Delta-Screening: A Fast and Efficient Technique to Update Communities in Dynamic Graphs

Neda Zarayeneh, *Member, IEEE,* and Ananth Kalyanaraman, *Senior Member, IEEE*

Abstract—Detecting communities in time-evolving/dynamic networks is an important operation used in many real-world network science applications. While there have been several proposed strategies for dynamic community detection, such approaches do not necessarily take advantage of the locality of changes. In this paper, we present a new technique called *Delta-Screening* (or simply, Δ-screening) for updating communities in a dynamic graph. The technique assumes that the graph is given as a series of time steps, and outputs a set of communities for each time step. At the start of each time step, the Δ-screening technique examines all changes (edge additions and deletions) and computes a subset of vertices that are likely to be impacted by the change (using the modularity objective). Subsequently, only the identified subsets are processed for community state updates. Our experiments demonstrate that this scheme, despite its ability to prune vertices aggressively, is able to generate significant savings in runtime performance (up to 38$\times$ speedup over static baseline and $5\times$ over dynamic baseline implementations), *without* compromising on the quality. We test on both real-world and synthetic network inputs containing both edge additions and deletions. The Δ-screening technique is generic to be incorporated into any of the existing modularity-optimizing clustering algorithms. We tested using two state-of-the-art clustering implementations, namely, Louvain and SLM. In addition, we also show how to use the Δ-screening approach to delineate appropriate intervals of temporal resolutions at which to analyze a given input network.

Index Terms—Dynamic graphs, community detection, incremental clustering, modularity optimization.

1 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 higher 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 scientist's 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. [1].

Designing efficient algorithms and implementations for community detection has been an area of active research for well over a decade. While the community detection problem is NP-hard [2], there are several efficient heuristics and related software already available for static community detection — e.g., [3], [4], [5], [6], [7], [8], [9]. For a review of works in both sequential and parallel settings, please refer to [1], [10].

However, many real-world networks are *dynamic* in nature, where vertices and edges can be added and/or removed over a period of time. For instance, consider a collaboration network, where over time, new collaborations can be added between pairs of users, or existing collaborative links may cease to exist [11]. Similarly, consider the social interactions over time defining edge additions and removals on a social network [12]. With increasing availability of high-throughput sensing, numerous such network application domains are experiencing an explosion of incrementally or dynamically growing networks [13]. Therefore, analytical solutions that factor in the dynamic evolving nature of these graph inputs are necessary.

Owing to the increasing availability of such networks, the problem of dynamic community detection has become an actively researched topic of late [14], [15], [16], [17], [18]. Section 2 provides an overview of related literature. Broadly, these algorithms can be categorized in two types. The first class are algorithms that identify communities from each time snapshot of the network [19], [20], i.e., without considering the information from previous time steps. These algorithms tend to be better suited for networks that change rapidly between time steps. The other class of algorithms uses information from the community structures of the previous time steps to compute the communities of new time steps [14], [15], [21]. The idea is to reduce time to solution by avoiding re-computation, while also helping with easier tracking of communities across time steps. For this latter class of algorithms, 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 [22], [23]. Approaches that can efficiently exploit such dynamic features have the potential to reduce the computational burden in dynamic community discovery; however, such approaches have not received much attention in the past.

Contributions: In this paper, we propose an algorithmic

• *N. Zarayeneh and A. Kalyanaraman are with the School of Electrical Engineering and Computer Science, Washington State University, Pullman, WA, 99164.*
E-mail: {neda.zarayeneh, ananth}@wsu.edu

Manuscript received December 24, 2020; revised March 1, 2021.

technique called `Delta-Screening` (or Δ`-screening`, for short) that can quickly identify parts of the graph impacted by a new batch of change. The input is a dynamic graph with an arbitrary (T) number of time steps, and with a new batch of edge additions and/or edge deletions at the start of each time step[1]. Using this as input, our technique selects vertices for processing at the start of each time step.

The technique is generic to any community detection method that uses modularity [5] as its clustering objective. Consequently, we present an incremental dynamic community detection framework to detect communities from dynamic networks with multiple time steps. 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 community detection methods 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 [3] and the smart local moving (SLM) method [7]—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. The algorithm shows up to $5\times$ speedup over the dynamic baseline and $38\times$ over the static baseline.
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.

The rest of the paper is organized as follows. We first present a review of relevant literature. Then, we present the Δ`-screening` technique, with provable guarantees. Next, we present a unified clustering framework that uses Δ`-screening` for dynamic community detection. The experimental results section presents a thorough evaluation of our implementations.

2 Related Work

According to Rossetti and Cazabet [22] algorithms to compute dynamic communities over time-evolving graphs can be broadly classified into three types.

Static-based methods: One class of methods try to identify communities based on the *current* state of the network. Typically, these methods follow 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 generation(s) to track evolution. Hopcroft *et al.* [19] 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 by computing community similarities between successive steps. Cuzzocrea *et al.* [20] also follow a similar approach, while trying to recover the evolution of events by matching the communities of two consecutive time steps using normalized mutual information. Alternatively, Asur *et al.* [24] apply a static community detection algorithm to construct a binary matrix that represents node-to-community memberships for each time step. He *et al.* [25] extend the Louvain algorithm [3] to detect community at each time step, while performing community matching with the communities of the previous time step. Seifikar *et al.* [26] also present an extension of the Louvain algorithm, by using a compression-based strategy to make the execution faster. Palla *et al.* [27] present an algorithm based on clique percolation [28] between consecutive time steps. Greene *et al.* [16] present another variant of the two-step strategy, by mainly focusing on a matching-based formulation to map the communities of the latest time step to the communities of previous generations.

In general, these static-based two-step algorithms are not *temporally smoothed*—making tracking of communities harder. Furthermore, these methods incur the additional overhead of running static implementations at each time step, *and* mapping of communities to track the evolution also at every time step—making the approach expensive.

Stability-based approaches: Another class of methods follow a strategy where communities of the current time step are identified based on the communities detected at previous time step(s). These incremental approaches have the advantage of generating more stable communities over time, thereby facilitating an easier tracking the evolution of communities. However, the divergence from the static methods is a potential quality concern for these methods and needs to be evaluated.

Maillard *et al.* [21] present a modularity-based incremental approach extending upon the classical Clauset-Newman-Moore static method [6]. Aktunc *et al.* [15] present the dynamic smart local movement (DSLM) method as an extension of its static predecessor [7]. Xie *et al.* [29] present an incremental method based on label propagation which is a fast heuristic. Saifi and Guillaume [30] provide a way to track and update community "cores" across time steps. Zakrzewska and Bader [17] 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.* [31] 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. While the modularity metric has known to have limitations such as the resolution limit [32], [33], it is still the most widely used objective in practice (as represented in the aforementioned methods).

Agarwal *et al.* [18] propose an incremental approach to detect communities based on permanence [34] as its clustering objective. Zeng *et al.* [35] present a consensus-based approach to finding communities across time steps. Their approach uses a genetic algorithm to solve for a multi-objective problem involving normalized mutual information (NMI) and modularity. Another multi-objective algorithm

1. Note that *vertex* additions and deletions can be implicitly encoded as part of edge additions and deletions as well. We, therefore, only refer to *edge* related events in the remainder of this paper.

was introduced in [36]. In a more recent study, Zhao *et al.* [37] present an incremental algorithm to cases where bulk changes happen via subgraph additions or deletions. Wu *et al.* [38] present an incremental approach to detect overlapping communities using a weighted network formulation. DynaMo [39] is an incremental method that updates the community structure at each time step according to the changes in the graph. In other words, this approach also avoids recomputing communities from scratch based on a set of local rules. Our results section includes a comparison with `DynaMo`.

Cross-time step approaches: Another class of methods work across time steps and identify communities at any given time step based on the knowledge of the entire graph across all time steps. Tantipathananandh *et al.* [40] present an algorithm to identify temporally persistent groups across multiple time steps, and use a combinatorial optimization procedure to identify community structures. Their algorithm assumes that different time steps could contain information pertaining to different parts of the graph. Bassett *et al.* [41] propose and evaluate the choice of alternative null hypothesis models in modularity-based dynamic community detection methods. Matias et al. [42], [43] introduce a Poisson Process Stochastic Block Model (PPSBM) to detect communities in temporal networks. Xu and Hero [44] defined a dynamic stochastic block model in which both affiliations and densities can change. In Viard *et al.* [45] and Himmel *et al.* [46], the authors consider the stream of the links and look for persistent minimal size cliques. Bu *et al.* [47] presents a multi-objective optimization framework for community detection.

The advantage of these cross-time step algorithms is that they are better at detecting temporally persistent communities. However these algorithms require prior knowledge of the global graph. In addition, several of these methods compare the graphs and/or community states from multiple time steps, incurring a higher runtime complexity.

The technique of Δ`-screening` proposed in this paper is orthogonal to all the above efforts, in that it is aimed at helping incremental methods (which fall into the category of stability-based approaches) 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. Consequently, the Δ`-screening` technique can be used to accelerate the computation in any of the incremental approaches. We demonstrate in this paper, the utility of this technique using the modularity-optimization function. A preliminary version of the method proposed in this paper appeared in [48]. That preliminary version of the method only addressed edge additions. In contrast, the work presented in this paper works for both addition and deletion events.

3 METHOD

3.1 Basic Notation and Terminology

A *dynamic graph* $G(V, E)$ can be represented as a sequence of graphs $G_1(V_1, E_1)$, $G_2(V_2, E_2)$,... $G_T(V_T, E_T)$, 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 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 neighbors of a vertex i as $\Gamma(i) = \{j \mid (i, j) \in E_t\}$. 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 both incrementally growing and incrementally shrinking dynamic graphs—i.e., edges (and vertices) can be added and deleted at each time step. We denote the newly added/deleted set of edges at any time step t as Δ_t, which is given by $\Delta_{t+} \cup \Delta_{t-}$, where $\Delta_{t+} = E_t \setminus E_{t-1}$ and $\Delta_{t-} = E_{t-1} \setminus E_t$.

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 [49]. 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 [5]:

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

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 [3]. We denote this modularity gain by $\Delta Q_{i\to C(j)}$. More specifically:

$$\begin{aligned}\Delta Q_{i\to C(j)} =& \frac{e_{i\to C(j)\cup\{i\}} - e_{i\to C(i)\setminus\{i\}}}{2m_t} \\ &+ \frac{d_\omega(i).a_{C(i)\setminus\{i\}} - d_\omega(i).a_{C(j)}}{(2m_t)^2}\end{aligned} \tag{2}$$

Note that if a vertex i is to migrate out of its current community, it can realistically move only to a neighboring community associated with one of its neighbors (as otherwise the gain cannot be positive). Consequently, computing a target community that yields the maximum modularity gain for a given vertex, can be achieved in time proportional to $d(i)$. This result is used in many algorithms [3], [4], [7], [49].

3.2 Problem Statement

We define the dynamic community detection problem as follows.

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 [2], it immediately follows that the dynamic version is also intractable.

For the static version, there exist several efficient heuristics have been developed (as surveyed in [1]). These approaches can be broadly classified into three categories: divisive approaches [50], [51], agglomerative approaches [5], [6], and multi-level approaches [4], [7], [52]. 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 a 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 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 [3], 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 $\Pi : V \to \mathbb{Z}$
Initialize Π by setting $\mathcal{C}(v) \leftarrow \{v\},\ \forall v \in V$
repeat
  repeat
    for *each* $v \in V$ **do**
      Compute a local (greedy) function $g(v, \mathcal{C}(v))$
      $C(v) \leftarrow$ Update community assignment for v using the results from $g(v, \mathcal{C}(v))$
    end
    Compute a global quality function Q for Π
  until *Convergence based on* Q
  Review communities of Π (optional step)
  $G(V, E) \leftarrow$ Coarsen by performing graph compaction for next level
until *Convergence based on* Q
return Π

3.3 A Naive Algorithm for Dynamic Community Detection

A simple approach to implement dynamic community detection is to simply apply the static version of the algorithm (Algorithm 1) on the graph at every time step. However, such an approach could suffer from lack of stability in the output communities. 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 be wasteful in computation. Consider an edge $(i, j) \in \Delta_t$ that has been newly introduced at time step t; it is reasonable to expect that only those vertices in the "vicinity" of this newly added edge to be impacted by this addition. However, the naive strategy is not suited to 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. The Δ-`screening` scheme described in this paper is aimed at providing an alternative to this naive approach by overcoming the above set of limitations.

3.4 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 $\mathcal{R}_t$ by the Δ-`screening` scheme 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 $t-1$). The main objective 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).

Since at each time step t we could have both deletion and addition of edges, at first we present our algorithm in two parts, one for handling edge deletions and the other for handling edge additions. Note that by capturing edge additions and deletions, we also implicitly capture vertex additions and deletions. Finally, we also present a unified algorithm that accounts for time steps where both additions and deletions occur. In the following sections, we present our algorithms for edge additions,edge deletions, and finally a combined framework to handle both events.

3.4.1 The Δ-`screening` Approach for Edge Additions

Let Δ_{t+} denote the set of edges added at time step t, and $\mathcal{R}_{t+}$ denote the set of nodes for reevaluation. Note that the objective here is to compute $\mathcal{R}_{t+}$. For simplicity, we will assume that these edges did *not* exist with the same edge

Algorithm 2: Δ−`screening` for edge additions at time step t

Input: G_t, Δ_{t+}
Output: $\mathcal{R}_{t+}$: a subset of vertices for reevaluation
1 $\mathcal{R}_{t+} \leftarrow \emptyset$
2 Sort edges in Δ_{t+} based on source vertices
3 **for** *each* $i \in S_{\Delta_t}$ **do**
4 Let $j_* \leftarrow \arg\max_{j \in T_{\Delta_t}(i)} \{\Delta Q_{i \to \mathcal{C}_{t-1}(j)}\}$
5 Let $gain_1 \leftarrow \Delta Q_{i \to \mathcal{C}_{t-1}(j_*)}$
6 Let $gain_2 \leftarrow \Delta Q_{j_* \to \mathcal{C}_{t-1}(i)}$
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** $\mathcal{R}_{t+}$

weight, at the previous time step $t-1$[2] We also 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_t} denote the set of all source vertices in Δ_{t+}, and let $T_{\Delta_t}(i)$ denote the set of all sinks (or "targets") for a given source $i \in S_{\Delta_t}$.

Algorithm 2 shows the algorithm for Δ−`screening` for edge additions. The algorithm can be described as follows. We initialize $\mathcal{R}_{t+}$ to $\emptyset$. Subsequently, we examine all edges of Δ_{t+} in the sorted order of its source vertices, breaking ties arbitrarily. Sorting helps in two ways: it helps us to consider all the new edges incident on a given source vertex collectively and identify an edge that maximizes the net modularity gain for the source (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. Furthermore, this sorted treatment helps reduce overhead by helping to update $\mathcal{R}_{t+}$ in a localized manner (relative to the source vertices) and avoiding potential duplication in the computations associated with the source vertex.

Once sorted, we read the list of edges added for each source vertex, identify a neighbor (j_*) that maximizes the modularity gain (line 4 of Algorithm 2), 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 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 $\mathcal{R}_{t+}$. Note that the actual decision to migrate itself is deferred until the stage of execution of the iterative clustering algorithm. In other words, the Δ−`screening` procedure does *not* modify the state of communities, but it sets the stage for which vertices (in which communities) to be visited during the main iterative procedure.

The main part of Algorithm 2 is in 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 (i.e., $\Gamma(i)$), 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 left out of $\mathcal{R}_{t+}$, 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.

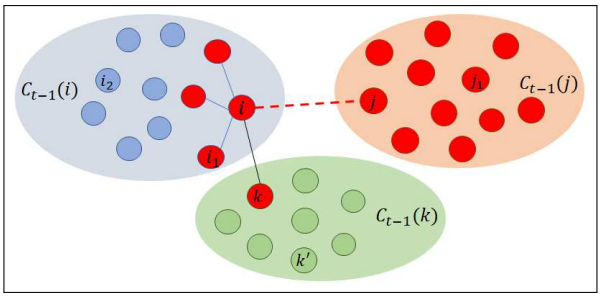


Fig. 1. Figure showing the impact of a newly added edge (i, j), shown in red dotted line. Representative cases of candidate vertices for potential inclusion in $\mathcal{R}_t$ are shown highlighted as labelled vertices. Note that we follow the naming convention of denoting the community containing a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

Provable Properties for Δ−`screening`: Edge additions

In the following 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.

First, note that there are two simple cases. If one of the two endpoints of an added edge is new, then the new node will simply join the community of the other endpoint. Also, if both endpoints are new, then they will be in a new community with these two vertices. In what follows, we prove certain properties for the more non-trivial cases.

Figure 1 shows the various 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 1. *Let i' be a neighbor of a vertex i that has a newly added edge to j_*. Then, the community assignment for i' at the end of time step t, i.e., $C_t(i')$, could be potentially different from its previous community $C_{t-1}(i')$.*

Intuitively, a change in i's community state could influence any of its neighboring vertices. For a formal proof, see Supplementary section.

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

2. If not, then we can simply skip over such edges during processing.

Proof. We have already considered the case where $i' \in \Gamma(i)$ (as part of Lemma 1 in the Supplementary section). Therefore we only need to consider the case where $i' \notin \Gamma(i)$. This is represented by vertex label i_2 in Figure 1. Since i_2 is already in $C_{t-1}(i)$ and i_2 does not share an edge with i, a departure of i from $C_{t-1}(i)$ can only positively reinforce i_2's membership in $C_{t-1}(i)$. More formally, this can be shown by comparing the old vs. new modularity gains, $\Delta Q_{i_2 \to C(j_*)}$, resulting from moving i to $C_{t-1}(j_*)$:

$$\begin{aligned}\Delta Q^{\text{new}}_{i_2 \to C_{t-1}(j_*)} =& \frac{e_{i_2 \to C_{t-1}(j_*)} - e_{i_2 \to C_{t-1}(i)\setminus\{i_2\}}}{2m_t} \\ &+ \frac{d_\omega(i_2).(a_{C_{t-1}(i)\setminus\{i_2\}} - d_\omega(i))}{(2m_t)^2} \\ &- \frac{d_\omega(i_2).(a_{C_{t-1}(j_*)} + d_\omega(i))}{(2m_t)^2} \\ =& \Delta Q^{\text{old}}_{i_2 \to C_{t-1}(j_*)} - \frac{2d_\omega(i_2).d_\omega(i)}{(2m_t)^2}\end{aligned}$$

Since the new modularity gain is less than the old, vertex i_2 will have no incentive to change community status, and therefore can be excluded from $\mathcal{R}_t$. □

Note that the above proof only analyzes the direct impact that vertex i's migration from $C_{t-1}(i)$ will have on non-neighbor vertices of $C_{t-1}(i)$. However, there could still be an indirect impact (cascading from Lemma 1)—e.g., the migration of a vertex $i_1 \in \Gamma(i)$ may trigger the migration of a vertex $i_2 \notin \Gamma(i)$ but in $\Gamma(i_1)$, and so on. However, intuitively, as this distance from the locus of change increases, the likelihood of its impact can be also expected to decay rapidly in practice (as observed in our experiments).

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

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

Proof. Intuitively, a new vertex (i) arrives in community $C_{t-1}(j_*)$, the degree of such a vertex contributes to the negative term of modularity, thereby possibly influencing the other vertices' decision to remain in that community. A formal proof is provided in the Supplementary section. □

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 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 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 $C_{t-1}(j_*)$, create an incentive for k to move to $C_{t-1}(j_*)$. This can be shown more formally using modularity gains as follows.

(A) This case is similar to the subcase (B) identified in the proof of Lemma 1. The same argument holds.

(B) Any node k connected to $C_{t-1}(j_*)$ which is located in other communities have the following modularity gain after moving i to $C_{t-1}(j_*)$:

$$\begin{aligned}\Delta Q^{\text{new}}_{k \to C_{t-1}(j_*)} \approx& \frac{e_{k \to C_{t-1}(j_*)} - e_{k \to C_{t-1}(k)\setminus\{k\}}}{2m_t} \\ &+ \frac{d_\omega(k).a_{C_{t-1}(k)\setminus\{k\}}}{(2m_t)^2} \\ &- \frac{d_\omega(k).(a_{C_{t-1}(j_*)} + d_\omega(i))}{(2m_t)^2} \\ \approx& \Delta Q^{\text{old}}_{k \to C_{t-1}(j_*)} - \frac{d_\omega(k).d_\omega(i)}{(2m_t)^2}\end{aligned}$$

This implies that the modularity gain will decrease if k were to migrate to $C_{t-1}(j_*)$, and therefore k will choose to remain in its current community.

(C) If a vertex is not connected to either $C_{t-1}(i)$ or $C_{t-1}(j_*)$, such as the example k' in Figure 1, then such a vertex has no incentive to move to either of the two communities.

□

Based on the above lemmas and arguments, the following theorem follows.

Theorem 5. *If an inter-community edge (i, j) is added, then the set of vertices that could be possibly affected by this change is given by: $\mathcal{R}_{t+} = \Gamma(i) \cup (C_{t-1}(j) \setminus \Gamma(i))$.*

3.4.2 The Δ-screening Method for Edge Deletion

Similar to the Δ-screening approach for edge addition introduced in the previous section, we identify a subset of vertices for reevaluation, at each time step with edge removals.

Algorithm 3 presents the Δ-screening scheme for handling edge removals. We assume that Δ_{t-} stores the set of all deleted edges of a given time step t, and that for each deleted edge (i, j), there will be two entries in Δ_{t-}: (i, j) and (j, i). We use the same definitions as in the previous section, to denote the sets of source and sink vertices: S_{Δ_t} and $T_{\Delta_t}(i)$, respectively, in this deletion batch.

Algorithm 3: Δ-screening for edge deletion at time step t

Input: G_t, Δ_{t-}
Output: $\mathcal{R}_{t-}$: Subset of vertices for reevaluation

$\mathcal{R}_{t-} \leftarrow \emptyset$
Sort edges in Δ_{t-} based on the source vertices
for *each source* $i \in S_{\Delta_t}$ **do**
  flag = False
  for *each sink* $j \in T_{\Delta_t}(i)$ **do**
    if $\mathcal{C}_{t-1}(i) = \mathcal{C}_{t-1}(j)$ **then**
      flag = True
    end
  end
  if *flag = True* **then**
    $\mathcal{R}_{t-} = \mathcal{R}_{t-} \cup \mathcal{C}_{t-1}(i) \cup \Gamma(i)$
  end
end
return $\mathcal{R}_{t-}$

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TNSE.2021.3067665, IEEE Transactions on Network Science and Engineering

In Algorithm 3, we have G_t and Δ_{t-} as inputs, and $\mathcal{R}_{t-}$ as output. Here, $\mathcal{R}_{t-}$ represents the nodes that need to be reevaluated.

At first, we observe that if the edge being removed is between two different communities, then such a deletion could *not* possibly result in a change of the existing community states (as the strength of the tie between the two communities only becomes weaker). Therefore, we discard such inter-community edge deletions from further consideration in our Δ-`screening` scheme. Lines 4 through 9 of Algorithm 3 implement this inter-community check for all deleted edges incident on a source vertex. As the edge list in Δ_{t-} is kept sorted by source vertices (line 2), it is easy to check and discard such source vertices that have *all* of their incident deleted edges between two communities.

We therefore only consider source vertices i which have at least one *intra-community* deleted edge incident on them—i.e., edge (i, j) where $C_{t-1}(i) = C_{t-1}(j)$. For such source vertices, the update scheme to $\mathcal{R}_{t-}$ is shown in line 11 of Algorithm 3. Our method adds all vertices in the community containing i along with the neighbors of i, to $\mathcal{R}_{t-}$. Through a combination of lemmas (stated below), we show that this update scheme sufficiently captures the effects of the new batch of deletions.

Provable Properties for Δ-`screening`: Edge deletions

In the following lemmas we analyze the potential impact of edge removals. Let (i, j) be an intra-community edge being removed at time step t from community $C_{t-1}(i)$ (equivalently, $C_{t-1}(j)$). We consider the two following possibilities:

- **Case A:** the deletion event could result in the migration of vertices i or j (or both) to a (different) neighboring community;
- **Case B:** the deletion event could result in the breaking of community $C_{t-1}(i)$ into two or more smaller communities.

Our main strategy is to populate $\mathcal{R}_{t-}$ to include those vertices which are likely to alter their community states as a result of this deletion event.

First, it should be clear that the deletion event has a direct impact on the vertices i and j—i.e., it is possible that either of i's or j's strength of connection to $C_{t-1}(i)$ could weaken as a result of this deletion, and therefore either of these two vertices could possibly choose to leave $C_{t-1}(i)$ at time step t. Therefore, we simply include i and j as part of our $\mathcal{R}_{t-}$.

Next, we show that *none* of the vertices in $C_{t-1}(i)$ other than i and j, will have any incentive to migrate *out* of $C_{t-1}(i)$ to any of the neighboring communities[3].

Lemma 6. *Let $(i, j) \in \Delta_{t-}$ be an intra-community edge to be deleted. Consider a vertex $i' \in C_{t-1}(i) \setminus \{i, j\}$. Then, the modularity gain achieved by moving vertex i' out of $C_{t-1}(i)$ will be zero or negative.*

Proof. This case is represented by vertex labels i_1 (immediate neighbor of i) and i_2 in Figure 2. Here, removing the edge (i, j) has no impact on the strength of vertex i_1's or

3. Note that this does not exclude the possibility of those vertices choosing to separate into smaller contained communities within $C_{t-1}(i)$ at the end of time step t (case B above).

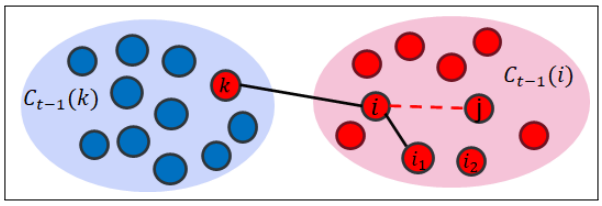


Fig. 2. Figure showing the impact of a deleted edge (i, j), shown in red dotted line. Representative cases of candidate vertices for potential inclusion in $\mathcal{R}_t$ are shown highlighted as labelled vertices. Note that we follow the naming convention of denoting the community containing a vertex i by $C(i)$. Also, note that not all edges are shown.

i_2's connection to community $C_{t-1}(i)$; i.e., the contribution of these vertices to the positive term in Eqn. 1 will remain the same. On the other hand, the negative term in Eqn. 1 for the community $C_{t-1}(i)$ will only improve with the removal of edge (i, j), owing to reduced vertex degree. Therefore, neither vertex i_1 nor vertex i_2 will have any incentive to leave $C_{t-1}(i)$ during time step t. □

Even though the above lemma shows that none of the vertices in $C_{t-1}(i)$ other than i and j, have an incentive to move out of $C_{t-1}(i)$, we note that it is still possible for those vertices to split to form smaller, sub-communities by splitting $C_{t-1}(i)$—i.e., Case B. Intuitively, this is because of the weakening of the strength of the intra-community edges within $C_{t-1}(i)$, resulting from the deleted edge. Therefore, to consider this possibility, our Δ-`screening` method simply adds all the vertices in $C_{t-1}(i)$ for re-evaluation (as shown in line 11 of Algorithm 3).

Next, we consider the impact of the edge deletion (i, j) on neighbors of i (alternatively, j) which are *outside* $C_{t-1}(i)$. The lemma is shown for only neighbors of i, and the same argument holds for neighbors of j as well.

Lemma 7. *Let $(i, j) \in \Delta_{t-}$ be an intra-community edge to be deleted. Consider a vertex $k \in \Gamma(i) \setminus C_{t-1}(i)$. Then, it is possible for vertex k to have an incentive to change its community at time step t.*

Algorithm 4: Dynamic community detection using Δ-`screening`

Input: $G_1, \{\Delta_{1+}, \Delta_{1-}, ..., \Delta_{T+}, \Delta_{T-}\}$
Output: $\{\mathcal{C}_1, \mathcal{C}_2, ..., \mathcal{C}_T\}$
1 Let $\mathcal{C}_1 \leftarrow Static(G_1)$ denote the initial step output
2 **for** *each* $t \in \{2, 3, ..., T\}$ **do**
3 Initialize $\mathcal{C}_t \leftarrow \mathcal{C}_{t-1}$
4 //handle deletions
5 $G_t \leftarrow$ Update G_{t-1} using Δ_{t-}
6 Compute $\mathcal{R}_{t-} \leftarrow \Delta$-`screening`$(G_t, \Delta_{t-})$
7 Update $\mathcal{C}_t \leftarrow Static(G_t, \mathcal{R}_{t-})$—i.e., run static clustering only for vertices in $\mathcal{R}_{t-}$
8 //handle additions
9 $G_t \leftarrow$ Update G_t using Δ_{t+}
10 Compute $\mathcal{R}_{t+} \leftarrow \Delta$-`screening`$(G_t, \Delta_{t+})$
11 Update $\mathcal{C}_t \leftarrow Static(G_t, \mathcal{R}_{t+})$—i.e., run static clustering only for vertices in $\mathcal{R}_{t+}$
12 **end**
13 **return** $\{\mathcal{C}_1, \mathcal{C}_2, ..., \mathcal{C}_T\}$

TABLE 1
Input network statistics. All the networks used have multiple time steps (shown in the rightmost column), and the cumulative number of edges represents peak number of edges across all those time steps. Networks whose labels end in '+' correspond to growing networks (i.e., edges being added each time step), and those that end in '-' correspond to shrinking networks. The networks whose labels are denoted by '+/-' have a combination of edge additions and deletions at each time step.

Input	Input graph	No. vertices	No. edges (peak)	No. timesteps
Synthetic	50kll+, 50kll-	50,000	2,362,448	10
	50kll+/-	50,000	2,256,197	10
	50khh+, 50khh-	50,000	2,367,024	10
	50khh+/-	50,000	2,260,550	10
	5Mll+, 5Mll-	5,000,000	213,656,492	10
	5Mll+/-	5,000,000	202,974,502	10
	5Mhh+, 5Mhh-	5,000,000	213,771,700	10
	5Mhh+/-	5,000,000	203,083,098	10
Real-world	Arxiv HEP-TH+, Arxiv HEP-TH-	27,770	352,807	10
	Arxiv HEP-TH+/-	27,770	352,807	10
	Enron+	36,692	183,831	45
	sx-stackoverflow+, sx-stackoverflow-	2,601,977	63,497,050	2-28
	sx-stackoverflow+/-	2,601,977	61,032,385	2-28

Proof. This case is represented by vertex label k in Figure 2. A formal proof is provided in the Supplementary section. □

Based on the above lemmas and arguments, the following theorem follows.

Theorem 8. *If an intra-community edge (i, j) is deleted, then the set of vertices that could be possibly affected by this change is given by: $\mathcal{R}_{t-} = C_{t-1}(i) \cup \Gamma(i) \cup \Gamma(j)$.*

3.5 A General Framework for Dynamic Community Detection using Δ-screening

Here, we present our unified dynamic community detection framework, based on the Δ-screening scheme, capable of handling both edge additions and edge deletions. Algorithm 4 summarizes the main steps of the algorithm over all T time steps. Our algorithm uses Δ-screening to prune the vertex space for processing, before handing over the actual clustering task to a static clustering code of choice, albeit only on the vertex subset selected in the $\mathcal{R}_t$. We note here that our algorithm is generic enough to be applied to any modularity-based static clustering scheme (denoted $Static()$ in Algorithm 4). In this work, we used the Louvain method [3] and smart local moving (SLM) method [7], as choices for $Static$ (described in the Experimental Evaluation section).

At the start of every time step t, we assume that the batch of changes Δ_t arrives simply as an unordered collection of edge additions and edge deletions (i.e., $\Delta_t = \Delta_{t+} \cup \Delta_{t-}$). In our algorithm, we first sort out the additions from the deletions, and then first process the edge deletions prior to edge additions—as shown in Algorithm 4. The motivating rationale behind this order of computation is a combination of performance and quality—i.e., by first removing edges, we shrink the graph and update the communities, before growing the graph again using the added edges and updating the communities. This strategy can be expected to lower processing times for each batch than doing it in an arbitrary order or handling additions first. Secondly, from a qualitative standpoint, we note that the strict order used in processing these edge additions and deletions could potentially impact output clustering. Fig. 3 shows the runtime comparison of first processing the edge deletions and then edge addition and conversely, if the edge addition gets processed first. The acquired clustering quality is 0.658227 and 0.652412, respectively, we have also checked the community overlap percentage for these two approaches and obtained around 95% overlap. However, as shown in the lemmas for our Δ-screening scheme, the effects of deletions are expected to be within communities (splits) whereas the effects of additions are expected to be across. Given this complementary nature of their effects, our algorithm chooses to optimize performance in practice by considering deletions first. It is worth noting that the challenge imposed by the order of processing the edge addition/deletion events also applies to the static clustering, and is not unique to the dynamic setting.

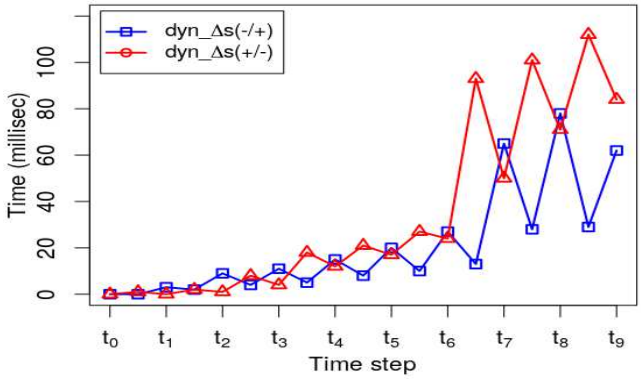


Fig. 3. Runtime comparison between processing edge deletion following by edge addition vs. processing edge addition following by edge deletion.

Note that there is also a simpler version that can be implemented for both these methods—by following all steps outlined in our 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 implementations as dyn-base[4].

3.6 Algorithmic Complexity of Δ-screening

The Δ-screening procedures described for edge additions and edge deletions have a worst-case linear time complexity in the size of the graph at time step t, i.e., $\mathcal{O}(n_t + M_t)$. The worst-case represents a scenario where the changes are distributed across the entire graph, possibly impacting almost all of the previous communities. However, in practice, both procedures are expected to take significantly less time as the computations are localized to where the changes (Δ_t) could have an impact.

Note that the time taken to compute the new clustering after the Δ-screening step of that time step (denoted by Step 6 of Algorithm 4) will depend on the computational cost of the underlying clustering algorithm. For instance, in the case of the Louvain implementation [3], clustering involves multiple iterations until a modularity-based convergence, with each iteration performing a linear scan of vertices in the worst-case. With the application of Δ-screening, we aim to significantly reduce this number of vertices visited.

3.7 Software Availability

We have implemented the Δ-screening technique and integrated it as part of the Louvain method. This open source implementation is available at https://bitbucket.org/nzarayeneh/dynamic_community_detection/src/master/. The software package is implemented in C++ and can be used for modularity-based community detection on dynamic networks.

4 Experimental Evaluation

4.1 Experimental Setup

All our experiments were conducted on a Linux compute node with an 2.13GHz AMD Ryzen thread ripper 1920x processor with 64 GB RAM.

Input data: For all our experiments, we used a combination of synthetic and realworld networks. Table 1 summarizes the main input statistics for the inputs used. Figure S1 (in the Supplementary section) shows the numbers of edge addition/deletions for the individual time steps.

As synthetic inputs, we used a collection of streaming networks available on the MIT Graph Challenge 2018 website [53]. 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 (or shrinks) linearly with each time step (see figure S1. 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 data sets as representatives of two size categories.

4. We note that our dyn-base for SLM method implementation is in effect same as [15].

As real-world inputs, we used two networks downloaded from SNAP database [54]:

1) **Arxiv HEP-TH:** This is a citation graph for 27,770 papers (vertices) with 352,807 edges (cross-citations). Even though the edges are directed, for the purpose of analysis in this paper we treated them as undirected. The dataset covers papers published between 1993 and 2002. Consequently, we partitioned this period into 10 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). Interactions could be one of many types—e.g., a user answers another user's query, a user commented on another user's answers, etc. We treated all these interactions equivalently (as edges), and for the purpose of our analysis we used only the first instance of a user-user interaction as an edge.
3) **Enron email:** This communication network covers all the email communication within a dataset of around half million emails. This data was originally made public, and posted to the web, by the Federal Energy Regulatory Commission during its investigation. Nodes of the network are email addresses, and edges represent (undirected) email exchanges between pairs of users. We partitioned this data set into 45 time steps based on the dates of the emails.

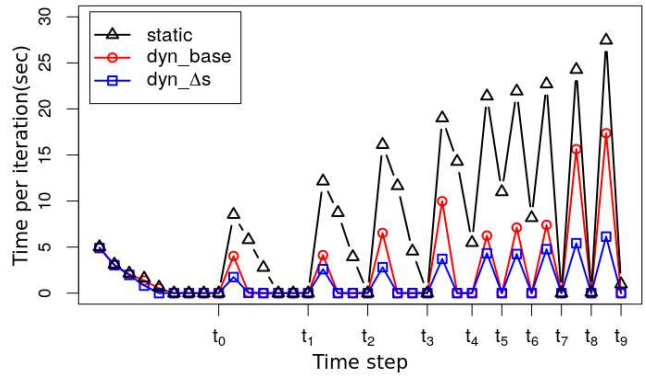
For testing purposes, we constructed three types of networks using the above network inputs, as shown in Table 1:

- Networks whose labels end with '+' are incrementally growing networks—i.e., at each time step, edges are only added. Note that these '+' networks are the same as the original input.
- Networks whose labels end with '-' are incrementally shrinking networks—i.e., at each time step, edges are only deleted. We constructed this sequence by simply reversing the time steps from the '+' networks.
- Finally, networks whose labels end with '+/-' contain a combination of additions and deletions at each time step. These networks were constructed by randomly selected one edge (from the entire network) for deletion, for every ten edges added in the given data set, at each time step.

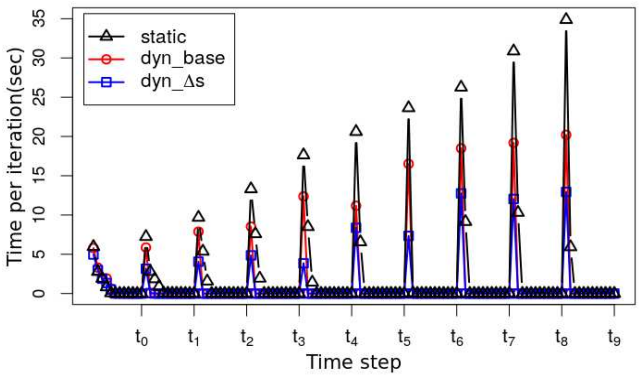
Implementations tested: In our experiments, we tested the following implementations:

1) *Static:* This is a (static) community detection code run from scratch, on the entire graph of a time step i (after incorporation of all edge additions and removals). Louvain [3] and SLM [7] 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 dyn-base; and for SLM, we use the already available incremental version, which is DSLM [15].

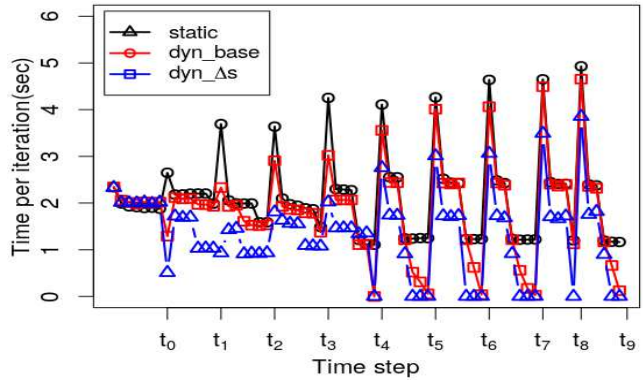
This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TNSE.2021.3067665, IEEE Transactions on Network Science and Engineering



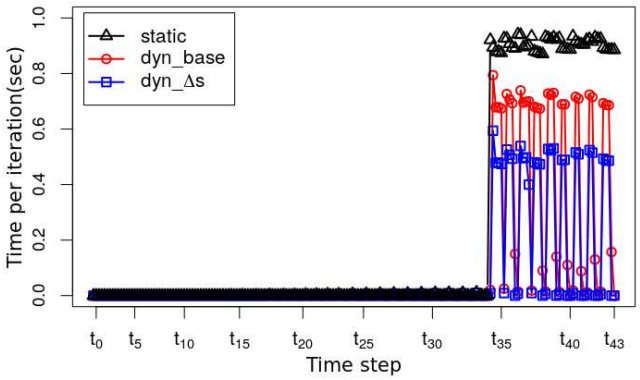
(a) Average time per iteration for 5Mll+/- using Louvain algorithm



(b) Average time per iteration for 5Mhh+/- using Louvain algorithm



(c) Average time per iteration for sx-stackoverflow+/- using Louvain algorithm



(d) Average time per iteration for Enron+/- using Louvain algorithm

Fig. 4. The average runtime per iteration, for all levels across all time steps, for two representative inputs: 5Mll+/-, 5Mhh+/-, and sx-stackoverflow+/-. The average is given by the mean time to execute an iteration within each level of a time step. Note that the variance of runtimes for the iterations of a given level is expected to be small, since the number of vertices processed per iteration remains the same through the level. All runs reported are from the `Louvain`-based implementations.

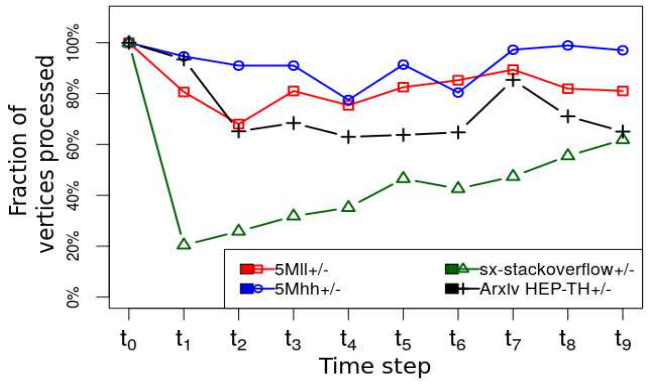


Fig. 5. The fraction of vertices processed at every iteration, given by $\frac{|\mathcal{R}_t|}{|V_t|}$. A lower percentage corresponds to a larger savings in performance.

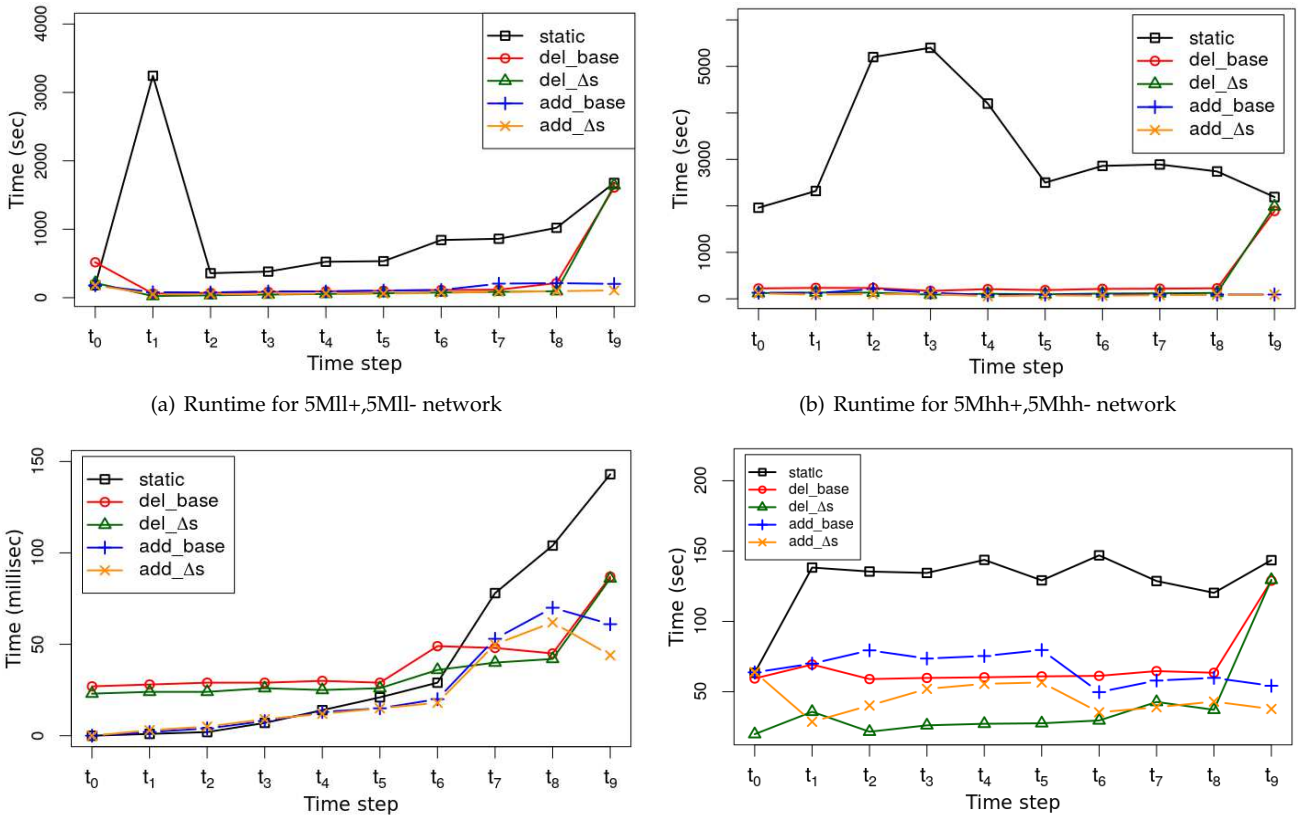
3) Δs: This is a modified baseline version incorporated with our Δ−`screening` step to identify the $\mathcal{R}_t$ set for use within each time step. We refer to the resulting two implementations as `dyn`−Δs (Louvain) and `dyn`−Δs (SLM).

4.2 Runtime and Quality Evaluation of Δ−`screening`

Table 2 shows a summary of all the runtime and quality results, comparing the Δ−`screening` implementation (Louvain) against the static scheme, for all inputs tested. As can be observed, the speedups obtained by Δ−`screening` ranges from $1.13\times$ to $38.62\times$ over the static Louvain scheme and from $1.14\times$ to $5.55\times$ over the dynamic Louvain baseline. The smaller speedups belong to those networks with smaller size or those synthetic networks where the location of the changes were randomly distributed. In contrast, we see higher speedups for larger real-world networks, and for those synthetic networks where there is a stronger community structure ('hh' label) and changes tend to happen with more locality. These results are achieved with little to no impact on the quality of the results (shown in the modularity column). To further understand these results, we present a thorough evaluation of the runtime and quality results below.

Runtime Evaluation: First, we evaluate the impact of Δ−`screening` technique on the clustering algorithm's performance and its filtering efficacy. Fig. 4 shows the runtimes for all clustering iterations within all the time steps, for the different schemes on different inputs tested.

As can be observed, the `dyn`−Δs that uses Δ−`screening`, achieves the best performance, with



(a) Runtime for 5Mll+,5Mll- network

(b) Runtime for 5Mhh+,5Mhh- network

(c) Runtime for Arxiv HEP-TH+,Arxiv HEP-TH- network

(d) Runtime for sx-stackoverflow+,sx-stackoverfl- network

Fig. 6. Parts (a), (b), (c), and (d) show the runtime for each time step, for the different growing inputs ('+') and shrinking inputs ('-').

the least runtime per iteration compared to both the static and baseline schemes. In fact, for some iterations, dyn−Δs is more than $5.74\times$ faster the static scheme, and $3.4\times$ faster than the baseline implementation.

The runtime savings are a direct result of the reductions in the number of vertices that are selected for processing by Δ−screening (i.e., $\mathcal{R}_t$ set size). Fig. 5 shows the $\mathcal{R}_t$ set sizes (as a percentage of the respective total number of vertices), selected for processing at each time step for the different inputs. Note that the location of the changes to the graph has a direct bearing on the $\mathcal{R}_t$ selected, and this accounts for the fluctuation in the $\mathcal{R}_t$ sizes. For instance, for the input sx-stackoverflow+/- the size of the changes increases linearly, but as we can see from Fig. 5. for time step t_2 the percentage of vertices eligible for re-evaluation is low (around 25%) while for t_8 it is higher (around 60%). Upon further examination, we found that this variability is because in t_2 the fraction of edge additions falling *within* communities is $\sim 72\%$ and the fraction of edge removals happening *between* two communities is $\sim 67\%$. In comparison, these two corresponding percentages for time step t_8 are $\sim 56\%$ and $\sim 49\%$ respectively. Therefore, for t_2, most of the changes do not add new work and hence generate more savings, compared to time step t_8.

On average, for the smaller inputs, approximately between 10% and 20% savings in the number of vertices is achieved at any time step. Notably, for sx-stackoverflow, which is also the largest network tested, the savings are significantly larger, ranging from 80% (at time step t_1) to 38% (at time step t_9).

Even though both classes of input graphs (synthetic and real-world) show linear growth rates in their respective sizes, for the synthetic inputs it is harder to benefit from Δ−screening because edges are connected almost randomly from new to existing vertices (by an edge sampling randomized procedure described in [53]); therefore, the location of the changes spreads throughout the whole network which is not ideal for Δ−screening. Whereas, in the real-world network sx-stackoverflow, we observed that the changes tend to happen more in a localized manner, giving a realistic opportunity to benefit from Δ−screening. This locality of changes tend to have a significant impact on the filtering efficacy of the Δ−screening technique. In fact, even between the two real-world networks, we observed a divergence—with the ArXiv HEP-TH input, the savings are relatively modest, compared to the large savings achieved for sx-stackoverflow.

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. 6 shows the runtime as a function of the time steps, for the different inputs tested.

In all the charts, static shows the runtime for static Louvain algorithm; add_base and add_Δs show the runtimes for the baseline and Δs implementations for edge additions (i.e., on the '+' inputs); and del_base and del_Δs show the runtimes for the baseline (Δs) for edge deletions (i.e., on the '-' inputs).

TABLE 2
Summary of the results.

Input	Time (sec)					Modularity		
	static (Louvain)	dyn_base (Louvain)	dyn_Δs (Louvain)	speedup (static)	speedup (dynamic)	static (Louvain)	dyn_Δs (Louvain)	last time step difference (Δs - static)(%)
50kll+	5.830	4.951	2.950	1.98×	1.68×	0.811420	0.810112	−0.161 %
50kll-	5.946	4.879	3.144	1.85×	1.22×	0.714571	0.714003	−0.079 %
50kll+/-	9.667	8.912	6.147	1.57×	1.45×	0.824450	0.821170	−0.397 %
50khh+	8.379	7.856	2.975	2.82×	2.64×	0.644720	0.640160	−0.707 %
50khh-	9.749	8.729	3.269	2.98×	2.67×	0.616282	0.612346	−0.638 %
50khh+/-	17.853	14.936	6.966	2.56×	2.14×	0.675580	0.672290	−0.486 %
5Mll+	11624.619	1355.518	809.842	14.35×	1.67×	0.773547	0.771104	−0.315 %
5Mll-	11624.619	2977.571	877.018	13.25×	3.40×	0.593547	0.596427	0.485 %
5Mll+/-	13674.270	2855.872	1785.081	7.66×	2.86×	0.798403	0.795219	−0.398 %
5Mhh+	32300.000	1140.275	870.660	37.10×	1.31×	0.564018	0.679368	20.451 %
5Mhh-	34200.000	3812.027	975.146	35.07×	3.91×	0.544984	0.536693	−1.521 %
5Mhh+/-	38600.000	5543.045	999.454	38.62×	5.55×	0.675542	0.681642	0.902 %
Arxiv HEP-TH+	0.399	0.260	0.218	1.83×	1.19×	0.664416	0.662571	−0.277 %
Arxiv HEP-TH-	0.399	0.401	0.352	1.13×	1.14×	0.822908	0.819265	−0.442 %
Arxiv HEP-TH+/-	0.537	0.498	0.389	1.28×	1.15×	0.661220	0.658227	−0.452 %
sx-stackoverflow+	1283.967	663.398	489.933	2.62×	1.35×	0.453040	0.457285	0.937 %
sx-stackoverflow-	1283.967	686.977	379.788	3.38×	1.81×	0.419580	0.416115	−0.825 %
sx-stackoverflow+/-	2575.719	1526.548	1042.741	2.47×	1.46×	0.460410	0.457926	−0.539 %
Enron+	275.844	136.429	70.231	3.93×	1.94×	0.689282	0.683282	−0.870 %

TABLE 3
Comparative evaluation of runtime and modularity for Δ−screening, DynaMo, and Batch algorithms. An entry '-' implies the run did not complete because of exceeding memory capacity.

Algorithms	Arxiv HEP-TH+/-			50k_ll+/-			sx-stackoverflow+/-		
	Time (sec)	Modularity	Peak memory	Time (sec)	Modularity	Peak memory	Time (sec)	Modularity	Peak memory
Δ−screening	**0.218**	**0.658227**	**<1 GB**	**6.147**	**0.821170**	**0.64 GB**	**1042.741**	**0.457926**	**3.2 GB**
DynaMo [39]	0.349	0.658180	3.2 GB	8.947	0.811556	54.6 GB	-	-	-
Batch [14]	1.941	0.650531	3.3 GB	33.747	0.803625	55.7 GB	-	-	-

We find that in all cases both baseline and Δs implementations consistently outperform the respective static implementation, providing more than two orders of magnitude in some 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 $4\times$ speedup at some time steps (e.g., input: sx-stackoverflow+; time steps t_6, t_8).

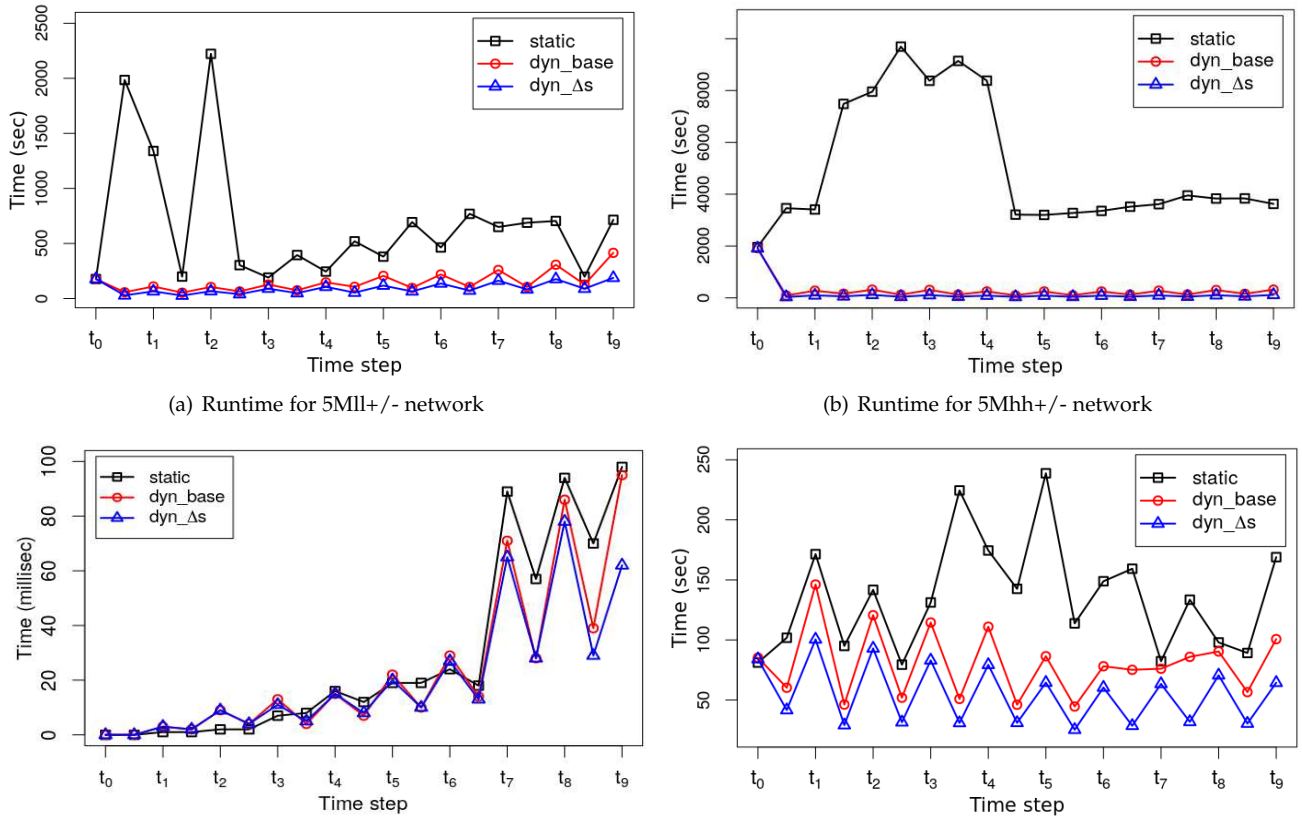
Fig. 7 shows the runtime performance of the unified framework (i.e., to handle a combination of additions and deletions; Algorithm 4), as a function of the time steps. Here too, we observe a similar trend, with the Δs scheme clearly outperforming static and performing comparably or outperforming the baseline.

Quality Evaluation: We also evaluated the quality, as measured by the modularity of the output clustering [5], achieved by the different clustering schemes. Figures S2 and S3 (in the Supplementary section) show the results of our qualitative evaluation. Basically, in nearly all cases, we observed no difference in the output modularity, across the three schemes—implying that the performance gains from Δ−screening does not have any negative impact on the quality.

4.3 Comparison with other tools

Louvain vs. SLM using Δ−screening: In order to demonstrate the generic applicability of the Δ−screening scheme for modularity optimizing frameworks, we also incorporated Δ−screening into the SLM algorithm [7]. Figure 8 shows a comparison between the runtimes for the Louvain implementations versus the corresponding SLM implementations, for the ArXiv HEP-TH network input. First, we note that the gap between the Δ−screening and static implementations, is significantly larger for SLM than for Louvain for this input. Secondly, note that the timings for SLM are about an order of magnitude larger than the timings for Louvain on this input. These results collectively demonstrate that the Δ−screening scheme has an even larger impact on generating savings for SLM than for Louvain. We generally observed higher modularity values in the output for SLM compared to Louvain.

Δ−screening vs. DynaMo vs. Batch: We performed a comparative evaluation of Δ−screening against two other state-of-the-art methods, namely DynaMo [39] and Batch [14] algorithms. Both provide incremental community update implementations. The results are shown in Table 3. As can be observed, Δ−screening outperforms these other two methods in runtime and modularity of output, with dynaMo coming in a close second on the runtime and modularity. The memory footprint of the other two tools were significantly larger than Δ−screening. For the larger input (sx-stackoverflow), only Δ−screening was able to



(a) Runtime for 5Mll+/- network

(b) Runtime for 5Mhh+/- network

(c) Runtime for Arxiv HEP-TH+/- network

(d) Runtime for sx-stackoverflow network

Fig. 7. Plots (a), (b), (c), and (d) show the runtime for each time step, for the network inputs which have both edge additions and deletions ('+/-').

complete using the available memory (64 GB).

4.4 Effect of Varying the Temporal Resolution

In many real-world dynamic graph 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 process of computing the dynamic communities. In order to facilitate such a study through dynamic community detection, in this section, 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.

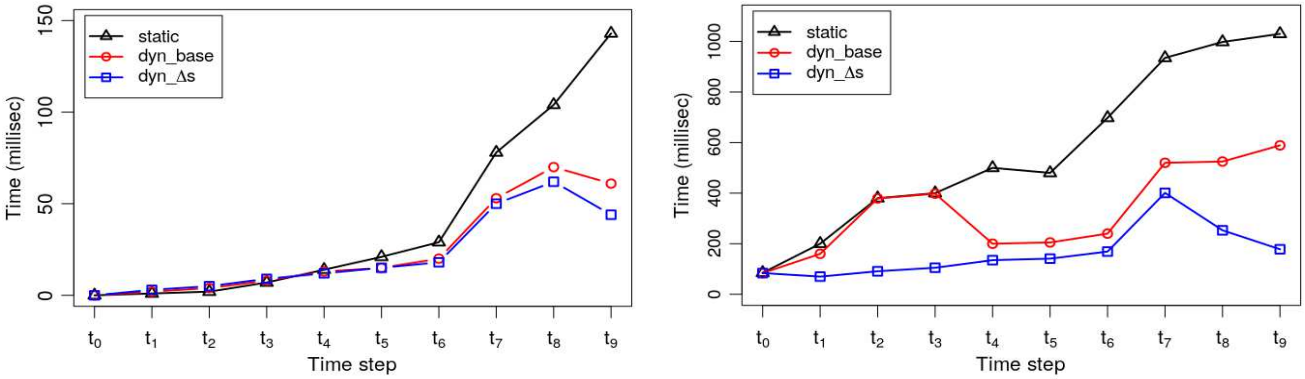
More specifically, using the sx-stackoverflow input, we first generated multiple temporal datasets, each of which representing the input stream divided into a certain number of time steps, ranging from {2, 4, 8, 12, 16, 20, 24, 28} steps. Note that in this scheme, there are multiple nested hierarchies—for instance, the 16-time steps partitioning can be achieved by splitting each of the 8-time steps partitions into two. Subsequently, we ran our Δs-enabled incremental clustering method on the different temporal input datasets (we used the dyn-Δs for Louvain in this analysis).

Figure 9 shows the results of this temporal resolution study. More specifically, Figure 9a shows, for the sx-stackoverflow+ input, the change in average modularity as we increase the temporal resolution from coarser to finer (left to right along the x-axis). Figure 9c shows the same for the sx-stackoverflow+/- input. 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 Figures 9b and 9d, which show 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).

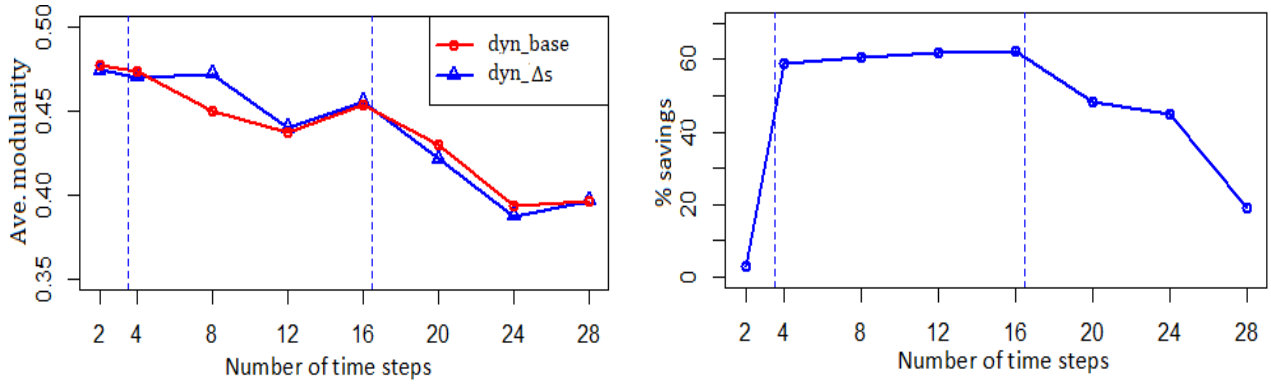
Note that to be able to conduct these kinds of temporal resolution studies, an application scientist should be able to run the dynamic community detection algorithm multiple times under different resolution configurations. This is one of the motivations for a fast implementation in practice.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TNSE.2021.3067665, IEEE Transactions on Network Science and Engineering



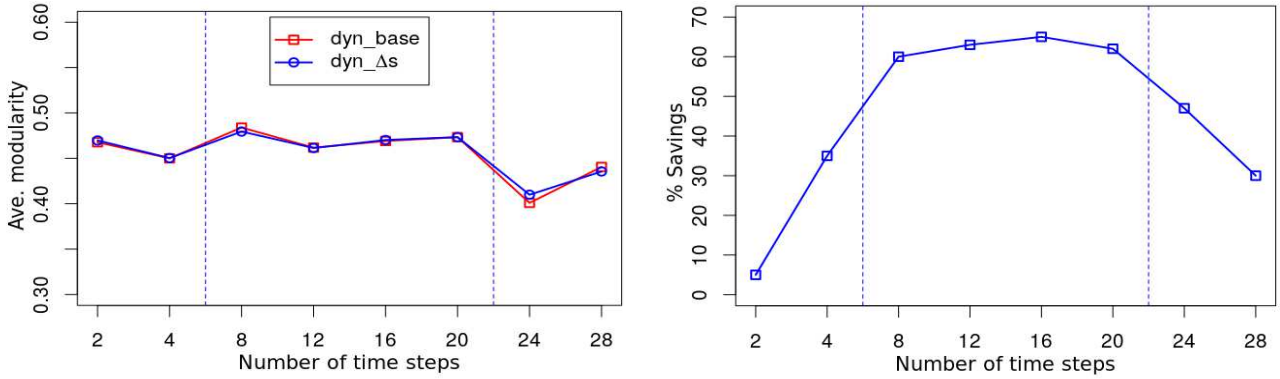
(a) Arxiv HEP-TH+ (using Louvain) (b) Arxiv HEP-TH+ (using SLM)

Fig. 8. Runtime comparison of Δ−screening scheme against the baseline and static schemes, for both Louvain and SLM implementations. For instance, part (b) shows SLM as the static baseline (shown in black), dynamic SLM (dSLM) as the dynamic baseline (shown in red), and the Δ−screening version shown in blue.



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

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



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

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

Fig. 9. 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.

5 Conclusion

Conducting community detection-based analysis on large dynamic networks is a time-consuming step in many discovery pipelines. In this paper, we visit a sub-problem 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 during any given time step, based on the loci of the changes.

Subsequently we incorporated this technique into two widely-used community detection tools (Louvain and SLM). Our experiments demonstrated speedups of up to $5\times$ and $38\times$ respectively, over state-of-the-art dynamic and static baseline implementation, without impacting the quality of the output. These experiments were conducted for a collection of synthetic and real-world inputs. Our tool is also more memory efficient demonstrating the ability to run on a larger input (sx-stackoverflow; 60M) where other implementations run out of memory. Our Δ-screening scheme is generic and can be incorporated into any modularity-optimizing community detection implementation. We also used the Δ-screening implementation to identify appropriate temporal resolutions at which to observe community structure within the dynamic network. Future research directions include parallelization on multicore platforms; and application to large-scale networks

toward a domain-specific analysis of the dynamic communities. The current implementation of the algorithm supports only undirected graphs and extending the algorithm to directed setting is also part of our future plan.

Acknowledgments

We thank Dr. Aurora Clark for providing us valuable input on potential scientific use-cases to motivate method development. The research was supported in part by the U.S. National Science Foundation (NSF) awards CCF 1815467 and OAC 1910213.

References

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

[2] 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.

[3] 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.

[4] 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.

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

[6] 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.

[7] 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.

[8] J. Xie, B. K. Szymanski, and X. Liu, "Slpa: Uncovering overlapping communities in social networks via a speaker-listener interaction dynamic process," in *2011 IEEE 11th International Conference on Data Mining Workshops*. IEEE, 2011, pp. 344–349.

[9] M. Rosvall and C. T. Bergstrom, "Maps of information flow reveal community structure in complex networks," *arXiv preprint physics.soc-ph/0707.0609*, 2007.

[10] A. Kalyanaraman, M. Halappanavar, D. Chavarría-Miranda, H. Lu, K. Duraisamy, P. P. Pande *et al.*, "Fast uncovering of graph communities on a chip: Toward scalable community detection on multicore and manycore platforms," *Foundations and Trends® in Electronic Design Automation*, vol. 10, no. 3, pp. 145–247, 2016.

[11] R. Cazabet, G. Rossetti, and F. Amblard, "Dynamic community detection," *Encyclopedia of Social Network Analysis and Mining*, 2017.

[12] F. Liu, J. Wu, S. Xue, C. Zhou, J. Yang, and Q. Sheng, "Detecting the evolving community structure in dynamic social networks," *World Wide Web*, vol. 23, no. 2, pp. 715–733, 2020.

[13] P. Holme and J. Saramäki, "Temporal networks," *Physics reports*, vol. 519, no. 3, pp. 97–125, 2012.

[14] W. H. Chong and L. N. Teow, "An incremental batch technique for community detection," in *Proceedings of the 16th International Conference on Information Fusion*. IEEE, 2013, pp. 750–757.

[15] 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.

[16] 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.

[17] 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.

[18] P. Agarwal, R. Verma, A. Agarwal, and T. Chakraborty, "Dyperm: Maximizing permanence for dynamic community detection," in *Pacific-Asia Conference on Knowledge Discovery and Data Mining*. Springer, 2018, pp. 437–449.

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

[20] A. Cuzzocrea, F. Folino, and C. Pizzuti, "Dynamicnet: an effective and efficient algorithm for supporting community evolution detection in time-evolving information networks," in *Proceedings of the 17th International Database Engineering & Applications Symposium*, 2013, pp. 148–153.

[21] 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.

[22] G. Rossetti and R. Cazabet, "Community discovery in dynamic networks: a survey," *ACM Computing Surveys (CSUR)*, vol. 51, no. 2, pp. 1–37, 2018.

[23] M. Takaffoli, "Community evolution in dynamic social networks–challenges and problems," in *2011 IEEE 11th International Conference on Data Mining Workshops*. IEEE, 2011, pp. 1211–1214.

[24] S. Asur, S. Parthasarathy, and D. Ucar, "An event-based framework for characterizing the evolutionary behavior of interaction graphs," *ACM Transactions on Knowledge Discovery from Data (TKDD)*, vol. 3, no. 4, pp. 1–36, 2009.

[25] J. He, D. Chen, C. Sun, Y. Fu, and W. Li, "Efficient stepwise detection of communities in temporal networks," *Physica A: Statistical Mechanics and its Applications*, vol. 469, pp. 438–446, 2017.
[26] M. Seifikar, S. Farzi, and M. Barati, "C-blondel: An efficient louvain-based dynamic community detection algorithm," *IEEE Transactions on Computational Social Systems*, vol. 7, no. 2, pp. 308–318, 2020.
[27] G. Palla, A.-L. Barabási, and T. Vicsek, "Quantifying social group evolution," *Nature*, vol. 446, no. 7136, pp. 664–667, 2007.
[28] I. Derényi, G. Palla, and T. Vicsek, "Clique percolation in random networks," *Physical review letters*, vol. 94, no. 16, p. 160202, 2005.
[29] 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.
[30] 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.
[31] 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.
[32] S. Fortunato and M. Barthelemy, "Resolution limit in community detection," *Proceedings of the National Academy of Sciences of the United States of America*, vol. 104, no. 1, pp. 36–41, 2007.
[33] A. Lancichinetti and S. Fortunato, "Limits of modularity maximization in community detection," *Physical review E*, vol. 84, no. 6, p. 066122, 2011.
[34] T. Chakraborty, S. Srinivasan, N. Ganguly, A. Mukherjee, and S. Bhowmick, "On the permanence of vertices in network communities," in *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2014, pp. 1396–1405.
[35] X. Zeng, W. Wang, C. Chen, and G. G. Yen, "A consensus community-based particle swarm optimization for dynamic community detection," *IEEE Transactions on Cybernetics*, vol. 50, no. 6, pp. 2502–2513, 2019.
[36] I. Messaoudi and N. Kamel, "A multi-objective bat algorithm for community detection on dynamic social networks," *Applied Intelligence*, vol. 49, no. 6, pp. 2119–2136, 2019.
[37] Z. Zhao, C. Li, X. Zhang, F. Chiclana, and E. H. Viedma, "An incremental method to detect communities in dynamic evolving social networks," *Knowledge-Based Systems*, vol. 163, pp. 404–415, 2019.
[38] L. Wu, Q. Zhang, K. Guo, E. Chen, and C. Xu, "Dynamic community detection method based on an improved evolutionary matrix," *Concurrency and Computation: Practice and Experience*, p. e5314, 2019.
[39] D. Zhuang, M. J. Chang, and M. Li, "Dynamo: Dynamic community detection by incrementally maximizing modularity," *IEEE Transactions on Knowledge and Data Engineering*, 2019.
[40] C. Tantipathananandh, T. Berger-Wolf, and D. Kempe, "A framework for community identification in dynamic social networks," in *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2007, pp. 717–726.
[41] 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.
[42] C. Matias and V. Miele, "Statistical clustering of temporal networks through a dynamic stochastic block model," *arXiv preprint arXiv:1506.07464*, 2015.
[43] C. Matias, T. Rebafka, and F. Villers, "A semiparametric extension of the stochastic block model for longitudinal networks," *Biometrika*, vol. 105, no. 3, pp. 665–680, 2018.
[44] K. S. Xu and A. O. Hero, "Dynamic stochastic blockmodels for time-evolving social networks," *IEEE Journal of Selected Topics in Signal Processing*, vol. 8, no. 4, pp. 552–562, 2014.
[45] T. Viard, M. Latapy, and C. Magnien, "Computing maximal cliques in link streams," *Theoretical Computer Science*, vol. 609, pp. 245–252, 2016.
[46] A.-S. Himmel, H. Molter, R. Niedermeier, and M. Sorge, "Enumerating maximal cliques in temporal graphs," in *2016 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM)*. IEEE, 2016, pp. 337–344.
[47] Z. Bu, H.-J. Li, C. Zhang, J. Cao, A. Li, and Y. Shi, "Graph k-means based on leader identification, dynamic game, and opinion dynamics," *IEEE Transactions on Knowledge and Data Engineering*, vol. 32, no. 7, pp. 1348–1361, 2019.
[48] N. Zarayeneh and A. Kalyanaraman, "A fast and efficient incremental approach toward dynamic community detection," in *Proceedings of the 2019 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining*, 2019, pp. 9–16.
[49] H. Lu, M. Halappanavar, and A. Kalyanaraman, "Parallel heuristics for scalable community detection," *Parallel Computing*, vol. 47, pp. 19–37, 2015.
[50] M. Girvan and M. E. Newman, "Community structure in social and biological networks," *Proceedings of the National Academy of Sciences of the United States of America*, vol. 99, no. 12, pp. 7821–7826, 2002.
[51] 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.
[52] 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.
[53] 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.
[54] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data, Jun. 2014.

Neda Zarayeneh received a BS degree in applied mathematics from Razi University, Kermanshah, Iran; an MS degree in applied mathematics from University of Tehran, Tehran, Iran; and an MS degree in computer science from Texas A&M University-Commerce, Commerce, TX, USA. Currently, Neda is a Ph.D. candidate in computer science at Washington State University, Pullman, Washington. Her research interests include graph algorithms, network and data science, and computational sciences.

Ananth Kalyanaraman (Senior Member, IEEE) received the bachelors degree from the Visvesvaraya National Institute of Technology, Nagpur, India, in 1998, and the MS and PhD degrees from Iowa State University, Ames, Iowa, in 2002 and 2006, respectively. Currently, he is a professor and boeing centennial chair in computer science, at the School of Electrical Engineering and Computer Science, Washington State University, Pullman, Washington. He also holds a joint appointment with Pacific Northwest National Laboratory, Richland, Washington. His research focuses on developing parallel algorithms and software for data-intensive problems originating in the areas of computational biology and graph-theoretic applications. He is a recipient of a DOE Early Career Award, Early Career Impact Award, and two best paper awards. He serves on editorial boards of the IEEE Transactions on Parallel and Distributed Systems and Journal of Parallel and Distributed Computing. He is a member of ACM, IEEE, and SIAM.

Supplementary Material: *Delta-Screening*: A Fast and Efficient Technique to Update Communities in Dynamic Graphs

Neda Zarayeneh, *Member, IEEE,* and Ananth Kalyanaraman, *Senior Member, IEEE*

SUPPLEMENTARY MATERIALS

Lemma 1. *Let i' be a neighbor of a vertex i that has a newly added edge to j_*. Then, the community assignment for i' at the end of time step t, i.e., $C_t(i')$, could be potentially different from its previous community $C_{t-1}(i')$.*

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 Figure 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 term of the modularity (Eqn. 1)). Even if the negative term of the same equation also decreases (due to 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 Figure 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 $\mathcal{R}_{t+}$. □

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

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). Consequently, we re-evaluate the community status of such vertices, by adding j_1 to $\mathcal{R}_t$. □

Lemma 7. *Let $(i, j) \in \Delta_{t-}$ be an intra-community edge to be deleted. Consider a vertex $k \in \Gamma(i) \setminus C_{t-1}(i)$. Then, it is possible for vertex k to have a positive incentive to change its community at time step t.*

Proof. This case is represented by vertex label k in Figure 2. First, we note that the deletion of edge (i, j) does not in anyway change the strength of vertex k's connection to its own community $C_{t-1}(k)$. However, if its neighbor i decides to migrate to a different community, say $C_{t-1}(j_*)$, then vertex k's strength to that community could also possibly improve owing to its edge to i. This might result in an increase in the positive term of Eqn. 1 relative to vertex k. Consequently, k is influenced by this edge deletion. □

Supplementary Results

Figure S1 shows the number of edges added and/or deleted at every time step of the different inputs. Figures S2 and S3 show the quality (measured in modularity) as a function of the time steps. As can be shown, all schemes deliver comparable modularities.

• *N. Zarayeneh and A. Kalyanaraman are with the School of Electrical Engineering and Computer Science, Washington State University, Pullman, WA, 99164.*
E-mail: {neda.zarayeneh, ananth}@wsu.edu

Manuscript received December 24, 2020; revised March 1, 2021.

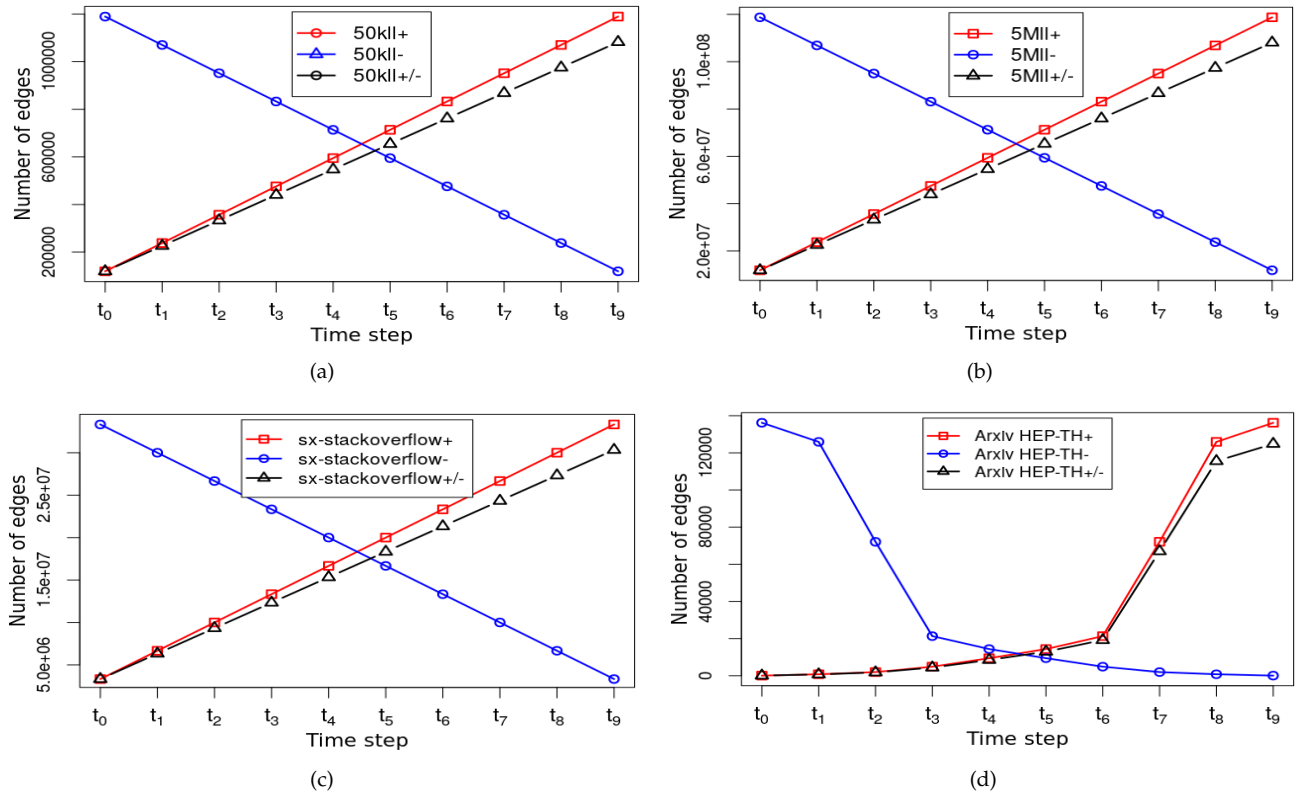


Fig. S1. The evolution of network size with each time step, for the synthetic and real world inputs considered for testing.

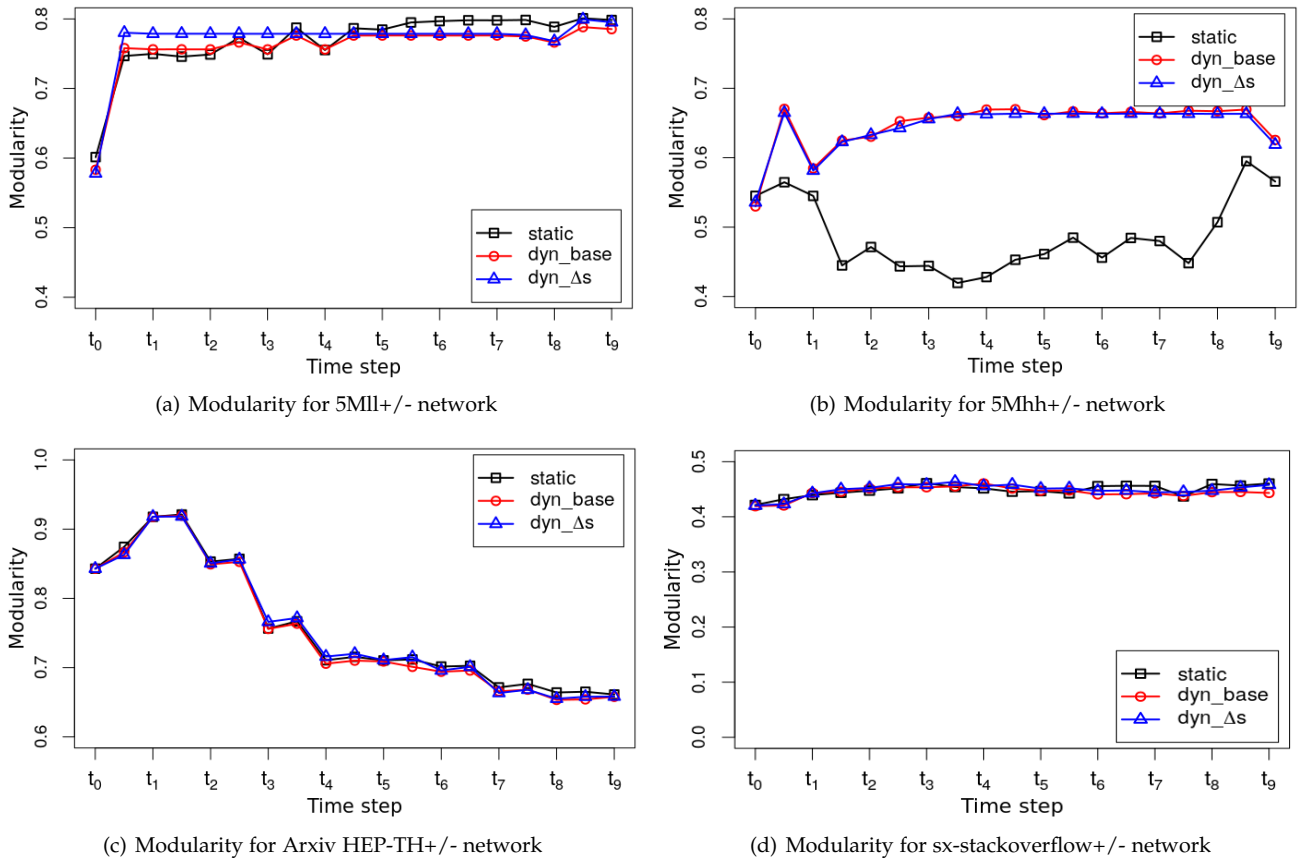
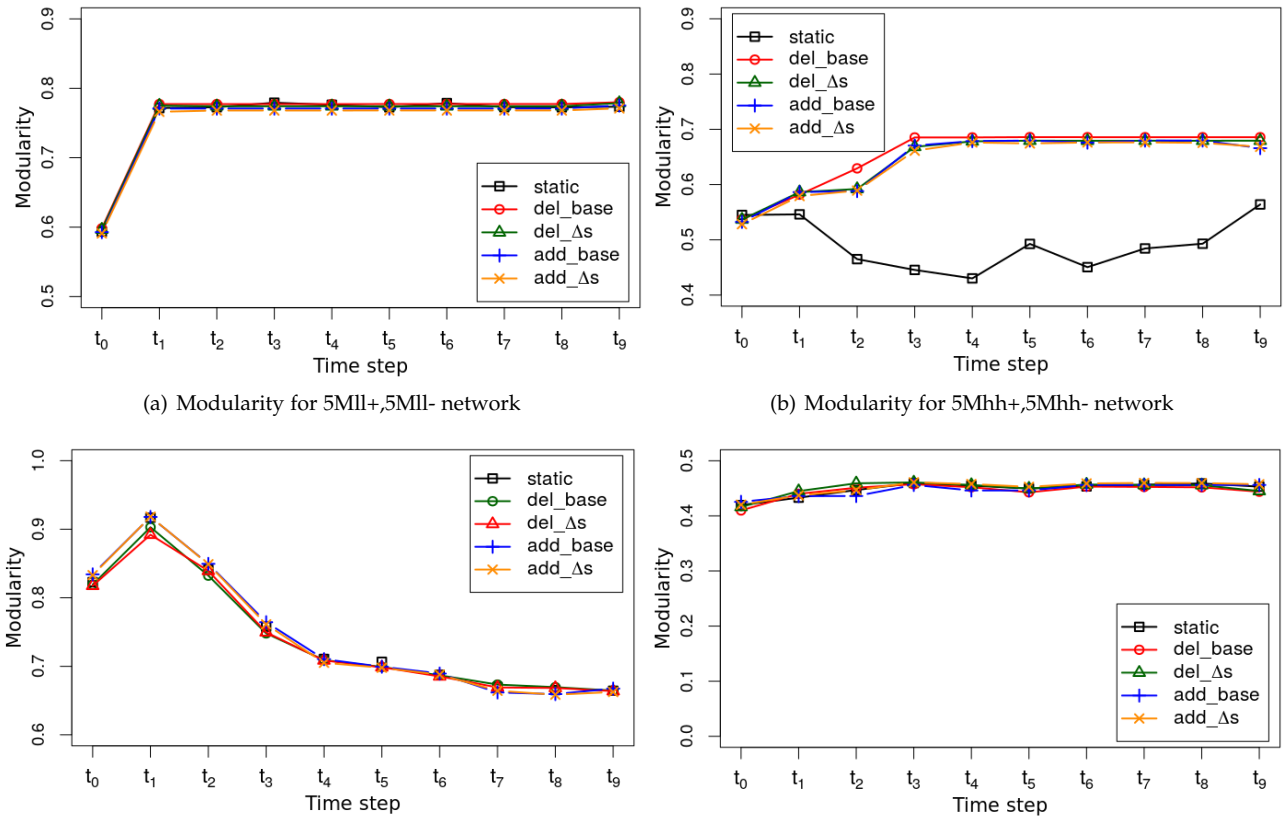


Fig. S2. Plots (a), (b), (c) and (d) show the change in average modularity across the different time steps, for different +/- inputs.



(a) Modularity for 5Mll+,5Mll- network

(b) Modularity for 5Mhh+,5Mhh- network

(c) Modularity for Arxiv HEP-TH+,Arxiv HEP-TH- network

(d) Modularity for sx-stackoverflow+,sx-stackoverflow- network

Fig. S3. Parts (a), (b), (c), and (d) show the change in average modularity with the time steps, for the growing and shrinking networks.