SOCIAL INFLUENCE MAXIMIZATION
USING DISCRETE SHUFFLED FROG-LEAPING ALGORITHM

MINOR PROJECT – II

SUBMITTED BY –

HARSHIT SHARMA (9919103016)
PRANAT JAIN (9919103017)
SIDDHARTH SINGH (9919103029)
YASHA JAFRI (9919103009)

UNDER THE SUPERVISION OF –

DR. SHIKHA K MEHTA

DEPARTMENT OF CS/IT

JAYPEE INSTITUTE OF INFORMATION TECHNOLOGY, NOIDA

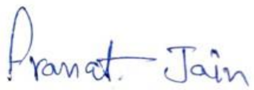
MAY 2022

ACKNOWLEDGEMENT

We would like to place on record our deep sense of gratitude to Dr. Shikha K Mehta, Associate Professor, Jaypee Institute of Information Technology, India for her generous guidance, help and useful suggestions for the progress of this project.

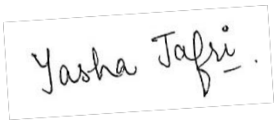
We also wish to extend our heartfelt gratitude to our batchmates for their insightful comments and constructive suggestions to improve the quality of this project work. Lastly, we would like to thank our panellists to be the source of motivation & change which we needed during the course of this project.

Harshit Sharma (9919103016)



Pranat Jain (9919103017)

Siddharth Singh (9919103029)



Yasha Jafri (9919103009)

DECLARATION

We hereby declare that this submission is our own work and that, to the best of our knowledge and beliefs, it contains no material previously published or written by another person nor material which has been accepted for the award of any other degree or diploma from a university or other institute of higher learning, except where due acknowledgment has been made in the text.

Place: Noida, Uttar Pradesh

Date: 18^{th} May, 2022

Harshit Sharma
9919103016

Pranat Jain
9919103017

Siddharth Singh
9919103029

Yasha Jafri
9919103009

CERTIFICATE

This is to certify that the work titled "Social Influence Maximization using Discrete Shuffled Frog-leaping Algorithm" submitted by Harshit Sharma, Pranat Jain, Siddharth Singh, Yasha Jafri of B.Tech of Jaypee Institute of Information Technology, Noida has been carried out under my supervision. This work has not been submitted partially or wholly to any other University or Institute for the award of any other degree or diploma.

Dr. Shikha K Mehta

(Associate Professor, Jaypee Institute of Information Technology, Noida)

18th May, 2022

ABSTRACT

Since the beginning, the biggest trait that has enhanced Human development has been ‘Learning from others. Many of our actions we take today are influenced by others. People learn from each other and this has important implications on a person’s thought processes. The biggest platform where people can influence others' choices and decisions is social media.

Due to the growth of the Internet and Web 2.0, many large-scale online social network sites like Facebook, Twitter, Instagram, etc. became successful because they are very effective tools in connecting people and bringing small and disconnected offline social networks together. Moreover, they are also becoming a huge dissemination and marketing platform, allowing information and ideas to influence a large population in a short period of time. However, to fully utilize these social networks as marketing and information dissemination platforms, many challenges have to be met.

In this project, we present our work towards addressing one of the challenges, finding influential individuals/groups efficiently in a large-scale social network. This problem, referred to as *Influence Maximization,* would be of interest to many companies as well as individuals who want to promote their products, services and innovative ideas through the powerful word-of-mouth effect (or called viral marketing). Online social networks provide good opportunities to address this problem, because they are connecting a huge number of people and they collect a huge amount of information about the social network structures and communication dynamics.

Influence Maximization is the problem of finding a small subset of nodes (seed nodes) in a social network that could maximize the spread of influence. This is an active area of research in the computational social network analysis domain. Due to its practical importance in various domains, such as viral marketing, target advertisement and personalized recommendation, the problem has been studied in different variants, and different solution methodologies have been proposed over the years.

This project mainly focuses on designing an algorithm that identifies such nodes in a social network that maximize influence. We also idealize to improve the efficiency of the algorithm using different approaches and models like Independent Cascade Model, Linear Threshold Model and Discrete Shuffled Frog-leaping Algorithm.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

INTRODUCTION

Since the beginning, the biggest trait that has enhanced Human development has been ‘Learning from others. Many of our actions we take today are influenced by others. People learn from each other and this has important implications on a person’s thought processes. The biggest platform where people can influence others' choices and decisions is social media.

Due to the growth of the Internet and Web 2.0, many large-scale online social network sites like Facebook, Twitter, Instagram, etc. became successful because they are very effective tools in connecting people and bringing small and disconnected offline social networks together. Moreover, they are also becoming a huge dissemination and marketing platform, allowing information and ideas to influence a large population in a short period of time. However, to fully utilize these social networks as marketing and information dissemination platforms, many challenges have to be met.

Influence Maximization is the problem of finding a small subset of nodes (seed nodes) in a social network that could maximize the spread of influence. This is an active area of research in the computational social network analysis domain. Due to its practical importance in various domains, such as viral marketing, target advertisement and personalized recommendation, the problem has been studied in different variants, and different solution methodologies have been proposed over the years.

BACKGROUND STUDY

WHAT IS SOCIAL INFLUENCE MAXIMIZATION?

Since the beginning, the biggest trait that has enhanced Human development has been ‘Learning from others. Many of our actions we take today are influenced by others. People learn from each other and this has important implications on a person’s thought processes. The biggest platform where people can influence others' choices and decisions is social media.

Due to the growth of the Internet and Web 2.0, many large-scale online social network sites like Facebook, Twitter, Instagram, etc. became successful because they are very effective tools in connecting people and bringing small and disconnected offline social networks together. Moreover, they are also becoming a huge dissemination and marketing platform, allowing information and ideas to

influence a large population in a short period of time. However, to fully utilize these social networks as marketing and information dissemination platforms, many challenges have to be met.

Influence Maximization is the problem of finding a small subset of nodes (seed nodes) in a social network that could maximize the spread of influence. This is an active area of research in the computational social network analysis domain. Due to its practical importance in various domains, such as viral marketing, target advertisement and personalized recommendation, the problem has been studied in different variants, and different solution methodologies have been proposed over the years.

SIM PROBLEM & ITS VARIANTS

- **Basic SIM Problem**
  In the basic version of the TSS Problem along with a directed social network G(V, E,θ, P), we are given two integers: k and λ and asked to find out a subset of most k nodes, such that after the diffusion process is over at least λ number of nodes are activated.
- **Top-K Node Influence Maximization / Social Influence Maximization Problem**
  For a given social network G(V, E,θ, P), this problem asks to choose a set S of k nodes (i.e., S ⊆ V(G) and |S| = k), such that the maximum number of nodes of the network become influenced at the end of diffusion process, i.e., σ (S) will be maximized. Most of the algorithms presented in Sect. 6 are solely developed for solving this problem
- **Influence Spectrum Problem**
  Along with the social network G(V , E, θ, P), we are also given with two integers: k lower and k upper with k upper > k lower. Our goal is to choose a set S for each k ∈ [k lower, k upper], such that social influence in the network (σ (S)) is maximum in each case. Intuitively, solving one instance of this problem is equivalent to solving (k upper −k lower+1) instances of SIM Problem.

As viral marketing is basically done in different phases and in each phase, seed set of different cardinalities can be used, influence spectrum problem appears in a natural way.

- **Multi-round Influence Maximization Problem**

  Most of the existing studies of influence maximization consider that the seed set selection is one shot task, i.e., the entire seed set has to be selected before the diffusion starts. However, in many real-world advertisement scenarios, it may be required that the viral marketing need to be conducted in multiple times. To model this scenario, 'Multi-Round Influence Maximization Problem' has been introduced by Sun et al. In this problem, along with the social network G (V, E , θ, P), we are also given with two integers: k and T . Here, T is the number of times the diffusion process needs to be conducted and k is the cardinality of the seed set. Here, the goal is to choose the seed nodes S1, S2 , ..., ST , such that at the end of the entire diffusion process, the total number of influenced nodes becomes maximum.

- **Target Set Selection Problem**

  For or a given social network G(V, E ,θ, P), the goal is to select a seed set, whose initial activation leads to the complete influence in the network, i.e., all the nodes are influenced at the end of diffusion process.

- **Weighted Target Set Selection Problem**

  Along with a social network G(V, E ,θ, P), we are given another vertex weight function, φ : V(G) → N0, signifying the cost associated with each vertex. This problem asks to find out a subset S, which minimizes total selection cost, and also all the nodes will be influenced at the end of diffusion.

- **Budgeted Influence Maximization**

  Along with a directed graph G(V, E,θ, P), we are given with a cost function C: V(G) −→ Z+ and a fixed budget B ∈ Z+. Cost function C assigns a nonuniform selection cost to every vertex of the network, which is the amount of incentive need to be paid, if that vertex is selected as a seed node. This problem asks for selecting a seed set within the budget, which maximizes the spread of influence in the network.

INDEPENDENT CASCADE MODEL

The Independent Cascade Model is an information diffusion model where the information flows over the network through cascade. Nodes can have two states -

Active: it means the node is already influenced by the information in diffusion
Inactive: node is unaware of the information or not influenced.

The process runs in discrete steps. At the beginning of ICM process, few nodes are given the information, they are known as seed nodes. Upon receiving the information, these nodes become active. In each discrete step, an active node tries to influence one of its inactive neighbours. Regardless of its success, the same node will never get another chance to activate the same inactive neighbour. The success of node u in activating the node v depends on the propagation probability of the edge (u, v) defined as p(v), each edge has its own value. The process terminates when no further node gets activated.

LINEAR THRESHOLD MODEL

For any node, all its neighbors, who are activated just at the previous timestamp together make a try to activate that node. This activation process will be successful, if the sum of the incoming active neighbor's probability becomes either greater than or equal to the node's threshold. then ui will become active at time stamp t + 1. This method will be continued until no more activation is possible. In this model, we can use the negative influence, which is not possible in IC Model.

DISCRETE SHUFFLED FROG-LEAPING ALGORITHM

It is a methodology, which depends on imitation of the behaviour patterns of frogs, taking into account a crowd of frogs leaping in a swamp, on the lookout for the place, which has the highest food quantity reachable, in which the swamp has multiple stones at distinct points that make it easier for the frogs to step on.

The aim is to identify a stone with the maximum food amount available. Communicating between frogs can progress their memes as infection can be propagated among them. As a result of improvement in memes, each frog's position will be changed by tuning its leaping step size. The population is meant to be a number of solutions which is represented as a swarm of frogs that is segregated into subgroups denoted to as memeplexes.

Each memeplex represents a distinct frog community, each conducting a local quest. The specific frogs grip ideas within each memeplex, which can be persuaded by further frogs' ideas and progress through a memetic evolution process.

In a shuffling process, ideas are passed between memeplexes exactly after several memetic evolution phases. This algorithm combines the advantages of two categories of genetics-based algorithms (such as memetics) and social behaviour-based algorithms (such as the PSO bird algorithm). It seeks to strike a balance between extensive scrutiny in the space of possible answers. In this population algorithm, a population of frogs (answers) consists, each frog will have a chromosome-like structure in the genetic algorithm. The whole population of frogs is divided into smaller groups, each group representing different types of frogs that are scattered in different places of the answer space. Each group of frogs then begins a precise local search around their habitat.

REQUIREMENT ANALYSIS

The requirements for this project are:

1) Python 3.x

   Python is a very varied tool used for Predictive Analysis, due to the availability of various modules which ease the implementation of the required analyses.

2) Python Modules –

   i) Pandas – to create and manage data frames containing data

   ii) NetworkX – provides tools for implementing Graph Networks in Python

   iii) Matplotlib – Plotting the Network Graph

3) Google Colab (For Implementation purposes)

   It is an Online tool to work on our projects without the need to install any modules on our own environment. It has several advantages over the traditional ways to create a Virtual Environment, where several users ran into problems.

DETAILED DESIGN

ALGORITHM DESIGN

1) To work on the Social Influence Maximization, we need a Social Network to work upon. We have used the Game of Thrones Network to start with. The Dataset was created from the famous books of “A Song of Ice and Fire” on which the popular TV Series “Game of Thrones” is based upon. The Dataset counts an undirected edge between two characters if they occur in any 15-word frame within the book. Here, characters are the fictional characters involved in the book. The Dataset used is divided into 5 different datasets which contains the edge counts between two characters in the same book in a single dataset.

2) We use the Pandas Library to import these datasets and concatenate these datasets to form a single data frame to work upon. The resultant Data Frame is converted into another data frame of different structure along 3 columns – Source, Target and Weight, using the ‘group by’ functionality of Pandas Library, where the ‘Source’ is the Source Node, ‘Target’ is the Target Node and ‘Weight’ refers to the Edge weight between two nodes.

3) Once we have received a Pandas Edge list, we use the ‘from_pandas_edgelist’ function of the NetworkX library to create an Undirected Graph which will be used later in the code for various purposes. To visualize the Graph, we also plot this graph using the Matplotlib Library.

4) To start with the DSFLA, we first have to create a Population of Memes, which is created by choosing Random nodes from the graph. The size of population, memeplexes to be created, memes in each memeplex and meme types to be contained in a single meme are taken as input.

5) Once the Population has been created, we calculate the Local Influence Estimation Value (LIE) for each meme to divide these memes into different memeplexes. To calculate the LIE for each meme we use the below given formula –

$$
\begin{aligned}
LIE(S) &= k + \sigma_1^*(S) + \frac{\sigma_1^*(S)}{|N_S^{(1)} \backslash S|} \sum_{u \in N_S^{(2)} \backslash S} p_u^* d_u^* \\
&= k + \left(1 + \frac{1}{|N_S^{(1)} \backslash S|} \sum_{u \in N_S^{(2)} \backslash S} p_u^* d_u^*\right) \\
&\quad \times \sum_{i \in N_S^{(1)} \backslash S} \left(1 - \prod_{(i,j) \in E,\, j \in S} (1 - p_{i,j})\right)
\end{aligned}
$$

where $N_S^{(1)}$ and $N_S^{(2)}$ represent the one-hop and two-hop area of candidate set S, respectively. p_u^* is a small constant probability of a propagation cascade model. d_u^* is the number of edges of node u within $N_S^{(1)}$ and $N_S^{(2)}$.

6) We sort the Population of Memes in descending order of their respective LIE Values and use the descending order nodes to create the memeplexes such that there is equal representation of fitness value or LIE Values in the Memeplexes. Once we have created the memeplexes, we iterate over each memeplex and follow the steps (a) to (d) in order –

a) Construct a Sub memeplex from the existing Memeplex

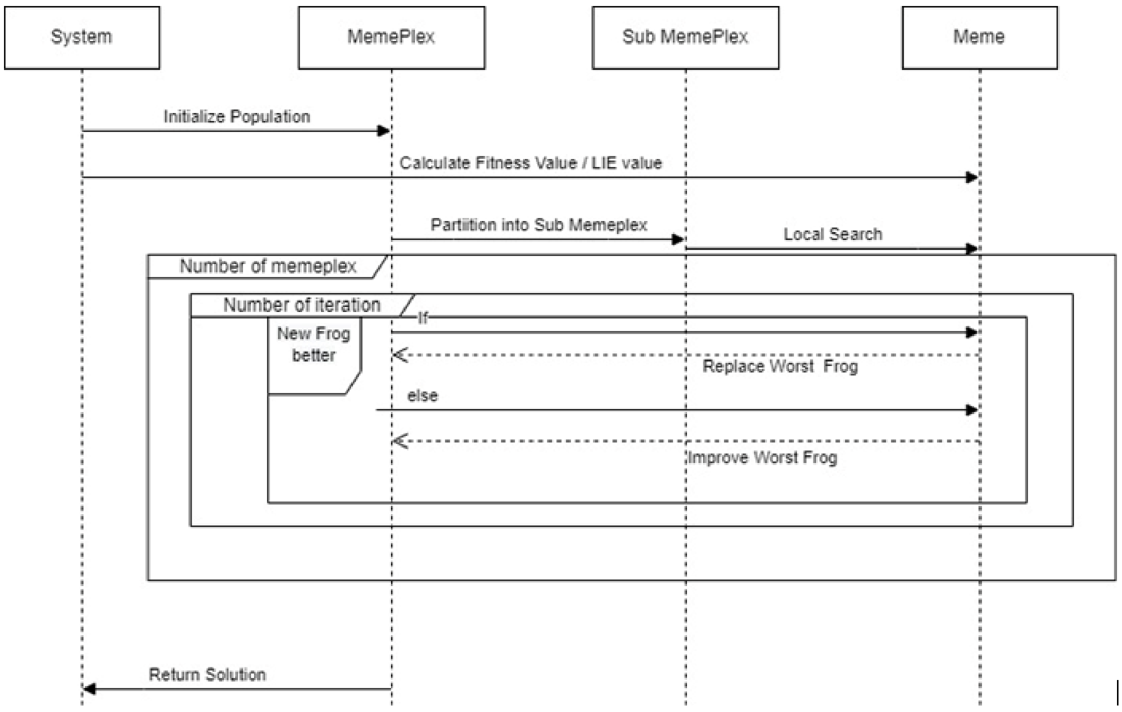
b) Choose the P_b and P_W, which represent the Best and Worst Frogs of the Submemeplex

c) We choose a New Meme with respect to the Local Best meme to replace the Worst Meme and calculate its LIE Value. If the calculated LIE value of the new meme using the Local Best Meme is greater than the current LIE value of the worst meme, we replace the worst meme by the New Meme, otherwise we repeat the same process for the New meme created with respect to Global Best Meme.

d) If the Worst Meme has been replaced by any of the new meme created in the above step, we go to the next step, otherwise we choose a random frog containing random meme types and replace the worst meme.

7) We iterate over the population of memeplexes to improve the fitness of memes in a memeplex. We rearrange the memes in the descending order of their LIE values, once the iterations are over. Once the resultant population is achieved, the Meme with the best LIE value is chosen as the Top- K Nodes for the purpose of Influence Maximization.

SEQUENCE DIAGRAM



IMPLEMENTATION

We have mainly worked with Pandas Edge list where we have converted the Data frame into an Edge list which has been fed into the NetworkX Library functions to create a NetworkX Graph.

1) Random Population Initialization

```
m = 10 # No. of memeplexes
n = 5 # No. of memes in each memeplex
k = 3 # No. of memetypes in each meme
itr = 1 # No. of iterations
F = m * n
population = []
for i in range(F):
    population.append([random.choice(list(weighted_degrees)) for i in
range(k)])
```

2) Local Influence Estimation Value Calculation

```
population_LIE = [] # List to store the LIE values of all memes
for i in range(F):
    population_LIE.append(LIE(population[i]))
# Sorting the Memes on the basis of their LIE values
population_LIE, population = sort_memes(population_LIE, population)
population = list(zip(population, population_LIE))
```

3) The description of functions used in (2) are as follows:

```
def one_hop_area(meme):
    """
    Compute the One-Hop Area, i.e., the nodes adjacent to the nodes in
the given meme
    Parameters:
        meme = Meme whose One Hop Area is to be calculated
    Returns:
        List of Nodes which are adjacent to atleast one node within a
meme
    """
    temp = [] # Temporary list to store the One-Hop Area nodes of the
meme
    for i in meme:
        temp.extend(list(GOT.neighbors(i)))
    return list(set(temp) - set(meme)) # Returning only unique nodes,
to avoid Node Repetition

def two_hop_area(meme):
    """
    Compute the Two-Hop Area, i.e., the nodes adjacent to the One-Hop
area nodes of the given meme
    Parameters:
        meme = Meme whose Two Hop Area is to be calculated
    Returns:
        List of Nodes which are adjacent to atleast one node within the
One-Hop Area Nodes but not a part of the Meme
    """
    one_hop = one_hop_area(meme)
```

```
    temp = [] # Temporary list to store the Two-Hop Area nodes of the
meme
    for i in one_hop:
        temp.extend(list(GOT.neighbors(i)))
    return list(set(temp) - set(meme) - set(one_hop)) # Returning only
unique nodes, to avoid Node Repetition

def calc_pcm_prob(nodes):
    """
    Calculates the Constant Propogation Cascade Probability of the nodes
of the given meme
    Parameters:
        nodes = List of Nodes whose Cascade Probability has to be
calculated
    Returns:
        List of Probabilities where ith probability corresponds to the
Cascade Probability of the ith node of the list of given nodes
    """
    prob = [] # Temporary List to store the corresponding Cascade
Probability of the Nodes
    for i in nodes:
        prob.append(weighted_degrees[i] / GOT.number_of_nodes())
    return prob

def calc_edges(group1, group2):
    """
    Calculates the number of edges each Group-2 Node has within the nodes
Group-1 & Group-2
    Parameters:
        group1 = First Group of Nodes
        group2 = Second Group of Nodes
    Returns:
        List of number of edges where ith entry corresponds to the number
of edges Node-i of Group-2 has within nodes of Group-1 & Group-2
    """
    count = [] # Temporary storage to store the Number of edges
corresponding each node
    for i in group2:
        edges = list(nx.edges(GOT, nbunch = [i]))
        temp = 0
        for i in edges:
            if (i[1] in group1) or (i[1] in group2):
                temp += 1
        count.append(temp)
    return count

def sum_pd(list1, list2):
    """
    Computes the sum-product of entries of two list
    Parameters:
        list1: A List of Floating-point Numbers
        list2: A List of Floating-point Numbers
    Returns:
        Sum of the Products of each corresponding Entry of the two lists
    """
    p = 0 # Temporary variable to store the Sum-Product of the two lists
```

```
    for i in range(len(list1)):
        p += (list1[i] * list2[i])
    return p

def calc_edge_prob(meme, one_hop):
    """
    Returns the sum of the Edges probabilities of the nodes with their
One-Hop area nodes
    Parameters:
        meme = Meme whose nodes' edge probabilities have to be calculated
        one_hop = One-Hop Area nodes of the given meme
    Returns:
        Sum of the Edge Probabilities using the formula - 1 - Product(1
- Pij)
    """
    N = GOT.number_of_nodes() # Total Nodes in the Graph
    prob_sum = 0 # Temporary variable to store the sum of the Edge
Probabilities
    for i in one_hop:
        prob_prod = 1
        for j in meme:
            pij = 0.01
            pij += ((GOT.degree(i) + GOT.degree(j)) / N)
            pij += (len(list(nx.common_neighbors(GOT, i, j))) / N)
            prob_prod *= (1 - pij)
        prob_sum += (1 - prob_prod)
    return prob_sum

def LIE(meme):
    """
    Calculates the Local Influence Spread Measure of the nodes of the
meme
    Parameters:
        meme = Meme whose LIE value has to be calculated
    Returns:
        A Floating-point Number denoting the Local Influence Spread
Measure
    """
    Ns1_S = one_hop_area(meme) # One-Hop area of the Meme
    Ns2_S = two_hop_area(meme) # Two-Hop Area of the Meme
    pu = calc_pcm_prob(Ns2_S)
    du = calc_edges(Ns1_S, Ns2_S)
    return k + ((1 + ((1 / len(Ns1_S)) * sum_pd(pu, du))) *
calc_edge_prob(meme, Ns1_S))

def sort_memes(ppl_lie, ppl):
    """
    Sorts the Population of Memes on the basis of their LIE values
    Parameters:
        ppl_lie = LIE values of each corresponding meme
        ppl = Population of Memes
    Returns:
        Tuple of List denoting the Sorted LIE values and Population of
Memes in descending order
    """
    result = sorted(list(zip(ppl_lie, ppl)), reverse = True)
```

```
    result = zip(*result)
    result = [list(tuple) for tuple in result]
    return result[0], result[1]
```

4) Memeplex Creation

```
# Creating Memeplexes using Uniform Distribution of Memes based on their
LIE values
memeplex = []
for i in range(m):
    meme = []
    for j in range(n):
        meme.append(population[i + (j * m)])
    memeplex.append(meme)
```

5) Local Exploitation

```
for x in range(itr):
    Px = degree_mplx[0][0][0]
    Px_LIE = degree_mplx[0][0][1]
    for i in range(1, m):
        if degree_mplx[i][0][1] < Px_LIE:
            Px = degree_mplx[i][0][0]
            Px_LIE = degree_mplx[i][0][1]
    temp2 = LDR(Px, 1)

    for i in degree_mplx:
        sub_degree_mplx, pos = create_sub_memeplex(i)
        Pb, Pb_LIE = sub_degree_mplx[0][0], sub_degree_mplx[0][1]
        Pw, Pw_LIE = sub_degree_mplx[-1][0], sub_degree_mplx[-1][1]
        temp1 = LDR(Pb, 1)
        temp3 = LDR(Pw, 1)
        if LIE(temp1) > Pw_LIE:
            Pw = temp1
        elif LIE(temp2) > Pw_LIE:
            Pw = temp2
        elif LIE(temp3) > Pw_LIE:
            Pw = temp3
        else:
            Pw = LDR_random()
        Pw_LIE = LIE(Pw)
        sub_degree_mplx[-1] = [Pw, Pw_LIE]

        j = 0
        for p in pos:
            i[p] = sub_degree_mplx[j]
            j += 1

        meme, meme_LIE = [], []
        for p in i:
            meme.append(p[0])
            meme_LIE.append(p[1])
        meme_LIE, meme = sort_memes(meme_LIE, meme)

        for p in range(n):
            i[p] = [meme[p], meme_LIE[p]]
```

6) The description of functions used in (5) are as follows:

```
def sort_nodes(nodes, centrality):
    """
    Sorts the Nodes in the given list of nodes based on the corresponding
Centrality Measure of each node
    Parameters:
        nodes = List of nodes which has to be sorted
        centrality = List of Centrality Measures corresponding to each
node in the list
    Returns:
        List of nodes sorted on the basis of their Centrality Measure
in descending order
    """
    result = sorted(list(zip(centrality, nodes)), reverse = True)
    result = zip(*result)
    result = [list(tuple) for tuple in result]
    return result[1]

def sort_degree_centrality(nodes):
    """
    Calculates the Degree Centrality of each Node in the given list of
nodes and returns a descendingly sorted list of nodes
    Parameters:
        nodes = List of nodes to be sorted on the basis of the Centrality
Measure
    Returns:
        List of Nodes sorted in descending order on the basis of the
Centrality Measure
    """
    centrality = nx.degree_centrality(GOT)
    cen_measure = []
    for i in nodes:
        cen_measure.append(centrality[i])
    return sort_nodes(nodes, cen_measure)

def sort_eigenvector_centrality(nodes):
    """
    Calculates the Eigenvector Centrality of each Node in the given list
of nodes and returns a descendingly sorted list of nodes
    Parameters:
        nodes = List of nodes to be sorted on the basis of the Centrality
Measure
    Returns:
        List of Nodes sorted in descending order on the basis of the
Centrality Measure
    """
    centrality = nx.degree_centrality(GOT)
    cen_measure = []
    for i in nodes:
        cen_measure.append(centrality[i])
    return sort_nodes(nodes, cen_measure)

def sort_betweenness_centrality(nodes):
    """
    Calculates the Betweenness Centrality of each Node in the given list
of nodes and returns a descendingly sorted list of nodes
```

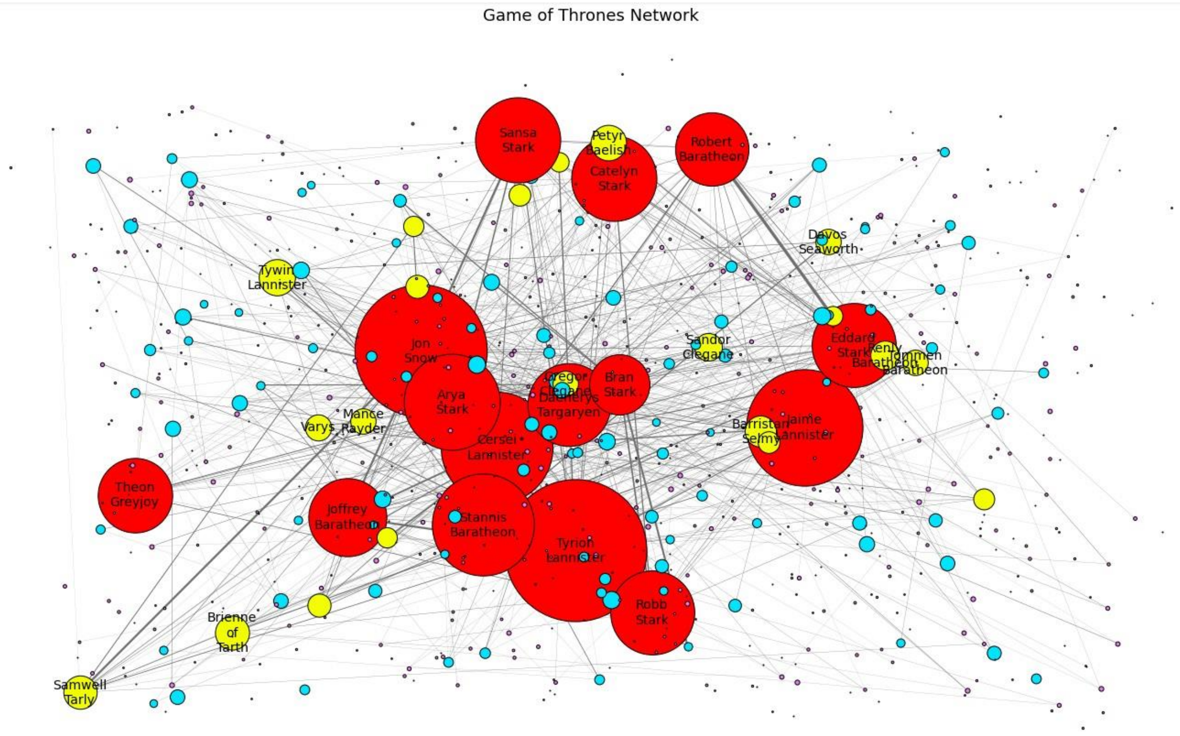
```
    Parameters:
        nodes = List of nodes to be sorted on the basis of the Centrality
Measure
    Returns:
        List of Nodes sorted in descending order on the basis of the
Centrality Measure
    """
    centrality = nx.degree_centrality(GOT)
    cen_measure = []
    for i in nodes:
        cen_measure.append(centrality[i])
    return sort_nodes(nodes, cen_measure)

def sort_closeness_centrality(nodes):
    """
    Calculates the Closeness Centrality of each Node in the given list
of nodes and returns a descendingly sorted list of nodes
    Parameters:
        nodes = List of nodes to be sorted on the basis of the Centrality
Measure
    Returns:
        List of Nodes sorted in descending order on the basis of the
Centrality Measure
    """
    centrality = nx.degree_centrality(GOT)
    cen_measure = []
    for i in nodes:
        cen_measure.append(centrality[i])
    return sort_nodes(nodes, cen_measure)

def sort_katz_centrality(nodes):
    """
    Calculates the Katz Centrality of each Node in the given list of
nodes and returns a descendingly sorted list of nodes
    Parameters:
        nodes = List of nodes to be sorted on the basis of the Centrality
Measure
    Returns:
        List of Nodes sorted in descending order on the basis of the
Centrality Measure
    """
    centrality = nx.degree_centrality(GOT)
    cen_measure = []
    for i in nodes:
        cen_measure.append(centrality[i])
    return sort_nodes(nodes, cen_measure)

def sort_percolation_centrality(nodes):
    """
    Calculates the Percolation Centrality of each Node in the given list
of nodes and returns a descendingly sorted list of nodes
    Parameters:
        nodes = List of nodes to be sorted on the basis of the Centrality
Measure
    Returns:
```

```
        List of Nodes sorted in descending order on the basis of the
Centrality Measure
    """
    centrality = nx.degree_centrality(GOT)
    cen_measure = []
    for i in nodes:
        cen_measure.append(centrality[i])
    return sort_nodes(nodes, cen_measure)

def LDR(meme, flag):
    """
    Calculates the Local-Replacement group for the given meme for the
Evolutionary Algorithm
    Parameters:
        meme = Meme whose Local-Replacement group has to be calculated
        flag = an integer (1-5) to denote the Centrality Measure to be
used for the Evolution Process
                1 - Degree Centrality
                2 - Eigenvector Centrality
                3 - Betweenness Centrality
                4 - Closeness Centrality
                5 - Katz Centrality
                6 - Percolation Centrality
    Returns:
        A List of Top-Centrality nodes equal to the length of the passed
meme
    """
    new_meme = [] # New Meme to improve the worst Meme(Frog) w.r.t. the
Local/Global Best Meme (Frog)
    N1 = []
    for i in meme: # For each Memetype in the Meme
        N1.extend(list(GOT.neighbors(i)))
    N1 = list(set(N1))
    # Sort Function to sort the One Hop Neighbors of the memetype based
on the basis of Centrality Metric
    if flag == 1:
        SN1 = sort_degree_centrality(N1)
    elif flag == 2:
        SN1 = sort_eigenvector_centrality(N1)
    elif flag == 3:
        SN1 = sort_betweenness_centrality(N1)
    elif flag == 4:
        SN1 = sort_closeness_centrality(N1)
    elif flag == 5:
        SN1 = sort_katz_centrality(N1)
    else:
        SN1 = sort_percolation_centrality(N1)
    for i in range(k):
        new_meme.append(SN1[i])
    return new_meme

def LDR_random():
    """
    Calculates a Random Meme for Local-Replacement
    Parameters:
        None
```

```
    Returns:
        A List of randomly selected nodes equal to the length of the
passed meme
    """
    return [random.choice(list(weighted_degrees)) for i in range(k)]

def create_sub_memeplex(memeplex):
    """
    Generates a Sub-memeplex for a given Memeplex
    Parameters:
        memeplex = Memeplex whose Sub-Memeplex has to be calculated
    Returns:
        A Sub-Memeplex
        A List of positions(indices) to locate the corresponding memes
within the memeplex
    """
    sub = []
    for i in range(n):
        sub.append(random.randint(0, n-1))
    sub = list(set(sub))
    sub_memeplex = [memeplex[i] for i in sub]
    return sub_memeplex, sub
```

EXPERIMENTAL RESULTS & ANALYSIS

The Test Runs were made on the Game of Thrones Network which were deduced from the A Book of Ice and Fire and the Network looks like below –



We ran our main code on the Popular Dataset – CondMat (Condensed Matter), Undirected Collaboration Networks for Scientific Collaboration.

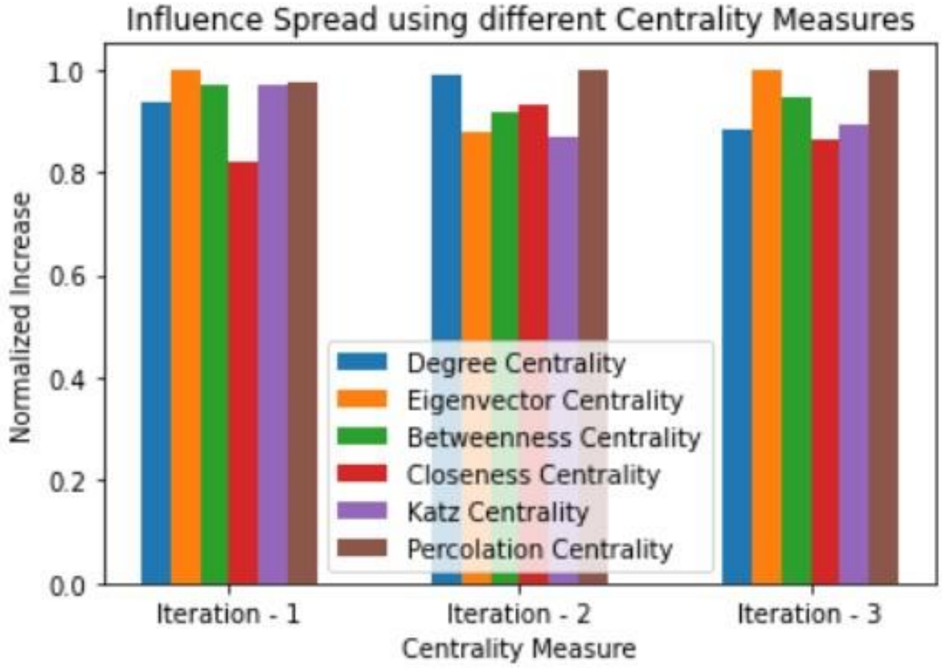
	CondMat [(m = 10), (n = 10), (k = 5), (itr = 1)] (Original Samples)						
	Iteration - 1						
	Initial	Degree	Eigenvector	Betweenness	Closeness	Katz	Percolation
Highest	15.27299371	78.74082382	78.74082382	78.74082382	50.50824548	78.74082382	70.61839808
Average	7.764549854	12.01847954	11.58598711	12.18464591	10.96519352	11.82404124	11.94916871
% Increase (Highest)		415.5559237	415.5559237	415.5559237	230.7029809	415.5559237	362.374302
% Increase (Average)		54.78655896	49.21646874	56.92662348	41.22123916	52.28237905	53.89390158
	Iteration - 2						
	Initial	Degree	Eigenvector	Betweenness	Closeness	Katz	Percolation
Highest	15.30778709	70.61839808	70.61839808	70.61839808	57.56105215	70.61839808	70.61839808
Average	7.939070891	11.94916871	12.16247711	11.50835831	11.8760811	11.80922057	12.00806236
% Increase (Highest)		361.3233621	361.3233621	361.3233621	276.0246456	361.3233621	361.3233621
% Increase (Average)		50.51092093	53.19773904	44.95850294	49.59031433	48.74814367	51.25274139
	Iteration - 3						
	Initial	Degree	Eigenvector	Betweenness	Closeness	Katz	Percolation
Highest	13.75872255	60.00128524	76.37559377	68.76314542	76.37559377	68.76314542	76.37559377
Average	7.681721879	11.70831945	12.74992492	12.26073433	12.49385752	12.08935067	12.15483949
% Increase (Highest)		336.0963383	455.1067223	399.7785599	455.1067223	399.7785599	455.1067223
% Increase (Average)		52.41790367	65.9774348	59.60919334	62.64397114	57.37813547	58.23066346
	Iteration - 4						
	Initial	Degree	Eigenvector	Betweenness	Closeness	Katz	Percolation
Highest	18.78624716	78.23241093	78.23241093	77.78137285	78.23241093	78.23241093	78.23241093
Average	7.857741074	12.52257912	12.97571626	12.17391031	13.17057242	12.41196478	12.30269377
% Increase (Highest)		316.4344813	316.4344813	314.0335863	316.4344813	316.4344813	316.4344813
% Increase (Average)		59.36614615	65.13290705	54.92888094	67.61270571	57.95843436	56.56781828
	Iteration - 5						
	Initial	Degree	Eigenvector	Betweenness	Closeness	Katz	Percolation
Highest	16.87576271	77.30822677	77.30822677	77.30822677	65.39532456	77.30822677	77.30822677
Average	7.742241562	11.92939721	12.30706475	12.28133269	11.68161071	12.42747568	12.29333429
% Increase (Highest)		358.1021201	358.1021201	358.1021201	287.5103346	358.1021201	358.1021201
% Increase (Average)		54.08195558	58.95996852	58.62760922	50.8815066	60.51521481	58.7826238
Mean Highest Increase %		357.5024451	381.3045219	369.7587104	313.1558329	370.2388894	370.6681976
Mean Average Increase %		54.23269706	58.49690363	55.01016198	54.38994739	55.37646147	55.7455497

	CondMat [(m = 10), (n = 5), (k = 3), (itr = 2)]						
	Iteration - 1						
	Initial	Degree	Eigenvector	Betweenness	Closeness	Katz	Percolation
Highest	6.715393135	51.09002735	51.09002735	51.09002735	51.09002735	51.09002735	51.09002735
Average	3.968066749	14.56459002	13.64216791	12.07221838	13.94494097	12.13843687	12.77845906
% Increase (Highest)		660.7898201	660.7898201	660.7898201	660.7898201	660.7898201	660.7898201
% Increase (Average)		267.0449854	243.7988516	204.2342567	251.4290926	205.9030414	222.0323616
	Iteration - 2						
	Initial	Degree	Eigenvector	Betweenness	Closeness	Katz	Percolation
Highest	5.712301953	49.29337549	51.09002735	35.92633847	42.12095927	44.34506033	51.09002735
Average	3.97456017	13.528178	12.81927595	10.46303154	10.8879408	11.09779958	12.559879
% Increase (Highest)		762.9336455	794.385972	528.9292611	637.3727722	676.3080574	794.385972
% Increase (Average)		240.3691835	222.5331962	163.2500476	173.9407717	179.220822	216.0067646
	Iteration - 3						
	Initial	Degree	Eigenvector	Betweenness	Closeness	Katz	Percolation
Highest	6.078199548	51.09002735	49.29337549	49.29337549	51.09002735	49.29337549	51.09002735
Average	4.013294369	11.10093696	10.22998113	11.70878528	11.45835817	10.56517454	9.870309771
% Increase (Highest)		740.5454106	710.9864624	710.9864624	740.5454106	710.9864624	740.5454106
% Increase (Average)		176.6041047	154.9023369	191.7499741	185.5100352	163.2544131	145.9403389
	Iteration - 4						
	Initial	Degree	Eigenvector	Betweenness	Closeness	Katz	Percolation
Highest	7.591844063	51.09002735	51.09002735	51.09002735	51.09002735	51.09002735	51.09002735
Average	3.899278245	13.11980821	13.64174882	14.17945652	13.24806494	13.7623003	13.10107912
% Increase (Highest)		572.9593881	572.9593881	572.9593881	572.9593881	572.9593881	572.9593881
% Increase (Average)		236.4676073	249.8531771	263.6431059	239.75685	252.9448127	235.9872853
	Iteration - 5						
	Initial	Degree	Eigenvector	Betweenness	Closeness	Katz	Percolation
Highest	5.70558027	51.09002735	28.07701408	51.09002735	46.28684388	32.59554887	51.09002735
Average	3.847879648	10.91154485	9.26872822	12.57207881	9.847392009	10.38992978	9.941303544
% Increase (Highest)		795.4396385	392.0974336	795.4396385	711.2556775	471.2924422	795.4396385
% Increase (Average)		183.5729245	140.8788493	226.7274437	155.9173599	170.0170153	158.3579647
Mean Highest Increase %		706.5335806	626.2438152	653.820914	664.5846137	618.467234	712.8240459
Mean Average Increase %		141.3067161	125.248763	130.7641828	132.9169227	123.6934468	142.5648092

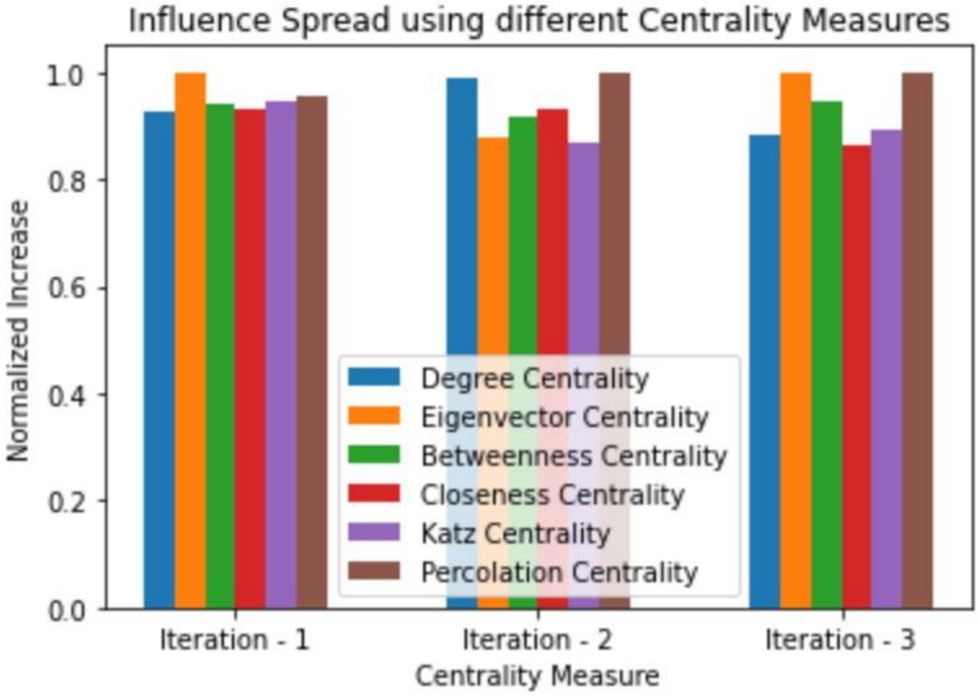
	CondMat [(m = 10), (n = 10), (k = 3), (itr = 1)] (Samples = 75000)						
	Iteration - 1						
	Initial	Degree	Eigenvector	Betweenness	Closeness	Katz	Percolation
Highest	11.59831776	46.28684388	46.28684388	46.28684388	31.55052324	31.13496099	46.28684388
Average	3.953387242	7.337053329	7.49593049	6.850003587	6.634343614	6.504016579	7.228321524
% Increase (Highest)		299.0823914	299.0823914	299.0823914	172.0267188	168.443766	299.0823914
% Increase (Average)		85.58903747	89.60779786	73.26922885	67.81416057	64.51756889	82.83869204
	Iteration - 2						
	Initial	Degree	Eigenvector	Betweenness	Closeness	Katz	Percolation
Highest	8.961989342	27.55311192	27.55311192	27.55311192	27.55311192	24.63959308	27.55311192
Average	3.94677041	7.0519882	6.741382179	6.829207406	6.685820803	6.091784846	6.82977566
% Increase (Highest)		207.4441496	207.4441496	207.4441496	207.4441496	174.9344162	207.4441496
% Increase (Average)		78.67743665	70.80755854	73.0328014	69.3997904	54.3485993	73.04719936
	Iteration - 3						
	Initial	Degree	Eigenvector	Betweenness	Closeness	Katz	Percolation
Highest	7.887958387	23.69416206	32.23727684	32.23727684	32.23727684	32.23727684	32.23727684
Average	3.919963723	6.200294869	6.692500203	6.488018304	6.778396794	6.026864859	6.356115384
% Increase (Highest)		200.3839637	308.6897428	308.6897428	308.6897428	308.6897428	308.6897428
% Increase (Average)		58.17225127	70.72862599	65.51220273	72.91988582	53.74797537	62.14730119
	Iteration - 4						
	Initial	Degree	Eigenvector	Betweenness	Closeness	Katz	Percolation
Highest	6.934621249	25.09726084	30.78198329	25.09726084	25.09726084	30.78198329	30.78198329
Average	4.022600915	7.039045677	6.862679084	6.891889766	6.591953619	6.803455939	7.083771973
% Increase (Highest)		261.9124959	343.8884574	261.9124959	261.9124959	343.8884574	343.8884574
% Increase (Average)		74.98742299	70.60303098	71.32919503	63.872921	69.13077095	76.09929803
	Iteration - 5						
	Initial	Degree	Eigenvector	Betweenness	Closeness	Katz	Percolation
Highest	6.315998733	30.55874902	29.93452998	29.93452998	29.93452998	29.93452998	29.93452998
Average	3.975220553	6.026093941	6.606239033	6.396986216	6.618958562	6.200494296	6.286597111
% Increase (Highest)		383.8308288	373.9476881	373.9476881	373.9476881	373.9476881	373.9476881
% Increase (Average)		51.5914365	66.18547182	60.92154209	66.50544222	55.97862339	58.14461176
Mean Highest Increase %		270.5307659	306.6104859	290.2152936	264.804159	273.9808141	306.6104859
Mean Average Increase %		54.10615318	61.32209718	58.04305871	52.96083181	54.79616282	61.32209718

On the basis of the results presented in the above tables, it can be clearly seen that although Degree Centrality has a good convergence and results, Eigenvector Centrality poses itself as the best Centrality measure.

The Mean Highest increase % can be seen in the below bar graph.



Similarly, the Mean Average Increase % has been depicted in the table below



CONCLUSION

In Conclusion, it can be deduced that Discrete Shuffled Frog-Leaping Algorithm is a very optimized algorithm to find the K-most influential Nodes in a Network Graph. The Original approach to use Degree Centrality can be replaced with Eigenvector Centrality as it can find the themselves most influential nodes and even those who are connected to such types of nodes.

Adding to the improvement of the code, it can also be said that giving the Worst meme a chance again after Local Best and Global Best Replacement groups cannot suffice the LIE value improvement can significantly improve the performance to a great measure.

FUTURE SCOPE

The project mainly focuses on designing an algorithm that identifies nodes with maximum influence, but can be expanded to include Budgets and Costs where the cost to engage with an influential node and the allotted budget can be kept in mind to design an algorithm to cater to the factors.

REFERENCES

1. Jianxin Tang, Ruisheng Zhang, Ping Wang, Zhili Zhao, Li Fan, Xin Liu, A discrete shuffled frog-leaping algorithm to identify influential nodes for influence maximization in social networks, Knowledge-Based Systems, Volume 187, 2020, 104833, ISSN 0950-7051, https://doi.org/10.1016/j.knosys.2019.07.004. (https://www.sciencedirect.com/science/article/pii/S0950705119303089)
2. Banerjee, S., Jenamani, M. & Pratihar, D.K. A survey on influence maximization in a social network. *Knowl Inf Syst* 62, 3417–3455 (2020). https://doi.org/10.1007/s10115-020-01461-4
3. https://towardsdatascience.com/a-survey-on-shuffled-frog-leaping-algorithm-d309d0cf7503
4. Maaroof, B.B., Rashid, T.A., Abdulla, J.M. *et al.* Current Studies and Applications of Shuffled Frog Leaping Algorithm: A Review. *Arch Computat Methods Eng* (2022). https://doi.org/10.1007/s11831-021-09707-2
5. https://www.tandfonline.com/doi/abs/10.1080/03052150500384759#:~:text=A%20memetic%20meta%2Dheuristic%20called,search%20and%20global%20information%20exchange.