INDEX

Name Shashank Patel C.J.

Standard 8 Section E Roll No. 255

Subject B.I.S

SL No.	Date	Title	Page No.	Teacher Sign / Remarks
1	03/10/24	Exploring different genetic algorithms		8
2	24/10/24	Genetic Algorithm.		9
3	07/11/24	Particle swarm optimization.		10
4	14/11/24	Ant Colony optimization.		10
5	21/11/24	Cuckoo Search (CS).		10
6	28/11/24	Grey Wolf Optimizer (GWO).		10
7	16/12/24	Parallel cellular Algorithm.		10
8	16/12/24	Gene expression Algorithms.		10

29/10/24

Genetic Algorithm

Algorithm: Genetic Algorithm to Maximize $f(x) = x^2$.

Input:

- Population Size (pop_size).
- Number of generations (generations).
- Mutation rate (mutation_rate).
- Crossover rate (crossover_rate).
- Range of x (x_range_low, x_range_high).

Output:

- Best Solution x that maximizes $f(x) = x^2$
- Best fitness $f(x)$.

Pseudocode:

1. Initialize population of size 'pop_size' with random values in range [x_range_low, x_range_high]
2. For generation in range(generations):
   a. Evaluate fitness of each individual:
      For each individual x_i in population:
      fitness(x_i) = x_i^2.
   b. Select two parents based on fitness value (roulette-wheel Selection):
      total_fitness = Sum of all fitness values.
      For each individual x_i in population:
      probability(x_i) = fitness(x_i) / total_fitness
      Select two parents (p1, p2) based on probability.
   c. Perform crossover with probability 'crossover_rate':
      if random_number < crossover_rate:

alpha = random value between 0 and 1
offspring 1 = alpha * p1 + (1 - alpha) * p2
offspring 2 = alpha * p2 + (1 - alpha) * p1

else:
offspring 1 = p1
offspring 2 = p2

d. Apply mutation with probability 'mutation_rate':
For each offspring:
if random_number < mutation_rate:
offspring = random value in range [x_range_low, x_range_high]:

e. Update population with new offspring.

3. Terminate after maximum generations or convergence.

4. Output the best individual and fitness value.

e. Update population with new offspring 24/10/24
new_population.extend([offspring 1, offspring 2])
population = new_population[: pop_size]. //

genetic algorithm(pop_size, num_of_gen, x=range_low, x_range_high, mutation_rate, crossover_rate)

07/11/24

Particle Swarm Optimization

1. Rastrigin function

Purpose:

The Rastrigin function is a benchmark function used for optimization, it is used to evaluate the fitness of the particle's position during optimization.

Algorithm:

Given a vector of n dimensions, the rastrigin function is defined as

$$f(x) = 10n + \sum_{i=1}^{n} [x_i^2 - 10\cos(2\pi x_i)]$$

Pseudocode:

```
function rastrigin(X):
    n = length of X
    result = 10*n
    for i = 1 to n
        result = result + [X[i]^2 - 10*cos(2*pi*X[i])]
return result
```

2. Particle class

Purpose:

The particle class represents an individual solution (particle) in the swarm. Each particle has a position, velocity, and the best position it has found so far.

Bafna Gold
Date: Page:

Pseudocode

Class particle:
function __init__(dim, bounds)
position = random values in bounds
velocity = random values between -1 and 1
best_position = position
best_value = ranorigin(position)

function evaluate():
current_value = ranorigin(position)
if current_value < best_value:
best_value = current_value
best_position = position

3. PSO Algorithm

purpose

The PSO algorithm is the main procedure that drives the swarm to converge towards an optimal solution, It updates particle velocities and positions iteratively based on their own best position and the global best position found by the entire swarm.

Pseudo code
function pso(dim, bounds, num_particles, max_iter, w, c1, c2)
particles = [create particle(dim, bounds) for i in
num_particles]
global_best_position = None
global_best_value = infinity
for iter = 1 to max_iter:
for each particle in particles:
particle.evaluate();

```
        if particle.best_value < global_best_value:
            global_best_value = particle.best_value
            global_best_position = particle.best_position

    for each particle in particles:
        inertia = w * particle.velocity
        cognitive = c1 * random_value * (particle.best_position
                    - particle.position)
        social = c2 * random_value * (global_best_position
                 - particle.position)
        particle.velocity = inertia + cognitive + social
        particle.position = particle.position + particle.velocity
        particle.position = clip(particle.position, bounds[0],
                            bounds[1]);
    return global_best_position, global_best_value
```

4. Velocity and Position Update:

The velocity and position of each particle are updated at each iteration based on three key components: inertia, cognitive, and social.

Pseudo code:

```
function update_velocity_and_position(particle, global_best_
         position, w, c1, c2, bounds):
    inertia = w * particle.velocity
    cognitive = c1 * random() * (particle.best_position - particle.
                position)
    social = c2 * random() * (global_best_position -
             particle.position)
```

particle.velocity = Inertial + Cognitive + Social.

~~particle.position = random + global_best_position~~

particle.position = particle.position + particle.velocity.

particle.position = clip (particle.position, boundsCo), bounds[1])

Ant colony optimization for travelling Salesman problem:

1) Define the problem - Create a set of cities with coordinates.

pseudocode

cities = array of city coordinates.

function calculate_distance(cities):
    num_cities = length(cities)
    distances = matrix of size num_cities x num_cities
    for each city i:
        for each city j:
            distances[i][j] = Euclidean distance between city i and city j
    return distances

explanation

The function calculate_distances computes the pairwise Euclidean distances between all cities. This forms a distance matrix that will be used later to evaluate the length of tours created by the ants.

2) Initialize Parameters

pseudocode

num_ants = number of ants.
alpha = influence of pheromone
beta = influence of heuristic (inverse distance)
rho = evaporation rate of pheromone
num_iterations = num of iterations.
initial_pheromone = initial pheromone value of all edges

distances = calculate_distances(cities)

pheromone_matrix = matrix of size num_cities x num_cities filled with initial_pheromone,

eta = matrix of heuristic information (1/distances)

* Set up the algorithm parameters such as the number of ants, the influence of pheromones and heuristic information, the pheromone evaporation rate, and the number of iterations.
* Initialize the pheromone_matrix where all values stand with same initial pheromone level.
* Calculate the heuristic matrix eta, which is the inverse of the distance matrix. That emphasizes the shorter distances during the construction of solutions.

3. Heuristic Information - heuristic function:

Function heuristic(distances)

Create an empty matrix eta of the same size of distances.

For i = 0 to number of rows in distances:

for j = 0 to number of columns in distances:

If distances[i][j] != 0:

Set eta[i][j] = 1/distances[i][j]

Else:

Set eta[i][j] to a large value (to avoid division by zero).

return eta.

* The heuristic function computes the inverse of the distance matrix where each element represents the

heuristic value for moving from one city to another. This value helps the ant favor paths by influencing the probability calculation.

3. Choose Next City - choose_next_city_function

Pseudo code:

Function choose_next_city(pheromone, eta, visited):
    initialize an empty list probs
    For each city j from 0 to num_cities - 1:
        If city j has not been visited:
            Calculate pheromone_ij = pheromone[visited[-1]][j] raised to alpha
            Calculate heuristic_ij = eta[visited[-1]][j] raised to beta
            Append pheromone_ij * heuristic_ij to probs
        Else:
            Append 0 to probs
    Normalize probs so that the sum of all probabilities equals 1
    Return a randomly selected city index based on the probabilities

Explanation:

This function selects the next city for an ant to visit based on a probabilistic approach.

4. Construct Solution - Construct_Solution function

Function Construct_Solution(pheromone, eta):
    Initialize an empty list tour
    Select a random city and add it to the tour
    While the length of tour is less than num_cities:
        Call choose_next_city to select the next city

Add the selected city to the tour
return the tour.
* This function constructs a complete tour for a single ant.

5. Update Pheromone

Function update_pheromone (pheromone, all_tours, distance, best_tour):
    Evaporate pheromone by multiplying it by (1 - rho).

    For each tour in all_tours:
        Calculate the tour length by summing the distances between consecutive cities in the tour
        For each consecutive pair of cities in the tour:
            Add 1/tour_length to pheromone[city1][city2]
    Calculate the length of the best tour
    For each consecutive pair of cities in the best tour:
        Add 1/best_length to pheromone[city1][city2]

* This function updates the pheromone matrix after each iteration. Pheromone values evaporate (decay) over time, and new pheromones are deposited based on the tours taken by all ants.

6. ACO Main loop - run_aco function

Function run_aco(distances, num_iterations):
    Initialize pheromone matrix with initial pheromone value.

Set best_tour to None and best_length to Infinity

For iteration = 1 to num_iterations:
Initialize an empty list all_tours
For each ant:
Construct a solution (tour) using construct_solution
Add the tour to all_tours.
Calculate the length of each tour in all_tours.
Find the best tour and best length from all_tours.
If the best tour found in this iteration is better than the previous best:
Update best_tour and best_length.
Update the pheromone matrix using update_pheromones.
print the best length at the end of this iteration
return best_tour and best_length

* The run_aco function runs the ACO algorithm over multiple iterations.
* In each iteration, multiple ants construct their tours.
* The best tour is selected, and pheromones are updated accordingly.
* The algorithm continues until the maximum number of iterations is reached, and the best solution is returned.

7) Plot the Best Route - plot_route function

function plot_route(cities, best_tour):
Create a figure for plotting
For each city in cities:
plot the city as a red point and label it with its index.

Create a list of cities in the best_tour, including the first city at the end to complete the loop
Plot the cities in the best_tour as blue lines connecting them.
Set title as "Best Tour (Length: best_length)"
Add labels for x and y axes
Show the plot.

This function visualizes the best tour found by the ACO algorithm. It plots the cities as red points, connects the cities in the order of the best tour with blue lines, and displays the result in a plot, showing the route and the length of the best tour.

14/11/24

21/11/24

Cuckoo Search (CS)

1. Define the problem

The problem is defined by an objective function, in this case, we use the Sphere function $f(x) = \sum x_i^2$ as the function to minimize.

Pseudocode

objective function f(x) = Sum(x_i^2)

2. Initialize parameters

Parameters such as the number of nests, number of iterations, discovery rate, and search space boundaries are initialized.

pseudocode:

```
num_nests = 25
num_iterations = 100
discovery_rate = 0.25
dim = 5
lower_bound = -10
upper_bound = 10
```

3. Initialize Population

* Population of nests is randomly initialized within the defined boundaries.

Pseudo code

```
nests = random.uniform(lower_bound, upper_bound, num_nests, dim).
fitness = [evaluate fitness(nest) for nest in nests]
```

4. Evaluate fitness

The fitness of each nest is evaluated using the objective function. The best nest is selected based on the lowest fitness value.

Pseudocode:

```
best_nest = min(nests, by = fitness)
best_fitness = min(fitness)
```

5. Generate New Solutions (Levy flight):

New solutions are generated via Levy flight, which helps explore the search space more efficiently.

Pseudocode:

```
for each nest:
    step_size = levy_flight(lambda)
    new_solution = nest + step_size * (nest - best_nest)
    new_solution = clip(new_solution, lower_bound,
                        upper_bound)
    fitness = evaluate_fitness(new_solution).
```

6. Abandon Worst nests

Some of the worst nests are abandoned and replaced with new random solutions based on the discovery rate.

Pseudocode:

```
for each nest with a probability discovery_rate:
    new_random_solution = random.uniform(lower_bound,
                                         upper_bound).
    fitness = evaluate_fitness(new_random_solution).
```

7. Iterate

The algorithm repeats the process of generating new solutions, updating best, and evaluate fitness over multiple iterations.

Pseudocode:

```
for iteration in range(num_iterations):
    generate_new_solutions()
    update_best_next()
    track_best_fitness()
```

8. Output the Best Solution

After completing the iterations, the best nest found is returned, representing the optimal solution.

Pseudo Code:

```
return best_nest, best_fitness
```

Applications:

Structural optimization, Aerospace design, Feature selection, Hyperparameter Tuning, bridge design, fault optimization, noise reduction.

21/11/24

Grey Wolf Optimizer

1. Define the problem (objective function).

Define the objective function that takes a position of a wolf (a candidate solution) and returns the function's value at that position. This value is called the fitness. Here I'll use the Sphere function, which calculates the sum of the squares of the position vector components.

Pseudocode

```
FUNCTION objective_function(position):
    Sum = 0
    For i in range(0, len(position)):
        Sum += position[i]^2
    Return SUM.
```

2) Initialize the parameters

Set the number of wolves (population size), the number of iterations and the bounds which the wolves will search for the optimal solution.

Pseudocode

```
num_wolves = 30
num_dimensions = 5
num_iterations = 100
bounds = [-10, 10]          # lb = -10, Ub = 10
```

3) Initialize population

Generate random positions for all the wolves

in the population within the defined bounds.

Pseudo code function initialize_wolves(num_wolves, num_dimensions, bounds):

```
Population = []
For i in range(num_wolves):
    position = []
    For d in range(num_dimensions):
        position[d] = random_value_within(bounds)
    population.append(position)
Return Population.
```

Explanation: Each wolf's position is initialized randomly within the search space bounds. Each wolf is a candidate solution represented by a vector in multi-dimensional space.

9) Evaluate fitness (Rank wolves at Alpha, beta, delta)
pseudo code

```
Function evaluate_fitness(population):
    alpha_fitness = infinity
    beta_fitness = infinity
    delta_fitness = infinity

    For wolf in population:
        fitness = objective_function(wolf_position)

        if fitness < alpha_fitness:
            delta_fitness = beta_fitness
            delta_position = beta_position
```

```
        beta_fitness = alpha_fitness
        beta_position = alpha_position
        alpha_fitness = fitness
        alpha_position = wolf_position.
    Else if fitness < beta_fitness:
        delta_fitness = beta_fitness
        delta_position = beta_position
        beta_fitness = fitness
        beta_position = wolf_position.
Else if fitness < delta_fitness:
        delta_fitness = fitness.
        delta_position = wolf_position..

Return alpha_position, beta_position, delta_position.
```

Explanation: Each wolf's fitness is calculated using the objective function and the wolves are ranked. The top three wolves (with the best fitness values) are classified as alpha, beta, and delta wolves.

5) Update Positions (Simulate wolves' Movement)

Pseudocode

```
Function update_positions(population, alpha_position,
beta_position, delta_position, a, bounds):
    For each wolf in population:
        For d in range(num_dimensions):
A1 = 2 * a * random_value() - a
A2 = 2 * a * random_value() - a
A3 = 2 * a * random_value() - a
C1 = 2 * random_value()
C2 = 2 * random_value()
C3 = 2 * random_value()
```

D_alpha = abs(C1 * alpha_position[d] - wolf_position[d])
D_beta = abs(C2 * beta_position[d] - wolf_position[d])
D_delta = abs(C3 * delta_position[d] - wolf_position[d])

X1 = alpha_position[d] - A1 * D_alpha
X2 = beta_position[d] - A2 * D_beta
X3 = delta_position[d] - A3 * D_delta

wolf_position[d] = (X1 + X2 + X3)/3
wolf_position[d] = (clamp(wolf_position[d], bounds[0], bounds[1])

Each wolf's position is updated by calculating the distance from alpha, beta, and delta wolves. New positions are based on the weighted influence of these three best wolves. The "a" parameter decreases over time to switch from exploration to exploitation.

6) Iterate

for iteration in range(num_iterations):
    alpha_position, beta_position, delta_position =
        evaluate_fitness(population)
    a = 2 - (iteration * (2/num_iterations))
    Update_position(population, alpha_position, beta_position, delta_position, a, bounds)

Exploration: The algorithm runs through multiple iterations, update wolves's positions in each iteration, The coefficient "a" decreases over time, shifting the focus from exploration to exploitation, The best solution is printed at each iteration.

3) Output the Best Solution.

Print "Final best Solution", alpha_position
Print "Fitness of best Solution", alpha_fitness

At the end of the iterations, the best Solution (alpha wolf's position) and its fitness are reported. This is the optimal or near-optimal solution found during the optimization process.

Applications:

1. optimization problems in Engineering (Structural design).
2. Machine learning and Artificial intelligence (feature extraction and selection).
3. optimization of Energy Systems (Renewable energy optimization).

Sneha B
28/11/24

17/12/24

Parallel cellular Algorithms:

→ objective function

The function is used as sphere function particularly used in the optimization algorithms.

```
def sphere_function(position):
    return sum(position_i^2 for each position_i in position)
```

→ Initialize the Population Grid:

This function is used to create a grid of shape and fill with random values uniformly distributed between lower_bound and upper_bound.

```
def initialize_population(grid_size, solution_dim, lower_bound, upper_bound):
    Initialize grid with random values in given bounds return grid.
```

→ Evaluate fitness

This function is used to evaluate the fitness.

for each cell in the grid

```
def evaluate_fitness(grid):
    for each cell in grid:
        Calculate fitness using sphere_function(cell position)
    return fitness grid
```

→ Get Neighbours of a cell:

```
def get_Neighbours(grid, i, j):
    Initialize empty list for neighbours
```

For each (di, dj) in [(-1,-1), (-1,0), ... (1,1)]
excluding (0,0):
ni = (i+di) % grid_size
nj = (j+dj) % grid_size
Add grid[ni][nj] to neighbours list
return neighbours

→ Update States of all cells in the Grid
the function is used for get the neighboring cells, identify the neighbor with best fitness, and move cells towards the best neighbor's position.

def Update_States(grid, fitness, learning_rate):
Initialize new_grid as a copy of grid
For each cell(i,j) in grid:
Get neighbours of cell(i,j)
Find the best_neighbour with minimum fitness update cell position:
new_grid[i,j] = grid[i,j] + learning_rate *
(best_neighbor - grid[i,j])
return new_grid

→ Main parallel Cellular Algorithm
The main algorithm includes initialize the population grid, and for a given number of iterations evaluate fitness, update states based on neighbors & track best solution, and output best solution & fitness.

def parallel_cellular_algorithm(grid_size, solution_dim, lower_bound, upper_bound, iterations, learning_rate):
Initialize grid with random positions
Set best_solution = None and best_fitness = infinity

```
For Iteration in range(iterations):
    Evaluate fitness of all cells.
    Identify current best solution and fitness
    of current fitness if better than best_fitness
        Update best_solution and best_fitness.
    Update all states using neighbour and learning
    _rate
    print current best_fitness.
Output best_solution and best_fitness
if __name__ == "__main__":
    parallel_cellular_algorithm(grid_size = 10, solution
                                _dim = 2, lower_bound = -5,
                                upper_bound = 5.0, iterations = 50,
                                learning_rate = 0.2)
```

Output:

Best Solutions: [-0.00959221 0.03401244]

Best Fitness: 0.00125

16/12/24

Optimization via Gene Expression Algorithm

→ objective function

This function calculates fitness of a given solution using rastrigin objective function.

```
def rastrigin(x):
    A = 10
    result = A * length(x)
    for each xi in x:
        result += (xi^2 - A * cos(2 * pi * xi))
    return result
```

→ Initialize population

Initializes a random population of solutions, each solution represented by a set of genes.

```
def initialize_population(pop_size, num_genes, lower_bound,
                          upper_bound):
    population = Randomly generate pop_size individuals
                 with num_genes in range [lower_bound,
                 upper_bound]
    return population
```

→ Evaluate fitness

```
def evaluate_fitness(population):
    fitness = list of . rastrigin (individual) for
              each individual in population.
    return fitness.
```

→ Tournament Selection:

```
def Tournament_selection(population, fitness, tournament_size):
    selected = for each individual, select the winner of a random tournament from the population.
    return selected
```

→ Crossover:

Create two offspring from two selected parents by combining their genetic material at a random crossover point.

```
def Crossover(parent1, parent2):
    crossover_point = Randomly choose a point
    child1 = Combine parent1 & parent2 at crossover_point
    child2 = Combine parent2 & parent1 at crossover_point
    return child1, child2
```

→ Mutation:

Introduce random mutations in the offspring with a given probability, allow the solution to explore new areas of the search space.

```
def mutate(child, mutation_rate, lower_bound, upper_bound):
    For each gene in child, randomly mutate with probability mutation_rate.
    return mutated child
```

→ Gene Expression Algorithm:

This is main loop that evolves the population over several generations, applying selection, crossover, mutation, and fitness evaluate to find the best solution

```
def gene_expression_algorithm(pop_size, num_genes,
lower_bound, upper_bound, mutation_rate, crossover_rate,
num_generations):
    population = Initialize population
    best_solution = None, best_fitness = infinity
    For generation = 1 to num_generations:
        fitness = evaluate fitness.
        Update best solution if needed
        Selected_population = Tournament_Selection
        new_population = For each pair of parents apply
        crossover and mutation
        population = new_population.
    return best_solution, best_fitness.
```

18/12/24