```
from bokeh.plotting import figure, show
import math
import random
import copy
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import networkx as nx
import seaborn as sns
```

Define the problem parameters such as n = number of nodes , adj = adjacency matrix of the graph , Graph_edge = array of graphs - this will simplify the graph visualization process - And lastly defualt_n_community = number of communities

```
class Problem:
    def __init__(self, n, adj, graph_edge, default_n_community):
        self.n = n
        self.adj = adj
        self.m = int(sum([len(x) for x in self.adj])/2)
        self.default_n_community = default_n_community
        self.graph_edge = graph_edge

```

We want to create a random first population and improve on this. To this we first shuffle the array list then we will iterate on it. In each iteration we will choose members of a community randomly and assign their community to them. Done by this section of the code :

```
selected = random.choices(vertices, k=int(self.n / self.default_n_community))
        vertices = [e for e in vertices if e not in selected]
        for i in selected:
          individual_map[i] = counter
        counter += 1
```

Keep in mind in the last iteration there is only one possible community to assign the remaining nodes

In the end we will sort the randomly generated populations based on the fitness function. We can see the formula and code below :

$$Q = \frac{1}{2m}\sum_{i,j}\left[A_{ij} - \frac{k_i k_j}{2m}\right]\delta(c_i, c_j)$$

```
self.population = sorted(self.population, key=lambda agent: self.fitness(
            agent), reverse=False)
```

```
class Problem(Problem):
    def initial_population(self):
        self.population = []
        # We want to create a number of populations

        for _ in range(self.population_size):

            # For each one we will randomly put the nodes on a community

            vertices = list(range(self.n))
            random.shuffle(vertices)
            individual_map = [None] * self.n

            counter = 0

            while len(vertices) != 0:
                if len(vertices) < int(self.n / self.default_n_community):
                    selected = vertices[:]
                    for i in selected:
                        individual_map[i] = counter
                    counter += 1
                    break
                else:
                    selected = random.choices(vertices, k=int(self.n / self.default
                    vertices = [e for e in vertices if e not in selected]
                    for i in selected:
                        individual_map[i] = counter
                    counter += 1

            self.population.append(individual_map)

        self.population = sorted(self.population, key=lambda agent: self.fitness(
            agent), reverse=False)
```

In the cuckoo algorithm each cuckoo will breed a random number of eggs and place them in its corresponding permitted radius (defined as elr). In our code we will select a random number of eggs.

```
number_of_eggs = random.randint(1, min(4, self.n) )
```

then we will select a number of nodes (number of eggs) and randomly change their community to one of their neighbors

```
gene = [random.randint(1, self.n - 1) for _ in range(number_of_eggs)]
            while len(set(gene)) < number_of_eggs :
                gene = [random.randint(1, self.n - 1) for _ in range(number_of_eggs)]
            for j in range(len(gene)):
                joining_node = random.choice([exc for exc in adj[gene[j]] if exc != gene[j] ]
                new_egg_pop = self.population[i][:]
                new_egg_pop[gene[j]] = self.population[i][joining_node]
                breaded_eggs.append(new_egg_pop)
```

As it is done in cuckoo optimization algorithm we will remove 10% of the least desirable eggs generated

```
num_to_remove = round(len(breaded_eggs)/10)
        breaded_eggs = sorted(breaded_eggs, key=lambda agent: self.fitness(agent),
                              reverse=False)[num_to_remove:]
```

Finally we will maintain our population by removing the ones performing badly. This is basically the notion of : survival of the fittest

```
self.population = sorted(self.population, key=lambda agent: self.fitness(
            agent), reverse=False)[:self.population_size]
```

```
class Problem(Problem):
    def egg_breading(self):
        breaded_eggs = []
        for i in range(self.population_size):
            number_of_eggs = random.randint(2, min(10, self.n) )
            gene = [random.randint(1, self.n - 1) for _ in range(number_of_eggs)]
            while len(set(gene)) < number_of_eggs :
                gene = [random.randint(1, self.n - 1) for _ in range(number_of_eggs
            for j in range(len(gene)):
                joining_node = random.choice([exc for exc in adj[gene[j]] if exc !=
                new_egg_pop = self.population[i][:]
                new_egg_pop[gene[j]] = self.population[i][joining_node]

                breaded_eggs.append(new_egg_pop)
        ############## remove 10%
        num_to_remove = round(len(breaded_eggs)/10)
        breaded_eggs = sorted(breaded_eggs, key=lambda agent: self.fitness(agent),
                              reverse=False)[num_to_remove:]
```

```
        for x in breaded_eggs:
            self.population.append(x)
        ############## maintain population
        self.population = sorted(self.population, key=lambda agent: self.fitness(
            agent), reverse=False)[len(self.population) - self.population_size:]


```

In this function we will perform the migration task. The best performing chromosome is chosen as the goal. Every other chromosome will move towards it with a predefined probability

```
class Problem(Problem):
    def migration(self):
        goal = self.population[len(self.population)-1]
        #move towards the best chromosome with a preset probibality
        for i in range(0, len(self.population) - 2):
            migration_rate = 0.7
            for j in range(self.n):
                if random.uniform(0, 1) > migration_rate:
                    continue;
                self.population[i][j] = goal[j]

```

In this function we will calculate the fitness of the cuckoo given based on the below formula

$$Q = \frac{1}{2m}\sum_{i,j}\left[A_{ij} - \frac{k_i k_j}{2m}\right]\delta(c_i, c_j)$$

```
class Problem(Problem):
    def fitness(self, individual):
        # Calculate the fitness of a chromosome based on the formula provided

        Q = 0
        for i in range(self.n):
            for j in range(self.n):
                Q += (int(j in self.adj[i]) - ((len(self.adj[i]) * len(self.adj[j])
        Q /= (2 * self.m)
        return Q


```

This function will calculate the fitness of every chromosome of our population

```
class Problem(Problem):
    def evalute(self):
        pop_fit = [None] * len(self.population)
```

```
        for i in range(len(self.population)) : pop_fit[i] = self.fitness(self.popul
        return sum(pop_fit), max(pop_fit)
```

This function will visulize our answer

```
class Problem(Problem):
    def graph_visulization(self):
        fig2, ax2 = plt.subplots()
        ax2.set_title('Communities')
        G = nx.Graph()
        G.add_edges_from(self.graph_edge)
        color_map = [node for node in self.population[-1:]]
        nx.draw_networkx(G, node_color = color_map[0])


```

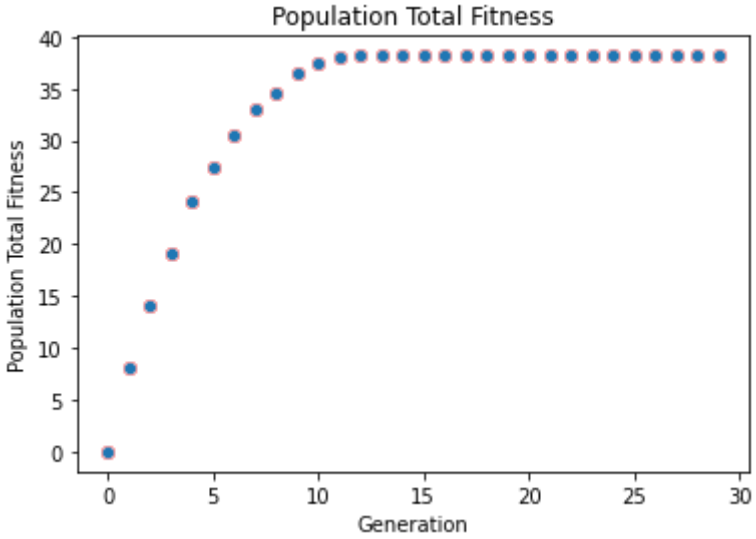
We will commence the cuckoo optimization algorithms. first we will breed the eggs than migrate each chromosome to the best one (base on fitness function)

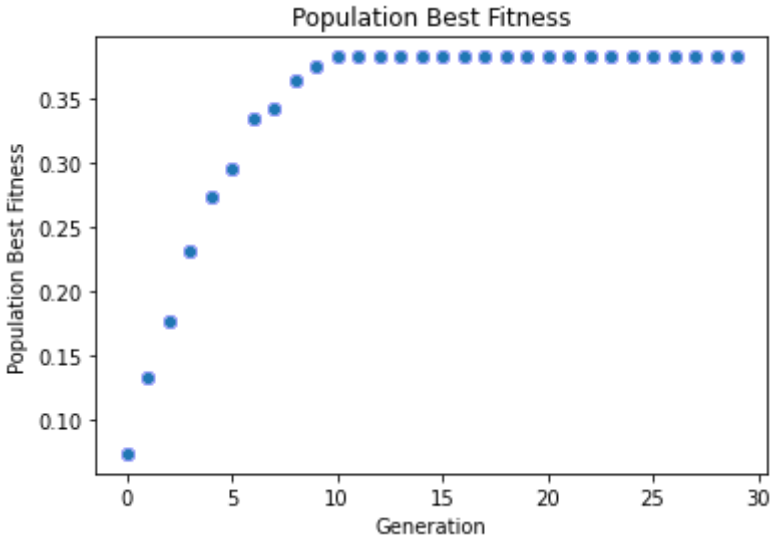
```
class Problem(Problem):
    def Cuckoo(self, population_size, n_generations, high_egg, low_egg):
        self.population_size = population_size
        self.n_generations = n_generations

        # Make the random first population
        self.initial_population()

        plotFitness = []
        plotEpoch = []
        plotPopFit = []
        for epoch in range(self.n_generations):
            # Start laying the eggs - after this function we only have a set of mat
            self.egg_breading()
            # Start migrating toward the best chromosome
            self.migration()
            eval_ = self.evalute()
            plotFitness.append(eval_[1])
            plotEpoch.append(epoch)
            plotPopFit.append(eval_[0])
            print("Epoch", epoch, ":\tPopulation total fitness:", eval_[0], "\tBest

        fig, ax = plt.subplots()
        ax.scatter(plotEpoch, plotPopFit, color = 'r')
        ax.set_title('Population Total Fitness')
        ax.set_xlabel('Generation')
        ax.set_ylabel('Population Total Fitness')
        sns.scatterplot(x=plotEpoch, y=plotPopFit)

```

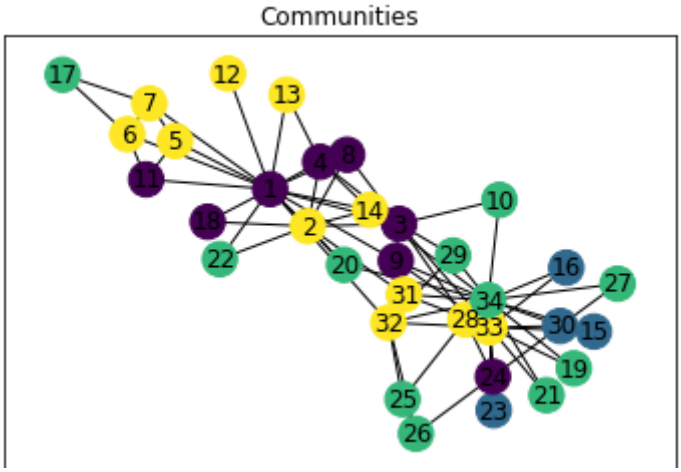
```
        fig1, ax1 = plt.subplots()
        ax1.scatter(plotEpoch, plotFitness, color = 'b')
        ax1.set_title('Population Best Fitness')
        ax1.set_xlabel('Generation')
        ax1.set_ylabel('Population Best Fitness')
        sns.scatterplot(x=plotEpoch, y=plotFitness)
        self.graph_visulization()

```

```
from google.colab import files
uploaded = files.upload()
f = open('sample dataset.txt', 'r')
lines = f.readlines()
n = int(lines[0])
lines = lines[1:]

adj = [[] for _ in range(n)]
graph_edge = []

for edge in lines:
  edge = edge.split()
  graph_edge.append(edge)

  adj[int(edge[0]) - 1].append(int(edge[1]) - 1)
  adj[int(edge[1]) - 1].append(int(edge[0]) -1)

# Define problem parameters : Number of nodes, adjacency matrix, array of edges anc
problem = Problem(n, adj, graph_edge, 7)

problem.Cuckoo(population_size = 100, n_generations = 30, high_egg = 5, low_egg = 1


```

```
Epoch 19 :      Population total fitness: 38.190335305719785    Best fitness: 0.
Epoch 20 :      Population total fitness: 38.190335305719785    Best fitness: 0.
Epoch 21 :      Population total fitness: 38.190335305719785    Best fitness: 0.
Epoch 22 :      Population total fitness: 38.190335305719785    Best fitness: 0.
Epoch 23 :      Population total fitness: 38.190335305719785    Best fitness: 0.
Epoch 24 :      Population total fitness: 38.190335305719785    Best fitness: 0.
Epoch 25 :      Population total fitness: 38.190335305719785    Best fitness: 0.
Epoch 26 :      Population total fitness: 38.190335305719785    Best fitness: 0.
Epoch 27 :      Population total fitness: 38.190335305719785    Best fitness: 0.
Epoch 28 :      Population total fitness: 38.190335305719785    Best fitness: 0.
Epoch 29 :      Population total fitness: 38.190335305719785    Best fitness: 0.
```







```
1
```

✓ 50s completed at 2:47 AM