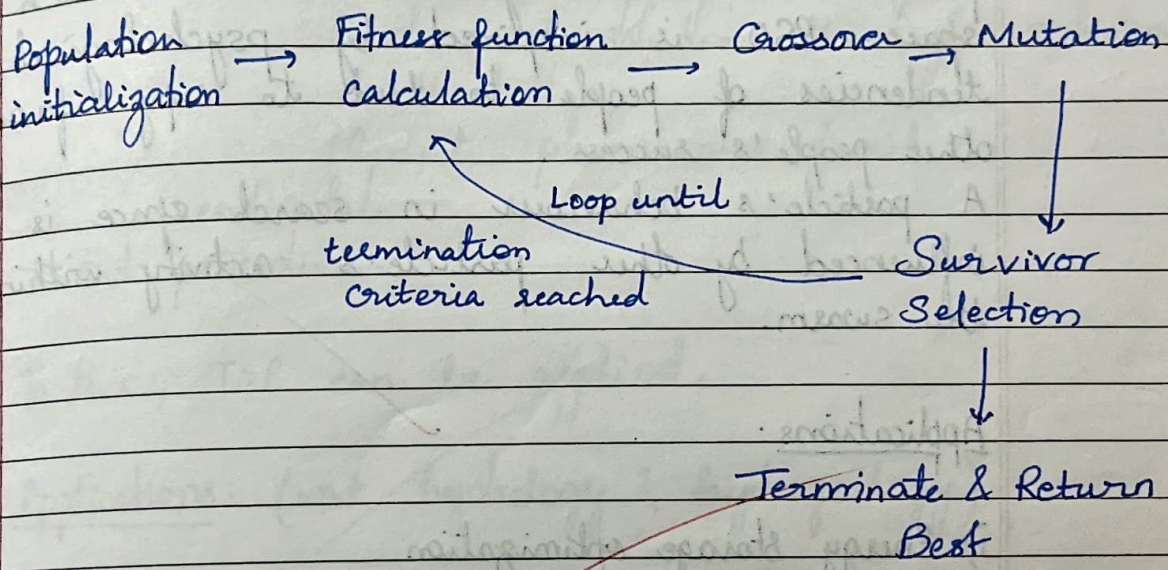
(1) Genetic Algorithm for Optimization Problems :

GA (Genetic Algorithm) is a method for solving both constrained & unconstrained optimization problems.

GA is a natural selection process that mimics biological evolution.

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



Applications :

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

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

3) Robotics : generating optimal routes for a robot so that it uses least amount of resources to get to the desired position.

(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 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.

Applications:

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

③ Ant Colony optimization for Travelling Salesman Problem:

Protocols used by ants to communicate and plan routes. They do so by coordinating through pheromon messages (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.

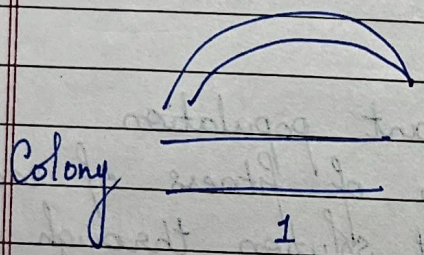
Applications: (ant hydrology & hydrogeology)

1) Irrigation water allocation
2) Urban drainage network design
3) Reservoir optimization
4) Water distribution system.

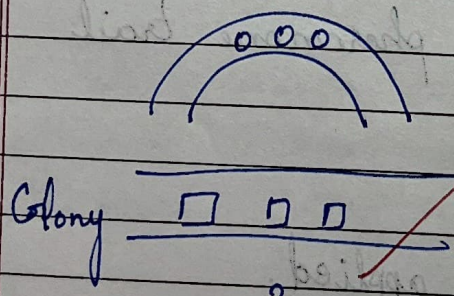
Stages:

(1) ants are divided into 2 groups for 2 groups for 2 paths with a probability of 0.5. (4 ants on the longer path & 4 on the shorter path)

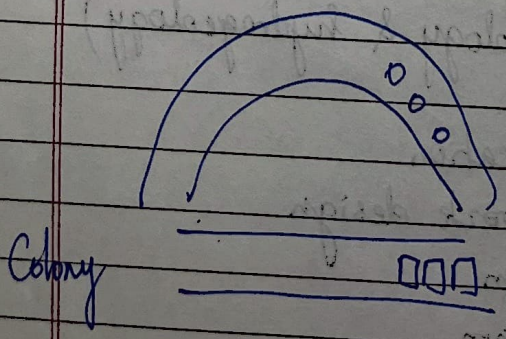
(2) Ants following the shorter path will reach to the food first, and then the pheromone concentration will be high on the shorter path causing other ants to follow it.



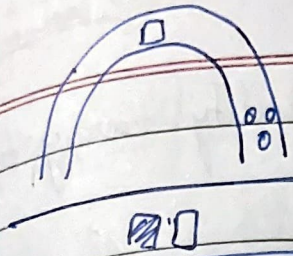
no pheromone content



(0 selection probability) 0.5 probability on shorter & longer path



the ants through the shorter path reach earlier (due to a pheromone trail along the shorter path, selection path is higher)



evaporation of pheromones along longu path reduces, gradually decreasing the selection probability of this path.

(1) Cuckoo Search (CS)

The main idea of the CS algorithm is inspired by the brood parasitism of cuckoo species by laying their eggs in the nest of host birds.
If the host bird discovers the eggs are not their own, it will throw the alien eggs away or just abandon the nest.

CS Algorithm representations:

a) Each egg in the nest is a representative of a new solution and a cuckoo egg represents a new solution.

b) The aim is to use new & better solutions (cuckoo) in place of existing eggs (present solution)

3 rules:

a) Each cuckoo lays an egg a time, and dumps its egg in a random nest.

b) The best nests with high quality carry over to the next generation

c) No. of available host nests are fixed & the egg laid by the cuckoo is discovered by the host with a probability range $Pa \in (0, 1)$.

A technique of Lévy flights is used rather than simple random walk.

Applications:

1) Cloud Computing: task scheduling problem

2) Solving 2 classification problems for training feed forward neural networks.

3) CS is used as an optimization techniques for job scheduling on computational grids.

(5) Grey Wolf Optimizer (GWO)

The GWO algorithm mimics the leadership hierarchy and hunting mechanism of grey wolves in nature. Four types of grey wolves such as alpha, beta, delta & omega are employed for simulating leadership hierarchy. Steps of hunting, searching for prey, encircling prey, and attacking prey are implemented.

Social hierarchy of solutions:

1) The fittest solution as an Alpha wolf (α)
2) Second best solution as Beta wolf (β)
3) Third best solution as Delta wolf (δ)
4) Remaining solutions as Omega wolves (ω)

Algorithm:

1) Random initialization of population of grey wolves (X_i, $i = 1, 2, \ldots n$)

2) Initializing coefficient vectors

3) Calculation of fitness of each member
   X_α = best fitness member
   X_β = member w/ second best fitness
   X_γ = member w/ third best fitness

4) through a for loop till max iterations, update all values.

5) Return X_α

Applications

1. GWO effectively solves optimization problems, particularly in various IoT applications
2. Used in applications of path planning
3. GWO can be used in solving combinatorial problems.

6 Parallel Cellular Algorithms (PCA)

- Parallel cellular algorithms are computational models based on the idea of dividing a problem into smaller units or cells that operate simultaneously and open independently.
- Inspired by cellular automata, where the cells interact with their ~~regin~~ neighbours based on local rules.
- Parallization in cellular algorithms is achieved by distributing these cells across multiple processing units, allowing for concurrent execution of many operations.
- 2 main concepts - (1) Cellular Automata
  (2) Parallel processing

classmate
Date
Page

Applications :

1. Image processing
2. Fluid dynamics
3. Biological system
4. Physics simulation

(7) Optimization via Gene Expression Algorithms

- GEA (Gene Expression Algorithms) is inspired by biological processes, particularly gene expression in living experiences and belong to family of evolutionary algorithms

- GEA's optimize problems by mimicking the way genes in DNA express themselves to develop specific traits in organisms.

- Optimization ~~Problems~~ Process:

Population Initialization → Expression → Fitness Evaluation → selection → crossover & mutation → Next generation

- Applications

① Machine Learning

② Data Mining

③ Resource allocations

④ Engineering Optimization

24/10/24 Genetic Algorithm for Optimization Problems

Algorithm:

① FUNCTION objective(x): # the function calculates the fitness of an individual solution.

return x**2

② FUNCTION initialize_population(bounds, n_pop):

The function creates an initial population of potential solutions

population = [] #empty list for population.
#loop to create 'n' individuals' population.

for i from 0 to n_pop-1:
generate a random individual within bounds
Add the individual to the population

for eg: [-10, 10] can be a search range

③ FUNCTION evaluate_fitness(population):
scores = [] # list to score fitness scores
# looping through all individuals to fit calculate the fitness using the objective function

④ FUNCTION roulette_wheel_selection (population, scores):

the aim of the function is to select an individual based on fitness score.

```
total_fitness = sum (scores)   # total fitness of pop
probabilities = [ ]            # selection probability

for score in scores:
    probabilities ← 1 - (score / total fitness)

selection ← random_choice (probabilities)
```

⑤ FUNCTION crossover (p1, p2, α=0.5):

the function performs crossover between 2 parents to create offspring

```
offspring ← α * p1[0] + (1-α) * p2
                          # linear combination
```

⑥ FUNCTION mutation (individual, bounds, r_mut)

```
#checking if mutation occurs
    IF RANDOM_NUM (0,1) < r_mut (mutation rate)
        return a new individual

    else
        return a original individual
```

#main function to run the genetic algorithm.

⑦ FUNCTION genetic_algorithms (bounds, n_iter, n_pop, r_mut):

```
pop = initial_pop (bounds, n_pop)
best = pop[0]
best_eval = objective (best)   # evaluate fitness.
for gen FROM 0 to niter-1:
#comparing individuals.

    for i from 0 to n_pop-1:
        if scores[i] < best_eval
            best = pop[i]
            best_eval = scores[i]
        #print generation & best population

#new offspring

    children = [ ]
    for i from 0 to n_pop-1
        p1 = roulette_wheel_selection (pop, score)
        p2 = ——"—— (pop, scores)
        offspring = crossover (p1, p2)
        offspring = mutation (offspring, bounds, r_mut)
#append children
        offspring to

    pop = children   # replace old pop with new offspring
```

n_iter #generations no.
n_pop # population count
r_mut = 0.1 (10%)
bounds = [-10.0, 10.0]

example.

24/10/24

7/11/24

Lab 3
Global Swarm Optimization

Pseudo code algorithm

```
FUNCTION Rastrigin (x, A=10)
    n = LENGTH(x)
    RETURN A*n + SUM (x[i]^2 - A*COS(2*PI*x[i])
    (FOR i=0 to n-1)

CLASS Particle
    FUNCTION INIT (dim, bounds)
        self.position = RANDOM_UNIFORM (bounds[0],
                        bounds[1], dim)
        self.velocity = RANDOM_UNIFORM (-1, 1, dim)
        self.best_position = COPY (self.position)
        self.best_value = INFINITY

    FUNCTION UPDATE_VELOCITY (global_best_
    position, w, c1, c2)
        inertia = w * self.velocity
        cognitive = c1 * RANDOM() * (self.
        best_position - self.position)
        Social = c2 * RANDOM() * (global_best_
                 position - self.position)
        Self.velocity = inertia + cognitive + social
```

```
FUNCTION UPDATE_POSITION (bounds)
    self.position += self.velocity

    self.velocity = CLIP (self.position,
    bounds[0], bounds[1])

FUNCTION EVALUATE (func)

    fitness = func (self.position)

    IF fitness < self.best_value THEN

        self.best_position = COPY (self.position)
        self.best_value = fitness

FUNCTION PSO (func, dim, bounds,
num_particle = 30, max_iter = 100, W = 0.5,
c1 = 1.5, c2 = 1.5)

    INITIALIZE particles = ARRAY OF
    Particle [num_particle]

    global_best_position = NONE
    global_best_value = INFINITY

    FOR iteration FROM 0 To max_iter-1 DO
        FOR each particle IN particles DO
            particle.EVALUATE (func)
            IF particle.best_value < global_
            best_value THEN
                global_best_position
                = COPY (particle.best_position)
                global_best_value =
                    particle.best_value.
```

FOR EACH particle IN particles DO
particle.UPDATE_VELOCITY(global_best_position, w, c1, c2)

particle.UPDATE_POSITION(bounds)

PRINT "Iteration", iteration+1, "Global best value:", global_best_value

RETURN global_best_position, global_best_value

MAIN

dim = 2
bounds = (-5.12, 5.12) // bounds for search space.

best_position, best_value = PSO(Rastrigin, dim, bounds)

best_position // print
best_value // print

14/11/24

Lab 4

Ant Colony Optimization for Travelling Salesman Problem.

Function Input_City_Coordinates():
    Prompt user to enter the number of cities
    For each city, prompt user for coordinates (x,y)
    Return the coordinates as a NumPy array

Function Distance(Point 1, Point 2):
    Calculate and return the Euclidean distance between Point1 and Point 2

Function Construct_solution():
    Initialize visited cities list and path
    Choose a random starting city

    While all cities are not visited:
        Calculate probabilities for each unvisited city
        Choose the next city based on probabilities
        Add the city to the path & mark as visited
        Update the path length

    Return the constructed path and its length

Function Update_Pheromones():

Evaporate pheromones by multiplying matrix by evaporation rate
for each ant's path:
Update pheromone levels along the path (both forward & backward edges)

Return updated pheromone matrix

Function Ant_Colony_Optimization():

Initialize pheromone matrix, best path, and best path length
For each iteration:
For each ant:
Construct a solution (path)
Update the best solution if necessary

~~Update~~Update pheromone levels based on all ants' paths
Optionally, log progress

Return the best path and best path length

Function Plot_Best_Path():

Plot the cities on a 2D graph
Draw the best path found.

Main():

cities = Input_City_Coordinates()

best_path, best_path_length = Ant_Colony_Optimization(cities)

Print best_path and best_path_length

Plot_Best_Path(cities, best_path)

n_ants = 10

n_iterations = 100

alpha = 1

beta = 1

evaporation_rate = 0.5

Q = 1

880

14/11/24

21/11/24

Lab 5
Cuckoo Search Algorithm

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

Levy_flight(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
    nests ← random.rand(num_nests, num_dim)
    fitness ← objective_function(nests)
    best_nest ← nests[np.argmin(fitness)]
    best_fitness ← np.min(fitness)
    for _ in range(iter):
        for i ← 0 to num_nests:
            new_nest ← nests[i] + levy_flight(num_dim)
            new_fitness ← objective_function(new_nest)
            if new_fitness < fitness[i]
                nests[i] ← new_nest
                fitness[i]
```

worst_nests ← argsort(fitness)[-int(pa* num_nests):]

for j in worst_nests:
    nests[j] ← random(num_dim) * 10^{-5}
    fitness[j] ← objective_function(nests[j])

current_best ← np.argmin(fitness)
current_best_fitness ← fitness[current_best]

if current_best_fitness < best_fitness:
    best_fitness ← current_best_fitness
    best_nest ← nests[current_best]

return best_nest, best_fitness

Inputs:
num_iterations = 1000
num_nests = 25
pa = 0.25
β = 1.5

17/12/24

Lab 6

Grey Wolf Optimization

```
initialize_wolves (search_space, num_wolves):
    dim ← len (search_space)
    wolves ← zeros ((num_wolves, dim))

    for i ← 0 to num_wolves:
        wolves[i] ← random_uniform (
                        search_space[:, 0],
                        search_space[:, 1])

    return wolves

fitness_function (x):
    return np.sum (x ** 2)

GWO_algorithm (search_space, num_wolves,
               max_iterations):

    dim ← len (search_space)
    wolves ← initialize_wolves (search_space,
                                num_wolves)

    alpha ← zeros (dim)
    beta ← zeros (dim)
    gamma ← zeros (dim)

    alpha_fit = beta_fit = gamma_fit ← float ('inf')

    best_fit ← float ('inf')
```

```
for iter in range(max):

    a ← 2 - (iteration/max) * 2
    for i ← 0 to num-wolves:
        fitness ← fitness-function(wolves[i])
        if fitness < alpha-fitness
            copy beta ~~cutoff~~ wolf & fitness to gamma
            copy alpha ~~cutoff~~ wolf & fitness to beta
            alpha-wolf ← wolves[i].copy()
            alpha-fitness ← fitness

        elif fitness < beta-fitness:
            copy beta wolf & fitness to gamma
            beta-wolf ← wolves[i].copy()
            beta-fitness ← fitness

        elif fitness < gamma-fitness:
            gamma-wolf ← wolves[i].copy()
            gamma-fitness ← fitness

if alpha-fitness < best-fitness:
    best-fitness ← alpha-fitness

for i ← 0 to num-wolves:
    for j ← 0 to dim:
        r1 & r2 random floats
        A1 ← 2*a*r1 - a
        C1 ← 2*r2
        D-alpha ← abs(C1*alpha-wolf[j]
                      -wolves[i,j])

        X1 ← alpha-wolf[j] - A1* D_alpha
```

New r1 & r2 random floats
A2 ← 2*a*r1 - a
C2 ← 2*r2

D-beta ← abs (c2* beta - wolf [j]
- wolves [i, j])

X2 ← beta - wolf [j] - A2* D- beta

New r1 & r2 random floats
A3 ← 2*a*r1 - a
C3 ← 2*r2

D-gamma ← 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] ← disp (wolves [i,j],
search-space [j, 0], search-space [j, 1])

Inputs:

search-space = array ([-5,5], [-5,5])
num-wolves = 10
max-iter = 100

Lab 7
Parallel Cellular Algorithms

```
Custom-function (x):
    return sum (x ** 2)

Initialize-population (grid-side, dim, lower, upper)
    population ← random (lower, upper, (grid-size, dim))

    return population

Evaluate-fitness (population, fitness-function)
    fitness ← zeros (population.shape[-1])
    for i in range (population.shape[0])
        for j in range (population.shape[1])

            fitness[i][j] ← custom-function (population[i,j])

    return fitness.

update-states (pop, fit, radius, lower, upper):
    grid-size, dim ← pop shape

    new-pop ← pop.copy()

    for i in 0 to grid-size:
        for j in 0 to grid-size:
            neighbours ← get-neighbors (i, j, grid-size, radius)
            best-neighbour ← None
            best-fitness ← float (inf)
```

```
if best_neighbour:
    ri, rj ← best_neighbor

    new_population[i,j] ← pop[ri, rj]
        + random(-0.5, 0.5, dim)

    new_population[i,j] ← clip(new_
    population[i,j], lower, upper)

return new_population

get_neighbors(i, j, grid_size, radius)
    neighbors ← []
    for di in range(-radius, radius+1):
        ni, nj ← (i+di) % grid_size,
                 (j+dj) % grid_size

        if (di != 0 or dj != 0)
            neighbors.append((ni, nj))

    return neighbors

Parallel_cellular_algorithm(grid_size, dim,
lower, upper, max_iters, radius, fitness_
                                    func)
    population ← initialize_population(grid_size,
    dim, lower, upper)

    fitness ← evaluate_fitness(population,
    fitness_function)

    min_fitness ← fitness.min()
    if min_fitness < best_fitness
        best_fitness ← min_fitness
```

```
best_sol ← None
best_fit ← ∞
for iter in max_iters
        pop ← update_states (population, fitness,
              radius, lower, upper)

        fitness ← evaluate_fitness (pop,
                  fitness_function)

        min_fitness ← fitness.min()
        if min_fitness < best_fitness :
                best_fitness ← min_fitness
                best_sol ← population (np.
                  unravel_index (fitness.min(),
                    fitness.shape)]

        print (Iteration, fitness)

return best_sol, best_fit
```

Inputs :

```
grid_size = 10
dim = 1
lower = -5
upper = 5
max_iters = 100
radius = 1
```

Lab 8

Optimization via Gene Expression

```
Function(x):
    return sum(x**2)

Initialize-Population(pop_size, num_genes, lower, upper)
    pop ← random(lower, upper, (pop_size, num_genes))
    return pop

Evaluate-fitness(pop, function):
    fitness ← zeros(pop.shape[0])
    for i in range pop.shape[0]:
        fitness[i] ← function(pop[i])
    return fitness

selection(pop, fitness, num_selected):
    prob ← fitness / fitness.sum()
    indices ← random.choice(range(len(pop)), size=num_selected, p=prob)
    selected_pop ← pop[indices]
    return selected_pop

Crossover(selected_pop, crossover):
    new_pop ← []
    num_individuals ← len(selected_pop)
```

```
for i in range 0 to num_individuals-1, step2:
    p1, p2 ← selected_pop[i], selected_pop[i+1]
    if len(p1) > 1 and random < crossover:
        crossover_point ← random(1, len(p1)-1)
        child1 ← concatenate(p1[:crossover_point], p2[crossover_point:])
        child2 ← concatenate(p2[:crossover_point], p1[crossover_point:])
        new_pop.extend([p1, p2])
    if num_individuals % 2 == 1:
        new_pop.append(selected_pop[-1])
return new_pop

Mutation(pop, rate, lower, upper):
    for i in range(pop.shape[0]):
        if random < rate:
            gene ← random 1:0, pop.shape[1]-1)
            pop[i, gene] ← random.uniform
                           (lower, upper)
return pop

Gene_expression(individual, function):
    return function(individual)
```

```
Gene_Expression_Algorithm(pop_size, num_genes, lower, upper, max_gen, mutation_rate, crossover_rate, function):

    pop ← initialize_population(pop_size, num_genes, lower, upper)

    best_sol ← NIL
    best_fit ← ∞

    for gen in 0 to max_gen:

        fitness ← evaluate_fitness(pop, function)
        min_fitness ← fitness.min()

        if min_fitness < best_fit:
            best_fit ← min_fitness
            best_sol ← pop[min(fitness)]

        selected_pop ← selection(pop, fitness, pop_size//2)

        offspring_pop ← crossover(selected_pop, crossover_rate)

        pop ← mutation(offspring_pop, mutation_rate, lower, upper)

        print(Generation, fitness)

    return best_sol, best_fit
```

Inputs :

pop_size = 50
num_genes = 1
lower = -5
upper = 5
max_gen = 100
mutation_rate = 0.1
crossover_rate = 0.7