A Fast and Efficient Incremental Approach toward Dynamic Community Detection

Neda Zarayeneh[1] and Ananth Kalyanaraman[1]

***Abstract*—Community detection is a discovery tool used by network scientists to analyze the structure of real-world networks. It seeks to identify natural divisions that may exist in the input networks that partition the vertices into coherent modules (or communities). While this problem space is rich with efficient algorithms and software, most of this literature caters to the static use-case where the underlying network does *not* change. However, many emerging real-world use-cases give rise to a need to incorporate dynamic graphs as inputs.**

In this paper, we present a fast and efficient incremental approach toward dynamic community detection. The key contribution is a generic technique called Δ-screening, which examines the most recent batch of changes made to an input graph and selects a subset of vertices to reevaluate for potential community (re)assignment. This technique can be incorporated into any of the community detection methods that use modularity as its objective function for clustering. For demonstration purposes, we incorporated the technique into two well-known community detection tools. Our experiments demonstrate that our new incremental approach is able to generate performance speedups without compromising on the output quality (despite its heuristic nature). For instance, on a real-world network with 63M temporal edges (over 12 time steps), our approach was able to complete in 1056 seconds, yielding a 3$\times$ speedup over a baseline implementation. In addition to demonstrating the performance benefits, we also show how to use our approach to delineate appropriate intervals of temporal resolutions at which to analyze an input network.

I. Introduction

Community detection is a fundamental problem in many graph applications. The goal of community detection is to identify tightly-knit groups of vertices in an input network, such that the members of each community share a high concentration of edges among them than to the rest of the network. Owing to its ability to reveal natural divisions that may exist in a network (in an unsupervised manner), community detection has become one of the fundamental discovery tools in a network scientists toolkit. The operation is widely used in a variety of application domains including (but not limited to) social networks, biological networks, internet and web networks, citation and collaboration networks, etc. Designing efficient algorithms and implementations for community detection has been an area of active research for well over a decade. While theoretical formulations are known to be NP-Hard [1], there are a number of efficient heuristics and related software already available. A comprehensive review of community detection methods and related applications is available in [2]. However, most of the existing tools target static networks; whereas most real-world networks are dynamic, where vertices and edges can be added and/or removed over a period of time.

Owing to the increasing availability of dynamic networks, the problem of dynamic community detection has become an actively researched topic of late, and multiple methods have been proposed over the last decade (e.g., [3]–[5]). In Section II we present a brief review of such related works. Despite these advances, a key remaining challenge in the design of these algorithms is in quickly identifying the parts of the graph that are likely to be impacted by a change (or collectively by a recent batch of changes), so that it becomes possible to update the community information with minimal recomputation effort.

Contributions: In this paper, we propose an algorithmic technique and a corresponding incremental approach that would complement the developments made in dynamic community methods, and in particular those that use the modularity function [6] as their clustering objective. More specifically, the main contributions are as follows:

i) We visit the problem of identifying vertex subsets that are likely to be impacted by the most recent batch of changes made to the graph. To address this problem, we present a technique called Δ-screening, which can be efficiently implemented and incorporated as part of existing dynamic community algorithms that use modularity.
ii) To demonstrate and evaluate this technique, we incorporated the technique into two well-known classical community detection methods—namely, the Louvain method [7] and the SLM method [8]—thereby generating two incremental clustering implementations.
iii) Using these two implementations, we present a thorough experimental evaluation on both synthetic and real-world inputs. Our results show that the Δ-screening technique is effective in pruning work (to reduce recomputation effort) without compromising on output quality.
iv) In addition to demonstrating its performance benefits, we also show how to use our approach to delineate appropriate intervals of temporal resolutions at which to analyze an input network.

[1]N. Zarayeneh and A. Kalyanaraman are with the School of Electrical Engineering and Computer Science, Washington State University, Pullman, WA 99164, USA. Email: neda.zarayeneh@wsu.edu, ananth@wsu.edu

ASONAM '19, August 27-30, 2019, Vancouver, Canada
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6868-1/19/08
http://dx.doi.org/10.1145/3341161.3342877

II. Related Work

Algorithms to compute dynamic communities over time-evolving graphs can be broadly classified into two types.

One class of methods follows a *two-step* strategy of first identifying the best set of communities for the current time step and then subsequently mapping them onto the communities from previous generations to track evolution. Hopcroft *et al.* [9] present a method in which a static community detection tool is individually applied to the graphs at all time steps and the results are later combined. Greene *et al.* [4] propose a variation where they use a matching-based heuristic to map communities of the latest time step to the communities of previous generations.

In general, the two-step strategy is better suited if the magnitude and/or complexity of changes to the input graph is more drastic or random. However, the strategy suffers two drawbacks: It can make community tracking difficult as an application of static community detection at every time step may *not* necessarily preserve previous communities, thereby making the outputs non-deterministic. Secondly, these approaches could become expensive to run on large inputs due to recomputation and community tracking overheads.

The other class of methods for detecting dynamic communities follows a more *incremental* strategy where communities from the previous generation(s) are propagated and updated using changes reflected in the current time step. Maillard *et al.* [10] propose a modularity-based incremental approach extending upon the classical Clauset-Newman-Moore static method [11]. Aktunc *et al.* [3] propose a method DSLM as an extension of its static predecessor [8]. Xie *et al.* [12] present an incremental method based on label propagation which is a fast heuristic. Saifi and Guillaume [13] provide a way to track and update community "cores" across time steps. Zakrzewska and Bader [5] present another variant that tracks the communities of a selected set of seed vertices in the graph. The FacetNet approach introduced by Lin *et al.* [14] is a hybrid approach that operates with a dual objective of maximizing modularity for the current time step while trying to also preserve as much of the previous generation communities.

In general, the incremental strategy has the advantage in runtime (because of the reuse of community information from previous steps), and it also has the advantage of outputting a relatively stable set of communities across time steps. The technique of Δ-`screening` proposed in this paper is aimed at helping these incremental methods to be able to quickly identify the relevant parts of the graph that are potentially impacted by a recent batch of changes, so that the computation effort in the incremental step can be reduced without compromising on clustering quality.

We note here that most of the existing methods use modularity or one of its variants as the objective function for optimizing community structure. Bassett *et al.* [15] propose and evaluate the choice of alternative null hypothesis models. A more thorough survey of dynamic community detection methods is presented in [16].

III. Method

A. Basic Notation and Terminology

A *dynamic graph* $G(V, E)$ can be represented as a sequence of graphs $G_1(V_1, E_1), \dots, G_T(V_T, E_T)$, where $G_t(V_t, E_t)$ denotes the graph at time step t; we use $n_t = |V_t|$ and $M_t = |E_t|$. In this paper, we consider only undirected graphs. The graphs may be weighted—i.e., each edge $(i, j) \in E_t$ is associated with a numerical positive weight $\omega_{ij} \geq 0$; if the graphs are unweighted, then the edges are assumed to be associated with unit weight, without loss of generality. We denote the neighbors of a vertex i as $\Gamma(i) = \{j \mid (i, j) \in E_t\}$. We use m_t to denote the sum of the weights of all edges in G_t—i.e., $m_t = \sum_{(i,j)\in E_t} \omega_{ij}$. We denote the degree of a vertex i by $d(i)$. The *weighted degree* of a vertex i, denoted by $d_\omega(i)$, is the sum of weights of all edges incident on i.

In this paper, we consider incrementally growing dynamic graphs, where edges and vertices can be added (but not deleted) from one time step to another. This implies that $V_t \supseteq V_{t-1}$ and $E_t \supseteq E_{t-1}$, for all $1 < t \leq T$. We denote the newly added edges at any time step t as $\Delta_t = E_t \setminus E_{t-1}$. We denote the set of communities detected at time step t as $\mathcal{C}_t$. Note that, by definition, $\mathcal{C}_t$ represents a partitioning of the vertices in V_t—i.e., each community $C \in \mathcal{C}_t$ is a subset of V_t; all communities in $\mathcal{C}_t$ are pairwise disjoint; and $\bigcup_{C\in\mathcal{C}_t} C = V_t$.

For any vertex $i \in V_t$, we denote the community containing i, at any point in the algorithm's execution, as $C(i)$, following the convention used in [17]. Also, let $e_{i\to C}$ denote the sum of the weights for the edges linking vertex i to vertices in community C—i.e., $e_{i\to C} = \sum_{j\in C\cap\Gamma(i)} \omega_{ij}$. Furthermore, let a_C denote the sum of the weighted degrees of all vertices in C—i.e., $a_C = \sum_{i\in C} d_\omega(i)$.

Given the above, the *modularity*, Q_t, as imposed by a community-wise partitioning $\mathcal{C}_t$ over G_t, is given by [6]:

$$Q_t = \frac{1}{2m_t}\Big(\sum_{i\in V_t} e_{i\to C(i)} - \frac{1}{2m_t}\sum_{C\in\mathcal{C}_t} a_C^2\Big) \qquad (1)$$

Given a community-wise partitioning on an input graph, the *modularity gain* that can be achieved by moving a particular vertex i from its current community to another target community (say $C(j)$) can be calculated in constant time [7]. We denote this modularity gain by $\Delta Q_{i\to C(j)}$.

B. Problem Statement

Definition III.1. Dynamic Community Detection: *Given a dynamic graph $G(V, E)$ with T time steps, the goal of dynamic community detection is to detect an output set of communities $\mathcal{C}_t$ at each time step t, that maximizes the modularity Q_t for the graph $G_t(V_t, E_t)$.*

Since the static version of the modularity optimization problem is NP-Hard [1], it immediately follows that the dynamic version is also intractable. For the static version, a number of efficient heuristics have been developed (as surveyed in [2]). These approaches can be broadly classified into three categories: divisive approaches [18], [19], agglomerative approaches [6], [11], and multi-level approaches [8], [20], [21]. Of these, the multi-level approaches have demonstrated to be fast and effective at producing high-quality partitioning in practice. In Algorithm 1 we show generic algorithmic pseudocode for this class of approaches. While they vary in the specific details of how each step is implemented, they share several common traits (note that this description is for the static use-case):

- At the start of each level, all vertices are assigned to a distinct community id.
- An iterative process is initiated, in which all vertices are visited (in some arbitrary order) within each iteration, and a decision is made on whether to keep the vertex in its current community, or to migrate it to one of its neighboring communities. This decision is typically made in a local-greedy fashion. For instance, in the Louvain algorithm [7], a vertex migrates to a neighboring community that maximizes the modularity gain of that vertex—i.e., let $j \in \Gamma(i) \cup \{i\}$. Then,

$$C(i) \leftarrow \arg\max_{C(j)} \Delta Q_{i \to C(j)}$$

- When the net modularity gain resulting from an iteration drops below a certain threshold τ, the current level is terminated (i.e., intra-level convergence), and the algorithm compacts the graph into a smaller graph by using the information from the communities. This procedure represents a graph coarsening step, and the coarsened graph is subsequently processed using the same iterative strategy until there is no longer an appreciable modularity gain between successive levels.

Algorithm 1 succinctly captures the main steps of the multi-level approaches.

Algorithm 1: Abstraction for Multi-level Approaches

```
Input: $G(V, E)$
Output: An assignment $\Pi : V \to \mathbb{Z}$
1  Initialize $\Pi$ by setting $\mathcal{C}(v) \leftarrow \{v\}$, $\forall v \in V$
2  repeat
3      repeat
4          for each $v \in V$ do
5              Compute a local (greedy) function $g(v, \mathcal{C}(v))$
6              $C(v) \leftarrow$ Update community assignment for $v$ using the results from $g(v, \mathcal{C}(v))$
7          end
8          Compute a global quality function $Q$ for $\Pi$
9      until Convergence based on $Q$
10     Review communities of $\Pi$ (optional step)
11     $G(V, E) \leftarrow$ Perform graph compaction for next level
12 until Convergence based on $Q$
13 return $\Pi$
```

C. A Naive Algorithm for Dynamic Community Detection

A simple approach for dynamic community detection is to directly apply the static algorithm (Algorithm 1) on the graph at every time step. However, such an approach suffers from multiple limitations. First, it completely ignores the communities identified at previous time steps, causing outputs to become non-deterministic. Furthermore, by ignoring the previous community information, the algorithm is essentially forced to recompute from scratch, and as a result, evaluate the community affiliation for *all* vertices at each time step. This can mean wasteful computation. For instance, it is reasonable to expect that only those vertices in the "vicinity" of a newly added edge to be impacted by the addition. However, the naive strategy cannot exploit such proximity information, thereby negatively impacting performance particularly for large real-world networks where event-triggered changes tend to happen in a more localized manner at different time steps.

D. An Incremental Approach via Δ-screening

Here, we present an alternative approach in which we first identify a subset of vertices to evaluate at the start of every time step, using the changes Δ_t. The idea is to identify all (or most) those vertices whose community affiliation could potentially change due to Δ_t; the remaining vertices will simply retain their previous community assignments. This new filtering technique, which we call Δ-screening (and abbreviated as ΔS), is generic enough to be applied to any incremental clustering approach that uses modularity. For the purpose of this paper, we demonstrate it on multi-level approaches.

More specifically, let $Static(G)$ denote any static community detection algorithm of choice, that takes in an input (static) graph G and outputs a set of communities $\mathcal{C}$. Then, our incremental approach is as follows.

1) At $t = 1$, we call $Static(G_1)$ to output $\mathcal{C}_1$.
2) For each subsequent time step $t > 1$:
   a) We initially assign each pre-existing vertex $i \in V_t \cap V_{t-1}$ to the same community label as $\mathcal{C}_{t-1}(i)$. Each remaining vertex (i.e., newly added at t) is assigned a distinct (new) community label.
   b) Next, we call a function Δ-screening(G_t, Δ_t) that returns a subset of vertices $\mathcal{R}_t \subseteq V_t$. This subset corresponds to the set of vertices that have been selected for processing during time step t.
   c) Subsequently, we call $Static\Delta S(G_t, \mathcal{R}_t)$, which is a variant of $Static(G_t)$ that loads G_t but visits only the vertex subset $\mathcal{R}_t$ for evaluation during each iteration—i.e., a modification to the for loop of line #4 in Algorithm 1. Note that this procedure that uses $\mathcal{R}_t$ is only relevant to the iterations at the first level, as in the subsequent levels, the algorithm uses compacted versions of the same graph.

To demonstrate the Δ-screening technique, we modified two well-known community detection methods: Louvain algorithm [7], and smart local moving (SLM) algorithm [8]. We call the resulting modified implementations as dLouvain-Δs and dSLM-Δs respectively.

Note that there is also a simpler incremental version that can be implemented for both these methods—by following all steps outlined in our incremental approach *except* for Δ-screening and instead trivially setting $\mathcal{R}_t = V_t$. For a comparative assessment of the Δ-screening strategy, we implemented this *baseline* version as well—we refer to the resulting two implementations as dLouvain-base and dSLM-base[1] respectively.

E. The Δ-screening Scheme

In what follows, we describe our Δ-screening scheme in detail. Given the graph (G_t) and changes (Δ_t) at time step t, the goal of Δ-screening is to identify a vertex subset $\mathcal{R}_t \subseteq V_t$ for reevaluation at time step t—i.e., any vertex that is added to $\mathcal{R}_t$ will be evaluated for potential migration by the iterative clustering algorithm (Algorithm 1); all other vertices are not evaluated (i.e., they retain their respective community assignment from the previous time step $t-1$). Our Δ-screening scheme is also a heuristic and it does not guarantee the reproduction of the results from the corresponding baseline version—dLouvain-base or dSLM-base—which are also heuristics. The main objective here is to save runtime by reducing the number of vertices to process, without significantly altering the quality. Despite its heuristic nature, however, our Δ-screening scheme is designed to preserve the key behavioral traits of the baseline version (as we show in lemmas later in this section).

Algorithm: We assume that Δ_t is stored as a list of *ordered* pairs of the form (i, j). This implies that for each newly added edge (i, j), there will be two entries stored in Δ_t: (i, j) and (j, i), as the input graph is undirected. We refer to the first entry (i) of an ordered pair ((i, j)) as the "source" vertex and the other vertex (j) as the "sink". Let S_Δ denote the set of all source vertices in Δ_t, and $T_\Delta(i)$ denote the set of all sinks for a given source i.

Algorithm 2 shows the steps for Δ-screening. We initialize $\mathcal{R}_t$ to $\emptyset$. Subsequently, we examine all edges of Δ_t in the sorted order of its source vertices. Sorting helps in two ways: It helps us to consider all the new edges incident on a given source vertex collectively and identify the edge that (locally) maximizes the net modularity gain (consistent with line #4 of Algorithm 2). This way we are able to mimic the behavior of the baseline versions which also use the same greedy scheme to migrate vertices. This sorted treatment also helps reduce overhead by updating $\mathcal{R}_t$ in a localized manner (relative to the source vertices) and avoiding potential duplications in the computations associated with a vertex.

Once sorted, we read the adjacency list for each source vertex (Algorithm 2:line #3), identify a neighbor (j_*) that maximizes the modularity gain (line #4), and update $\mathcal{R}_t$ based on that vertex (line #8). However, prior to updating $\mathcal{R}_t$, we check if the selected vertex j_* has a better incentive to move to i's community $C_{t-1}(i)$ (line #7); if that happens, then $\mathcal{R}_t$ is not updated from source i and instead, that decision is left/deferred until j_* is visited as the source. This way we avoid making conflicting decisions between source and sink while decreasing the time for processing (by reducing $\mathcal{R}_t$ size). Note that we only use the direction of migration that results in the larger of the two gains for updating $\mathcal{R}_t$. The decision to migrate itself is deferred until the stage of execution of the iterative algorithm. In other words, the Δ-screening procedure does *not* modify the state of communities, but it sets the stage for which communities to be visited during the main iterative process.

Algorithm 2: Δ-screening at time step t

```
Input: $G_t$, $\Delta_t$
Output: $\mathcal{R}_t$: Subset of vertices for reeavaluation
1  $\mathcal{R}_t \leftarrow \emptyset$
2  Sort edges in $\Delta_t$ based on the source
3  for each $i \in S_\Delta$ do
4      Let $j_* \leftarrow \arg\max_{j \in T_\Delta(i)}\{\Delta Q_{i \rightarrow \mathcal{C}_{t-1}(j)}\}$
5      Let $gain_1 \leftarrow \Delta Q_{i \rightarrow \mathcal{C}_{t-1}(j_*)}$
6      Let $gain_2 \leftarrow \Delta Q_{j_* \rightarrow \mathcal{C}_{t-1}(i)}$
7      if $gain_1 \geq gain_2$ and $gain_1 > 0$ then
8          $\mathcal{R}_t \leftarrow \mathcal{R}_t \cup \{i, j_*\} \cup \Gamma(i) \cup \mathcal{C}_{t-1}(j_*)$
9      end
10 end
11 return $\mathcal{R}_t$
```

The main part of Algorithm 2 is on line #8, where $\mathcal{R}_t$ is updated. Our scheme adds the following subset to $\mathcal{R}_t$: vertices i and j_*, all neighbors of i ($\Gamma(i)$), and all vertices in the community containing j_*. In what follows, using a combination of lemmas, we show that the $\mathcal{R}_t$ so constructed is positioned to capture all (or most of the) "essential" vertices that are likely to be impacted by the edge additions in Δ_t. In other words, if a vertex is not added to $\mathcal{R}_t$, it can be concluded that it is less likely (if at all) to be impacted by the changes to the graph, and therefore it can stay in its previous community state—thereby saving runtime.

In all these lemmas, for sake of convenience (and without loss of generality), we analyze the potential impact of the event represented by moving i to j_*'s community. Intuitively, the key to populating $\mathcal{R}_t$ is in anticipating which vertices are likely to alter their community status triggered by this migration event. Fig. 1 shows the different representative cases that originate for consideration in our lemmas.

First, we claim that any vertex that is a neighbor of i can be potentially impacted.

Lemma III.1. *If $i' \in \Gamma(i)$, then the community state for i' could potentially alter at time step t if i migrates to $C(j_*)$.*

Proof. There are two subcases: (A) if i' is also in $C_{t-1}(i)$; and (B) otherwise.

Subcase (A) is represented by vertex label i_1 in Fig. 1. If i were to leave $C_{t-1}(i)$, the strength of the connection of i_1 to $C_{t-1}(i)$ can only weaken because of a decrease in the positive

[1]We note that our dSLM-base implementation is in effect same as [3].

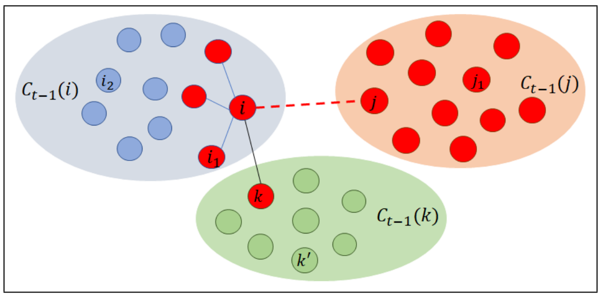


Fig. 1: Figure showing the impact of a newly added edge (i, j), shown in red dotted line. Representative cases of candidate vertices for potential inclusion in $\mathcal{R}_t$ are shown highlighted as labelled vertices. Note that we follow the naming convention of denoting the community containing a vertex i by $C(i)$. Also, note that not all edges are shown.

term of the modularity (Eqn. 1)). Even if the negative term of the same equation also decreases (due to the departure of i from $C_{t-1}(i)$, it may or may not be sufficient to keep i_1 in $C_{t-1}(i)$. Therefore, we add i_1 to $\mathcal{R}_t$.

Subcase (B) is represented by vertex label k in Fig. 1. Here, k is in a community different from $C_{t-1}(i)$. However, the situation with k is similar to that of i_1 in subcase (A), as k's connection to its present community could potentially weaken if it discovers a stronger connection to $C(j_*)$ as a result of i's move. Therefore, we add k to $\mathcal{R}_t$. $\square$

Next, we analyze the potential of vertices that are in $C_{t-1}(i)$ but *not* in $\Gamma(i)$ to be impacted by the migration of i. In fact, we conclude that there is no need to include such vertices in $\mathcal{R}_t$.

Lemma III.2. *If $i' \in C_{t-1}(i)$, then at time step t, a change to the community state of i' is possible,* only if *i' is also a neighbor of i.*

Proof. We have already considered the case where $i' \in \Gamma(i)$ (as part of Lemma III.1). Therefore we only need to consider the case where $i' \notin \Gamma(i)$. This is represented by vertex label i_2 in Fig. 1. Since i_2 is already in $C_{t-1}(i)$ and since i_2 does not share an edge with i, a departure of i from $C_{t-1}(i)$ can only positive reinforce i_2's membership in $C_{t-1}(i)$. This can be shown more formally by comparing the modularity gains associated with i_2. Owing to space limitations, we show the expanded proof in Appendix: Section A.1[2] in [22]. In summary, vertex i_2 will have little incentive to change community status and therefore can be excluded from $\mathcal{R}_t$. $\square$

Next, we analyze the potential impact of i's migration on members of j_*'s community.

Lemma III.3. *If $j_1 \in C_{t-1}(j_*)$, then at time step t, a change to the community status of any such j_1 is possible.*

[2]https://arxiv.org/abs/1904.08553

Proof. Regardless of whether j_1 shares a direct edge with j_* or not, the migration of a new vertex (i) into its present community $(C_{t-1}(j_*))$ increases the negative term in Eqn. 1. This may or may not be accompanied with an increase in the positive term as well (depending on whether j_1 shares an edge with the incoming vertex i). In either case, however, we need to re-evaluate the community status of such vertices. Therefore, we add j_1 to $\mathcal{R}_t$. $\square$

Finally, we analyze the impact of i's potential migration from $C_{t-1}(i)$ to $C_{t-1}(j_*)$, on vertices that are in neither of those two communities *and* are also not in $\Gamma(i)$.

Lemma III.4. *If $k \in V_t \setminus \{C(i) \cup C(j)\}$, then at time step t, unless k is also in $\Gamma(i)$, there is no need to include k in $\mathcal{R}_t$.*

Proof. We consider only vertices $k \notin \Gamma(i)$, as the other case was already covered in Lemma III.1. There are three subcases: (A) k shares an edge with some vertex in $C_{t-1}(i)$ except i; (B) k shares an edge with some vertex in $C_{t-1}(j_*)$; and (C) k has no neighbors in $C_{t-1}(i)$ or $C_{t-1}(j_*)$. However, in none of these cases a migration of i to $C_{t-1}(j_*)$, could create an incentive for k to move to $C_{t-1}(j_*)$. This is shown formally in Appendix: Section A.2 in [22]. $\square$

IV. Experimental Evaluation

A. Experimental Setup

Input data: For experimental testing, we used a combination of synthetic and real-world networks. Table I shows the input statistics for the inputs used.

TABLE I: Input network statistics. See Fig. B.2 of Appendix in [22] for more details on individual time steps.

Input	Input graph	No. vertices	No. edges (cumulative)	No. timesteps
Synthetic	50k_ll	50,000	2,362,448	10
	50k_hh	50,000	2,367,024	10
	5M_ll	5,000,000	213,656,492	10
	5M_hh	5,000,000	213,771,700	10
Real-world	Arxiv HEP-TH	27,770	352,807	11
	sx-stackoverflow	2,601,977	63,497,050	2-28

As synthetic inputs, we used a collection of streaming networks available on the MIT Graph Challenge 2018 [23]. We used two types of networks: i) Low block overlap, Low block size variation (abbreviated as "`ll`"), and ii) High block overlap, High block size variation (abbreviated as "`hh`"). These two types are in the increasing order of their community complexity (`ll<hh`). However, in both cases, the number of edges grows linearly with time step (see Appendix Fig. B.2 in [22]). The datasets are available from sizes of 1K nodes to 20M nodes, and each of these datasets has ten time steps. For our testing purposes, we used the 50K and 5M datasets.

As real-world inputs, we used two networks downloaded from SNAP database [24]:

1) **Arxiv HEP-TH:** This is a citation graph for 27,770 papers (vertices) with 352,807 edges (cross-citations). For the purpose of analysis we treated edges to be undirected.

The dataset covers papers published between 1993 and 2003. Consequently, we partitioned this period into 11 time steps (one for each year).

2) **sx-stackoverflow:** This is a temporal network of interactions on Stack Overflow, with $2,601,977$ vertices (users) and $63,497,050$ temporal edges (interactions). User-user interactions can be of multiple types but for the purpose of our analysis we treated all pairwise interactions equivalently and used only the first instance of each interaction as an edge.

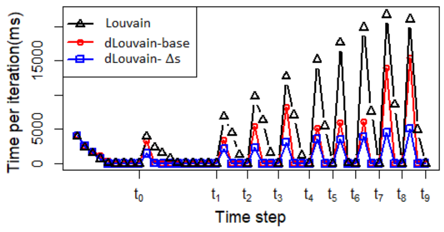
Implementations tested: In our experiments, we tested the following implementations:

1) *Static:* This is a (static) community detection code run from scratch on the graph at each time step i. `Louvain` [7] and `SLM` [8] are the two tools we used for this purpose.
2) *Baseline:* This is a community detection code run *incrementally* on the graph at each time step i. "Incremental" here implies that at the start of every time step i, we initialize the state of communities to that of the end of the previous time step $i-1$ (for $i>0$). For this purpose, we implemented our own incremental version of the `Louvain` tool—we call this `dLouvain-base`); and for SLM, we use the already available incremental version DSLM [3]—we call this `dSLM-base`).
3) Δ`-screening`*:* This is a modified baseline version incorporated with our Δ`-screening` step to identify the $\mathcal{R}_t$ set for use within each time step. The corresponding two implementations are referred to as `dLouvain-`Δ`S` and `dSLM-`Δ`S`.

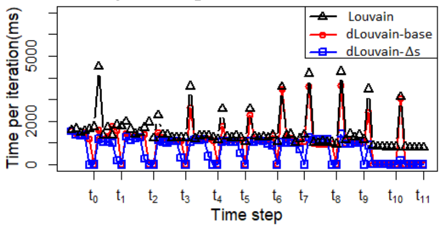
B. Runtime and Quality Evaluation

First, we evaluate the impact of Δ`-screening` technique on performance. Since the main part of the algorithm is the iterative loop that scans every vertex and assigns communities (the `for` loop in Algorithm 1), we measured the average time taken per iteration of a given level, and the results are plotted in Fig. 2. As can be observed, Δ`-screening` achieves a significant reduction in the time spent within each iteration (compared to both static and baseline).

The savings are a result of the reductions in the number of vertices processed per iteration in the Δ`S` version (i.e., $\mathcal{R}_t$ set size). We found the $\mathcal{R}_t$ set-based savings to be more significant for the real-world inputs compared to the synthetic. This is shown in Fig. 3, which shows the percentage of vertices processed per iteration—the $\mathcal{R}_t$ set size fractions range from as little as under 10% in some time steps (for real-world inputs) to as much as 100% in some time steps (for the synthetic inputs). This wide variation in efficacy can be attributed to the nature of input changes. Even though both classes of input graphs (synthetic and real-world) show linear growth rates in size, for the synthetic inputs it is harder to benefit from Δ`-screening` because edges are connected almost randomly from new to existing vertices (as introduced by an edge sampling randomized procedure [23]); whereas, in the real-world networks, changes happen more



(a) Average time per iteration for $5M\_ll$



(b) Average time per iteration for sx-stackoverflow

Fig. 2: The variation of average time per iteration by time step, for two representative inputs: 5M_ll, and sx-stackoverflow. The average is given by the mean time to execute an iteration within each level of a time step. All runs reported are from the `Louvain`-based implementations.

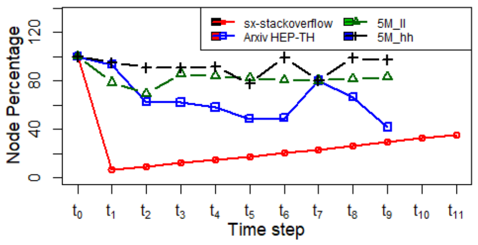
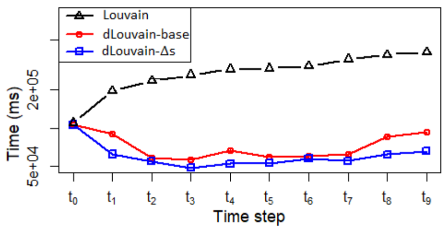


Fig. 3: The fraction of vertices processed at every iteration, given by $\frac{|\mathcal{R}_t|}{|V_t|}$. A lower percentage corresponds to a larger savings in performance.

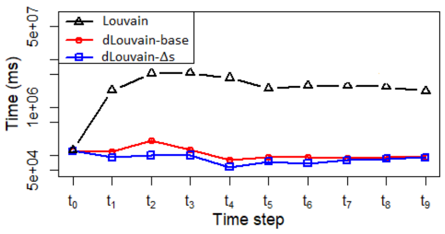
in a localized manner, giving an opportunity to benefit from Δ`-screening`. In fact, even between the two real-world networks, we observed a significant difference in the filtering efficacy of the Δ`-screening` technique. Specifically, with the ArXiv HEP-TH input, the $\mathcal{R}_t$-size fraction varied between 50%-90%; whereas the savings were more significant in the case of sx-stackoverflow (and also showing a linear trend).

Next, we evaluate the total runtime including the time taken to execute all levels. Note that in multi-level codes, the number of iterations per level may vary across the different implementations. Fig. 4 shows the runtime as a function of the time steps, for different combinations of four inputs (5M_ll, 5M_hh, Arxiv HEP-TH, sx-stackoverflow) and two sets of implementations (Louvain and SLM).

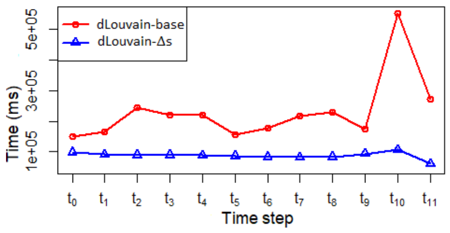
We find that in all cases both baseline and Δ`S` implementations consistently outperform the respective static implementation, providing more than 2 orders of magnitude in some



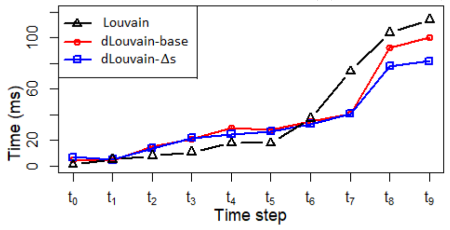
(a) $5M\_ll$ (using Louvain)



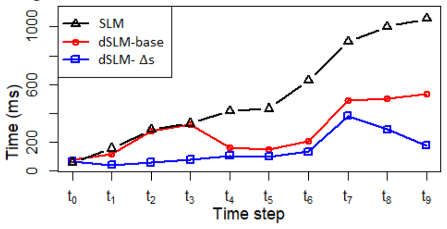
(b) $5M\_hh$ (using Louvain)



(c) sx-stackoverflow (using Louvain)



(d) Arxiv HEP-TH (using Louvain)



(e) Arxiv HEP-TH (using SLM)

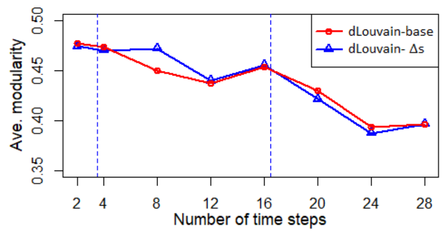
Fig. 4: Runtime as a function of the time steps for various inputs, running either Louvain- or SLM-based implementations. The static Louvain implementation's run and the static/baseline SLM runs on sx-stackoverflow experienced long runtimes and therefore we omit those curves.

cases. Between the baseline and ΔS implementations, the difference varies based on the input. For the synthetic inputs, both versions perform comparably with a slight advantage to the ΔS implementation in some time steps. As discussed earlier, this can be attributed to the random nature of changes induced in the synthetic inputs. For the real-world inputs, ΔS significantly outperforms baseline, yielding over 5X speedup in some time steps. For example, in Fig. 4c we have a 5X speedup in time step t_{10}, since in time step t_9 we have 55158947 edges and this number has increased to 60714297 in time step t_{10} and most of the new edges are intra-community edges which result in less number of nodes for reevaluation and consequently less time.

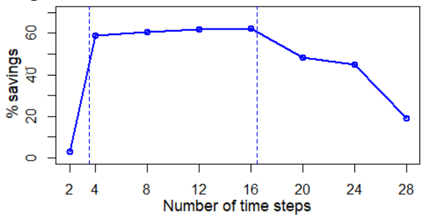
We also evaluated the quality (measured by modularity) achieved by each version. We observed that the S version yielded almost the same modularity as the baseline version despite its heuristic nature, in nearly all cases except one shown in Appendix Fig. B.3 (a) in [22] where ΔS gives a much superior quality compared to the baseline. Results on all inputs tested are shown in Appendix Fig. B.3 in [22].

C. Effect of Varying the Temporal Resolution

In many real-world use-cases, even though the input graph is available as a temporal stream, the appropriate temporal scale to analyze them is not known *a priori*. In fact, this scale is an input property that a domain expert expects to discover through the analysis of dynamic communities. In order to facilitate such a study through dynamic community detection, we study the effect of varying the *temporal resolution*, as defined by the number of time steps used to partition a graph stream, on the output clustering. Using the sx-stackoverflow input, we first generated multiple temporal datasets, each of which representing the input stream divided into a certain number of time steps, ranging from {2, 4, 8, 12, 16, 20, 24, 28} steps. In this scheme, there are multiple nested hierarchies—for instance, the 16-time steps partitioning can be achieved by splitting each of the 8-time steps partitions into two. Subsequently, we ran dLouvain-ΔS on the different temporal datasets.



(a) The change in average modularity achieved for different temporal resolutions (input: sx-stackoverflow)



(b) Percentage savings in total time for Δ-screening, compared to baseline, for different temporal resolutions (input: sx-stackoverflow)

Fig. 5: Plots showing the effect of varying the temporal resolution—as measured by the number of temporal bins (i.e., time steps) used to partition the input graph stream. The resolution of partitioning changes from coarser to finer, from left to right on the x-axis.

Fig. 5 shows the results of our analysis. Fig. 5a shows the change in average modularity as we increase the temporal resolution from coarser to finer (left to right along the x-axis). We observe that the modularity values decline gradually until around 16 time steps, after which the decline starts to accelerate. The decline in modularity suggests that the community-based structure of the underlying network (at different scales) starts to weaken as we increase the temporal resolution. This is to be expected as the temporally binned graphs tend to only become sparser with increasing resolution. Notably, the more rapid slide that starts to appear after the 16 time steps-resolution suggests that the community structure starts to deteriorate after that resolution for *this* input.

Interestingly, this property is better captured by Fig. 5b, which shows the % savings in total runtime achieved by our Δ-`screening` filtering scheme. Intuitively, when the % savings remains approximately steady (highlighted by the plateau region from the resolution of 4 time steps to 16 time steps), it means that the nature of the evolution of the graphs within those resolutions is also relatively consistent. However, a steeper decline (on either end of the plateau) suggests that under those temporal resolution scales the temporally partitioned graphs become either too sparse (right) or too dense (left).

Note that software speed becomes an important enabling factor for conducting these types of experiments, where one needs to run the dynamic community analysis repeatedly under different configurations.

V. Conclusion

Conducting community detection-based analysis on large dynamic networks is a time-consuming problem and there have been many incremental strategies proposed. In this paper, we visit a subproblem in this context—one of identifying vertices that are likely to be impacted by a new batch of changes. We presented a generic technique called Δ-`screening` that examines and selects provably "essential" vertices for evaluation at every time step. We incorporated this technique into two widely-used community detection tools (Louvain and SLM) and demonstrated speedups in performance without compromising on the output quality, for a collection of synthetic and real-world inputs. Future research directions include: i) extension of the Δ-`screening` technique to edge deletions; ii) parallelization on multicore platforms; iii) extensions to other incremental community detection tools; and iv) application and dynamic community characterization on large-scale real-world networks.

Acknowledgment

This research was supported by U.S. National Science Foundation (NSF) grant 1815467.

References

[1] U. Brandes, D. Delling, M. Gaertler, R. Gorke, M. Hoefer, Z. Nikoloski, and D. Wagner, "On modularity clustering," *IEEE Transactions on Knowledge and Data Engineering*, vol. 20, no. 2, pp. 172–188, 2008.

[2] S. Fortunato, "Community detection in graphs," *Physics Reports*, vol. 486, no. 3-5, pp. 75–174, 2010.

[3] R. Aktunc, I. H. Toroslu, M. Ozer, and H. Davulcu, "A dynamic modularity based community detection algorithm for large-scale networks: DSLM," in *Advances in Social Networks Analysis and Mining (ASONAM), 2015 IEEE/ACM International Conference on*. IEEE, 2015, pp. 1177–1183.

[4] D. Greene, D. Doyle, and P. Cunningham, "Tracking the evolution of communities in dynamic social networks," in *2010 International Conference on Advances in Social Networks Analysis and Mining*. IEEE, 2010, pp. 176–183.

[5] A. Zakrzewska and D. A. Bader, "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*. ACM, 2015, pp. 559–564.

[6] M. E. Newman, "Fast algorithm for detecting community structure in networks," *Physical Review E*, vol. 69, no. 6, p. 066133, 2004.

[7] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre, "Fast unfolding of communities in large networks," *Journal of Statistical Mechanics: Theory and Experiment*, vol. 2008, no. 10, p. P10008, 2008.

[8] L. Waltman and N. J. Van Eck, "A smart local moving algorithm for large-scale modularity-based community detection," *The European Physical Journal B*, vol. 86, no. 11, p. 471, 2013.

[9] J. Hopcroft, O. Khan, B. Kulis, and B. Selman, "Tracking evolving communities in large linked networks," *Proceedings of the National Academy of Sciences*, vol. 101, no. suppl 1, pp. 5249–5253, 2004.

[10] P. Maillard, R. Görke, C. Staudt, and D. Wagner, "Modularity-driven clustering of dynamic graphs," in *Experimental Algorithms, 9th International Symposium, SEA 2010*. Springer, 2009, pp. 436–448.

[11] A. Clauset, M. E. Newman, and C. Moore, "Finding community structure in very large networks," *Physical Review E*, vol. 70, no. 6, p. 066111, 2004.

[12] J. Xie, M. Chen, and B. K. Szymanski, "Labelrankt: Incremental community detection in dynamic networks via label propagation," in *Proceedings of the Workshop on Dynamic Networks Management and Mining*. ACM, 2013, pp. 25–32.

[13] M. Seifi and J.-L. Guillaume, "Community cores in evolving networks," in *Proceedings of the 21st International Conference on World Wide Web*. ACM, 2012, pp. 1173–1180.

[14] Y.-R. Lin, Y. Chi, S. Zhu, H. Sundaram, and B. L. Tseng, "Facetnet: a framework for analyzing communities and their evolutions in dynamic networks," in *Proceedings of the 17th International Conference on World Wide Web*. ACM, 2008, pp. 685–694.

[15] D. S. Bassett, M. A. Porter, N. F. Wymbs, S. T. Grafton, J. M. Carlson, and P. J. Mucha, "Robust detection of dynamic community structure in networks," *Chaos: An Interdisciplinary Journal of Nonlinear Science*, vol. 23, no. 1, p. 013142, 2013.

[16] G. Rossetti and R. Cazabet, "Community discovery in dynamic networks: a survey," *ACM Computing Surveys (CSUR)*, vol. 51, no. 2, p. 35, 2018.

[17] H. Lu, M. Halappanavar, and A. Kalyanaraman, "Parallel heuristics for scalable community detection," *Parallel Computing*, vol. 47, pp. 19–37, 2015.

[18] M. Girvan and M. E. Newman, "Community structure in social and biological networks," *Proceedings of the National Academy of Sciences*, vol. 99, no. 12, pp. 7821–7826, 2002.

[19] S. Gregory, "A fast algorithm to find overlapping communities in networks," in *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*. Springer, 2008, pp. 408–423.

[20] P. De Meo, E. Ferrara, G. Fiumara, and A. Provetti, "Generalized louvain method for community detection in large networks," in *2011 11th International Conference on Intelligent Systems Design and Applications*. IEEE, 2011, pp. 88–93.

[21] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM Journal on scientific Computing*, vol. 20, no. 1, pp. 359–392, 1998.

[22] N. Zarayeneh and A. Kalyanaraman, "A fast and efficient incremental approach toward dynamic community detection," 2019. [Online]. Available: https://arxiv.org/abs/1904.08553

[23] E. Kao, V. Gadepally, M. Hurley, M. Jones, J. Kepner, S. Mohindra, P. Monticciolo, A. Reuther, S. Samsi, W. Song *et al.*, "Streaming graph challenge: Stochastic block partition," in *2017 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2017, pp. 1–12.

[24] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data, Jun. 2014.