INDEX

Name Sharanabasappa T Sub. BIS

Std.: Div. CE ~~Roll No.~~ USN: 253

Telephone No. E-mail ID.

Blood Group. AB+ Birth Day.

Sr.No.	Title	Page No.	Sign. Remarks
01	Exploring algorithms		8
02	Genetic algorithm		9
03	Particle Swarm - Optimization		10
04	Ant Colony - Optimization		10
05	Cuckoo Search		8
06	Gray Wolf Optimization		10
07	Parallel Cellular Algorithm		10
08	Gene Expression Algorithm		

Genetic Algorithm for Optimization Problems

GAs are a powerful optimization technique inspired by the principle of natural selection

Population: Population of potential solutions (individuals) to the optimization problem.

Fitness Evaluation: Each individual is evaluated based on fitness function. How good the solution it is.

Selection: Individuals are selected based on their fitness. fittest individuals are having higher chance of being selected.

Crossover: Selected individuals combine their traits to produce new offspring.

Mutation: Random changes are introduced to some offspring to maintain diversity within the population and explore new solution.

Iteration: The process of selection, crossover and mutation is repeated for several generation

Termination: The algorithm is stoped after fixed number of generation or an acceptable solution is found.

Application

1. Engineering Design Optimization
2. Machine Learning and Feature Selection
3. Scheduling Problems.

Particle Swarm Optimization (PSO)

PSO is population based optimization technique inspired by the behaviour of birds and fish

Inspiration:- PSO mimics the social behaviour of birds flocking to find food. Each individual adjust their position based on its own experience and of its neighbour.

Particles:- Each particle represents a potential solution in the search space problem and its position and velocity.

Fitness Evaluation:- Each particles position is evaluated using fitness function to determine the quality of solution

Updating Velocity and Positions:- Particle update their velocity based on their best-position and best-position of the swarm. This allows them to explore the search space more effectively

Application

1) Neural Network Training
2) Robotics and path planning
3) Financial Modelling and Portfolio Optimization

Ant Colony Optimization (ACO)

ACO is the algorithm inspired by the foraging behaviour of ants, particularly in how they find shortest paths.

Problem Definition:- The TSP involves finding the shortest route that a set of cities and return back to the starting city.

Pheromone Trails:- Ants deposit pheromones on the paths they take, with shortest paths receiving more pheromones, that attracting more ants.

Heuristic Information: Additional Information (like reverse distance) can be used to select the path

Cuckoo Search (CS)

The algorithm effectly combines the exploration and exploitation technique to find the solutions for complex problems.

Inspiration:- CS is based on the reproductive strategy of certain cuckoo species that lay their eggs in the nests of their other birds. This strategy ensures that host birds raise the cuckoo eggs, increasing their chances of survival.

Levy Flight:- CS employs the levy flights, which are random walks where steps are drawn from a Levy distribution. This allows a algorithm to explore the search space broadly: enhancing its ability to escape local minima and find global optima.

GWO (Grey Wolf Optimizer)

The Grey optimizer is a swarm intelligence algorithm inspired by the social hierarchy and hunting behaviour of grey wolves. It uses a leadership structure consisting of alpha, beta, delta and omega wolves, where alpha wolves guide the search process and the others assist in refining the direction. GWO is effective for continious optimization problems and is used in various fields like engineering, data analysis, and machine learning. Its advantages include simplicity and efficiency in finding optimal solution.

Parallel Cellular Algorithms

Parallel Cellular Algorithms are inspired by the behaviour of biological cells, operating in a highly parallel and distributed manner. They utilize principles from cellular automata and parallel computing to efficiently tackle complex optimization problem. In this framework, each cell represents a potential solution and interacts with the neighbouring cells based on predefined rules, facilitating the diffusion of information across the grid. This interaction allows the algorithm to effectively explore the search space.

Gene Expression Algorithm (GEA)

GEA are inspired by the biological process of gene expression, where genetic information is translated from DNA to functional protein. In GEA, potential solutions for optimization of problems is encoded like genetic sequences. The algorithm evolves these solutions using processes such as selection, crossover, mutation of offspring, and gene expression to identify optimal or near-optimal solutions. GEA is particularly effective for complex optimization challenges across various fields, including engineering, data analysis, and machine learning.

Genetic Algorithm for Optimization Problems

1) Initilize population as 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 the fitness values (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 < cross_rate:

- alpha = random values between 0 and 1

offspring1 = alpha * P1 + (1-alpha) * P2

offspring2 = alpha * P2 + (1-alpha) * P1

ELSE:

offspring1 = P1

offspring2 = 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 the new offspring

8. TERMINATE after maximum generations or convergence

4. OUTPUT the best individual and fitness value

Sneha S R
-24/10/24

Particle Swarm Optimization

```
def PSO( num_particles, dim, lowerbound, upper_bound, max_iter,
         w=0.5, c1=1.5, c2=1.5):
    #initilize swarm (create particles)
    swarm = [Particle(dim, low_bound, upper_bound)
             for _ in range(num_particles)]
    global_best_position = swarm[0].best_position #set init best pos
    global_best_value = swarm[0].best_value # set init global best
                                              fitness val
    #Main Optimization loop (iterates through generations)
    for iterations in range(max_iter):
        for particle in swarm:
            particle.evaluate() # Eval fitness of each particle
            #update global best if particle has better fitness
            if particle.best_value < global_best_value:
                global_best_value = particle.best_value
                global_best_position = particle.best_position
    # update position and velocity of particle
    for particle in swarm:
        particle.update_velocity(global_best_position, w, c1, c2) #update vel
        particle.update_position(lower_bound, upper_bound) # update pos

    # print progress to monitor optimization
    print(f"iteration {iteration+1}/{max_iter}, Best_Fitness:
                                        {global_best_value}")
# Return the global solution and fitness value
                                    global_best_value
```

pseudocode for Particle Swarm Optimization

1. Initiize sworm of particles:
   - For each particle
     - searcat position within the search space bounds.
     - set random velocity within the search space bounds.
     - set initial best position to the current position
     - calculate fitness as personal best position (using the objective function)

2. Initiize global best:
   - set the global best position and fitness to the personal best of the first particle.

3. Set hyper parameter
   - inertia weight (w)
   - Cognitive constant (c1)
   - social constant (c2)
   - Number_of_iteration (max_iter):

4 For each iteration (from 1 to max_iter):

   a. For each particle in the swarm:
      - evaluate the fitness of the current position
      - if the fitness of the current position is better than the personal best:
      - update the personal best position and fitness.

b. for each particle in the swarm:

- Update velocity:

- velocity = w * previous velocity + c1 * random_factor1 * (Personal_best - position - current_position) + c2 * random_factor * (global_best_position - current_position)

b Update position:

- velocity = w * previous velocity + c1 * random_factor1 *

- position = current position + updated velocity.

- Ensure the new position is within the search space bounds

c. Update global best:

d. print progress (optional):

- print current iteration and the global best fitness value

After the final iteration:

- return the global best position and its corresponding fitness value.

Application

Neural Network Training

Support Vector Machine Hyperparameter Tuning

Image Registration

Portfolio Optimization

14-11-2024

Ant Colony Optimization

Initilize the parameters

num-ants ← 30

num-iterations ← 50

alpha ← 1.0

beta ← 2.0

rho ← 0.1

Q ← 100 # pheromone deposite (scalling factor)

cities ← [(x1,y1), (x2,y2), (x3,y3) ---- (xn,yn)]

Initilize the number of cities

num_cities = len(cities)

Calculating the Euclidean distance matrix:

distance_matrix[i][j] ← dist b/w city i & j for all i & j

Initilize the phermone matrix:

pheromone[i][j] ← 1/num_cities for all i,j

Function to calculate the tour length:

Function calculate_tour_length(tour):

length ← 0

For i ← 0 to length(tour) -1:

length ← length + dist_matrix[tour[i]][tour[i+1]]

length ← length + dist_matrix[tour[last]][tour[first]]

return length

Function select_next_city (current_city, visited, pheromone, dist_matrix):

Initilize total_pheromone = 0

Initilize probability list

For each city (from 0 to num_cities - 1):

If city is not in visited:

pheromone_val ← pheromone[current_city][city] raised to alpha

visibility_val ← (1/dist_matrix[current_city][city]) raised to beta

total_pheromone ← total_pheromone + pheromone_val * visibility_val

Add pheromone_val * visibility_val to probability list

Else

Add 0 to probability list

Normalize the probabilities

For each probability in probabilities list:

Probability ← Probability / total_pheromone

Choose next city based on probability_list using a random choice.

Return next_city

Main ACO Algorithm:

Function aco_tsp():

Initilize best_tour ← None

Initilize best_length ← infinity

For iteration ← 1 to num_iterations:

Initilize all_ants_tour as empty list

initilize all_ants_lengths as empty list

For each ant (from 1 to num_ants):

Initilize visited cities set as empty

Choose random city to start (current_city) and add to visited set

Initilize tour list with current city.

While length_visited set < num_cities:

next_city ← select_next_city (current_city, visited, pheromone, dist_matrix)

Add next_city to tour

Add next_city to visited

current_city ← next_city.

Calculation tour length using calculate_tour_length (tour)

Add the tour to all_ant_tours

Add the tour length to all_ants_lengths.

If the current tour length is better then best_length:

Update best_length and best_tour with current tour's length and tour

Pheromone update:

Evaporate pheromone on all the edeges:

· pheromone ← pheromone * (1-rho)

For each ant (from 1 to num_ants):

For each edege (from city i to city j) in the ant's tour:

pheromone[i][j] ← pheromone[j][j] + Q / tour_length

Return best_tour, best_length

Run the ACO algorithm:

best_tour, best_length ← aco_tsp()

Output

print the best_tour

print the best_length

Applications

(TSP) Travelling salesman problem)

(VRP) Vehical Routing problem)

Job Scheduling Problem)

Network design and Optimization)

14/11/24

Cuckoo Search:

Pseudocode:-

-> Initilize Nests:-

The function initilizes the nest with random solutions within the define bounds.

```
function InitializeNests(bounds, n_nests):
# bounds: list of tuples defining min & max values for
        each dimension.

#n_nest: Number of nests to create.
  nests: EmptyList()
  for i=1 to n_nests:
     nest = CreateRandomSolution(bounds)
     Append nest to nests.
  return nests.
```

-> Create Random Solution:-

This function creates a random solution within the specified bounds

```
function CreateRandomSolution(bounds):
# bounds:- Defining min & max values for each dimension
  Solution = EmptyList()
  for i=1 to length(bounds):
     value = RandomBetween(bounds[i][0], bounds[i][1])
     Append value to Solution
  return Solution
```

This function evaluates the fitness of the given objective function.

```
function EvaluateFitness(nests, func):
  # nests: List of solutions
  # func: Objective function to evaluate fitness
    fitness: EmptyList()

  for each nest in nests:
      fitness_value = func(nest)
      Append fitness_value to fitness

  return fitness
```

→ Generate New Solutions Using Levy Flights

Levy flights are used to create new solutions by introducing random perturbations that help explore the search space

```
function GenerateNewSolutions(nests):
# nests: Current population of nests (solutions)
  new_nests = EmptyList()
  for each nest in nests:
    new_nest = ApplyLevyFlight(nest)
    Append new_nest to new_nests

return new_nests.
```

→ Apply Levy Flight :-

This function generates a new solution by applying a

```
function ApplyLevyFlight(nest):
#nest: The current solution (nest)
  new_nest = copy(nest)
  for each dimension of the nest:
    #apply a Levy Flight step (random jump)
    random_step = RandomLevyStep()
    new_nest[dimension] += random_step.
  return new_nest

#The RandomLevyStep can be modeled using a
 distribution with heavy tails.
Function RandomLevyStep():
  Step = RandomNormal(0,1) * Abs(RandomNormal(0,1))
  return step.
```

→ Generate Random solution:

This function generates a random solution within the problem's bounds.

```
function GenerateRandomSolution(bounds):
  random_solution = EmptyList()
  for i=1 to length(bounds):
    Value = RandomBetween(bounds[i][0], bounds[i][1])
    Append value to random_solution
```

→ Cuckoo search (Main function)

```
Function CuckooSearch (func, bounds, n_nest, n_iterations)

#n_nests:- No. of nests
#n_iterations: No. of iterations
nests = Initialize_vests(bounds, n_nests)
fitness = EvaluateFitness (nests, func)
# Tracks the best solution found so far
best_idx = IndexofMin(fitness)
best_fitness = fitness[best_idx]
best_Solution = nests[best_indx]

for iteration = 1 to n_iterations:
    new_fitness = Evaluate fitness (new_nests, func)
    new_nests = GenerateNewSolutions(nests)
    for i=1 to n_nests:
        If Random() < pa:
            new_nests[i] = GenerateRandomSolution(bounds)
    for i=1 to n_nests:
        If new_fitness[i] < fitness[i]:
            nests[i] = new_nests[i]
            fitness[i] = new_fitness[i]

# track best solution
best_idx = IndexofMin(fitness)
If fitness[best_idx] < best_fitness:
    best_solution = nests[best_idx]
    best_fitness = fitness[best_idx]
    return best_solution, best_fitness
```

Applications

1) Machine Learning
2) Image and Signal processing
3) optimization in Robotics.

Sneha SB
28/11/24

Grey Wolf Optimizer (GWO):

→ Objective Function:

The function is used to define the sphere function, commonly used in optimization algorithm.

```
def objective-function(x):
    return np.sum(x**2)
```

→ GreyWolfOptimizer Class :-

We use the grayWolf class to initilize the various parameters.

```
def_init-(self, objective-function, n-wolves, max-iter, dim,
          lower-bound, upper-bound);

    self.obj-function = objective-function
    self.n-wolves = n-wolves
    self.max-iter = max-iter
    self.dim = dim
    self.lower-bound = lower-bound
    self.upper-bound = upper-bound
    self.alpha-position = np.zeros(dim)
    self.delta-position = np.zeros(dim)
    self.alpha-score = float("int")
    Self.delta-score = float("int")
    Self.wolves-position = pp.random.uniform(lower-bound,
                                             (upper.bound, n.wolves,
```

→ Update position :-

This function calculate and returns the new position of a wolf based on the positions of the alpha, beta, or delta wolves,

The position update is calculate as

$D = |C \cdot X_{best} - X_{wolf}|$

- C is a coefficient helps control the movement
- X_{best} is the best position of best wolf

The new position is updated as:

$X_{wolf}^{new} = X_{best} - A \cdot D$

and A is coefficient that helps control the size of the step.

```
def update_position(self, A, C, wolf_position, best_position):
    D = np.abs(C * best_position - wolf_position)
    return best_position - A * D
```

→ Optimize function

```
def optimize(self):
```

This function is used to the fitness calculation for each wolf, updating Alpha, Beta, delta wolves, coefficient calculation of A and C for each iteration, where a linearly decrease & A & C are used to compute movement of each wolf

```
def optimize(self):
    for t in range(self.max_iter):
        for i in range(self.n_wolves):
            fitness = self.obj_func(self.wolves_position[i])
            if fitness < self.alpha_score
                self.alpha_score = fitness
                self_alpha_position = self.wolves_position[i].copy()
            elif fitness < self.beta_score and fitness > self.alpha_score.
```

```
        self.beta_score = fitness
        self.beta_position = self.wolves_position[i].copy()
    elif fitness < self.delta_position = self.wolves_position[i].copy()

a = 2 - t * (2 / self.max_iter)
r1, r2, r3 = np.random.rand(3, self.n_wolves, self.dim)

A1 = 2 * a * r1 - a

A2 = 2 * a * r2 - a

A3 = 2 * a * r3 - a

C1 = 2 * r1

C2 = 2 * r2

C3 = 2 * r3

for i in range(self.n_wolves):
    self.wolves_position[i] = self.update_position(
            A1[i], C1[i], self.wolves_position[i],
            self.alpha_position)

    self.wolves_position[i] = self.update_position(A2[i],
            C2[i], self.wolves_position[i] self.
                beta_position)

    self.wolves_position[i] = self.update_position(A3[i], C3[i]
            self.wolves_position[i], self.delta_position

# to ensure wolves stay in the bounds.
    self.wolves_position[i] = np.clip(self.wolves_position[i],
                self.lower_bound, self.upper_bound

return self.alpha_position, self.alpha_score
```

Main Code (outside the class)

```
n_wolves = 50
max_iter = 100
dim = 2
lower_bound = -10
upper_bound = 10

gwo = GreyWolfOptimizer(objective_function, n_wolves, max_iter,
                        dim, lower_bound, upper_bound)

best_position, best_score = gwo.optimize()
#print the best_position & best_score.
```

Output

```
Best Solution found : [-4.03126745e-19
                       4.16286600e-19]

Objective function value: 3.35805755881874 2e-37
```

Applications

- Structural design
- Control System design
- Aerospace designing
- Machine learning
  - Hyperparameter Tuning
  - Feature Selection
- Robotics
  - Path planning
  - Control System for robotics.

28/11/24

Parallel Cellular Algorithm

⇒ Parallel cellular Algorithm often used in cellular automata, where a grid of cells is evolved through discrete time steps according to rules that dependence upon the states of neighbouring cells.

⇒ Parallelism is also achieved because the update of each cell can be computed independently and simulbaneously.

Core Components

1. Grid initialization :- Define the grid size and initilize the states of all the cells.

2. Neighborhood Definition :- Specify the neighbourhood of each cells

3. Update Rules :- Define the rules to determine the next state of each cell based on the its current state and neighbours.

4. Parallel Update :- Simultaniously compute the next state of all cells

5. Iteration : Repeate the update for a fixed number of steps or until a termination condition is met.

Pseudocode : Parallel Cellular Algorithm

Input:

Grid-size: dimensions of the grid

Initial-states: initial states of all the cells.

MAX-STEPS: number of times steps to simulate.

UPDATE-RULE(cell-state, neighbours): function defining cell evolution rules.

Output:

Objective function: sphere function
Procedure Parallel Cellular Algorithm:

1. Initilize the grid with INITIAL_STATE.

2. For step = 1 to max_steps:

   a. Create a new grid NEW_GRID to store Updated states.

   b. In parallel for each cell C[i,j] in the grid:

      i. Determine the NEIGHBORS of cells C[i,j]

      ii. Compute NEW_GRID[i][j] = UPDATE_RULE(GRID[i][j], NEIGHBORS)

   c. Upate GRID = NEW_GRID

3. Return GRID as FINAL_STATE

output:- Best solution : [-0.00095422 0.03240244]
Best Fitness :- 0.00125

Applications

1. Simulation of Natural Phenomena.

- Weather simulation:- Modeling a cloud formation, rain patterns, and wind dynamics.

- Forest fire modeling:- Simulating the spread of fire based on vegetation density, wind, and terrain.

2) Traffic Flow Simulation

- Road Traffic Modeling:- Simulating the Road flow on roads, incorporating factors like traffic lights, lane changes, and accidents.

3) Image processing

Edge detection:- Highlighting the boundaries of the objects in image

Noise Reduction:- Smoothing an image by applying the rules iteratively

Texture generation:- Creating a complex texture using local rules.

Gene Expression Algorithms

Gene expression algorithms are used to model the process of gene expression in biological system or sove problems using evolutionary algorithms inspired by this system.

1. Chromosome Encoding :- Represents the potential solution as Chromosomes. Each chromosome has a fixed length linear representation (genes). Which is later expressed as a tree-like structure (phenotype)

2. Initialization: Create an initial population of random chromosom

3. Fitness Evaluation :- Evaluate the fitness of each chromosom based on the how well they sove the problem.

4. Selection :- Select chromosomes based on their fitness Valu for reproduction.

5. Variation Operators :- Apply mutation, crossover and gene -transposition to produce offspring.

6. Expression :- Convert offspring from their linear structure representation to tree-like structure.

7. Termination :- Repeate the process until a stopping criteri is met.

Objective function: $C \sin 2 - A * \cos(2 * \pi * x)$

Pseudocode

```
function GeneExpressionAlgorithm()
    Initilize population P with random chromosomes
    for generation = 1 to max_generations do
      EvaluateFitness(P) then
      if CheckTermination(P) then
          return BestChromosome(P)
      end if
    // selection
    P_new = SelectParent(P)

    // Apply genetic operators.
    for i=1 to size(P) do
      if Random() < crossover_rate then
          Parent1, Parent2 = SelectTwo(P_new)
          Child1, Child2 = Crossover(Parent1, Parent2)
          AddToPopulation(P_new, Child1, Child2)
      end if
      if Random() < mutation_rate then
          individual = SelectOne(P_new)
          mutated = Mutate(individual)
          AddToPopulation(P_new, mutated)
      end if
      if Random() < transposition_rate then
          individual = selectOne(P_new)
          transposed = Transpose(individual)
          AddToPopulation(P_new, transposed)
      end if
end for
```

```
    //convert linear chromosomes to tree-like expressions.
    for each individual in P_new do
        Express (individual)
    end for
    P = P_new
end for
    return BestChromosome(P)
end function

function EvaluateFitness(P)
    for each chromosome in P do
        fitness = ProblemSpecEvaluation(chromosome)
        AssignFitness(chromosore, fitness)
    end for
end function

function Express(chromosome)
    //convert linear representation to a tree-like phenotype
    phenotype = Decode Chromosome(chromosome)
    AssignPhenotype(Chromosome, phenotype)
end function.
```

output:

Best solution :- (0.0265271 0.3184912 1.0011450712
0.0083775 2 -0.2494732 -0.451763)

Best Fitness: 21.444198

18/12/24