2018 IEEE 34th International Conference on Data Engineering

On Efficiently Detecting Overlapping Communities over Distributed Dynamic Graphs

Xun Jian [#], Xiang Lian [*], Lei Chen [#]

[#] *Hong Kong University of Science and Technology, Hong Kong, China*
{xjian, leichen}@cse.ust.hk

[*] *Kent State University, Ohio, USA*
xlian@kent.edu

***Abstract*—Modern networks are of huge sizes as well as high dynamics, which challenges the efficiency of community detection algorithms. In this paper, we study the problem of overlapping community detection on distributed and dynamic graphs. Given a distributed, undirected and unweighted graph, the goal is to detect overlapping communities incrementally as the graph is dynamically changing. We propose an efficient algorithm, called *randomized Speaker-Listener Label Propagation Algorithm* (rSLPA), based on the label propagation approach. Besides detecting high-quality communities, rSLPA can incrementally update the detected communities after a batch of edge insertion and deletion operations. To the best of our knowledge, rSLPA is the first algorithm that can incrementally capture the same communities as those obtained by applying the detection algorithm from the scratch on the updated graph. Extensive experiments are conducted on both synthetic and real-world datasets, and the results show that our algorithm can achieve high accuracy and efficiency at the same time.**

I. Introduction

In many real-world applications, we often model the underlying data as graphs. For example, we can consider the World Wide Web as a graph by treating each single page as a vertex and hyper-links between pages as edges. Similarly, a social network can also be represented by a graph where users are vertices and friend relationships between users are edges. It has been shown that many real-world networks have a significant property of community structure, which means that the vertices in the network can be partitioned into communities, such that vertices within a community are densely connected and vertices from different communities are sparsely connected.

Various works like [1], [2], [3], [4], [5], [6] have been proposed in recent years, trying to detect communities from different perspectives. Most of those works share a common property that is they all aim at detecting communities on a static graph or on a single machine. However, many real-life networks are both of large scale and dynamic. Identifying communities in such big and rapidly changing networks is challenging.

In this paper, we focus on incremental community detection over distributed and dynamic binary (unweighted and undirected) graphs. Similar to the *Speaker-Listener Label Propagation Algorithm* SLPA [5], we propose the *randomized Speaker-Listener Algorithm* (rSLPA) based on label propagation approach, which is able to incrementally detect communities with high efficiency. To the best of our knowledge, our work is the first one that can detect overlapping communities over distributed and dynamic graphs accurately. Then we discuss the time complexity of the proposed incremental algorithm. Experimental results show that rSLPA can handle dynamic graphs efficiently and effectively.

Specifically, we make the following contributions.

- We design an efficient algorithm rSLPA that can incrementally detect communities over distributed and dynamic graphs.
- We derive the expected complexity of incremental updating of rSLPA.
- We verify the effectiveness and efficiency of rSLPA by conducting extensive algorithms over both synthetic and real-world datasets.

II. The rSLPA Algorithm

The rSLPA algorithm can be divided into four main stages as follows:

1) Initialization;
2) Label Propagation;
3) Incremental Updating; and
4) Post-processing.

When a new graph $G(V, E)$ comes, it will first pass the *Initialization* stage and *Label Propagation* stage, and results in an intermediat graph in which vertices hold labels and other indices. On this intermediat graph, two further operations can be applied. If the user wants to output the communities, the intermediat graph is sent to the *Post-processing* stage, and the result is then extracted. If there are changes (edge insertions/deletiongs) on the original graph, the intermediat graph and the changed edges are sent to the *Incremental Updating* stage, and a new intermediat graph will be calculated, which is ready for the same further operations. We will describe the four stages later.

Two thing should be noted here. First, we only consider edge insertion/deletion as possible graph changes. The reason is that isolated vertices are trivial cases in community detection and can be removed, and therefore we can infer vertex insertion/deletion from the insertion/deletion of its adjacent edges. Second, all the algorithms discussed later consider each single vertex as one independent computation unit, so the whole method can be easily implemented on MapReduce-like systems. We will ignore the details for simlicity.

2375-026X/18/$31.00 ©2018 IEEE
DOI 10.1109/ICDE.2018.00142

IEEE computer society

For the detailed analysis and proofs, please refer to our technical report [7].

A. Initialization

In the *Initialization* stage, each vertex is assigned with a label sequence, which stores a unique label (which is usually the ID of this vertex). Later, the label sequence of each vertex will be enlarged in the label propagation stage.

B. Label Propagation

In the *Label Propagation* stage, there are two super-steps *label sending* and *label selection*. In the *label sending* step, every vertex sends one label to each of its neighbors. Those labels are randomly picked from its label sequence. After this step, every vertex will also receive one label from each of its neighbors. In the *label selection* step, every vertex randomly select one received label and add it as the last one in its label sequence. If the selected label is sent by vertex v_j, we also say this label "is picked from" v_j.

An optimized way of sending and selecting labels would be first decide one neighbor whose label would be selected, and then only this neighbor needs to send the label. This would significantly reduce the communication cost from $O(|E|)$ to $O(|V|)$.

Those two super-steps run iteratively, until the number of iterations is equal to a user-specified parameter T. After T times of label propagation, there are $T+1$ labels in the label sequence of each vertex.

C. Incremental Updating

If we look at each single label received by vertex v_i in the *Label Propagation* stage, every neighbor of v_i has equal probability to be the one that sent this label. However, when the graph changes, this property may not be satisfied by all those labels. In this stage, according to the graph changes, we detect and correct labels that do not satisfy this property.

For a vertex v_i, the correction of a label is done through simulating the *Label Propagation* once within a subset of neighbors of v_i, and replace the old label with the new one. For simplicity, we use *pick* to refer this correction process.

According to how the neighbor set changes, we classify the vertices into 3 categories.

Category 1: Vertices with no neighbor changed. For such a vertex, we can keep all its received labels unchanged.

Category 2: Vertices that only lose neighbor(s). For such a vertex, we need to check if any of the labels is picked from the removed neighbor(s). If so, this label is wrong and we need to randomly *pick* a new label from current neighbors to replace the old one. If not, we can keep the label unchanged.

Category 3: Vertices with new neighbor(s). Suppose the number of unchanged neighbors is n_u and the number of new neighbors is n_α. For such a vertex v_i, if a label is picked from one of the remaining neighbors, with probability $\frac{n_u}{n_u+n_\alpha}$, we keep the label unchanged, otherwise we *pick* a label from the new neighbors. If a label is picked from a deleted neighbor, we have to *pick* a label from all current neighbors.

Besides the label updates that are caused by adjacent edge changes, the label correction can cause updates of other labels. For example, v_i sends label l to v_j, and l is selected by v_j, then after graph changes, v_i updates l to be l'. In this situation, v_j should also update l to be l'.

D. Post-processing

In this stage, we take an intermediate graph as input, and output the extracted communities.

First, for each edge e_{ij} in the graph, we assign a weight w_{ij} to be the fraction of common labels between v_i and v_j's sequences.

After this, we use two thresholds to extract communities. We use the first threshold τ_1, to filter out edges of low weight, then each connected component (with at least two vertices) on the filtered graph is considered to be a community. We apply the second smaller threshold τ_2 between the isolated vertices and their neighbors. If an isolated vertex v_i is connected to a non-isolated one v_j, and the edge weight $w_{ij} \geq \tau_2$, and then v_i is considered to weakly belong to the community that contains v_j. Two communities will overlap when some vertices belong to both of them weakly.

We pick the τ_1 that maximizes the information entropy w.r.t. the relative size of extracted communities, which is

$$\tau_1 = \arg\max_{\tau_1} - \sum_i \frac{|C_i|}{|V|} \cdot \log \frac{|C_i|}{|V|}. \tag{1}$$

τ_2 is picked so that every vertex belongs to at least one community, which means$\tau_2 \leq \min_i \max_j w_{ij}$. To prevent assigning two vertices with very low similarity to the same community, we set $\tau_2 = \min_i \max_j w_{ij}$.

E. Complexity of rSLPA

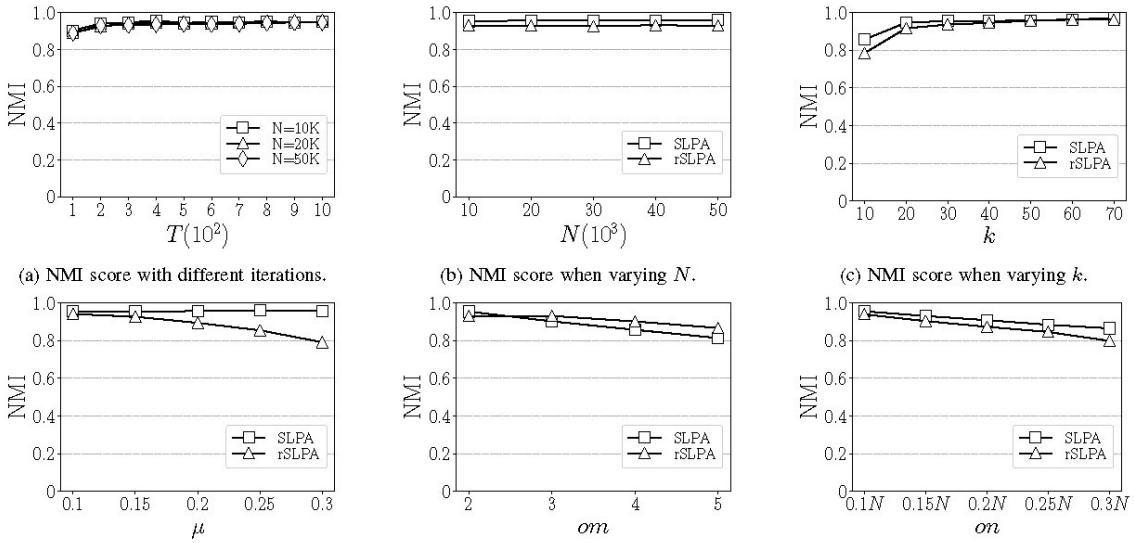
The *Initialization* stage and the *Label Propagation* stage takes $O(T)$ MapReduce rounds in total, and $O(T|V|)$ communication cost.

In the *Incremental Updating* stage, it takes $O(T)$ MapReduce rounds and in average $O(T \cdot |V| - |V| \cdot \sum_{t=1}^{T} \prod_{k=1}^{t}(1 - \frac{p_c}{k}))$ communication cost, where $p_c = \frac{m_d}{|E|} + (1 - \frac{m_d}{|E|})(\frac{|E|-m_d}{|E|-m_d+m_\alpha})$, m_d is the number of deleted edges, and m_α is the number of added edges.

During the *Post-processing* stage, the cost comes from calculating connected components. Using the algorithm proposed in [8], it takes $O(\log d)$ rounds, each with $O(|E| + |V|)$ communication cost, where d is the diameter of the graph.

III. Experiments

To evaluate the effectiveness and efficiency of rSLPA, we conduct extensive experiments on both synthetic and real-world graphs. We use generated networks that contain known communities to test the quality of communities given by rSLPA. Then we use large-scale real-world networks to test its efficiency in both static and dynamic scenarios. We compare our rSLPA with SLPA on both effectiveness and efficiency.



(a) NMI score with different iterations.

(b) NMI score when varying N.

(c) NMI score when varying k.

(d) NMI score when varying μ.

(e) NMI score when varying om.

(f) NMI score when varying on.

Fig. 1. Experiment results on synthetic dataset.

TABLE I
PARAMETERS OF THE LFR BENCHMARK (SELECTED)

Parameter	Description
N	the number of vertices
$maxk$	the max degree
k	the average degree
μ	the mixing parameter
on	the number of overlapping vertices
om	the number of memberships of overlapping vertices

A. Synthetic Dataset

1) Data Generation: We use the LFR benchmark [9] to generate graphs with known communities. The most important parameters to generate a graph are listed in Table I. We choose $N = 10,000$, $k = 30$, $maxk = 100$, $om = 2$, $on = 0.1N$ and $\mu = 0.1$ as the default setting and keep the value of each parameter unchanged unless specified.

2) Evaluation Metrics: In community detection, with known communities as the ground truth, the *Normalized Mutual Information (NMI)* is one of the most widely used measures to evaluate the quality of detected communities. Basically it reveals the similarity between two membership assignments based on the information theory. The score is in range $[0, 1]$, and higher value indicates higher similarity (better quality). We average the NMI score over 10 runs for each single experiment.

For SLPA, we set $T = 100$ and $\tau = 0.2$. For rSLPA, we have conducted some pivot experiments for choosing the iteration number of rSLPA.

3) Convergence Speed: For rSLPA, we vary T from 100 to $1,000$ to test after how many iterations, the result will converge. Considering that the graph size N may also affect the convergence speed, we vary the graph size in [10K, 20K, 50K].

Figure 1a shows the convergence speed of rSLPA. We can see that for different N, it gives a relatively stable result when $T \geq 200$. According to this result, we will use $T = 200$ for rSLPA in the rest experiments.

4) Community Quality: In this part we compare the detected community quality of SLPA and rSLPA under different parameter settings. We vary the value of each single parameter N, k, μ, om, on, and compare the NMI scores.

In Figure 1b we vary the graph size N from $10,000$ to $50,000$. Both algorithms have very high and stable scores, and the difference between two algorithms is small. In Figure 1c we vary the average degree k from 10 to 70, which covers sparse and dense graphs. As k increases, the score of each algorithm grows gradually, and remains unchanged when k is large enough (≥ 50). Both algorithms have high scores over dense and sparse graphs.

In Figure 1d, we increase the mixing parameter μ from 0.1 to 0.3. The score of SLPA is nearly unchanged as μ increases. On the other hand, the score of rSLPA also remains at a high level, but it drops slowly as μ increases, which means rSLPA has less ability to detect better-mixed communities. In Figure 1e, om increases from 2 to 5, which means each overlapping vertex belongs to more and more communities simultaneously. The scores of both algorithms decrease slowly as om increases. The reason is that it becomes harder to correctly assign a vertex when it belongs to more communities. Compared to SLPA, rSLPA has better performance when $om \geq 3$. This means for a single vertex, rSLPA keeps more information about its belongingness, so that rSLPA can correctly detect more memberships of a vertex. In Figure 1f, the number of overlapping vertices is increased from $0.1N$ to $0.3N$. With the increase of overlapping vertices, the performance of both algorithms becomes worse. The reason is that the boundary between communities becomes fuzzier and harder to detect.

Overall, SLPA and rSLPA keep high scores on different situations. When om is large, rSLPA can benefit from more detailed information it keeps, and have better performance.

TABLE II
STATISTICS OF DATASET EU-2015-TPD.

Statistics	Value
# nodes	6,650,532
# edges	170,145,510
avg. degree	25.584

B. Real-World Dataset

In the sequel, we use real-world dataset to test the efficiency of SLPA and rSLPA on Spark [10], a modern cluster-computing framework based on MapReduce, to investigate its performance in a distributed dynamic scenario. We use 7 Linux servers, each with 125 GBytes of main memory and 2 CPU of Intel Xeon Processor E5-2630 v3. Both algorithms are implemented in Scala.

1) Preparing the data set: We use a public dataset *eu-2015-tpd*, which can be downloaded from law.di.unimi.it. This dataset consists Web pages in Europe countries crawled in 2015. Statistics of this dataset are given in Table II.

To test the incremental algorithm, we generate the graph edit batch by randomly selecting edges for insertion and deletion. Typically, the batch size is set from 100 to 100,000, and then for each size we randomly pick half edges to insert and half to delete. We run rSLPA on the original graph for 200 iterations, then apply edge insertion and deletion to the graph, and finally run the incremental updating algorithm to get the running time.

2) Evaluation Metrics: We use the actual running time (wall clock time) to measure the efficiency of the algorithms. For both algorithms, the post-processing part can be done separately from the label propagation part. For example, if we run rSLPA on a social network, we may not want to calculate the communities in every minute, instead, we can let the algorithm handling changes continuously, and calculate the communities once per hour. Thus, in this part, we compare the running time of label propagation and post-processing of each algorithm separately.

3) Results: Figure 2 compares the running time of SLPA and rSLPA on real static graphs. In the label propagation stage, rSLPA is more than two times as fast as SLPA. Considering that rSLPA runs for 200 iterations while SLPA runs for 100 iterations, SLPA is over five times more than rSLPA in terms of running time per iteration. However, in the post-processing stage, SLPA takes much less time than rSLPA does. This is because in rSLPA, complex operations like finding connected components are needed to find communities. Overall, rSLPA is a bit faster than SLPA in terms of the total running time.

Figure 3 compares the running time of incremental updating and running from scratch of rSLPA on different batch sizes. The results show that rSLPA can perform efficient incremental updating. The increase of running time is sublinear to the increase of batch size. This is because multiple edges can have the same influence on a single label, and this overlapping is more frequent with more edges changed. This also implies that rSLPA is suitable for edit batches of large size.

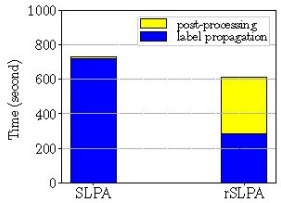


Fig. 2. Running time of SLPA and rSLPA on real static graphs.

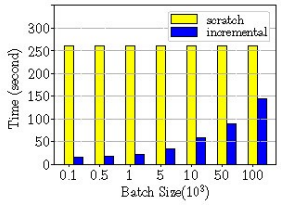


Fig. 3. Running time of rSLPA (incremental updating and calculating from scratch) on different batch sizes.

IV. CONCLUSION

In this paper, we study the problem of overlapping community detection on distributed and dynamic graphs. We propose a new algorithm rSLPA by relaxing the probability distribution in the label propagation stage. To our best knowledge, rSLPA is the first algorithm that can incrementally detect overlapping communities over distributed and dynamic graphs accurately. We conduct extensive experiments to confirm the good performance of our approaches on synthetic/real data.

V. ACKNOWLEDGEMENT

The work is partially supported by the Hong Kong RGC GRF Project 16202215, National Grand Fundamental Research 973 Program of China under Grant 2014CB340303, the National Science Foundation of China (NSFC) under Grant No. 61729201, Science and Technology Planning Project of Guangdong Province, China, No. 2015B010110006, Huawei Co. Ltd Collaboration Project, YBCB2009041-45, Microsoft Research Asia Collaborative Research Grant, NSFC Guang Dong Grant No. U1301253, Lian Start Up No. 220981, Kent State University, and NSF OAC No. 1739491.

REFERENCES

[1] G. Palla, I. Derényi, I. Farkas, and T. Vicsek, "Uncovering the overlapping community structure of complex networks in nature and society," *Nature*, vol. 435, no. 7043, 2005.

[2] J. M. Kumpula, M. Kivelä, K. Kaski, and J. Saramäki, "Sequential algorithm for fast clique percolation," *Physical Review E*, vol. 78, no. 2, 2008.

[3] A. Lancichinetti, S. Fortunato, and J. Kertész, "Detecting the overlapping and hierarchical community structure in complex networks," *New Journal of Physics*, vol. 11, no. 3, 2009.

[4] Y. Zhang, J. Wang, Y. Wang, and L. Zhou, "Parallel community detection on large networks with propinquity dynamics," in *SIGKDD*, 2009.

[5] J. Xie and B. K. Szymanski, "Towards linear time overlapping community detection in social networks," in *PAKDD*, 2012.

[6] R. Cazabet, F. Amblard, and C. Hanachi, "Detection of overlapping communities in dynamical social networks," in *2010 IEEE Second International Conference on Social Computing (SocialCom)*, 2010.

[7] X. Jian, X. Lian, and L. Chen, "On Efficiently Detecting Overlapping Communities over Distributed Dynamic Graphs," *ArXiv e-prints*, Jan. 2018.

[8] L. Chitnis, A. Das Sarma, A. Machanavajjhala, and V. Rastogi, "Finding connected components in map-reduce in logarithmic rounds," in *ICDE*, 2013.

[9] A. Lancichinetti and S. Fortunato, "Benchmarks for testing community detection algorithms on directed and weighted graphs with overlapping communities," *Physical Review E*, vol. 80, no. 1, 2009.

[10] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets." *HotCloud*, vol. 10, no. 10-10, 2010.