best fitness value at each iteration

Step 7: Output best Solution

1. Find cell with minimum fitness value in grid
2. Output position and corresponding fitness value
of this cell

Application

- Image Processing
- Machine Learning
- Travelling Salesman
- Fluid dynamics
- Heat diffusion Simulation

OPTIMIZATION VIA GENE EXPRESSION ALGORITHM

Algorithm:

Step 1: Define the fitness func
$f(x) = x^2$

Step 2: Initialize Parameters
population_size, num_genes, lower_bound, upper bound, max generations, mutation_rate, crossover rate

Step 3: Initialize Population

- Generate a population of random genetic sequences
  - Each individual has num_genes genes
  - Gene value distributed by [lower bound, upper bound

Step 4: Evaluate Fitness
For each individual in population:
Calculate its fitness using f(x)

Store all the fitness values in an array

Step 5: Perform Selection
1. Compute the selection probability for each individual based on fitness:
   - Normalize fitness values to probabilities

2. Roulette wheel selection to choose a subset of individuals for reproduction:

population_size // 2 individuals selected

Step 6. Perform Crossover

- For each pair of selected parents:
  - With crossover_rate probability:
    - Choose random crossover point in gene seq
    - Swap genes after the crossover point to prod two offsp
  - Otherwise:
    - Retain parents without changes.

Add the offspring to the new population

Step 7. Perform Mutation

with probability mutation_rate:
select random gene and assign it in
a new random_value in [lower_b
upper_b

Ensure mutated values remain within bounds

Step 8. Track Best Solution (lowest fitness) and store it

Step 9 Repeat for All generations (Repeat step 4

Step 10 Output the Results

Applications:

- Engineering Optimizations
- Data Analysis
- Machine Learning
- Bioinformatics
- DNA sequence alignment