Fast Incremental Community Detection on Dynamic Graphs

Anita Zakrzewska(✉) and David A. Bader

Computational Science and Engineering, Georgia Institute of Technology, Atlanta, GA, USA
`azakrzewska3@gatech.edu`

Abstract. Community detection, or graph clustering, is the problem of finding dense groups in a graph. This is important for a variety of applications, from social network analysis to biological interactions. While most work in community detection has focused on static graphs, real data is usually dynamic, changing over time. We present a new algorithm for dynamic community detection that incrementally updates clusters when the graph changes. The method is based on a greedy, modularity maximizing static approach and stores the history of merges in order to backtrack. On synthetic graph tests with known ground truth clusters, it can detect a variety of structural community changes for both small and large batches of edge updates.

Keywords: Graphs · Community detection · Graph clustering · Dynamic graphs

1 Introduction

Graphs are used to represent a variety of relational data, such as internet traffic, social networks, financial transactions, and biological data. A graph $G = (V, E)$ is composed of a set of vertices V, which represent entities, and edges E, which represent relationships between entities. The problem of finding dense, highly connected groups of vertices in a graph is called community detection or graph clustering. In cases where communities are non-overlapping and cover the entire graph, the term community partitioning may also be used. Many real world graphs contain communities, such as groups of friends on social networks, lab colleagues in a co-publishing graph, or related proteins in biological networks. In this work, we present a new algorithm for incremental community detection on dynamic graphs.

1.1 Related Work

Most work in community detection has been done for static, unchanging graphs. Popular methods include hierarchical vertex agglomeration, edge agglomeration, clique percolation, spectral algorithms, and label propagation. Details of a variety of approaches can be found in the survey by Fortunato [11]. The quality of a community C is often measured using a fitness function. Modularity, shown in Eq. 1, is a popular measure that compares the number of intra-community edges to the expected number under a random null model [16]. Let k_{in}^{C} be the sum of edge weights for edges with both endpoint vertices in C and k_{out}^{C} be the sum of edge weights for edges with only one endpoint in C.

$$Q(C) = \frac{1}{|E|}(k_{in}^{C} - \frac{(k_{in}^{C} + k_{out}^{C})^2}{4\,|E|}) \tag{1}$$

CNM [6] is a hierarchical, agglomerative algorithm that greedily maximizes modularity. Each vertex begins as its own singleton community. Communities are then sequentially merged by contracting the edge resulting in the greatest increase in modularity. Riedy *et al.* present a parallel version of this algorithm in which a weighted maximal matching on edges is used [19].

Real graphs evolve over time. Many works study community detection on dynamic graphs by creating a sequence of data snapshots over time and clustering each snapshot. Chakrabarti *et al.* [5] introduce evolutionary clustering and GraphScope uses block modeling and information compression principles [21]. There are many other works which study the evolution of communities in changing graphs [9,14,22]. However, these are not incremental methods. A second category of algorithms incrementally updates previously computed clusters when the underlying graph changes. By reusing previous computations, incremental algorithms can run faster, which is critical for scaling to large datasets. This is especially useful for applications in which community information must be kept up to date for a quickly changing graph and low latency is desired. Moreover, incremental approaches can result in smoother community transitions.

Duan *et al.* [10] present an incremental algorithm for finding k-clique communities [18] in a changing graph and Ning *et al.* [17] provide a dynamic algorithm for spectral partitioning. Other algorithms search for emerging events in microblog streams [3,4]. Dinh *et al.* [7,8] present an incremental updating algorithm to modularity based CNM. First, standard CNM is run on the initial graph. For each graph update, every vertex that is an endpoint of an inserted edge is removed from its community and placed into a new singleton community. The standard, static CNM algorithm is then restarted and communities may be further merged to increase modularity.

The work most closely related to ours is by Görke *et al.* [13], in which the authors also present two dynamic algorithms that update a CNM based clustering. In their first algorithm, endpoint vertices of newly updated edges, along with a local neighborhood, are freed from their current cluster and CNM is restarted. The second algorithm stores the dendrogram of cluster merges formed by the initial CNM clustering. When the graph changes, the algorithm backtracks cluster merges using this dendrogram until specified conditions are met. If an intra-community edge is inserted, it backtracks till the two endpoint vertices are in separate communities. If an intra-community edge is deleted or an inter-community edge is inserted, it backtracks till both endpoints are their own singleton communities. No backtracking occurs on a inter-community edge deletion. After communities have been modified, CNM is restarted.

1.2 Contribution

In this work we present two incremental algorithms that update communities when the underlying graph changes. The first version, *BasicDyn* is similar to previous work by Görke *et al.* [13]. Our second algorithm, *FastDyn*, is a modification of *BasicDyn* that runs faster. We use various synthetic dynamic graphs to test the quality of our methods. Both *BasicDyn* and *FastDyn* are able to detect community changes.

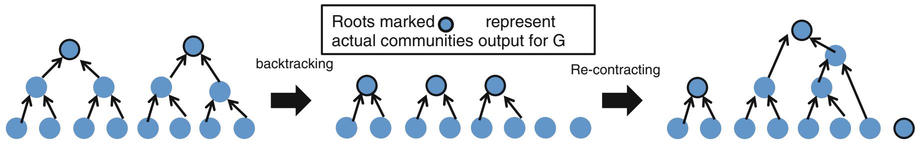

Fig. 1. The left image shows $Forest_G$ created after agglomeration. The center image shows the $Forest_G$ after backtracking undos merges using *BasicDyn* or *FastDyn*. Right shows after re-starting agglomeration using *Agglomerate* or *FastAgglomerate*.

2 Algorithm

The basis of our algorithm is the parallel version of CNM presented by Riedy *et al.* [19]. We will denote this by *Agglomerate*. Each vertex in the graph G is initialized as its own singleton community, forming the community graph G_{comm}. Note that while G_{comm} is initially the same as G, each vertex of G_{comm} represents a group or cluster of vertices in G. The weighted degree of a vertex $c \in G_{comm}$ corresponds to k_{out}^c and the weight assigned to c (initially 0), corresponds to the intra-community edges k_{in}^c.

Next, the following three steps are repeated until the termination criterion is reached. (1) For each edge in G_{comm}, the change in modularity resulting from an edge contraction is computed. Given an edge (c_1, c_2) in G_{comm}, the change in modularity is given by the value $\Delta Q(c_1, c_2) = Q(c_1 \cup c_2) - (Q(c_1) + Q(c_2))$, which can be easily computed using Eq. 1. This score is assigned to the edge. (2) A weighted maximal matching is computed on the edges of G_{comm} using these scores. (3) Edges in the maximal matching are contracted in parallel. An edge contraction merges the two endpoint vertices into one vertex. Termination occurs when no edge contraction results in a great enough modularity increase.

All vertices that have been merged together by edge contractions correspond to a single community and so by repeatedly merging vertices in G_{comm}, we create ever larger communities in G. The history of these contractions can be stored as a forest of merges. Pairs of vertices in G_{comm} that were merged together by an edge contraction are children of the same parent. The root of each tree corresponds to the final communities found for G. We refer to this forest by $Forest_G$. Edge contraction scores can be computed in $\mathcal{O}(|E|)$ time. If the maximal matching is also computed in $\mathcal{O}(|E|)$, then the complexity of *Agglomerate* is $\mathcal{O}(K * |E|)$, where K is the number of contraction phases. In the best case, the number of communities halves in each phase and the complexity is $\mathcal{O}(log(|V|) * |E|)$. In the worst case, the graph has a star formation and only one edge contracts in each phase, resulting in a complexity of $\mathcal{O}(|V| * |E|)$.

Our dynamic algorithm begins with the initial graph G and runs the *Agglomerate* algorithm to create $Forest_G$. We then apply a stream of edge insertions and deletions to G. These updates can be applied in batches of any size. Small batch sizes are used for low latency applications, while aggregating larger batches can result in a relatively lower overall running time. For each batch of updates, our algorithm uses the history of merges in $Forest_G$ and undoes certain merges that previously had taken place. This breaks apart certain previous communities and un-contracts corresponding vertices in G_{comm}. Next, the *Agglomerate* algorithm is restarted and new modularity increasing merges may be performed. Figure 1 shows this process. Next we describe two versions of the dynamic algorithm, each of which follows the basic steps described.

BasicDyn: When a new batch of edge insertions and deletions is processed, each vertex that is an endpoint of any edge in the batch is marked. Merges are then backtracked in $Forest_G$ until each marked vertex is in its own singleton community. After backtracking, *Agglomerate* is restarted to re-merge communities.

FastDyn: The *FastDyn* method is based on *BasicDyn*, but has two modifications that improve computational speed. The first is that backtracking of previous merges in $Forest_G$ occurs under more stringent conditions. Merges are only undone if the quality of the merge, as measured by the induced change in modularity, has significantly decreased compared to when the merge initially took place. Because merges may be performed in parallel using a weighted maximal matching on edges, not every vertex $c \in G_{comm}$ merges with its highest scoring neighbor $best(c)$. When a vertex $c \in G_{comm}$ merges at time t_i, we store both the score of that merge $\Delta Q_{t_i}(c, match(c))$ and the score of the highest possible merge c could have participated in $\Delta Q_{t_i}(c, best_{t_i}(c))$. After a batch of edge updates is applied at time t_{curr} and the graph G changes, the best and actual merge scores are rechecked. The score of a merge has significantly decreased if: $\Delta Q_{t_i}(c, best_{t_i}(c)) - \Delta Q_{t_i}(c, match(c)) + threshold < \Delta Q_{t_{curr}}(c, best_{t_{curr}}(c)) - \Delta Q_{t_{curr}}(c, match(c))$. This occurs if either the score of the merge taken has decreased or the score of the best possible merge has increased.

For each merge evaluated, *FastDyn* checks the above condition and if it is met, $Forest_G$ is backtracked to undo the merge. Merges that are evaluated are those that occur between clusters that contain at least one member that is directly affected by the batch of edge updates (is an endpoint of an newly inserted

```
Data: $G_{comm}$ and $Forest_{G,prev}$
while contractions occuring do
    for $v \in G_{comm}$ in parallel do
        $paired[v] = 0$;
        $match[v] = v$;
    end
    while matches possible do
        for $v \in G_{comm}$ s.t. $paired[v] == 0$ in parallel do
            $max = 0$, $match[v] = v$;
            for neighbors $w$ of $v$ s.t. $paired[w] == 0$ do
                if $\Delta Q(v, w) > max$ then
                    $max = \Delta Q(v, w)$;
                    $match[v] = w$;
                end
            end
        end
        for $v \in G_{comm}$ s.t. $paired[v] == 0$ in parallel do
            if $match[v] > 0$ and ($match[match[v]] == v$ or
            $prevroot[v] == prevroot[match[v]]$) then
                $paired[v] = 1$;
            end
        end
    end
    for $v \in G_{comm}$ in parallel do
        Pointer jump match array to root;
        $dest[v]$=root;
    end
    Contract in parallel all vertices in $G_{comm}$ with same value of dest;
end
```

Algorithm 1. The *FastAgglomerate* algorithm used by *FastDyn*. *prevroot* labels each vertex by its root in $Forest_{G,prev}$, which corresponds to its previous community membership.

or deleted edge). In $Forest_G$, this corresponds to leaf nodes that represent such endpoint vertices as well as any upstream parent of such a leaf node.

The second modification of *FastDyn* occurs in the re-merging step. Instead of running *Agglomerate* after backtracking merges, *FastDyn* runs a modified method, which we will call *FastAgglomerate*. Because *Agglomerate* only allows two vertices to merge together in a contraction phase, star-like formations containing n vertices take n phases to merge, which results in a very long running time. We have found that backtracking merges creates such structures so that running *Agglomerate* after backtracking results in a very long tail of contractions, each of which only contacts a small number of edges. We address this problem with *FastAgglomerate*, which uses information about previous merges to speed up contractions. Instead of performing a maximal edge matching, *FastAgglomerate* allows more than two vertices of G_{comm} to contract together if in the previous time step these vertices eventually ended up contracted into the same community (if they correspond to nodes in $Forest_G$ that previously shared the same root). In the static case, merging several vertices together in one contraction phase could lead to deteriorating results. $FastAgglomerate$ is able to do this, however, because it uses information from the merges of the previous time step. Intuitively, merges that previously occurred are more likely to be acceptable later. Pseudocode is given in Algorithm 1. In the worst case, both $BasicDyn$ and $FastDyn$ will backtrack enough to undo many or most merges. In this case, the running time is the same as that of $Agglomerative$. For small updates, the dynamic algorithms run faster in practice. $FastDyn$ decreases the number of contraction steps needed by allowing several vertices to merge in a single step.

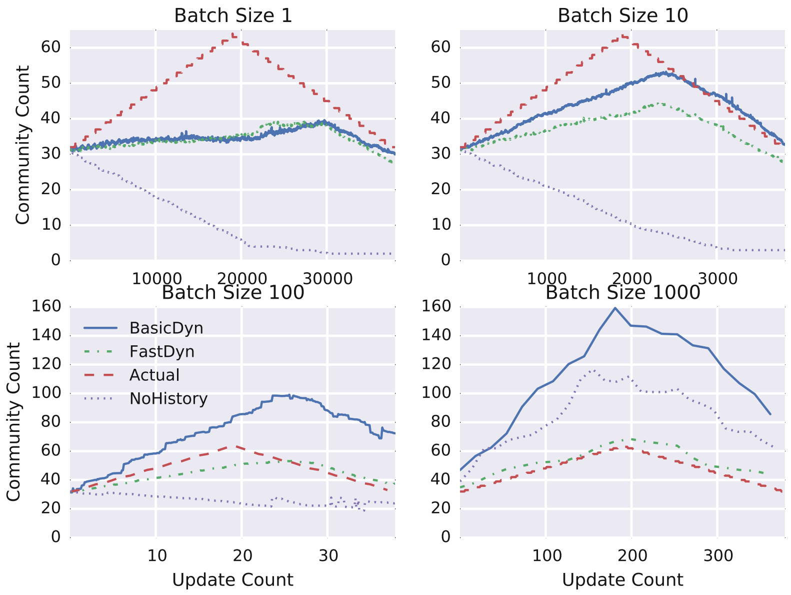

Fig. 2. Results for the synthetic test of splitting and merging clusters are shown. The actual and detected number of communities across time are plotted for batch sizes of 1, 10, 100, and 1000. An alternative approach is shown and labeled "NoHistory" (Color figure online).

3 Results

Dynamic community detection is only useful if the algorithm is able to detect structural changes. We evaluate our algorithm using two tests. First, we form a ring of 32 communities, each with 32 vertices. Each vertex in a community is on average connected to 16 other vertices in the cluster. Over time, edges are both inserted and removed so that each of the 32 clusters consecutively splits in two. Once each community has been split, edges are then updated so that pairs of clusters merge together (different pairs than at the start), resulting in 32 clusters again. Figure 2 shows the number of communities detected over time by *BasicDyn* with the solid blue line, the number detected by *FastDyn* with the green dots and dashes, and the actual number with the dashed red line. Because graphs are randomly generated, each curve is the average of 10 runs. The x-axis represents the number of updates that have been made to the graph. Batch sizes used are 1,10,100, and 1000. A batch size of n means that n edge insertions and deletions are accumulated before the communities are incrementally updated. Since the curves rise and then fall, both algorithms are able to detect first splits and then merges. As the batch size increases, the performance of *FastDyn* improves, which is expected because a larger batch size allows more communities to be broken apart during backtracking and the algorithm has more options for how to re-contract communities. Therefore, the algorithm can output results closer to those that would be found with a static recomputation beginning from scratch. We also compare the ability of *FastDyn* and *BasicDyn* to detect community changes to the dynamic algorithm from [20], which also uses greedy modularity maximization. After a batch of edges is inserted and removed, this method removes each vertex with an updated edge from its current community into its own singelton community. Agglomeration is then restarted. The results of this alternative approach are shown in Fig. 2 with the purple dotted line labeled "NoHistory". Unlike the backtracking approaches, community splits and merges are only detected for a large batch size of 1000.

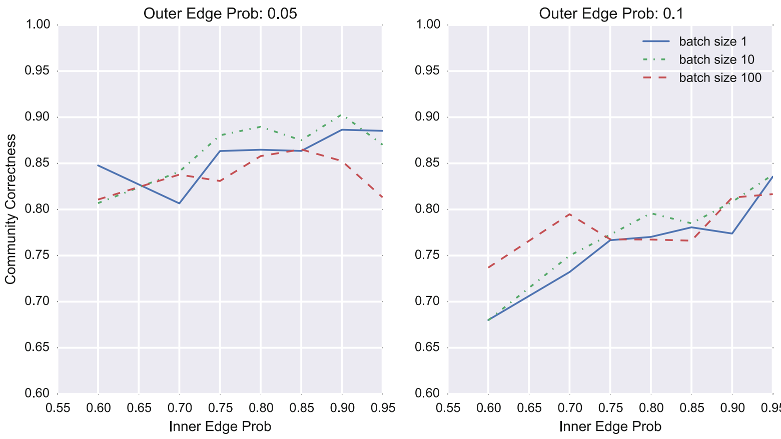

Fig. 3. Results for *FastDyn* on the second synthetic test using a shifting stochastic block model are shown. The y-axis shows the quality of output and x-axis shows the probability of an intra-community edge.

The second test is run on a graph formed using a stochastic block model [15]. We generate two separate graphs, blockA and blockB, each with two communities in which the probability of an intra-community edge and that of an inter-community edge are both specified. BlockB uses the same parameters as blockA, but with the vertex members of each community changed. Half of the vertices that were in community one in blockA move to the second community of blockB and half of those that were in community two of blockA move to community one of blockB. We begin by running *Agglomerate* on the entire blockA. Then, in a stream of updates, we remove edges from blockA and add edges from blockB. At the end of the stream, the graph will equal blockB. This test differs from the first test in an important way. In test one, the algorithm could always detect some number of distinct communities. Here, distinct communities only exist near the beginning and end of the stream, while in the middle the graph is a mixture of blockA and blockB. The updates to the graph in this test are randomly distributed across all vertices so that much of $Forest_G$ may be affected. In contrast, the updates in test one were applied to one cluster at a time. Lastly, test one specifically tests splits and merges, while test two uses a general rearrangement. Figure 3 shows how well *FastDyn* is able to distinguish the two communities of blockB at the end of the stream. Communities are detected for each batch, but we only evaluate at the end of the stream since that is when the graph returns to a known block structure. The x-axis varies the probability of an intra-community edge. Unsurprisingly, the algorithm performs better when the communities in blockA and blockB are denser. The y-axis shows the correctness of *FastDyn*, which is determined as follows. For each vertex v in the graph, we compare its actual community in blockB to its community output by *FastDyn*. The Jaccard index then measures the normalized overlap between the two sets, with 1 measuring perfect overlap, and 0 no overlap. The average Jaccard index across all vertices measures the algorithm correctness. Because blockA and blockB are randomly generated, each point plotted is an average of 50 separate runs on different graphs. The two synthetic tests described above show that *FastDyn* is able to distinguish structural community changes. Synthetic tests are useful because communities have ground truth and known changes can be applied.

Table 1. The time to compute the initial static community detection and average time to update with incremental algorithms is shown.

Graph	Initial	Batch size	FastDyn	BasicDyn
Facebook	3.5 s	10	0.1 s	0.2 s
		1000	0.7 s	4.9 s
DBLP	122 s	10	7.1 s	21.7 s
		1000	72 s	101 s
Slashdot	45 s	10	0.6 s	2.1 s
		1000	7.5 s	289 s

Table 1 shows the running time of the initial static community detection used by the dynamic algorithms and the average running time to process a batch of size 10 and 1000 edges for *FastDyn* and *BasicDyn*. The algorithms were tested on a Slashdot graph [12], a DBLP graph [1,24], and a Facebook wall post graph [2,23]. The *FastDyn* algorithm is faster than *BasicDyn* for all graphs and batch sizes. The speedup of using a dynamic algorithm is greater for small batch sizes because the incremental algorithms perform less work, while running time for static computation remains the same. Therefore, an incremental approach is most useful for monitoring applications where community results must be updated after a small number of data changes.

4 Conclusion

In this work, we present two incremental community detection algorithms for dynamic graphs. Both use hierarchical, modularity maximizing clustering as their base and update results by storing the history of previous merges in a forest. While *BasicDyn* is similar in nature to a previous approach, *FastDyn* has two improvements that significantly improve running time while maintaining clustering quality. *FastDyn* is faster than *BasicDyn* on all real graphs tested for both small and large update batch sizes. Incrementally updating allows for speedup over recalculating from scratch whenever the graph changes, especially for small batch sizes when low latency updates are needed. While the algorithm is based off a parallel static algorithm, and can be parallelized, the focus of this work was to study improvements in performance due to incremental updates. Parallel scalability is a topic for future work.

Acknowledgments. The work depicted in this paper was sponsored by Defense Advanced Research Projects Agency (DARPA) under agreement #HR0011-13-2-0001. The content, views and conclusions presented in this document do not necessarily reflect the position or the policy of DARPA or the U.S. Government, no official endorsement should be inferred.

References

1. Dblp co-authorship network dataset – KONECT, May 2015. http://konect.unikoblenz.de/networks/com-dblp
2. Facebook wall posts network dataset – KONECT, May 2015. http://konect.unikoblenz.de/networks/facebook-wosn-wall
3. Agarwal, M.K., Ramamritham, K., Bhide, M.: Real time discovery of dense clusters in highly dynamic graphs: identifying real world events in highly dynamic environments. Proc. VLDB Endowment **5**(10), 980–991 (2012)
4. Angel, A., Sarkas, N., Koudas, N., Srivastava, D.: Dense subgraph maintenance under streaming edge weight updates for real-time story identification. Proc. VLDB Endowment **5**(6), 574–585 (2012)
5. Chakrabarti, D., Kumar, R., Tomkins, A.: Evolutionary clustering. In: Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 554–560. ACM (2006)
6. Clauset, A., Newman, M.E., Moore, C.: Finding community structure in very large networks. Physical Rev. E **70**(6), 066111 (2004)

7. Dinh, T.N., Shin, I., Thai, N.K., Thai, M.T., Znati, T.: A general approach for modules identification in evolving networks. In: Hirsch, M.J., Pardalos, P.M., Murphey, R. (eds.) Dynamics of Information Systems. Springer Optimization and Its Applications, vol. 40, pp. 83–100. Springer, Heidelberg (2010)
8. Dinh, T.N., Xuan, Y., Thai, M.T.: Towards social-aware routing in dynamic communication networks. In: 2009 IEEE 28th International Performance Computing and Communications Conference (IPCCC), pp. 161–168. IEEE (2009)
9. Duan, D., Li, Y., Jin, Y., Lu, Z.: Community mining on dynamic weighted directed graphs. In: Proceedings of the 1st ACM International Workshop on Complex Networks Meet Information & Knowledge Management, pp. 11–18. ACM (2009)
10. Duan, D., Li, Y., Li, R., Lu, Z.: Incremental k-clique clustering in dynamic social networks. Artif. Intell. Rev. **38**(2), 129–147 (2012)
11. Fortunato, S.: Community detection in graphs. Phys. Rep. **486**(3), 75–174 (2010)
12. Gómez, V., Kaltenbrunner, A., López, V.: Statistical analysis of the social network and discussion threads in slashdot. In: Proceedings of the 17th International Conference on World Wide Web, pp. 645–654. ACM (2008)
13. Görke, R., Maillard, P., Schumm, A., Staudt, C., Wagner, D.: Dynamic graph clustering combining modularity and smoothness. J. Exp. Algorithmics (JEA) **18**, 1–5 (2013)
14. Greene, D., Doyle, D., Cunningham, P.: Tracking the evolution of communities in dynamic social networks. In: 2010 International Conference on Advances in Social Networks Analysis and Mining (ASONAM), pp. 176–183. IEEE (2010)
15. Holland, P.W., Laskey, K.B., Leinhardt, S.: Stochastic blockmodels: first steps. Soc. Netw. **5**(2), 109–137 (1983)
16. Newman, M.E., Girvan, M.: Finding and evaluating community structure in networks. Phys. Rev. E **69**(2), 026113 (2004)
17. Ning, H., Xu, W., Chi, Y., Gong, Y., Huang, T.S.: Incremental spectral clustering by efficiently updating the eigen-system. Pattern Recogn. **43**(1), 113–127 (2010)
18. Palla, G., Derényi, I., Farkas, I., Vicsek, T.: Uncovering the overlapping community structure of complex networks in nature and society. Nature **435**(7043), 814–818 (2005)
19. Riedy, E.J., Meyerhenke, H., Ediger, D., Bader, D.A.: Parallel community detection for massive graphs. In: Wyrzykowski, R., Dongarra, J., Karczewski, K., Waśniewski, J. (eds.) PPAM 2011, Part I. LNCS, vol. 7203, pp. 286–296. Springer, Heidelberg (2012)
20. Riedy, J., Bader, D., et al.: Multithreaded community monitoring for massive streaming graph data. In: 2013 IEEE 27th International Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), pp. 1646–1655. IEEE (2013)
21. Sun, J., Faloutsos, C., Papadimitriou, S., Yu, P.S.: Graphscope: parameter-free mining of large time-evolving graphs. In: Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 687–696. ACM (2007)
22. Tantipathananandh, C., Berger-Wolf, T., Kempe, D.: A framework for community identification in dynamic social networks. In: Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 717–726. ACM (2007)

23. Viswanath, B., Mislove, A., Cha, M., Gummadi, K.P.: On the evolution of user interaction in facebook. In: Proceedings of the 2nd ACM Workshop on Online Social Networks, pp. 37–42. ACM (2009)
24. Yang, J., Leskovec, J.: Defining and evaluating network communities based on ground-truth. Nature **42**(1), 181–213 (2015)