NIT Silchar

**Department of Computer Science and**

**Engineering**

****

Social Network Analysis

**CS 331**

B.Tech (CSE) Sixth Semester

(January 2023 – June 2023)

**MINI – PROJECT REPORT**

**Mini – Project Title**

Localization of multiple diffusion sources based on overlapping community detection

Research Papers Implemented:

1. Localization of multiple diffusion sources based on overlapping community detection (2021)

**Team Member Details**

Team Leader: Vikas Suthar 2012085

Adithya Sirivella 2012119

Bishal Karmakar 2012084

Rishav Bora 2012080

Pronab Bhattacharjee 2012169

Nitesh Kumar 2012068

Tushar Rathi 2012174

**Individual Contributions**

Adithya Sirivella (2012119):

* Implementation & Coding of “Diffusion source number estimation”
* Report Formation
* Research

Vikas Suthar (2012085):

* Implementation & Coding of “Diffusion source number estimation”
* Implementation & Coding of “Infected graph partition based on the topological potential field”
* Implementation & Coding of “Single source localization algorithm in each partition”

Bishal Karmakar (2012084):

* Overviewing of the Research Paper
* Synthesizing of the PowerPoint presentation
* Report Formation

Pronab Bhattacharjee (2012169):

* Presentation

Rishav Bora (2012080):

* Overviewing of the Research Paper
* Synthesizing of the PowerPoint presentation
* Prepared short-notes and methodology charts

Nitesh Kumar (2012068):

* Presentation

Tushar Rathi (2012174):

* Implementation & Coding of “Infected graph partition based on the topological potential field”
* Synthesizing of the PowerPoint presentation
* Prepared short-notes and methodology charts

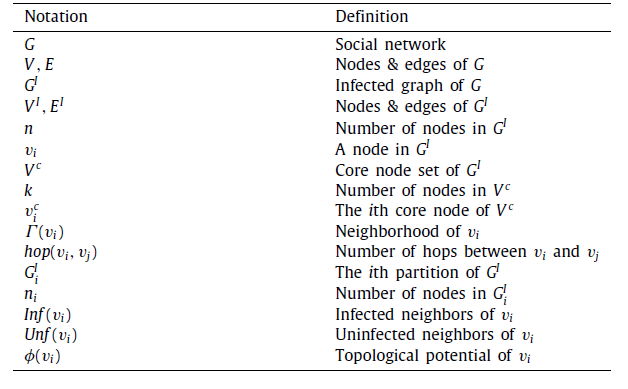
**Abstract/Summary**

1. **Prerequisites:**

Diffusion and propagation are fundamental processes that occur in nature and society, from infectious diseases to computer viruses and rumours. Locating diffusion sources is important for various practical applications, such as controlling the spread of diseases or identifying the sources of rumours. Identifying a single source is often done using centrality measures or other methods such as maximum-likelihood estimation or interactive query. However, there are typically multiple sources in real-world scenarios, and locating them is a more significant and practical task than identifying a single source.

A novel approach is proposed that divides the infected graph using the topological potential field, which captures the information diffusion dynamics. By partitioning the infected graph into a series of local high potential areas, the multiple sources locating problem can be transformed into several single source locating problems.

Notations used in this report:



1. **Diffusion source number estimation**
   1. **Introduction:**

In this Algorithm we locate the multiple diffusion sources in a method consisting

of three main steps. Firstly, it estimates the number of diffusion sources that may exist in the infected graph. Secondly, based on the number of sources, it divides the infected graph into the corresponding number of overlapping partitions. Finally, it locates the single diffusion source individually in each divided partition.

Required Formulae:

The impact factor controls the impact scope of each field source and thus determines the distribution of topological potentials. If σ is too small, each field source can only impact a limited scope of nodes. When σ is close to 0, each source can only have an impact on themselves. If σ is too large, every source would impact all nodes in the network, which is not reasonable in the real-world scenarios. Therefore, an optimal σ is needed to control the impact scope and thereby truly reflect the structural characteristics of the infected graph. Here the impact factor is dependent on the dataset used. The impact factor for Zachary’s Karate Club Network is 1.02.

* 1. **Algorithm:**

This algorithm aims to estimate the number of sources in a diffusion process on a graph. The diffusion process is modelled as an infected graph GI with vertices VI and edges EI. The output of the algorithm is an estimate of the number of sources (k) and a set of core nodes with local maximum potential (VC).

The algorithm works as follows:

1. Initializes the core node set VC to an empty set.
2. Obtains the optimal impact factor σ using potential entropy method.
3. Calculates the potential φ(vi) of each vertex vi in the graph using Topological Potential Formula.
4. Identifies local maximum potential nodes in the graph by comparing the potential of each vertex with its neighbours. If the potential of a vertex is greater than its neighbours, then it is considered a local maximum potential node and added to VC.
5. Continues to refine the set of core nodes in VC by removing nodes that are too close together. Two nodes are considered too close if their hop distance is less than ⌊3σ/√2⌋, where hop(vi, vj) represents the number of hops between vi and vj.
6. Returns the estimated number of sources k as the cardinality of the core node set VC.
7. **Infected graph partition based on the topological potential field.** 
   1. **Introduction:**

If the given infected graph GI and the corresponding topological potential field. Suppose there is a node vi ∈ VI, then within this graph, there are three types of nodes: peak, valley, and slope.

* A peak node is a node that has the highest potential value among all its neighbouring nodes.
* A valley node is a node that has the lowest potential value among all its neighbouring nodes.
* A slope node is a node that has both a neighbouring node with a higher potential value and a neighbouring node with a lower potential value.

Each high potential area corresponds to a partition and the peak node is the central node of this partition. Peak and slope nodes are internal nodes of their respective partitions. Neighbouring partitions are connected by valley nodes, which are nodes that overlap adjacent partitions.

If the peak nodes of two neighbouring high potential areas are too close, they will be combined into one partition, and the valley nodes in these combined areas will also be internal nodes rather than overlapping nodes.

The algorithm starts by initializing each overlapping partition to an empty set and adding the core node to the ith partition. Then, for each neighbour of the core node, if the neighbour is not already in the overlapping partition, the Function(vj, ) is called.

Function(vj, ) is called when a new node is found during the expansion. If the node satisfies the Valley, it is considered an overlapping node and is added to the partition. Local expansion is terminated from the node. If the node satisfies the Slope, it is considered an internal node and is added to the partition. Then, for each neighbour of the node, if the neighbour is not already in the overlapping partition, Function(vt, ) is called recursively.

The algorithm terminates once all the nodes in the infected graph are included in one or more overlapping partitions.

* 1. **Algorithm:**

This algorithm partitions an infected graph based on the topological potential field. The input is the infected graph GI, which consists of nodes VI and edges EI, and the set of core nodes VC. The output is the overlapping partitions of GI.

Definition:

• if ∀ vj ∈ Γ (vi), φ(vi) ≥ φ(vj), then vi is a peak node.

• if ∀ vj ∈ Γ (vi), φ(vi) ≤ φ(vj), then vi is a valley node.

• if ∃ vj ∈ Γ (vi), φ(vi) > φ(vj) and ∃ vj ∈ Γ (vi), φ(vi) < φ(vj), then vi is a slope node.

1. Set k to the number of core nodes in VC.
2. For each core node in VC, create an empty overlapping partition for the ith partition of GI.
3. Add the ith core node VC to .
4. For each neighbour node vj of VC in Γ(VC), do the following:

If vj is not already in , call the Function(vj, ) procedure.

If vj satisfies the Valley condition in Definition, then it is an overlapping node.

Add vj to .

1. Terminate the local expansion from vj.
2. If vj satisfies the Slope condition in Definition, then it is an internal node.
3. Add vj to .
4. For each neighbour node vt of vj in Γ(vj), do the following:

If vt is not already in , call the Function(vt, ) procedure.

The Function(vj, ) procedure performs the following steps:

If vj satisfies the Valley condition in Definition, then it is an overlapping node.

1. Add vj to .
2. Terminate the local expansion from vj.
3. If vj satisfies the Slope condition in Definition, then it is an internal node.
4. Add vj to .
5. For each neighbour node vt of vj in Γ(vj), do the following:

If vt is not already in , call the Function(vt, ) procedure.

1. **Single source localization algorithm in each partition.**
   1. **Introduction:**

This is a single source localization algorithm that aims to find the most

probable single source of an overlapping partition in a graph. The algorithm calculates the probability of each node in the overlapping partition being the source, based on the topological potential field value, BFS tree rooted at the node, and the probability of the overlapping partition given the node. It then selects the node with the highest probability as the single source of the overlapping partition.

Required Formulae:

* 1. **Algorithm:**

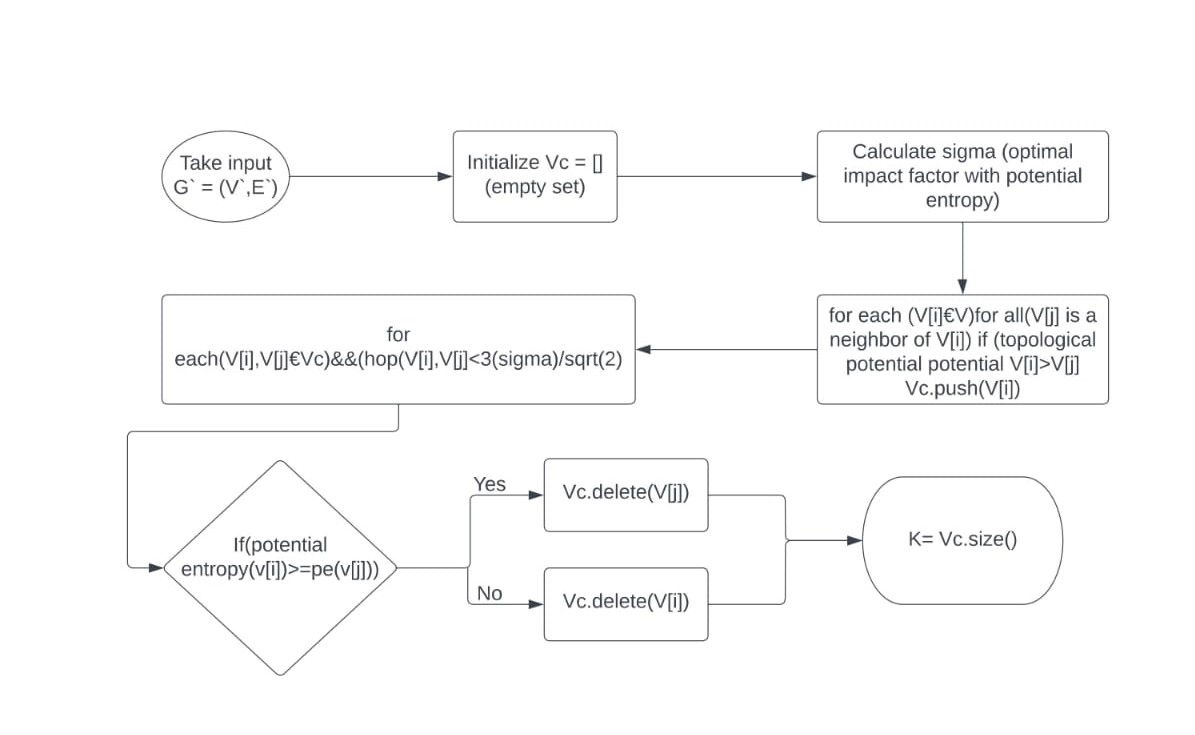
This is an algorithm that tries to find the location of a single source in a specific area, which is divided into smaller overlapping parts.

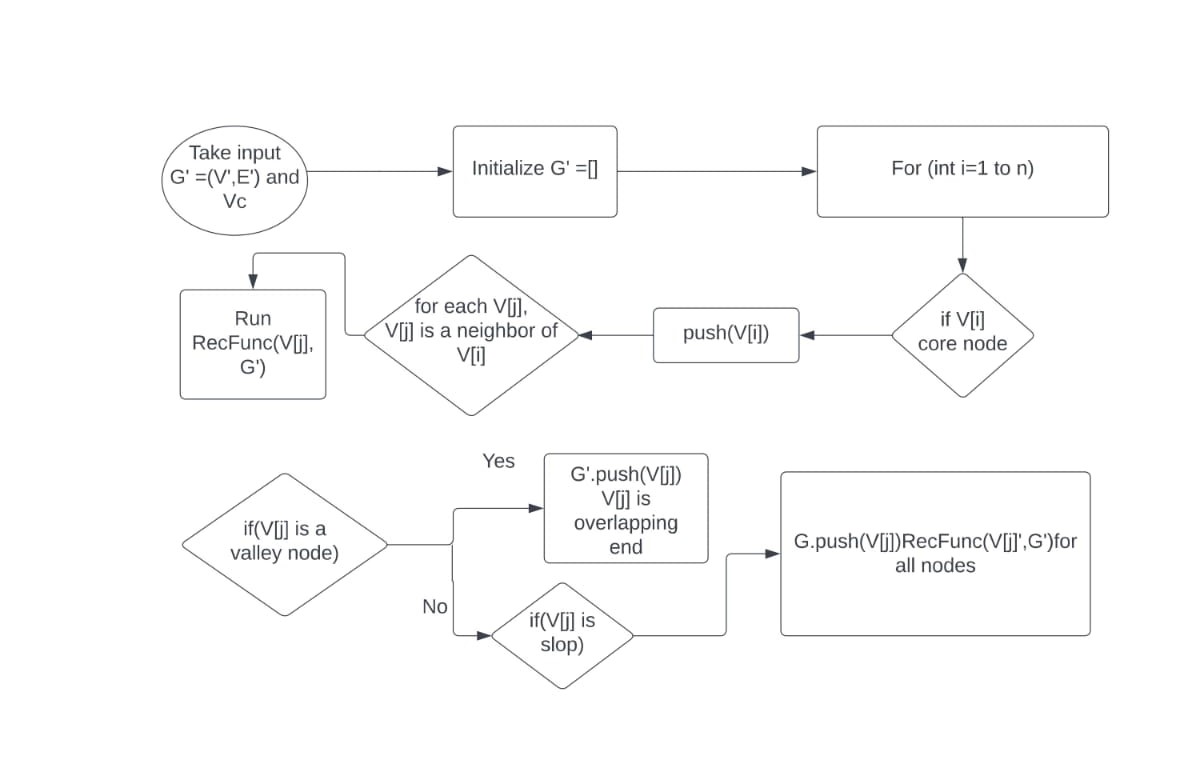
The algorithm works by taking each part of the area and performing the following steps:

1. Count how many nodes (represented by " ni ") are in the current part.
2. For each node " vi " in the part, generate a BFS (breadth-first search) tree starting at that node.
3. Calculate "β(vi)”.
4. Calculate the probability of the current part of the area given that the source is at node " vi " using Likelihood Probability.
5. Calculate the probability of node " vi " being the source given the current part of the area using Integrated Centrality.
6. Repeat steps 2 to 5 for all nodes in the current part.
7. Return the node with the highest probability of being the source in the current part.

Overall, the algorithm tries to find the most likely node that the single source is located in each of the overlapping parts of the area.

**Methodology**

****

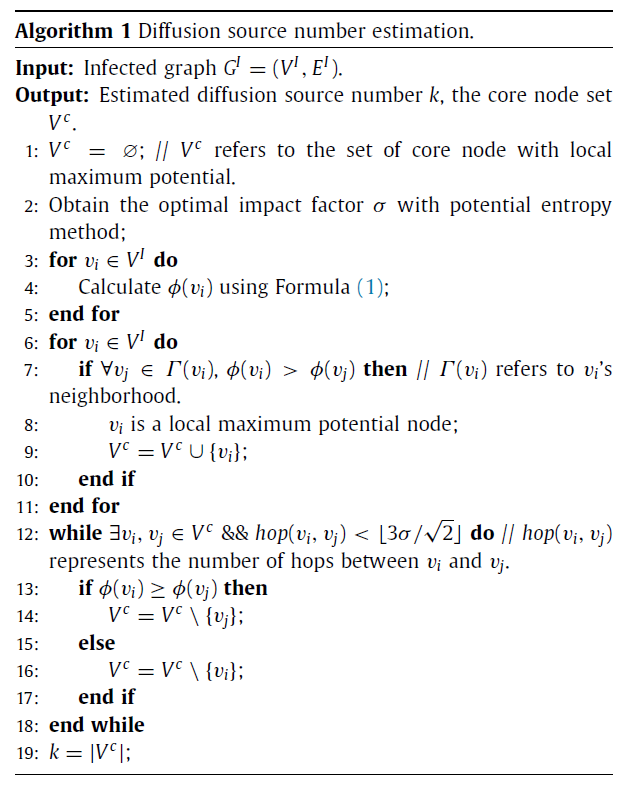
****

**Software Requirements and Specifications**

* PyCharm IDE 2023 Community Edition
* Python Version 3.10.4
* Required Libraries:
  + Networkx
  + Matplotlib
  + random
  + math
  + ndlib
* Conda (Optional)

**Source Code**

1. Diffusion Source Number Estimation:



Code:

from math import exp, sqrt, floor, log, factorial

import ndlib.models.epidemics as ep

import networkx as nx

import random

import matplotlib.pyplot as plt

from matplotlib.colors import ListedColormap

from typing import List, Tuple

phi = {}

ck = None

# Implementation of Algorithm 1 as Given in the paper

def estimate\_diffusion\_source\_number(GI):

# Step 1: Initialize core node set Vc

Vc = set()

# Step 2: Obtain the optimal impact factor sigma with potential entropy method

sigma = get\_optimal\_sigma(GI)

# Step 3: Calculate the topological potential for each node in VI

for vi in GI.nodes():

phi[vi] = sum([G.degree(vj) \* exp(-(nx.shortest\_path\_length(G, vi, vj) / sigma) \*\* 2) for vj in GI.nodes()])

# Step 4: Find local maximum potential nodes in VI

for vi in GI.nodes():

if all(phi[vi] > phi[vj] for vj in GI.neighbors(vi)):

Vc.add(vi)

# print(Vc)

# Step 5: Refine Vc using hop distance criteria

while True:

found\_swap = False

for vi in Vc:

for vj in Vc:

if vi == vj:

continue

if nx.shortest\_path\_length(G, vi, vj) < floor(3 \* sigma / sqrt(2)):

if phi[vi] >= phi[vj]:

Vc.discard(vj)

found\_swap = True

else:

Vc.discard(vi)

found\_swap = True

if not found\_swap:

break

# Step 6: Return the estimated diffusion source number k and the core node set Vc

k = len(Vc)

return k, Vc

def get\_optimal\_sigma(GI):

# Initialize the set of candidate sigmas

candidate\_sigmas = [0.01 \* rp for rp in range(1, 101)]

# Calculate entropy for each sigma in candidate\_sigmas

entropy\_values = []

for sigma in candidate\_sigmas:

phi = {}

for vi in GI.nodes():

phi[vi] = sum([G.degree(vj) \* exp(-(nx.shortest\_path\_length(G, vi, vj) / sigma) \*\* 2) for vj in GI.nodes()])

entropy = -sum([phi[vi] \* log(phi[vi]) for vi in GI.nodes() if phi[vi] != 0])

entropy\_values.append(entropy)

# Find the sigma that minimizes entropy

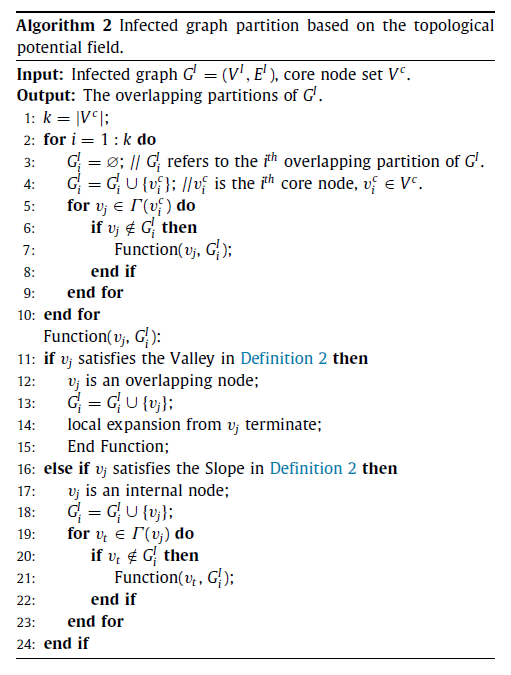
min\_entropy = min(entropy\_values)

min\_entropy\_index = entropy\_values.index(min\_entropy)

optimal\_sigma = candidate\_sigmas[min\_entropy\_index]

return optimal\_sigma

1. Infected Graph Partition Based on the Topological Potential Field:



Code:

# Implementation of Algorithm 2 to get partition on the basis of peak and valley

def get\_partition(GI, Vcc):

partitions = []

for i in Vcc:

GIi = nx.Graph()

vci = i

GIi.add\_node(vci, state=True)

for vj in GI.neighbors(vci):

if vj not in GIi.nodes:

partition\_node(vci, vj, GIi, GI)

partitions.append(GIi)

return partitions

def partition\_node(vci, vj, GIi, GI):

if all(phi[vj] <= phi[neighbors] for neighbors in GI.neighbors(vj)):

GIi.add\_node(vj, state=True)

GIi.add\_edge(vci, vj)

return

if any(phi[vj] > phi[neighbors] for neighbors in GI.neighbors(vj)) and any(

phi[vj] < phi[neighbors] for neighbors in GI.neighbors(vj)):

GIi.add\_node(vj, state=True)

GIi.add\_edge(vci, vj)

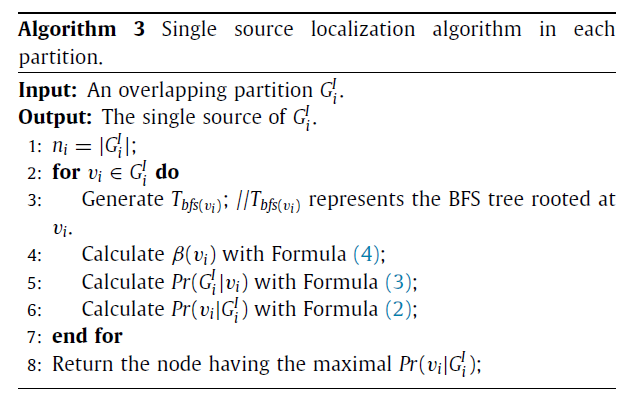
for vt in GI.neighbors(vj):

if vt not in GIi.nodes:

partition\_node(vj, vt, GIi, GI)

return

1. Single Source Localization Algorithm in each Partition:



Code:

# Function to generate BFS tree rooted at a node

def generate\_Tbfs(G, root):

Tbfs = nx.bfs\_tree(G, root)

return Tbfs

def bfs(graph, start):

visited = set()

queue = []

nn = graph.number\_of\_nodes()

# print(nn)

node\_colors = ['none' for \_ in range(nn)]

col = 0

# print(start)

for s in start:

visited.add(s)

node\_colors[s] = plt.get\_cmap('tab20')(col)

queue.append(s)

col += 1

# print(colored\_nodes)

while queue:

node = queue.pop(0)

for neighbor in graph.neighbors(node):

if neighbor not in visited:

queue.append(neighbor)

node\_colors[neighbor] = node\_colors[node]

visited.add(neighbor)

# print(node\_colors)

pos = nx.spring\_layout(graph)

nx.draw(graph, pos, node\_color=node\_colors, with\_labels=True)

plt.show()

# Function to calculate β(vi)

def calculate\_beta(G, vi):

inf\_neighbors = [n for n in G.neighbors(vi) if G.nodes[n]['state'] == 1]

unf\_neighbors = [n for n in G.neighbors(vi) if G.nodes[n]['state'] == 0]

ni = len(G.nodes)

beta = (len(inf\_neighbors) / (len(inf\_neighbors) + len(unf\_neighbors))) \*\* ni

return beta

def get\_all\_sources(infectedPartitions):

sources = []

for partition in infectedPartitions:

sources = find\_single\_source(partition)

sources = Vcc

return sources

# Function to calculate Pr(GIi|vi)

def calculate\_Pr\_GIi\_given\_vi(GIi, vi):

ni = len(GIi.nodes)

Pr\_GIi\_given\_vi = factorial(ni)

for u in GIi.nodes:

Tbfs = generate\_Tbfs(GIi, vi)

subtree\_nodes = nx.descendants(Tbfs, u) | {u}

Pr\_GIi\_given\_vi \*= 1 / len(subtree\_nodes)

return Pr\_GIi\_given\_vi

# Function to calculate Pr(vi|GIi)

def calculate\_Pr\_vi\_given\_GIi(GIi, vi):

Pr\_GIi\_given\_vi = calculate\_Pr\_GIi\_given\_vi(GIi, vi)

beta = calculate\_beta(GIi, vi)

Pr\_vi\_given\_GIi = Pr\_GIi\_given\_vi \* beta

return Pr\_vi\_given\_GIi

# Function to find the single source of GIi

def find\_single\_source(GIi):

max\_pr = -1

single\_source = None

for vi in GIi.nodes:

pr\_vi\_given\_GIi = calculate\_Pr\_vi\_given\_GIi(GIi, vi)

if pr\_vi\_given\_GIi > max\_pr:

max\_pr = pr\_vi\_given\_GIi

single\_source = vi

return single\_source

# Creation of graph

G = nx.connected\_watts\_strogatz\_graph(60, 4, 0.2)

# Initialize the state of each node

# All nodes are initially susceptible

node\_states = {}

for node in G.nodes:

node\_states[node] = 'green'

for node in G.nodes():

G.nodes[node]['state'] = 0

n = 5

k = 2

initial\_node = random.sample(list(G.nodes), n)

print("initial Nodes: ", initial\_node)

no = random.randint(n - k, n)

selected\_nodes = []

selected\_nodes.extend(initial\_node)

remaining\_nodes = list()

for i in initial\_node:

neighbors = list(G.neighbors(i))

random.shuffle(neighbors)

remaining\_nodes.extend(neighbors[:k])

G.nodes[i]['state'] = 1

node\_states[i] = 'red'

selected\_nodes.extend(random.sample(remaining\_nodes, k))

Vcc = random.sample(list(selected\_nodes), no)

# Define the parameters of the model

p\_infect = 0.2 # Probability of infecting a neighbor

p\_recover = 0 # Probability of recovering

# Simulate the diffusion of the rumor

t = 0

while True:

cascade\_ended = True

plt.figure()

pos = nx.spring\_layout(G)

plt.title(f'Time stamp: {t}')

nx.draw(G, pos, node\_color=[node\_states[node] for node in G.nodes], with\_labels=True)

plt.show()

t += 1

for node in G.nodes():

if G.nodes[node]['state'] == 1:

for neighbor in G.neighbors(node):

if G.nodes[neighbor]['state'] == 0:

# Node adopts the information with probability p

if random.random() < p\_infect:

G.nodes[neighbor]['state'] = 1

node\_states[neighbor] = 'red'

cascade\_ended = False

if cascade\_ended:

break

plt.figure()

pos = nx.spring\_layout(G)

plt.title(f'Final Time stamp: {t}')

nx.draw\_networkx(G, pos, node\_color=[node\_states[node] for node in G.nodes], with\_labels=True)

plt.show()

k, Vc = estimate\_diffusion\_source\_number(G)

# print("Estimated diffusion source number:", k)

# print("Core node set:", Vc)

# print(initial\_node)

# bfs(G, initial\_node)

# for i in range(len(phi)):

# print("i :", i, "phi: ", phi[i])

infectedPartitions = get\_partition(G, Vc)

bfs(G, Vcc)

color\_map = {}

for i, p in enumerate(infectedPartitions):

for n in p:

color\_map[n] = i

# nx.draw(G, pos, node\_color=[color\_map[n] for n in G.nodes()], with\_labels=True)

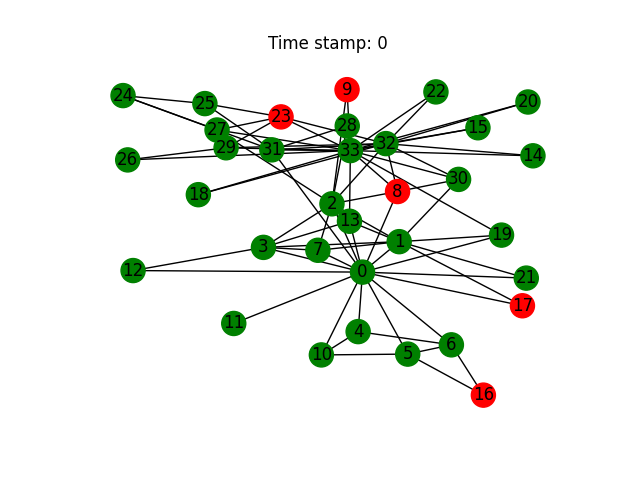
# plt.show()

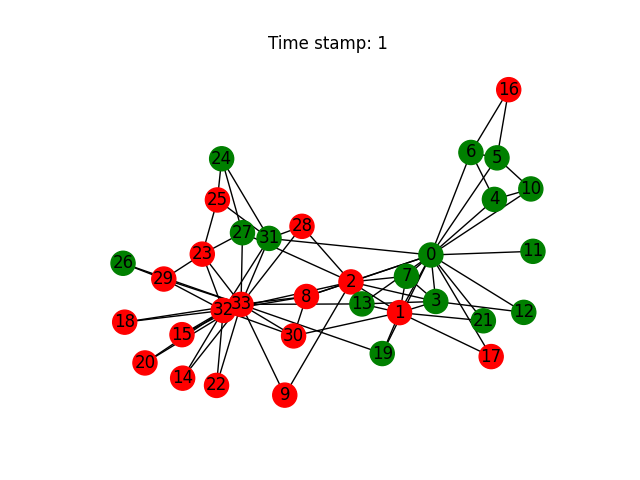
print("final: ", get\_all\_sources(infectedPartitions))

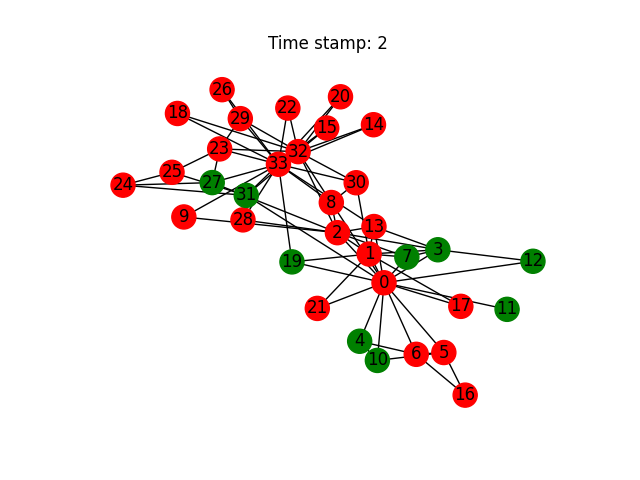
**Results**

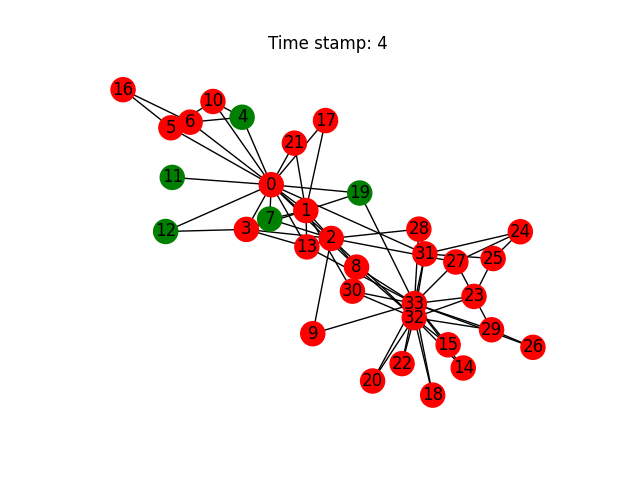
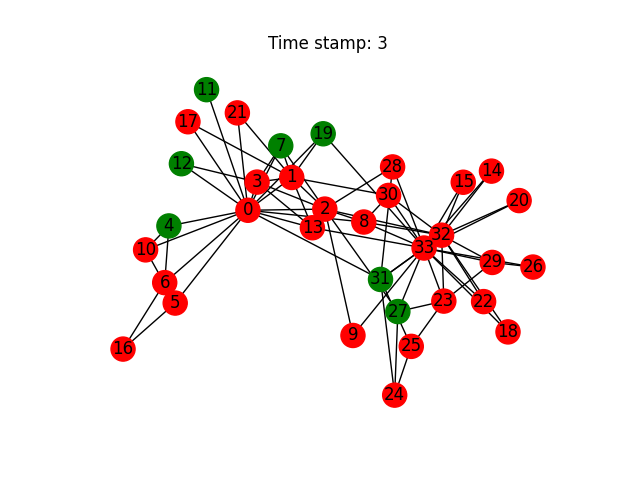
Output and Graphs for the above code:

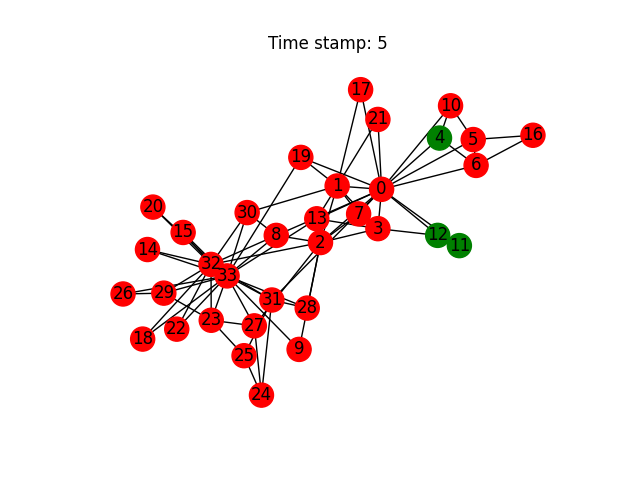
This Output is for the “Zachary’s Karate Club Network” dataset.

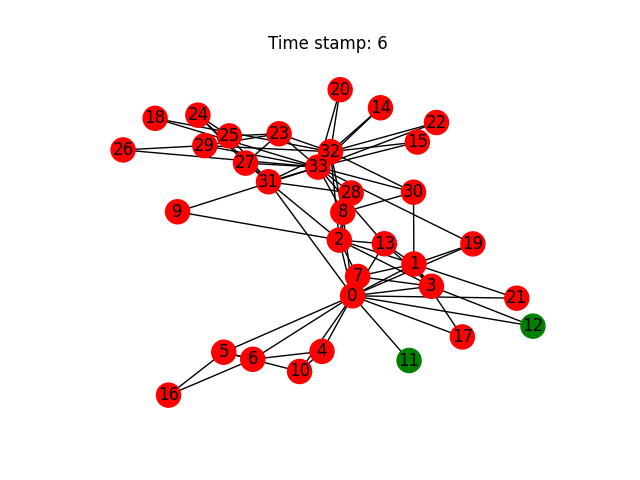
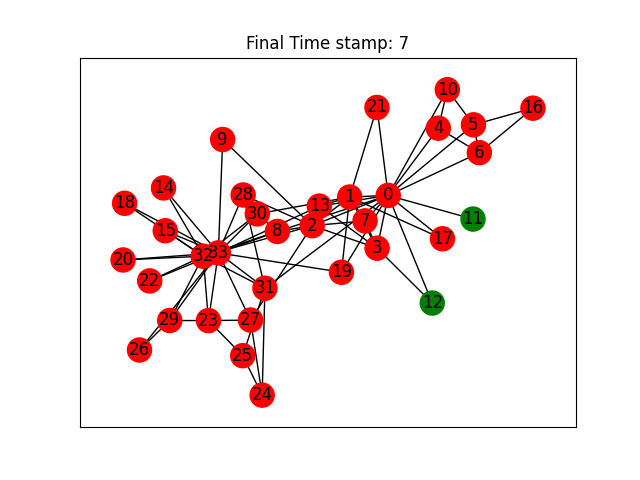


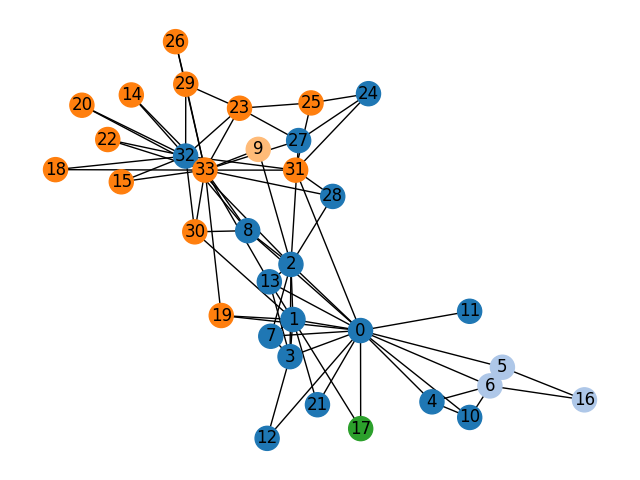


****

****

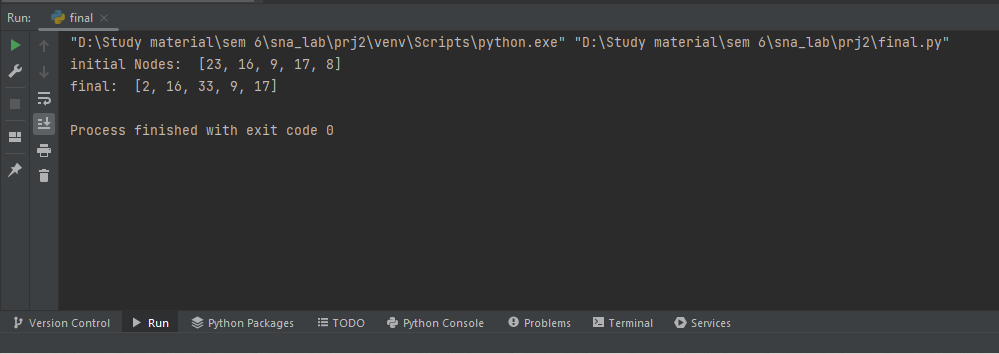
****

****

****

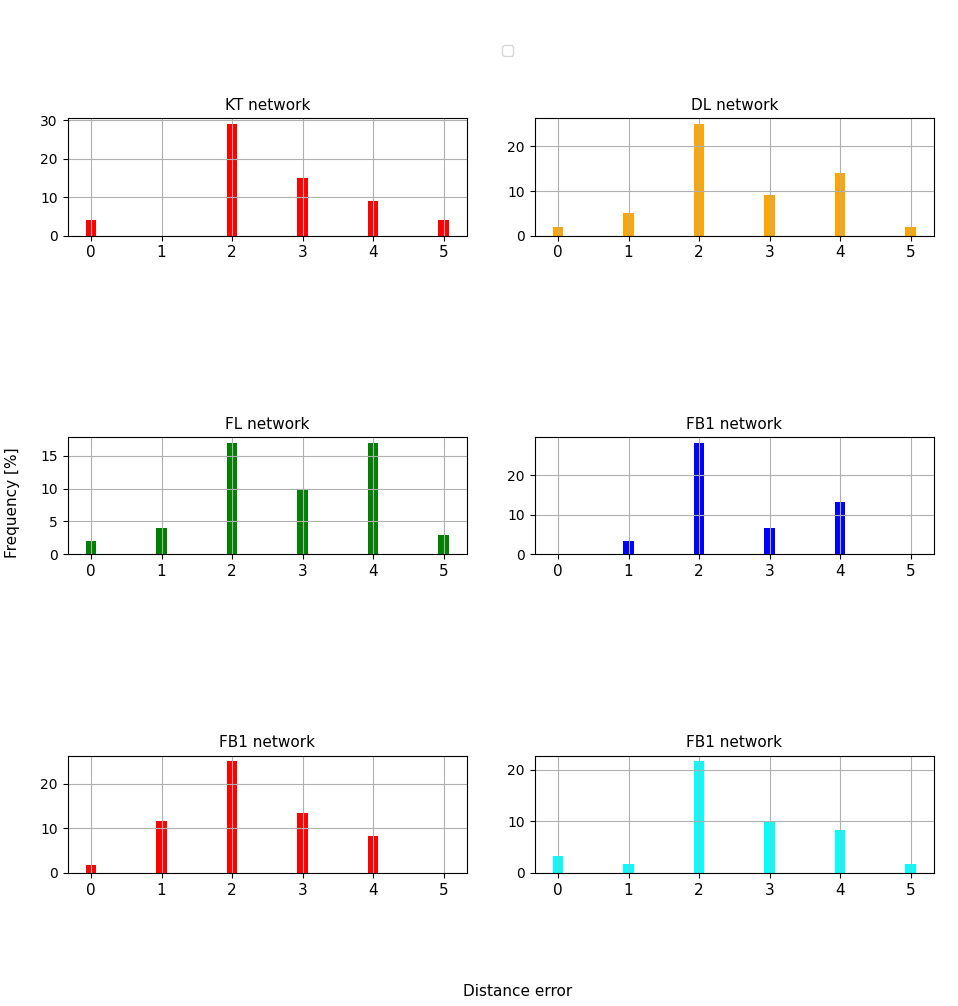
Infected Graph Partition (Each partition has exactly one source)

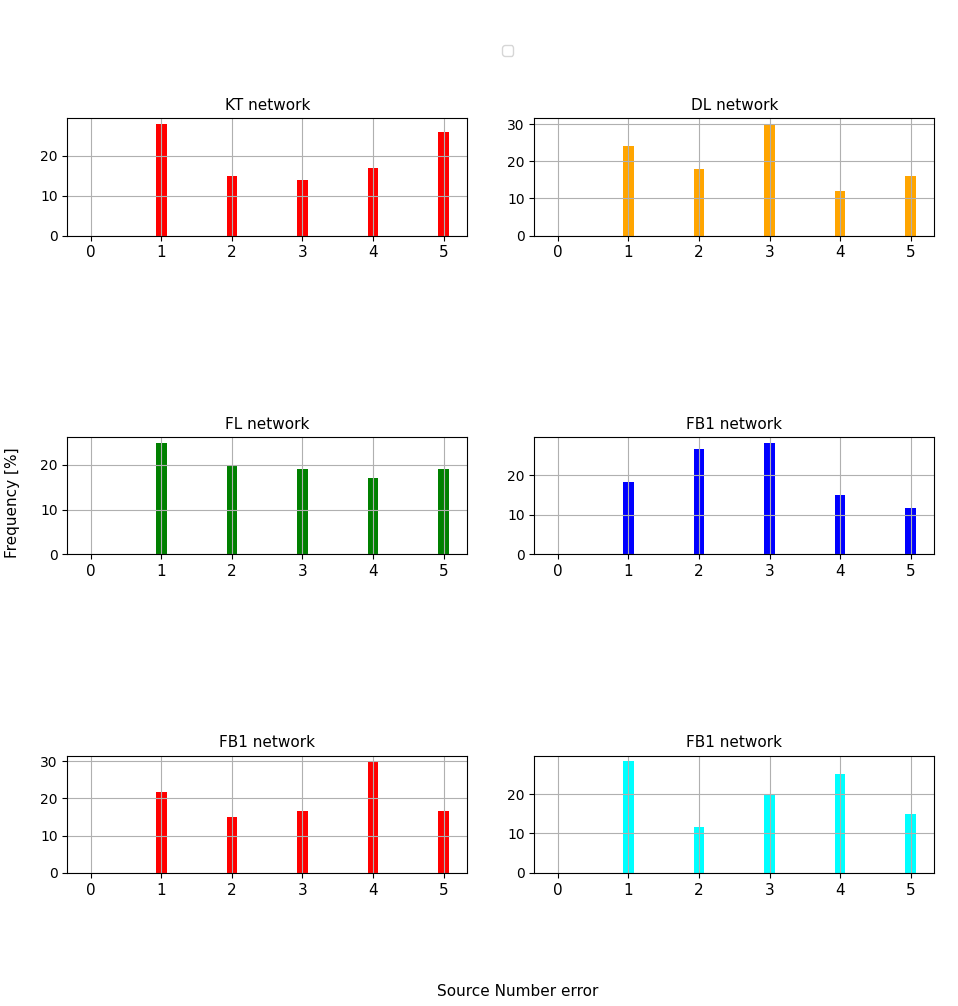
The Final Output:

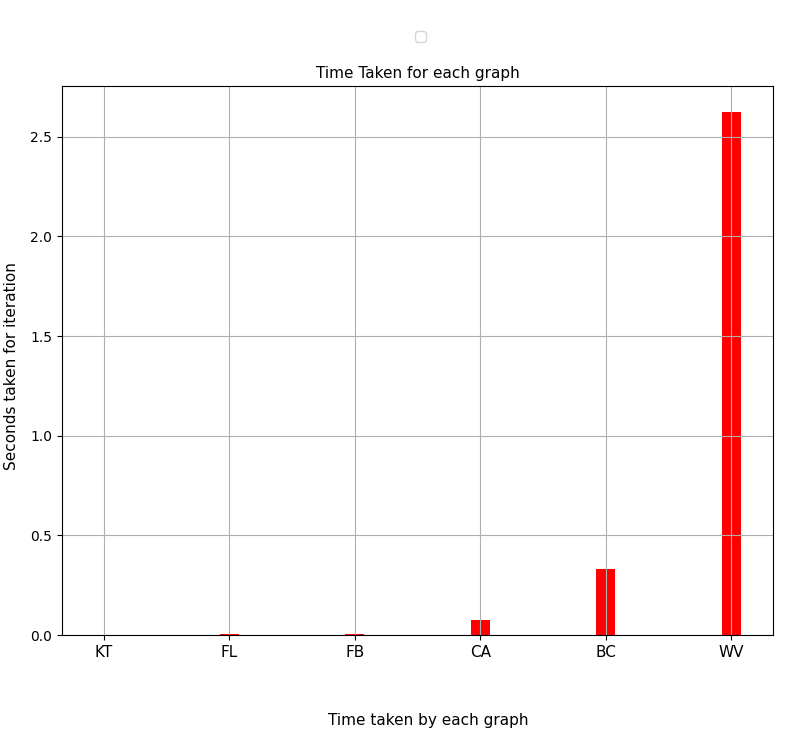


**Result Analysis**

**Data Visualization:**







**Conclusion**

As seen in the results the algorithm works well in most cases. The Algorithm needs to optimised more to ensure we obtain more accurate results i.e., source nodes estimation. The major constraint for this is that the topological potential field needs to obtain the shortest path for each pair of nodes which is time consuming even with high end hardware.

**References**

* Wang, Zhixiao & Sun, Chengcheng & Rui, Xiaobin & Yu, Philip & Sun, Lichao. (2020). Localization of multiple diffusion sources based on overlapping community detection. Knowledge-Based Systems. 226. 106613. 10.1016/j.knosys.2020.106613.
* J.-P. Zhang, H.-B. Li, J. Yang, J.-B. Bai, L.-J. Zhang, Y. Chu, Variable scale network overlapping community identification based on identity uncertainty, Dianzi Xuebao (Acta Electron. Sin.) 40 (12) (2012) 2512–2518.
* Z. Wang, Z. Li, G. Yuan, Y. Sun, X. Rui, X. Xiang, Tracking the evolution of overlapping communities in dynamic social networks, Knowl.-Based Syst.157 (2018) 81–97.
* Yanni HAN, Deyi LI, Teng WANG. Identifying different community members in complex networks based on topology potential. Front Comput Sci Chin, 2011, 5(1): 87‒99
* Zhixiao Wang, Zhaotong Chen, Ya Zhao, Shaoda Chen, "A Community Detection Algorithm Based on Topology Potential and Spectral Clustering", The Scientific World Journal, vol. 2014, Article ID 329325, 9 pages, 2014.