week - 01

2-10-24

1) Genetic Algorithm for optimization problems:-

Genetic Algorithm (GA) is an optimization techni -que inspired by the process of natural selection & genetics. it is part of evolutionary computa -tion & is used to find approximate solutions to complex problems by simulating evolution. GAs work with a population of potential solu -tions. evolving them over generations thro -ugh selection crossover & mutation

Key components of genetic algorithm

1. Population: A set of potential solutions to the problem
2. chromosome: A representation of a solution (often in the form of strings, array, or other data structures
3. Fitness Function: A function that evaluates how good a solution is relative to others
4. Selection:- The process of choosing the best solutions to create the next genera -tion
5. crossover: The process of combining two parent solutions to create offspring
6. Mutation: Random alterations to offspri -ng to maintain diversity in population

Application of genetic algorithm for optimization problems

1. Travelling Salesman Problem(TSP)

* Description: the goal is to find the shortest possible route that visits each city exactly once & returns to the origin city

* Application: GAs can generate & evolve routes by selecting shorter paths, applying crossover to combine segments of different routes & using mutation to introduce variations

2. Knapsack problem

* Description: Given a set of items with weight & values, the task is to maximize the total value without exceeding a weight limit

* Application: GAs can encode item selection as chromosomes using fitness functions to evaluate total values while adhering to weight constraints

3. Job scheduling

* Description: Assigning jobs to resource in a way that optimizes objectives like minimizing total completion time (or) maximize resource utilization

* Application :- GAs can create schedules as potential solutions, applying selection & crossover to find optimal arrangements using mutation to explore new configur-ations.

Algorithm for Travelling Salesman problem :-

Function Genetic Algorithm (population_size, gene-rations, distance_matrix)

Initialize population with random tours (permu-tations of cities)

For generation from 1 to generations Do

calculate fitness for each tour in population

fitness = 1 / (total distance of the tour)

Create empty list next_generation

While next_generation is not full Do

Parent1 = Selection (population, fitness)

Parent 2 = Selection (Population, fitness_scores)

child = crossover (parent1, Parent2)

child = Mutate (child)

ADD child to next_generation

SET population = next_generation

best_tour = Find the tour with the best fitness in population

best_distance = 1 / best_fitness

Return best_tour, best_distance

Function crossover(parent1, parent2)

Select two crossover points

Create child by combining segments from parent1 & parent2 while maintaining order

Return child

Function Mutate(tour)

If random_value < mutation_rate then

Select 2 positions in the tour

Swap cities at those 2 positions

Return tour

function Selection(population, fitness_scores)

Select a candidate based on fitness scores.

Return selected candidate

Enter number of cities:

Lab-2 24/10/24

Particle Swarm Optimization for function Optimization

Particle Swarm optimization is a powerful & versatile optimization technique inspired by collective behaviour of nature particularly in the movement of birds in flocks & fish in schools. This algorithm harness the principles of social collaboration & individual learning to explore complex solution spaces & converge towards optimal solution

Define the problem

Mathematical function: We will create a mathematical function to optimize for example lets define the function $f(x) = x^2$. The goal is to find the value of x that minimizes the function. which known to be at $x = 0$

Initialize parameters

Number of particles: Choose an appropriate number of particles for the swarm

inertia weight: Set inertia weight to control the impact of the previous velocity.

cognitive coefficient: which influences how much particle is influenced by its own best position

Social co-efficient: which influences how much particle is influenced by the best position found by the swarm

Generate initial population:-

Create initial population of particles. for each particle

Randomly initialize its position within a defined range

Randomly initialize its velocity

Set its best position = initial position

Evaluate its initial best fitness

Evaluate fitness:-

For each particle, calculate its current fitness using the defined mathematical function, update its best_position & best_fitness if the current fitness is better than its previous

Pseudo code

function PSO(fitness_function, num_particles, num_iterations, inertia_weight, cognitive_coefficient, social_coefficient):

Create an array of particles with size num_particles

For i from 0 to num_particles - 1;

Particles[i].position = Randomly initialize position in range [-5, 5]

Particles[i].~~position~~ velocity = Randomly initialize velocity in range [-1, 1]

particles.best_position = particles[i].position

Particles.best_fitness = fitness_function(Particles[i].position)

global_best_position = particles[0].best_position

global_best_fitness = particles[0].best_fitness

```
for iteration from 1 to num_iterations:
    For each particle in particles:
        current_fitness = fitness_functi
            -on(particle.posi
            -tion)
        if current_fitness < particle.best_fit
            -ness:
            particle.best_position = parti
            -cle.position
            Particle.best_fitness = current
            -fitness
        if current fitness < global best_fitness
            global best_position = particle.posi
            tion
            global_best_fitness = current_fitness
    r1 = Random value in range [0,1]
    r2 = Random value in range [0,1]
    particle.velocity = (inertia_weight * parti
    -cle.velocity + cognitive_coff * r1 *
    (particle.best_position - particle.posi
    -tion) + social co-efficient * r2
    * (global_best_position - particle
    .position))
    particle.position = particle position + particle
    velocity.
```

return, global_best_position, global_best_fitness

best position: [-1.8777 7640 e+4 -1.5801104 1e-13

best value: 5.80651522205570 e-25

SAA
24-10-24

7/11/24

Lab-3

Ant colony optimization for travelling salesman problem

ACO is inspired by the behaviour of real Ants when they forage for food. Ant explore their environment & deposite a chemical substance called pheromone on the ground, the pheromone trail is a stronger on paths that are shorter & more frequently used by Ants. This leads other ants to favour these paths thus creating a positive feedback loop, over time shortest path will accumulate the most pheromone & be chosen by more ants

Define problem:

In this we define the cities as a dictionary where each key represents a city ID & the corresponding value is a tuple of its (x, y) co-ordinate in 2D plane. The n_cities variable stores the total no of cities to be visited in the TSP

Initialize parameters

Number of Ants (n_ants): Number of ants in the colony

pheromone Importance (alpha): Influence of pheromone on decision

Heuristic Importance (beta): Influence of the distance

Evaporation Rate (rho): Rate at which pheromone evaporates

Initial pheromone: Set initial pheromone value for each edge

Construct Solutions:

Ant path construction: Each ant starts at a random city & probabilistically chooses the next city based on pheromones & heuristic information

Transition probability: Ants are more likely to visit cities with higher pheromone levels & shorter distances

Repeat for 3 to 4 steps for a fixed no of iterations

Applications of ACO

Travelling Salesman

Vehicle routing

Network Routing

Robotics

Pseudocode:

Initialize parameters: n_ants, n_iterations, alpha, beta, rho, Q, cities

calculate distance matrix for all cities

Initialize pheromone matrix (small values)

for iteration = 1 to n_iterations:

for each ant:

Initialize the tour of the ant (start city)

while ant not visited all cities:

calculate probability next choosing city on pheromone level & distance using alpha, beta & gamma

update tour by adding next cities

Calculate total length for ant

update pheromone matrix

Evaporate pheromone using probabilistic function

Deposit pheromone on the edge of each ant's tour on the edges it visited

update the best solution found

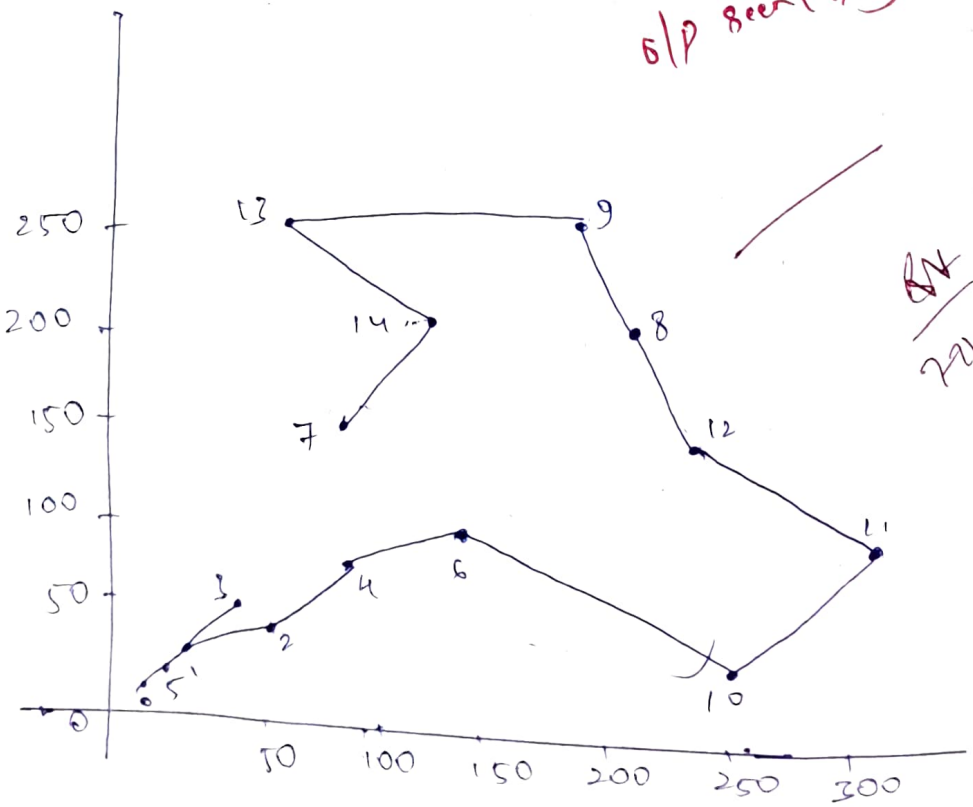
If any stopping condition is met end for

Return the ~~best~~ solution found

output :

Best tour length: 986.30

o/p seen (5,6)



Lab-4

cuckoo search(CS)

cuckoo search is a nature-inspired optimization algorithm based on brood parasitism of some cuckoo species. This behaviour involves laying eggs in the nests of other birds, leading to optimi-zation of survival strategies. CS uses levy flight to generate new solutions promoti-ng global search capabilities & avoiding local minima

Algorithm

Initialize nests randomly within in solution space

best_nest = nest with best fitness

best_fitness = fitness of best nest

for iteration = 1 to max_iter

for each nest i

new_nest_i = nest_i + alpha * levy_flight(lambda)

new_nest_i = clip(new_nest_i, lower_bound, upper_bound)

```
        new_fitness_i = objective_function(new_nest_i)

        if new_fitness_i < fitness_i:
            nest_i = new_nest_i
            fitness_i = new_fitness_i

    for i in range(num_nests):
        if random() < beta:
            nest_i = random_solution()

    current_best_fitness = min(fitness)

    if current_best_fitness < best_fitness:
        best_fitness = current_best_fitness
        best_nest = nest with best fitness

return best_nest, best_fitness
```

Parameters:

Levy flight: A random walk where the step size follows a heavy tailed distribution

Alpha(α): Scaling factor that controls the size in levy flight

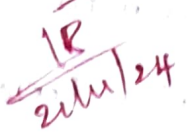
Beta (β): Probability of abandoning nest

Lambda(λ): Defines characteristics of levy distribution

Applications:

Transportation (vehicle routing, traffic control)

manufacturing (job scheduling, resource allocation)



Lab-5

Grey wolf optimizer (GWO)

The grey wolf optimizer algorithm is a swarm intelligence algorithm inspired by the social hierarchy & hunting behaviour of grey wolves. it mimics the leadership structure of alpha, beta, delta & omega wolves & their collaborative hunting strategies. The GWO algorithm uses these social hierarchies to model the optimization process. The alpha wolves guide the search process while beta & delta wolves assist in refining the search direction

Algorithm:-

Step 1:- Randomly initialize Grey wolf population of N particles x_i $(i = 1 \ldots n)$

Calculate the fitness value of each individuals

Sort grey wolf population based on fitness values

alpha_wolf = wolf with least fitness value

beta_wolf = wolf with second least fi
-tness value

gamma_wolf = wolf with third least fit
-ness value

for iter in range(max_iter):

calculate the value of a

a = 2*(1 - iter/max_iter)

for i in range(N):

a. compute the value of A1, A2,
A3 & C1, C2 & C3

A1 = a*(2*r1 - 1), A2 = a*(2*r2 - 1)

A3 = a*(2*r3 - 1)

C1 = 2*r1 C2 = 2*r2, C3 = 2*r3

Compute x1 x2 x3

x1 = alpha_wolf.position -

A1*abs(C1*alpha_wolf.
Posith - ith_wolf.position)

x2 = beta_wolf.position.

A2*abs(C2*beta_wolf - positi
- it.wolf.position)

x3 = gamma_wolf.position.

A3*abs(C3*gamma_wolf.Posi
-tion - ith_wolf.positio
-n)

compute new solution & its fitness

$x_{new} = (x_1 + x_2 + x_3)/3$

$f_{new} = fitness(x_{new})$

if ($f_{new} <$ ith wolf fitness)

ith-wolf.position = x_{new}

ith-wolf.fitness = f_{new}

- Sort grey wolf population based on fitness value

alpha_wolf = wolf with least fitness value

beta_wolf = wolf with second least value

gamma_wolf = wolf ~~third~~ third least fitness value

Return the best wolf in the population

IB
21/4/24

Parallel cellular Algorithms & programs

Parallel cellular algorithms are inspired by functioning of biological cells that operate in a highly parallel & distributed manner These algorithms leverage the principles of cellular automata & parallel computing to solve complex optimization problems efficien -tly each cell represents a potential solut -on & interacts with its neighbours to update its state based on predefined rules This interaction models the diffusion of informatio -n across cellular grid.

Algorithm

```
function initialize_grid(width, height):
    grid = create_grid(width, height)
    for each cell in grid:
        grid[cell.x][cell.y] = randomState()
    return grid

function count_alive_neighbors(grid, x, y):
    count=0
    for each neighbor (dx,dy) in [-1,0,1]:
        if (dx,dy) != (0,0):
            neighbor_x = (x+dx+width)%. width
```

```
        neighbor_y = (y + dy + height) % height
        count += grid[neighbor_x][neighbor_y]
    return count

function next_state(grid, x, y):
    alive_neighbors = count_alive_neighbors(grid, x, y)
    if grid[x][y] == 1:
        return 1 if alive_neighbors == 2 or alive_neighbors == 3 else 0
    else:
        return 1 if alive_neighbors == 3 else 0

function parallel_update(grid):
    next_grid = create_grid(width, height)
    parallel_for each cell(x, y) in grid:
        next_grid[x][y] = next_state(grid, x, y)
    return next_grid

function run_simulation(width, height, num_steps):
    grid = initialize_grid(width, height)
    for step in 1 to num_steps:
        grid = parallel_update(grid)
        display_grid(grid)
```

Gene Expression optimization

Gene Expression optimization is a bio inspired optimization algorithm that mimics gene expression in biological systems & It is designed to solve complex optimization problems by finding optimal solution through iterative improvement using principles such as expression, mutation & natural selection

Pseudocode:

```
population size - p
Max iteration T
Mutation rate - m
crossover rate - c
fitness function F

population = initialize_population(population
    size)
fitness value = evaluate_fitness(population,
                fitness function
for t in range(max_iterations):
        Selected_parents = Selection(populati
            - on, fitness_value)
```

```
offspring = crossover(selected_parents,
                      crossover_rate)
offspring = mutation(offspring,
                     mutation_rate)
offspring_fitness = evaluate_fitness
    (offspring, fitness
     function)

population = combine_&_select(population,
    offspring, fitness_values, offspring_fitness)

fitness_values = evaluate_fitness(population,
                                  fitness_function)

best_solution = get_best_solution(population,
                                  fitness_values)
print("Best solution found:", best_solution)
```