2019 IEEE 35th International Conference on Data Engineering (ICDE)

Tracking Influential Nodes in Time-Decaying Dynamic Interaction Networks

Junzhou Zhao[1] Shuo Shang[2] Pinghui Wang[3] John C.S. Lui[4] Xiangliang Zhang[1]

[1]*King Abdullah University of Science and Technology, KSA*
{junzhou.zhao, xiangliang.zhang}@kaust.edu.sa
[2]*Inception Institute of Artificial Intelligence, UAE*
jedi.shang@gmail.com
[3]*Xi'an Jiaotong University, China*
phwang@mail.xjtu.edu.cn
[4]*The Chinese University of Hong Kong, Hong Kong*
cslui@cse.cuhk.edu.hk

***Abstract*—Identifying influential nodes that can jointly trigger the maximum influence spread in networks is a fundamental problem in many applications such as viral marketing, online advertising, and disease control. Most existing studies assume that social influence is static and they fail to capture the dynamics of influence in reality. In this work, we address the dynamic influence challenge by designing efficient streaming methods that can identify influential nodes from highly dynamic node interaction streams. We first propose a general *time-decaying dynamic interaction network* (TDN) model to model node interaction streams with the ability to smoothly discard outdated data. Based on the TDN model, we design three algorithms, i.e.,** SIEVEADN**,** BASICREDUCTION**, and** HISTAPPROX**.** SIEVEADN **identifies influential nodes from a special kind of TDNs with efficiency.** BASICREDUCTION **uses** SIEVEADN **as a basic building block to identify influential nodes from general TDNs.** HISTAPPROX **significantly improves the efficiency of** BASICREDUCTION**. More importantly, we theoretically show that all three algorithms enjoy constant factor approximation guarantees. Experiments conducted on various real interaction datasets demonstrate that our approach finds near-optimal solutions with speed at least** 5 **to** 15 **times faster than baseline methods.**

I. INTRODUCTION

Online social networks allow their users to connect and interact with each other such as one user re-tweets/re-shares another user's tweets/posts on Twitter/Facebook. Interactions between connected users can cause members in the network to be influenced. For example, a video goes viral on Twitter after being re-tweeted many times, a rumor spreads like a wildfire on Facebook via re-sharing, etc. In these scenarios, users in the network are influenced (i.e., they watched the video or got the rumor) via a cascade of user interactions. Understanding and leveraging social influence have been hot in both academia and business. For example, in academia, identifying k users who can jointly trigger the maximum influence spread in a network is known as the *influence maximization* (IM) problem [1]; in business, leveraging social influence to boost product sales is known as viral marketing.

*Shuo Shang and Xiangliang Zhang are the corresponding authors.

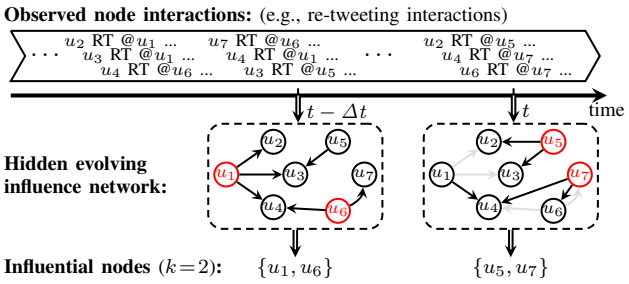


Fig. 1. Observed node interactions ("u RT @v ..." denotes that user u re-tweeted user v's tweet) and hidden evolving influence network (a directed edge (u, v) denotes that u influenced v). Influential nodes also evolve.

User interactions are first-hand evidences reflecting one user's influence on another, e.g., user a re-tweeting user b's tweet implies that b influenced a. Most studies on social influence [1]–[9] need to estimate *influence probabilities* based on observed user interactions [10]–[12]. Then, user influence is evaluated on an *influence network*, which is a directed graph with influence probabilities among nodes.

Dynamic influence challenge. One major issue in these studies is that both influence probabilities and influence network are assumed to be static, i.e., *social influence is assumed to be static*. However, social influence in real-world could be dynamic driven by the highly dynamic user interactions. For instance, user a frequently re-tweeted user b's tweets in the past few weeks, but stopped re-tweeting recently because b posted offensive content and a unfollowed b, thereby b no longer exerting influence on a. Indeed, [13] reported that Twitter network is highly dynamic with about 9% of all connections changing in every month. Moreover, user interactions can be drastically affected by external out-of-network sources such as mass media, newspapers, and TV stations [14]. Consequently, it is no longer suitable to assume that social influence is static; otherwise the identified influential users

2375-026X/19/$31.00 ©2019 IEEE
DOI 10.1109/ICDE.2019.00102

in IM may just quickly become outdated. This raises the following problem: *in a world with highly dynamic user interactions, how can we efficiently identify k most influential users at any time, as illustrated in Fig. 1?*

A straightforward way to handle the dynamics of influence is that we re-compute everything from scratch every time we need to query influential users, i.e., re-estimate influence probabilities and re-identify influential users on the updated influence network. Obviously, this approach incurs too much computational overhead which may be unaffordable if we need to query influential users frequently. There have been some recent research efforts trying to address the dynamic influence challenge, such as the heuristic approaches [15,16], building updatable sketches [17,18], and the interchange greedy approach [19]. However, these methods either do not have theoretical guarantees on the quality of selected users (e.g., heuristic approaches [15,16]), or they cannot handle highly dynamic data (e.g., the interchange greedy approach [19] actually degrades to the re-computation approach when influential users change significantly over time). In addition, these methods assume that influence probabilities are given in advance; however, estimating influence probabilities itself could be a high complexity inference task [10]–[12,20], especially when influence probabilities are time-varying.

Present work: a streaming optimization approach. In this work, we explore the potential of designing efficient streaming methods to address the dynamic influence challenge.

When user interactions are continuously generated and aggregated in chronological order, they form a stream. An appealing approach for identifying influential users is to process this user interaction stream directly, in a streaming fashion [21]. Specifically, we attempt to maintain some compact *intermediate results* while processing the stream. We keep updating these intermediate results when new user interactions are examined. At any query time, we can quickly obtain a solution using the maintained intermediate results. This streaming approach has the potential to allow us to track influential users over time continuously. However, to materialize this ideal streaming approach, we need to carefully address following concerns.

- *Data recency.* Older user interactions are less significant than more recent ones in evaluating users' current influence. For example, the observation that user b's tweet was re-tweeted a year ago, is less valuable than the observation that b's tweet was re-tweeted yesterday, when evaluating b's current influence. The streaming approach is required to have a mechanism that can properly discard outdated data.
- *Space and update efficiency.* Space used by storing intermediate results should be compact and upper bounded with the progression of the stream. Meanwhile, the update operation should be as efficient as possible so that we can handle high-speed user interaction streams which are common in practice.
- *Solution quality.* The output solution should be close to the optimal solution at any query time.

This paper is essentially devoted to address above concerns by proposing a general streaming model and designing a set of streaming algorithms.

To address the data recency concern, a commonly used streaming model in the literature is the *sliding-window* model [22]–[24] where only the most recent W elements in the stream remain active and the rest are discarded. For example, [25] recently developed a streaming method based on the sliding-window model to solve IM in a streaming fashion. However, the sliding-window model has its limitation which can be exposed by the following example.

Example 1. *Suppose we want to identify influential users on Twitter based on re-tweeting interactions, i.e., if a user's tweets were re-tweeted by many other users, the user is considered to be influential. Alice is an influential user on Twitter for many years. Recently, Alice is ill and has been in hospital for weeks. During this period, Alice cannot use Twitter. Because Alice disappeared from her followers' timelines, no user re-tweeted her tweets during this period.*

In Example 1, if the sliding-window size is too small that no re-tweeting interaction related to Alice is observed, then Alice will be considered not influential, even though she has been influential for many years and her absence is merely temporal. This example demonstrates that sliding-window model does not discard outdated data in a smooth manner and results in unstable solutions. It thus motivates us to find better models.

- **TDN model.** In this work, we propose a general ***t**ime-decaying **d**ynamic interaction **n**etwork* (TDN) model to enable smoothly discarding outdated user interactions, rather than the non-smooth manner in sliding-window model. In TDN, each user interaction is assigned a *lifetime*. The lifetime is the maximum time span that a user interaction can remain active. Lifetime automatically decreases as time elapses. If the lifetime becomes zero, the corresponding user interaction is discarded. By choosing different lifetime assignments, TDN model can be configured to discard outdated user interactions in various ways, which include the sliding-window model as a special case. In short, TDN is a general streaming model to address the data recency issue.
- **Efficient streaming algorithms.** We address the other concerns by designing three streaming algorithms, i.e., SIEVEADN, BASICREDUCTION, and HISTAPPROX, all based on the TDN model. SIEVEADN can identify influential nodes over a special kind of TDNs. BASICREDUCTION leverages SIEVEADN as a basic building block to identify influential nodes over general TDNs. HISTAPPROX significantly improves the efficiency of BASICREDUCTION. Our streaming algorithms are inspired by the streaming submodular optimization (SSO) techniques [24,26]. Current SSO techniques can only handle insertion-only [26] and sliding-window [24] streams. To the best of our knowledge, the work in this paper is the first to handle more general time-decaying streams. More importantly, we theoretically show that our approach can find near-optimal solutions with both time and space efficiency.

Contributions. In summary, our contributions are as follows:

- We propose a general TDN model to model user interaction streaming data with the ability to discard outdated

user interactions smoothly (§II).

- We design three streaming algorithms based on the TDN model, namely SIEVEADN, BASICREDUCTION, and HISTAPPROX. Our algorithms are applicable to time-decaying streams and achieve a constant $(1/2-\epsilon)$ approximation guarantee (§III and §IV).
- We conduct extensive experiments on various real interaction datasets. The results demonstrate that our approach outputs near-optimal solutions with speed at least 5 to 15 times faster than baseline methods. (§V).

II. PRELIMINARIES AND PROBLEM FORMULATION

Notice that interactions are not necessarily occurred between two users but could be between any two entities, or nodes in networks. In this section, we first formally define the general node interaction data. Then we propose a time-decaying dynamic interaction network model. Finally, we formulate the problem of tracking influential nodes.

A. Node Interactions

Definition 1 (Interaction)**.** *An interaction between two nodes in a network is a triplet $\langle u, v, \tau\rangle$ representing that node u exerts an influence on node v at time τ.*

For example, user v re-tweets/re-shares user u's tweet/post at time τ on Twitter/Facebook, user v adopted a product recommended by user u at time τ, etc. In these scenarios, u influenced v. An interaction $\langle u, v, \tau\rangle$ is a direct evidence indicating that u influences v. If we observe many such evidences, then we say that u has strong influence on v.

In some scenarios, we may not directly observe the interaction between two nodes, but if they do have an influence relationship, we are still able to convert these scenarios to the scenario in Definition 1, e.g., by one-mode projection.

Example 2 (One-mode Projection)**.** *User u bought a T-shirt recently. His friend v also bought a same T-shirt two days later at time τ. Then, it is very likely that u influenced v. We still denote this interaction by $\langle u, v, \tau\rangle$.*

When interactions are continuously generated and aggregated, they form an interaction stream.

Definition 2 (Interaction Stream)**.** *An interaction stream is an infinite set of interactions generated in discrete time, denoted by $\mathcal{S} \triangleq \{\langle u, v, \tau\rangle : u, v$ are two distinct nodes, $\tau = 1, 2, \ldots\}$.*

For ease of presentation, we use discrete time in this work, and we allow a batch of node interactions arriving at the same time. Interaction stream will be the input of our algorithms.

As we discussed previously, older interactions are less significant than more recent ones in evaluating current influence. Next, we propose a time-decaying mechanism to satisfy this recency requirement desired by the streaming approach.

B. Time-Decaying Dynamic Interaction Network Model

We propose a simple and general dynamic network model to model an interaction stream. The model leverages a *time-decaying mechanism* to smoothly discard outdated interactions. We refer to our model as the ***time-decaying dynamic interaction network*** (TDN) model.

Formally, a TDN at time t is simply a directed network denoted by $G_t \triangleq (V_t, E_t)$, where V_t is a set of nodes and E_t is a set of edges *survived* by time t. Each edge $(u, v, \tau) \in E_t$ is directed and timestamped representing an interaction. We assume there is no self-loop edge (i.e., a user cannot influence himself) but allow multiple edges between two nodes (e.g., a user influences another user multiple times at different time).

TDN model leverages a *time-decaying mechanism* to handle continuous node/edge additions and deletions while evolving. The time-decaying mechanism works as follows: an edge is added to the TDN at its creation time; the edge survives in the network for a while then expires; when the edge expires, the edge is removed from the network; if edges attached to a node all expire, the node is removed from the network.

Formally, for an edge $e = (u, v, \tau)$ arrived at time τ, it is assigned a lifetime $l_\tau(e) \in \{1, \ldots, L\}$ upper bounded by L. The edge's lifetime decreases as time elapses, and at time $t \geq \tau$, its lifetime decreases to $l_t(e) = l_\tau(e) - t + \tau$. If $l_{t'}(e) = 0$ at some time $t' > \tau$, edge e is removed from the network. This also implies that $e \in E_t$ iff $\tau \leq t < \tau + l_\tau(e)$.

Note that lifetime plays the same role as a *time-decaying weight*. If an edge has a long lifetime at its creation time, the edge is considered to be important and will survive in the network for a long time. An example of such a TDN evolving from time t to $t+1$ is given in Fig. 2.

$e \triangleq (u, v, \tau)$	$l_\tau(e)$
$e_1 = (u_1, u_2, t)$	1
$e_2 = (u_1, u_3, t)$	1
$e_3 = (u_1, u_4, t)$	2
$e_4 = (u_5, u_3, t)$	3
$e_5 = (u_6, u_4, t)$	1
$e_6 = (u_6, u_7, t)$	1
$e_7 = (u_5, u_2, t+1)$	1
$e_8 = (u_7, u_4, t+1)$	2
$e_9 = (u_7, u_6, t+1)$	3

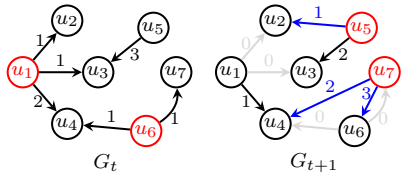


Fig. 2. A TDN example. Label on each edge denotes the edge's current lifetime. Influential nodes ($k = 2$) also evolve from time t to $t+1$.

We find that such a simple time-decaying mechanism makes TDN model highly configurable by choosing different lifetime assignment methods. Here, we consider a few special TDNs.

Example 3. *A network only grows with no node/edge deletions. It is equivalent to saying that every edge in a TDN has an infinite lifetime, i.e., $l_\tau(e) = \infty$. We refer to such networks as **addition-only dynamic interaction networks** (ADNs).*

Example 4. *A network consists of edges in the most recent W time steps. It is equivalent to saying that every edge in a TDN has same lifetime W, i.e., $l_\tau(e) = W$. We refer to such networks as sliding-window dynamic interaction networks.*

Example 5. *At each time step, we first delete each existing edge with probability p, then add the new arrival edges. To understand why this kind of dynamics can also be modeled using TDN, we can think of deleting an edge as a Bernoulli trial with success probability p. Therefore, an edge surviving for l time steps in the graph has probability $(1-p)^{l-1}p$, aka*

the geometric distribution. Hence, above dynamic process is equivalent to saying that each edge in a TDN has a lifetime independently sampled from a geometric distribution, i.e., $Pr(l_\tau(e) = l) = (1-p)^{l-1}p$. *We refer to such networks as probabilistic time-decaying dynamic interaction networks.*

The time-decaying mechanism not only helps discard outdated edges in G_t but also reduce the storage of maintaining G_t in computer main memory. For example, if edge lifetimes follow a geometric distribution $Pr(l_\tau(e) = l) = (1-p)^{l-1}p$ (with $L = \infty$), and we assume at most m new edges arrive at each time, then the memory needed to store G_t at any time t is upper bounded by $O(\sum_{i=0}^{\infty} m(1-p)^i) = O(m/p)$.

In the following discussion, in order to keep our methodology general, we will not assume a particular form of $l_\tau(e)$ but **assume that** $l_\tau(e)$ **is given as a user-chosen input** to our framework so that G_t can be stored in computer main memory. We also introduce some shortcut notations for later use. Given an edge $e \in E_t$, we will use u_e, v_e, τ_e and l_e to denote the edge's attributes, and lifetime at time t, respectively.

C. Problem Formulation

Up till now, we have modeled an interaction stream as a TDN. We now define the *influence spread* on TDNs.

Definition 3 (Influence Spread on TDNs)**.** *At time t, the influence spread of a set of nodes $S \subseteq V_t$ is the number of distinct nodes that are reachable from S in G_t, i.e.,*

$$f_t(S) \triangleq |\{v \in V_t : v \text{ is reachable from nodes } S \text{ in } G_t\}|.$$

Remember that each edge $e \in E_t$ represents that node u_e can influence node v_e. Thus, Definition 3 actually states that, if nodes in S can influence many nodes in G_t in a cascading manner, S has large influence in G_t. It is not hard to see that f_t satisfies following property [1].

Theorem 1. *$f_t : 2^{V_t} \mapsto \mathbb{R}_{\geq 0}$ defined in Definition 3 is a normalized monotone submodular set function*[1].

Of course, readers can define more complicated influence spread in TDNs. As long as Theorem 1 holds, our developed framework, which will be elaborated on in the remainder of this paper, is still applicable. We are now ready to formulate the influential nodes tracking problem.

Problem 1 (Tracking Influential Nodes in TDNs)**.** *Let $G_t = (V_t, E_t)$ denote an evolving TDN at time t. Let $k > 0$ denote a given budget. At any time t, we want to find a subset of nodes $S_t^* \subseteq V_t$ with cardinality at most k such that these nodes have the maximum influence spread on G_t, i.e., $f_t(S_t^*) = \max_{S \subseteq V_t \wedge |S| \leq k} f_t(S)$.*

Remarks.

- Figure 2 gives an example to show the time-varying nature of influential nodes in TDNs.
- Problem 1 is NP-hard in general. When G_t is large in scale, it is only practical to find approximate solutions. We say a solution S_t is an α-approximate solution if $f_t(S_t) \geq \alpha f_t(S_t^*)$ where $0 < \alpha < 1$.
- A straightforward way to solve Problem 1 is to re-run existing algorithms designed for static networks, e.g., the greedy algorithm [1], at every time the network is updated. This approach gives a $(1-1/e)$-approximate solution with time complexity $O(k|V_t|\gamma)$ where γ is the time complexity of evaluating f_t on G_t. We hope to find faster methods than this baseline method with comparable approximation guarantees.

[1]A set function $f : 2^V \mapsto \mathbb{R}_{\geq 0}$ is monotone if $f(S) \leq f(T)$ holds for all $S \subseteq T \subseteq V$; f is submodular if $f(S \cup \{s\}) - f(S) \geq f(T \cup \{s\}) - f(T)$ holds for all $S \subseteq T \subseteq V$ and $s \in V \backslash T$; f is normalized if $f(\emptyset) = 0$ ([27]).

III. A Basic Approach

In this section, we elaborate a basic approach on solving Problem 1. To motivate this basic approach, we first consider solving a special problem: tracking influential nodes over addition-only dynamic interaction networks (ADNs, refer to Example 3). We find that this special problem is closely related to a well-studied insertion-only streaming submodular optimization problem [26]. It thus inspires us to design SieveADN to solve this special problem efficiently. Using SieveADN as a basic building block, we show how to design BasicReduction to solve Problem 1 on general TDNs.

A. SieveADN: Tracking Influential Nodes over ADNs

In an ADN, each edge has an infinite lifetime. Arriving new edges simply accumulate in G_t. Therefore, the influence spread of a fixed set of nodes cannot decrease, i.e., $f_t(S) \geq f_{t'}(S)$ holds for all $S \subseteq V_{t'}$ whenever $t \geq t'$.

We find that identifying influential nodes on ADNs is related to a well-studied *insertion-only streaming submodular optimization (SSO) problem* [26]. In the follows, we first briefly recap the insertion-only SSO problem, and describe a streaming algorithm, called SieveStreaming, that solves the insertion-only SSO problem efficiently.

The insertion-only SSO problem considers maximizing a monotone submodular set function f over a set of elements U with a cardinality constraint, i.e., choosing at most k elements from U to maximize f. Each element in the set is allowed to be accessed only once in a streaming fashion. The goal is to find algorithms that use sublinear memory and time. One such algorithm is SieveStreaming [26].

SieveStreaming lazily maintains a set of thresholds $\Theta \triangleq \{\frac{(1+\epsilon)^i}{2k} : (1+\epsilon)^i \in [\Delta, 2k\Delta], i \in \mathbb{Z}\}$, where $\Delta \triangleq \max_u f(\{u\})$ is the maximum value of a singleton element seeing in the stream so far. Each threshold $\theta \in \Theta$ is associated with a set S_θ which is initialized to be empty. For each arriving element v, its marginal gain w.r.t. each set S_θ is calculated, i.e., $\delta_{S_\theta}(v) \triangleq f(S_\theta \cup \{v\}) - f(S_\theta)$. If $\delta_{S_\theta}(v) \geq \theta$ and $|S_\theta| < k$, v is saved into S_θ; otherwise v is not saved. At query time, SieveStreaming returns a set $S_{\theta^\star}$ that has the maximum value, i.e., $f(S_{\theta^\star}) = \max\{f(S_\theta) : \theta \in \Theta\}$. SieveStreaming is proven to achieve an $(1/2-\epsilon)$ approximation guarantee.

We leverage SieveStreaming to solve our special problem of tracking influential nodes over ADNs as follows. Let $\bar{E}_t$ denote a set of edges arrived at time t. Let $\bar{V}_t$

denote a set of nodes whose influence spread changes due to adding new edges $\bar{E}_t$ in G_t. We feed each node in $\bar{V}_t$ to SIEVESTREAMING whose output is our solution. We refer to this algorithm as SIEVEADN, as illustrated in Fig. 3 and described in Alg. 1.

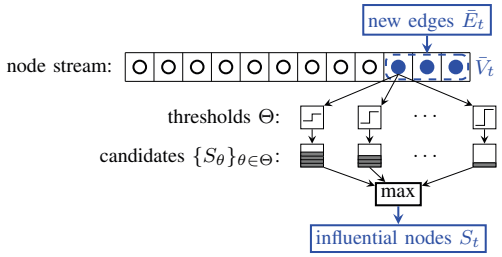


Fig. 3. Illustration of SIEVEADN

Alg. 1: SIEVEADN

Input: A sequence of edges arriving over time; k and ϵ
Output: A set of influential nodes at time t

1 $\Delta \leftarrow 0, \Theta \leftarrow \emptyset$, and $\{S_\theta\}_{\theta\in\Theta}$ denotes a family of sets;
2 **for** *arriving edges* $\bar{E}_t$ *at time* $t = 1, 2, \ldots$ **do**
3 $\bar{V}_t \leftarrow$ a set of nodes whose influence spread changes;
// Lines 4-7 lazily maintain a set of thresholds.
4 $\Delta \leftarrow \max\{\Delta, \max_{v\in\bar{V}_t} f_t(\{v\})\}$;
5 $\Theta' \leftarrow \{\frac{(1+\epsilon)^i}{2k} : (1+\epsilon)^i \in [\Delta, 2k\Delta], i \in \mathbb{Z}\}$;
6 Delete S_θ for $\theta \in \Theta\backslash\Theta'$ and let $S_\theta \leftarrow \emptyset$ for $\theta \in \Theta'\backslash\Theta$;
7 $\Theta \leftarrow \Theta'$;
// Lines 8-11 filter nodes by thresholds.
8 **foreach** *node* $v \in \bar{V}_t$ **do**
9 **foreach** *threshold* $\theta \in \Theta$ **do**
10 **if** $|S_\theta| < k$ *and* $\delta_{S_\theta}(v) \geq \theta$ **then**
11 $S_\theta \leftarrow S_\theta \cup \{v\}$;
// Return a solution having the maximum value.
12 $S_t \leftarrow S_{\theta^\star}$ where $\theta^\star = \arg\max_{\theta\in\Theta} f_t(S_\theta)$;

We emphasize that our special problem has two major differences with the insertion-only SSO problem. In the insertion-only SSO problem, each element appears only once in the stream, and the objective f is invariant to time. While in our problem, we allow same nodes to appear multiple times in the node stream, and our objective f_t is time-varying. Therefore, we still need to strictly prove that SIEVEADN outputs solutions with constant approximation guarantees. Thanks to the property of ADNs: when new edges are added, the influence spread of a fixed set of nodes cannot decrease. We can leverage this property and demonstrate that SIEVEADN indeed preserves the approximation guarantee of SIEVESTREAMING, and achieves an approximation factor of $(1/2 - \epsilon)$.

Theorem 2. SIEVEADN *achieves an* $(1/2-\epsilon)$ *approximation guarantee.*

Proof. Please refer to our Technique Report [28]. □

The intermediate results maintained by SIEVEADN are sets $\{S_\theta\}_{\theta\in\Theta}$. The following theorem states the time and space complexity of updating and storing these sets.

Theorem 3. *Let* b *denote the average size of set* $\bar{V}_t$. *Then* SIEVEADN *uses* $O(b\gamma\epsilon^{-1}\log k)$ *time to process each batch of edges (where* γ *is the time complexity of evaluating* f_t*) and* $O(k\epsilon^{-1}\log k)$ *space to maintain intermediate results.*

Proof. Please refer to our Technique Report [28]. □

Remarks.

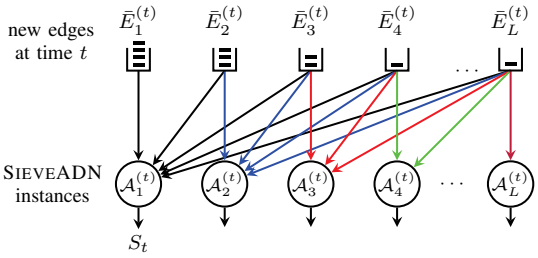
- b is typically small and $b \ll |V_t|$ in practice. Note that even if $b = |V_t|$, SIEVEADN still has lower time complexity than greedy algorithm which has time complexity $O(k|V_t|\gamma)$.
- Lines 8-11 in Alg. 1 can be easily implemented using parallel computation to further reduce the running time.

B. BASICREDUCTION*: Using* SIEVEADN *as A Basic Building Block*

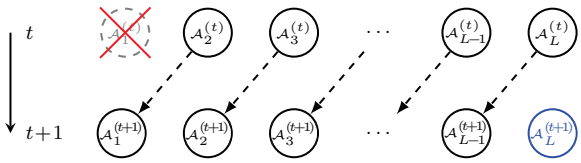
SIEVEADN can be used as a basic building block to design a basic method, called BASICREDUCTION, to solve Problem 1. In the follows, we first describe BASICREDUCTION, and then use an example to help readers understand its correctness.

Recall that $\bar{E}_t$ is a set of edges arrived at time t. We partition edges $\bar{E}_t$ into (at most) L groups by their lifetimes. Let $\bar{E}_l^{(t)} \subseteq \bar{E}_t$ denote the group having lifetime l, i.e., $\bar{E}_l^{(t)} \triangleq \{e : e \in \bar{E}_t \wedge l_e = l\}$, and $\bar{E}_t = \cup_{l=1}^{L} \bar{E}_l^{(t)}$.

BASICREDUCTION maintains L SIEVEADN instances at each time t, denoted by $\{\mathcal{A}_i^{(t)}\}_{i=1}^{L}$. At each time t, $\mathcal{A}_i^{(t)}$ only processes edges $\cup_{l=i}^{L} \bar{E}_l^{(t)}$, i.e., edges with lifetime no less than i. The relationship between input edges and SIEVEADN instances is illustrated in Fig. 4(a).



(a) Processing new edges $\bar{E}_t$ at time t



(b) Shifting SIEVEADN instances from time t to time $t+1$

Fig. 4. Illustration of BASICREDUCTION.

These SIEVEADN instances are maintained in a way that existing instances gradually expire and are terminated as edges processed in them expire; meanwhile, new instances are created as new edges arrive. Specifically, after processing edges $\bar{E}_t$, we do following operations at the beginning of time step $t+1$ to prepare for processing edges $\bar{E}_{t+1}$: (1) $\mathcal{A}_1^{(t)}$ expires and is terminated; (2) $\mathcal{A}_i^{(t)}$ is renamed to $\mathcal{A}_{i-1}^{(t+1)}$, for $i = 2, \ldots, L$; (3) a new SIEVEADN instance $\mathcal{A}_L^{(t+1)}$ is created

and appended at the tail. In short, SIEVEADN instances from index 2 to L at time t are "shifted" one unit to the left, and a new SIEVEADN instance is appended at the tail, as illustrated in Fig. 4(b). Then, BASICREDUCTION processes $\bar{E}_{t+1}$ similarly to processing $\bar{E}_t$ previously. It is easier to understand its execution using the following example.

Example 6. BASICREDUCTION *processes the TDN in Fig. 2 as follows.* $L = 3$ SIEVEADN *instances are maintained.* $\mathcal{A}_1^{(t)}, \mathcal{A}_2^{(t)}$ *and* $\mathcal{A}_3^{(t)}$ *process edges arrived at time* t *according to their lifetimes (see the table below). At time* $t+1$, $\mathcal{A}_1^{(t)}$ *expires;* $\mathcal{A}_2^{(t)}$ *is renamed to* $\mathcal{A}_1^{(t+1)}$ *and continues processing edges* $\{e_7, e_8, e_9\}$*;* $\mathcal{A}_3^{(t)}$ *is renamed to* $\mathcal{A}_2^{(t+1)}$ *and continues processing edges* $\{e_8, e_9\}$*.* $\mathcal{A}_3^{(t+1)}$ *is newly created, and processes edge* e_9*.*

▲ $\mathcal{A}_1^{(t)}: \{e_1, e_2, e_3, e_4, e_5, e_6\}$	● $\mathcal{A}_1^{(t+1)}: \{e_3, e_4, e_7, e_8, e_9\}$
● $\mathcal{A}_2^{(t)}: \{e_3, e_4\}$	■ $\mathcal{A}_2^{(t+1)}: \{e_4, e_8, e_9\}$
■ $\mathcal{A}_3^{(t)}: \{e_4\}$	◆ $\mathcal{A}_3^{(t+1)}: \{e_9\}$

The procedure repeats. It is clear to see that $\mathcal{A}_1^{(t)}$ *always processed all of the edges in* G_t *at any time* t*.*

Because at any time t, $\mathcal{A}_1^{(t)}$ processed all of the edges in G_t, the output of $\mathcal{A}_1^{(t)}$ is the solution at time t. The complete pseudo-code of BASICREDUCTION is given in Alg. 2.

Alg. 2: BASICREDUCTION

Input: A sequence of edges arriving over time
Output: A set of influential nodes at time t

Initialize SIEVEADN instances $\{\mathcal{A}_i^{(1)}: i = 1, \ldots, L\}$;
for $t = 1, 2, \ldots$ **do**
  for $i = 1, \ldots, L$ **do** Feed $\mathcal{A}_i^{(t)}$ with edges $\cup_{l=i}^{L} \bar{E}_l^{(t)}$;
  $S_t \leftarrow$ output of $\mathcal{A}_1^{(t)}$;
  Terminate $\mathcal{A}_1^{(t)}$;
  for $i = 2, \ldots, L$ **do** $\mathcal{A}_{i-1}^{(t+1)} \leftarrow \mathcal{A}_i^{(t)}$;
  Create and initialize $\mathcal{A}_L^{(t+1)}$;

Since $\mathcal{A}_1^{(t)}$ is a SIEVEADN instance, its output has an approximation factor of $(1/2 - \epsilon)$ according to Theorem 2. We hence have the following conclusion.

Theorem 4. BASICREDUCTION *achieves an* $(1/2 - \epsilon)$*-approximation guarantee on TDNs.*

Furthermore, because BASICREDUCTION contains L SIEVEADN instances, so its time complexity and space complexity are both L times larger (assuming SIEVEADN instances are executed in series).

Theorem 5. BASICREDUCTION *uses* $O(Lb\gamma\epsilon^{-1}\log k)$ *time to process each batch of arriving edges, and* $O(Lk\epsilon^{-1}\log k)$ *memory to store the intermediate results.*

Remarks.

- SIEVEADN instances in BASICREDUCTION can be executed in parallel. In this regard, the computational efficiency can be greatly improved.
- Notice that edges with lifetime l will be input to $\mathcal{A}_i^{(t)}$ with $i \leq l$. Hence, edges with large lifetime will fan out to a large fraction of SIEVEADN instances, and incur high CPU and memory usage, especially when L is large. This is the main bottleneck of BASICREDUCTION.
- On the other hand, edges with small lifetime only need to be processed by a few SIEVEADN instances. If edge lifetime is mainly distributed over small lifetimes, e.g., geometrically distributed, exponentially distributed or power-law distributed, then BASICREDUCTION could be as efficient as SIEVEADN.

IV. HISTAPPROX: IMPROVING EFFICIENCY USING HISTOGRAM APPROXIMATION

BASICREDUCTION needs to maintain L SIEVEADN instances. Processing large lifetime edges is a bottleneck. In this section, we design HISTAPPROX to address its weakness. HISTAPPROX allows infinitely large L and improves the efficiency of BASICREDUCTION significantly.

A. Basic Idea

BASICREDUCTION does not leverage the outputs of SIEVEADN instances until they are shifted to the head (refer to Fig. 4(b)). We show that these intermediate outputs are actually useful and can be used to determine whether a SIEVEADN instance is redundant. Roughly speaking, if outputs of two SIEVEADN instances are close to each other, it is not necessary to maintain both of them because one of them is redundant; hence, we can terminate one of them earlier. In this way, because we maintain less than L SIEVEADN instances, the update time and memory usage both decrease. On the other hand, early terminations of SIEVEADN instances will incur a loss in solution quality. We will show how to bound this loss by using the *smooth submodular histogram* property [24,29].

Specifically, let $g_t(l)$ denote the value of output of instance $\mathcal{A}_l^{(t)}$ at time t. We know that $g_t(1) \geq (1/2 - \epsilon)\mathrm{OPT}_t$ at any time t. Instead of maintaining L SIEVEADN instances, we propose HISTAPPROX that removes redundant SIEVEADN instances whose output values are close to an *active* SIEVEADN instance. HISTAPPROX can be viewed as using a histogram $\{g_t(l): l \in \mathbf{x}_t\}$ to approximate $g_t(l)$, as illustrated in Fig. 5. Here, $\mathbf{x}_t = \{x_1^{(t)}, x_2^{(t)}, \ldots\}$ is a set of indices in descending order, and each index $x_i^{(t)} \in \{1, \ldots, L\}$ indexes an *active* SIEVEADN instance[2]. Finally, HISTAPPROX approximates $g_t(1)$ by $g_t(x_1)$ at any time t. Because HISTAPPROX only maintains a few SIEVEADN instances at any time t, i.e., those $\mathcal{A}_l^{(t)}$'s with $l \in \mathbf{x}_t$ and $|\mathbf{x}_t| \leq L$, HISTAPPROX reduces both update time and memory usage with a little loss in solution quality which can still be bounded.

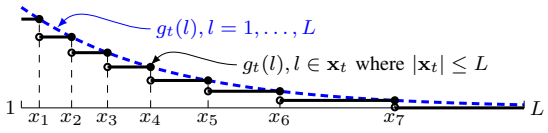


Fig. 5. Approximating $\{g_t(l): l = 1, \ldots, L\}$ by $\{g_t(l): l \in \mathbf{x}_t\}$. (Note that $g_t(l)$ may not be a monotone function of l.)

[2]We will omit superscript t if time t is clear from context.

B. Algorithm Description

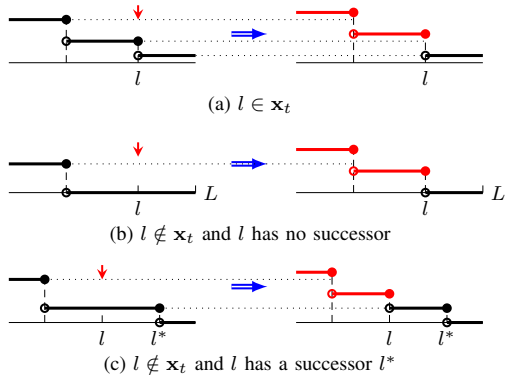
HISTAPPROX mainly consists of two steps: 1) creating and updating SIEVEADN instances; and 2) removing redundant SIEVEADN instances.

Creating and Updating SIEVEADN Instances. Consider a set of edges $\bar{E}_l^{(t)}$ with lifetime l arrived at time t. If $l \in \mathbf{x}_t$, we simply feed $\bar{E}_l^{(t)}$ to $\{\mathcal{A}_i^{(t)} : i \in \mathbf{x}_t \wedge i \leq l\}$ (as illustrated in Fig. 6(a)). Otherwise, we need to create a new SIEVEADN instance $\mathcal{A}_l^{(t)}$ first, and then feed $\bar{E}_l^{(t)}$ to $\{\mathcal{A}_i^{(t)} : i \in \mathbf{x}_t \wedge i \leq l\}$. There are two cases when creating $\mathcal{A}_l^{(t)}$.

(1) If l has no successor in $\mathbf{x}_t$, as illustrated in Fig. 6(b), we simply create a new SIEVEADN instance as $\mathcal{A}_l^{(t)}$.

(2) Otherwise, let l^* denote l's successor in $\mathbf{x}_t$, as illustrated in Fig. 6(c). Let $\mathcal{A}_l^{(t)}$ denote a copy of $\mathcal{A}_{l^*}^{(t)}$. Recall that $\mathcal{A}_l^{(t)}$ needs to process edges with lifetime no less than l at time t. Because $\mathcal{A}_{l^*}^{(t)}$ has processed edges with lifetime no less than l^*, then, we only need to feed $\mathcal{A}_l^{(t)}$ with edges in G_t such that their lifetimes are in interval $[l, l^*)$.

After this step, we guarantee that each active SIEVEADN instances processed the edges that they should process.



(a) $l \in \mathbf{x}_t$

(b) $l \notin \mathbf{x}_t$ and l has no successor

(c) $l \notin \mathbf{x}_t$ and l has a successor l^*

Fig. 6. Processing arrived edges with lifetime l

Removing Redundant SIEVEADN Instances. Intuitively, if the outputs of two SIEVEADN instances are close to each other, there is no need to maintain both of them because one of them is redundant. To quantify the redundancy of SIEVEADN instances, we need the following formal definition.

Definition 4 (ϵ-redundancy)**.** *At time t, consider two* SIEVEADN *instances $\mathcal{A}_i^{(t)}$ and $\mathcal{A}_l^{(t)}$ with $i < l$. We say $\mathcal{A}_l^{(t)}$ is ϵ-redundant with $\mathcal{A}_i^{(t)}$ if their exists $j > l$ such that $g_t(j) \geq (1-\epsilon)g_t(i)$.*

The above definition simply states that, since $\mathcal{A}_i^{(t)}$ and $\mathcal{A}_j^{(t)}$ are already close with each other, then the SIEVEADN instances between them are considered to be redundant and it is not necessary to maintain them. In HISTAPPROX, after processing each batch of edges, we check the outputs of each SIEVEADN instances and terminate those redundant ones.

The complete HISTAPPROX is given in Alg. 3. Pseudo-codes of the two steps are described in `ProcessEdges` and `ReduceRedundancy`, respectively.

Alg. 3: HISTAPPROX

Input: A sequence of edges arriving over time
Output: A set of influential nodes at each time t

$\mathbf{x}_1 \leftarrow \emptyset$;
for $t = 1, 2, \ldots$ **do**
  foreach $l = 1, \ldots, L$ **do** `ProcessEdges` $(\bar{E}_l^{(t)})$;
  $S_t \leftarrow$ output of $\mathcal{A}_{x_1}^{(t)}$;
  if $x_1 = 1$ **then** Terminate $\mathcal{A}_1^{(t)}$, let $\mathbf{x}_t \leftarrow \mathbf{x}_t \backslash \{1\}$;
  for $i = 1, \ldots, |\mathbf{x}_t|$ **do**
    $\mathcal{A}_{x_i - 1}^{(t+1)} \leftarrow \mathcal{A}_{x_i}^{(t)}$, $x_i^{(t+1)} \leftarrow x_i^{(t)} - 1$;
Procedure `ProcessEdges` $(\bar{E}_l^{(t)})$
  if $l \notin \mathbf{x}_t$ **then**
    if *l has no successor in $\mathbf{x}_t$* **then** // refer to Fig. 6(b)
      Create and initialize $\mathcal{A}_l^{(t)}$;
    else // refer to Fig. 6(c)
      Let l^* denote the successor of l in $\mathbf{x}_t$;
      $\mathcal{A}_l^{(t)} \leftarrow$ a copy of $\mathcal{A}_{l^*}^{(t)}$;
      Feed $\mathcal{A}_l^{(t)}$ with edges $\{e : e \in E_t \wedge l \leq l_e < l^*\}$.
    $\mathbf{x}_t \leftarrow \mathbf{x}_t \cup \{l\}$;
  foreach $l' \in \mathbf{x}_t$ *and* $l' \leq l$ **do** Feed $\mathcal{A}_{l'}^{(t)}$ with $\bar{E}_l^{(t)}$;
  `ReduceRedundancy()`;
Procedure `ReduceRedundancy()`
  foreach $i \in \mathbf{x}_t$ **do**
    Find the largest $j > i$ in $\mathbf{x}_t$ s.t. $g_t(j) \geq (1-\epsilon)g_t(i)$;
    Delete each index $l \in \mathbf{x}_t$ s.t. $i < l < j$ and kill $\mathcal{A}_l^{(t)}$;

C. Algorithm Analysis

We now theoretically show that HISTAPPROX achieves a constant approximation factor.

Notice that indices $x \in \mathbf{x}_t$ and $x + 1 \in \mathbf{x}_{t-1}$ are actually the same index but appear at different time. In general, we say $x' \in \mathbf{x}_{t'}$ is an *ancestor* of $x \in \mathbf{x}_t$ if $t' \leq t$ and $x' = x + t - t'$. An index and its ancestors will be considered as the same index. We will use x' to denote x's ancestor in the follows.

First, histogram $\{g_t(l) : l \in \mathbf{x}_t\}$ maintained by HISTAPPROX has the following property.

Theorem 6. *For two consecutive indices $x_i, x_{i+1} \in \mathbf{x}_t$ at any time t, one of the following two cases holds:*

- **C1** *G_t contains no edge with lifetime in interval (x_i, x_{i+1}).*
- **C2** *$g_{t'}(x'_{i+1}) \geq (1-\epsilon)g_{t'}(x'_i)$ at some time $t' \leq t$, and from time t' to t, there is no edge with lifetime between x'_i and x'_{i+1} arrived, exclusive.*

Proof. Please refer to our Technique Report [28]. □

Histogram with property C2 is known as a *smooth histogram* [23]. Smooth histogram together with the submodularity of objective function can ensure a constant factor approximation factor of $g_t(x_1)$.

Theorem 7. HISTAPPROX *achieves a $(1/3-\epsilon)$ approximation guarantee, i.e., $g_t(x_1) \geq (1/3-\epsilon)\mathrm{OPT}_t$ at any time t.*

Proof. The high-level idea is to leverage the property found in Theorem 6 and *smooth submodular histogram* property

reported in [24,29]. Please refer to our Technique Report [28] for details. □

HISTAPPROX also significantly reduces the complexity.

Theorem 8. HISTAPPROX *uses* $O(b(\gamma+1)\epsilon^{-2}\log^2 k)$ *time to process each batch of edges and* $O(k\epsilon^{-2}\log^2 k)$ *memory to store intermediate results.*

Proof. Please refer to our Technique Report [28]. □

Remark. Can HISTAPPROX achieve the $(1/2-\epsilon)$ approximation guarantee? Notice that HISTAPPROX outputs slightly worse solution than BASICREDUCTION due to the fact that $\mathcal{A}_{x_1}^{(t)}$ did not process all of the edges in G_t, i.e., those edges with lifetime less than x_1 in G_t are not processed. Therefore, we can slightly modify Alg. 3 and feed $\mathcal{A}_{x_1}^{(t)}$ with these unprocessed edges. Then, HISTAPPROX will output a solution with $(1/2-\epsilon)$ approximation guarantee. In practice, we observe that $g_t(x_1)$ is already close to OPT_t, hence it is not necessary to conduct this additional processing. More discussion on further improving the efficiency of HISTAPPROX can be found in our technical report [28].

V. EXPERIMENTS

In this section, we validate the performance of our methods on various real-world interaction datasets. A summary of these datasets is given in Table I.

TABLE I
SUMMARY OF INTERACTION DATASETS

interaction dataset	# of nodes	# of interactions
Brightkite (users/places)	51,406/772,966	4,747,281
Gowalla (users/places)	107,092/1,280,969	6,442,892
Twitter-Higgs	304,198	555,481
Twitter-HK	49,808	2,930,439
StackOverflow-c2q	1,627,635	13,664,641
StackOverflow-c2a	1,639,761	17,535,031

A. Interaction Datasets

• **LBSN Check-in Interactions** [30]. Brightkite and Gowalla are two location based online social networks (LBSNs) where users can check in places. A check-in record is viewed as an interaction between a user and a place. If user u checked in a place y at time t, we denote this interaction by $\langle y,u,t\rangle$. Because $\langle y,u,t\rangle$ implies that place y attracts user u to check in, it thus reflects y's influence on u. In this particular example, a place's influence (or *popularity*), is equivalent to the number of distinct users who checked in the place. Our goal is to maintain k most popular places at any time.

• **Twitter Re-tweet/Mention Interactions** [30,31]. In Twitter, a user v can re-tweet another user u's tweet, or mention another user u (i.e., @u). We denote this interaction by $\langle u,v,t\rangle$, which reflects u's influence on v at time t. We use two Twitter re-tweet/mention interaction datasets Twitter-Higgs and Twitter-HK. Twitter-Higgs dataset is built after monitoring the tweets related to the announcement of the discovery of a new particle with the features of the elusive Higgs boson on 4th July 2012. The detailed description of this dataset is given in [30]. Twitter-HK dataset is built after monitoring the tweets related to Umbrella Movement happened in Hong Kong from September to December in 2014. The detailed collection method and description of this dataset is given in [31]. Our goal is to maintain k most influential users at any time.

• **Stack Overflow Interactions** [30]. Stack Overflow is an online question and answer website where users can ask questions, give answers, and comment on questions/answers. We use the comment on question interactions (StackOverflow-c2q) and comment on answer interactions (StackOverflow-c2a). In StackOverflow-c2q, if user v comments on user u's question at time t, we create an interaction $\langle u,v,t\rangle$ (which reflects u's influence on v because u attracts v to answer his question). Similarly, in StackOverflow-c2a, if user v comments on user u's answer at time t, we create an interaction $\langle u,v,t\rangle$. Our goal is to maintain k most influential users at any time.

B. Settings

We assign each interaction a lifetime sampled from a geometric distribution $Pr(l_\tau(e)=l)\propto(1-p)^{l-1}p$ truncated at the maximum lifetime L. As discussed in Example 5, this particular lifetime assignment actually means that we forget each existing interaction with probability p. Here p controls the skewness of the distribution, i.e., for larger p, more interactions tend to have short lifetimes. We emphasize that other lifetime assignment methods are also allowed in our framework.

Each interaction will be input to our algorithms sequentially according to their timestamps and we assume one interaction arrives at a time.

Note that in the following experiments, all of our algorithms are **implemented in series**, on a laptop with a 2.66GHz Intel Core I3 CPU running Linux Mint 19. We refer to our technical report [28] for more discussion on implementation details.

C. Baselines

• **Greedy** [27]. We run a greedy algorithm on G_t which chooses a node with the maximum marginal gain in each round, and repeats k rounds. We apply the *lazy evaluation* trick [32] to further reduce the number of oracle calls[3].

• **Random.** We randomly pick a set of k nodes from G_t at each time t.

• **DIM**[4] [17]. DIM updates the index (or called sketch) dynamically and supports handling fully dynamical networks, i.e., edges arrive, disappear, or change the diffusion probabilities. We set the parameter $\beta=32$ as suggested in [17].

• **IMM**[5] [6]. IMM is an index-based method that uses martingales, and it is designed for handling *static graphs*. We set the parameter $\epsilon=0.3$.

• **TIM+**[6] [4]. TIM+ is an index-based method with the two-phase strategy, and it is designed for handling *static graphs*. We set the parameter $\epsilon=0.3$.

[3] An *oracle call* refers to an evaluation of f_t.
[4] https://github.com/todo314/dynamic-influence-analysis
[5] https://sourceforge.net/projects/im-imm/
[6] https://sourceforge.net/projects/timplus/

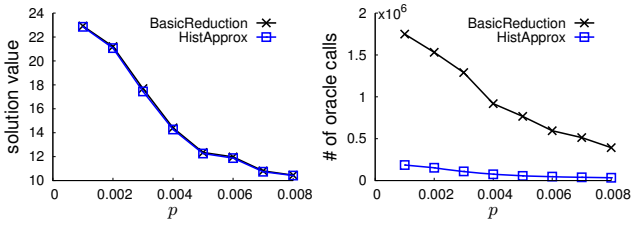
Note that the first five methods can be used to address our problem (even though some of them are obviously not efficient); while SIM cannot be used to address our general problem as it is only designed for the sliding-window model. Some of the above methods assume that the diffusion probabilities on each edge in G_t are given in advance. Strictly, these probabilities should be learned use complex inference methods [10]–[12], which will further harm the efficiency of existing methods. Here, for simplicity, if node u imposed x interactions on node v at time t, we assign edge (u, v) a diffusion probability $p_{uv} = 2/(1 + \exp(-0.2x)) - 1$.

When evaluating computational efficiency, we prefer to use the *number of oracle calls*. Because an oracle call is the most expensive operation in each algorithm, and, more importantly, the number of oracle calls is independent of algorithm implementation (e.g., parallel or series) and experimental hardwares. The less the number of oracle calls an algorithm uses, the faster the algorithm is.

D. Results

Comparing BASICREDUCTION with HISTAPPROX. We first compare the performance of HISTAPPROX and BASICREDUCTION. Our aim is to understand how lifetime distribution affects the efficiency of BASICREDUCTION, and how significantly HISTAPPROX improves the efficiency upon BASICREDUCTION.

We run the two methods on the two LBSN datasets for 5000 time steps, and maintain $k = 10$ places at each time step. We vary p from 0.001 to 0.008. For each p, we average the solution value and number of oracle calls along time, and show the results in Fig. 7.

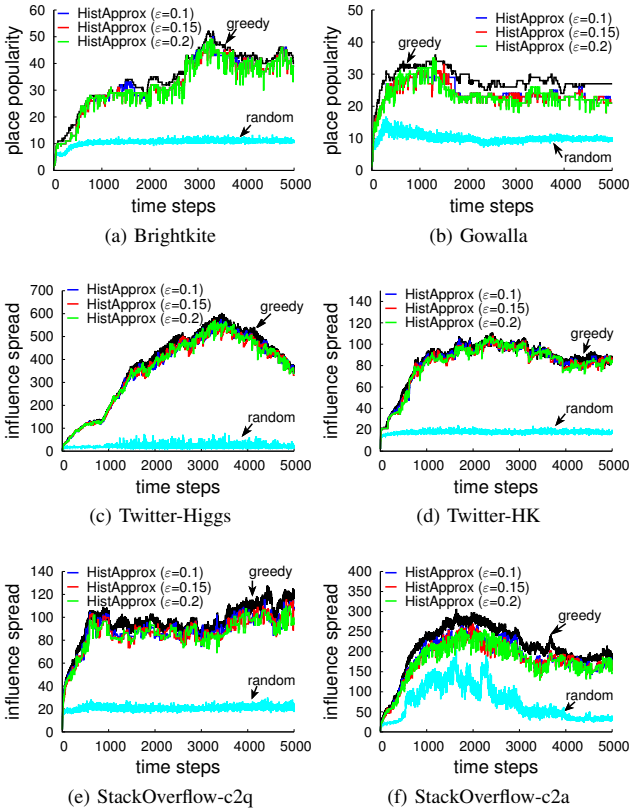


(a) Solution value (Brightkite) (b) Efficiency (Brightkite)

(c) Solution value (Gowalla) (d) Efficiency (Gowalla)

Fig. 7. Comparing BASICREDUCTION with HISTAPPROX. ($\epsilon = 0.1, k = 10, L = 1000$, each point is averaged over 5000 time steps)

First, we observe that the solution value of HISTAPPROX is very close to BASICREDUCTION. The solution value ratio



(a) Brightkite (b) Gowalla

(c) Twitter-Higgs (d) Twitter-HK

(e) StackOverflow-c2q (f) StackOverflow-c2a

Fig. 8. Solution value over time (higher is better). $k = 10, L = 10K$

of HISTAPPROX to BASICREDUCTION is larger than 0.98. Second, we observe that the number of oracle calls of BASICREDUCTION decreases as p increases, i.e., more interactions tend to short lifetimes. This observation consists with our analysis that processing edges with long lifetimes is the main bottleneck of BASICREDUCTION. Third, HISTAPPROX needs much less oracle calls than BASICREDUCTION, and the ratio of the number of oracle calls of HISTAPPROX to BASICREDUCTION is less than 0.1.

This experiment demonstrates that HISTAPPROX outputs solutions with value very close to BASICREDUCTION but is much efficient than BASICREDUCTION.

Evaluating Solution Quality of HISTAPPROX Over Time. We conduct more in-depth analysis of HISTAPPROX in the following experiments. First, we evaluate the solution quality of HISTAPPROX in comparison with other baseline methods.

We run HISTAPPROX with $\epsilon = 0.1, 0.15$ and 0.2 for 5000 time steps on the six datasets, respectively, and maintain $k = 10$ nodes at each time step. We compare the solution value of HISTAPPROX with Greedy and Random. The results are depicted in Fig. 8.

We observe that on all of the six datasets, Greedy achieves the highest solution value, and Random achieves the lowest. The solution value of HISTAPPROX is very close to Greedy, and is much better than Random. To clearly judge ϵ's effect

on solution quality, we calculate the ratio of solution value of HISTAPPROX to Greedy, and average the ratios along time, and show the results in Fig. 9. It is clear to see that when ϵ increases, the solution value decreases in general.

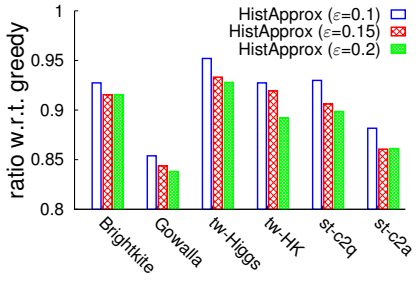


Fig. 9. Ratio of solution value averaged along time (higher is better)

These experiments demonstrate that HISTAPPROX achieves comparable solution quality with Greedy, and smaller ϵ helps improving HISTAPPROX's solution quality.

Evaluating Computational Efficiency of HISTAPPROX Over Time. Next we compare HISTAPPROX's computational efficiency with Greedy. When calculating the number of oracle calls of Greedy, we use the lazy evaluation [32] trick, which is a useful heuristic to reduce Greedy's number of oracle calls.

We run HISTAPPROX with $\epsilon = 0.1, 0.15$ and 0.2 on the six datasets, respectively, and calculate the ratio of cumulative number of oracle calls at each time step of HISTAPPROX to Greedy. The results are depicted in Fig. 10.

On all of the six datasets, HISTAPPROX uses much less oracle calls than Greedy. When ϵ increases, the number of oracle calls of HISTAPPROX decreases further. For example, with $\epsilon = 0.2$, HISTAPPROX uses 5 to 15 times less oracle calls than Greedy.

This experiment demonstrates that HISTAPPROX is more efficient than Greedy. Combining with the previous results, we conclude that ϵ can trade off between solution quality and computational efficiency: larger ϵ makes HISTAPPROX run faster but also decreases solution quality.

Effects of Parameter k and L. We evaluate HISTAPPROX's performance using different budgets $k = 10, \ldots, 100$ and lifetime upper bounds $L = 10K, \ldots, 100K$. For each budget k (and L), we run HISTAPPROX for 5000 time steps, and calculate two ratios: (1) the ratio of HISTAPPROX's solution value to Greedy's solution value; and (2) the ratio of HISTAPPROX's number of oracle calls to Greedy's number of oracle calls. We average these ratios along time, and show the results in Figs. 11 and 12.

We find that under different parameter settings, HISTAPPROX's performance consists with our previous observations. In addition, we find that, for larger k, the efficiency of HISTAPPROX improves more significantly than Greedy. This is due to the fact that HISTAPPROX's time complexity increases logarithmically with k; while Greedy's time complexity increases linearly with k. We also find that L does not affect HISTAPPROX's performance very much.

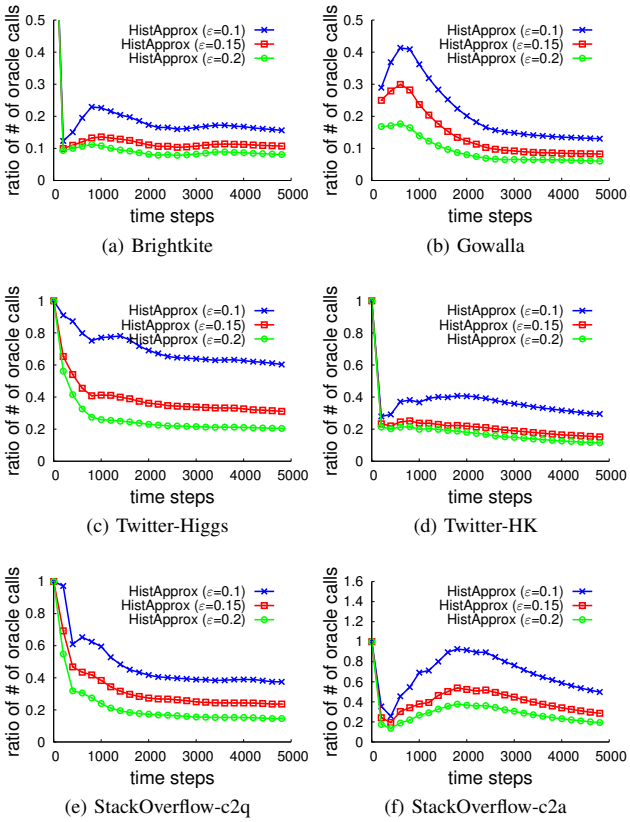


Fig. 10. Number of oracle calls ratio (lower is better). $k = 10, L = 10K$

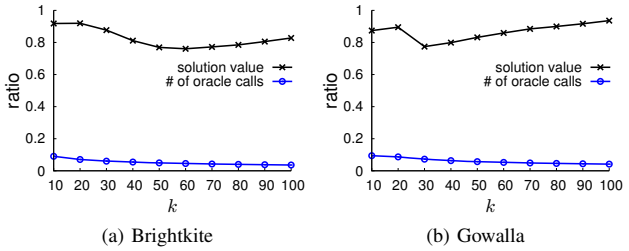
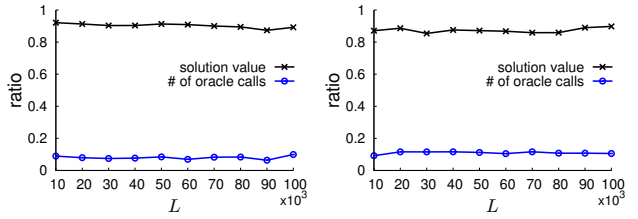


Fig. 11. Performance of HISTAPPROX w.r.t. different k. ($\epsilon = 0.2, L = 10K$. Each point is averaged over 5000 time steps.)

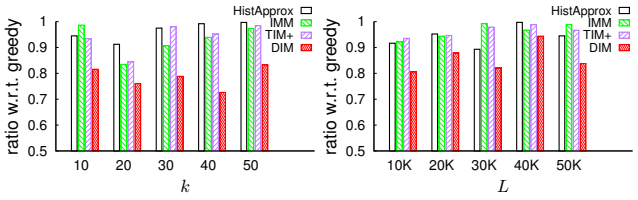
Performance and Throughput Comparisons. Next, we compare the solution quality and throughput with the other baseline methods. Since Greedy achieves the highest solution quality among all the methods, we use Greedy as a reference and show solution value ratio w.r.t. Greedy when evaluating solution quality. For throughput analysis, we will show the maximum stream processing speed (i.e., number of processed edges per second) using different methods.

We set budgets $k = 10, \ldots, 100$. Lifetimes are sampled from Geo(0.001) with upper bounds $L = 10K, \ldots, 100K$ respectively. We set $\epsilon = 0.3$ in HISTAPPROX. All of the algorithms are ran for $10,000$ time steps. For throughput analysis, we fix the budget to be $k = 10$. The results are

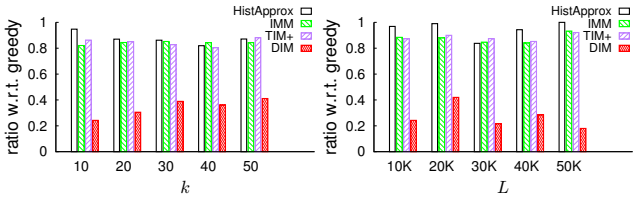


(a) Brightkite (b) Gowalla

Fig. 12. Performance of HISTAPPROX w.r.t. different L. ($\epsilon = 0.2, k = 10$. Each point is averaged over 5000 time steps.)

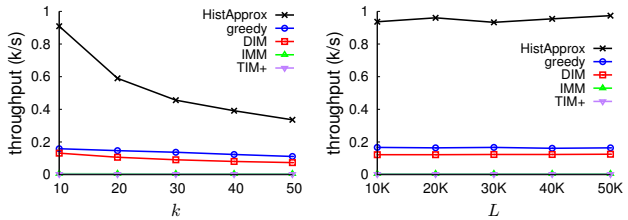


(a) Twitter-Higgs (b) Twitter-Higgs

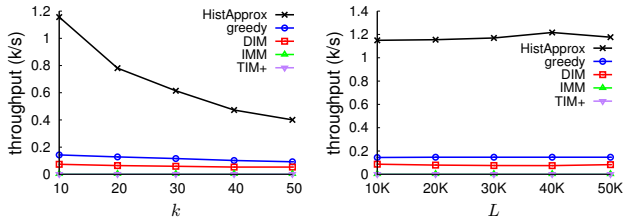


(c) StackOverflow-c2q (d) StackOverflow-c2q

Fig. 13. Solution quality comparison (higher is better)



(a) Twitter-Higgs (b) Twitter-Higgs



(c) StackOverflow-c2q (d) StackOverflow-c2q

Fig. 14. Throughput comparison (higher is better)

depicted in Figs. 13 and 14.

From Fig. 13, we observe that HISTAPPROX, IMM and TIM+ always find high quality solutions. In contrast, DIM seems not so stable, and it performs even worse on the StackOverflow-c2q dataset than on the Twitter-Higgs dataset. From Fig. 14, we observe that HISTAPPROX achieves the highest throughput than the other methods, then comes Greedy and DIM, the two methods IMM and TIM+ designed for static graphs have the lowest throughput (and they almost overlap with each other). Greedy using the lazy evaluation trick is faster than DIM, IMM, and TIM+ because the oracle call implementation in Greedy is much faster than oracle call implementations in the other three methods, which relay on Monte-Carlo simulations.

This experiment demonstrates that our proposed HISTAPPROX algorithm can find high quality solutions with high throughput.

VI. RELATED WORK

Influence Maximization (IM) over Dynamic Networks. IM and its variants on static networks have been extensively studied (e.g., [1]–[9], to name a few). Here we devote to review some literature related to IM on dynamic networks.

Aggarwal et al. [15] and Zhuang et al. [16] propose heuristic approaches to solve IM on dynamic networks. These heuristic approaches, however, have no theoretical guarantee on the quality of selected nodes.

Song et al. [19] use the interchange greedy approach [27] to solve IM over a sequence of networks with the assumption that these networks change smoothly. The interchange greedy approach updates current influential nodes based on influential nodes found in previous network and avoids constructing the solution from an empty set. This approach has an approximation factor $(1/2 - \epsilon)$. However, the interchange greedy approach degrades to running from scratch if these networks do not change smoothly. This limits its application on highly dynamic networks.

Both [17] and [18] extend the reverse-reachable (RR) sets [2] to updatable index structures for faster computing the influence of a node or a set of nodes in dynamic networks. In particular, [18] considers finding influential individuals in dynamic networks, which is actually a different problem. It is worth noting that RR-sets are approximation methods designed to speed up the computation of the influence of a set of nodes under the IC or LT model, which is #P-hard; while our approach is a data-driven approach without any assumption of influence spreading model.

There are also some variants of IM on dynamic networks such as topic-specific influencers query [33,34] and online learning approach for IM [35].

Streaming Submodular Optimization (SSO) Methods. SSO over insertion-only streams is first studied in [36] and the state-of-the-art approach is the threshold based SIEVESTREAMING algorithm [26] which has an approximation factor $(1/2 - \epsilon)$.

The sliding-window stream model is first introduced in [22]

for maintaining summation aggregates on a data stream (e.g., counting the number of 1's in a 0-1 stream). Chen et al. [29] leverage the smooth histogram technique [23] to solve SSO under the sliding-window stream, and achieves a $(1/4-\epsilon)$ approximation factor. Epasto et al. [24] further improve the approximation factor to $(1/3-\epsilon)$.

Note that our problem requires solving SSO over time-decaying streams, which is more general than sliding-window streams. To the best of our knowledge, there is no previous art that can solve SSO on time-decaying streams.

Recently, [25] leveraged the results of [29] to develop a streaming method to solve IM over sliding-window streams. The best approximation factor of their framework is $(1/4-\epsilon)$. In our work, we consider the more general time-decaying stream, and our approach can achieve the $(1/2-\epsilon)$ approximation factor.

Maintaining Time-Decaying Stream Aggregates. Cohen et al. [37] first extend the sliding-window model in [22] to the general time-decaying model for the purpose of approximating summation aggregates in data streams. Cormode et al. [38] consider the similar problem by designing time-decaying sketches. These studies have inspired us to consider the more general time-decaying streams.

VII. Conclusion

This work studied the problem of identifying influential nodes from highly dynamic node interaction streams. Our methods are based on a general TDN model which allows discarding outdated data in a smooth manner. We designed three algorithms, i.e., SieveADN, BasicReduction, and HistApprox. SieveADN identifies influential nodes over ADNs. BasicReduction uses SieveADN as a basic building block to identify influential nodes over TDNs. HistApprox significantly improves the efficiency of BasicReduction. We conducted extensive experiments on real interaction datasets and the results demonstrated the efficiency and effectiveness of our methods.

Acknowledgment

We would like to thank the anonymous reviewers for their valuable comments and suggestions to help us improve this paper. This work is financially supported by the King Abdullah University of Science and Technology (KAUST) Sensor Initiative, Saudi Arabia. The work of John C.S. Lui was supported in part by the GRF Funding 14208816.

References

[1] D. Kempe, J. Kleinberg, and E. Tardos, "Maximizing the spread of influence through a social network," in *KDD*, 2003.
[2] C. Borgs, M. Brautbar, J. Chayes, and B. Lucier, "Maximizing social influence in nearly optimal time," in *SODA*, 2014.
[3] W. Chen, C. Wang, and Y. Wang, "Scalable influence maximization for prevalent viral marketing in large-scale social networks," in *KDD*, 2010.
[4] Y. Tang, X. Xiao, and Y. Shi, "Influence maximization: Near-optimal time complexity meets practical efficiency," in *SIGMOD*, 2014.
[5] B. Lucier, J. Oren, and Y. Singer, "Influence at scale: Distributed computation of complex contagion in networks," in *KDD*, 2015.
[6] Y. Tang, Y. Shi, and X. Xiao, "Influence maximization in near-linear time: A martingale approach," in *KDD*, 2015.
[7] W. Chen, T. Lin, Z. Tan, M. Zhao, and X. Zhou, "Robust influence maximization," in *KDD*, 2016.
[8] I. Litou, V. Kalogeraki, and D. Gunopulos, "Influence maximization in a many cascades world," in *ICDCS*, 2017.
[9] Y. Lin, W. Chen, and J. C. Lui, "Boosting information spread: An algorithmic approach," in *ICDE*, 2017.
[10] K. Saito, R. Nakano, and M. Kimura, "Prediction of information diffusion probabilities for independent cascade model," in *KES*, 2008.
[11] A. Goyal, F. Bonchi, and L. V. S. Lakshmanan, "Learning influence probabilities in social networks," in *WSDM*, 2010.
[12] K. Kutzkov, A. Bifet, F. Bonchi, and A. Gionis, "STRIP: Stream learning of influence probabilities," in *KDD*, 2013.
[13] S. Myers and J. Leskovec, "The bursty dynamics of the Twitter information network," in *WWW*, 2014.
[14] S. Myers, C. Zhu, and J. Leskovec, "Information diffusion and external influence in networks," in *KDD*, 2012.
[15] C. Aggarwal, S. Lin, and P. S. Yu, "On influential node discovery in dynamic social networks," in *SDM*, 2012.
[16] H. Zhuang, Y. Sun, J. Tang, J. Zhang, and X. Sun, "Influence maximization in dynamic social networks," in *ICDM*, 2013.
[17] N. Ohsaka, T. Akiba, Y. Yoshida, and K. Kawarabayashi, "Dynamic influence analysis in evolving networks," in *VLDB*, 2016.
[18] Y. Yang, Z. Wang, J. Pei, and E. Chen, "Tracking influential individuals in dynamic networks," *TKDE*, vol. 29, no. 11, pp. 2615–2628, 2017.
[19] G. Song, Y. Li, X. Chen, X. He, and J. Tang, "Influential node tracking on dynamic social network: An interchange greedy approach," *TKDE*, vol. 29, no. 2, pp. 359–372, 2017.
[20] R. Xiang, J. Neville, and M. Rogati, "Modeling relationship strength in online social networks," in *WWW*, 2010.
[21] N. Alon, Y. Matias, and M. Szegedy, "The space complexity of approximating the frequency moments," *Journal of Computer and System Sciences*, vol. 58, no. 1, pp. 137–147, 1999.
[22] M. Datar, A. Gionis, P. Indyk, and R. Motwani, "Maintaining stream statistics over sliding windows," *SIAM Journal on Computing*, vol. 31, no. 6, pp. 1794–1813, 2002.
[23] V. Braverman and R. Ostrovsky, "Smooth histograms for sliding windows," in *FOCS*, 2007.
[24] A. Epasto, S. Lattanzi, S. Vassilvitskii, and M. Zadimoghaddam, "Submodular optimization over sliding windows," in *WWW*, 2017.
[25] Y. Wang, Q. Fan, Y. Li, and K.-L. Tan, "Real-time influence maximization on dynamic social streams," in *VLDB*, 2017.
[26] A. Badanidiyuru, B. Mirzasoleiman, A. Karbasi, and A. Krause, "Streaming submodular maximization: Massive data summarization on the fly," in *KDD*, 2014.
[27] G. Nemhauser, L. Wolsey, and M. Fisher, "An analysis of approximations for maximizing submodular set functions - i," *Mathematical Programming*, vol. 14, pp. 265–294, 1978.
[28] "TechReport," https://arxiv.org/pdf/1810.07917.pdf.
[29] J. Chen, H. L. Nguyen, and Q. Zhang, "Submodular maximization over sliding windows," in *arXiv:1611.00129*, 2016.
[30] "Stanford network dataset," http://snap.stanford.edu/data.
[31] J. Zhao, J. C. Lui, D. Towsley, P. Wang, and X. Guan, "Tracking triadic cardinality distributions for burst detection in social activity streams," in *COSN*, 2015.
[32] M. Minoux, "Accelerated greedy algorithms for maximizing submodular set functions," *Optimization Techniques*, vol. 7, pp. 234–243, 1978.
[33] K. Subbian, C. Aggarwal, and J. Srivastava, "Content-centric flow mining for influence analysis in social streams," in *CIKM*, 2013.
[34] K. Subbian, C. C. Aggarwal, and J. Srivastava, "Querying and tracking influencers in social streams," in *WSDM*, 2016.
[35] S. Lei, S. Maniu, L. Mo, R. Cheng, and P. Senellart, "Online influence maximization," in *KDD*, 2015.
[36] R. Kumar, B. Moseley, S. Vassilvitskii, and A. Vattani, "Fast greedy algorithms in MapReduce and streaming," in *SPAA*, 2013.
[37] E. Cohen and M. J. Strauss, "Maintaining time-decaying stream aggregates," *Journal of Algorithms*, vol. 59, pp. 19–36, 2006.
[38] G. Cormode, S. Tirthapura, and B. Xu, "Time-decaying sketches for robust aggregation of sensor data," *SIAM Journal on Computing*, vol. 39, no. 4, pp. 1309–1339, 2009.