Social Network Analysis

A Mini Project Report on

A Cascade Information Diffusion based Label Propagation Algorithm for community detection in dynamic social networks

Group 6

Rishu Kumar: 1912084 (Team Lead)
Abhishek Pandey: 1912168
Bandaru Sugandhi: 1912166
Nehal Choudhury: 1912141
Aditya Singh: 1912165

Contribution

Rishu Kumar: 1912084

- Group Leader
- Implementation
- Report Review

Abhishek Pandey: 1912168

- Implementation
- Analysis

Bandaru Sugandhi: 1912166

- Implementation
- Plotting

Nehal Choudhury: 1912141

- Report Preparation
- Finding datasets

Aditya Singh: 1912165

- Report Preparation
- Finding datasets

Summary

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. In static networks, the connections between the nodes are fixed wheras 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 that condiers overlapping communities as well.

The paper presents a new method for finding overlapping communities in dynamic social networks. This method is based on Cascade Information Diffusion model (CID) and Label Propagation Approach (LPA) and so it has been termed as CIDLPA.

In our mini-project, we have implemented the CIDLPA algorithm in C++ and used python for plotting and analysis. We have also implemented and used overlapping permanence as our evaluation metric for analysis.

Our implementation can be found on this github repository:
https://github.com/rishu110067/Community-Detection-in-Dynamic-Social-Networks .

Definitions

Community Detection in Dynamic Network:

A community is a set of densely connected nodes that have weaker links with the rest of network. Detection communities can be associated with clustering nodes based on the network topology while no information on the size and structure of communities is available. Detecting communities can be both static and dynamic. In static mode, one snapshot is considered. In dynamic mode, more than one snapshot is considered. In fact, the dynamic mode is matched with more realistic results than static mode.

Cascade Information Diffusion (CID):

The process of effectively spreading information such as new ideas, breaking news, and advertisements, and is called information diffusion. In cascade model, each node can be informed/affected or uninformed/unaffected. In the CID model, each node has two states. The first state is S0-state consisting of uninformed nodes, which tend to receive information. The second state is S1-state consisting of the nodes, which try to cascade information diffusion to S0-state neighbours.

Label Propagation Approach (LPA):

The LPA is a simple and highly efficient approach which discovers communities by exchanging labels. Every node is initialized with a unique label and at every step each node adopts the label that most of its neighbors currently have. So the labels propagate through the network. As the labels propagate, densely connected groups of nodes quickly reach a consensus on a unique label. LPA reaches convergence when each node has the majority label of its neighbours. LPA stops if either convergence or the user-defined maximum number of iterations is achieved. It detects communities with linear time complexity but can only handle disjoint communities.

Belonging Factor:

The belonging factor represents the strength of x's membership to the community. It is computed by the following equation:

$$b_t(c,x) = \frac{\sum_{y \in N(x)} b_{t-1}(c,y)}{|N(x)|}$$

where bt(c,x) of node x to community c in iteration t and N(x) is set of x's neighbours.

Value Strength:

Value strength (j to i) is a measure of the role of j in information diffusion to i. It is given by

$$V_{(i \to j)} = \frac{|C_j \backslash C_i|}{|C_j|}$$

$$V_{(j \to i)} = \frac{|C_i \backslash C_j|}{|C_i|}$$

where Ci and Cj represents the set of i and j neighbours respectively and Ci \ Cj represents the difference set of Ci against Cj.

S1-Belonging and S0-Belonging:

S1-belonging and S0-belonging tells us the amount of belonging of a node to S1-state (affected state) and S0-state (unaffected state) respectively in the CID model. For each node i, they are calculated as

$$S_{1i} = \frac{\sum_{j \in neighbours(i)} V_{(j \to i)}}{n}$$

$$S_{0i} = 1 - S_{1i}$$

where V(j --> i) is value strength from j to i and n is the number of neighbours of node i.

Methodology

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 maximized.

- 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

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

  where belonging factor(sl) represents the amount of belonging of the selected label to node j. Symbols S_{1j} and S_{0j} represents the amount of belonging of the node j to S1 and S0 states.

- 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 that belonging factor is below 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∈G1}
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 ∉ Gts } ts≠T
end for
ΔV={v|v∈G1}
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 S1 state
          calculate the amount of belonging node v to S0 state
    end for
   ΔV={v | v∈ G(ts+1) ∩ v ∉ Gts } ts≠T
end for
ΔE={e|e∈G1}
for ts:=1T do
     ChangedNodes ={ u,v |(u,v) ∈ΔE}
     Vold={v | v∈ Gts∩ v ∉ G(ts+1) } }
     ChangedNodes= ChangedNodes- Vold
     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 ∉ Gts }∪{e|e ∈ Gts∩ e ∉ G(ts+1) } ts≠T
end for
```

Implementation Details

We have implemented the algorithm in C++ and used Python for visualization of results.

Input Format:

The first line of the input should consist of two inetegers N - number of nodes in the network and T - number of timestamps. For each timestamp t, the first line should consist of an integer Mt - number of edges at timestamp t, followed by Mt pairs of integers x and y which represents an edge from node x to y.

N T
M1
x1 - y1
x2 - y2
x3 - y3
…..
M2
x1 - y1
x2 - y2
x3 - y3
…..

Output Format:

If the algorithm finds K communities then the output will consist of K lines where i-th line consists of nodes in the i-th community.

C++ code:

First we are setting some global variables

```
const int N = 10000;
const int T = 100;          // setting maximum no. of nodes 10000 and max no. of timestamps 100
set<int> adj[T][N];         // adj[T] is adjacency list representation of graph at timestamp t
set<pair<int,int>> edge[T]; // edge[t] stores the edge list for timestamp t
set<int> G[T];              // G[t] is vertex set of nodes of Graph at timestamp t
map<int,double> Label[N];   // Label[x] is labelset of node x
                            // label[x][l] tells belonging factor of node x in community l
map<int,double> b[T][N];    // b[t][x][l] tells belonging factor of node x in community l at timestamp t
double S[2][N];             // S[0][x] and S[1][x] tells S0 and S1 belonging of node x
```

Part 1:

```
// Part 1: Initialization
set<int> v = G[0];
for(int t = 0; t < ts; t++)
{
    for(auto x: v)
    {
        Label[x][x] = 1;
        b[t][x][x] = 1;
    }
    if(t != ts-1) v = v_change(t+1, t);
}
```

v_change:

```
// finds change in vertex set from timestamp t1 to t2
set<int> v_change(int t1, int t2)
{
    set<int> s;
    for(auto x: G[t1])
    {
        if(G[t2].find(x) == G[t2].end())
            s.insert(x);
    }
    return s;
}
```

Part 2:

```
// Part 2: CID
v = G[0];
for(int t = 0; t < ts; t++)
{
    for(auto x: v)
    {
        vector<int> neighb;
        for(auto i: adj[t][x]) neighb.push_back(i);
        vector<double> strength = cal_strength(x, neighb, t);
        find_belonging(x, strength);
    }
    if(t != ts-1) v = v_change(t+1, t);
}
```

Value Strength and S1 & S0 belonging

```
// finds value strength (i -> j)
double find_strength(int i, int j, int t)
{
    int set_div = 0;
    for(auto x: adj[t][j])
    {
        if(x!=i && adj[t][i].find(x) == adj[t][i].end())
            set_div++;
    }
    double val = (double)set_div/adj[t][j].size();
    return val;
}

// finds value strength from x to all neighbours of x
vector<double> cal_strength(int x, vector<int> neighb, int t)
{
    vector<double> strength;
    for(auto i: neighb)
    {
        double val = find_strength(i, x, t);
        strength.push_back(val);
    }
    return strength;
}

// finds s1 and s0 belonging based on the value strengths
void find_belonging(int i, vector<double> &strength)
{
    double sum_strength = 0;
    for(auto it:strength) sum_strength+=it;
    S[1][i] = sum_strength/strength.size();
    S[0][i] = 1.00 - S[1][i];
}
```

Part 3

```
// Part 3: Label Propagation
set<pair<int,int>> e = edge[0];
set<int> set_changedNodes = find_nodes(e);
for(int t = 0; t < ts; t++)
{
    set<int> set_changedNodes = find_nodes(e);
    set<int> Vold;
    if(t!=ts-1) Vold = v_change(t, t+1);
    for(auto x: Vold) set_changedNodes.erase(x);

    for(int it = 0; it < ts; it++)
    {
        vector<int> changedNodes;
        for(auto x: set_changedNodes) changedNodes.push_back(x);
        shuffle(changedNodes.begin(), changedNodes.end(), default_random_engine(0));

        for(auto x: changedNodes)
        {
            vector<int> neighb;
            for(auto i: adj[t][x]) neighb.push_back(i);
            vector<int> candidateLabels = get_labels(neighb);
            vector<double> vote = compute_vote(candidateLabels, neighb);
            int mx_vote_label = get_maximum_vote(vote, candidateLabels);
            if(Label[x].find(mx_vote_label) == Label[x].end())
                Label[x][mx_vote_label] = 0;
            normalize(x,t);
        }
    }
    remove_labels(t, r, set_changedNodes);
}
```

Candidate Label Set and Voting

```
// finds vote for each node in candidate label set
vector<double> compute_vote(vector<int> &candidateLabels, vector<int> &neighb)
{
    // each neighbour chooses a label and candidateLabels set and votes it

    vector<double> vote;
    for(int i = 0; i < candidateLabels.size(); i++)
    {
        double v = 0.00;
        for(int k = 0; k < neighb.size(); k++) {
            int j = neighb[k];
            int sl = candidateLabels[i];
            if(Label[j].find(sl) != Label[j].end()) {
                v += (double)S[0][j] * Label[j][sl] + (double)S[1][j] * ((1 - Label[j][sl]) / 3.0);
            }
        }
        vote.push_back(v);
    }
    return vote;
}

// returns the label in candidate label set with the maximum vote
int get_maximum_vote(vector<double> &vote, vector<int> &candidateLabels)
{
    double mx_vote = 0;
    int mx_vote_label;
    for(int j = 0; j < vote.size(); j++)
    {
        if(vote[j] > mx_vote)
        {
            mx_vote = vote[j];
            mx_vote_label = candidateLabels[j];
        }
    }
    return mx_vote_label;
}


// finds candidate label set for each neighbour node
vector<int> get_labels(vector<int> &neighb)
{
    vector<int> labels;
    for(auto x: neighb) {
        double mx_bf = 0;
        int mx_label = x;
        for(auto it: Label[x]) {
            auto l = it.first;
            auto bf = it.second;
            if(bf > mx_bf) {
                mx_label = l;
                mx_bf = bf;
            }
        }
        labels.push_back(mx_label);
    }
    return labels;
}
```

Normalization

```
// normalizes and updates the belonging factor of label set of node x
void normalize(int x, int t)
{
    vector<pair<int,int>> remove;
    for(auto it: Label[x]) {
        auto c = it.first;
        auto bf = it.second;

        // Finding new belonging factor for the next timestamp
        double new_bf = 0;
        for(auto y: adj[t][x]) {
            // If c community is present in the label set of y and has non zero belongiong factor
            if(b[t][y].find(c) != b[t][y].end())
                new_bf += b[t][y][c];
        }

        new_bf /= adj[t][x].size();
        Label[x][c] = new_bf;
        b[t+1][x][c] = new_bf;
        if(new_bf == 0.00) {
            // If new belonging factor is 0 then we will remove c from label set of x
            remove.push_back({x,c});
        }
    }

    // Removing all the communities with new belonging factor 0 from their corresponding node's label set
    for(auto it: remove) {
        auto x = it.first;
        auto c = it.second;
        Label[x].erase(c);
        b[t+1][x].erase(c);
    }

     // Normalize sum of bf to 1
     double sum = 0;
     for(auto it: b[t+1][x]) {
         auto l = it.first;
         auto bf = it.second;
         sum += bf;
     }
     if(sum == 0){ Label[x][x] = 1; b[t+1][x][x] = 1; }
     else {
         double add_val = (1.00 - sum)/(b[t+1][x].size());
         for(auto it: b[t+1][x]) {
             auto l = it.first;
             auto bf = it.second;
             b[t+1][x][l] += add_val;
             Label[x][l] = b[t+1][x][l];
         }
     }
 }
```

Removing Labels

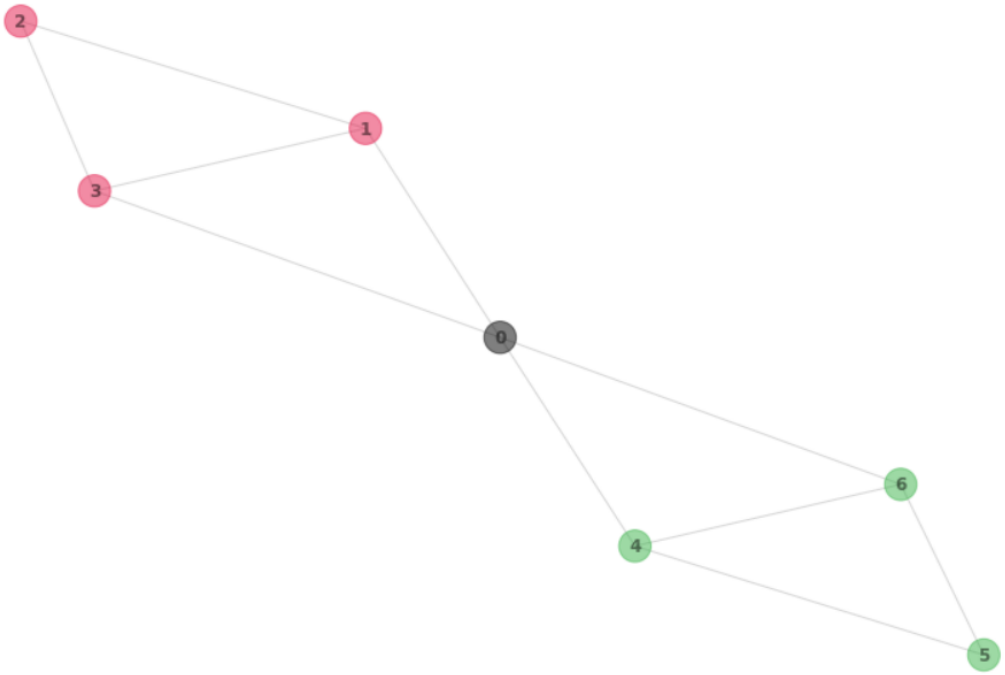
```
// remove labels with belonging factor less than threshold r in the label set of each node
void remove_labels(int t, int r, set<int> &set_changedNodes)
{
    for(auto x: set_changedNodes) {
        // Remove labels with belonging factor less than r
        vector<int> remove;
        double sum = 0;
        int mx_label = x;
        double mx_bf = 0;
        for(auto it: Label[x]) {
            auto l = it.first;
            auto bf = it.second;
            if(bf < r) {
                remove.push_back(l);
                Label[x].erase(l);
                b[t+1][x].erase(l);
                sum += bf;
                if(bf > mx_bf) {
                    mx_bf = bf;
                    mx_label =l;
                }
            }
        }

        // If label set of node x becomes empty
        // then pick the label with max belonging factor removed, and set it bf = 1
        if(Label[x].empty()) {
            Label[x][mx_label] = 1.00;
            b[t+1][x][mx_label] = 1.00;
        }
        else {
            // Add the val to belonging factor of all the labels of node x remaining
            // so that sum of bf of labels remain 1
            double val = (1.00-sum) / Label[x].size();
            for(auto it: Label[x]) {
                auto l = it.first;
                auto bf = it.second;
                Label[x][l] += val;
                b[t+1][x][l] += val;
            }
        }
    }
}
```

Experiments and Result Analysis

Results

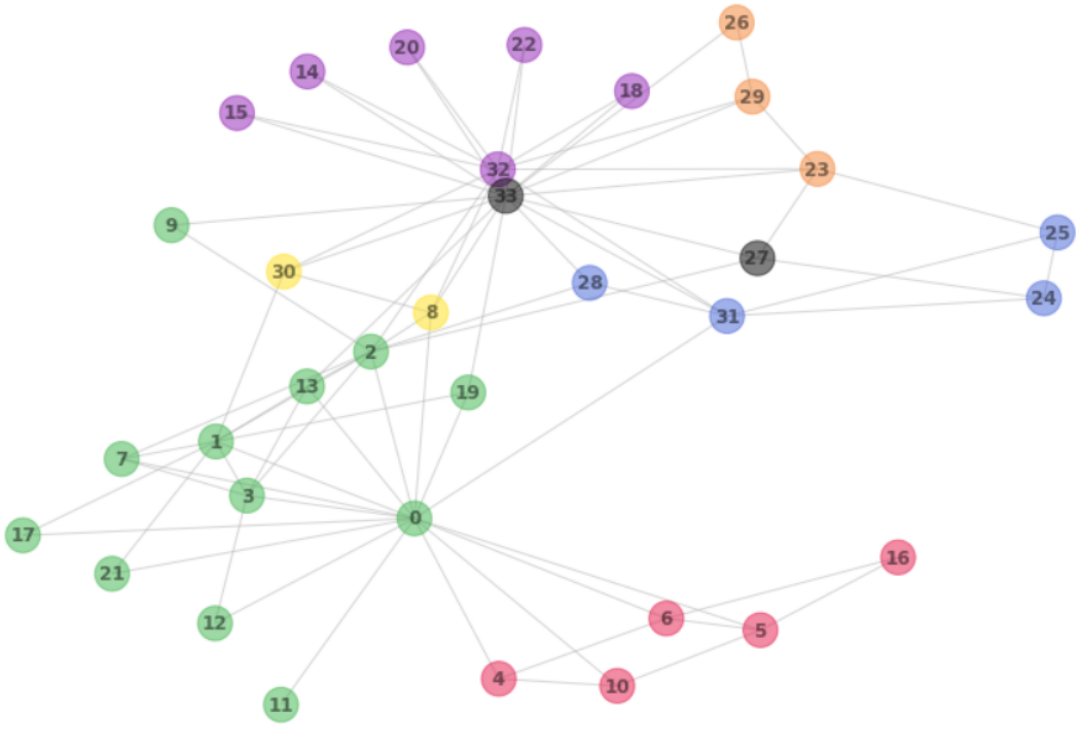
- **Sample graph, no. of nodes = 7, static**

```
Communities:
[0, 1, 2, 3]
[0, 4, 5, 6]
```



- **Karate Club graph, no. of nodes = 34, static**

```
Communities:
[4, 5, 6, 10, 16]
[0, 1, 2, 3, 7, 9, 11, 12, 13, 17, 19, 21]
[8, 30]
[24, 25, 27, 28, 31]
[23, 26, 27, 29, 33]
[14, 15, 18, 20, 22, 32, 33]
```

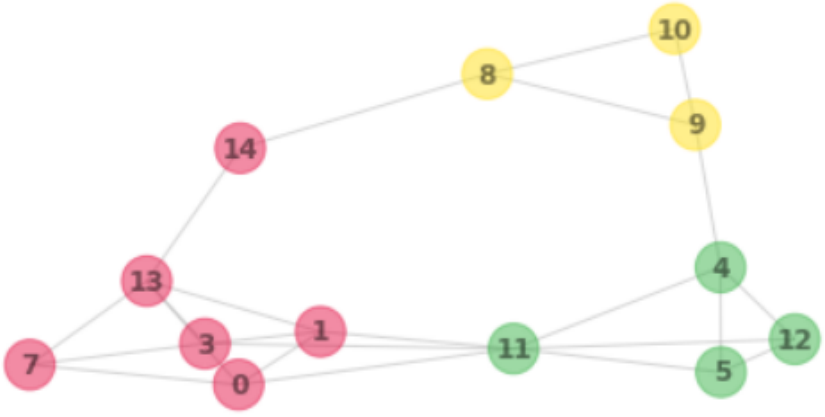


- **dolphin network, no. of nodes = 62, static**

```
Communities:
[1, 5, 6, 9, 13, 17, 22, 31, 32, 39, 41, 48, 54, 56, 57, 60]
[12, 14, 16, 20, 33, 34, 36, 37, 38, 40, 43, 44, 50, 52]
[4, 11, 15, 18, 21, 23, 24, 29, 35, 45, 51, 55]
[23, 36]
[1, 25, 26, 27]
[43, 53, 61]
[0, 2, 10, 20, 28, 30, 42, 47]
[34, 46, 49]
[1, 7, 19, 54]
[38, 58]
[3, 8, 59]
```

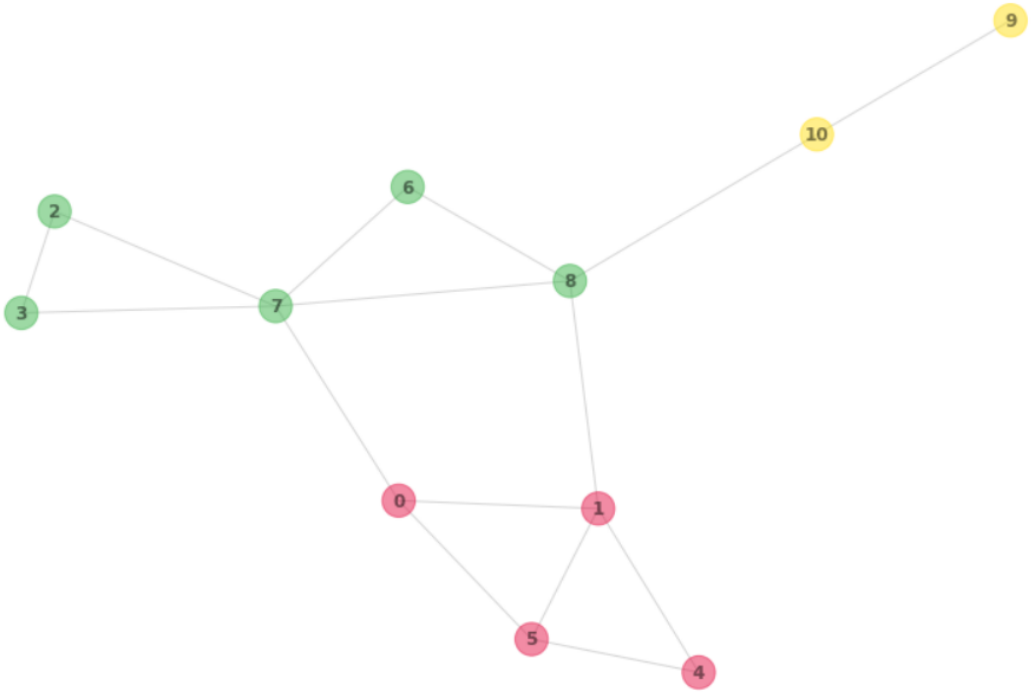
- **random graph, no. of nodes = 15, dynamic**

```
Communities:
[0, 1, 3, 7, 13, 14]
[4, 5, 11, 12]
[8, 9, 10]
```



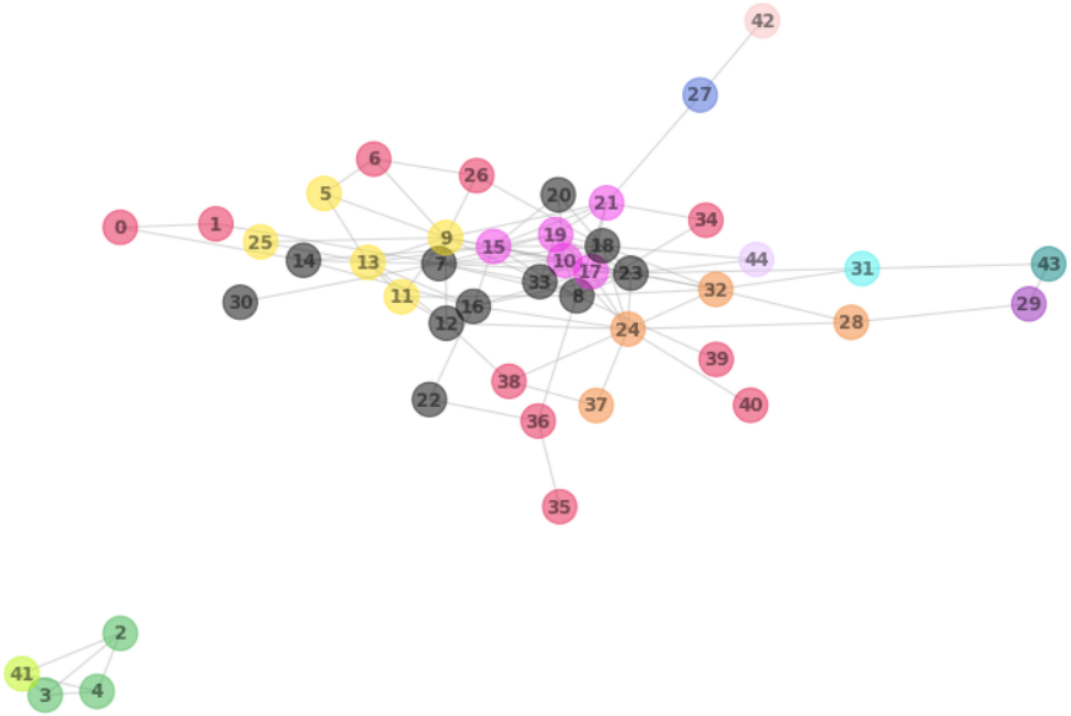
- **reptilia-tortoise graph_1, no. of nodes = 7, dynamic**

```
Communities:
[0, 1, 4, 5]
[2, 3, 6, 7, 8]
[9, 10]
```



- **reptilia-tortoise graph_2, no. of nodes = 45, dynamic**

```
Communities:
[0, 1, 6, 7, 12, 14, 16, 20, 22, 26, 30, 33, 34, 35, 36, 38, 39, 40]
[2, 3, 4]
[5, 8, 9, 11, 12, 13, 25]
[27]
[24, 28, 32, 37]
[29]
[31]
[7, 8, 10, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 30, 33]
[41]
[42]
[43]
[18, 23, 44]
```



Analysis

Permanence Metric:
The central idea of permanence is based on the observation that the strength of membership of a vertex to a community depends upon the following two factors:
(i) the distribution of External connectivity of the vertex to individual communities and **not** the total external connectivity, and
(ii) the strength of its internal connectivity and **not** just the total internal edges

Permanence provides
(i) a more accurate estimate of a derived community structure to the ground-truth community and
(ii) is more sensitive to perturbations in the network

Overlapping Permanence:
Permanence calculation for overlapping communities

$$P_{ov}^{c}(v) = \frac{I^{c}(v)}{E_{max}(v)} \times \frac{1}{D(v)} - (1 - c_{in}^{c}(v)) \cdot \frac{I^{c}(v)}{I(v)}$$

$D(v)$=degree of v
$E_{max}(v)$=Max connection to an external community
$C^{c}in(v)$=clustering coeff. of internal neighbors of v in c
$I(v)$ = # of internal neighbors of v

$$I^{c}(v) = \sum_{e \in \Gamma_{v}^{e}} 1/x_{e}$$

Γ_{v}^{e} = internal edges of v in community
χ_{e} = # of communities edge e shares

Range of Permanence Metric:
It ranges from -1 to 1.
Permanence value will be closer to 1 when the network has **strong community structure**. The maximum value is obtained when the graph consists of series of disconnected cliques.
For a grid, the best value of permanence will be zero.
It's value will be closer to -1 when the communities are weakly detected or wrongly assigned.

Network	Type	Number of nodes	Overlapping Permanence value
Sample Graph	Static	7	0.89
Karate Club	Static	34	0.23
Dolphins	Static	62	0.17
Random graph	Dynamic	11	0.47
Reptilia-tortoise graph_1	Dynamic	15	-0.32
Reptilia-tortoise graph_2	Dynamic	45	0.06

Conclusions

We implemented the proposed CIDLPA algorithm and analysed it for some static and dynamic networks based on the overlapping permanance value. On increasing the number of iterations for the same timestamp, many time we got somewhat better result. Also, we needed to unify few communities which were complete subset of some other large community.

As you can observe from the obtained value of the overlapping permanance metric, we can say that the aforementioned algorithm is quite average to detect overlapping community static/dynamic but it has some advantages.

The advantages of this method are:

- It adopts the CID model on the label propagation algorithm in order to improve the accuracy of community detection in dynamic social networks.
- It can detect overlapping communities in a dynamic social network
- It improves label propagation approach and accuracy

Detecting communities in dynamic networks is very challenging and discovering dynamic communities is still in its infancy and more research is required in this field..

References

- Original Research Paper: https://www.sciencedirect.com/science/article/abs/pii/S1877750317303009v
- Label Propagation Algorithm (LPA): https://journals.aps.org/pre/abstract/10.1103/PhysRevE.76.036106
- Value Strength: https://ieeexplore.ieee.org/document/7552614?reload=true
- Dynamic Networks Repository: https://networkrepository.com/dynamic.php
- Permanence: https://www.researchgate.net/publication/263012336_On_the_Permanence_of_Vertices_in_Network_Communities
- Belonging Factor: https://iopscience.iop.org/article/10.1088/1367-2630/12/10/103018
- Github: https://github.com/rishu110067/Community-Detection-in-Dynamic-Social-Networks