METRO
Page:
Date: 03/10/2024

1. Genetic Algorithm for Optimization Problems

Genetic Algorithms are inspired by the process of natural selection and genetics where the fittest individuals are selected for reproduction to produce the next generations. These are widely used for solving optimization and search problems.

Implementation Steps:

1. Define the Problem — Consider the optimize mathematical function, " $f(x) = \sin(x) \cdot x^2$ "
where x is in the range -10 to 10. Our goal is to find the value of x to maximize the function.

2. Initialization — Initialize the parameters such as population size, mutation_rate, crossover_rate, generations, gene_length, x_range = (-10, 10).

3. Create Initial Population: Generating a number of initial populations by altering them noting down the changes by changing number of population size.

4. Evaluate fitness: Generating an evaluate function to each individual to measure the fitness of each individual in the population.

5. Selection: Selecting only fittest individuals who are eligible to undergo genetic algorithm according to their fitness.

6. Crossover: A crossover is a process where the it is applied to individuals to generate offspring from the parents which allows the algorithm to new regions of the solution space.

7. Mutation: Mutation of individuals by adding a small value to maintain diversity in the population.

8. Iteration: The iteration process is repeating the process for a set number of generations and track the best solution at each step.

Applications of Genetic Algorithm

Optimization Problems

1. Travelling Salesman Problem:

Objective: Minimize the total distance travelled while visiting a set of cities once and returning to starting point.

Application: Logistics and delivery routing.

2. Knapsack Problem:

Objective: Maximize the total value of items selected for a knapsack without exceeding weight capacity

Application: Resource Allocation in finance management

3. Job Scheduling:

Objective: Schedule jobs on machines to minimize total completion time, latencies (delay) or idle time

Application: Manufacturing, service industries

→ Python Code Description for Optimization of Travelling Salesman Problem using Genetic Algorithm.

```
Function Genetic Algorithm (Population size, generations, distance matrix)
    Initialize population with parameters such as
    random tours (permutation of cities)

    For generation from 1 to end Do
        Calculate fitness for each tour in population
        fitness = 1 / (total distance of tour)

    Create empty list next_generation.
    while next_generation is not full do
        parent1 = Selection (population, fitness_score)
        parent2 = Selection (population, fitness_score)
        child = crossover (parent1, parent2).
        child = mutate (child).
        Add child to next_generation.

best_tour = Find the tour with the best fitness in populatn
best_distance = 1 / best_fitness
Return best_tour, best_distance
End Function.

Function Crossover (parent1, parent2)
    Select two crossover points
    Create child by combing segments from parent 1 & parent2
    return child
End Function
```

Function. Mutate (tour).

If random_value < mutation_rate then

Select two positions in the tour

Swap the cities at those positions

End if

Return Tour

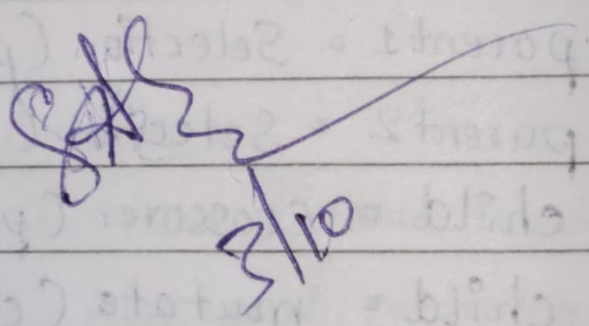
End function.

Function Selection (population, fitness_scores)

Select a candidate based on fitness scores

return selected candidate

End function:



METRO
Page:
Date: 24, 10, 2024

2. Particle Swarm Optimization

→ To find optimal solutions for a problem.

The objective of this algorithm is to optimize a mathematical function.

Implementation Steps:

1. Define the Problem, creating a mathematical function to optimise. The functions fitness can be ① Rastrigin function or ② Sphere function.
2. Initialize parameters such as number of particles, set bounds for solution space and choose coefficients that influence.
3. Create a function to evaluate the function of each particle based on optimization function defined.
4. Update velocities & positions during fixed number of iterations.
5. Output the best solution found during the iterations.

→ Fitness function used : Rastrigin function.

$$f(x_1 \ldots x_n) = 10n + \sum_{i=1}^{n} \left(x_i^2 - 10\cos(2\pi x_i)\right)$$

minimum value at $f(0,0)$ is 0.

Pseudo Code:

```
function PSO (num_particles, bounds, num_iterations)
    Initialise parameters such as inertia_weight, coefficients.
    Initialise particles as empty list.

    for i from 1 to num_particles,
        create a particle within the bound range.
        add each particle to the particle list.
```

best_position = particles[0].best_position.
best_fitness is calculated by evaluation function by pushing its best position.

from 1 to num_iterations:
    for each particle in particles:
        fitness = EvaluvateFitness(particle.position)
        if fitness < particle.best_fitness
            particle.best_fitness = fitness
            particle.best_position = particle.position.

        if fitness < best_fitness:
            best_fitness = fitness
            best_position = particle.position:

    r1, r2 = Randomvalues().
    particle.velocity = (inertia_weight * particle.velocity
        + coefficient * r1 * (paiticle.best_
        position) + r2 * (particle.position))

return best_position, best_fittness.

function CreateParticle(bounds):
    position = Random Uniform (bounds)
    velocity = Random Uniform ([-1, 1])
    best_position = position.
    best_fitness = EvaluateFitness (position)

return Particle (position, velocity, best_position, best_fitness)

function EvaluateFitness(particle.position)
    return RastriginFunction(position)

Output best_solution.

Output:

Best position : [1.03137107 -1.00702547]
Best fitness (Rastrigin): 9.48491320

METRO
Page:
Date: 07, 11, 2024

3. Ant Colony Optimization for the Travelling Salesman Problem

The behaviour of ants inspired the development of optimization algorithms.

ACO simulates the way ants find the shortest path between food sources and their nest.

TSP → Shortest possible route that visits a list of cities and return to origin city.

Implementation Steps:

1. Define the Problem: Create a set of cities with their coordinates. (x,y) coordinates.
2. Initialize Parameters: Set number of ants, importance of pheromone (alpha), importance of heuristic info (beta), the evaporation rate (rho), intial pheromone value
Pheromone — chemical substance released by ants when they move between / travel. → Guiding Search for solutions.
3. Construct Solutions: Probabilistically choosing next city on the basis of pheromone & heuristic info.
4. Update Pheromones: After all solutions, update the pheromone based on quality of solutions.
5. Iterate: Repeat the process for no. of iterations
6. Output the solution best possible.

Applications of ACO:

- Travelling Salesman Problem - shortest possible route.
- Vehicle Routing - most efficient set of routes
- Network Routing - Best path for data to travel.
- Robotics - optimal / feasible for path robot to move.
- Supply Chain & Inventory Management.

Optimize flow of goods & services from services to customers. . .

Pseudo Code

Initialise parameters : n_ants, n_iterations, alpha, beta, rho, Q (pheromone value), cities.
Calculate distance matrix for all cities
Initialise pheromone matrix (small values)

for iteration = 1 to n_iterations:
    for each ant:
        Initialise the tour of the ant (start city).
        while ant not visited all cities:
            Calculate the probability next choosing city on pheromone level and distance. using alpha, beta, gamma.
            Update the tour by adding next city.
        Calculate total length for the ant.

    update phermone matrix:
        Evaluate phermone using probabilistic function.
        Deposit phermone on the edger 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.

Cities = [(0,0), (1,3), (3,1), (6,4), (8,0)]

Result : Iteration 1/20, Best length : 22.655 10 n_ants = 10
Iteration 20/20, Best Length : 20.994 n_iterations = 20
Best Tour : [4, 3, 1, 0, 2]
Best Tour Length : 20.9947

METRO
Page:
Date: 14/11/24

4. Cuckoo Search (CS)

Nature inspired optimization algorithm based on the brood parasitism of some cuckoo species.

Laying eggs in the nests of other birds, optimization of survival strategies.

Using Levy flights to generate new solutions.

Used mainly to solve continuous optimization problems and in various domains — Engg. Design, ML & Data mining.

implementation

1. Define the Problem by math function which is to be optimised
2. Initialise Parameters such as No. of nests, probability of discovery and no. of iterations.
3. Initialize population of nests with random positions.
4. Evaluate fitness based on optimization function.
5. Create new solutions via levy flights.
6. Abandon worst nests & replace with new positions.
7. Iterate through each of the iterations.
8. Output the best solution.

Pseudo Code:

```
Initialize population of nests randomly
for i = 1 to no_of_iterations
    for each nest:
        evaluate fitness
        generate new solution (using levy flight)
        evaluate fitness of new solution
        if new solution is better:
            replace current solution = new solution
    for each nest of probability pa:
        Abandon and replace with a new nest
    Keep track of best solution
Output the best solution.
```

IMP Image Processing METRO

Page:
Date: 21/11/2024.

5. Grey Wolf Optimizer (GSO) :

Swarm Intelligence algorithm inspired by the social hierarchy and hunting behaviour of grey wolves.
Leadership structure alpha, beta, delta & omega wolves and their collaborative hunting strategies.
Effective for continuous optimization problems
Applications in Data analysis & Machine Learning
SVM, Image Processing.

1. Define the Problem by create a mathematical function to optimize
2. Initialise Parameters set number of wolves & iterations.
3. Initialise population of wolves with random positions.
4. Evaluate fitness of each wolf based on optimization function
5. Update positions of wolves based on position of alpha, beta, delta wolves
6. Iterate through each position until necessary criteria are met
7. Output the best solution found during the iterations.

Pseudo Code :

Initialise wolves' positions randomly in search space
Evaluate fitness of each wolf.
Sort wolves based on fitness to find alpha, beta, delta wolves
For each iteration
    Update position of each wolf based on alpha beta delta
    Update parameters/positions
    Evaluate fitness of updated wolves
    Update best solution if found
Return position of alpha wolf as best solution.

Ved.R
21/11/24

6. Parallel Cellular Algorithms and Programs:

Inspired by the functioning of biological cells that operate in a highly parallel and distributed manner.

Principles of cellular automata and parallel computing

Each cell represents a potential solution and interacts with its neighbours to update its state based on predefined rules.

Suitable for large scale optimization problems & implemented on parallel computing architeture.

- Define the problem. Create a math function to optimize
- Initialise parameters lik no. of cells, grid size, neighbourhood structure and no. of iterations
- Initialise population of cells with random positions
- Evaluate fitness of each cell based on optimization functio
- Update states of each cell based on its neighbours
- Iterate through each state for fixed iterations until convergence criteria met & Output the best solution

Pseudo Code:

```
Initialise Parameters Grid_Size, No_Cells, Iterations, Neighbours
Function Optimize (cell_position):
    Define mathematical function f.
    return f(cell_position)
For each cell in the grid:
    cell.position, grid[cell.x][cell.y] = random_state()
    cell.fitness = Optimize(cell.position)
Function UpdateCellState(cell, neighbours)
    best_neighbour = FindBestNeighbour(neighbours)
    cell.position = best_neighbour.position
    EvaluateFitness(cell)
```

```
Function FindBestNeighbour(neighbours):
    best_fitness = Infinity
    best_neighbour = null
    for each neighbour in neighbours:
        If neighbour.fitness < best_fitness:
            best_fitness = neighbour.fitness
            best_neighbour = neighbour.
    return best_neighbour.

For iteration = 1 to Iterations
    Parallel for each cell in the grid:
        neighbour = GetNeighours(cell, neighbours)
        UpdateCellState(cell, neighbours)
    best_cell = FindBestSolution(grid).

Function FindBestSolution(grid):
    best_fitness = Infinity
    best_cell = null
    for each cell in grid
        If cell_fitness < best_fitness:
            best_fitness = cell_fitness
            best_cell = cell
    return best_cell.
```

"Traffic Simulation."

Application of 0/1 Knapsack Problem

Each cell represent a potential solution which is a combination of items in the knapsack. 1 - included 0 - excluded. Fitness of solution will be the total value of the items in the knapsack provided total weight doesn't exceed capacity.

Items = [1, 0, 0, 1, 0, 1, 1, 0, 1, 0]

Fitness = 23

METRO
Page:
Date: 14/11/2024

7. Optimization via Gene Expression Algorithm:

GEA inspired by biological processes of gene expression in living organisms.

Solutions to optimization problems are encoded in a manner similar to genetic sequences.

Selection, crossover, mutation & gene expression to find optimal solutions.

Applications: Data Analysis & Machine Learning

- Define the Problem by creating a math function to optimise
- Initialise Parameters, population size, no/. of genes, mutation, crossover rate & no/. of generations
- Generate initial population & Evaluate fitness for each
- Select genetic sequences on their fitness
- Perform crossover b/w selected sequences to produce offspring.
- Mutation to offspring hence variability achieved
- Translate genetic sequences into functional solutions
- Repeat Selection, Mutation, Crossover, Gene expression processes until critiria is met.
- Output the best solution.

Pseudo Code:

```
Initialize Parameters Population_Size, Genes, Generations,
                                CrossOverRate, MutationRate
Define fitness function
Initialize Population each gene in random population
Evaluate fitness function for each in the population
Function Selection (population, fitness_value)
    if fitness_value(for i) > fitness_value(j)
    i. append (select)
```

(genes)
while next_generation not null
    parent1 = Selection (population, fitness value)
    parent2 = Selection (population, fitness_value)
    child = Crossover (parent1, parent2)
    child = Mutate (child)
    Add child to next genes

Output best_solution (max (fitness_value)) population
Best_value = max (fitness_values).