Name: **Kunal Sinha** Batch: Swarm **R1- G2** Roll no: **2K17/CO/164**

Experiment 2

Aim: Write a program to implement Cuckoo Search Optimization algorithm. .

Theory:

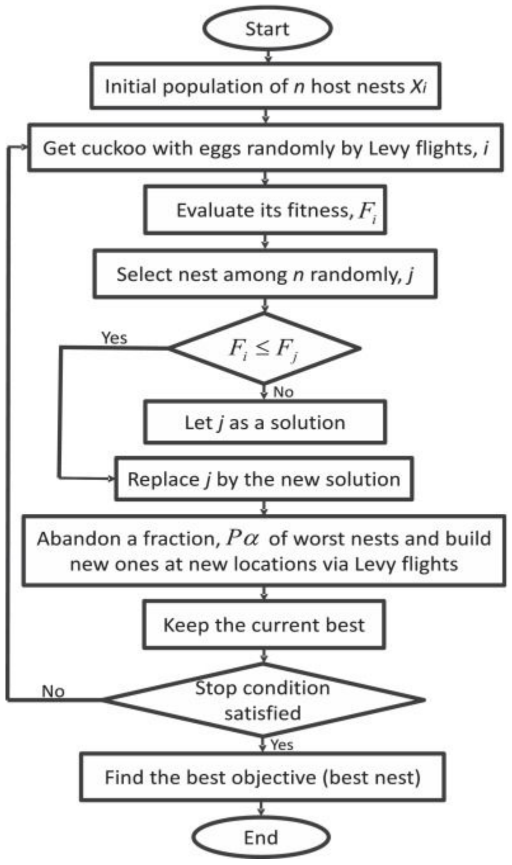
Cuckoo Search (CS) is a meta-heuristic algorithm based on the breeding pattern of certain species of cuckoo birds. In our research, we have implemented CS for the NP-hard optimization problem, the Traveling Salesman Problem (TSP). CS is based on three idealized rules:

1. Each cuckoo lays one egg at a time, and dumps its egg in a randomly chosen nest;
2. The best nests with high quality of eggs will carry over to the next generation;
3. The number of available hosts nests is fixed, and the egg laid by a cuckoo is discovered by the host bird with a probability pa $\in$ (0,1).

Discovering operate on some set of worst nests, and discovered solutions dumped from farther calculations.The algorithm can be extended to more complicated cases in which each nest has multiple eggs representing a set of solutions.

Algorithm:

Input: Input is the distance between cities given in form of a matrix. (distanceMatrix in code).


Start
Initial population of n host nests X_i
Get cuckoo with eggs randomly by Levy flights, i
Evaluate its fitness, F_i
Select nest among n randomly, j
$F_i \le F_j$
Yes
No
Let j as a solution
Replace j by the new solution
Abandon a fraction, $P\alpha$ of worst nests and build new ones at new locations via Levy flights
Keep the current best
Stop condition satisfied
No
Yes
Find the best objective (best nest)
End

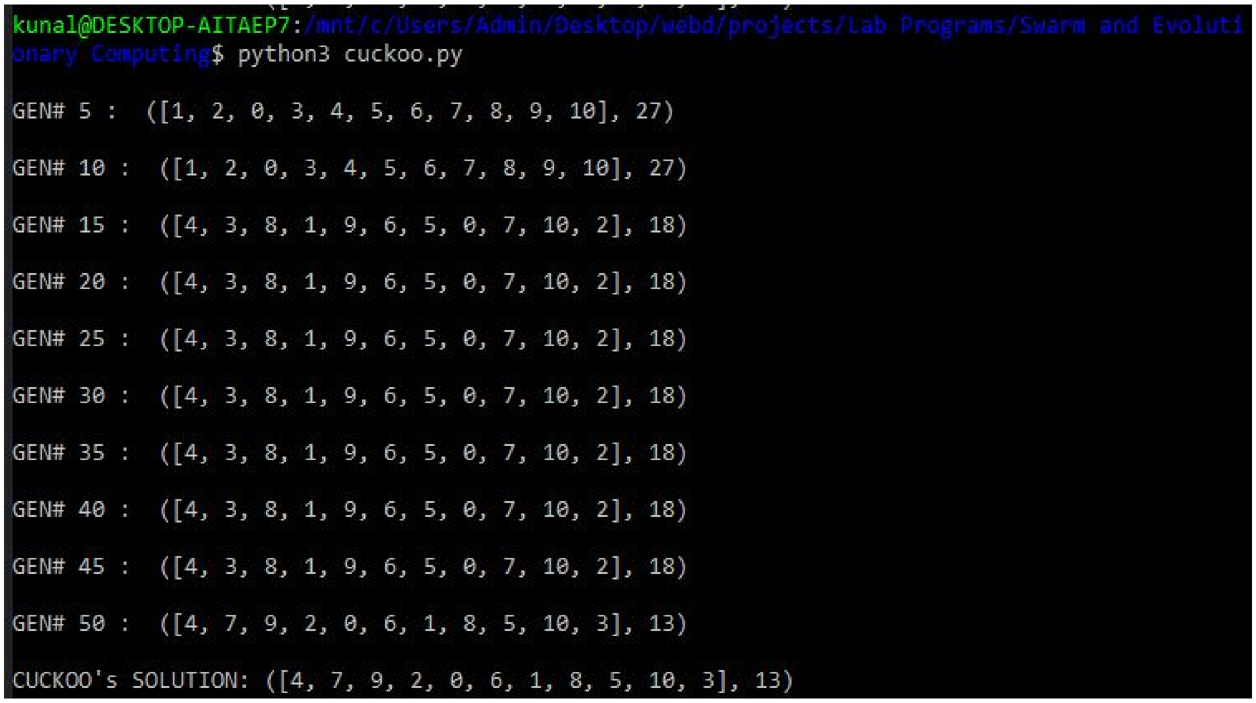
Source Code:

cuckoo.py

```
from random import uniform
from random import randint
import math
distanceMatrix = [
                [0, 29, 20, 21, 16, 31, 100, 12, 4, 31, 18],
                [29, 0, 15, 29, 28, 40, 72, 21, 29, 41, 12],
                [20, 15, 0, 15, 14, 25, 81, 9, 23, 27, 13],
                [21, 29, 15, 0, 4, 12, 92, 12, 25, 13, 25],
                [16, 28, 14, 4, 0, 16, 94, 9, 20, 16, 22],
                [31, 40, 25, 12, 16, 0, 95, 24, 36, 3, 37],
                [100, 72, 81, 92, 94, 95, 0, 90, 101, 99, 84],
                [12, 21, 9, 12, 9, 24, 90, 0, 15, 25, 13],
                [4, 29, 23, 25, 20, 36, 101, 15, 0, 35, 18],
                [31, 41, 27, 13, 16, 3, 99, 25, 35, 0, 38],
                [18, 12, 13, 25, 22, 37, 84, 13, 18, 38, 0]             ]

def levyFlight(u):
   return math.pow(u, -1.0/3.0)

def randF():
   return uniform(0.0001, 0.9999)

def calculateDistance(path):
   index = path[0]
   distance = 0
   for nextIndex in path[1:]:
      distance += distanceMatrix[index][nextIndex]
      index = nextIndex
      return distance+distanceMatrix[path[-1]][path[0]]

def swap(sequence, i, j):
   temp = sequence[i]
   sequence[i] = sequence[j]
   sequence[j] = temp

def twoOptMove(nest, a, c):
   nest = nest[0][:]
   swap(nest, a, c)
```

```
    return (nest, calculateDistance(nest))

def doubleBridgeMove(nest, a, b, c, d):
    nest = nest[0][:]
    swap(nest, a, b)
    swap(nest, b, d)
    return (nest, calculateDistance(nest))

numNests = 10
pa = int(0.2*numNests)
pc = int(0.6*numNests)
maxGen = 50
n = len(distanceMatrix)
nests = []
initPath = list(range(0, n))
index = 0
for i in range(numNests):
    if index == n-1:
        index = 0
    swap(initPath, index, index+1)
    index += 1
    nests.append((initPath[:], calculateDistance(initPath)))
nests.sort(key=lambda tup: tup[1])
for t in range(maxGen):
    cuckooNest = nests[randint(0, pc)]
    if(levyFlight(randF()) > 2):
        cuckooNest = doubleBridgeMove(cuckooNest, randint(0, n-1), randint(0, n-1), randint(0,
                        n-1), randint(0, n-1))
    else:
        cuckooNest = twoOptMove(cuckooNest, randint(0, n-1), randint(0, n-1))
    randomNestIndex = randint(0, numNests-1)
    if(nests[randomNestIndex][1] > cuckooNest[1]):
       nests[randomNestIndex] = cuckooNest
    for i in range(numNests-pa, numNests):
        nests[i] = twoOptMove(nests[i], randint(0, n-1), randint(0, n-1))
    nests.sort(key=lambda tup: tup[1])
    if (t+1) % 5 == 0:
        print("\nGEN#", t+1, ": ", nests[0])
print("\nCUCKOO's SOLUTION", end=': ')
print(nests[0])
```

Output:



```
kunal@DESKTOP-AITAEP7:/mnt/c/Users/Admin/Desktop/webd/projects/Lab Programs/Swarm and Evoluti
onary Computing$ python3 cuckoo.py

GEN# 5 :  ([1, 2, 0, 3, 4, 5, 6, 7, 8, 9, 10], 27)

GEN# 10 :  ([1, 2, 0, 3, 4, 5, 6, 7, 8, 9, 10], 27)

GEN# 15 :  ([4, 3, 8, 1, 9, 6, 5, 0, 7, 10, 2], 18)

GEN# 20 :  ([4, 3, 8, 1, 9, 6, 5, 0, 7, 10, 2], 18)

GEN# 25 :  ([4, 3, 8, 1, 9, 6, 5, 0, 7, 10, 2], 18)

GEN# 30 :  ([4, 3, 8, 1, 9, 6, 5, 0, 7, 10, 2], 18)

GEN# 35 :  ([4, 3, 8, 1, 9, 6, 5, 0, 7, 10, 2], 18)

GEN# 40 :  ([4, 3, 8, 1, 9, 6, 5, 0, 7, 10, 2], 18)

GEN# 45 :  ([4, 3, 8, 1, 9, 6, 5, 0, 7, 10, 2], 18)

GEN# 50 :  ([4, 7, 9, 2, 0, 6, 1, 8, 5, 10, 3], 13)

CUCKOO's SOLUTION: ([4, 7, 9, 2, 0, 6, 1, 8, 5, 10, 3], 13)
```

Finding and Learnings:

We have successfully implemented cuckoo search algorithm technique in python. The optimal solution was calculated using a Naïve brute force approach which has a complexity of (n!).An important advantage of this algorithm is its simplicity. In fact, compared with other population- or agent-based metaheuristic algorithms such as particle swarm optimization and harmony search, there is essentially only a single parameter pa in Cuckoo Search (apart from the population size n). Therefore, it is very easy to implement