2015 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining

A Dynamic Algorithm for Local Community Detection in Graphs

Anita Zakrzewska and David A. Bader
Computational Science and Engineering
Georgia Institute of Technology
Atlanta, Georgia 30332
azakrzewska3@gatech.edu, bader@cc.gatech.edu

***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 is constantly produced, and vertices and edges are inserted and removed from a graph. We present an algorithm for dynamic seed set expansion, which incrementally updates the community 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. The dynamic approach also improves performance compared to re-computation, achieving speedups of up to 600x.**

I. 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 on-line project management, graph communities correspond to groups of friends on Facebook, on-line 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. Existing global algorithms include random walk methods, spectral partitioning, label propagation, greedy agglomerative and divisive algorithms, and clique percolation [1]. While most methods partition the graph into mutually disjoint groups, there is a growing body of work in detecting overlapping communities [2]. However, there has been little work done in detecting a local community relevant to a small set of vertices of interest, which we call seed vertices. This problem is called seed set expansion. Because many graphs may now have 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, relevant subgraph.

A. Contributions

In contrast to static seed set expansion, which is run once on an unchanging graph, dynamic seed set expansion is a new area. 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. Here we develop a dynamic algorithm to incrementally update a local community when the underlying graph changes. This algorithm improves performance compared to re-computing. It can handle batch updates of various sizes and is easily parallelized for multiple expansions from different seeds. To the best of our knowledge, this is the first dynamic algorithm for greedy seed set expansion.

B. Definitions and Related Work

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.

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

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

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 Equation 3, compares the number of intra-community edges to the expected number under a random null model [3].

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

Conductance is another popular fitness score and measures the community cut, or inter-community edges [4].

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

Many overlapping community detection methods use a modified ratio of intra-community edges to all edges with at least one endpoint in the community, such as in Equation 5 [5].

$$f(C)_{MONC} = \frac{2k_{in}^C + 1}{(2k_{in}^C + k_{out}^C)^{\alpha}} \tag{5}$$

ASONAM '15, August 25-28, 2015, Paris, France
© 2015 ACM. ISBN 978-1-4503-3854-7/15/08 $15.00
DOI: http://dx.doi.org/10.1145/2808797.2809375

Andersen *et al.* [6] 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 [7] some vertices in the seed set may not be placed in the final community. Clauset gives a greedy method that starts from a single vertex and then iteratively adds neighboring vertices maximizing the local modularity score [8]. Riedy *et al.* expand multiple vertices via maximizing modularity [9].

Several algorithms for detecting global, overlapping communities use a greedy, agglomerative approach and run multiple separate seed set expansions [10] [5] [2]. Lancichinetti *et al.* run greedy seed set expansions, each with a single seed vertex [10]. 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 [11]. Havemann *et al.* [5] greedily expand cliques. Each of these greedy algorithms 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 the basis for our new dynamic method, presented in Section II, using the fitness metric $f(C)_{MONC}$ from Equation 5, though the approach will work for other fitness functions as well.

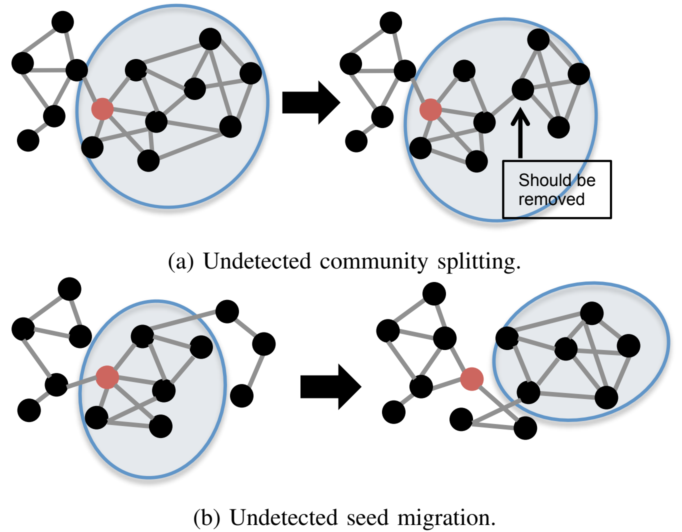
Algorithm 1: Static, Greedy Seed Set Expansion

```
Data: graph G and seed set seed
C = seed;
while progress do
    maxscore = −1;
    maxvtx = −1;
    for v ∈ Nb(C) do
        s(v) = fit(C ∪ v) − fit(C);
        if s(v) > maxscore then
            maxscore = s(v);
            maxvtx = v;
        end
    end
    if maxscore > 0 then
        C = C ∪ maxvtx;
    end
end
```

II. Dynamic Seed Set Expansion Algorithm

A. Motivation

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 Section I.



(a) Undetected community splitting.

(b) Undetected seed migration.

Fig. 1: Shortcomings of the simplistic algorithm from Section II-A. Undesired community evolution shown left to right.

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 non-existent edge, while a deletion is represented by a decrement of the edge weight to 0.

To motivate our approach, we first discuss a simple algorithm for dynamic seed set expansion and the problems it may run into. After every edge update, the simple method can check each endpoint vertex and calculate the fitness score of the community C both with and without that vertex. For example, consider an edge update $(u, v, \Delta\omega)$ with $\Delta\omega > 0$, $u \in C$ and $v \notin C$. We check $fit(C)$, $fit(C \cup v)$, and $fit(C \setminus u)$ and select the community update with the highest score. For an update with $\Delta\omega < 0$, $u \in C$, and $v \in C$, we calculate $fit(C)$, $fit(C \setminus v)$, $fit(C \setminus u)$, and $fit(C \setminus u, v)$ and choose the set with the highest score. Next, all vertices in $Nb(C)$ can be rechecked for inclusion in C. This is a simple local approach because only endpoints of updated edges are evaluated.

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 removal of no individual vertex increases the community score. This is shown in Figure 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

position	0	1	2	...	n
members	m_0	m_1	m_2	...	m_n
inner edges	$k_{0,in}$	$k_{1,in}$	$k_{2,in}$	...	$k_{n,in}$
border edges	$k_{0,out}$	$k_{1,out}$	$k_{2,out}$	...	$k_{n,out}$
fitness score	s_0	s_1	s_2	...	s_n

TABLE I: Community evolution sequence Ψ

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 Figure 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.

B. 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 $k_{i,out}$, and a fitness score of s_i. Note that $k_{i,in}$ is equal to $k_{in}^{M_i}$ and $k_{i,out}$ is equal to $k_{out}^{M_i}$ as in Equations 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 $\Psi = \{m_i, k_{i,in}, k_{i,out}, s_i \mid 0 \leq i \leq end\}$, as shown in Table I. 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.

For each update, we modify the sequence of community members m_i to ensure that the corresponding fitness scores are increasing. After a batch of updates, the algorithm detects any decreases in the sequence of fitness scores and removes vertices from the community to eliminate any such decrease. Next, it checks if any new vertices should be added and updates Ψ if needed.

C. 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 $k_{i,in}$, $k_{i,out}$, 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.

Algorithm 2 summarizes our method. To save space, two 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 only consider a single edge update, even though the algorithm can handle batch updates. In the case of a batch, step 1 is first performed for each edge, then step 2 is performed for each edge, and then steps 3 and 4 are only executed once.

The complexity of the static seed set expansion 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 the worst case, the dynamic algorithm must recompute a large portion of the community, in which case the 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 and the complexity becomes $\mathcal{O}(n)$. Additional details of each step are given next.

Step 1: First, $k_{i,in}$, $k_{i,out}$, and s_i 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.

Algorithm 2: Dynamic Seed Set Expansion Algorithm

```
Data: edge (u, v, Δω)
//Step 1
if u ∈ C and v ∈ C then
    for p = ρ(u) to ρ(v) − 1 do
        k_{p,out} += Δω;
        update s_p;
    end
    for p = ρ(v) to end do
        k_{p,in} += Δω;
        update s_p;
    end
else if u ∈ C and v ∉ C then
    for p = ρ(u) to end do
        k_{p,out} += Δω;
        update s_p;
    end
//Step 2
if Δω < 0 and u ∈ C and v ∈ C then
    Queue ← v
else if Δω > 0 and u ∈ C and v ∉ C and u ≠ seed then
    Queue ← u
else if Δω > 0 and u ∈ C and v ∈ C and u ≠ seed then
    Queue ← u
while Queue not empty do
    u ← Queue;
    if u ∈ C and s_{ρ(u)−1} ≥ s_{ρ(u)} then
        remove u from C;
        update Ψ to reflect removal;
        for neighbors w of u do
            if w ∈ C and ρ(w) > ρ(u) then
                Queue ← w;
            end
        end
    end
end
//Step 3
for i = max(ρ(u), 1) to end do
    if s_{i−1} ≥ s_i then
        end ← i − 1;
        C = M_{i−1};
        break;
    end
end
//Step 4
Check for new members using static algorithm;
```

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: $k_{i,in}$, $k_{i,out}$, 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 if $s_{\rho(u)-1} \geq s_{\rho(u)}$. 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 $k_{\rho(v),out}$. 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 counter-intuitive 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 $k_{\rho(v),out}$ 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 if the 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 section III 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 $k_{end,in}$, sum of border edges $k_{end,out}$, and fitness score s_{end}.

III. Results

We test the dynamic seed set expansion algorithm on three social network graphs, listed in Table II. The first graph is built from Slashdot data [12] [13], where vertices represent website users and edge weights correspond to numbers of replies between users. The second graph represents user replies on Digg [14], and the third is a computer science co-authorship network DBLP [15]. As these graphs represent social interactions, they are likely to display group structure. The Slashdot and Digg datasets contain timestamps, which define the order for edge insertion or incrementation. There are no timestamps in the DBLP dataset, so the edge order is defined by random permutation. In phase A we run static seed set expansion on the initial graph, which is formed from the first half of edges. In phase B, we sequentially insert the remaining edges and dynamically update the community. For edge decrements and removals, old data is removed from the graph. The fitness function f_{MONC} is used with $\alpha = 1.0$ and $\alpha = 0.5$. For seeds, we chose the top 1000 vertices in the graph by degree centrality, and then used each of them as a seed vertex in a separate seed set expansion. The code was implemented in C and run on an 8 core Intel i7-2600K CPU at 3.40GHz.

In order to compare the communities output by the dynamic algorithm to those from static re-computation, we repeatedly re-run the static algorithm as the graphs are updated. The algorithm performance is measured by three metrics. The first is the average ratio of the fitness scores in the dynamic algorithm vs. those obtained by re-computation. The average is taken across all timestamps and all expansions run. In Table II these ratios are above 1 for each graph, showing that the dynamic algorithm produces higher fitness scores. Two other metrics, precision and recall, compare the members of communities output by the dynamic and static algorithms. For a given graph update, let C_U be the of community produced by the dynamic algorithm and C_R be the community output by the static method. Then Equations 6 and 7 give precision and recall. In Table II precision and recall are in most cases

Graph	Vertices	Edges	Score Ratio	Recall	Precision
Slashdot, α=1.0	51,083	140,778	1.0	0.92	0.82
Slashdot, α=0.5	51,083	140,778	1.2	0.84	0.70
Digg, α=1.0	30,398	187,627	2.4	0.94	0.81
Digg, α=0.5	30,398	187,627	2.8	0.88	0.57
DBLP, α=1.0	317,080	1,049,866	1.1	0.91	0.87
DBLP, α=0.5	317,080	1,049,866	3.6	0.85	0.79

TABLE II: Quality of communities found with the dynamic algorithm compared to recomputing with the static algorithm

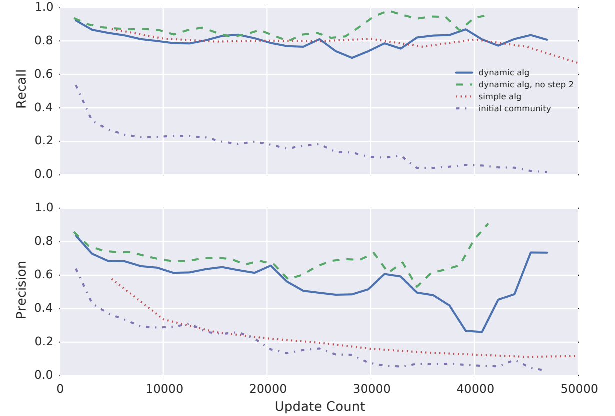


Fig. 2: The average precision and recall for the full dynamic algorithm (solid blue line), a modified version without step 2 (dashed green line), and the simple approach from Section II-A (red dotted line). The evolution of baseline communities vs. the initial communities is shown in purple dots and dashes.

above 0.8, indicating good quality. Also, better performance is obtained with α is set to 1.0, which gives smaller, denser communities.

$$precision = \frac{|C_U \cap C_R|}{|C_U|} \tag{6}$$

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

Figure 2 shows details of the dynamic algorithm's performance on the Slashdot graph. The average recall and precision are plotted against the number of updates. Averages are taken across multiple independent expansions, each with its own seed. The solid blue line represents the dynamic algorithm. Despite occasional dips, there is no downward trend in either precision or recall, showing that the community quality does not deteriorate over time. In addition to the full dynamic algorithm, we also evaluate a modified version, in which step 2, selective pruning, is not performed. As expected, without pruning, the quality of results, shown with the dashed green line, is somewhat higher, but with a slower running time. Next we examine if the high precision and recall of our algorithm could be due to the fact that the communities in the dataset simply do not change much as the graph is updated over time. We test this possibility using two approaches. First, for each

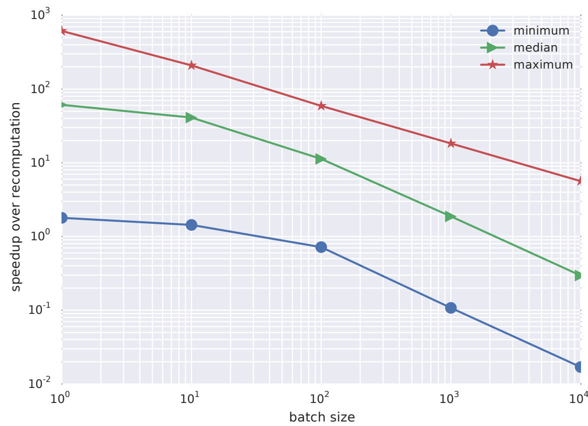


Fig. 3: The speedup of our dynamic algorithm over recomputing with the standard, static algorithm. Higher is better.

seed vertex, we take the initial community calculated using static seed set expansion. For each graph update over time, we use precision and recall to compare that initial community to the one produced by static re-computation. The results, plotted in Figure 2 with the purple dots and dashes, show the recall and precision of initial communities severely decrease as the graph evolves. This means that the communities in the Slashdot graph do change over time and our algorithm is able to update appropriately. The second test compares the precision and recall produced by the simple, local updating approach described in Section II-A. These are plotted with the red, dotted line in Figure 2. Although recall remains high, the precision deteriorates. This occurres because the simple approach expands communities to contain a very large portion of the graph.

Figure 3 plots the speedup of the dynamic algorithm over re-computation. For this experiment we used batches of updates of varying sizes. The x-axis shows the batch size used and the y-axis shows the ratio of re-computation running time to dynamic running time, 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 as the batch size is increased by a factor of x, re-computation occurs x times less frequently and its running time also decreases by a factor of x. The dynamic algorithm also performs less work with larger batch sizes, as 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 regardless of the batch size. The dynamic algorithm produces speedups of over $600x$ for a batch size of 1 and over $200x$ for a batch size of 10. The median speedup is over $60x$ for a batch size of 1 and remains greater than $1x$ for batch sizes up to 1000. This suggests that the dynamic algorithm is faster than re-computation even for large batches of 1000 updates. On graphs that contain larger communities, we would expect the dynamic algorithm to be even more advantageous, as re-computing a large community from scratch takes longer.

IV. Conclusion

We have presented a new algorithm that incrementally updates the local community of a seed set when the underlying graph changes. Our approach produces communities with high fitness scores and similar to the communities produced by a standard greedy algorithm that must be re-run whenever the graph is updated. 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, which may be addressed in future work.

Acknowledgment

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

References

[1] S. Fortunato, "Community detection in graphs," *Physics Reports*, vol. 486, no. 3, pp. 75–174, 2010.
[2] J. Xie, S. Kelley, and B. K. Szymanski, "Overlapping community detection in networks: The state-of-the-art and comparative study," *ACM Computing Surveys (CSUR)*, vol. 45, no. 4, p. 43, 2013.
[3] M. E. Newman and M. Girvan, "Finding and evaluating community structure in networks," *Physical review E*, vol. 69, no. 2, p. 026113, 2004.
[4] F. R. Chung, *Spectral graph theory*. American Mathematical Soc., 1997, vol. 92.
[5] F. Havemann, M. Heinz, A. Struck, and J. Gläser, "Identification of overlapping communities and their hierarchy by locally calculating community-changing resolution levels," *Journal of Statistical Mechanics: Theory and Experiment*, vol. 2011, no. 01, p. P01023, 2011.
[6] R. Andersen, F. Chung, and K. Lang, "Local graph partitioning using pagerank vectors," in *Foundations of Computer Science, 2006. FOCS'06. 47th Annual IEEE Symposium on*. IEEE, 2006, pp. 475–486.
[7] R. Andersen and K. J. Lang, "Communities from seed sets," in *Proceedings of the 15th international conference on World Wide Web*. ACM, 2006, pp. 223–232.
[8] A. Clauset, "Finding local community structure in networks," *Physical review E*, vol. 72, no. 2, p. 026132, 2005.
[9] J. Riedy, D. A. Bader, K. Jiang, P. Pande, and R. Sharma, "Detecting communities from given seeds in social networks," 2011.
[10] A. Lancichinetti, S. Fortunato, and J. Kertész, "Detecting the overlapping and hierarchical community structure in complex networks," *New Journal of Physics*, vol. 11, no. 3, p. 033015, 2009.
[11] C. Lee, F. Reid, A. McDaid, and N. Hurley, "Detecting highly overlapping community structure by greedy clique expansion," in *4th SNA-KDD Workshop*, 2010, p. 3342.
[12] V. Gómez, A. Kaltenbrunner, and V. López, "Statistical analysis of the social network and discussion threads in slashdot," in *Proceedings of the 17th international conference on World Wide Web*. ACM, 2008, pp. 645–654.
[13] "Slashdot threads network dataset – KONECT," Aug. 2014. [Online]. Available: http://konect.uni-koblenz.de/networks/slashdot-threads
[14] M. D. Choudhury, H. Sundaram, A. John, and D. D. Seligmann, "Social synchrony: Predicting mimicry of user actions in online social media," in *Proc. Int. Conf. on Computational Science and Engineering*, 2009, pp. 151–158.
[15] J. Yang and J. Leskovec, "Defining and evaluating network communities based on ground-truth," in *Proc. ACM SIGKDD Workshop on Mining Data Semantics*. ACM, 2012, p. 3.