Soc. Netw. Anal. Min. (2016) 6:65
DOI 10.1007/s13278-016-0374-5

ORIGINAL ARTICLE

Tracking local communities in streaming graphs with a dynamic algorithm

Anita Zakrzewska[1] · David A. Bader[1]

Received: 19 December 2015 / Revised: 4 July 2016 / Accepted: 9 August 2016 / Published online: 30 August 2016

Abstract A variety of massive datasets, such as social networks and biological data, are represented as graphs that reveal underlying connections, trends, and anomalies. Community detection is the task of discovering dense groups of vertices in a graph. Its one specific form is seed set expansion, which finds the best local community for a given set of seed vertices. Greedy, agglomerative algorithms, which are commonly used in seed set expansion, have been previously designed only for a static, unchanging graph. However, in many applications, new data are constantly produced, and vertices and edges are inserted and removed from a graph. We present an algorithm for dynamic seed set expansion, which maintains a local community over time by incrementally updating as the underlying graph changes. We show that our dynamic algorithm outputs high-quality communities that are similar to those found when using a standard static algorithm. It works well both when beginning with an already existing graph and in the fully streaming case when starting with no data. The dynamic approach is also faster than re-computation when low latency updates are needed.

✉ Anita Zakrzewska
azakrzewska3@gatech.edu

David A. Bader
bader@cc.gatech.edu;
http://www.cc.gatech.edu/~bader

[1] Computational Science and Engineering, Georgia Institute of Technology, Atlanta, GA 30332, USA

1 Introduction

Graphs are used to represent relationships and communication between entities in fields such as Web traffic, financial transactions, online communications, and biology. A commonly studied feature of graphs is community structure. A graph community may be broadly defined as a set of vertices that is densely connected. In datasets representing online social networks, multiplayer games, or online project management, graph communities can correspond to groups of friends on social networks, online players, or officemates who work together on the same project. They can also be found in a variety of other graphs, such as protein–protein interaction networks.

Global community detection methods divide the entire graph into groups, which may form a partition or overlap. Local community detection finds a set relevant to a small set of vertices of interest, which we call seed vertices. This problem is sometimes called seed set expansion. Because many graphs may now have millions or billions of vertices, visualization is difficult and many computationally intensive algorithms cannot be run on commodity platforms. Seed set expansion can be used in such cases to extract a relatively small subgraph relevant to the vertices of interest. It can also used to find overlapping, global communities. In this work, however, we do not address the issues of global community detection, but focus on finding a local community for a given seed set. We use the terms seed set expansion and local community detection interchangeably. We also use the terms community and cluster interchangeably.

In contrast to static seed set expansion, which is run once on an unchanging graph, dynamic seed set expansion incrementally updates a local community. Edges may be inserted or removed to reflect evolving actions, communications, or relationships between entities. When the graph changes, the community of a seed vertex must be updated as well.

1.1 Contributions

We develop a dynamic algorithm to incrementally update a local community when the underlying graph changes. This algorithm is faster than re-computing, while maintaining good community quality. It can handle batch updates of various sizes and is easily parallelized for multiple expansions from different seeds. This paper presents the extended version of the work by Zakrzewska and Bader in Zakrzewska and Bader (2015). We evaluate the dynamic algorithm using a larger number of datasets and analyze how community size affects both the quality of communities output and the speedup over re-computation. We also extend our algorithm to the fully streaming case. Finally, we discuss an approach for tracking the interaction of entities of interest using dynamic seed set expansion.

2 Related work

Global community detection seeks to find dense groups of vertices in an entire graph. There is no single definition of a community, and various metrics are used. In general, structural graph communities have high internal edge density and few inter-community edges.

Existing global algorithms include random walk methods, spectral partitioning, label propagation, greedy agglomerative and divisive algorithms, and clique percolation. For surveys on community detection, see Fortunato (2010) Plantié and Crampes (2013) Tang and Liu (2010). While most methods partition the graph into mutually disjoint groups, there is a growing body of work in detecting overlapping communities Xie et al. (2013), where each vertex may belong to multiple groups. Methods include OSLOM Lancichinetti et al. (2011), link partitioning Evans and Lambiotte (2010), label propagation Xie and Szymanski (2012), clique percolation Derényi et al. (2005), and multiple local expansions Lancichinetti et al. (2009) Havemann et al. (2011) Lee et al. (2010).

Local community detection is the task of finding an appropriate cluster for a specific vertex or set of vertices. This has also been called seed set expansion, and we will use those two terms interchangeably. The task of finding a community relevant to a given set of vertices is interesting in its own right, but it can also be used for applications such as visualization. Additionally, local community detection can be used in cases where the entire graph is too large or changing to be completely known.

Clauset presents a greedy method for seed expansion that starts from a single vertex and then iteratively adds neighboring vertices to maximize the local modularity score Clauset (2005). The complexity of this approach for general graphs is $\mathcal{O}(n^2d)$, where d is the average degree and n the final community size, although this will depend on the graph structure. Riedy et al. assume the set of seed vertices may belong to different communities. Each vertex starts out in its own cluster. Merges may then occur between a seed's community and either another seed's community or a singleton vertex in order to maximize global modularity Riedy et al. (2011). Bagrow and Bollt use a different approach in the L-shell method Bagrow and Bollt (2005), in which vertices are added from successive shells. A shell is a set of vertices at a fixed distance from the seed. Unlike in Clauset (2005), multiple vertices are added to the cluster at once. This will likely improve running time, but may lower quality.

Local community detection has also been achieved through spectral methods. Andersen et al. Andersen et al. (2006) use the Spectral PageRank-Nibble method. Their final community minimizes conductance and is formed by adding vertices in order of decreasing PageRank values. In the random walk approach of Andersen and Lang Andersen and Lang (2006), some vertices in the seed set may not be placed in the final community. Our work addresses the problem of local community detection.

Several algorithms for detecting global, overlapping communities use a greedy, agglomerative approach and run multiple separate seed set expansions Lancichinetti et al. (2009) Havemann et al. (2011) Xie et al. (2013) Lee et al. (2010). Lanchichinetti et al. use single vertices as seeds Lancichinetti et al. (2009). Overlapping communities are produced by sequentially running expansions from a node not yet in a community. Lee et al. use maximal cliques as seed sets Lee et al. (2010). These approaches relate to our work because they find and expand cores in order to find global communities. However, in this work we do not address global communities. Rather, the focus is on finding the best community for a given seed or set of seeds.

Finding clusters in dynamic graphs poses a variety of challenges and the work in this field fits into several categories. A survey of work involving communities in dynamic graphs can be found in Aynaud et al. (2013) and Cazabet and Amblard (2014). Many methods seek to find the best community sequence given the dynamic data. One type of approach relies on the entire history of temporal data to obtain communities Tantipathananandh et al. (2007) Mucha et al. (2010) Jdidia et al. (2007). Using all snapshots over time may produce better community evolutions, but it can be computationally expensive and requires knowledge of all data. In many real life applications, online results are needed at regular intervals when the data change. This type of approach would need to be rerun whenever new data appear and may produce different community histories for each run.

Dynamic communities may also be computed in an online fashion, with clusters sequentially found for each new snapshot of data available. The online approach can be achieved by both maximizing the quality of clusters in the new snapshot and minimizing the transition cost from the previous community decomposition. Examples include evolutionary clustering by Chakrabarti et al. (2006) and FaceNet by Lin et al. (2009). Such an approach may result in smooth transitions.

Another body of work finds communities in online data by using the result previously found as a starting point for the detection algorithm. Often the aim is to reduce computational cost by reusing much of the previous community structure and incrementally updating. Our work falls into this category. The workflow for many of these algorithms is as follows. A current graph exists with previously detected communities. When a set of updates occur, the previous community structure is modified based on the updates. In some cases, a static community detection algorithm is then applied to this modified community state. We will refer to all algorithms that do not incrementally update results as static methods and those that do as dynamic or incremental methods. Note that an incremental algorithm may use a static algorithm at some steps. Some of these approaches are discussed below.

In Ning et al. (2010), the authors present incremental spectral clustering by updating eigenvalues. In Aynaud and Guillaume (2010), the authors use an incremental version of the Louvain algorithm Blondel et al. (2008). When the graph changes, instead of restarting from scratch, the previous community assignment is used as a starting point. Each node can then move to a different community to increase modularity. Shang et al. Shang et al. (2014) also present an incremental algorithm to update the Louvain method. After each update, communities either remain the same, are merged due to inter-community edge addition, or new vertices are placed in an existing or new community. Static Louvain clustering is then restarted. The MIEN algorithm Dinh et al. (2009) is an incremental version of greedy agglomerative community detection, such as CNM. After edges and vertices are added and removed at a time step, all directly affected vertices (endpoints of an inserted or removed edge or a vertex that was added or removed) are moved into their own singleton communities. Next, the chosen static community detection method is applied to current community structure to obtain any further merges. Aktunc et al. Aktunc et al. (2015) present an incremental version of the SLM algorithm Waltman and Eck (2013), which uses the clustering from the previous time step as a starting point, with new vertices in their own singleton communities. Takaffoli et al. Takaffoli et al. (2013) present an incremental version of the local community detection algorithm from Chen et al. (2009). The static algorithm greedily adds the best neighboring vertex to the community, based on a fitness function defined in the paper, after which all vertices are checked for removal. The incremental version uses the connected components of communities found at the previous time step as starting points before continuing expansion with the static algorithm. Riedy and Bader move vertices of inserted or deleted edges from their communities into singleton clusters before restarting their static, parallel, agglomerative algorithm Riedy and Bader (2013).

It is clear that many of these incremental approaches have a similar principle. Vertices directly affected by graph updates are removed from their previous community and either moved to a different one or placed as a singleton. A static algorithm then uses this modified community state as a starting point. In Sect. 4, we describe an incremental approach for local communities that follows this principle and then compare its output to that of our algorithm.

In addition to detecting communities, the question of tracking community operations has been studied Spiliopoulou (2011). Over time communities may grow, shrink, split apart, merge together, disappear, and re-appear. Detecting these operations requires both finding correct communities and matching them across time intervals. One of the challenges in doing so is cluster instability. The output of many algorithms is sensitive to small variations in input. This issue is relevant to tracking because community changes detected over time may be true changes or the result of algorithm instability. Hopcroft et al. Hopcroft et al. (2004) address this problem by using multiple runs to detect stable clusters, or natural communities, which are then tracked. In Asur et al. (2009), Asur et al. define and detect a variety of community and vertex level events using overlap of consecutive community snapshots. Greene et al. define a dynamic community as a sequence of similar static communities and match static communities in each new graph snapshot to the most recent frontier of a dynamic community Greene et al. (2010). In this way, intermittent dynamic communities can be discovered. Palla et al. use overlap to match communities over time in a co-authorship and a phone call graph to track how long they persist Palla et al. (2007). Our work does not focus on detecting such community operations over time. However, the relationship between community operations and our algorithm is discussed in Sect. 7.

3 Definitions and background

Let $G = \{V, E\}$ be a graph, where V is the set of vertices and E the set of undirected edges. An edge $(u, v, \omega) \in E$ consists of two unordered vertices u, v, and a weight ω. Let k_{in}^C be the sum of all edge weights interior to community C and k_{out}^C be the sum of all edge weights on the border of C.

$$k_{\text{in}}^C = \sum_{(u,v,\omega)\in E | u\in C \wedge v \in C} \omega \tag{1}$$

$$k_{\text{out}}^C = \sum_{(u,v,\omega)\in E | u\in C \wedge v \notin C} \omega \tag{2}$$

The quality of a community C is often measured using a fitness function. As there is no single definition of a community, many fitness functions are commonly used. Modularity, shown in Eq. 3, compares the number of intra-community edges to the expected number under a random null model Newman and Girvan (2004).

$$Q(C) = \frac{1}{|E|}\left(k_{\text{in}}^C - \frac{(2k_{\text{in}}^C + k_{\text{out}}^C)^2}{4|E|}\right) \tag{3}$$

Conductance is another popular fitness score and measures the community cut, or number of inter-community edges Chung (1997).

$$\phi(C) = \frac{k_{\text{out}}^C}{\min(2k_{\text{in}}^C + k_{\text{out}}^C, 2k_{\text{in}}^{V\setminus C} + k_{\text{out}}^{V\setminus C})} \tag{4}$$

Many overlapping community detection methods use a modified ratio of intra-community edges to all edges with at least one endpoint in the community, as in Eq. 5 Lancichinetti et al. (2009) Lee et al. (2010). Havemann *et al.* use a slightly modified version shown in Eq. 6, which allows vertices to remain singleton communities Havemann et al. (2011).

$$f(C)_{LFM} = \frac{2k_{\text{in}}^C}{(2k_{\text{in}}^C + k_{\text{out}}^C)^\alpha} \tag{5}$$

$$f(C)_{\text{MONC}} = \frac{2k_{\text{in}}^C + 1}{(2k_{\text{in}}^C + k_{\text{out}}^C)^\alpha} \tag{6}$$

The greedy local expansion algorithms used in Clauset (2005) Lee et al. (2010) Havemann et al. (2011) can be generalized to the form given by Algorithm 1. The community is iteratively expanded by adding the neighboring vertex that maximizes the chosen fitness function. The algorithm terminates when there exists no vertex whose inclusion in the community increases the fitness score. In Algorithm 1, *seed* represents the initial set of seed vertices, *fit*(C) the fitness score for a community C, and *Nb*(C) the set of vertices not in C with at least one neighbor in C. We use this static algorithm as part of our new dynamic method. For the experiments of Sect. 6, we use the fitness metric $f(C)_{\text{MONC}}$ from Eq. 6, though the approach will work for other fitness functions as well. We chose this metric because unlike modularity and conductance, it is local in nature. $f(C)_{\text{MONC}}$ and $f(C)_{LFM}$ have been used for local expansions to produce overlapping communities with good results. Although the two are very similar, $f(C)_{\text{MONC}}$ allows vertices to remain singletons, which is beneficial because not all vertices have a natural community.

Algorithm 1: Static, Greedy Seed Set Expansion

```
Data: graph $G$ and seed set $seed$
$C = seed$;
while progress do
    $maxscore = -1$;
    $maxvtx = -1$;
    for $v \in Nb(C)$ do
        $s(v) = fit(C \cup v) - fit(C)$;
        if $s(v) > maxscore$ then
            $maxscore = s(v)$;
            $maxvtx = v$;
        end
    end
    if $maxscore > 0$ then
        $C = C \cup maxvtx$;
    end
end
```

4 Motivation and alternative approach

Our dynamic seed set expansion algorithm incrementally updates the community when the underlying graph changes. Since incremental updates are faster than re-computation, our method can be used to improve performance for any application of seed set expansion, as described in Sect. 1. We begin with an initial graph G and perform a static seed set expansion, as in Algorithm 1, resulting in the initial community C. Next, a sequence of updates is applied to G and we incrementally update C to reflect changes in graph structure. Each graph update is of the form $(u, v, \Delta\omega)$, where u and v are edge endpoint vertices and $\Delta\omega$ is an increment or a decrement in edge weight. An edge insertion is represented by a weight increment to a nonexistent edge, while a deletion is represented by a decrement of the edge weight to 0.

To motivate our approach, we first discuss an alternative algorithm for dynamic seed set expansion and the problems it may run into. We will compare the quality of our algorithm to this alternative in Sect. 6. It is based on the updating approach found in Aynaud and Guillaume (2010) Shang et al. (2014) Aktunc et al. (2015) Riedy and Bader (2013). Each of these methods update only the directly affected vertices (endpoints of modified edges) and then use the updated community structure as a starting point for a static community detection algorithm. Although these are global approaches, they all update greedy static algorithms, and we can easily use the same principle to create an alternative local approach to test against.

After every edge update, we remove certain affected vertices from the local community before restarting the static expansion from Algorithm 1. We remove both endpoint vertices of a deleted intra-community edge and the member vertex of an inserted border edge. Edges outside the community do not require any vertex removals. Deleted border edges and inserted intra-community edges strengthen the membership of the corresponding endpoint vertices and therefore also do not require vertex removals. Unfortunately, this simple method has severe shortcomings. For example, the community may split apart and the algorithm may not be able to detect this because the neighbors of removed vertices remain in the community. This is shown in Fig. 1a, where the community is no longer optimal as it is actually composed of two natural communities. In the set of vertices that has split off and should be removed, most neighbors of each vertex are also in the community, and therefore, no vertex will be removed. Even if we evaluate multiple vertices at once for removal, the same problem may occur if the set that has split off is large enough.

The simple updating method may fail even when it outputs a valid community in the graph. This is because seed set expansion differs from global community detection in an important way: the local community is chosen for a particular seed set. The task is not simply to find any good community in the graph, but rather the appropriate community for the seed. Changes to the graph may shift the community C to one not centered around the original seed, as shown in Fig. 1b. While C may still have a good fitness score, it may not be a local community of the seed and would not be produced by a complete re-computation using static seed set expansion.

Given these considerations, quality evaluation for an updated community of a seed is more difficult than for general communities. We must consider not only the degree to which the chosen set of vertices resembles a community, but also whether it is a good community for the seed. A static seed set expansion algorithm detects the best community for the seed set using full information. Thus, one method of determining quality is to use the community found using static seed set expansion as a baseline and consider an incremental updating algorithm to be successful if it produces similar results.

5 Dynamic seed set expansion algorithm

5.1 Algorithm overview

The dynamic seed set expansion algorithm begins with the computation of a community using a static expansion on the initial graph as in Algorithm 1. When the algorithm begins, the community initially contains only the seed, and new vertices are then iteratively added. In each iteration, the neighboring vertices of the current community are potential new members and the vertex producing the greatest increase in the fitness score is chosen. The initial computation thus results in an ordered sequence of vertices added to the community and a corresponding sequence of nested sets, each with an increasingly greater fitness score. As the goal is to maintain a community centered around the seed, it is necessary to keep track of the order in which vertices were added.

Let m_i denote the ith vertex added as a member of the community in Algorithm 1 and $M_i = \{m_j \mid j \leq i\}$. M_i has an interior edge weight sum of $k_{i,in}$, a border edge weight sum of $k_{i,\mathrm{out}}$, and a fitness score of s_i. Note that $k_{i,in}$ is equal to $k_{\mathrm{in}}^{M_i}$ and $k_{i,\mathrm{out}}$ is equal to $k_{\mathrm{out}}^{M_i}$ as in Eqs. 1 and 2. If m_i is vertex v, then we say that v has position i or $\rho(v) = i$. We refer to this collection of sequences by $\Psi = \{m_i, k_{i,in}, k_{i,\mathrm{out}}, s_i \mid 0 \leq i \leq end\}$, as shown in Table 1. Here by *end* we represent the last position in the sequence Ψ, and M_{end} is the current community, which we also call C.

The dynamic algorithm works as follows. In phase A, we start with the initial graph and perform static seed set expansion (Algorithm 1) to produce Ψ. In phase B, a stream of graph updates is applied. With each graph update, the algorithm updates the community by modifying Ψ while ensuring that the sequence s_i remains monotonically increasing. That is, we require the updated community to contain vertices that, if added one by one as in the static algorithm, result in an increasing sequence s_i. This guarantees that the resulting community remains relevant to the source seed.

After each update to the graph, we modify the sequence of community members m_i to ensure that the corresponding fitness scores are increasing. The algorithm then detects any decreases in the sequence of fitness scores and removes

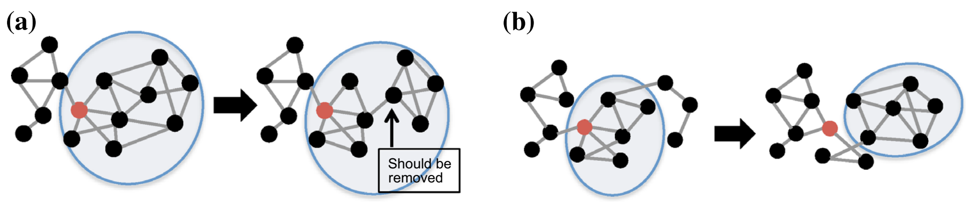


Fig. 1 Shortcomings of the simplistic algorithm from Sect. 4. Undesired community evolution shown *left* to *right*. **a** Undetected community splitting. **b** Undetected seed migration

Table 1 Community evolution sequence Ψ

Position	0	1	2	…	n
Members	m_0	m_1	m_2	…	m_n
Inner edges	$k_{0,\mathrm{in}}$	$k_{1,\mathrm{in}}$	$k_{2,\mathrm{in}}$	…	$k_{n,\mathrm{in}}$
Border edges	$k_{0,\mathrm{out}}$	$k_{1,\mathrm{out}}$	$k_{2,\mathrm{out}}$	…	$k_{n,\mathrm{out}}$
Fitness score	s_0	s_1	s_2	…	s_n

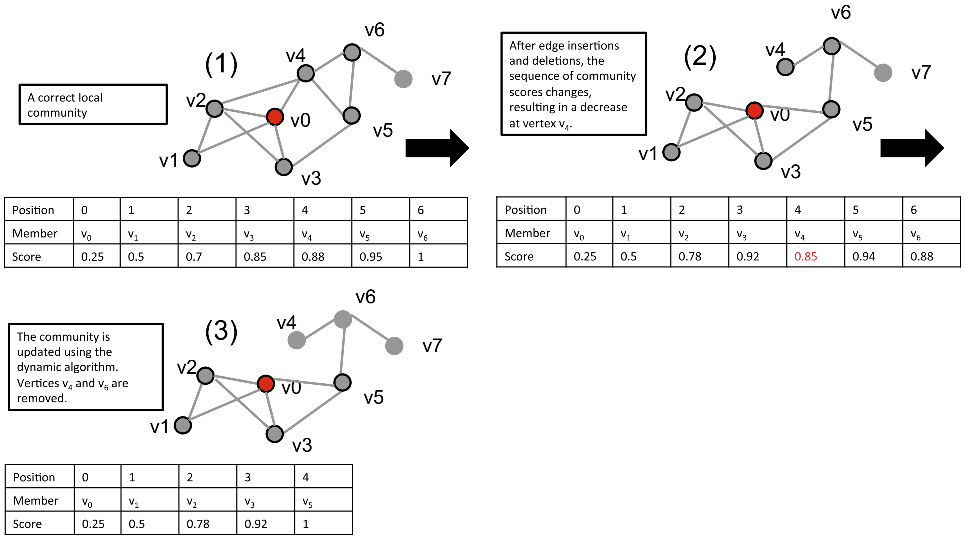
vertices from the community to eliminate any such decrease. Next, it checks whether any new vertices should be added and updates Ψ if needed.

This process is shown in Fig. 2, where the first image shows a community centered around the seed vertex. The order of the community members, m, is shown along with the corresponding scores. The second image shows the state of the community after edges have been inserted into and removed from the graph. The members of the community still remain the same, but the number of internal and border edges has changed. As a result, the sequence of scores is no longer increasing and the community is invalid. The third image of Fig. 2 shows community members adjusted using our dynamic algorithm so that the score sequence remains increasing. The details of the dynamic algorithm are given below.

5.2 Algorithm details

The dynamic algorithm updates Ψ after each graph update and ensures both that the sequence of fitness scores s_i remains monotonically increasing and that there are no additional vertices in G whose inclusion in the community would increase the fitness score further. For each batch of edge updates, the following four steps are performed (some may be omitted depending on the case). Further explanations are given later.

1. Values $k_{i,\mathrm{in}}$, $k_{i,\mathrm{out}}$, and s_i in Ψ are updated to reflect new internal and border edges.
2. Vertices that are endpoints of an updated edge are checked for removal. If a vertex v is removed, the community members are further pruned and Ψ is updated to reflect the pruning.
3. Ψ is scanned to check that the sequence of fitness scores s_i is still monotonically increasing. If a dip exists at position i, Ψ is truncated after position i (set $\Psi = \Psi_{0,i-1}$).
4. The static seed set expansion algorithm is restarted to check whether neighboring vertices in Nb(C) should be added to the community.


A correct local community
(1)
v0
v1
v2
v3
v4
v5
v6
v7
After edge insertions and deletions, the sequence of community scores changes, resulting in a decrease at vertex v4.
(2)
The community is updated using the dynamic algorithm. Vertices v4 and v6 are removed.
(3)

Position	0	1	2	3	4	5	6
Member	v_0	v_1	v_2	v_3	v_4	v_5	v_6
Score	0.25	0.5	0.7	0.85	0.88	0.95	1

Position	0	1	2	3	4	5	6
Member	v_0	v_1	v_2	v_3	v_4	v_5	v_6
Score	0.25	0.5	0.78	0.92	0.85	0.94	0.88

Position	0	1	2	3	4
Member	v_0	v_1	v_2	v_3	v_5
Score	0.25	0.5	0.78	0.92	1

Fig. 2 The process of storing and maintaining Ψ for a community is shown. The seed vertex v_0 is in *red*, and all members of the community have a *black* border. The *Top right* image shows a correctly detected local community with member vertices ordered and a corresponding sequence of increasing fitness scores. *Top right* shows the state of the graph and Ψ after edges have been inserted into and removed from the graph. The sequence of members is the same, but the corresponding scores have changed so the score sequence is no longer increasing at all points. In the *bottom left*, our dynamic algorithm has been applied to update the community. Vertices v_4 and v_6 are removed, and the sequence of scores is once again increasing (color figure online)

Algorithm 2: Dynamic Seed Set Expansion Algorithm

```
Data: edge $(u, v, \Delta\omega)$
//Step 1
if $u \in C$ and $v \in C$ then
    for $p = \rho(u)$ to $\rho(v) - 1$ do
        $k_{p,out} += \Delta\omega$;
        update $s_p$;
    end
    for $p = \rho(v)$ to end do
        $k_{p,in} += \Delta\omega$;
        update $s_p$;
    end
else if $u \in C$ and $v \notin C$ then
    for $p = \rho(u)$ to end do
        $k_{p,out} += \Delta\omega$;
        update $s_p$;
    end
//Step 2
if $\Delta\omega < 0$ and $u \in C$ and $v \in C$ then
    Queue $\leftarrow v$
else if $\Delta\omega > 0$ and $u \in C$ and $v \notin C$ and $u \neq seed$ then
    Queue $\leftarrow u$
else if $\Delta\omega > 0$ and $u \in C$ and $v \in C$ and $u \neq seed$ then
    Queue $\leftarrow u$
while Queue not empty do
    $u \leftarrow$ Queue;
    if $u \in C$ and $s_{\rho(u)-1} \geq s_{\rho(u)}$ then
        remove $u$ from $C$;
        update $\Psi$ to reflect removal;
        for neighbors $w$ of $u$ do
            if $w \in C$ and $\rho(w) > \rho(u)$ then
                Queue $\leftarrow w$;
            end
        end
    end
end
//Step 3
for $i = max(\rho(u), 1)$ to end do
    if $s_{i-1} \geq s_i$ then
        $end \leftarrow i - 1$;
        $C = M_{i-1}$;
        break;
    end
end
//Step 4
Check for new members using static algorithm;
```

The four steps of the dynamic algorithm are given in Algorithm 2. To save space, three simplifying assumptions are made. First, as edges are undirected, the order of vertices in an edge update $(u, v, \Delta\omega)$ is arbitrary. Therefore, we only consider $\rho(u) < \rho(v)$. Second, we assume one seed vertex, referred to as *seed*. The algorithm can handle any number of seed vertices, though this only makes sense if there is prior knowledge that all those vertices will belong to the same community. Third, we only consider a single edge update, even though the algorithm can handle batches of more than one update. To process a batch, several edge updates are accumulated before updating the graph and the community. Step 1 is first performed for each update in the batch, then step 2 is performed for each update, and finally steps 3 and 4 are only executed once per batch. Because steps 3 and 4 of the algorithm can be performed once per batch, accumulating a larger set of updates before processing results in a faster running time. On the other hand, the community is not updated as frequently. Using a larger batch size can be thought of as a compromise between a fully dynamic algorithm, which updates results immediately, and infrequently using the static algorithm to re-compute communities.

The complexity of obtaining one community with the static expansion algorithm is $\mathcal{O}(n^2 d)$, where n is the final community size and d is the average degree, though this is an overestimate for graphs whose vertices share many neighbors. In each iteration, in order to add the best candidate, all vertices neighboring the current set (the border set) are checked and the corresponding change in fitness score is computed. With n current members of average degree d, there may be nd distinct border set vertices to check, resulting in a time complexity of $\mathcal{O}(nd)$. In reality, however, many member vertices will have the same neighbors, so not all nd are distinct. By maintaining a list of current border set vertices, such as with a hash map, it is only necessary to process unique neighbors in each iteration. Thus, in practice, the time complexity of checking neighboring vertices each iteration may be less than $\mathcal{O}(nd)$. Assuming the community and border set are each represented with a hash map, adding a vertex v has $\mathcal{O}(d)$ complexity because each of v's neighbors may need to be either added to the border set or have their count of edges touching the community updated. To obtain a final community of size n, n iterations must be completed.

In the worst case, the dynamic algorithm must re-compute a large portion of the community. Because this re-computation is performed with the static expansion, the worst case time complexity is $\mathcal{O}(n^2 d)$ as well. In practice, many of the updates result in no decrease in the fitness score sequence so that only a scan of Ψ is needed. In this case, the complexity becomes $\mathcal{O}(n)$. The time complexity of each step is given next. Step 1 updates the values of $k_{i,in}$, $k_{i,\text{out}}$, and s_i by iterating once over each sequence in Ψ. The complexity is $\mathcal{O}(n)$ where the n is the length of Ψ. In step 2, if no vertices are removed, the complexity is $\mathcal{O}(1)$. The complexity of removing a vertex v in step 2 is $\mathcal{O}(d)$ because each neighbor of v must either also be checked for removal if it is a member or else have its count of edges touching the community decreased. With a community size of n, at most n vertices can be removed in step 2, taking $\mathcal{O}(nd)$ time. Step 3 requires a scan of the scores in Ψ and takes $\mathcal{O}(n)$ time. Step 4 uses the static expansion and therefore has a worst case complexity of $\mathcal{O}(n^2 d)$. The data structures required are a representation of the community and of the set of border vertices, both of which may be, for example, a hash map. The sequences in Ψ are each an array with length n. Additional details of each step are given next.

Step 1: First, $k_{i,in}$ and $k_{i,\text{out}}$ in Ψ are updated to reflect new edges internal to and on the border of the community.

The input is $(u, v, \Delta\omega)$, where u and v are vertices and $\Delta\omega$ the corresponding change in weight.

Step 2: Once Ψ has been updated in step 1, the fitness score sequence s_i may no longer be monotonically increasing. For some edge updates, keeping one of the edge endpoints in C may cause a decrease in fitness score. For an edge update $(u, v, \Delta\omega)$ with $v \in C$, we check whether $s_{\rho(v)-1} \geq s_{\rho(v)}$. If so, then keeping v as the $\rho(v)$ th member of C causes a non-increase in the fitness score. Accordingly, v is removed from C and Ψ must be updated: $k_{i,\text{in}}$, $k_{i,\text{out}}$, and s_i for $\rho(v) \leq i \leq end$ must be recalculated to reflect the fact that edges of v are no longer inside C. For each edge (v, u), if $u \in C$, the edge changes from an internal community edge (contributing to k_{in}) to a border edge (contributing to k_{out}). If $u \notin C$, the edge changes from a border edge to an edge with no influence on the fitness score. Only entries in Ψ after position $\rho(v)$ must be updated because previous entries were added to the community before v.

The removal of a vertex v from C in step 2 may cause other vertices in C to be removed as well. Candidate vertices are neighbors of v that were added to C after v. Let u be such a neighbor. At the time of u's inclusion in C, adding u increased the fitness score by increasing k_{in}, which was due to u having neighbors already in the community. However, at least one such neighbor was v, which is now no longer in C. Thus, it is possible that without v in C, u would not have enough neighbors in C to be added. We can check this by testing whether $s_{\rho(u)-1} \geq s_{\rho(u)}$. If v is removed from C, all such neighbors u of v in C are also checked. Neighbors of v added to C before v $(\rho(u) < \rho(v))$ need not be checked because they were added to C without the assistance of v. If any neighbor u of v is removed, then we must in turn check neighbors of u that were added to C after u. In order to perform the entire pruning process, a selective breadth first search beginning from v is performed, as in step 2 of Algorithm 2.

Step 2 is only performed if there is a specific candidate vertex for removal, which will always be an endpoint of an updated edge. An edge update $(u, v, \Delta\omega)$ can cause the removal of an endpoint v only when $s_{\rho(v)-1} \geq s_{\rho(v)}$ due to either a decrease in $k_{\rho(v),in}$ or an increase in $k_{\rho(v),\text{out}}$. This occurs in three cases. The first case is an edge decrement with $v \in C$, $u \in C$, and $\rho(u) < \rho(v)$. The second is an edge increment with $v \in C$ and $u \notin C$. The third is an edge increment with $v \in C$, $u \in C$, and $\rho(v) < \rho(u)$. This third case may seem counterintuitive because an intra-community edge is incremented, densifying the community. However, we must maintain an increasing sequence of fitness scores s_i in Ψ. As v was added to C before u, the edge between v and u is a border edge at position $\rho(v)$ and becomes internal only starting at position $\rho(u)$. Thus, by incrementing it, the sum of border edges $k_{\rho(v),\text{out}}$ increases. If, due to this increase, $s_{\rho(v)} \leq s_{\rho(v)-1}$, then v must be removed from C. In a later step, v may be re-added to C, but it must be removed from position $\rho(v)$ of Ψ because it causes a non-increase of s_i.

Step 3: Next we scan all of Ψ to check whether s_i are still monotonically increasing. If $s_{i-1} \geq s_i$, we truncate Ψ at position $i-1$ and set $end = i-1$. The community is now $C = M_{i-1}$ with fitness score s_{i-1}, and the sequence of fitness scores in Ψ is monotonically increasing.

Step 3 differs from step 2 because instead of a selective pruning, all of Ψ after the chosen position is deleted. It also serves a different purpose than step 2. We perform step 2 only when there is a specific candidate vertex to check for removal from C. Step 3 can check all vertices. For example, let the update be $(u, v, \Delta\omega)$, with $v \in C$, $u \in C$, $\rho(v) < \rho(u)$, and $\Delta\omega > 0$. By incrementing an intra-community edge, $k_{\rho(u),in}$ increases and the set $M_{\rho(u)}$ becomes denser. Thus, any vertex added after position $\rho(u)$ may no longer increase the fitness score, and all Ψ after position $\rho(u)$ must be scanned for such vertices. In addition, after step 2 the entire sequence of fitness scores may still no longer be increasing. Step 3 is more computationally expensive than step 2 because after detecting a score drop at position i, step 3 truncates all of Ψ after $i-1$, while step 2 selectively prunes. However, unlike step 2, it guarantees a monotonically increasing sequence of fitness scores. Of course, step 3 could replace step 2 entirely, but this increases running time. In Sect. 6, we show results for a modified dynamic algorithm that skips step 2.

Step 4: Finally, new vertices can be added to the community. Vertices neighboring C are checked for inclusion by running the loop in Algorithm 1. For every vertex added to C, Ψ is updated by appending a new entry that includes that vertex and the corresponding sum of interior edges $k_{\text{end,in}}$, sum of border edges $k_{\text{end,out}}$, and fitness score s_{end}.

5.3 Fully streaming version

The algorithm, as described above, begins with an initial existing graph and an initial community for this graph. However, the method can be extended to work on a fully streaming graph. Instead of starting with an initial graph, it can begin with an empty graph and use the dynamic algorithm to build and then maintain a community. Given a seed vertex v (or set of seed vertices), the community will be initialized containing only v (or the set of seeds), with no interior or border edges. After each edge insertion or deletion (or batch of such updates), the dynamic algorithm will update the community as usual. The community will begin to grow as edges are inserted around v. We show results for our dynamic algorithm both when beginning with an initial graph and for a fully streaming graph in Sect. 6.

6 Results

6.1 Experimental setup

We test our dynamic seed set expansion algorithm on six social network graphs, listed in Table 2. All datasets were obtained from the Koblenz Network Collection The koblenz network collection KONECT (2015). The graphs used represent Slashdot thread replies in which each vertex represents a user, wall posts between Facebook users, contact between users carrying wireless devices, replies between users on the Digg Web site, and posts of students on UC Irvine forums. As these graphs represent social interactions, they are likely to display group structure. These graphs were chosen because they contain timestamped data, allowing us to track real community evolution. We can insert and remove edges in the order given by timestamps.

We perform two types of experiments with each of these real social networks. In the first type, an initial graph is formed out of the first one-third of edges. Static seed set expansion is run on this initial graph as in Algorithm 1. The remaining two-thirds of edges are streamed in as edge insertions or edge weight increments. Edge deletions and edge weight decrements are created by removing old edges with a sliding window approach. Edges are inserted and removed in the same timestamped order, but with edge deletions lagging by a gap. This gap is given for each graph in Table 2. The update stream ends when no new edges can be inserted. The removal of old interactions as new interactions are added allows communities to evolve. For all graphs except the Manufacturing Emails graph, the sliding window gap is set to one-third of the edges. Because the Manufacturing Emails graph is very dense, we set both the initial number of edges and the sliding window gap to one-ninth, instead of one-third, of the edge count.

The second type of experiment performed on each graph uses the dynamic algorithm in a fully streaming manner. Instead of beginning with an existing initial graph, we begin with an empty graph and process all edge insertions and deletions as a stream. Because there is no initial community for the dynamic algorithm to begin with, this approach is more challenging. Any community must be incrementally built.

For seed vertices, we chose from each of the Facebook, Slashdot, Digg, and UC Irvine graphs 100 random vertices whose degree was in the top 75th percentile for the given graph and 100 random vertices whose degree was in the top 99th percentile. Both medium- and high-degree vertices were chosen to allow variety in the experiments. We did not choose low-degree vertices because the graphs tended to have skewed degree distributions so vertices with low-degree percentiles appeared only a few times in the dataset. For the Haggle Contact and Manufacturing Email graphs, because the total number of vertices was small, we simply chose 100 random vertices as seeds. We use the fitness function f_{MONC} with parameter $\alpha = 1.0$ and $\alpha = 0.8$. A smaller α allows for larger communities, and different α parameters were chosen to evaluate results for different types of communities. A value of $\alpha = 1.0$ is recommended Lee et al. (2010) Lancichinetti et al. (2009), and we used $\alpha = 0.8$ to obtain slightly large communities. Our results consist of the two experiment types for all seed vertices of the six datasets with both α parameters. The code was implemented in C and run on an 8 core Intel i7-2600 K CPU at 3.40 GHz.

Table 2 Datasets used as test graphs with number of edges and vertices and the size of the sliding window

Graph	Vertices	Edges	Sliding window
Facebook	46,952	876,993	292,331
Slashdot	51,083	140,778	46,926
Haggle contact	274	28,244	9414
Digg	30,398	87,627	29,209
Uc irvine forum	1899	59,835	11,240
Manufacturing emails	167	82,927	9214

6.2 Quality of communities

In order to compare the communities output by the dynamic algorithm to those from static re-computation, we repeatedly rerun the static algorithm as a graph is updated. This means that at any point in time (after each number of graph updates) for each seed vertex of each graph, we have the community computed with our dynamic algorithm and the community computed by running the static algorithm. The community obtained by the static algorithm can serve as a baseline ground truth. Of course, in a real dataset there may be more than one good local community for a seed vertex, but in the absence of real ground truth, using the results of the static algorithm is suitable.

The algorithm performance is measured by four metrics. The first is the ratio of the fitness scores in the dynamic algorithm vs. those obtained by re-computation. The second is the ratio of the size of the community output by the two methods. Because our approach maintains vertices that induce increasing fitness scores, the output will be relevant to the seed. Therefore even if the vertex members of the two sets differ, as long as the scores and sizes are similar, we can say that the communities are comparable in quality. Communities in real graphs are known to be overlapping, so there may be multiple sets for an algorithm to return.

The remaining two metrics are the precision and recall, which compare the overlap between the members of communities output by the dynamic algorithm and those output by the static algorithm. For a given graph update, let C_U be the community produced by the dynamic algorithm and C_R be the community output by the static method. Then Eqs. 7 and 8 give precision and recall.

$$precision = \frac{|C_U \cap C_R|}{|C_U|} \tag{7}$$

$$recall = \frac{|C_U \cap C_R|}{|C_R|} \tag{8}$$

Table 3 shows the mean score ratio, size ratio, precision, and recall for each graph. In both tables, the top section shows results when the first third edges are used to form an initial graph before using the dynamic algorithm. The bottom section shows results when starting with an empty graph. A batch size of 1 is used, which means that communities are updated after each edge.

Table 3a shows results for our algorithm. While both the fitness score and community size tend to be higher for the dynamic algorithm, the values are near 1 for most graphs, showing similar quality. Recall is higher than precision, which makes sense given that community sizes of the dynamic method are larger. The fact that average recall is high, with most values at or above 0.9, means that all relevant vertices are returned, which may be important for many applications. At the same time, the size of the community is not on average much larger, so not many additional vertices are returned. While precision is not as high as recall, the average is above 0.8 for half the graphs and above 0.7 for most graphs. As mentioned before, the lack of perfect overlap does not mean poor quality because a different equally good community may be returned. The fact that the fitness function score is high with community size similar to re-computation indicates good results.

Table 3 The average score ratio, size ratio, precision, and recall for each graph with a batch size of 1

Type	Graph	Score ratio	Size ratio	Precision	Recall
(a) Results for our dynamic algorithm					
Our Dynamic algorithm					
With initial	Facebook	1.07	1.56	0.72	0.86
	Slashdot	1.02	1.17	0.84	0.90
	Haggle contact	1.09	1.23	0.98	0.99
	Digg	1.06	1.33	0.77	0.92
	Uc Irvine forum	1.14	2.13	0.67	0.81
	Manufacturing	1.23	2.22	0.88	0.95
Full streaming	Facebook	1.08	1.83	0.63	0.84
	Slashdot	1.02	1.20	0.81	0.90
	Haggle contact	1.12	1.30	0.98	0.99
	Digg	1.07	1.41	0.75	0.92
	Uc Irvine forum	1.27	3.94	0.59	0.80
	Manufacturing	1.29	2.75	0.85	0.94
(b) Results for the alternative algorithm from Sect. 4					
Alternative approach					
With initial	Facebook	1.69	29.95	0.28	0.78
	Slashdot	1.44	15.82	0.34	0.86
	Haggle contact	2.20	6.00	0.94	0.99
	Digg	1.57	33.00	0.21	0.84
	Uc irvine forum	2.63	33.53	0.23	0.86
	Manufacturing	3.49	22.06	0.51	0.98
Full streaming	Facebook	2.44	196.21	0.17	0.78
	Slashdot	1.95	173.86	0.25	0.85
	Haggle contact	2.61	7.98	0.90	0.99
	Digg	2.36	212.98	0.12	0.84
	Uc irvine forum	3.16	56.60	0.12	0.89
	Manufacturing	3.53	23.30	0.48	0.94

Re-computing with the static algorithm is used as the baseline. The bottom section of each table shows results for the fully streaming case

The results of the dynamic algorithm are slightly more similar to those of the static algorithm when beginning with an initial graph, seen in the top half of Table 3a, compared to the fully streaming case, seen in the bottom half. This result intuitively makes sense as the dynamic algorithm does not start with a pre-computed community in the latter case. However, the results are still good in the fully streaming case. The recall and precision remain fairly high, the score ratio is close to 1 for all graphs, and the size ratio is below 2 for all but two graphs. This shows that our dynamic algorithm still works well when applied in a fully streaming manner.

Table 3b shows the results of the alternative algorithm described in Sect. 4. It is clear that the alternative approach grows very large communities with low precision. However, the problem is not restricted to the large size of the clusters. For half the graphs, the recall of the alternative is lower than that of our algorithm. Therefore, the communities also contain fewer of the ground truth members compared to our algorithm. When run with a small batch size, our algorithm performs better. With a large enough batch size, we expect that the quality of alternative algorithm would improve because most vertices would be removed from the community at each update. However, that would be very similar to re-computing from scratch.

Table 5 shows the average score ratio, size ratio, precision, and recall of both dynamic approaches (again with the results of the static algorithm as a baseline) for different community sizes. Averages are taken across all seed vertices on all six graphs. The runs for each seed vertex on each graph are divided by the average size of the community output by the dynamic algorithm. If the community of a seed vertex is on average 20, the values for that experiment will contribute to the 10–24 size bin in Table 5. Again, results are shown separately for the case when we start with an initial graph and the fully streaming case where we begin with an empty graph.

The results in Table 5a show that when the dynamic algorithm outputs smaller communities, the results are more similar to the output of the static method. However, the quality of the results remains high up to a size of 500 so our algorithm performs well on a wide range of sizes. Table 5b shows these statistics for the alternative approach described in Sect. 4. Even for the same community sizes, the precision of the alternative is much worse than that of our algorithm.

Figure 3a shows the average precision and recall of our dynamic algorithm, compared to the static algorithm, against the number of updates applied to the graph. We begin with the first third of the edges as an initial graph before streaming updates. A batch size of 1 is used. Averages are taken across multiple independent expansions, each with its own seed. The x-axis represents the number of insertions and deletions applied to the graph, and the y-axis shows the average precision or recall of communities output by the dynamic algorithm after that many updates. For each graph, the number of edge insertions and size of the deletion sliding window is given in Table 2.

Apart from the Facebook graph, there is no downward trend in either precision or recall of our approach as the number of insertions and deletions increases. This shows that we can use our dynamic algorithm for a large number of updates before a static recalculation should be applied. In fact, for these datasets, there is no indication that a static re-computation would be necessary.

Figure 3b shows the precision and recall of our algorithm for the fully streaming case when we begin with an empty graph. For several of the graphs, the precision and recall are lower in the fully streaming case, which makes sense given that the dynamic algorithm begins with no initial community. However, the values are still high in most cases with no downward trend. In fact, for the Digg, UC Irvine forum, and Slashdot forum graphs, the precision increases at the beginning. This suggests that the dynamic algorithm can improve its output.

6.3 Performance results

In this section, we evaluate the performance of using various batch sizes of updates with the dynamic algorithm. Using a batch size of x means that x edge updates are accumulated before applying them to the graph and updating the community. A smaller batch size provides updated results more frequently. When working with a static algorithm, in order to produce updated results for each batch, the algorithm must be rerun. For example, if there are 2000 edge insertions and deletions, then using a batch size of 1 would require 2000 batches to be processed, while using a batch size of 10 would require processing 200 batches. With a batch size of 10 and 200 batches, our algorithm, or the static algorithm, would be run 200 times, each time processing 10 updates.

In Fig. 4, we compare the running times of our dynamic algorithm and the static algorithm. For each seed set of each graph, the running time is measured as the total time taken to process all edge insertions and deletions. This is not the time of processing a single batch, but the time to process all batches. To fairly compute the running time of the static algorithm, we only re-compute with the static algorithm when an edge update occurs that may affect the community result. Many edge updates will affect vertices not related to the community, and we need not update in those cases. When accumulating a batch, we only count edge insertions with at least one endpoint vertex in the

Table 4 The average score ratio, size ratio, precision, and recall across all graphs binned by community size

Type	Size range	Score ratio	Size ratio	Precision	Recall
(a) Results for our algorithm					
Our dynamic algorithm					
With initial	1–4	1.00	1.02	0.98	0.99
	5–9	1.04	1.27	0.84	0.94
	10–24	1.05	1.34	0.78	0.89
	15–49	1.07	1.44	0.71	0.83
	50–99	1.14	1.98	0.73	0.84
	100–499	1.41	3.34	0.76	0.91
	500+	1.99	11.35	0.52	0.92
Full streaming	1–4	1.00	1.02	0.98	0.99
	5–9	1.04	1.31	0.84	0.95
	10–24	1.05	1.44	0.74	0.88
	15–49	1.08	1.64	0.67	0.83
	50–99	1.16	2.17	0.67	0.82
	100–499	1.58	5.23	0.69	0.90
	500+	2.77	25.87	0.32	0.91
Results for the alternative algorithm from Sect. 4					
Alternative approach					
With initial	1–4	1.02	1.04	0.99	1.00
	5–9	1.12	2.01	0.65	0.93
	10–24	1.20	3.05	0.49	0.88
	15–49	1.22	4.67	0.35	0.82
	50–99	1.32	7.65	0.27	0.79
	100–499	2.85	23.46	0.30	0.87
	500+	3.64	61.95	0.10	0.90
Full streaming	1–4	1.03	1.07	0.98	1.00
	5–9	1.22	2.22	0.61	0.92
	10–24	1.22	3.49	0.46	0.90
	15–49	1.24	5.06	0.31	0.81
	50–99	1.33	9.63	0.22	0.79
	100–499	2.87	24.25	0.27	0.87
	500+	3.76	85.00	0.06	0.92

A batch size of 1 is used. Re-computing with the static algorithm is used as the baseline. The bottom section of each table shows results for the fully streaming case

current community. We count edge deletions with at least one endpoint vertex either in the community or with neighbors in the community.

Figure 4a shows the mean, median, and quartiles of the speedup of the dynamic algorithm over static re-computation. The speedup is the ratio of the running time of the static algorithm over the running time of the dynamic algorithm. This speedup ratio is computed for every seed set of every graph. A speedup of x means that using the dynamic algorithm is x times faster than re-computing, so higher values are better. The x-axis shows the batch size used and the y-axis shows the speedup, both on a log scale.

It is clear that the advantage of the dynamic algorithm is greatest for small batch sizes. This is expected because the total running time of the static algorithm decreases proportionally as the batch size increases. When the batch size is increased by a factor of x, there are x times fewer batches and re-computation occurs x times less frequently. Because the running time of the static algorithm does not depend on the batch size, when the number of batches decreases by a factor of x, the total running time also decreases by a factor of x. The dynamic algorithm, however, performs more work as the batch size increases. Steps 3 and 4 are only run once per batch, but the decrease is not by a factor of x as some steps must occur the same number of times, once per edge update, regardless of the batch size. This is shown in Fig. 4b, which shows the mean running time, across all seed sets of all graphs.

For batch sizes of 1, 10, and 100, using the dynamic algorithm is faster than using the static algorithm. Of

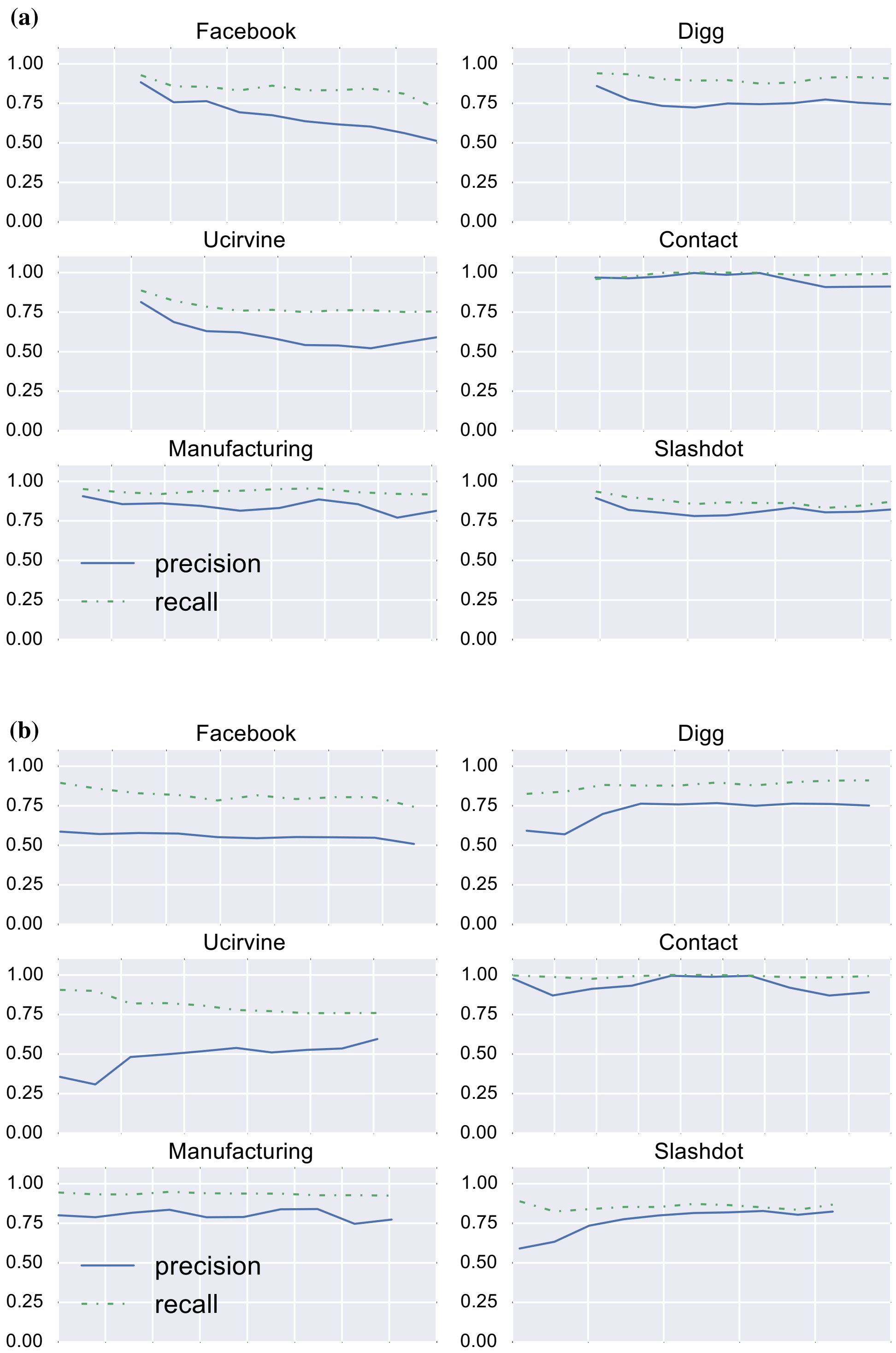


Fig. 3 The precision and recall of our dynamic algorithm, using results of re-computation with the static algorithm as a baseline, are shown over time for all graphs. The x-axis represents the number of edge insertions and deletions made to the graph. Thus, *left* to *right* shows how average precision and recall change as the graph changes. For each graph, all edges are inserted and edges are removed with a sliding window. The number of edges and size of sliding window is given in Table 2. **a** Starting with one-third of the dataset as the initial graph before streaming updates. **b** Fully streaming version: starting with an empty graph and streaming the entire graph as updates

course, the dynamic algorithm performs less work and solves a slightly different problem because it updates the results instead of computing from scratch. However, for applications where it is desirable to continually know the current community as the underlying graph changes, the community must be updated when the graph is modified. In such cases, the comparison of the dynamic and static algorithms is fair. The updated output can be obtained either by static re-computation or incrementally with a dynamic algorithm.

Figure 4c shows the mean dynamic speedup over re-computation for each graph. For a batch size of 1, we achieve mean speedups of two orders of magnitude on some graphs. The dynamic approach is faster for batch sizes of up to and including 100. Figure 4d shows the mean dynamic speedup over re-computation for different

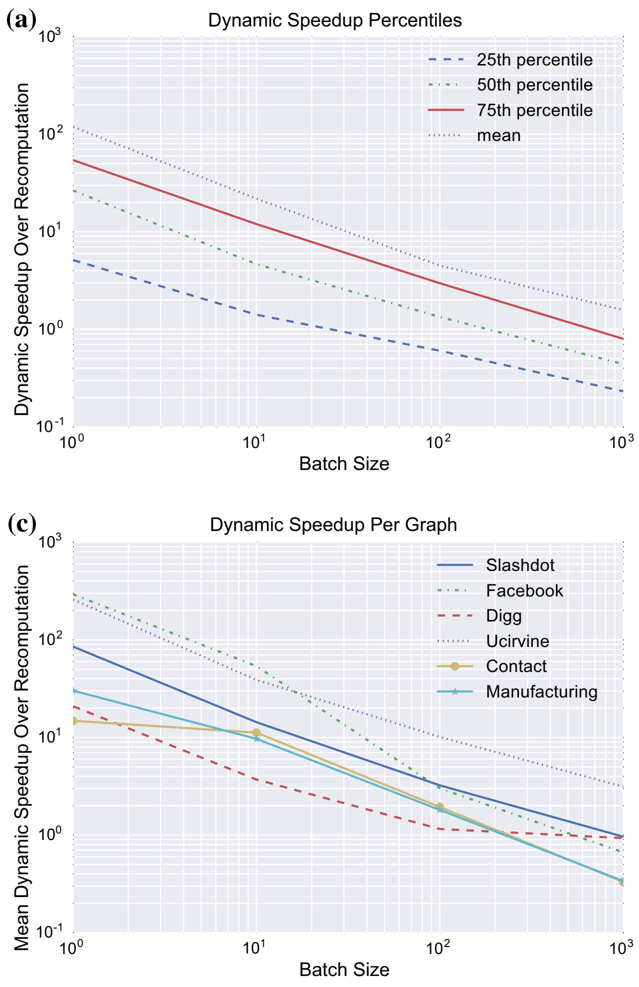


Fig. 4 Subfigures **a**, **c**, and **d** show the speedup of our dynamic algorithm compared to re-computing with the standard, static algorithm. A speedup of x means that using the dynamic algorithm is x times faster than re-computing. Subfigure **b** shows the running time. **a** The speedup mean, median, and quartiles are plotted for all runs on all graphs. **b** The mean running time over all seed sets is shown for both the dynamic and static methods. **c** The mean speedup is given for each of the six graphs. **d** The mean speedup is shown for different community sizes. *Each point* shows the mean using all expansions with an average community size in the specified range

community sizes. The dynamic approach performs relatively better for communities of size 25 and up. This is not surprising because the static algorithm will take longer to re-compute a large community.

7 Community operations

Tracking the evolution of communities through operations is an important topic when studying dynamic graphs Spiliopoulou (2011) Cazabet and Amblard (2014). When dealing with global clusters, it is typical to compare the overlap of communities found at different times in order to detect continuing, growing, shrinking, merging, splitting, appearing, and disappearing communities. The focus of our work has been to present an algorithm that maintains a local community over time by incrementally updating. However, we will briefly discuss potential approaches to detecting community operations to motivate future work.

First, it is necessary to address the number of seed vertices used to expand a single community and the implications in a dynamic context. Although each seed set consists of a single vertex in our experiments, the algorithm can be run with a seed set of multiple vertices of interest. However, because a single community is found for the seed set, it only makes sense to use multiple vertices if there is reason to believe that there exists some community containing all of them. In a dynamic graph, the seed vertices should remain in a single community over time. Therefore, for the community operations addressed below, we limit our discussion to the use of one seed vertex per community.

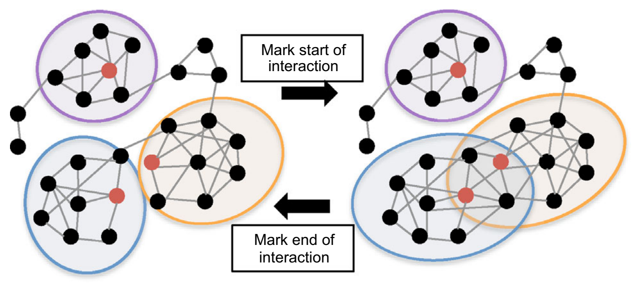


Fig. 5 This figure illustrates tracking key vertex interactions using seed set expansion. Vertices of interest are used as seeds and shown in *red*, and communities are marked with *shaded ovals*. *Left* to *right* marks the beginning of interaction between two seed vertices when the community of one seed grows to include another seed. *Right* to *left* marks the end of an interaction (color figure online)

For local communities, we can track the evolution of individual local communities and detect interactions between communities. A community can grow, shrink, and disappear, which is easy to detect by the size of the community. The volatility of a single cluster could be measured by comparing the members of consecutive times.

Independent local communities can merge and split by increasing and decreasing their overlap. If, for example, two vertices have highly overlapping communities, then this indicates similarity or interaction between them. A metric based on overlap, such as the Jaccard index, and a threshold could be used to determine when the communities of two seeds have merged or split. A simpler option marks a merge, or beginning of interaction, when the community of one seed includes another seed vertex. A split, or end of interaction, is marked when the latter seed is no longer included in the former seed's community. This is shown in Fig. 5.

8 Conclusion

We have presented a new algorithm that incrementally updates the local community of a seed set when the underlying graph changes. For a variety of real social networks with timestamps, this dynamic approach produces communities with high fitness scores and with high overlap with the communities produced by a standard greedy algorithm that must be rerun whenever the graph is updated. The dynamic algorithm works well both when beginning with an initial existing graph and in a fully streaming manner when beginning with no initial data. The dynamic method is faster than re-computation, and the performance improvement is greatest when low latency updates are needed. The speedup achieved varies based on the size of a local community, with the dynamic algorithm performing relatively better on large communities. The algorithm is easily parallelized across independent expansions. We also discuss an approach for tracking vertex interaction over time using local communities, which may be further addressed in future work.

Acknowledgments The work depicted in this paper was partially sponsored by Defense Advanced Research Projects Agency (DARPA) under agreement #HR0011-13-2-0001 (DARPA PERFECT). 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

Aktunc R, Toroslu IH, Ozer M, Davulcu H (2015) A dynamic modularity based community detection algorithm for large-scale networks: DSLM. In: Proceedings of the 2015 IEEE/ACM international conference on advances in social networks analysis and mining 2015. ACM, pp 1177–1183

Andersen R, Chung F, Lang K (2006) Local graph partitioning using pagerank vectors. In: 47th Annual IEEE symposium on foundations of computer science, 2006. (FOCS'06). IEEE, pp 475–486

Andersen R, Lang KJ (2006) Communities from seed sets. In: Proceedings of the 15th international conference on World Wide Web. ACM, pp 223–232

Asur S, Parthasarathy S, Ucar D (2009) An event-based framework for characterizing the evolutionary behavior of interaction graphs. ACM Trans Knowl Discov Data (TKDD) 3(4):16

Aynaud T, Fleury E, Guillaume JL, Wang Q (2013) Communities in evolving networks: definitions, detection, and analysis techniques. In: Mukherjee A, Choudhury M, Peruani F, Ganguly N, Mitra B (eds) Dynamics on and of complex networks, vol. 2. Springer, New York, pp 159–200

Aynaud T, Guillaume JL (2010) Static community detection algorithms for evolving networks. In: WiOpt'10: modeling and optimization in mobile, Ad Hoc, and wireless networks. IEEE, pp 508–514

Bagrow JP, Bollt EM (2005) Local method for detecting communities. Phys Rev E 72(4):046–108

Blondel VD, Guillaume JL, Lambiotte R, Lefebvre E (2008) Fast unfolding of communities in large networks. J Stat Mech: Theory Exp 10:P10008

Cazabet R, Amblard F (2014) Encyclopedia of social network analysis and mining, chapter dynamic community detection. Springer, New York, pp 404–414

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

Chen J, Zaiane OR, Goebel R (2009) Detecting communities in large networks by iterative local expansion. In: International conference on computational aspects of social networks, 2009. (CASON'09). IEEE, pp 105–112

Chung FR (1997) Spectral graph theory, vol 92. American Mathematical Society, Providence

Clauset A (2005) Finding local community structure in networks. Phys Rev E 72(2):026–132

Derényi I, Palla G, Vicsek T (2005) Clique percolation in random networks. Phys Rev Lett 94(16):160–202

Dinh TN, Xuan Y, Thai MT (2009) Towards social-aware routing in dynamic communication networks. In: 2009 IEEE 28th International on performance computing and communications conference (IPCCC). IEEE, pp 161–168

Evans T, Lambiotte R (2010) Line graphs of weighted networks for overlapping communities. Eur Phys J B 77(2):265–272

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

Greene D, Doyle D, Cunningham P (2010) Tracking the evolution of communities in dynamic social networks. In: 2010 international conference on advances in social networks analysis and mining (ASONAM). IEEE, pp 176–183

Havemann F, Heinz M, Struck A, Gläser J (2011) Identification of overlapping communities and their hierarchy by locally calculating community-changing resolution levels. J Stat Mech: Theory Exp 1:P01023

Hopcroft J, Khan O, Kulis B, Selman B (2004) Tracking evolving communities in large linked networks. Proc Natl Acad Sci 101(suppl 1):5249–5253

Jdidia MB, Robardet C, Fleury E (2007) Communities detection and analysis of their dynamics in collaborative networks. In: ICDIM, pp 744–749

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

Lancichinetti A, Radicchi F, Ramasco JJ, Fortunato S (2011) Finding statistically significant communities in networks. PLoS One 6(4):e18,961

Lee C, Reid F, McDaid A, Hurley N (2010) Detecting highly overlapping community structure by greedy clique expansion. In: 4th SNA-KDD workshop, p 3342

Lin YR, Chi Y, Zhu S, Sundaram H, Tseng BL (2009) Analyzing communities and their evolutions in dynamic social networks. ACM Trans Knowl Discov Data (TKDD) 3(2):8

Mucha PJ, Richardson T, Macon K, Porter MA, Onnela JP (2010) Community structure in time-dependent, multiscale, and multiplex networks. Science 328(5980):876–878

Newman ME, Girvan M (2004) Finding and evaluating community structure in networks. Phys Rev E 69(2):026–113

Ning H, Xu W, Chi Y, Gong Y, Huang TS (2010) Incremental spectral clustering by efficiently updating the eigen-system. Pattern Recognit 43(1):113–127

Palla G, Barabási AL, Vicsek T (2007) Quantifying social group evolution. Nature 446(7136):664–667

Plantié M, Crampes M (2013) Survey on social community detection. In: Ramzan N, van Zwol R, Lee J-S, Clüver K, Hua X-S (eds) Social media retrieval. Springer, London, pp 65–85

Riedy J, Bader DA (2013) Multithreaded community monitoring for massive streaming graph data. In: 2013 IEEE 27th international parallel and distributed processing symposium workshops and PhD Forum (IPDPSW). IEEE, pp 1646–1655

Riedy J, Bader DA, Jiang K, Pande P, Sharma R (2011) Detecting communities from given seeds in social networks. Technical Report GT-CSE-11-01, Georgia Institute of Technology. https://smartech.gatech.edu/handle/1853/36980

Shang J, Liu L, Xie F, Chen Z, Miao J, Fang X, Wu C (2014) A real-time detecting algorithm for tracking community structure of dynamic networks. arXiv preprint arXiv:1407.2683

Spiliopoulou M (2011) Evolution in social networks: a survey. In: Aggarwal CC (ed) Social network data analytics. Springer, pp 149–175

Takaffoli M, Rabbany R, Zaïane OR (2013) Incremental local community identification in dynamic social networks. In: Proceedings of the 2013 IEEE/ACM international conference on advances in social networks analysis and mining. ACM, pp 90–94

Tang L, Liu H (2010) Community detection and mining in social media. Synth Lect Data Min Knowl Discov 2(1):1–137

Tantipathananandh C, Berger-Wolf T, Kempe D (2007) A framework for community identification in dynamic social networks. In: Proceedings of the 13th ACM SIGKDD international conference on knowledge discovery and data mining. ACM, pp 717–726

The koblenz network collection KONECT (2015). http://konect.uni-koblenz.de

Waltman L, van Eck NJ (2013) A smart local moving algorithm for large-scale modularity-based community detection. The Eur Phys J B 86(11):1–14

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

Xie J, Szymanski BK (2012)Towards linear time overlapping community detection in social networks. In: Advances in knowledge discovery and data mining. Springer, pp 25–36

Zakrzewska A, Bader DA (2015) A dynamic algorithm for local community detection in graphs. In: Proceedings of the 2015 IEEE/ACM international conference on advances in social networks analysis and mining 2015, (ASONAM 15). ACM, New York, pp 559–564