Data Min Knowl Disc (2016) 30:819–847
DOI 10.1007/s10618-015-0440-z

SONIC: streaming overlapping community detection

Ahmet Erdem Sarıyüce[1] · Buğra Gedik[2] · Gabriela Jacques-Silva[3] · Kun-Lung Wu[3] · Ümit V. Çatalyürek[4]

Received: 11 December 2014 / Accepted: 19 October 2015 / Published online: 3 November 2015
© The Author(s) 2015

Abstract A community within a graph can be broadly defined as a set of vertices that exhibit high cohesiveness (relatively high number of edges within the set) and low conductance (relatively low number of edges leaving the set). Community detection is a fundamental graph processing analytic that can be applied to several application domains, including social networks. In this context, communities are often *overlapping*, as a person can be involved in more than one community (e.g., friends, and family); and evolving, since the structure of the network changes. We address the problem of streaming overlapping community detection, where the goal is to maintain communities in the presence of streaming updates. This way, the communities can be updated more efficiently. To this end, we introduce SONIC—a find-and-merge type of community detection algorithm that can efficiently handle streaming updates.

Responsible editors: G. Karypis.

✉ Ahmet Erdem Sarıyüce
asariyu@sandia.gov

Buğra Gedik
bgedik@cs.bilkent.edu.tr

Gabriela Jacques-Silva
g.jacques@us.ibm.com

Kun-Lung Wu
klwu@us.ibm.com

Ümit V. Çatalyürek
umit@bmi.osu.edu

1 Sandia National Labs, Livermore, CA, USA

2 Bilkent University, Ankara, Turkey

3 IBM Thomas J. Watson Research Center, IBM Research, New York, USA

4 The Ohio State University, Columbus, USA

SONIC first detects when graph updates yield *significant* community changes. Upon the detection, it updates the communities via an *incremental* merge procedure. The SONIC algorithm incorporates two additional techniques to speed-up the incremental merge; *min-hashing* and *inverted indexes*. Results show that SONIC can provide high quality overlapping communities, while handling streaming updates several orders of magnitude faster than the alternatives performing from-scratch computation.

Keywords Streaming graph processing · Community detection · Overlapping communities

1 Introduction

In many application domains, graphs are used to represent relationships between people, systems, and the physical world. Data analytics performed on these graphs can bring new business insights and improve decision-making. For instance, the graph structure may represent the relationships in a social network, where finding communities in the graph (Lancichinetti and Fortunato 2009) can facilitate targeted advertising. As another example, in the Telecommunications domain, call details records can be used to capture the call relationships between people (Nanavati et al. 2006), and locating closely connected groups of people can help generate promotions.

As these examples illustrate, a fundamental graph analytic is *community detection*. We can define a community within a graph as a set of vertices that exhibit high *cohesiveness* and low *conductance*. High cohesiveness means that the vertices in the community have relatively high number of edges connecting them, and low conductance means that the vertices in the community have relatively small number of edges going outside of the community.

Communities in social networks have two key characteristics. The first is that communities are *overlapping*, as different communities can have common users. This is a typical scenario, as a single user can be involved in different communities, such as co-workers, friends, and family. The second is that communities are *dynamic*. They evolve as a result of the continuous interactions between people. These interactions can result in the addition/removal of new/existing relationships in the network. For instance, the follower-followee graph of Twitter (2014) is highly active, with millions of updates to the graph structure every day. This number is even higher if we consider the mention graph of Twitter. It is also common to analyze the graph over a recent time window, such as the mention graph of Twitter over the last week. In such scenarios, both insertions and removals are equally frequent.

In this paper, we present SONIC—an algorithm to detect overlapping communities on dynamic graphs in a *streaming* manner. Upon each edge insertion or removal, we *incrementally* maintain the overlapping communities. This way, the communities are updated more efficiently and without the need for periodic re-computations that are typically performed in batch. SONIC maintains multiple community ids for each vertex and updates these ids upon edge insertions and removals. By doing so, it can answer any query for the communities of a given vertex (or a set of vertices) by a simple traversal.

SONIC adopts the find-and-merge style of community detection. In find-and-merge style algorithms (Coscia et al. 2012; Rees and Gallagher 2010), local communities of each vertex are found first, as part of the *find step*. These local communities are then merged into global communities based on a configurable merge condition, as part of the *merge step*. SONIC uses the label propagation algorithm to detect local communities during the find step. In the label propagation algorithm (Raghavan et al. 2007), each vertex is initially assigned a unique id. Then each vertex gets the most commonly used id of its neighbors. This procedure continues either for a specified number of rounds or until there are no changes in ids. After that, SONIC merges local communities based on a given merge factor. Different than earlier works, SONIC incorporates an *incremental merge* step to avoid rebuilding of global communities from scratch.

SONIC faces several challenges in tackling the streaming overlapping community detection problem. First, in a streaming setup, the number of updates are very high, yet many of these updates are not sufficiently important by themselves to result in any change in the global community structure. Fully processing each one of these updates will unnecessarily increase the cost of the solution. SONIC addresses this problem by detecting updates that are *significant* via a fast procedure that involves re-adjusting only the local communities. SONIC initiates the merge process only for the significant updates, effectively reducing the cost of an edge update. Second, the merge step is non-trivial to perform incrementally, which led earlier work on find-and-merge style of algorithms to use a non-incremental merge (Coscia et al. 2012). SONIC solves this problem by maintaining the global communities as a collection of sub-communities. Upon an edge insertion/removal, it detects the global communities whose sub-communities are impacted, dissolves such global communities, and regenerates the global community structure by a partial merge. Finally, even an incremental merge algorithm can be costly to execute when the local changes cascade to bring about a major change in the global communities. To address this issue, SONIC incorporates two alternative merge strategies: (i) a *min-hash* based and (ii) an *inverted index* based merge.

In summary, this paper contributes the following:

- The SONIC algorithm for incremental overlapping community detection over dynamic graphs.
- A technique to detect significant changes in local community structures, in order to avoid costly merges when the local community changes are unlikely to cause global community changes.
- Inverted index and min-hash based techniques to further accelerate the incremental merge used in SONIC.
- An extensive experimental evaluation of SONIC on real-world and synthetic data sets, with respect to quality and running time performance.

The rest of this paper is organized as follows. Section 2 overviews related work. Section 3 gives the background on basic techniques from the literature. Section 4 lists fundamental theoretical properties that we leverage for developing the SONIC algorithm. Section 5 describes the base version of the SONIC algorithm with an illustrative example. Section 6 presents several improvements over base SONIC, such as the significant change detection, the min-hash based merge, and the inverted index

based merge algorithms. Section 7 presents our experimental evaluation and Sect. 8 concludes the paper.

2 Related work

Vast amount of work has been done on community detection and various aspects of the problem have been studied in the literature. Fortunato (2010) covers all the popular techniques to find communities in complex networks. Leskovec et al. (2010) compares different community detection algorithms empirically. Significant number of spectral methods are based on the *modularity* metric proposed by Newman (2006). There are also information theoretic approaches to discover community structure of networks. Particularly, Infomap (Rosvall and Bergstrom 2008) is currently one of the best performing non-overlapping community detection algorithms. As an alternative technique to community detection, past works have proposed community search, where the communities are detected locally based on given query vertices. This idea first appeared in Hildrum and Yu (2005) and was further investigated in Sozio and Gionis (2010) and Padrol-Sureda et al. (2010). Recently, Cui et al. (2013) proposed online search of overlapping communities based on clique adjacency graphs. Our approach incrementally maintains the communities, in the tradition of continuous queries; whereas (Cui et al. 2013) answers community search requests on demand, in the tradition of snapshot queries.

In our work, we are particularly interested in (i) overlapping community detection techniques, and (ii) dynamic methods that can handle streaming updates.

Overlapping Community detection in social networks is different from the classical clustering and partitioning problems in which the identified clusters/partitions do not overlap (i.e., each vertex belongs to a single cluster). In contrast, communities in social networks often overlap. Palla et al. (2005) showed that most real networks have overlapping community structure. They also introduced a percolation based method to detect overlapping communities. A recent survey (Xie et al. 2013) summarizes most of the existing overlapping community detection algorithms and categorizes them into five classes: (1) Clique percolation (Palla et al. 2005), (2) Link partitioning (Ahn et al. 2010) (3) Local expansion and optimization (Lancichinetti et al. 2009; Whang et al. 2013), (4) Fuzzy detection (Gregory 2010; Wang et al. 2011; Zhang and Yeung 2012), and (5) Agent-based Raghavan et al. (2007). Among them, Hierarchical Link Clustering (HLC) (Ahn et al. 2010) is a popular approach due to its simplicity. It partitions the links instead of vertices to explore the overlapping community structure. In the local expansion and optimization based algorithms category, Whang et al. (2013) recently proposed an algorithm which finds good seeds and then expands them using a personalized PageRank clustering procedure. Fuzzy detection algorithms measure the strength of association between vertices and communities. They use membership vectors to determine the strength of these associations and determine communities according to these vectors. Apart from this classification, Yang and Leskovec have recently proposed a new approach to enable detection of overlapping communities at large scale (Yang and Leskovec 2013). Their algorithm relies on the observation that

overlapping regions of communities are more densely connected than non-overlapping parts. They create a community model for a given network by the Cluster Affiliation Model, which uses this observation, and then fit this model for a given network. In the fitting phase, they make use of matrix factorization.

Communities in large graphs can have different granularities. For example, given the co-author graph, one possible community would be “people working on databases”. However such a community is very coarse-grained and not so effective in many scenarios. Instead, finding the communities based on vertices provide finer granularity and more focused information. For example, a query like “people working with Prof. X on graph processing” could be more interesting and useful. Recent studies (Gleich and Seshadhri 2012) also show that vertex based approaches provide better communities in terms of widely accepted metrics, such as conductance. Thus, finding fine-grained communities around vertices, which provide grounds for answering queries like “Which communities contain the vertex u?” and “Which communities include the vertices $u_1, \ldots, u_n$?”, is highly valuable. Motivated by this observation, Rees and Gallagher (2010) developed a new class of overlapping community detection algorithms that follow a bottom-up approach, which we refer to as find-and-merge algorithms (Rees and Gallagher 2012, 2013a, b). In this method, the algorithm first finds the local communities of each vertex. It then merges these local communities into global communities based on a configurable merge condition, as part of the merge step. Two years after the Rees and Gallagher (2010) and Coscia et al. (2012) introduced a quite similar method. The SONIC algorithm we describe in this paper adopts a similar approach. Different than Coscia et al. (2012) and Rees and Gallagher (2010), SONIC is an incremental find-and-merge algorithm that can handle streaming updates.

Streaming Several researchers have investigated evolutionary and dynamic community detection algorithms. Evolutionary clustering techniques capture how the clusters change as a function of time (Chakrabarti et al. 2006; Kim and Han 2009; Lin et al. 2008). These techniques focus on temporal evolution at a coarser level and do not address the issues of incremental processing and streaming updates. As an example of incremental community detection algorithm, Xie et al. (2013) proposed an incremental version of label propagation which can be used to find non-overlapping communities in evolving networks. SONIC also uses an incremental label propagation algorithm, but only as an initial step to find the local non-overlapping communities to be merged later into global overlapping communities.

Other dynamic community detection works include Cazabet et al.’s incremental algorithm to detect overlapping communities (Cazabet et al. 2010); Lin et al.’s framework (Lin et al. 2008), which analyses communities and their evolution in dynamic networks; Goldberg et al.’s framework (Goldberg et al. 2011), which investigates the evolution of communities and concludes that lifespan of a community can be correlated with structural parameters of its early evolution; Qi et al.’s online algorithm (Qi et al. 2013) to detect communities in social sensing applications; and Sarr et al.’s work (Sarr et al. 2013) that studies group disappearance in evolving networks. There are also several prior works focusing on streaming dense subgraph detection, which is a problem similar to community detection. Agarwal et al. (2012) and Angel et al. (2012) present algorithms for real-time discovery of events and stories from micro

blog streams. They model the events and stories as dense subgraphs and track their evolution in a streaming fashion.

3 Background

In this section, we provide the necessary background on concepts and algorithms that are relevant to our work.

Ego-minus-ego network Let G be an undirected and unweighted graph. For a vertex u, $N(u)$ is the set of neighbors of vertex u in graph G. The *ego-network* Freeman (1982) of u is the vertex induced sub-graph of G that consists of the vertices $\{u\} \cup N(u)$. Subtracting u and the edges incident upon it from the ego-network results in the *ego-minus-ego network* Rees and Gallagher (2010), formally defined as:

Definition 1 Ego-minus-ego network of vertex u, denoted by $EmEn(u)$, in graph $G=\langle V, E\rangle$ is the subgraph $G'=\langle V', E'\rangle$, where $V'=N(u)$ and $E' \subseteq E$ is the set of edges (v, w) such that $v, w \in V'$. Vertex u is called the *center* of $EmEn(u)$.

Find-and-merge style overlapping community detection This style of community detection algorithms can compute overlapping global communities from local communities Rees and Gallagher (2010). The basic idea is to find local communities in each *EmEn* via a non-overlapping community detection algorithm, such as label propagation, and then add the center of the *EmEn* to each one of the local communities found. After the find step is complete, the algorithm merges the local communities to construct the global ones. The merge is performed based on a *merge factor* parameter, denoted by β, where $\beta \in [0, 1]$. During the merge step, two communities are merged if at least β fraction of the smaller community resides in their intersection. If A and B are two communities, the merge condition is given by $|A \cap B|/\min(|A|, |B|) \geq \beta$. This is the well-known *overlap similarity* metric. The merge process continues until no new merges can be computed.

Incremental local community detection via label propagation Our non-overlapping local community detection algorithm of choice for the find step is label propagation Raghavan et al. (2007). This algorithm works in multiple rounds. Initially, each vertex is assigned a unique id. Then at each round, each vertex is assigned the id of the most commonly used id of its neighbors. Ties are broken randomly. This procedure is performed continuously either for a specified number of rounds or until there is no change in the ids. When an edge (u, v) is inserted/removed into/from the graph, we perform incremental label propagation starting with vertices u and v. That is, we assign their ids to the most commonly used id of their neighbors. If this results in a change in the id values of u or v, then we apply the same procedure to all neighbors of the vertex whose id has changed, and continue the procedure recursively.

Observation 1 After the merge step, each local community takes part in a single global community. Thus, the global communities are a partitioning of the local ones.

Definition 2 Local communities of a vertex $u \in V$, denoted by $LC(u)$, is the set of communities found in $EmEn(u)$ by a non-overlapping community detection algo-

rithm, except that each such community is augmented by the vertex u itself. Global communities, denoted by GC, are the set of communities constructed by merging the local communities of all vertices in the graph. Finally, the set of local communities that were merged to form a global community $g \in GC$ is denoted as $MC(g)$.

4 Observations

In this section we list fundamental observations that we rely on for developing the SONIC algorithm.

Theorem 1 *Given a graph $G=\langle V, E\rangle$, if we insert or remove an edge (u, v), only the EmEns of u, v, and mutual neighbors of u and v change.*

Proof Consider the insertion case. The *EmEn*s of u and v change, as *EmEn*(u) will now contain v, and *EmEn*(v) will contain u. *EmEn*s of common neighbors, that is $N(u) \cap N(v)$, will change too, since for a vertex $w \in N(u) \cap N(v)$, the edge (u, v) will now be contained in *EmEn*(w). For any other vertex $x \notin N(u) \cap N(v)$ that is not u or v, it is easy to see that *EmEn*(x) cannot change. Assume that it does. This change cannot involve a new vertex, say u', being added into *EmEn*(x), as that would require (x, u') to be a new edge, and since $x \neq u$ and $x \neq v$, this is a contradiction. The other possibility is that a new edge is added into *EmEn*(x), which must be (u, v). But then we have $u \in N(x)$ and $v \in N(x)$, which means $x \in N(u) \cap N(v)$, contradicting the initial assumption.

The removal proof follows trivially. Assume that the *EmEn* of some vertex other than u or v or a common neighbor of them has changed. Then inserting the removed edge back will also result in a change for the *EmEn* of that vertex, which contradicts the first part. By the same logic, *EmEn*s of u, v, and vertices in $N(u) \cap N(v)$ will change as a result of the removal.

Corollary 1 *If u and v have no mutual neighbors and if an edge is inserted between them, EmEns of u and v will gain an* unconnected *vertex v and u, respectively.*

Theorem 2 *Given a set of communities, the result of merging them based on an overlap similarity threshold of $\beta < 1$ is sensitive to the merge order.*

Proof Assume that the merge-order is not important and same solution is obtained for any given set and β coefficient. Say that we have three communities $A = \{1, 2, 3\}$, $B = \{3, 4\}$ and $C = \{4, 5, 6\}$ and $\beta = 0.5$. If we merge A and B first, then resulting communities will be $AB = \{1, 2, 3, 4\}$ and $C = \{4, 5, 6\}$. However, if B and C are merged first, then the resulting communities are $A = \{1, 2, 3\}$ and $BC = \{3, 4, 5, 6\}$. Therefore, the merge result is sensitive to the merge order.

Theorem 2 implies that there is non-determinism in the resulting communities when the overlap similarity is not equal to 1.0. In order to escape from this non-determinism, we used $\beta = 1.0$ in our quality measurement experiments. We also note that, even when $\beta < 1.0$, quality of the resulting communities from different incremental runs are very similar, indicating very minor changes in the community structure. Note that,

Rees and Gallagher introduced a different solution to this non-determinism problem by replacing the merge phase with a label propagation scheme (Rees and Gallagher 2012).

Definition 3 For a find-and-merge type of algorithm, any set of communities that is reached via some merge order, such that no further merges are possible between any two communities, is a *valid* solution.

Theorem 3 *Given the set of global communities GC, and the local communities $MC(g), \forall g \in GC$; if a local community $l \in MC(g')$ that is part of a global community $g' \in GC$ changes, then merging the set of local communities within $MC(g')$ plus the remaining global communities $GC \setminus \{g'\}$ will give a valid solution.*

Proof Proof follows from Definition 3. Since any merge order is valid as long as no further merges are possible, we change the merge order of communities to get a valid solution. When we merge the communities in $\bigcup_{g \in GC \setminus \{g'\}} MC(g)$, we will get $GC \setminus \{g'\}$. Then, merging $GC \setminus \{g'\}$ with $MC(g')$ will give us GC. In other words, a from-scratch merge can have such an order that results in first generating the global communities in $GC \setminus \{g'\}$ and then merging in the local communities in $MC(g')$. This is exactly what the incremental merge performs.

Corollary 2 *Given the set of global communities GC, and the local communities $MC(g), \forall g \in GC$; if a set of local communities L that are contained within a set of global communities $\{g_j\} \subset GC$ change, then merging the local communities within $\bigcup_j MC(g_j)$ plus the remaining global communities $GC \setminus \{g_j\}$ will give a valid solution.*

5 The SONIC algorithm

In this section, we introduce the SONIC algorithm for incremental overlapping community detection.

5.1 An overview

SONIC is a find-and-merge style of community detection algorithm with a particular focus on incremental processing, as it aims to support streaming updates. SONIC's algorithmic steps are as follows:

(1) Find the vertices whose local communities are impacted upon an edge insertion or removal.
(2) Perform incremental, non-overlapping local community detection to update the local communities of the impacted vertices.
(3) Detect significant changes and terminate if a change in impacted local communities is found to be insignificant.
(4) Incrementally merge communities and update the global communities.
    (a) Determine a small set of communities to be merged.
    (b) Perform recursive merge of these in an efficient manner.

For local community detection SONIC uses the incremental label propagation algorithm, as described in Sect. 3. The significant change detection capabilities of SONIC are described in Sect. 6.1. In the rest of this section, we focus on the core capability of SONIC: determining the set of communities to be merged, which is significantly smaller in size compared to the entire set of local communities. The efficient procedures for merging these communities are described in Sects. 6.2 and 6.3.

5.2 SONIC core

SONIC handles edge insertions/removals by (i) locating the impacted local communities, (ii) dissolving the global communities that contain them, (iii) replacing the impacted local communities with their updated versions, and finally, (iv) performing a partial re-merge to create the new set of global communities.

In particular, when a new edge is inserted/removed, some local communities, say L, are changed. Assume that these changed local communities are part of some set of global communities, denoted by $C(L) = \{g : \exists l \in L \text{ s.t. } l \in MC(g) \wedge g \in GC\}$. Further assume that the local communities are replaced with their updated versions, say L'. By Corollary 2, SONIC regenerates the new global communities by merging L' and $GC \setminus C(L)$, that is:

$$GC \leftarrow \text{merge}(L', GC \setminus C(L))$$

To facilitate this merge, SONIC keeps track of which local communities were merged to construct each global community. For this purpose, it keeps the following additional data structures:

(1) GC: For each vertex, the global community ids associated with that vertex.
(2) LC: For each vertex, the local community ids associated with that vertex.
(3) For each global community, the local community ids that constitute it.
(4) For each local community, the global community id that it belongs to.

SONIC maintains global communities at each vertex to speedup the merge process. Each vertex u can be part of at most $|N(u)|$ local/global communities, requiring $\mathcal{O}(|E|)$ storage for global community ids. In practice, the number of global communities a vertex u belongs to is considerably smaller than the upper bound of $|N(u)|$. The total number of local communities, $|LC|$, can be at most $2 \cdot |E|$. Thus, keeping the global communities with their constituent local communities also takes $\mathcal{O}(E)$ space. Again, this happens only for the highly unlikely scenario of each edge belonging to a different local community. Given that $\mathcal{O}(E)$ with a low constant is a workable bound in practice, we implemented our auxiliary data structures in a way that enables us to do faster lookups by using more memory. Our evaluation shows that the space overhead of the data structures kept by SONIC corresponds to a small constant times the number of edges, which makes our algorithms applicable to the graphs with million of edges (see Sect. 7). Note that we only store auxiliary information along with the connection related information, i.e., we do not replicate the entire attributed graph.

Given a graph G and global communities GC, if an edge (u, v) is inserted/removed, SONIC updates the global communities using Algorithm 1. First, it performs the

Algorithm 1: SONIC $(G, (u, v), op, \beta, LC, GC)$

Input: G: graph, (u, v): updated edge, op: operation ('i'/'r') LC: local communities, GC: global communities, β: merge factor

1 **if** $op =$ 'i' **then** $G \leftarrow G \cup \{(u, v)\}$ ▷ Insertion operation
2 **else** $G \leftarrow G \setminus \{(u, v)\}$ ▷ Removal operation
3 **if** $N(u) \cap N(v) = \emptyset$ **then** ▷ No common neighbors
4   **return** ▷ No update needed
5 $S \leftarrow \{u, v\} \cup (N(u) \cap N(v))$ ▷ Vertices with changed *EmEn*
6 $R \leftarrow \{u \mapsto LC(u) : u \in S\}$ ▷ Local comm. of vertices in S
7 INCRLABELPROPAGATION(S, LC) ▷ Update local comm.
▷ Find the removed local communities
8 $L^- \leftarrow \{l : l \notin LC(u) \wedge l \in R[u] \wedge u \in S\}$
▷ Find the added local communities
9 $L^+ \leftarrow \{l : l \in LC(u) \wedge l \notin R[u] \wedge u \in S\}$
▷ Find the set of global communities to be dissolved
10 $C \leftarrow \{g : \exists l \in L^- \text{ s.t. } l \in MC(g) \wedge g \in GC\}$
▷ Dissolve comm. in C, remove non-existing local comms.
11 $\mathcal{F} \leftarrow \{MC(g) \setminus L^- : g \in C\}$
12 $GC \leftarrow GC \setminus C$ ▷ Remove dissolved global comms.
13 MERGE($G, \beta, GC, \mathcal{F}, L^+$) ▷ Re-merge into global comms.

insertion/removal. Then it checks the mutual neighbors of u and v. If there are no mutual neighbors, we know that only u's and v's *EmEn*s change, and v and u are added as unconnected vertices to the *EmEn*s of u and v, respectively (see Corollary 1). In this case, it assumes that there is no change in the local community structure of u and v, and therefore it performs no further operation to update GC (line 4).

When u and v have mutual neighbors, the *EmEn*s of u, v, and their mutual neighbors change (see Theorem 1). Thus, SONIC collects these vertices in a set S (line 5). Next, it creates a map R to keep the set of local communities associated with the vertices whose *EmEn*s has changed (line 6). This map temporarily keeps the old local communities so that they can be compared to the new ones formed after the insertion/removal. The next step computes the new local communities by running the incremental label propagation algorithm (see Sect. 3).

After updating the local communities, SONIC finds the set of old local communities that no longer exist, denoted by L^- (line 8); as well as the set of newly created local communities, denoted by L^+ (line 9). Using L^-, it finds the set of global communities to be dissolved, denoted by C (line 10). These are the global communities that currently contain nonexistent local communities. SONIC then dissolves each such global community g by converting it into a set of local communities $MC(g)$ and removing the nonexistent local communities, that is $MC(g) \setminus L^-$. The end result is a *set of local community sets* (denoted by $\mathcal{F}$), where each one of the local community sets represents a dissolved global community (line 11).

Finally, SONIC merges the set of dissolved global communities $\mathcal{F}$ and the new set of local communities L^+ together with the global communities that are kept intact, that is $GC \setminus C$, using the merge factor β. In the base version of SONIC, we apply the merge operation given in Algorithm 2, named INCNAIVE.

Algorithm 2: INCNAIVE MERGE($G, \beta, GC, \mathcal{F}, L^+$)

Data: G: graph, β: merge factor, GC: global communities, $\mathcal{F}$: local community sets of previously dissolved global communities, L^+: newly created local communities

▷ Perform the Distribute New Phase
1. **for each** $l \in L^+$ **do** ▷ For each new local comm.
   - ▷ Find a suitable dissolved comm. for l
2. Find an F s.t. $\exists f \in F, f \cap l \neq \emptyset$
3. $F \leftarrow F \cup \{l\}$ ▷ Add local comm. to a dissolved comm.

▷ Perform the Regroup Dissolved Phase
4. $L \leftarrow \emptyset$ ▷ Initialize the global merge list
5. **for each** $F \in \mathcal{F}$ **do** ▷ For each dissolved global comm.
6. **for each** $f_i \in F$ **do**
7. **for each** $f_j \in F$, where $j > i$ **do**
8. **if** OVERLAP(f_i, f_j)$\geq \beta$ **then** ▷ There is a merge
9. $f_i \leftarrow f_i \cup f_j$ ▷ Merge latter into former
10. $F \leftarrow F \setminus f_j$ ▷ Get rid of the latter
11. $L \leftarrow L \cup F$ ▷ Add merged comms. to global merge list

▷ Perform the Global Merge Phase
12. **for each** $c_i \in L$ **do**
13. **for each** $c_j \in L$, where $j > i$ **do**
14. **if** OVERLAP(c_i, c_j)$\geq \beta$ **then** ▷ Meets merge criteria
15. $c_i \leftarrow c_i \cup c_j$ ▷ Merge latter into former
16. $L \leftarrow L \setminus c_j$ ▷ Get rid of the latter
17. **for each** $g_k \in GC$ **do**
18. **if** OVERLAP(c_i, g_k)$\geq \beta$ **then** ▷ Meets merge criteria
19. $c_i \leftarrow c_i \cup g_k$ ▷ Merge latter into former
20. $GC \leftarrow GC \setminus g_k$ ▷ Get rid of the latter
21. $GC \leftarrow L \cup GC$ ▷ Update the global comms.

The INCNAIVE algorithm consists of three phases. The first one is called the *Distribute New Phase* (lines 1–3). In this phase, the algorithm distributes the newly created local communities, $l \in L^+$, over the dissolved communities, $\mathcal{F}$. To do that, it selects a set $F \in \mathcal{F}$ for each l such that there is a community $f \in F$ that overlaps with l, i.e., $f \cap l \neq \emptyset$. The goal here is to assign each new local community to one of the dissolved communities where it is likely to merge with an existing local community. This heuristic results in a higher number of cheaper merges in the second phase, yet a lower number of more expensive merges in the third phase. Thus, it is effective in reducing the overall merge cost.

The second phase is called the *Regroup Dissolved Phase* (lines 4–11). In this phase, the algorithm performs a merge within each dissolved community $F \in \mathcal{F}$, separately. The motivation is that it is highly likely for the local communities that made up a dissolved global community to re-merge. By doing smaller-scale merges that are localized to the dissolved communities, we aim at reducing the overall cost of the merge. During the merge, the algorithm checks each pair of communities, f_i and f_j, where $j > i$, to see if their overlap similarity is above the merge threshold β. If so, it merges f_j into f_i and removes f_j from its belonging set F. After it completes the

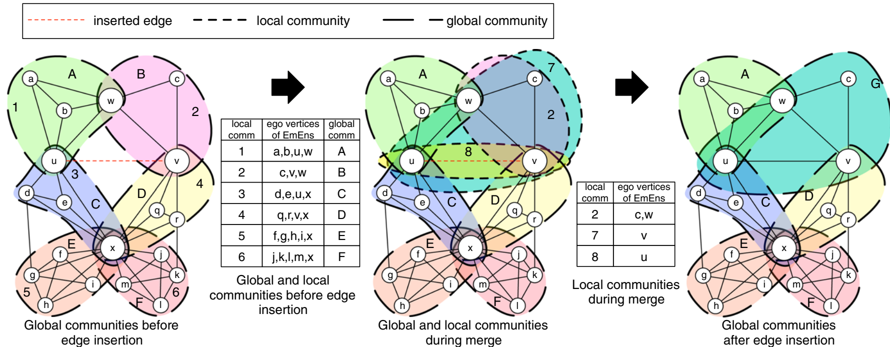

Fig. 1 Illustration of the community changes upon an edge insertion. After inserting an edge between u and v, global community B evolves into a bigger global community G

merge for a dissolved community, it adds the resulting merged communities to a *global merge list*, denoted as L (line 11).

In the last phase, called the *Global Merge Phase* (lines 12–20), the algorithm performs a merge between communities in the global merge list, that is L, and the intact global communities, that is GC. Here we check whether any community within the global merge list can be merged with each other (c_i and c_j, $i < j$), as well as with any of the intact global communities (GC). This is why the outer loop's body contains loops that iterate over both L and GC. However, the outer loop only iterates over L, because once all the communities in L are merged with the rest, the only possible comparisons that remain are between the intact global communities, and we are guaranteed that those cannot merge (as they are intact, and thus known to have overlap similarity of less than β).

Thanks to the incremental merge procedure, our SONIC algorithm is expected to be much faster than the existing non-incremental find-and-merge type of algorithms. SONIC aims to merge as few and as small communities as possible. We stated in Definition 3 that any order of merge operations is a valid solution as long as no further merges are possible. In this work, we do not elaborate on the impact of different merge orders on quality, instead we try to make the merge phase as fast as possible while not sacrificing the quality. Our experimental results (see Sect. 7) show that the quality is on par with the non-incremental algorithm (Coscia et al. 2012).

5.3 Illustrative example

Figure 1 illustrates the community changes when we insert an edge into a sample graph. The example assumes that the merge factor (β) is 1.0, i.e., two communities can merge only if one of them is a subset of the other. The leftmost figure and table show the global and local communities before the edge insertion is performed. Global communities are shown with bold dashed lines and capital letter ids. For example, community A is a global community consisting of vertices a, b, u, and w. Local communities within the ego-minus-ego networks (*EmEn*s) of ego vertices are given

in the tables. For example, in the leftmost table, community 2 is a local community belonging to the *EmEn* of c, as well as v and w. The local communities that form a global community can also be read from the table by finding all rows that contain the target global community. The local communities corresponding to these rows form the global community. For example, in the leftmost table, the global community A is composed of the local communities $(a, 1)$, $(b, 1)$, $(u, 1)$, and $(w, 1)$. Note that, finding the local communities belonging to a global community does not require a full scan of the table in the implementation. We make use of our data structures to access local communities of a global community in constant time.

When we insert the edge (u, v), we first check the vertices whose *EmEn*s are changed. Based on Algorithm 1, those vertices are u, v and their mutual neighbors w and x (all shown using larger circles in the figure). For each of these vertices, we check if there is a change in their local community structure by performing incremental label propagation in their *EmEn*s (line 7 of Algorithm 1). For vertex w, it turns out that there is no change in the local community structure, because the inserted edge is not strong enough to bind local communities 1 and 2. Similarly, there is no change in the local community structure of vertex x, as the inserted edge cannot cause a connection between the local communities 3 and 4. On the other hand, the local community structures of u and v change: u gets a new local community 8, and v's local community 2 changes into local community 7. After the removed and newly created local communities are detected (lines 8 and 9 in Algorithm 1), local community 2 of vertex v is the only removed local community in the new local community structure, which is shown as $L^- = \{(v, 2)\}$; whereas local communities 7 and 8 are the newly formed communities of vertices v and u, respectively, shown as $L^+ = \{(v, 7), (u, 8)\}$. Next, we find the global communities to be dissolved and it turns out that global community B is the only one that must be dissolved, i.e., $C = \{B\}$ (local community 2 belongs to global community B). Then, we find the set of local communities to be merged ($\mathcal{F}$) by dissolving B and subtracting local community 2 of v, i.e., $\mathcal{F} = \{\{(c, 2), (w, 2)\}\}$.

Finally, we set the global communities $GC = \{A, C, D, E, F\}$ and merge global communities (GC), dissolved local communities ($\mathcal{F}$), and newly created local communities (L^+). During the first phase of Algorithm 2, we distribute both newly created communities, L^+, to the only set $F \in \mathcal{F}$ and obtain the resulting community list to be merged as $F = \{(c, 2), (w, 2), (v, 7), (u, 8)\}$. In the second phase, we merge those four communities, shown with thin dashed lines in the middle figure, and obtain the global community G in the rightmost figure. In the third phase, we attempt merging the global community G with the communities in global community set GC, but no merge happens. The rightmost figure shows the final global communities. Overall, the global community B in the leftmost figure evolved to form the global community G in the rightmost figure.

6 SONIC improvements

In this section, we describe several improvements over the SONIC core.

6.1 Significant change detection

Insertion and removal of edges cause changes in the local community structure. However, the base version of SONIC does not quantify the *significance* of these changes. In particular, any change that connects vertices that already have a common neighbor leads to dissolving some global communities and performing the merge step. As an improvement, we introduce the notion of *significant change* with respect to the local community structure of vertices. When there is a change in the local community structure of the *EmEn* of a vertex, we compare the existing community structure to the new community structure using the Normalized Mutual Information (NMI) index (Danon et al. 2005), which enables the comparison of two overlapping sets (McDaid et al. 2011). For two groups of local communities, L_1 and L_2, their NMI is defined as:

$$\text{NMI}(L_1, L_2) = \frac{I(L_1; L_2)}{(H(L_1) + H(L_2))/2},$$

where $H(L_1)$ and $H(L_2)$ are the entropies of L_1 and L_2, respectively; and $I(L_1; L_2)$ is the mutual information of L_1 and L_2, defined as:

$$I(L_1; L_2) = H(L_1, L_2) - H(L_1|L_2) - H(L_2|L_1),$$

where $H(L_1, L_2)$ is the joint entropy of L_1 and L_2, and $H(L_1|L_2)$ is the entropy of L_1 conditional on L_2. The NMI values lie within the range [0, 1], where higher values indicate higher similarity.

We compute the NMI score for each vertex whose ego-minus-ego networks (*EmEn*s) are impacted (line 5 in Algorithm 1) by comparing the set of local communities in its *EmEn* before the insertion/removal (line 6 in Algorithm 1), with the ones after incremental label propagation. If this similarity is above a specified threshold, then we assume that there is no significant change and do not update removed (L^-) and newly created (L^+) local communities. Otherwise, we update them and also set the $R[u]$ to $LC(u)$, i.e., update the local community structure. Continued changes in the graph structure are accumulated if no significant change is observed in the community structure. We denote the threshold by α, and name it as the *significant change threshold*. If $\alpha = 1$, then every local change is accepted as significant, whereas $\alpha = 0$ means that any local change is regarded as insignificant. As such, we take $\alpha \in (0, 1]$.

Using the α parameter provides the ability to adjust the trade-off between update cost and the community detection accuracy. This is very useful, especially for scenarios where the update rate of the graph is high relative to the query rate. Setting a lower α means that we do not keep the communities perfectly up to date after each update. If the query rate is low, it is acceptable to have a lower α, as the staleness in the responses will be relatively low compared to the query period. If the query rate is high and the application can tolerate responses with less up-to-date data, it may still be acceptable to have a lower α. This way, less computing resources are spent on edge updates and more resources are available to respond to queries.

6.2 Minhash-based merge

In Sect. 5.2, we introduced the INCNAIVE algorithm for performing the re-merge of the communities. However, this algorithm is expected to get costly when the size of the merged communities increase. This is because the cost of computing the overlap similarity is linear in the size of the smaller set, since we keep the communities as hash sets. As the communities get larger towards the end of the merge process, this cost significantly increases. In this section, we propose an adaptation of the *min-hashing* technique to alleviate this problem.

Min-hashing is a technique for quickly estimating the Jaccard similarity between two sets (Broder et al. 1998). Let A and B be two sets, then the Jaccard similarity between them is given by $|A \cap B|/|A \cup B|$. Min-hashing uses n random hash functions to map the elements of the two sets to values, and for each one of the hash functions, it finds the smallest hash values for the two sets. If the smallest hash value for A and B agree for m number of the hash functions, then the Jaccard similarity is estimated as m/n. The probability of the minimum hash values of A and B being the same is equal to the probability of the item having the minimum hash value being in the intersection of the two sets. It is easy to see that the latter is equal to the Jaccard similarity, as there are $|A \cap B|$ items in the intersection and there are $|A \cup B|$ items in total.

The speed advantage of min-hashing compared to the explicit computation is that, min-hashing based similarity can be computed in $\mathcal{O}(n)$ time, where n is the number of hash functions used. This number is expected to be smaller than the size of the sets. Importantly, we assume that the min-hashes are computed once for all the sets and many comparisons are made over these sets to compute pairwise Jaccard similarities. The min-hashing based computation of the similarity will lose its accuracy if the number of hash functions is small. As a result, there is a trade-off between performance and accuracy that can be adjusted by setting n properly.

One important issue in using min-hashing for our merge problem is that min-hashing is based on Jaccard similarity, whereas we use overlap similarity for our merge process. To convert a given overlap similarity coefficient (β) to the corresponding Jaccard similarity coefficient (θ), we apply the following formula:

$$\theta = \beta \cdot ((1 - \beta) + |B|/|A|)^{-1},$$

where $|A| \leq |B|$. This is obtained as follows. We have $\beta = |A \cap B|/|A|$ from the definition of overlap similarity. Thus, $|A \cap B| = \beta \cdot |A|$. Since $|A \cup B| = |A| + |B| - |A \cap B|$, by plugging in $\beta \cdot |A|$ in place of $|A \cap B|$, we get $|A \cup B| = (1 - \beta) \cdot |A| + |B|$. From the definition of Jaccard similarity we have $\theta = |A \cap B|/|A \cup B|$ and plugging in our derivations of $|A \cap B| = \beta \cdot |A|$ and $|A \cup B| = (1 - \beta) \cdot |A| + |B|$, we get $\theta = \beta/((1 - \beta) + |B|/|A|)$.

In our min-hash based merge, we initally compute n min-hash values, one for each local community. Later, when we merge two communities, we only need $\mathcal{O}(n)$ operations to compute the new min-hash values for the merged community, as we only need to take the smaller of the min-hash value pairs for each hash function. As a result,

we perform hashing only once over the base communities and re-use the results many times during the merge. We evaluate the accuracy vs. performance trade-off involved in setting the number of hash functions as part of our experimental evaluation in Sect. 7.

6.3 Inverted index based merge

In Sect. 6.2, we proposed the use of min-hashing as a cheaper alternative to the explicit overlap similarity computations performed within the incNaive algorithm during the re-merge of the communities. While min-hashing reduces the cost of similarity computations, especially for large communities, the number of such computations made for comparing communities for possible merges is still high. One idea that comes to mind to alleviate this problem is *locality-sensitive hashing* (*LSH*) (Indyk and Motwani 1998). The motivation behind using locality-sensitive hashing is to hash similar items to the same buckets, so that the pair-wise similarity comparisons can be limited to the confines of individual buckets. For our re-merge problem, this would significantly reduce the number of similarity computations made and thus the overall merge time. However, it has been shown that locality sensitive hash functions do not exist for the overlap similarity metric (Charikar 2002). As a result, the LSH technique cannot be adapted for our problem.

In this section, we propose to leverage our support data structures, particularly the global community ids for each vertex, to reduce the number of comparisons made during the re-merge. This alternative merge algorithm, called incInvIndex, is based on inverted indices of global communities. The algorithm attempts to make small number of comparisons by traversing the vertices of communities to be merged and computing their intersections with the surrounding communities via a simple counting procedure, utilizing fast lookups and updates on a map structure.

Algorithm 3 gives the pseudocode of incInvIndex. The algorithm consists of two phases. The first one is called the *Pre-formation Phase* (lines 1–3). In this phase, we collect both the local communities within the dissolved global communities ($F \in \mathcal{F}$) and the newly created local communities (L^+) into L, called the *global merge list* (line 1). We then update the data structure that keeps the list of global communities each vertex belongs to (line 3). This data structure serves as the inverted index. For each vertex u in a local community l within the global merge list L, we remove the dissolved communities (GC^C in the pseudocode, denoting anything other than the intact communities) from its list of global communities $GC(u)$, and add the local community l as a global community to $GC(u)$. After the first step is complete, we are ready to merge L with the intact global communities in GC. Note that we perform the update of the inverted index $GC(u)$ as part of the first phase, so that we can do efficient merges in the second phase.

The second phase is called the *Global Merge Phase* (lines 4–20). In this phase, we merge L and GC. For each community $l \in L$, we check to see whether it can merge with any other community in L or GC. But rather than doing this by iterating over L and GC, we do it by using the inverted index. In particular, for each community $l \in L$, we iterate over its vertices. For each vertex $u \in l$, we go over the global communities

Algorithm 3: INCINVINDEX MERGE($G, \beta, GC, \mathcal{F}, L^+$)

Data: G: graph, β: merge factor, GC: global communities, $\mathcal{F}$: local community sets of previously dissolved global communities, L^+: newly created local communities

```
▷ Perform the Pre-Formation Phase
1  L ← ∪_{F∈𝓕} F ∪ L^+                          ▷ Put local comms. into the merge list
2  for l ∈ L do                                  ▷ For each comm. in the global merge list
       ▷ Update the global community mappings
3      GC(v) ← GC(v) \ GC^C ∪ {l}, ∀_{v∈l}
▷ Perform the Global Merge Phase
4  for each l ∈ L do                             ▷ For each comm. in the merge list
5      M ← {}_0                                  ▷ Initialize the intersection counter map
6      c ← false                                 ▷ Initialize the change flag (no changes)
7      for each u ∈ l do                         ▷ For each vertex in the comm.
8          for each g ∈ GC(u) do                 ▷ For global comms. of u
9              M[g] ← M[g] + 1                   ▷ Incr. # of intersections
               ▷ Already meets merge criteria
10             if (M[g]/min(|l|, |g|)) ≥ β then
11                 c ← true                      ▷ Mark the change
12                 l ← l ∪ g                     ▷ Merge the communities
13                 if g ∈ GC then                ▷ g is from intact comms.
14                     GC ← GC \ g               ▷ Remove g from GC
15                 else                          ▷ g is from the global merge list
16                     L ← L \ g                 ▷ Remove g from L
                   ▷ Update the global community mappings
17                 GC(v) ← GC(v) \ {g} ∪ {l}, ∀_{v∈l}
18                 break                         ▷ Break out of the loop
19         if c is true then break               ▷ Re-merge the new l
20 GC ← GC ∪ L                                   ▷ Update the global comms.
```

that contain it. These communities are listed in the inverted index, as $GC(u)$. For each such community $g \in GC(u)$, we increment a counter stored in a map data structure, denoted by M (line 9). $M[g]$ represents the current count of common vertices between l and g. The moment this value is high enough to satisfy the overlap similarity condition (line 10), we can merge l and g. We do this by merging g into l. We then remove g from either GC or L, depending on which one it came from. Finally, we update the inverted index by removing g from the list of global communities $GC(v)$ of each vertex $v \in l$, and adding l into the same (see line 17). Once a merge happens, we need to break and go back to the start of processing l for new merges (line 19). For this purpose, a boolean variable c is kept to break out of the inner two for loops at once.

As in the INCNAIVE algorithm, the outer loop only goes over the global merge list L. As before, we know that once no more merges can happen between L and any other community in L or GC, the intact communities in GC cannot get involved in any merges, since they cannot merge among themselves. Accordingly, we return the final set of global communities as $L \cup GC$ (line 20).

7 Experimental evaluation

This section presents the evaluation of our algorithms using various datasets under different scenarios. The first set of experiments focus on comparing the proposed algorithms to the previous work with respect to the quality of the identified communities. The second set evaluates the running time performance of our algorithms when processing real-world datasets of different types and sizes. The third set compares the running time performance of the different merge algorithms introduced in Sects. 5.2, 6.2 and 6.3. The fourth set investigates the impact of the two algorithmic parameters, namely the significant change threshold (α) and the merge factor (β), on the algorithm's running time performance and community detection quality. The last set of experiments compare the running time performance of our algorithms when processing synthetic graphs of different sizes.

Setup Algorithms were implemented in C++ and compiled with `gcc 4.8.1` at -O3 optimization level. Experiments were executed sequentially on a Linux operating system running on a machine with an Intel Xeon E5520 2.27 GHz CPU and 48 GBs of RAM.

Datasets We obtained real-world datasets from SNAP (2014). They are the co-purchasing network (amazon0601 (AM)), friendship network (facebook (FB)), follower–followee network (twitter (TW)) and email communication network (email-Enron (EE)). We also extracted the co-authorship network of DBLP (2014) papers. Table 1 shows the properties of these datasets. For each graph, we give its size, the time taken to run the non-incremental find-and-merge algorithm and the memory space overhead (in terms of number edges) of the support data structures used by our algorithms. Note that, runtimes of all other competitor algorithms are significantly larger than the non-incremental time shown in Table 1, thus not stated here. For example, HLC method takes prohibitively long time for graphs with only 10 K nodes, as also stated in Yang and Leskovec (2013). We also use synthetic graphs in our experiments, in order to better evaluate the impact of changing graph size. These graphs, generated using SNAP's R-MAT generator (SNAP 2014), follow a power law vertex degree distribution and exhibit small world properties. To achieve that, we set the partition probabilities of the generator to [0.40; 0.25; 0.20; 0.15]. We set the average degree of the R-MAT graphs to 4.

Table 1 Real-world graph datasets and their properties

Graph dataset	Number of vertices	Number of edges	Average degree	Non-incremental time	Memory overhead
Amazon0601 (AM)	403,394	3,387,388	16.79	16 h	$4.25 \cdot \lvert E\rvert$
Facebook (FB)	4,039	88,234	43.69	5.530 s	$4.17 \cdot \lvert E\rvert$
Email-Enron (EE)	36,692	367,662	20.04	3.42 m	$3.59 \cdot \lvert E\rvert$
Twitter (TW)	81,306	2,684,324	66.03	21.53 m	$3.68 \cdot \lvert E\rvert$
DBLP_coauthor (DB)	1,236,220	15,897,220	25.72	76 h	$4.79 \cdot \lvert E\rvert$

7.1 Quality

In this section we evaluate the quality of the core SONIC algorithm and the improvements introduced in Sect. 6, using real-world datasets. We use four previously published state-of-the-art community detection algorithms for comparison: Hierarchical Link Clustering (HLC) (Ahn et al. 2010), Infomap (Rosvall and Bergstrom 2008), Modularity (Newman 2006), and DEMON (Coscia et al. 2012). HLC has been shown to outperform other existing overlapping community detection algorithms. Infomap is a non-overlapping algorithm that aims to minimize the random walk entropy. Modularity is an eigenvector-based non-overlapping community detection method, which maximizes the modularity metric. DEMON serves as our baseline, since it is the non-incremental (static) version of SONIC. It is worth noting that our goal in this comparison is twofold: showing that (i) SONIC performs similar to DEMON, (ii) SONIC is a competitive community detection algorithm in terms of quality.

For the SONIC algorithms, we construct the communities by first bootstrapping the 90 % of the graph and then inserting the remaining 10 % of the edges one by one applying SONIC at each step. This way, we capture the impact of SONIC on the quality of the communities by realistic changes in the graph which preserve to topology structure. The significant change threshold α and the merge factor β are both set to 1.0 to provide up-to-date and deterministic results. Remember that, if β value is less than 1.0, output is non-deterministic (Theorem 2) and we selected that value to make our experiments more robust. By setting α value to 1.0, we aim to consider each change in the local community structure as significant. In all figures, SONIC NV represents core SONIC using INCNAIVE merge algorithm (Sect. 5.2), SONIC II represents SONIC using the INCINVINDEX merge algorithm (Sect. 6.3) and SONIC MHx is SONIC using the minhash-based merge algorithm (Sect. 6.2) with x number of hash functions.

In the first experiment, we quantify the quality of the communities found using two metrics: *conductance* and *cohesiveness*. The conductance metric measures how connected a community is to the rest of the graph. It measures the fraction of total edge volume pointing outside of the community. If we denote the number of edges crossing the boundaries of the community as $|E_o|$ and the total number of edges of the community as $|E|$, then conductance is given by $c_d = |E_o|/|E|$. Lower conductance values imply better communities. In networks that contain overlapping communities, communities are not disjoint and thus conductance is expected to be relatively high compared to those that contain non-overlapping communities. The cohesiveness metric quantifies how connected the members of a community are to each other. That is, it measures the density of a community. If we denote the number of edges within the boundaries of the community as $|E_i|$ and the total number of vertices in the community as $|V|$, then cohesiveness is given by $c_h = |E_i|/(|V| \cdot (|V| - 1|)/2)$. Higher values imply better communities. We also define a *quality index* by combining conductance and cohesiveness by taking their geometric mean, that is $q = \sqrt{(1 - c_d) \cdot c_h}$. The quality index metric provides a bigger picture to show the impact of both conductance and cohesiveness. Algorithms balancing the two metrics are expected to give higher scores for *quality index*. Since the number of communities reported by each competitor algorithm is significantly different, we focus on top 1000 resulting communities with

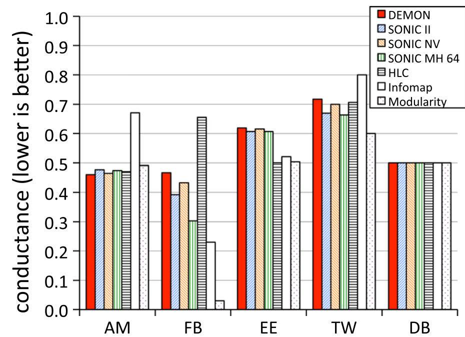

Fig. 2 Conductance on real-world graphs. Modularity is the best, as it is an optimization algorithm for conductance

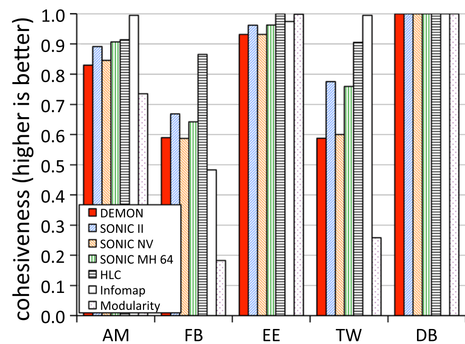

Fig. 3 Cohesiveness on real-world graphs. Results depend on the graphs

the best quality index score and report the geometric means. Similar methodology is used by Yang and Leskovec in their work (Yang and Leskovec 2012).

Figure 2 shows the results for the conductance metric. Modularity is the best performing algorithm for most graphs since it optimizes the modularity metric to find non-overlapping communities. The modularity metric measures the fraction of edges that fall within the given communities minus the expected fraction of such edges if all edges were distributed at random. Thus, it is closely related with the conductance metric. As a SONIC variant with minhash-based merge, we selected SONIC MH 64 as representative. We observe that DEMON, SONIC NV, SONIC II, and SONIC MH 64 perform very similar to each other. For DBLP_coauthor graph, all algorithms report the same 1000 communities with the best quality indexes. Figure 3 shows the cohesiveness results. HLC and Infomap perform best for this metric in all graphs and again DEMON, SONIC NV, SONIC II and SONIC MH 64 perform similar to each other.

Figure 4 shows the quality indexes for all graphs. SONIC variants and DEMON show similar results and are close to the best results on amazon0601 and facebook graphs. Overall, SONIC and DEMON provide a *good balance* between cohesiveness and conductance, which cannot be said for any of the other algorithms.

We also investigate how the number of hash functions affect the quality of the communities found by SONIC when using the minhash-based merge. For this purpose, we obtained the communities with different number of hash functions (from 1 to 64) and measured the similarity of the results to those obtained by running the DEMON algorithm. We applied a version of Normalized Mutual Index (NMI) that is adopted for

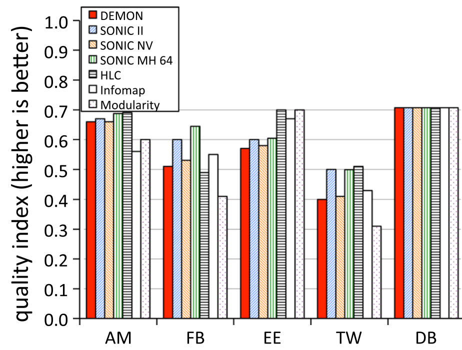

Fig. 4 Quality index scores on real-world graphs. DEMON and SONIC variants show competitive behavior

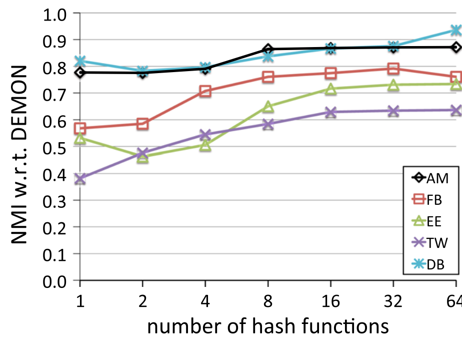

Fig. 5 NMI scores of SONIC MH wrt. DEMON with varying # of hash functions on real-world graphs

overlapping communities (Lancichinetti et al. 2009) as a scoring function to determine the similarity between two sets of communities computed by SONIC and DEMON. If the two sets of community are identical, then their NMI score is a perfect 1. Figure 5 shows the NMI scores of SONIC with respect to DEMON algorithm. As expected, increasing the number of hash functions provides results that are more similar to DEMON, since the NMI increases. The same trend is observed for all graphs. For most of the graphs, the NMI stabilizes after 16 hash functions.

Last, we checked the difference between SONIC II and SONIC NV by comparing their NMI scores. For amazon0601 and facebook graphs, the similarity between SONIC II and SONIC NV is higher than the similarity between SONIC MH 64 and SONIC NV. For amazon0601 network, comparing SONIC II results in 0.89 NMI score similarity with SONIC NV, which is greater than all SONIC MH variants. In general, SONIC II is significantly similar to SONIC NV, resulting in 0.70 NMI score similarity on average.

7.2 Running time performance

In this section, we evaluate the running time performance of SONIC II, which is expected to be the fastest algorithm. We use real-world graphs for the evaluation, which are originally static. We emulate a streaming scenario by treating the whole set of vertices and edges as a *sliding window* snapshot. To evaluate streaming execution,

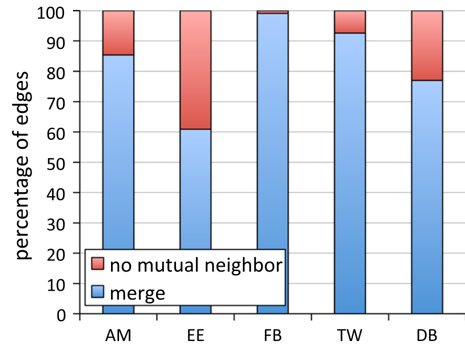

Fig. 6 Most edge removal/insertions result in a merge. Yet, for some graphs, a sizable fraction of updates skip the merge

we first evict a random edge from the current graph. This emulates the behavior of a full sliding window and opens up space for inserting a new edge. We then insert a new edge to form a full window again. As we remove and insert edges, we preserve the graph's structure. This is achieved by initially putting aside a small sample of graph edges and not including them in the first snapshot. Later they are used for insertions, and as a holding place for the removed edges. Other scenarios, like dynamic interaction graphs or specific event based edge removal (fixed time to live, a decay factor, etc.), are similar to our setup for the purpose of measuring runtime, and will give the same results. Thus, they are not investigated further in this work.

We measure the average execution time for removing and inserting one edge to each dataset. We set the significant change threshold α and the merge factor β to 1.0. For relative throughput results, we compute the relative throughput of each single edge update with respect to non-incremental (static) community construction and report the geometric mean of throughputs over multiple updates. Relative throughput reflects how well our algorithm performs. Since there are no alternative incremental algorithms for find-and-merge style of community detection, we compare our incremental methods with the existing non-incremental algorithms and report the relative throughput.

Figure 6 shows the relative frequency of the two code paths executed by SONIC (Algorithm 1) when inserting/removing the 1000 randomly picked edges, for each dataset. Recall that when the vertices connected by the edge have *no mutual neighbors*, then the algorithm terminates early. If they share neighbors, then the algorithm executes the merge step. We observe that, in general, most updates result in a merge. However, depending on the structure of the graph, a non-significant number of edges may take the early termination path. For the facebook and twitter graphs, more than 93 % of the edges result in a merge operation, whereas this number is 85, 60 and 77 % on amazon0601, email-Enron and DBLP_coauthor graphs, respectively. The reason is that facebook and twitter graphs have higher average degree and thus are denser than other graphs. When the graph is denser, the probability of having a mutual neighbor for the incident vertices of a randomly selected edge is higher.

Figure 7 shows the average running times of a single edge insertion and a single edge removal. For all graphs, we manage to stay below 1 s per edge insertion or removal. Furthermore, SONIC II is able to keep the insertion/removal time below 0.01 s for our biggest two graphs, amazon0601 and DBLP_coauthor.

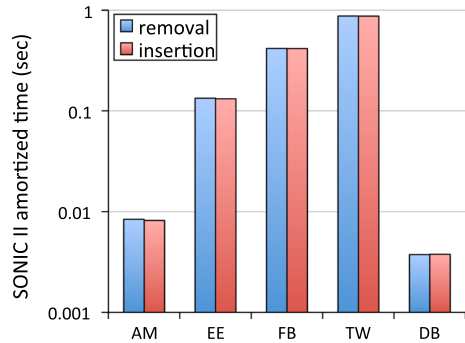

Fig. 7 Average runtimes of one edge removal and insertion on real-world graphs when 1000 edges are removed and inserted

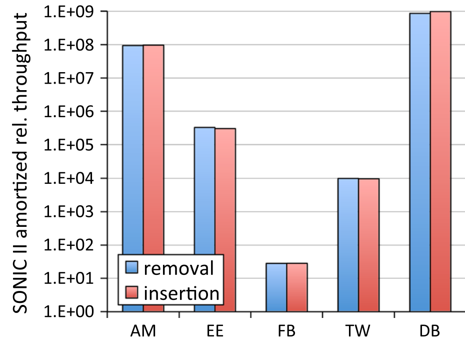

Fig. 8 Average relative throughput of one edge insertion/removal w.r.t. static algorithm when 1000 edges are removed/inserted

Figure 8 shows the relative throughput of a single edge insertion and a single edge removal relative to the non-incremental find-and-merge algorithms. We compute the relative throughput by dividing the from-scratch computation time to the one edge insertion/removal time. We compute the geometric mean of all relative throughputs. The resulting number increases as the graph size gets larger, since it takes more time to re-compute communities from scratch. For the amazon0601 and DBLP_coauthor graphs, 8 and 9 orders of magnitude relative throughputs are reached, respectively.

Experiments on real temporal data Apart from the sliding window scenario, we also investigated the performance of the SONIC II algorithm using real temporal data from the DBLP_coauthor graph, which has an explicit ordering on the stream of edges based on timestamps. In particular, we inserted the co-authorship edges for the papers published after Jan 1, 2013 and measured the resulting execution time and relative throughputs. Average execution time is 0.018 s per edge insertion and average relative throughput observed is 405 M, which is 8 orders of magnitude better than from-scratch computation.

7.3 Comparison of merge variants

In this section, we compare the runtime performance of different merge algorithms, namely SONIC NV, SONIC II, and the SONIC MH variants. We use the same experiment setup as in Sect. 7.2.

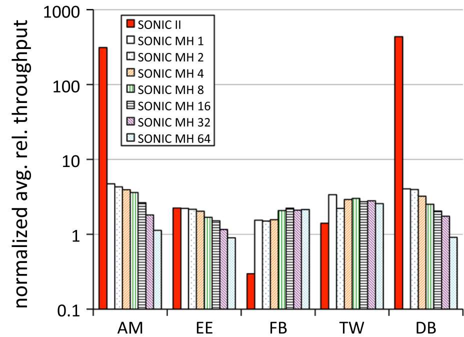

Fig. 9 Normalized insertion/removal relative throughputs of SONIC variants w.r.t. SONIC NV. SONIC II performs best for large networks

Figure 9 shows the average of normalized insertion and removal relative throughputs of SONIC II and SONIC MH variants with respect to SONIC NV. The best performing merge algorithm depends on the dataset. For the amazon0601 and DBLP_coauthor graphs, SONIC II performs the best with a significant difference, as it is 312 and 435 times faster than SONIC NV, respectively. In general, SONIC MH variants perform better as the number of hash functions decrease. SONIC MH1 is 2.95 times faster than SONIC NV whereas SONIC MH64 is 1.38 times faster. Considering the large sizes of the amazon0601 and DBLP_coauthor graphs, the low runtime of SONIC II can be explained by the efficient merge operations. SONIC II, as explained in Sect. 6.3, tries to merge only the spatially close communities (that have common vertices), therefore provides an efficient merge operation. For the SONIC MH variants, the trend is expected because as the number of hash functions decrease, the size of the merged community signatures decrease as well, which results in lower execution times.

The email-Enron graph shows trend similar to the amazon0601 and DBLP_coauthor graphs for the SONIC MH variants. However, SONIC II is only slightly better than the best MH variant. The facebook graph shows a different trend, where SONIC II performs worse than SONIC NV, and SONIC MH variants.

Summary Overall, the size and the structure of the graph have a significant impact on the merge variant to be selected. For large size networks, like amazon0601, SONIC II is the best option, whereas, denser graphs with smaller sizes, like facebook and twitter, SONIC MH variants can give better performance with best fitting number of hash functions. However, we need to keep in mind that with very few hash functions, the quality results of the SONIC MH variants are not good (Fig. 5). As such, we conclude that the SONIC II merge algorithm is the most robust option for general use.

7.4 The α and β effect

In this section, we report the impacts of the significant change threshold (α) and the merge factor (β) on the running time performance and community detection quality of SONIC.

For the significant change threshold experiment, we experimented on all datasets and selected the email-Enron dataset to show as representative since all the results

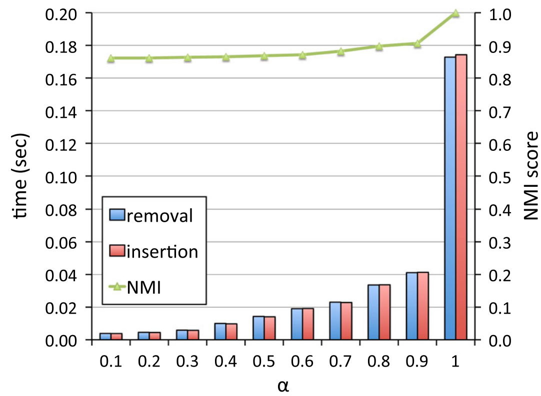

Fig. 10 Impact of α on the email-Enron dataset. Lower values of α provide significant throughputs with little impact on quality

are similar. We removed and inserted 1000 edges to our dataset using the SONIC NV algorithm and experimented with different α values from 0.1 to 1.0. The β value is fixed at 1.0. Remember that as the α value decreases, changes in the local community structure are regarded as less significant and, therefore, less merge operations occur. After the removals and insertions, we compute the NMI score of the communities resulting from an α value variant with respect to the communities resulting from a setting of $\alpha = 1.0$. This way, we can see how much divergence occurs due to the lower α values.

Figure 10 shows the NMI scores (using the right y-axis) for each α variant. The figure also shows the average time taken (on the left y-axis) for removing and inserting an edge as α varies. When we set α to 0.1, the quality degradation is not that significant, as the average NMI decreases by only 14 %. On the other hand, the removal/insertion of an edge executes 45 times faster. Another observation is that even if we set α to 0.9, we have speedups of 4.2 times, while sacrificing little in quality (10 %). These results show the advantage of using lower values for α parameter.

For the merge factor experiment, we chose the facebook dataset. As before, we used SONIC NV, but with a fixed value of α (1.0) and varying β (0.1 to 1). Figure 11 shows the running time (using the left y-axis) and the quality index (on the right y-axis) introduced in Sect. 7.1. We observe slower running times as β decreases. The reason is that there is an increased number of merges happening with lower values of β. On the other hand, the quality index is better for higher values of beta. The communities resulting from the default settings we have used in the experiments, i.e., $\beta = 1.0$, have the best quality index.

7.5 Scalability

In this section, we report the scalability of SONIC and its variants when processing the synthetic R-MAT graphs of different sizes, which vary from 2^{10} to 2^{18} vertices. We set both α and β to 1.0.

Figure 12 shows the average relative throughput of a single edge removal and insertion as a function of increasing R-MAT graph size. We compute relative throughputs

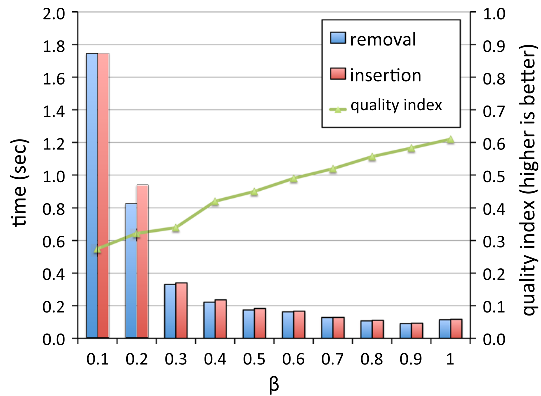

Fig. 11 Impact of β on the average execution time of insertions and removals. Runtimes get slower with lower values of β. Quality index increases with higher β values

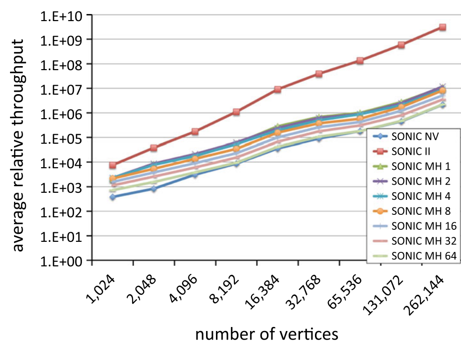

Fig. 12 Average of removal & insertion relative throughputs on R-MAT graphs as a function of the graph size. All merge variants show increasing relative throughputs with increasing scale. SONIC II has the best scalability, reaching 3.1B× relative throughput

with respect to the from-scratch computation of communities with the DEMON algorithm. All of the proposed algorithms present a scalable behavior with the increasing graph size. As the scale gets larger, SONIC II shows outstanding performance, reaching to 3.1B times relative throughput, which is 9 orders of magnitude better than the from-scratch computation. SONIC NV and SONIC MH variants show decent scalability results as well, reaching 6 and 7 orders of magnitude relative throughputs, respectively, but they are not better than SONIC II. Considering the quality being traded off by SONIC MH variants with small number of hashes, SONIC II turns out to be the best performing algorithm for our scalability experiments.

The scalability experiments illustrate the overall effectiveness of SONIC compared to batch re-computation. For instance, for an R-MAT graph of size 2^{18}, batch re-computation can reach the performance of SONIC only if it is performed with a period of *3.1*B or more insertions/deletions. Furthermore, the larger the graph, the bigger the difference. For small re-computation periods, the batch based approach cannot maintain high throughput. As an extreme case, if we are to re-compute every edge insertion/deletion, from-scratch computation is limited to 9 orders of magnitude lower throughput for the largest R-MAT graph. If we are to re-compute every million edge insertions/deletions, then it is limited to thousand times lower throughput. Clearly, SONIC has significant performance advantage and high practical value for a broad range of scenarios.

8 Conclusion

In this paper, we introduced incremental algorithms for the streaming overlapping community detection problem. The main benefit of these algorithms is that they provide maintenance of global community ids of the vertices in the graph, as updates are taking place. This avoids the from-scratch computation of communities and brings the ability to serve fresh community results to on-demand queries coming in at high rates.

Our core SONIC algorithm produces high quality communities and is efficient in terms of running time, when applied on the real-world and synthetic graphs of different sizes. The core SONIC algorithm is further enhanced by techniques such as the significant change detection, minhash based merge, and inverted-index based merge. For instance, we reach 9 orders of magnitude relative throughput with our SONIC variant that uses inverted-index based merge, compared to the from-scratch (non-incremental) alternatives, on real DBLP_coauthor network and synthetic R-MAT graphs of size 2^{18}. Given the dynamic nature of social networks and the importance of community detection in graph analytics, we believe that our incremental algorithms will be beneficial in many real-world applications with streaming update requirements. Thanks to the outstanding execution times, our algorithms will make it possible to analyze complete dynamic scenarios so that community evolution trends can be observed in real social networks.

References

Agarwal MK, Ramamritham K, Bhide M (2012) Real time discovery of dense clusters in highly dynamic graphs: identifying real world events in highly dynamic environments. Proc Very Large Data Bases Endow 5(10):980–991

Ahn YY, Bagrow JP, Lehmann S (2010) Link communities reveal multiscale complexity in networks. Nature 466(7307):761–764

Angel A, Sarkas N, Koudas N, Srivastava D (2012) Dense subgraph maintenance under streaming edge weight updates for real-time story identification. Proc Very Large Data Bases Endow 5(6):574–585

Broder AZ, Charikar M, Frieze AM, Mitzenmacher M (1998) Min-wise independent permutations. J Comput Syst Sci 60:327–336

Cazabet R, Amblard F, Hanachi C (2010) Detection of overlapping communities in dynamical social networks. In: IEEE second international conference on social computing (SocialCom), pp 309–314

Chakrabarti D, Kumar R, Tomkins A (2006) Evolutionary clustering. In: Proceedings of the 12th ACM SIGKDD international conference on knowledge discovery and data mining, KDD'06, pp 554–560

Charikar MS (2002) Similarity estimation techniques from rounding algorithms. In: Proceedings of the thiry-fourth annual ACM symposium on theory of computing, STOC'02, pp 380–388

Coscia M, Rossetti G, Giannotti F, Pedreschi D (2012) Demon: a local-first discovery method for overlapping communities. In: Proceedings of the 18th ACM SIGKDD international conference on knowledge discovery and data mining, KDD'12, pp 615–623

Cui W, Xiao Y, Wang H, Lu Y, Wang W (2013) Online search of overlapping communities. In: Proceedings of the 2013 ACM SIGMOD international conference on management of data, SIGMOD'13, pp 277–288

Danon L, Daz-Guilera A, Duch J, Arenas A (2005) Comparing community structure identification. J Stat Mech 9:P09008

DBLP (2014) http://www.informatik.uni-trier.de/~ley/db/. Accessed Mar 2014

Fortunato S (2010) Community detection in graphs. Phys Rep 486(3–5):75–174

Freeman LC (1982) Centered graphs and the structure of ego networks. Math Soc Sci 3(3):291–304

Gleich DF, Seshadhri C (2012) Vertex neighborhoods, low conductance cuts, and good seeds for local community methods. In: Proceedings of the 18th ACM SIGKDD international conference on knowledge discovery and data mining, KDD'12, pp 597–605

Goldberg M, Magdon-Ismail M, Nambirajan S, Thompson J (2011) Tracking and predicting evolution of social communities. In: Privacy, security, risk and trust (PASSAT) and 2011 IEEE third international conference on social computing (SocialCom), pp 780–783

Gregory S (2010) Finding overlapping communities in networks by label propagation. New J Phys 12(10):103,018

Hildrum K, Yu P (2005) Focused community discovery. In: Fifth IEEE international conference on data mining, pp 27–30

Indyk P, Motwani R (1998) Approximate nearest neighbors: towards removing the curse of dimensionality. In: Proceedings of the thirtieth annual ACM symposium on theory of computing, STOC'98, pp 604–613

Kim MS, Han J (2009) A particle-and-density based evolutionary clustering method for dynamic networks. Proc Very Large Data Bases Endow 2(1):622–633

Lancichinetti A, Fortunato S (2009) Community detection algorithms: a comparative analysis. Phys Rev E 80:056117

Lancichinetti A, Fortunato S, Kertesz J (2009) Detecting the overlapping and hierarchical community structure in complex networks. New J Phys 11(3):033,015

Leskovec J, Lang KJ, Mahoney M (2010) Empirical comparison of algorithms for network community detection. In: Proceedings of the 19th international conference on world wide web, WWW'10, pp 631–640

Lin YR, Chi Y, Zhu S, Sundaram H, Tseng BL (2008) Facetnet: a framework for analyzing communities and their evolutions in dynamic networks. In: Proceedings of the 17th international conference on world wide web, WWW'08, pp 685–694

McDaid AF, Greene D, Hurley N (2011) Normalized mutual information to evaluate overlapping community finding algorithms. CoRR. arXiv:1110.2515

Nanavati AA, Gurumurthy S, Das G, Chakraborty D, Dasgupta K, Mukherjea S, Joshi A (2006) On the structural properties of massive telecom call graphs: findings and implications. In: Proceedings of the 15th ACM international conference on information and knowledge management, CIKM'06, pp 435–444

Newman MEJ (2006) Modularity and community structure in networks. Proc Natl Acad Sci USA 103(23):8577–8582

Padrol-Sureda A, Perarnau-Llobet G, Pfeifle J, Muntes-Mulero V (2010) Overlapping community search for social networks. In: IEEE 26th international conference on data engineering (ICDE), 2010, pp 992–995

Palla G, Derenyi I, Farkas I, Vicsek T (2005) Uncovering the overlapping community structure of complex networks in nature and society. Nature 435:814–818

Qi GJ, Aggarwal CC, Huang TS (2013) Online community detection in social sensing. In: Proceedings of the sixth ACM international conference on web search and data mining, WSDM'13 pp 617–626

Raghavan UN, Albert R, Kumara S (2007) Near linear time algorithm to detect community structures in large-scale networks. Phys Rev E 76:036,106

Rees B, Gallagher K (2010) Overlapping community detection by collective friendship group inference. In: International conference on advances in social networks analysis and mining (ASONAM), 2010, pp 375–379

Rees BS, Gallagher KB (2012) Overlapping community detection using a community optimized graph swarm. Soc Netw Anal Min 2(4):405–417

Rees BS, Gallagher KB (2013a) Detecting overlapping communities in complex networks using swarm intelligence for multi-threaded label propagation. Complex networks. Springer, Berlin

Rees BS, Gallagher KB (2013b) Egoclustering: overlapping community detection via merged friendship-groups. The influence of technology on social network analysis and mining. Springer, Vienna

Rosvall M, Bergstrom CT (2008) Maps of random walks on complex networks reveal community structure. Proc Natl Acad Sci USA 105(4):1118–1123

Sarr I, Missaoui R, Lalande R (2013) Group disappearance in social networks with communities. Soc Netw Anal Min 3(3):651–665

SNAP (2014) Stanford network analysis package. http://snap.stanford.edu/snap. Accessed Mar 2014

Sozio M, Gionis A (2010) The community-search problem and how to plan a successful cocktail party. In: Proceedings of the 16th ACM SIGKDD international conference on knowledge discovery and data mining, KDD'10, pp 939–948

Twitter (2014) http://www.twitter.com/. Accessed Mar 2014

Wang F, Li T, Wang X, Zhu S, Ding C (2011) Community discovery using nonnegative matrix factorization. Data Min Knowl Discov 22(3):493–521

Whang JJ, Gleich DF, Dhillon IS (2013) Overlapping community detection using seed set expansion. In: Proceedings of the 22nd ACM international conference on information and knowledge management, CIKM'13 pp 2099–2108

Xie J, Chen M, Szymanski BK (2013) Labelrankt: incremental community detection in dynamic networks via label propagation. In: Proceedings of the workshop on dynamic networks management and mining, DyNetMM'13, pp 25–32

Xie J, Kelley S, Szymanski BK (2013) Overlapping community detection in networks: the state-of-the-art and comparative study. ACM Comput Surv 45(4):43:1–43:35

Yang J, Leskovec J (2012) Defining and evaluating network communities based on ground-truth. In: Proceedings of the ACM SIGKDD workshop on mining data semantics, MDS'12 pp 3:1–3:8

Yang J, Leskovec J (2013) Overlapping community detection at scale: a nonnegative matrix factorization approach. In: Proceedings of the sixth ACM international conference on web search and data mining, WSDM'13 pp 587–596

Zhang Y, Yeung DY (2012) Overlapping community detection via bounded nonnegative matrix tri-factorization. In: Proceedings of the 18th ACM SIGKDD international conference on knowledge discovery and data mining, KDD'12 pp 606–614