REPORT

SNA Project : Implementation of *"A cascade information diffusion based label propagation algorithm for community detection in dynamic social networks, 2018"* in **python** to detect communities in an undirected, temporal and unweighted dynamic network based on label propagation approach using cascade information diffusion model.

Team Members : -

Anuj Yadav (2012024) — [Team Leader]
Nrittik Sarmah (2012032)
Chandrakanta Choudhury (2012041)
Shashank Maurya (2012036)
Udit Talukdar (2012043)

23.04.2023 6th Semester, B tech

CONTRIBUTIONS

Anuj Yadav	:	Algorithm Implementation in Python, Report Writing, Improvement in Code Implementation
Nrittik Sarmah	:	Structured, Integrated and Implemented Complete Workflow, Jupyter Notebook work, Metric Calculation
Chandrakanta Choudhury	:	Algorithm Implementation, Quality and Accuracy Metric Measures Calculation, Plotting
Shashank Maurya	:	Dataset Collection and Correction as per required format, Report Preparation
Udit Talukdar	:	Plotting Graph and communities and It’s Editing

INTRODUCTION

Communities are seen as groups, clusters, coherent subgroups, or modules in different fields; community detection in a social network is identifying sets of nodes in such a way that the connections of nodes within a set are more than their connection to other network nodes i.e. number of internal edges within the network is higher than the number of external edges amongst the communities.

In static networks, the connections between the nodes are fixed whereas in dynamic networks the connections can change with time. There are a lot of efficient algorithms for finding communities in static networks but very few algorithms exist for finding communities in dynamic networks.

In our mini-project, we have implemented the CIDLPA algorithm in python and then performed analysis and plotted the communities and evaluated the result using 6 quality and 3 accuracy measures such as Modularity, Conductance, Coverage, Cut Ratio, ARI and NMI to name a few.

Our implementation can be found in the following github repository:

https://github.com/inrittik/CIDLPA

METHODOLOGY

General Overview:

The algorithm is based upon the label propagation approach (LPA) using cascade information diffusion (CID) model. In this method, each node is a community with only one node. Subsequently, in the CID model, each node has two states. The first state is S0 state consisting of uninformed nodes which tend to receive information and the second state is S1 consisting of the nodes which try to cascade information diffusion to S0 state neighbours, based upon this CID model, LPA is applied.

This algorithm has two main stages. The first principal stage consists of applying the CID model, and the second principal stage is the Label Propagation Approach affected by the previous stage.

Algorithm Description:

The proposed method consists of three parts:

First Part -

● This is the initialization part, where each node is given a unique label equal to the node number, and the belonging factor of each label is considered as 1.

Second Part -

● In this part, the **CID** model is used. Each node can belong to S1 (affected) state and S0 (unaffected) state, while how much each node belongs to these states is varied. In this part, value strengths are calculated and then using the value strengths S1 and S0 belonging is calculated.

Third Part -

● In this part, one of the nodes is randomly chosen as the **node tries to update its label set**. Next, the neighbours of the node choose one of the labels whose belonging factor is maximised.

● Then, each neighbour node specifies its vote based on the belonging factor of the selected label and amount of belonging of the node to S1 and S0 States. If neighbour node `j` is giving its vote to label `sl` then the vote is computed by

$$vote_j = S_{0j} * belongingfactor(sl) + S_{1j} * \left(\frac{1 - belongingfactor(sl)}{3}\right) j \in neighbour(i)$$

● Then, the node updates its labels to the label that has the highest vote. After specifying the label that node accepts, it updates the belonging factor of its labels.

● The third part repeats for each node separately. Finally, the node label with belonging factor less than the threshold is removed.

Pseudocode :

Algorithm 1: CIDLPA

```
Input: snapshot G={G_1=<V_1, E_1>,G_2=<V_2, E_2>,…,G_n=<V_n, E_n>}, T
Output: set of communities of G_n
Method:
ΔV={v|v∈G_1}
for ts:=1T do        // ts stands for timestamp
    for v ∈ ΔV
        Node(v).Mem=v;
        Node(v).label.belongingfactor=1
    end for
    ΔV={v | v∈ G_(ts+1) ∩ v ∉ G_ts } ts≠T
end for
ΔV={v|v∈G_1}
for ts:=1T do        // ts stands for timestamp
    for v ∈ ΔV do
        neighb(v)=v. ObtainNbs();
        calculate the strength value of the neighb(v) to node v
        calculate the amount of belonging node v to S_1 state
        calculate the amount of belonging node v to S_0 state
    end for
    ΔV={v | v∈ G_(ts+1) ∩ v ∉ G_ts } ts≠T
end for
ΔE={e|e∈G_1}
for ts:=1T do
    ChangedNodes ={ u,v |(u,v) ∈ΔE}
    V_old={v | v∈ G_ts∩ v ∉ G_(ts+1) } }
    ChangedNodes= ChangedNodes- V_old
    for it = 1 : T do // it means iteration
        ChangedNodes.ShuffleOrder();
            for v ∈ ChangedNodes do
                neighb(v)= v.ObtainNbs();
                candidatelabels=neighb(v).GetLabels();
                ComputeVote(candidatelabels)
                labelset(v).update(candiadatelabels.The maximum vote);
                Nodes(v).Normalize(labelset(v));
    end for
    end for
    remove Nodes(i) labels seen with belonging factor < r
    ΔE={e | e ∈ G_(ts+1) ∩ e ∉ G_ts }∪{e | e ∈ G_ts∩ e ∉ G_(ts+1) } ts≠T
end for
```

Implementation details:

Input Format :
N, TS // N : No of nodes, TS : No of timestamps/snapshots
M1 // No of edges in 1st Snapshot
x1 y1 // edge list's 1st member of the snapshot
.

.

.
xm1, ym1 // edge list's M1th member of the current snapshot

M2 // No of edges in 2nd Snapshot
x1 y1

.

.

.

xm2 ym2

…

Output Format : N (No of Communities) - Each line with the set of nodes in the specific community

[x1,x2,x3…] // Output Community 1

[y1,y2…] // Output Community 2

[z1,z2,z3,z4,z5…] // Output Community 3

…

Steps for running the the algorithm :

1. Go inside the `src` folder in your terminal/Open your code editor inside the `src` folder.

2. To add a new dataset, go into the dataset folder and create a folder by the name of your dataset.

3. Add 2 files inside your dataset, `t1.txt` - for the edgeslist of the network at the 1st timestamp and `input.txt` - for the entire input of nodes, timestamps and the edgeslist of each timestamp.

4. Go to the project notebook, import `coderunner` and call `run_cidlpa` by passing in the name of your dataset.

Code in python :

```
# Header imports

import random

import sys

from collections import defaultdict

from typing import List, Tuple, Set

import time


start = time.time()


# Read input from file and write the output to a file

sys.stdin = open('input.txt', 'r')
```

```
sys.stdout = open('output.txt', 'w')


# input :

# n - No of Nodes

# ts - No of timestamps

n, ts = map(int, input().split())

r = 0.5


# Data Structures Used :


# adj - adjacency list

# adj[t] : adjacency list at timestamp t

adj = [[set() for i in range(n+1)] for j in range(ts + 10)]


# edge - Edge list

# edge[t] : edge list at time t: {(1,2), (2,3)}

edge = [set() for i in range(ts + 10)]


# G[t] - Contains nodes at timestamp t

G = [set() for i in range(ts + 10)]


# Label : Dictionary containing details of labels of community to which a node
belongs for a node x

# Access Method : Label[node] = {labels to which a node belong} with it's
belonging factor
```

```
# "label" notice lowercase l denotes community no (like for community 1 : 0,
community 2 : 1 etc)

# where as Label (Uppercase) denotes data structure as mentioned above


Label = defaultdict(dict)


# b - belonging factor

# Access Method : b[t] : b[t][node][label] = bf

# at timestamp t, denotes the belonging factor of the selected node to labels
belonging (of community)

#  gives the value of belonging factor of node to that label


b = {}


# S[0] and S[1] tells the ability of node not to be affected or to be affected

# by it's neighbour nodes respectively

S = [[0.0 for x in range(n+1)] for i in range(2)]


# Initialization of belonging factor

for i in range(ts + 10):

    b[i] = {}

    for j in range(n+1):

        b[i][j] = {}


# for i in range(ts):
```

```
#     sum += 1


# Helper Functions
# from line 55 to 195

# Changed vertices list
def v_change(t1: int, t2: int) -> Set[int]:
    s = set()
    for x in G[t1]:
        if x not in G[t2]:
            s.add(x)
    return s

# Calculating amount of affectedness and non affectedness for each node
def find_belonging(i: int, strength: List[float]) -> None:
    sum_strength = sum(strength)
    S[1][i] = sum_strength / len(strength)
    S[0][i] = 1.0 - S[1][i]

# Calculate the strength between node i and node j
def find_strength(i: int, j: int, t: int) -> float:
    set_div = 0
    for x in adj[t][j]:
```

```python
        if x != i and x not in adj[t][i]:
            set_div += 1
    val = set_div / len(adj[t][j])
    return val

# Calculate the strength of a node to it's neighbours
def cal_strength(x: int, neighb: List[int], t: int) -> List[float]:
    strength = []
    for i in neighb:
        val = find_strength(i, x, t)
        strength.append(val)
    return strength

# Find the nodes in the given edge list
def find_nodes(e: Set[Tuple[int, int]]) -> Set[int]:
    nodes = set()
    for it in e:
        x, y = it
        nodes.add(x)
        nodes.add(y)
    return nodes

# Get the labels of communities of a list of nodes' neighbours
def get_labels(neighb: List[int]) -> List[int]:
```

```
    labels = []

    for x in neighb:

        mx_bf = 0.0

        mx_label = x

        for l, bf in Label[x].items():

            if bf > mx_bf:

                mx_label = l

                mx_bf = bf

        labels.append(mx_label)

    return labels


# Compute the vote of the candidate labels for a node

def compute_vote(candidateLabels: List[int], neighb: List[int]) -> List[float]:

    vote = []

    for i in range(len(candidateLabels)):

        v = 0.0

        for j in neighb:

            sl = candidateLabels[i]

            if sl in Label[j]:

                v += S[0][j] * Label[j][sl] + S[1][j] * ((1 - Label[j][sl]) /
3.0)

        vote.append(v)

    return vote


# Get the Community label having the maximum vote
```

```python
def get_maximum_vote(vote: List[float], candidateLabels: List[int]) -> int:

    mx_vote = 0

    mx_vote_label = 0

    for j in range(len(vote)):

        if vote[j] > mx_vote:

            mx_vote = vote[j]

            mx_vote_label = candidateLabels[j]

    return mx_vote_label


# Normalize the function to make it dynamic, update the belonging factors of each label for

# each of the node at a given timestamp as needed

def normalize(x: int, t: int) -> None:

    remove = []

    for c, bf in Label[x].items():

        new_bf = 0

        for y in adj[t][x]:

            if c in b[t][y]:

                new_bf += b[t][y][c]

        if len(adj[t][x]) == 0 :

            new_bf = 0

        else:

            new_bf /= len(adj[t][x])

        Label[x][c] = new_bf

        b[t+1][x][c] = new_bf
```

```
        if new_bf == 0.00:
            remove.append((x,c))
    for x,c in remove:
        Label[x].pop(c)
        b[t+1][x].pop(c)
    sum = 0
    for l, bf in b[t+1][x].items():
        sum += bf
    if sum == 0:
        Label[x][x] = 1
        b[t+1][x][x] = 1
    else:
        add_val = (1.00 - sum) / len(b[t+1][x])
        for l, bf in b[t+1][x].items():
            b[t+1][x][l] += add_val
            Label[x][l] = b[t+1][x][l]


# Remove the labels with belonging factor below the threshold value r
def remove_labels(t: int, r: float, set_changedNodes: Set[int]) -> None:
    global adj, edge, G, Label, b, S

    for x in set_changedNodes:
        remove = []
        sum_bf = 0
```

```
mx_label = x
mx_bf = 0

for l, bf in list(Label[x].items()):
    if bf < r:
        remove.append(l)
        del Label[x][l]
        del b[t+1][x][l]
        sum_bf += bf
        if bf > mx_bf:
            mx_bf = bf
            mx_label = l

for l in remove:
    b[t+1][x].pop(l, None)

if not Label[x]:
    Label[x][mx_label] = 1.00
    b[t+1][x][mx_label] = 1.00
else:
    val = (1.00 - sum_bf) / len(Label[x])
    for l in Label[x]:
        Label[x][l] += val
        b[t+1][x][l] += val
```

```
# Main algorithm to detect the communities

# Store the information about the graph for different timestamps



for t in range(ts) :

    m = int(input())

    for i in range(m) :

        x, y = map(int, input().split())

        edge[t].add((x,y))

        adj[t][x].add(y)

        adj[t][y].add(x)

        G[t].add(x)

        G[t].add(y)

        if(t == ts - 1) :

            edge[ts].add((x,y))

            adj[ts][x].add(y)

            adj[ts][y].add(x)

            G[ts].add(x)

            G[ts].add(y)

ts = ts + 1


# Step 1 :

# Initialization
```

```
# Update the label for each of the node and and intialize it's belonging factor

# as needed during the different timestamps

v = set(G[0])

for t in range(ts) :

    # print(v)

    for element in v :

        Label[element][element] = 1

        b[t][element][element] = 1

    if t != ts - 1:

        v = v_change(t + 1, t)


# Step 2 :

# Calculate the amount of connectedness and non connectedness for each of the node

# at different timestamps and also update it's belonging factor accordingly as needed


v = set(G[0])

for t in range(ts):

    for x in v:

        neighb = []

        for i in adj[t][x]:

            neighb.append(i)

        strength = cal_strength(x, neighb, t)

        find_belonging(x, strength)
```

```
    if t != ts-1:
        v = v_change(t+1, t)

# Step 3 :
# Normalize it to make it dynamic
# and update it's belonging factor accordingly and remove the community labels
# below the threshold value


# Get the edge list at initial timestamp
e = edge[0]
# Find set of nodes in the timestamp
set_changedNodes = find_nodes(e)

# normalize and remove labels for each timestamp
# according to the maximum vote of the neighbours of a node

for t in range(ts):
    set_changedNodes = find_nodes(e)
    Vold = set()
    if t != ts-1:
        Vold = v_change(t, t+1)
    for x in Vold:
        set_changedNodes.discard(x)
```

```
    for it in range(ts):

        changedNodes = list(set_changedNodes)

        random.shuffle(changedNodes)


        for x in changedNodes:

            neighb = []

            for i in adj[t][x]:

                neighb.append(i)

            candidateLabels = get_labels(neighb)

            vote = compute_vote(candidateLabels, neighb)

            mx_vote_label = get_maximum_vote(vote, candidateLabels)

            if mx_vote_label not in Label[x]:

                Label[x][mx_vote_label] = 0

            normalize(x, t)


        remove_labels(t, r, set_changedNodes)


# Store the result in a list according to the label of the community

res = [set() for i in range(n+1)]

for i in range(n+1):

    for l, bf in Label[i].items():

        res[l].add(i)


# if for any of label of the community, if empty, leave it or else store it in
the list
```

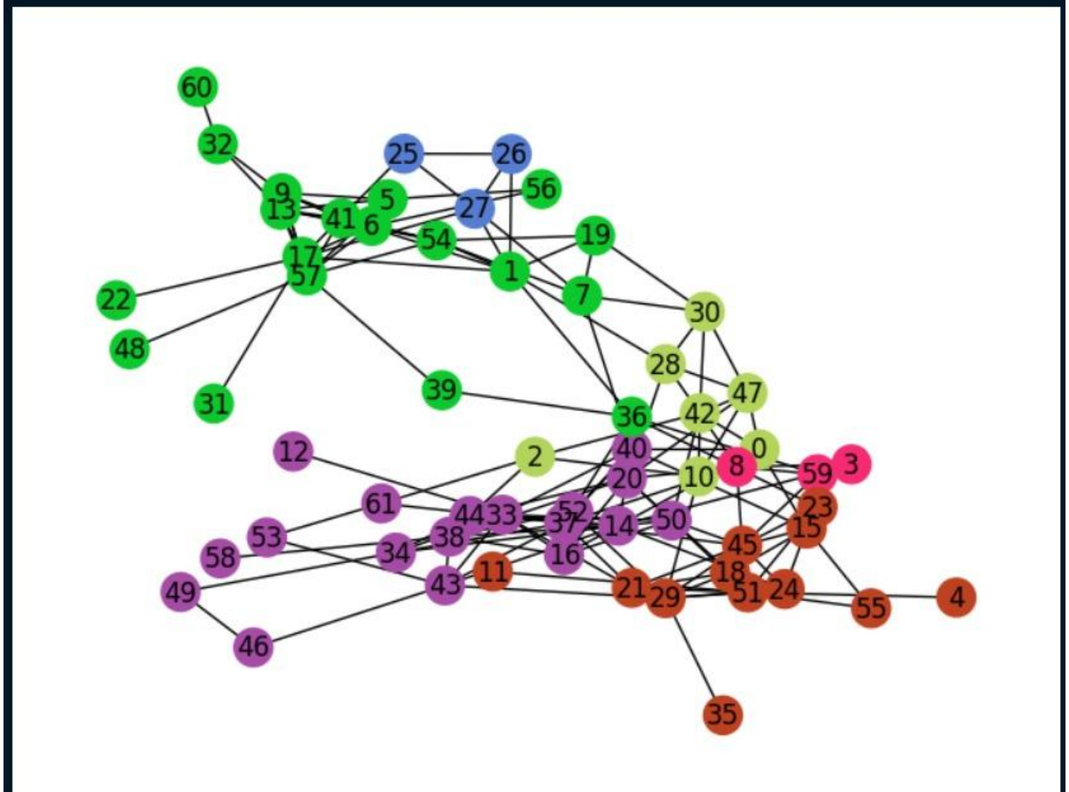
```
comm_set = []
for i in range(n+1):
    if len(res[i]) == 0:
        continue
    s = set(res[i])
    comm_set.append(s)

# Remove the subset communities if present
communities = []
for i in range(len(comm_set)):
    is_subset = False
    for j in range(len(comm_set)):
        if i == j:
            continue
        if comm_set[i].issubset(comm_set[j]):
            is_subset = True
            break
    if not is_subset:
        communities.append(comm_set[i])

#printing the communities
for s in communities:
    print(*s)
```

RESULTS

Small datasets -

Dolphins :

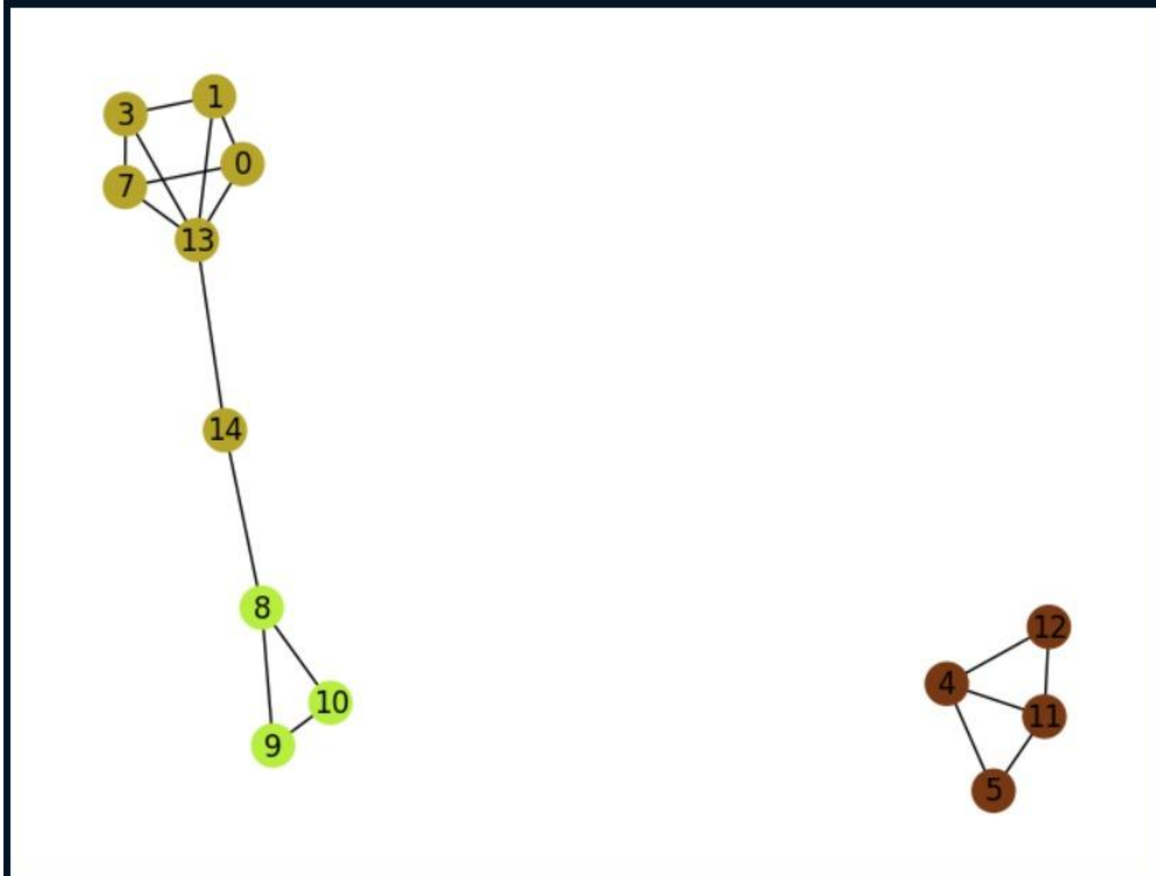
Execution Time: 0.12859082221984863



Modularity: 0.5296665480004737
Conductance: 0.7484276729559749
Covarance: 0.7484276729559748
ARI: 0.5010741880245764
Cut Ratio: 0.25157232704402516
NMI: 0.5052843949796858

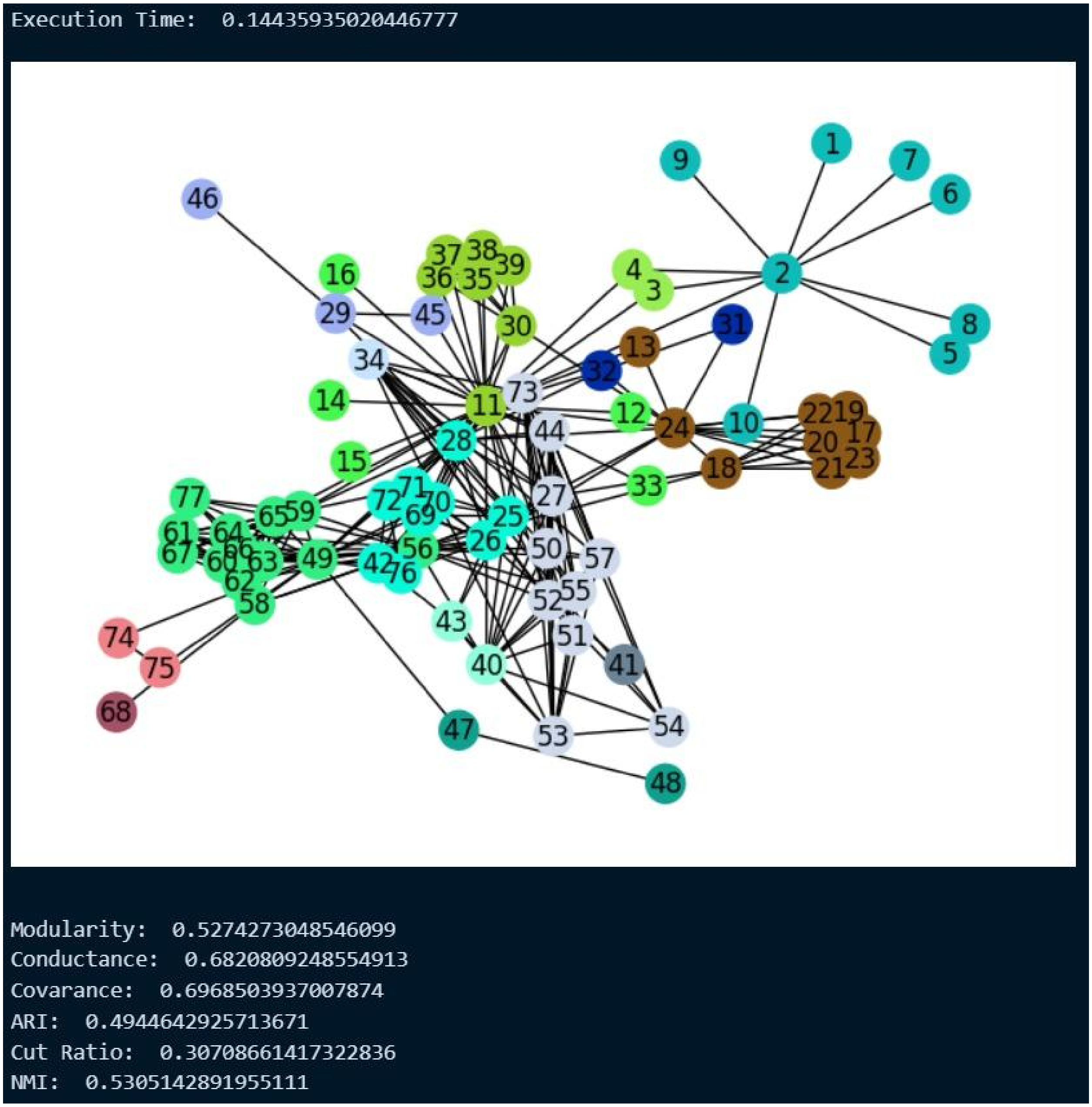
Karate :

```
Execution Time:  0.12749433517456055
```




```
Modularity:  0.6342592592592592
Conductance:  0.9444444444444444
Covarance:  0.9444444444444444
ARI:  0.8299685126875347
NMI:  0.8485515741015118
```

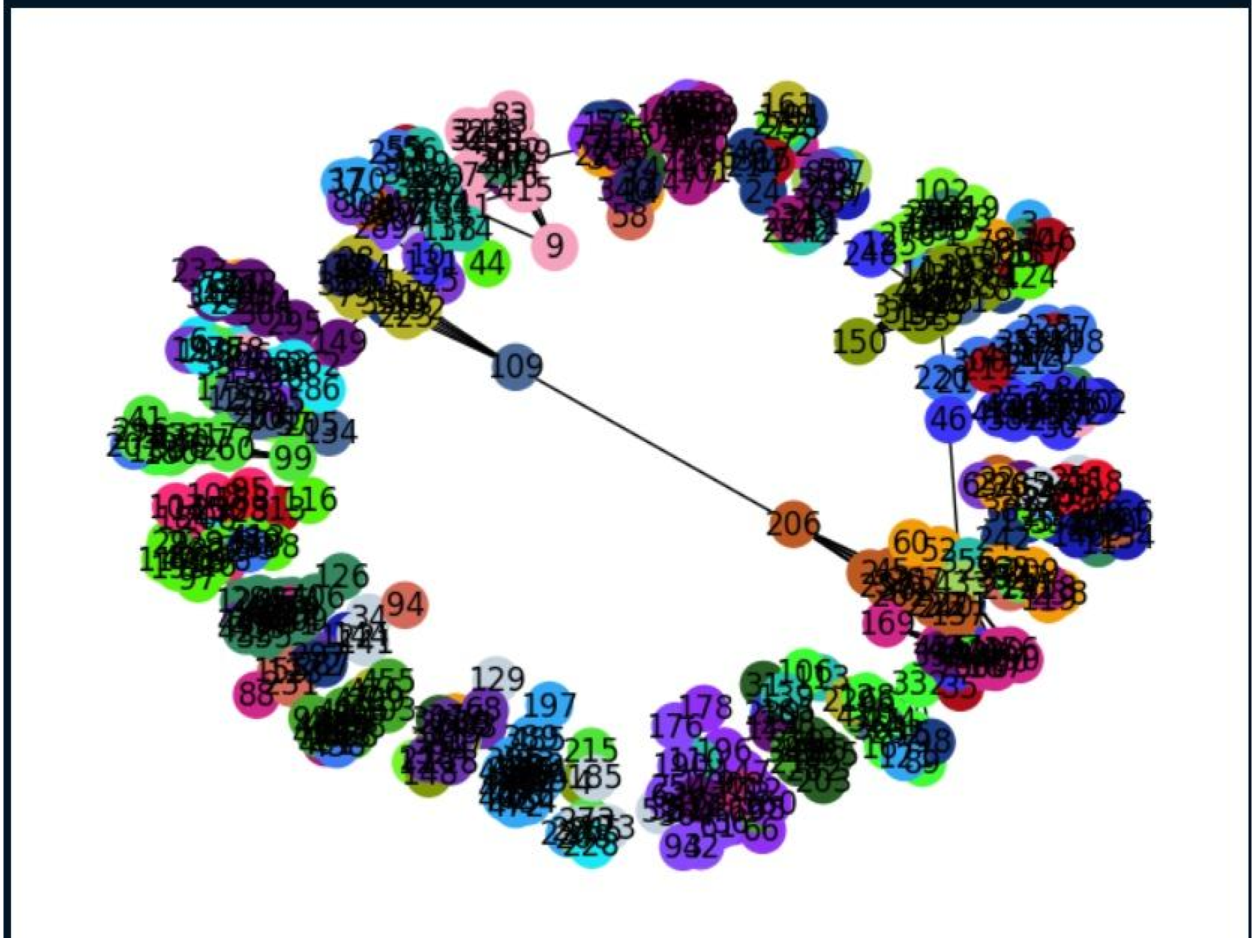
Lesmis :



Large Datasets :

500 Nodes

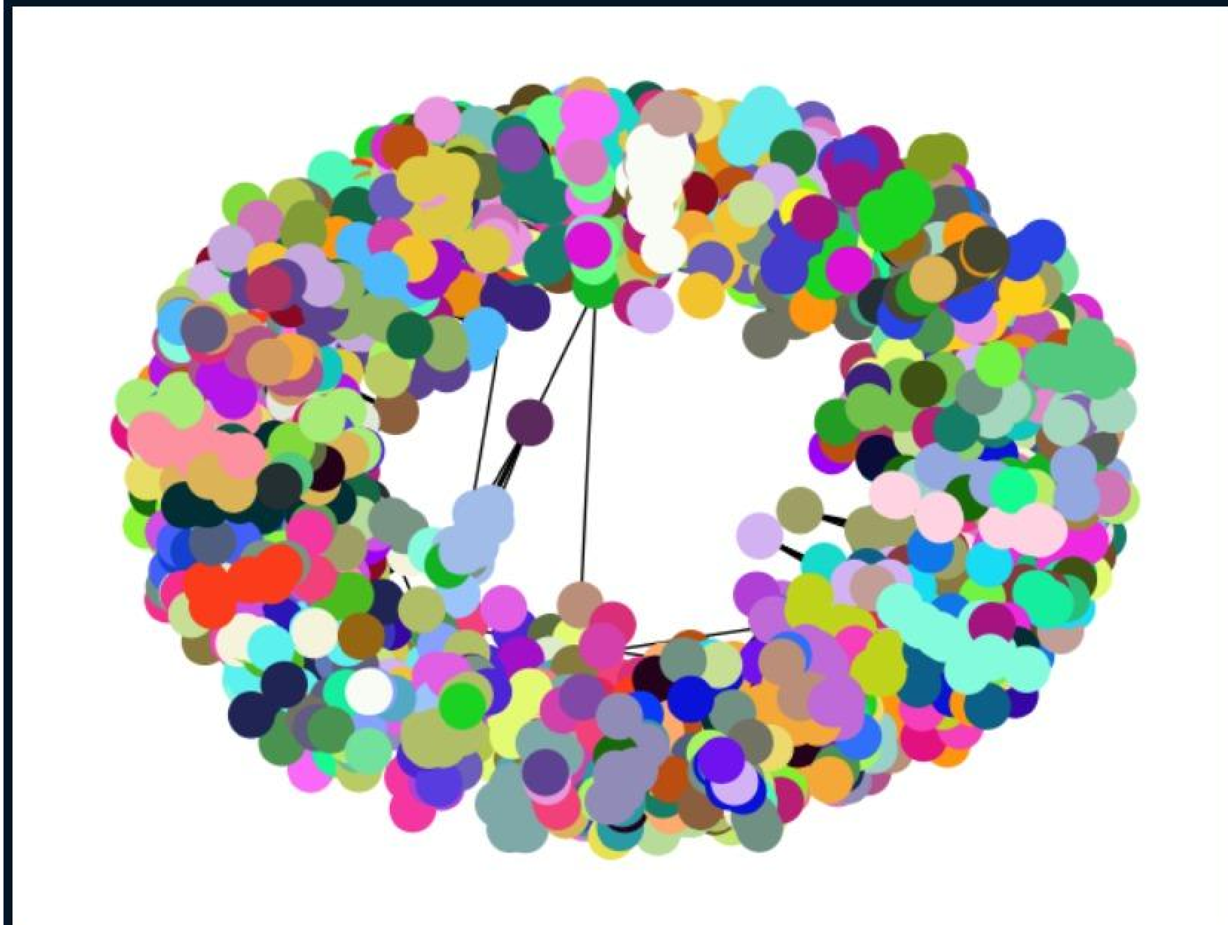
Execution Time: 1.3770039081573486



Modularity: 0.46377590463082785
Conductance: 0.5008006405124099
Covarance: 0.5008006405124099
ARI: 0.38709080665129364
Cut Ratio: 0.4991993594875901
NMI: 0.4755829566058029

2500 Nodes :

```
Execution Time:  7.037355899810791
```



```
Modularity:  0.44192851441757749
Conductance:  0.448427929767252
Covarance:  0.44842792976725193
ARI:  0.2769678890877704
Cut Ratio:  0.551572070232748
NMI:  0.599167295971756
```

CONCLUSION

As indicated in the paper and visible from the Result Section of the report, the algorithm runs at much better speed than most of the already existing algorithms based on the label propagation algorithm. The proposed algorithm is implemented with running time of O(n) and produces better and more accurate results as it uses the cascade information diffusion (CID) model as visible from the different evaluation measures.

SCOPES OF IMPROVEMENT IN CODE & IMPLEMENTATION

Since the implemented code is python a little slow due to its interpretation during the run time rather than being compiled first. In the code implemented, Since the belonging factor, edge list, G, and S can be implemented both in the dictionary and list easily upto 1 million nodes. Replacing the list with the Dictionary gives a better running time as accessing a value in the node in a dictionary is O(1) roughly whereas in list it is O(n).

Although in the highly unlikely and extreme case, during to high collisions and bad hash function, It may shoot upto to O(n) which itself is no less than O(n), thus always performing better than the List on average.

Pseudocode :

S = {}

b = {}

G = {i: set() for i in range(ts + 10)}

edge = {i: set() for i in range(ts + 10)}

The implemented improved algorithm is uploaded in the following github repository -

https://github.com/inrittik/CIDLPA, in the file named as 'improvedAlgo.py'.

REFERENCES

1. https://www.researchgate.net/publication/323205885_A_cascade_information_diffusion_based_label_propagation_algorithm_for_community_detection_in_dynamic_social_networks

2. https://github.com/sunwww168/DCDID/blob/master/data/realworlddata1.rar
3. https://chat.openai.com/
4. https://github.com/
5. https://google.com/