DATE PAGE

⇒ Genetic Algorithm for Optimization Problems

Genetic Algorithms (GA) are inspired by the process of natural selection and genetics, where the fittest individuals are selected for reproduction to produce the next generation

Implementation Steps:
1. Define the problem
2. Initialize parameters
3. Create initial population
4. Evaluate fitness
5. Selection
6. Crossover
7. Mutation
8. Iteration
9. Output the best solution

Applications:
1. Optimization
2. Scheduling applications
3. Machine learning
4. Traveling salesman problem
5. Neural networks

Algorithm:
1. Start
2. Create initial population
3. Calculate fitness score for each individual
4. Repeat
   4.1. selection

4.2. crossover
4.3. mutation
4.4. calculation of fitness score
until convergence is found
5. Choose the individual with highest fitness value
6. Stop

Output:
Enter the population size: 5
Enter the number of generations: 3
Enter the mutation rate (0 to 1): 0.1
Enter the crossover rate (0 to 1): 0.01
Optimal x: 0.8680898792401004
Best function value: 1.7316362025897157

DATE PAGE

⇒ Particle Swarm Optimization for Function Optimization

Particle Swarm Optimization (PSO) is a powerful meta-heuristic optimization algorithm and, inspired by swarm behaviours observed in nature such as fish and bird schooling.

Implementation Steps:
1. Define the problem
2. Initialize parameters
3. Initialize particles
4. Evaluate fitness
5. Update velocities and positions
6. Iterate
7. Output the best solution

Applications:
1. Intelligent diagnosis
2. Agriculture monitoring
3. Economic dispatch problem
4. Smart grids and microgrid
5. Wild vegetation monitoring

~~Alg~~

P.T.O →

DATE PAGE

Algorithm:

Step 1: Randomly initialize swarm population of N particles X_i $(i = 1, 2, \ldots N)$

Step 2: Select hyperparameter values w, c1 and c2

Step 3: For I to in range (max_iter):

For i in range (N):

a. Compute new velocity of ith particle

```
swarm[i].velocity =
    w * swarm[i].velocity +
    r1 * c1 * (swarm[i].bestPos -
    swarm[i].position) +
    r2 * c2 * (best_pos - swarm-
    swarm[i].position)
```

b. Compute new position of ith particle using its new velocity

```
swarm[i].position += swarm[i].velocity
```

c. If position is not in range [minx, maxx] then clip it

```
if swarm[i].position < minx:
    swarm[i].position = minx
elif swarm[i].position > maxx:
    swarm[i].position = maxx
```

d. Update new best of this particle and new best of swarm

if swa Insensitive to scaling of disign Variables. $s_m[i]$.fitness < swarm[i].bestFitness:

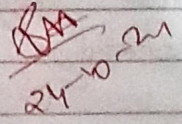
```
swarm[i].bestFitness =
```

```
        swarm[i].fitness
        swarm[i].bestPos = swarm[i].position
    if swarm[i].fitness < best_fitness_swarm
        best_fitness_swarm = swarm[i].fitness
        best_pos_swarm = swarm[i].position
    End for
End for
```

Step 4: Return best particle of Swarm

Output:

```
Enter the number of particles: 3
Enter the number of dimensions: 2
Enter the number of iterations: 5
Enter inertia weight (w): 23
Enter cognitive coefficient (c1): 21
Enter social coefficient (c2): 23
Enter minimum boundary for positions: 45
Enter maximum boundary for positions: 56
Best Position: [46.19899533  54.79269823]
Best Value: 5150.773900413809
```



DATE PAGE

⇒ Ant Colony Optimization for the Traveling Salesman Problem

Ant Colony Optimization (ACO) is a population-based meta-heuristic for combinatorial optimization problems. It is inspired by the ability of ants to find the shortest path between their nest and a source of food.

Implementation Steps:
1. Define the problem
2. Initialize parameters
3. Construct solution
4. Update pheromone
5. Iterate
6. Output the best solution

Applications:
1. Logistics and supply chain management
2. Transportation networks
3. Telecommunications and computer networks
4. Manufacturing and robotics
5. Computer-Aided Design (CAD)
6. Airline scheduling

Algorithm

initialize pheromone values $\tau_{ij} \in [1, n] : \tau_{ij} \to \tau_0$

for each ant $l \in \{1, \dots, m\}$ do
  initialize solution set $S \to \{1, \dots, n\}$
  randomly choose starting city $i_0 \in S$ for ant l
  move to starting city $i \to i_0$
  while $S \neq \emptyset$ do
    remove current city from selection set
    $S \to S \setminus [i]$
    choose next city j in tour with
    probability $p_{ij} = \frac{\tau_{ij}^{\alpha} \cdot \eta_{ij}^{\beta}}{\sum_{h \in S} \tau_{ih}^{\alpha} \cdot \eta_{ij}^{\beta}}$

    update solution vector $\pi_l(i) \to j$
    move to new city $i \to j$
  end while
  finalize ~~solution~~ vector $\pi_l(i) \to i_0$
end for
for each solution π_l, $l \in \{1, \dots, m\}$ do
  calculate tourlength $f(\pi_l) \to \sum_{i=1}^{n} d_{i \pi_l(i)}$
end for
for all (i,j) do
  evaporate pheromone $\tau_{ij} \to (1-\rho) \cdot \tau_{ij}$
end for
determine best solution of iteration
$\pi^+ = \arg\min_{l \in [1,m]} f(\pi_l)$

if π^+ better than current best π^*,
i.e. $f(\pi^+) < f(\pi^*)$ then
  set $\pi^* \to \pi^+$

end if
for all $(i,j) \in \pi^*$ do
    update $\tau_{ij} \rightarrow \tau_{ij} * \Delta/2$
end for
for all $(i,j) \in \pi^*$ do
    update $\tau_{ij} \rightarrow \tau_{ij} * \Delta/2$
end for
until condition for termination met

Output:
Enter the number of cities: 4
Enter the distance matrix for 4 cities.
Each row should contain the distances to other cities (eg. 0 for the distance to itself).
Enter distances from city 1: 0 2 4 6
Enter distances from city 2: 1 0 3 5
Enter distances from city 3: 3 6 0 9
Enter distances from city 4: 5 10 15 0
Best tour: [3, 0, 1, 2]
Best tour length: 19.0

AM
2-11-24

DATE: PAGE:

⇒ Cuckoo Search (CS)

The Cuckoo Search (CS) algorithm is a powerful metaheuristic inspired by the brood ~~parasitism~~ parasitism of cuckoos, which lay their eggs in the nests of other birds.

Implementation Steps:
1. Define the problem
2. Initialize parameters
3. Initialize population
4. Evaluate fitness
5. Generate new solutions
6. Abandon worst ~~flights~~ nests
7. Iterate
8. Output the best solution

Applications:
1. Cost and vulnerability optimization in the cloud
2. Antenna array design
3. Feature selection
4. Optimal Design of a PID (Proportional-Integral-Derivative) controllers ~~are widely used~~ for simple control system

DATE: PAGE:

Algorithm:

algorithm CuckooSearch():
// Input
// n = initial population size
// Pa = fraction of worse nests to be abandoned and replaced
// Max iterations = the maximum number of iterations
// f = the objective function to optimize
// Output
// the best solution found
Generate the initial population of n host nests X_i ($i = 1, 2, \ldots, n$)
while $t <$ Max iterations:
    Get a cuckoo randomly by Levy flights
    Evaluate its quality fitness F_i
    j ← choose a nest among n randomly
    if $F_i > F_j$:
        replace j by the new solution
    if # fraction (Pa) of worse nests are abandoned and new ones are built:
        Keep the best solutions
        rank the solutions and find the current best
Postprocess results and visualization

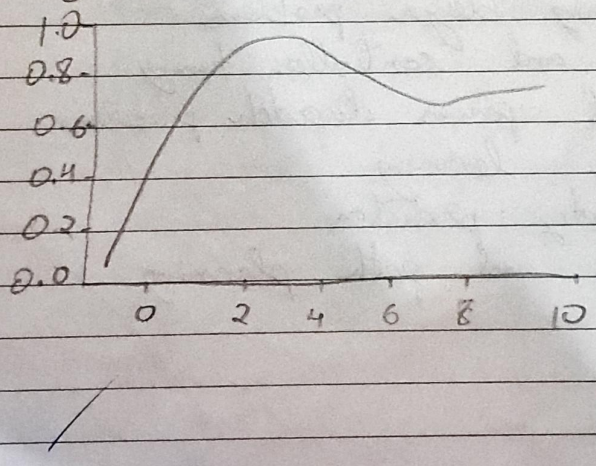
DATE: PAGE:

Output:

```
Enter the parameters for Cuckoo Search Algorithm:
Enter the population size (n): 6
Enter the fraction of worse nests to
abandon (Pa): 0.2
Enter the maximum number of
iterations (Max iterations): 5
Iteration 1, Best ITAE: 218.68974666636399
:
Iteration 5, Best ITAE: 92.50929069551009

Best PID Parameters Found (Kp, Ki, Kd):
[-1.98857 69  -3.36543246  -1.39493233]
Best ITAE Value: 92.50929069551009
```

Step Response of the closed loop system with Optimized PID



14-11-24

O/P seen

DATE: PAGE:

⇒ Grey Wolf Optimizer (GWO)

Grey Wolf Optimizer (GWO) is a population-based meta-heuristics algorithm that simulates the leadership hierarchy and hunting mechanism of grey wolves in nature.

Implementation Steps:
1. Define the problem
2. Initialize parameters
3. Initialize population
4. Evaluate fitness
5. Update positions
6. Iterate
7. Output the best solution

Applications:
1. Engineering design problems
2. Designs and controllers tuning
3. Optimal power dispatch problems
4. Machine learning
5. Bankruptcy prediction
6. Robotic and path planning

DATE: PAGE:

Algorithm:

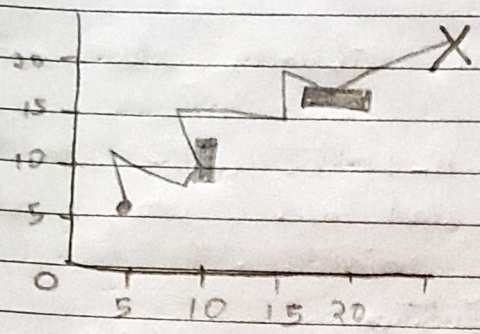
~~Step 1 Objective Function~~
~~Input —~~

1. Initialize
   Define grid, obstacles, start and goal positions
   Initialize n-wolves with random paths between start and goal, adding random noise
2. Fitness Function
   For each wolf, calculate path length and penalize for collisions with obstacles
3. Optimization
   For each iteration:
      Evaluate fitness of each wolf
      Identify alpha, beta, and delta wolves
      Update all wolves paths using the alpha, beta, and delta wolves positions with exploration and exploitation
      Clip wolves positions to stay within bounds
4. Return Best Path
   Output the path of the alpha wolf after optimization and plot it

DATE: PAGE:

Output:

Enter start x position : 2
Enter start y position : 3
Enter goal x position : 17
Enter goal y position : 18



Best Path Fitness (Path length): 21.912134352

21/4

DATE: PAGE:

⇒ Parallel Cellular Algorithms and Programs

Parallel Cellular Algorithms are inspired by the functioning of biological cells that operate in a highly parallel and distributed manner. These algorithms leverage the principles of cellular automata and parallel computing to solve complex optimization problems efficiently.

Implementation Steps:
1. Define the problem
2. Initialize parameters
3. Initialize population
4. Evaluate fitness
5. Update states
6. Iterate
7. Output the best solution

Applications:
1. Combinatorial and continuous optimization problems
2. Machine learning and hyperparameter tuning
3. Bioinformatics and computational biology
4. Environmental modeling and simulation
5. Image processing and computer vision
6. Pattern matching

DATE: PAGE:

Algorithm:

```
algorithm optimize (fitness_function, grid_size,
iterations, solution_range):

    grid = [random.uniform(*solution_range) for
            _ in range (grid_size)]
    best_solution, best_fitness = None, float('inf')

    for iteration in range (iterations):
        fitness_values = [fitness_function(x) for
        x in grid]
    for i in range (grid_size)
    grid = [ min ([grid[(i-1)%. grid_size],
                   grid[(i+1)%. grid_size]],
             key = fitness_function)

    if fitness_function(grid[i]) > fitness_function(
    grid[(i-1)%. grid_size]) or
    fitness_function(grid[i]) > fitness_function
    (grid [(i+1)%. grid_size])

    else
        grid[i]

    current_best_fitness = min(fitness_values)
    current_best_solution = grid[fitness_values.
            index (current_best_fitness)]

    if current_best_fitness < best_fitness
    best_solution, best_fitness =
    current_best_solution, current_best_fitness
```

return best_solution, best_fitness

Output:
Enter the target pattern to search for: HI
Enter the search space string: HEYHELLOHI
Iteration 0: Best fitness = 0

Iteration 90: Best fitness = 0
Best solution found: HI
Best fitness (number of mismatches): 0

DATE: PAGE:

⇒ Optimization via Gene Expression Algorithms

Gene Expression Algorithms (GEA) are inspired by the biological process of gene expression in living organisms. This process involves the translation of genetic information encoded in DNA into functional proteins.

Implementation Steps:
1. Define the problem
2. Initialize parameters
3. Initialize population
4. Evaluate fitness
5. Selection
6. Crossover
7. Mutation
8. Gene expression
9. Iterate
10. Output the best solution

Applications:
1. Machine learning and data mining
2. Data clustering and classification
3. Optimization in manufacturing
4. Healthcare and medical applications
5. Smart grids and energy distribution

DATE: PAGE:

Algorithm:

```
initialize_parameters(pop_size, num_genes,
mutation_rate, crossover_rate, max_generations,
fitness_threshold, stagnation_limit)
population = initialize_population(pop_size,
num_genes)
generation = 0
best_fitness = -infinity
stagnation_counter = 0

while generation < max_generations and
best_fitness < fitness_threshold:
    fitness_values = evaluate_fitness(population)
    parents = selection(population, fitness_values)
    offspring = crossover(parents, crossover_rate)
    mutated_offspring = mutate(offspring, mutation_rate)
    phenotypic_solutions = gene_expression(
                            mutated_offspring)
    fitness_values_new = evaluate_fitness(phenotypic_
                            solutions)

    population = update_population(population,
    phenotypic_solutions, fitness_values, fitness_values_new)

    current_best_solution = get_best_solution(population,
    fitness_values_new)
    if current_best_solution.fitness > best_fitness:
        best_solution = current_best_solution
        best_fitness = current_best_solution.fitness
        stagnation_counter = 0
    else:
```

```
        stagnation_counter += 1

        if stagnation_counter >= stagnation_limit:
            break

        generation += 1

    return best_solution
```

Output:

```
Enter total energy demand (in kWh): 200
Enter solar energy capacity (in kWh): 100
Enter conventional power plant capacity (in kWh): 100
Enter cost per kWh of solar energy ($): 50
Enter cost per kWh of conventional energy ($): 30

Best solution found:
Solar energy generated: 99.00 kWh
Conventional energy generated: 96.87 kWh
Total energy generated: 195.87 kWh
Total cost: $9983.45
```