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

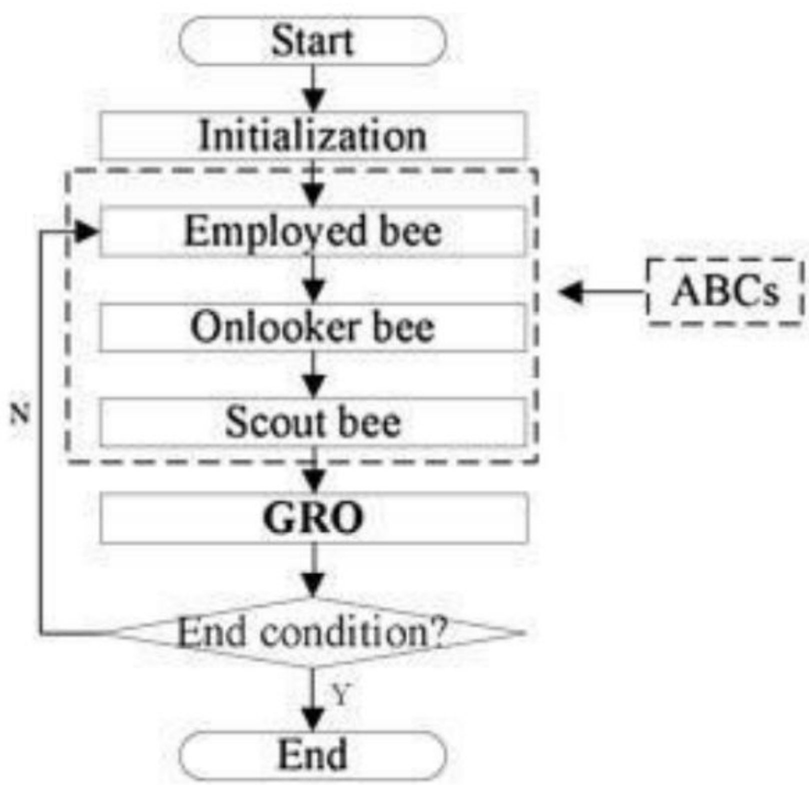
Experiment 3

Aim: Write a program to implement Artificial Bee Colony (ABC) optimization algorithm.

Theory:

In the Artificial Bee Algorithm model, the colony consists of three groups of bees: employed bees, onlookers and scouts. Scouts perform random searches, employed bees collect previously found food and onlookers watch the dances of employed bees and choose food sources depending on dances. Onlookers and scouts are called non-working bees. Communication between bees is based on dances. Before a bee starts to collect food it watches dances of other bees. A dance is the way bees describe where food is.

Working and non-working bees search for rich food sources near their hive. A working bee keeps the information about a food source and shares it with onlookers. Working bees whose solutions can't be improved after a definite number of attempts become scouts and their solutions are not used after that. The number of food sources represents the number of solutions in the population. The position of a food source represents a possible solution to the optimization problem and the nectar amount of a food source corresponds to the quality (fitness) of the associated solution.



Algorithm:

1. BEGIN
2. Initialize the population
3. Find current best agent for the initial iteration
4. Calculate the number of scouts, onlookers and employed bees
5. SET global best to current best
6. FOR iterator = 0 : iteration
   a. evaluate fitness for each agent
   b. sort fitness in ascending order and get best agents
   c. from best agents list select agents from a to c
   d. Create new bees which will fly to the best solution
   e. Evaluate current best agent
   f. IF function(current best) < function (global best)
      i. global best = current best
   g. END IF
7. END FOR
8. Save global best

Source Code:

Artificialbeecolony.py

```
import random
from collections import Iterable
class ABC:
    def __init__(self, objective_function, sn, bound, trial_limit, maximum_cycle_number):
        self.objective_function = objective_function
        self.bound = bound
        self.maximum_cycle_number = maximum_cycle_number
        self.trial_limit = trial_limit
        self.trial = [0] * sn

        self.solutions = \
            [
                [random.uniform(-bound, bound) for arg in
range(self.objective_function.__code__.co_argcount)]
                for f in range(sn)
            ]
        self._eval_solutions()
        for c in range(self.maximum_cycle_number):
            self._employed_phase()
            self._eval_prob()
```

```
        self._onlookers_phase()

    @staticmethod
    def _fitness_function(function_f):
        if function_f >= 0:
            return 1 / (1 + function_f)
        else:
            return 1 + function_f

    def _eval_prob(self):
        sum_fit = sum(self.fit)
        self.prob = [self.fit[i] / sum_fit for i in range(len(self.solutions))]

    def eval_solution(self, solution):
        """Calculates objective_function and fitness_function values"""
        if isinstance(solution, int):
            obj_val = self.objective_function(self.solutions[solution])
        elif isinstance(solution, Iterable):
            obj_val = self.objective_function(*solution)
        else:
            raise Exception("Expected solution to be int or Iterable, instead found ", type(solution))
        fit_val = ABC._fitness_function(obj_val)
        return obj_val, fit_val

    def _eval_solutions(self):
        self.function = list(map(lambda args: self.objective_function(*args), self.solutions))
        self.fit = list(map(ABC._fitness_function, self.function))

    def best_solution(self):
        i = self.fit.index(max(self.fit))
        return self.solution_detail(i)

    def worst_solution(self):
        i = self.fit.index(min(self.fit))
        return self.solution_detail(i)

    def solution_detail(self, i):
        return {"solution": self.solutions[i], "function": self.function[i], "fitness": self.fit[i],
                "trial": self.trial[i]}

    def _new_v_solution(self, i):
        k = random.choice([k for k in range(len(self.solutions)) if k != i])
```

```
    j = random.randrange(self.objective_function.__code__.co_argcount)
    xkj = self.solutions[k][j]
    xij = self.solutions[i][j]
    phi = random.uniform(-1, 1)
    new_xj = xij + phi * (xij - xkj)
    new_xj = self._bound(new_xj)
    new_solution = self.solutions[i][:]
    new_solution[j] = new_xj
    return new_solution

def _new_x_solution(self, i):
    # Randomly select a variable j
    j = random.randrange(self.objective_function.__code__.co_argcount)
    # Generate new solution new_x and bound it
    xij = self.solutions[i][j]
    r = random.uniform(0, 1)
    new_xj = -self.bound + r * (self.bound - (-self.bound))
    new_xj = self._bound(new_xj)
    new_solution = self.solutions[i][:]
    new_solution[j] = new_xj
    return new_solution

def _bound(self, value):
    if value >= self.bound:
        return self.bound
    elif value <= -self.bound:
        return -self.bound
    return value

def _accept_solution(self, i, new_solution, new_obj_val=None, new_fit_val=None):
    if not new_obj_val:
        new_fit_val = ABC._fitness_function(new_obj_val)
        if not new_fit_val:
            new_obj_val, new_fit_val = self.eval_solution(new_solution)
    self.solutions[i] = new_solution
    self.fit[i] = new_fit_val
    self.function[i] = new_obj_val
    self.trial[i] = 0

def _employed_phase(self):
    for i in range(len(self.solutions)):
        new_solution = self._new_v_solution(i)
```

```
            self._general_phase(new_solution, i)

    def _onlookers_phase(self):
        for n in range(len(self.solutions)):
            i = random.choices(range(len(self.solutions)), weights=self.prob)[0]
            new_solution = self._new_v_solution(i)
            self._general_phase(new_solution, i)

    def _scout_phase(self, i):
        new_solution = self._new_x_solution(i)
        self._general_phase(new_solution, i)

    def _general_phase(self, new_solution, i=None):
        new_obj_val, new_fit_val = self.eval_solution(new_solution)

        if new_fit_val > self.fit[i]:
            self._accept_solution(i, new_solution, new_obj_val, new_fit_val)
        else:
            self.trial[i] += 1
            if self.trial[i] >= self.trial_limit:
                self.trial[i] = 0
                self._scout_phase(i)
```

main.py

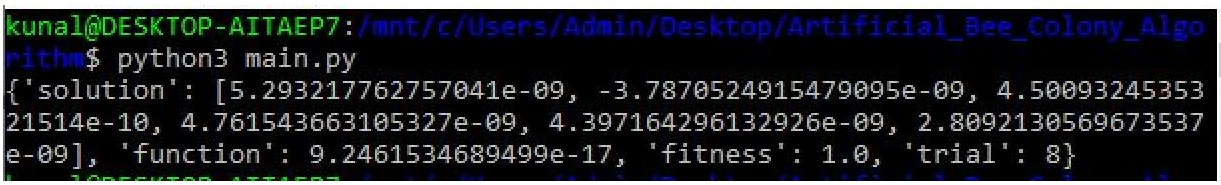
```
from Artificialbeecolony import ABC
import math
Bukin_function_N_6 = lambda x, y: 100 * (math.sqrt(abs(y - 0.01 * x ** 2)) + 0.01 * abs(x + 
10))

Ackley_function = lambda x, y: -20 * math.exp(-.02 * math.sqrt(0.5 * (x ** 2 + y ** 2))) - 
math.exp(0.5 * (math.cos(2 * math.pi * x) + math.cos(2 * math.pi * y))) + math.e + 20

sphere_function = lambda x1, x2, x3, x4, x5, x6: x1 ** 2 + x2 ** 2 + x3 ** 2 + x4 ** 2 + x5 ** 2 
+ x6 ** 2

SN = 10
limit = 50
MCN = 1000
bound = 40
result = ABC(Ackley_function, SN, bound, limit, MCN)
print(result.best_solution())
```

Output:

Ackley_function

Bukin_function_N_6

sphere_function



Finding and Learnings:

We have successfully implemented the Artificial Bee colony Algorithm in python. The ABC(Artificial Bee Colony) model consists of four phases that are accomplished sequentially, Initialization Phase, Exploitation Phase, Refinement Phase and Exploration Phase where scout bees are sent out to unexplored regions of the search domain.