LAB - 04 (20-11-24)

Cuckoo Search (CS):

Cuckoo Search (CS) is a nature-inspired optimization algorithm based on the brood parasitism of some cuckoo species. This behavior involves laying eggs in the nests of other birds, leading to the optimization of survival strategies. CS uses Lévy flights to generate new solutions, promoting global search capabilities and avoiding local minima. The algorithm is widely used for solving continuous optimization problems and has applications in various domains, including engineering design, machine learning, and data mining.

Implementation Steps:

1.Define the Problem: Create a mathematical function to optimize.
2. Initialize Parameters: Set the number of nests, the probability of discovery, and the number of iterations.
3. Initialize Population: Generate an initial population of nests with random positions.
4. Evaluate Fitness: Evaluate the fitness of each nest based on the optimization function.
5.Generate New Solutions: Create new solutions via Lévy flights.
6.Abandon Worst Nests: Abandon a fraction of the worst nests and replace them with new random positions.
7. Iterate: Repeat the evaluation, updating, and replacement process for a fixed number of iterations or until convergence criteria are met.
8.Output the Best Solution: Track and output the best solution found during the Iterations.

ALGORITHM / LOGIC-

MATHEMATICAL MODEL:

1. Encircling the prey →

(1) → $\vec{D} = |\vec{C} \cdot \vec{X_p}(t) - \vec{X_p}(t)|$

(2) → $\vec{X}(t+1) = \vec{X_p}(t) - \vec{A} \cdot \vec{D}$

$\vec{A} = 2\vec{a}\vec{r_1} - \vec{a}$

$\vec{C} = 2\vec{r_2}$

where t is current iteration, $\vec{A}$ and $\vec{C}$ are coefficient vectors, $\vec{X_P}$ is position of the prey and $\vec{X}$ indicates position of the wolf. $\vec{a}$ is linearly decreased from 2 to 0 and $\vec{r_1}$ and $\vec{r_2}$ are random vectors in [0, 1]

2. Hunting →

$D_\alpha = |C_1 \cdot X_\alpha - X|$

$D_\beta = |C_2 \cdot X_\beta - X|$

$D_\delta = |C_3 \cdot X_\delta - X|$

$X_1 = X_\alpha - A_1 \cdot (D_\alpha)$

$X_2 = X_\beta - A_2 \cdot (D_\beta)$

$X_3 = X_\delta - A_3 \cdot (D_\delta)$

$$X(t+1) = \frac{X_1 + X_2 + X_3}{3}$$

3. Attacking (Exploitation) →

- $\vec{A}$ is a randome value in [-2a, 2a] where a is decreased from 2 to 0 over course of iterations
- When $|A| < 1$, wolves attack prey, representing exploitation process.

PURPOSE:
GWO (Grey Wolf Optimization) is inspired by grey wolves and mimics the leadership heirarchy and hunting mechanism of grey wolves. The heirarchy is distributed between alpha (α) which is the top most and most powerful, beta (β) second-best, follows alpha; delta (δ) third best, divided in categories of elders, sentinel, scouts etc; and lastly omega (ω) that are least important. In this model three main steps of hunting, searching, encircling and attacking prey are implemented.

APPLICATION:
1. GWO for classical engineering problems →
GWO is used in three constrained engineering design problems: tension/compression spring, welded beam, pressure vessel designs are employed. These problems have several equality and inequality constraints.

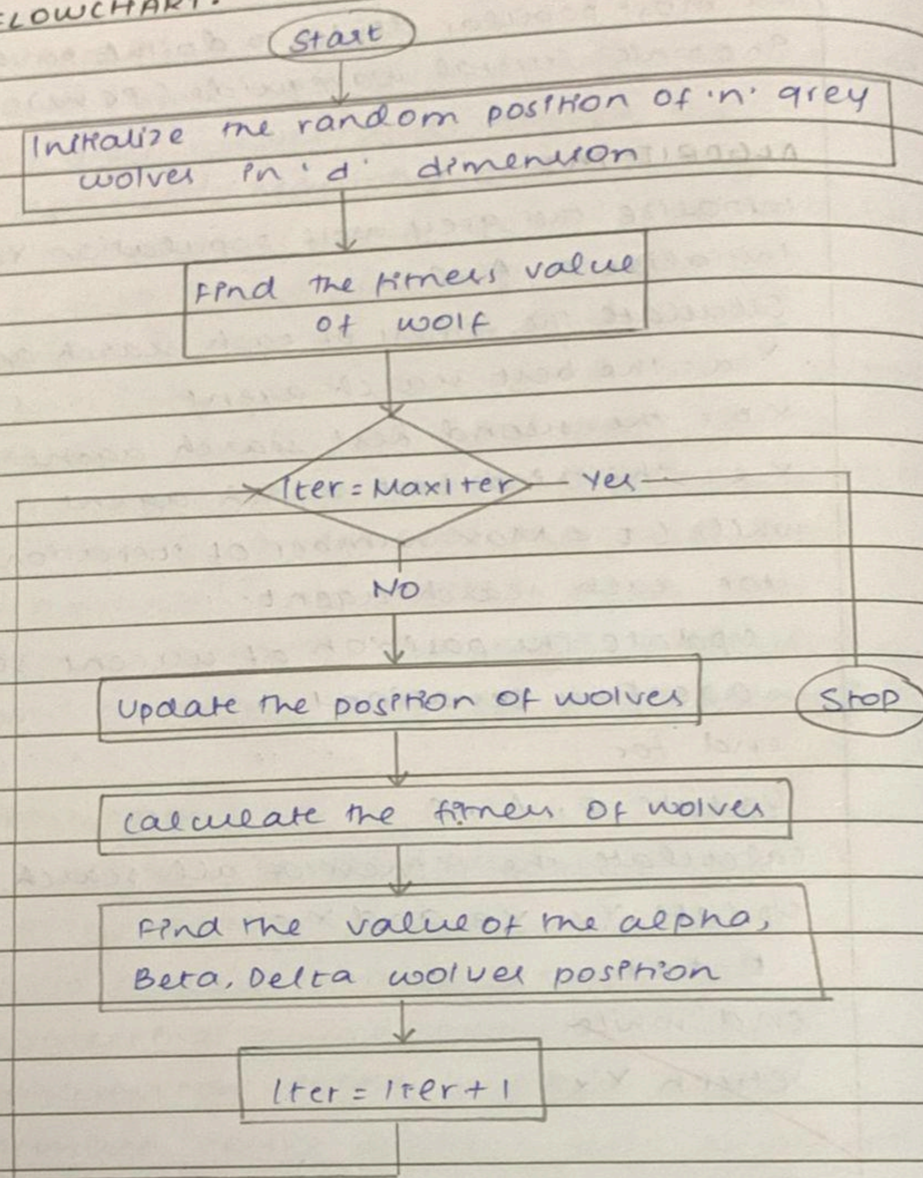
2. GWO in optical engineering →
GWO is used in optical buffer design. Optical buffer is one of the main components of optical CPUs. Optical buffer slows group velocity of light and allows optical CPUs to process optical packets or adjust its timing.

The most popular device to do this is a Photonic crystal waveguide (PCW).

ALGORITHM:

Initialize the grey wolf population X_i $(i=1,2,..n)$
Initialize a, A, C
Calculate the fitness of each search agent
X_α = the best search agent
X_β = the second best search agent
X_δ = the third best search agent
while (t < Max number of iterations)
for each search agent
Update the position of current search agent by equations
end for
Update a, A, C
calculate the fitness of all search agents
Update X_α, X_β and X_δ
t = t+1
end while
return X_α.

FLOWCHART:



IMPLEMENTATION:

The Grey wolf optimization problem is implemented using previously mentioned mathematical model in a python program.

OUTPUT :

Iteration 1/50 , Alpha score : 4.8885121

Iteration 2/50 , Alpha score : 4.8885121

Iteration 3/50 , Alpha score : 4.8885121

Iteration 4/50 , Alpha score : 2.825963

Iteration 5/50 , Alpha score : 2.252555

⋮

Iteration 50/50 , Alpha score : 0.00018417

Best Position (Alpha) : [0.0023 -0.0059 0.0056 -0.0028 -0.0027 0.0022]

Best score (Alpha) = 7.555845.. e-05

Best Position (Beta) : [0.0023 -0.0059 0.0056 -0.0028 -0.0027 0.0023]

Best score (Beta) : 7.97923.. e-0.5

Best Position (Delta) : [0.0023 -0.0059 0.0056 -0.0028 -0.0027 0.0023]

Best score (Delta) : 7.98789.. e-05

20/11/24

INPUT-

```
import numpy as np
import math
# Objective Function (e.g., Sphere Function)
def objective_function(x):
    return np.sum(x**2)
# Lévy flight implementation
def levy_flight(Lambda, dim):
    sigma = (math.gamma(1 + Lambda) * math.sin(math.pi * Lambda / 2) /
             (math.gamma((1 + Lambda) / 2) * Lambda * 2**((Lambda - 1) / 2)))**(1 / Lambda)
    u = np.random.normal(0, sigma, size=dim)
    v = np.random.normal(0, 1, size=dim)
    step = u / abs(v)**(1 / Lambda)
    return step
def cuckoo_search(num_nests, dim, lower_bound, upper_bound, max_gen, pa):
    nests = np.random.uniform(lower_bound, upper_bound, (num_nests, dim))
    fitness = np.apply_along_axis(objective_function, 1, nests)
    best_nest = nests[np.argmin(fitness)]
    best_fitness = np.min(fitness)
    history = [best_fitness]
    for gen in range(max_gen):
        new_nests = nests.copy()
        for i in range(num_nests):
            step = levy_flight(1.5, dim)
            new_solution = nests[i] + step * np.random.uniform(-1, 1, size=dim)
            new_solution = np.clip(new_solution, lower_bound, upper_bound)
            new_fitness = objective_function(new_solution)
            if new_fitness < fitness[i]:  # Replace if new solution is better
                new_nests[i] = new_solution
                fitness[i] = new_fitness
        abandon = np.random.rand(num_nests) < pa
        for i in range(num_nests):
            if abandon[i]:
                new_nests[i] = np.random.uniform(lower_bound, upper_bound, dim)
                fitness[i] = objective_function(new_nests[i])
        best_nest = new_nests[np.argmin(fitness)]
        best_fitness = np.min(fitness)
        history.append(best_fitness)
        nests = new_nests
        print(f"Generation {gen + 1}: Best Fitness = {best_fitness}")
```

```python
    return best_nest, best_fitness, history
num_nests = 25
dim = 5
lower_bound = -10
upper_bound = 10
max_gen = 50
pa = 0.25
best_solution, best_fitness, history = cuckoo_search(num_nests, dim, lower_bound,
upper_bound, max_gen, pa)
print("\nOptimal Solution Found:")
print("Best Solution:", best_solution)
print("Best Fitness:", best_fitness)
```

OUTPUT-

```
Generation 1: Best Fitness = 81.4345727383787
Generation 2: Best Fitness = 70.86126465228396
Generation 3: Best Fitness = 97.18961167672038
Generation 4: Best Fitness = 63.8153902492081
Generation 5: Best Fitness = 63.8153902492081
Generation 6: Best Fitness = 63.8153902492081
Generation 7: Best Fitness = 70.00307044133422
Generation 8: Best Fitness = 60.880738509269406
Generation 9: Best Fitness = 58.93642237249307
Generation 10: Best Fitness = 43.54676034829737
Generation 11: Best Fitness = 26.29979776534075
Generation 12: Best Fitness = 19.214803244954275
Generation 13: Best Fitness = 19.214803244954275
Generation 14: Best Fitness = 19.214803244954275
Generation 15: Best Fitness = 32.96461698828331
Generation 16: Best Fitness = 32.96461698828331
Generation 17: Best Fitness = 43.21564365053212
Generation 18: Best Fitness = 43.21564365053212
Generation 19: Best Fitness = 43.21564365053212
Generation 20: Best Fitness = 43.21564365053212
Generation 21: Best Fitness = 43.21564365053212
Generation 22: Best Fitness = 21.11850431057947
Generation 23: Best Fitness = 21.11850431057947
Generation 24: Best Fitness = 60.49728118293927
Generation 25: Best Fitness = 48.656934512915846
Generation 26: Best Fitness = 35.650538139076794
Generation 27: Best Fitness = 35.650538139076794
Generation 28: Best Fitness = 35.650538139076794
Generation 29: Best Fitness = 29.957059520515074
Generation 30: Best Fitness = 25.00704597380304
Generation 31: Best Fitness = 42.02364933860991
Generation 32: Best Fitness = 42.02364933860991
Generation 33: Best Fitness = 40.90119673417235
Generation 34: Best Fitness = 40.90119673417235
Generation 35: Best Fitness = 40.90119673417235
Generation 36: Best Fitness = 54.59431880880369
Generation 37: Best Fitness = 62.480562263285975
Generation 38: Best Fitness = 83.49104623355358
Generation 39: Best Fitness = 53.37095015242599
Generation 40: Best Fitness = 53.37095015242599
Generation 41: Best Fitness = 52.668397081275046
Generation 42: Best Fitness = 52.668397081275046
Generation 43: Best Fitness = 52.668397081275046
Generation 44: Best Fitness = 44.77865762253813
Generation 45: Best Fitness = 42.56559431773917
Generation 46: Best Fitness = 37.39149563901286
Generation 47: Best Fitness = 37.39149563901286
Generation 48: Best Fitness = 35.026362424913934
Generation 49: Best Fitness = 25.22333083293221
Generation 50: Best Fitness = 23.790985164565125

Optimal Solution Found:
Best Solution: [ 2.38882567  1.40045001  3.82181146 -0.97300359 -0.75515435]
Best Fitness: 23.790985164565125
```