Dynamic Influence Analysis in Evolving Networks

Naoto Ohsaka [*#1], Takuya Akiba [†‡#2], Yuichi Yoshida [†§3], Ken-ichi Kawarabayashi [†#4]
[*] The University of Tokyo, [†] National Institute of Informatics, [‡] JST, PRESTO
[§] Preferred Infrastructure, Inc., [#] JST, ERATO, Kawarabayashi Large Graph Project
ohsaka@is.s.u-tokyo.ac.jp[1], {takiba[2], yyoshida[3], k_keniti[4]}@nii.ac.jp

ABSTRACT

We propose the first real-time fully-dynamic index data structure designed for influence analysis on evolving networks. With this aim, we carefully redesign the data structure of the state-of-the-art sketching method introduced by Borgs *et al.*, and construct corresponding update algorithms. Using this index, we present algorithms for two kinds of queries, *influence estimation* and *influence maximization*, which are strongly motivated by practical applications, such as viral marketing. We provide a thorough theoretical analysis, which guarantees the non-degeneracy of the solution accuracy after an arbitrary number of updates. Furthermore, we introduce a *reachability-tree-based technique* and a *skipping method*, which greatly reduce the time consumption required for edge/vertex deletions and vertex additions, respectively, and *counter-based random number generators*, which improve the space efficiency.

Experimental evaluations using real dynamic networks with tens of millions of edges demonstrate the efficiency, scalability, and accuracy of our proposed indexing scheme. Specifically, it can reflect a graph modification within a time of several orders of magnitude smaller than that required to reconstruct an index from scratch, estimate the influence spread of a vertex set accurately within a millisecond, and select highly influential vertices at least ten times faster than state-of-the-art static algorithms.

1. INTRODUCTION

Recently, the increasing popularity of online social networks has resulted in the opportunity to analyze the spread of product adoption, ideas, and news through the population, and exploit the results. A considerable amount of work has focused on the analysis of the diffusion process [1, 2, 22], the prediction of future diffusions [7], learning the strength of influence between a pair of individuals [13, 26], estimating the influence spread [10, 20], and finding a small seed set of individuals to maximize the spread of influence, which has been termed *influence maximization* [15, 19, 24, 29]. One of the central elements in such work is the mathematical formalization of stochastic cascade models for diffusion processes. Therefore, the efficient simulation and estimations of influence in these models is essential to such studies.

However, in reality, social networks have a highly dynamic nature, and evolve rapidly over time [17, 18]. Consequently, results concerning the estimation and maximization of influence can quickly become outdated. Because such diffusion models involve complicated stochastic processes, even with state-of-the-art scalable algorithms, to re-run them on modern massive networks from scratch requires a non-negligible computational cost every time.

In this study, we address this issue by developing the first fully-dynamic index data structure for analyzing influence in a large time-evolving network. Our index is *fully dynamic*, i.e., it can instantly incorporate graph updates of any kind, including the addition and deletion of vertices and edges, and propagation probability updates. By applying this updated index, two kinds of queries for diffusion analysis, *influence estimation* and *influence maximization*, can be efficiently answered on the latest graph snapshot. These are described below.

1.1 Influence estimation

The first query type that we consider is *influence estimation*. Given a seed set of vertices, we can use our index to quickly estimate the expected number of vertices that would be influenced if the seed set is activated. Despite the fact that an exact computation of the answer is a #P-hard problem (under the independent cascade model), our index enables us to obtain an estimate in under a millisecond, where the error is theoretically bounded and empirically small.

This type of query is useful for evaluating, comparing, and identifying influential people or groups on the latest snapshot of a dynamic network. Moreover, we can keep track of the transition of the influence spread for a person or group. For example, Figure 1 illustrates the transition of the influences of vertices in a real-world evolving network of Flixster dataset.[1] We observe that several vertices have critical times at which their influences rapidly grow or decline.

1.2 Influence maximization

Another type of query that can be efficiently answered using our index is *influence maximization*. Given an integer k, this returns the seed vertex set of size k comprising the most influential vertices on the latest snapshot. This query is motivated by the recent cost-effective marketing strategy called

[1] http://www.cs.ubc.ca/~jamalim/datasets/

Proceedings of the VLDB Endowment, Vol. 9, No. 12

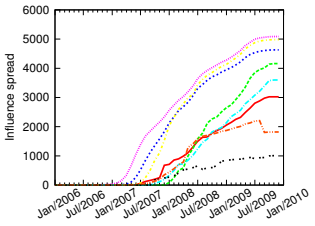


Figure 1: Transition of the influence spread of popular vertices in a real-world network.

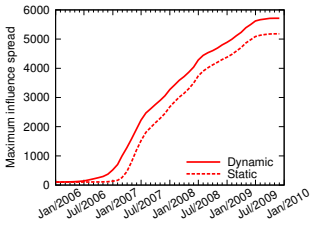


Figure 2: Transition of the approximated maximum influence spread of a seed set of size 100.

viral marketing, where products are promoted by presenting free or discounted items to a selected group of highly influential individuals, with the aim of achieving a large number of product adoptions through "word-of-mouth" effects [9, 25]. Owing to the success and growth of viral marketing, the influence maximization problem on static graphs has been intensively studied [3–5, 8, 14–16, 19, 24, 28–30]. However, in contrast to these methods, which compute the answer from scratch for each query, we can quickly obtain the answer on the latest snapshot using our dynamic index.

Figure 2 illustrates the transition of the (approximated) maximum influence spread of a seed vertex set of size 100 computed by our method using the real evolving network of Flixster dataset. Remarkable difference can be observed between the static setting (i.e., the outdated solution) and the dynamic setting (i.e., the updated solution), which suggests the usefulness of dynamic methods.

1.3 Our solution

The key idea underlying our proposed index is to focus the intermediate data structures that are used in state-of-the-art influence maximization algorithms [3, 28, 29]. Conceptually, we can say that whereas these are thrown away after each computation in such algorithms, we maintain and reuse them after their construction. However, as the original data structure itself does not contain sufficient information for an efficient and correct dynamic update, our main challenge is the careful redesign of the data structure. We examine the necessary information that should be stored, and prove the non-degeneracy of our estimation accuracy after an arbitrary number of updates. Moreover, we further improve the update time by maintaining *reachability trees* and employing a *skipping method*, and the space efficiency by using *counter-based random number generators*.

Through our experiments using real dynamic networks with tens of millions of edges, we confirm the scalability, efficiency, and accuracy of the proposed index. It can update the index within a time of several orders of magnitude smaller than that required to reconstruct an index from scratch. Our speed-up techniques for dynamic updates achieve several orders of magnitude improvements over naive implementations, and the index size is reduced by a factor of up to 50. Furthermore, using our index we can obtain an accurate estimation of the influence spread within a millisecond, and select highly influential vertices at least ten times faster than state-of-the-art static algorithms.

The contributions of this study are summarized as follows.

- We carefully redesign the data structure of the sketching method introduced by [3], present query algorithms for two kinds of queries, *influence estimation* and *influence maximization*, and construct update algorithms (Section 4).
- We provide a thorough theoretical analysis, which guarantees the non-degeneracy of our update algorithms and solution accuracy of our query algorithms (Section 5).
- We introduce several techniques for improving the performance of our indexing method, i.e., a *reachability-tree-based technique* for edge/vertex deletions, a *skipping method* for vertex additions, and *counter-based random number generators* for the space efficiency (Section 6).
- We experimentally demonstrate the scalability and efficiency of our indexing method, the effectiveness of the proposed techniques compared to naive implementations, and the accuracy and efficiency of our query algorithms for diffusion analysis (Section 7).

The remainder of this paper is organized as follows. In Section 2, we describe related work on influence analysis. In Section 3, we provide definitions, notations, and base algorithms to be used in this paper. Section 4 is devoted to explaining our dynamic indexing scheme. In Section 5, we present a theoretical analysis of our method. In Section 6, we introduce several techniques that further improve the performance of our method. We present our experimental results in Section 7, and our conclusions in Section 8.

2. RELATED WORK

2.1 Influence maximization and influence estimation in static networks

Motivated by the work of Domingos and Richardson [9, 25], Kempe *et al.* [15] formulated influence maximization as a discrete optimization problem, involving the extraction of a vertex set of a given size that maximizes the influence spread. They proved the NP-hardness of the influence maximization problem, and the monotonicity and submodularity of the influence spread function under two well-established information diffusion models, called the *independent cascade* and *linear threshold*. Thus, the greedy strategy guarantees an approximation ratio of $1 - 1/e$. The drawback of this strategy is that no efficient method to compute the influence spread is known, and this was later proven to be #P-hard [4]. Kempe *et al.* [15] resorted to using the Monte-Carlo simulation to approximate the influence spread, which is effective but too slow for large graphs. This motivates the development of efficient influence estimation methods that improve the scalability of Kempe *et al.*'s greedy algorithm. Here, we classify existing methods of influence maximization on static graphs, some of which can be also used to estimate the influence spread, into the following three types: simulation-based, heuristic-based, and sketch-based.

Simulation-based methods simulate the diffusion process repeatedly in order to accurately estimate the influence spread with a theoretical guarantee. To enhance the scalability, existing approaches have introduced several techniques for pruning unnecessary influence computations [19], as well as a sample average approximation approach [5, 8, 16, 24].

Heuristic-based methods avoid the use of Monte-Carlo simulations by restricting the spread of influence into communities [30], the most probable influence paths [4], and

other groups, or approximating the influence spread using linear systems [14]. Such heuristics are more scalable than simulation-based methods, but they often produce solutions of poor quality, owing to an absence of accuracy guarantees.

Sketch-based methods resolved the inefficiency of Monte-Carlo simulations without sacrificing accuracy guarantees. Instead of simulating the diffusion process directly, the method introduced by Borgs *et al.* [3], called *reverse influence sampling* (RIS), conducts reverse simulations to build sketches consisting a family of vertex sets to efficiently estimate the influence spread. RIS has been proven to run in nearly linear time, and to return a constant-factor approximate solution. Subsequently, several techniques for reducing the number of sketches have been developed [28, 29].

Despite such efforts, direct applications of these static methods for tracking the influence of a certain vertex or highly influential vertices in highly evolving networks are computationally expensive, because they require at least linear time in the graph size.

2.2 Influence maximization in dynamic networks

There have been a few studies that have considered influence maximization in a dynamic setting. Zhuang *et al.* [31] considered a related but different problem, in which a graph dynamically changes, but only a small number of vertices can be probed and only at certain times. Hence, information is only available on a small part of the current network. The objective is to maximize the influence spread in the current network. Note that in our problem, we have full knowledge on how the graph has been changed, and our primary concern is to efficiently maintain the index.

Chen *et al.* [6] considered the problem of tracking a seed set that maximizes the influence spread when a graph dynamically changes, and proposed a method that iteratively updates the current seed set. It should be noted that their method does not incorporate vertex additions and deletions. Furthermore, our method concerns not only influence maximization, but also influence estimations for *any* vertex set.

3. PRELIMINARIES

3.1 Notations

For an integer k, we use $[k]$ to denote the set $\{1, 2, \ldots, k\}$. For a finite set S, $s \leftarrow_R S$ means that we sample s from S uniformly at random.

Let $G = (V, E, p)$ be a directed influence graph, where V is a vertex set of size n, E is an edge set of size m, and $p : E \to [0, 1]$ is a propagation probability function representing the magnitude of influence between a pair of vertices. For a vertex v in G, we use $d^+_G(v)$ and $d^-_G(v)$ to denote the out-degree and the in-degree of v, respectively. When G is clear from the context, we will omit the subscripts.

3.2 Problem definition of influence estimation and influence maximization

In this paper, we adopt the most standard information diffusion model, called the *independent cascade* (IC) model, which was formulated by Goldenberg *et al.* [11, 12]. In the IC model, given a graph $G = (V, E, p)$ and a vertex set $S \subseteq V$, called a *seed* set, we first activate the vertices in S. Then, the process evolves in discrete steps according to the following randomized rule. When a vertex u becomes active for the first time at the step t, it is given a single chance to activate each current inactive vertex v among its neighbors. It will succeed with probability p_{uv}. If u succeeds, then v will become active in the step $t + 1$. Whether or not u succeeds, it cannot make any further attempts to activate v in subsequent steps. This process continues until no further activation is possible. The *influence spread* of a seed set S under the IC model, denoted by $\sigma(S)$, is defined as the expected total number of active vertices for S.

We formally define the influence estimation problem and the influence maximization problem as follows.

PROBLEM 1 (INFLUENCE ESTIMATION). *Given a graph $G = (V, E, p)$ and a vertex set $S \subseteq V$, this problem asks to compute the influence spread $\sigma(S)$ of S.*

PROBLEM 2 (INFLUENCE MAXIMIZATION). *Given a graph $G = (V, E, p)$ and an integer k, this problem asks to find a vertex set $S \subseteq V$ of size k that maximizes $\sigma(S)$.*

Note that it has been proven that the influence estimation problem is #P-hard [4], and the influence maximization problem is NP-hard [15].

3.3 Naive method for influence estimation

Because the exact computation of $\sigma(\cdot)$ is #P-hard [4], previous studies have employed Monte-Carlo simulations for this purpose [5, 8, 15, 19, 30]. That is, we simulate the diffusion process several times, and then compute the average number of activated vertices. The time complexity of Monte-Carlo simulations is $O(mr)$, where r is the number of simulations. As this naive method requires linear time just to estimate the influence of a single set, we cannot adopt this method in the dynamic setting.

3.4 Naive method for influence maximization

Although the influence maximization problem has been proven to be NP-hard [15], a natural greedy algorithm can achieve a constant approximation to the optimum solution, owing to the non-negativity, monotonicity, and submodularity of $\sigma(\cdot)$. We say that a set function $f : 2^V \to \mathbb{R}$ is *monotone* if $f(S) \leq f(T)$ for all $S \subseteq T$, and *submodular* if $f(S \cup \{v\}) - f(S) \geq f(T \cup \{v\}) - f(T)$ for all $S \subseteq T$ and $v \in V \setminus T$. The greedy algorithm, introduced by [15], repeatedly adds the vertex with the maximum marginal influence into S until k vertices have been added. The following theorems guarantee that the greedy algorithm approximates the optimum solution within a factor of $1 - 1/e$, by evaluating the influence spread function $\sigma(\cdot)$ kn times.

THEOREM 3.1. *[15] For the IC model, the influence spread function $\sigma(\cdot)$ is non-negative, monotone, and submodular.*

THEOREM 3.2. *[23] For a non-negative, monotone, and submodular function f, let S be a set of size k obtained using the greedy strategy. Then, $f(S) \geq (1 - 1/e) f(S^*)$, where S^* is the optimum solution.*

Although we cannot evaluate $\sigma(\cdot)$ exactly, we can estimate it using the Monte-Carlo simulations described in the previous section. By doing so, although the approximation guarantee does not deteriorate greatly, the time complexity is $O(knmr)$, which is prohibitively large for even static graphs.

3.5 The RIS sketching method

Borgs *et al.* [3] addressed the inefficiency of these naive methods by introducing a sketching method. Because we will provide the details in Section 4, we will only present an overview of their method here.

First, we repeatedly construct subgraphs as follows. In the i-th step, we choose a target vertex $z_i \in V$ and an *activation function* $x_i : E \to [0,1]$ uniformly at random. We call an edge $uv \in E$ *active* with respect to x_i if $x_i(uv) \leq p_{uv}$, and *inactive* with respect to x_i otherwise. We regard a non-edge $uv \notin E$ as inactive irrespective of x_i. We write $u \overset{x_i}{\rightsquigarrow} v$ to indicate that there is a directed path from u to v consisting only of active edges with respect to x_i. Then, we compute a subgraph H_i that consists of vertices v such that $v \overset{x_i}{\rightsquigarrow} z_i$ by performing a reverse BFS from z_i. We cease this process when the total number of traversed edge exceeds $\Theta(\frac{1}{\epsilon^3}(n+m)\log n)$, where ϵ is an error parameter.

Intuitively, influential vertices seem likely to appear in H_i's. For a vertex set $S \subseteq V$, let $d'(S)$ be the number of subgraphs H_i such that $S \cap V(H_i) \neq \emptyset$. It is not hard to see that $d'(S)$ is proportional to the expected number of vertices influenced by some vertex in S. More precisely, we estimate the influence spread of S by $n \cdot d'(S)/t$, where t is the total number of generated subgraphs. With high probability, this value is a $(1 \pm \epsilon)$-approximation to $\sigma(S)$ [3].

We can also solve the influence maximization problem, as follows. We begin with an empty set S, and then add the vertex t that has the maximum marginal degree, i.e., $t = \mathrm{argmax}_{v \in V \setminus S} d'(S \cup \{v\}) - d'(S)$, into S until k vertices have been added. The resulting set is a $(1-1/e-\epsilon)$-approximation to the optimum set with high probability [3].

The advantage of this sketching method is that no matter what k is, the total time complexity is bounded by $O(\frac{1}{\epsilon^3}(n+m)\log n)$, which is much smaller than the time complexity $O(knmr)$ of the naive method. However, it is too expensive to apply this sketching method from scratch every time the graph changes. In the next section, we will develop a method that efficiently updates the sketch when the graph changes.

4. PROPOSED METHOD

In this section, we will present our efficient indexing method for influence analysis in evolving graphs. First, we will explain what we store in our index, and how it is constructed from a static graph. Then, we will demonstrate how to compute the influences of vertex sets and extract a set of highly influential vertices using the index. Finally, we will explain how the index is dynamically updated.

4.1 Index structure

Let us begin with our index structure for a static graph $G = (V, E, p)$. Let $\epsilon > 0$ be a precision parameter. Our index consists of a set of tuples $I = \{(z_i, x_i, H_i)\}_i$, where $z_i \in V$ is a *target* vertex, $x_i : E \to [0,1]$ is an activation function, and H_i is a subgraph of G consisting of active edges with respect to x. Here, every vertex in $V(H_i)$ is able to reach z_i in H_i. We call each (z_i, x_i, H_i) a *sketch*.

The number of sketches in I is determined as follows: The *weight* of a subgraph H, denoted by $w(H)$, is defined as $|V(H)| + \sum_{v \in V(H)} d^-(v)$, which equals the space required to store H. Then, we will always keep the condition that $\sum_{i \in [|I|-1]} w(H_i) < W$ and $\sum_{i \in [|I|]} w(H_i) \geq W$, where $W = \Theta(\frac{1}{\epsilon^3}(n+m)\log n)$. Note that the space complexity of our index is roughly bounded by $O(W)$.

For a vertex v, let I_v denote the set of indices $i \in [|I|]$ with $v \in V(H_i)$, and for a vertex set S, let I_S denote the set of indices $i \in [|I|]$ with $S \cap V(H_i) \neq \emptyset$. That is, $I_S = \bigcup_{v \in S} I_v$. Our index stores I_v for every vertex v, so that we can quickly fetch the sketches that include a particular vertex.

We note that the sketch adopted in RIS [3] cannot support dynamic updates efficiently, because it only stores the vertex set $V(H_i)$. On the other hand, we will show that we can update sketches when the graph dynamically changes.

4.2 Index construction

Here, we describe how we construct our index from a static graph. Given a graph $G = (V, E, p)$, we begin with an empty index $I = \emptyset$, and we repeatedly create new sketches and add them into I for as long as the total weight of the current index is less than W. A sketch is constructed using the following reverse-BFS-like method: First, we sample a target vertex $z_i \in V$ uniformly at random, and add it to an empty queue. We also sample $x_i : E \to [0,1]$ uniformly at random. If the queue is not empty, then we remove a vertex v from the queue. If this is the first time that v is visited, then for each active edge uv, we add u to the queue. Finally, we set $V(H_i)$ to the visited vertices and $E(H_i)$ to the visited active edges. Building a sketch (z_i, x_i, H_i) requires $O(w(H_i))$ time, and thus the entire index construction requires $O(W)$ time.

4.3 Supporting queries

In this subsection, we will describe how we approach influence estimation and influence maximization queries using the index constructed in the previous subsection.

4.3.1 Influence estimation

We will start with influence estimation. The number of vertices that a vertex v influences can be approximated by $n \cdot |I_v|/|I|$. Because we actually simulate the propagation process several times and take the average, the expected value is exactly equal to the influence $\sigma(\{v\})$. Similarly, for a vertex set S, the number of vertices that S influences can be approximated by $n \cdot |I_S|/|I|$.

Given a single vertex v, we can compute $n \cdot |I_v|/|I|$ in constant time, as we have I_v stored in our index. For a vertex set S of size greater than one, by storing each $V(H_i)$ using a hash so that we can check $v \in V(H_i)$ in constant time, the time complexity of EstimateInfluence is bounded by $O(|S| \cdot |I|)$. However, we can improve the running time by using I_v stored in our index. That is, we compute I_S as $I_S = \bigcup_{v \in S} I_v$. This technique reduces the running time to $O(\sum_{v \in S} |I_v|)$, and we will demonstrate that $|I_v|$ is much smaller than $|I|$ by means of our thorough experiments. See EstimateInfluence in Algorithm 1 for further details.

Because our method is equivalent to the (non-indexing) method by Borgs *et al.* [3], we inherit the same guarantee of its accuracy. To summarize, the following guarantee holds.

Theorem 4.1. *By choosing* $W = \Theta(\frac{1}{\epsilon^3}(n+m)\log n)$, EstimateInfluence$(I, S)$ *estimates* $\sigma(S)$ *with an additive error of* ϵ *with probability at least* $1 - \frac{1}{n}$ *over the choice of* z_i*'s and* x_i*'s. The time complexity of* EstimateInfluence(I, S) *is* $O(\sum_{v \in S} |I_v|)$.

4.3.2 Influence maximization

Algorithm 1 Influence queries.

procedure EstimateInfluence(I, S)
  if $S = \{v\}$ **then**
    return $n \cdot |I_v|/|I|$.
  else
    return $n \cdot \left|\bigcup_{v \in S} I_v\right| / |I|$.

procedure MaximizeInfluence(I, k)
  $S \leftarrow \emptyset$.
  for $i = 1$ **to** k **do**
    $v_i \leftarrow \mathrm{argmax}_{v \in V \setminus S} d_{I-S}(S \cup \{v_i\})$.
    $S \leftarrow S \cup \{v_i\}$.
  return S.

Next, we consider the problem of finding a vertex set S of size k that maximizes the expected number of vertices influenced by S, where k is the input parameter. The greedy algorithm is a standard approach for this problem, and Borgs *et al.* [3] reduced this greedy algorithm to the greedy strategy on a set of subgraphs. We will describe their procedure in our language. For a vertex v, define the *degree* $d_I(v)$ of v in I as the number of $i \in [|I|]$ with $v \in V(H_i)$. In other words, $d_I(v) = |I_v|$ initially. We choose the vertex with the maximum degree in I and denote this by v_1. Then, examining each sketch (z_i, x_i, H_i), we remove it if $v_1 \in V(H_i)$. Next, we choose the vertex with the maximum degree in the resulting index, and denote this by v_2. Then, examining each sketch (z_i, x_i, H_i), we remove it if $v_2 \in V(H_i)$. We repeat this process k times, and then output the vertex set $\{v_1, v_2, \ldots, v_k\}$. The details are provided in MaximizeInfluence in Algorithm 1. Here, $I - S$ denotes the index obtained by removing all of the sketches (z_i, x_i, H_i) with $S \cap V(H_i) \neq \emptyset$. To boost the empirical performance, we also employ the lazy evaluation technique [21].

Theorem 4.2. *Let $W = \Theta(\frac{1}{\epsilon^3}(n+m)\log n)$. Then,* MaximizeInfluence(I, k) *returns a set S of size k, such that $\sigma(S) \geq (1 - 1/e - \epsilon)\sigma(S^*)$ with probability at least $1 - \frac{1}{n}$ over the choice of z_i's and x_i's, where $S^* = \mathrm{argmax}_{S \subseteq V : |S| = k} \sigma(S)$. The time complexity of* MaximizeInfluence(I, k) *is bounded by $O(\sum_{i \in [|I|]} |V(H_i)|)$.*

Proof. Because this method is equivalent to the (non-indexing) method by Borgs *et al.* [3], we inherit the same guarantee of its accuracy.

Now, we consider the time complexity. When we add a vertex v to the output, we need to decrement the degrees of vertices $u \in V(H_i)$ for each $i \in I_v$. However, this decrementation occurs only once for each sketch. Hence, the total time complexity is $O(\sum_{i \in [|I|]} |V(H_i)|)$. □

4.4 Supporting dynamic update operations

In this subsection, we explain how we update our index efficiently. We consider five operations: vertex additions, vertex deletions, edge additions, edge deletions, and propagation probability updates. First, we present three subroutines used in our update operations, and then we describe how we update the index when the graph changes. The details are provided in Algorithms 2, 3, and 4. The correctness will be demonstrated in Section 5.

4.4.1 Auxiliary subroutines

Expand(I, i, z): Suppose that we have added an edge zw or increased the propagation probability of an edge zw. Then, for each $i \in [|I|]$ with $w \in V(H_i)$, we want to add vertices from which we can newly reach the vertex z_i to H_i. To this end, we perform a reverse BFS from z, and add the traversed vertices to H_i. Note that all of the newly added vertices can reach z.

Algorithm 2 Auxiliary functions.

procedure Expand(I, i, z)
  $Q \leftarrow$ a queue with only one element z.
  $H_i \leftarrow H_i \cup \{z\}$.
  while $Q \neq \emptyset$ **do**
    Dequeue v from Q.
    for all $uv \in E$ **do**
      if $x_i(uv) < p_{uv}$ **then**
        $E(H_i) \leftarrow E(H_i) \cup \{uv\}$.
        if $v \notin V(H_i)$ **then**
          Enqueue u onto Q.
          $V(H_i) \leftarrow V(H_i) \cup \{u\}$.

procedure Shrink(I, i)
  $H_i \leftarrow$ the subgraph consisting of vertices that can reach z_i by passing through active edges.

procedure Adjust(I)
  $W \leftarrow \Theta(\frac{1}{\epsilon^3}(n+m)\log n)$.
  while $\sum_{1 \leq i \leq |I|} w(H_i) < W$ **do**
    $z_{|I|+1} \leftarrow_R V$.
    Expand$(I, |I|+1, z_{|I|+1})$.
  while $\sum_{1 \leq i \leq |I|-1} w(H_i) \geq W$ **do**
    Discard the last element from I.

Shrink(I, i): Suppose that we have removed an edge uv or decreased the propagation probability of an edge uv. Then, for $i \in [|I|]$ such that $uv \in E(H_i)$, we want to remove the vertices in H_i from which we can no longer reach z_i anymore. To this end, we recompute the set of vertices that can reach z_i by conducting a reverse BFS from z_i.

Adjust(I): While we are processing edge and vertex updates, the total weight of the index may violate the condition on the total weight. In such a case, we create new sketches or remove current sketches as follows. If the total weight is smaller than the threshold W, then we create a new sketch (z, x, H) by sampling $z \in V$, and calling Expand on z to make H. On the other hand, if the total weight of sketches, excluding the last one, is larger than or equal to W, then we remove the last sketch from the index.

4.4.2 Dynamic update routines

Now, we explain how we update our index when the graph changes.

AddVertex(I, v): Suppose that we have added a new vertex v to the current graph. In such a case, we must update the target vertices in the index to preserve the property that each vertex in the graph is chosen uniformly at random as a target vertex.

Let V and V' denote the vertex set of a graph before and after we add a new vertex v, respectively. That is, $V' = V \cup \{v\}$. Suppose that we construct an index from scratch after inserting v. Obviously, for each time we choose a sketch, the probability that the target vertex is chosen from V is $\frac{n}{n+1}$, and the probability that the target vertex is v is $\frac{1}{n+1}$. In order to ensure that this property holds, we update the target vertex in the current index to v with probability $\frac{1}{n+1}$.

Algorithm 3 Vertex operations.

procedure AddVertex(I, v)
  for $i = 1$ **to** $|I|$ **do**
    continue with probability $1 - \frac{1}{n+1}$.
    $H_i \leftarrow \emptyset$, $z_i \leftarrow v$.
    Expand(I, i, z_i).
  Adjust(I).

procedure DeleteVertex(I, v)
  Remove edges incident to V from each $E(H_i)$.
  for all $i \in I_v$ **do**
    if $z_i = v$ **then**
      $H_i \leftarrow \emptyset$, $z_i \leftarrow_R V$.
      Expand(I, i, z_i).
    else
      Shrink(I, i).
  Adjust(I).

Algorithm 4 Edge operations.

procedure Change(I, uv, p)
  $p_{uv} \leftarrow p$.
  for all $i \in I_v$ **do**
    $P \leftarrow [\![uv \in E(H_i)]\!]$, $Q \leftarrow [\![x_i(uv) < p]\!]$.
    if $\neg P \wedge Q$ **then** ▷ inactive → active
      $E(H_i) \leftarrow E(H_i) \cup \{uv\}$.
      Expand(I, i, u).
    if $P \wedge \neg Q$ **then** ▷ active → inactive
      $E(H_i) \leftarrow E(H_i) \setminus \{uv\}$.
      Shrink(I, i).
  Adjust(I).

procedure AddEdge(I, uv, p)
  $p_{uv} \leftarrow 0$.
  Change(I, uv, p).

procedure DeleteEdge(I, uv)
  Change($I, uv, 0$).

DeleteVertex(I, v): Suppose that we have removed a vertex v from the current graph. Then, for each $i \in [|I|]$, we check whether v is contained in H_i. If this is the case, then we update the tuple (z_i, x_i, H_i) as follows: If $z_i = v$, we sample z_i from $V \setminus \{v\}$ uniformly at random and compute H_i. Otherwise, we remove v and all edges incident to v from H_i, and then we call Shrink(I, i) to shrink H_i.

Change(I, uv, p): Suppose that we have changed the propagation probability of an edge uv from p' to p. If the state of uv with respect to x_i changes, then we need to update subgraphs in the index I. More specifically, we carry out the following for each $i \in [|I|]$. If $p' < x_i(uv) \leq p$, then we expand H_i by calling Expand(I, i, u). If $p < x_i(uv) \leq p'$, then we shrink H_i by calling Shrink(I, i).

AddEdge(I, uv, p): Suppose that we have added an edge uv with propagation probability p to the current graph. First, we add uv to the current edge set E and set $p_{uv} = 0$. Then, for each sketch $(z_i, x_i, H_i) \in I$ such that $v \in V(H_i)$, we define $x_i(uv)$ by choosing a value from $[0, 1]$ uniformly at random. Finally, we update the propagation probability p_{uv} to p, by calling Change(I, uv, p).

DeleteEdge(I, uv): Suppose that we have deleted an edge uv from the current graph. Then, we first update the edge probability p_{uv} to zero, by calling Change($I, uv, 0$), and then remove uv from E. Then, we remove uv from the domain of x_i for each $i \in [|I|]$. We only examine the sketches containing v, and thus the expected number of examined sketches is $\frac{\sigma(\{v\})}{n}|I|$, which follows from Lemma 5.11.

5. THEORETICAL ANALYSIS

In this section, we will show the correctness of our indexing method and then proceed to analyze its time complexity.

5.1 Correctness

Now, we consider the correctness of our indexing method. Note that our method is randomized. We first define $\mathcal{I}_W^{\mathrm{sta}}(G)$ and $\mathcal{I}_W^{\mathrm{dyn}}(G)$ as the distribution of indices in the case that we apply our method to a static graph G and the sequence of dynamic updates that results in G, respectively. Our goal is to show that $\mathcal{I}_W^{\mathrm{sta}}(G) = \mathcal{I}_W^{\mathrm{dyn}}(G)$. If this is the case, then queries on the index following dynamic updates will inherit the same guarantees that furnish Theorems 4.1 and 4.2.

Given a graph $G = (V, E, p)$, consider the following random process that generates a sequence of pairs. For each step, we sample a target vertex $z \in V$ and an activation function $x : E \to [0, 1]$ uniformly at random, and add the pair (z, x) to the sequence. Let $\mathcal{X}_\infty(G)$ denote the distribution of (infinite) sequences of pairs obtained in this way. Let $H(z, x)$ be the subgraph consisting of vertices that can reach z under x. Furthermore, let $\mathcal{I}_\infty(G)$ denote the distribution of tuple sequences obtained from $\mathcal{X}_\infty(G)$ by replacing each pair (z, x) by $(z, x, H(z, x))$.

We say that a distribution $\mathcal{X}$ of pair sequences is *valid* for G if it can be obtained by sampling a random sequence from $\mathcal{X}_\infty(G)$ uniformly at random and taking a prefix of it (of arbitrary length). Similarly, we say that a distribution $\mathcal{I}$ of tuple sequences is *valid* for G if it can be obtained by sampling a random sequence from $\mathcal{I}_\infty(G)$ uniformly at random and taking a prefix of it.

For a positive integer W, we define $\mathcal{I}_W(G)$ as the distribution over prefixes of tuple sequences in $\mathcal{I}_\infty(G)$ that are obtained as follows: We sample a tuple sequence (z_1, x_1, H_1), (z_2, x_2, H_2), ... from $\mathcal{I}_\infty(G)$ and take the minimum prefix of it such that the total weight of the subgraphs is at least W. We sample a tuple sequence $(z_1, x_1, H_1), (z_2, x_2, H_2), \ldots$ from $\mathcal{I}_\infty(G)$ and take the minimum prefix of it such that the total weight of the subgraphs is at least W. It is easy to see that $\mathcal{I}_W(G) = \mathcal{I}_W^{\mathrm{sta}}(G)$. We will establish that $\mathcal{I}_W^{\mathrm{sta}}(G) = \mathcal{I}_W^{\mathrm{dyn}}(G)$ by showing that $\mathcal{I}_W^{\mathrm{dyn}}(G) = \mathcal{I}_W(G)$.

For the empty graph G, we clearly have $\mathcal{I}_W^{\mathrm{dyn}}(G) = \mathcal{I}_W(G)$. We will show that, for any graph G with $\mathcal{I}_W^{\mathrm{dyn}}(G) = \mathcal{I}_W(G)$ and a graph G' obtained from G by a dynamic update, we again have $\mathcal{I}_W^{\mathrm{dyn}}(G') = \mathcal{I}_W(G')$. Then, we are done by induction on the number of updates.

The following auxiliary lemma states that we can change the length of a valid distribution to W using Adjust($\cdot$).

Lemma 5.1. *Let G be a graph, and let $\mathcal{I}$ be a valid distribution of finite pair sequences for G. Then, $\mathcal{I}' =$ Adjust($\mathcal{I}$) is equal to $\mathcal{I}_W(G)$, where W is a parameter used in* Adjust.

Remark 5.2. *By Adjust($\mathcal{I}$), we mean the distribution of sequences obtained by applying* Adjust *to a sequence I sampled from $\mathcal{I}$. We will use similar conventions for other procedures in the following.*

Proof. We can obtain $\mathcal{I}_W(G)$ from $\mathcal{I}$ as follows. Let $I = (z_1, H_1), (z_2, H_2), \ldots, (z_{|I|}, H_{|I|})$ be a pair sequences sampled from $\mathcal{I}$. Then, we repeat the following process. If $\sum_{1 \leq i \leq |I|} w(H_i) < W$, then we sample a target vertex z and an activation function $x : E \to [0, 1]$ uniformly at random. Then, we compute the corresponding subgraph H,

and add the pair (z, H) to I. If $\sum_{1 \leq i \leq |I|-1} w(H_i) \geq W$, then we remove the last element from I. This is exactly corresponds to what is carried out in ADJUST, and it follows that $\mathcal{I}' = \mathcal{I}_W(G)$. □

In the following, we will show that our update routines in Algorithms 3 and 4 transform a valid distribution for the original graph to a valid distribution for the new graph.

LEMMA 5.3. *Let G be a graph, $e \in E(G)$, and $p \in [0, 1]$. Let G' be the graph obtained from G by changing the propagation probability of e to p. If $\mathcal{I}$ is a valid distribution of tuple sequences for G, then the distribution $\mathcal{I}' = \text{CHANGE}(\mathcal{I}, e, p)$ is a valid distribution of tuple sequences for G'.*

PROOF. Let $\mathcal{X}$ and $\mathcal{X}'$ be the distributions of the pair sequences corresponding to $\mathcal{I}$ and $\mathcal{I}'$, respectively. Then, it is clear that $\mathcal{X} = \mathcal{X}'$. Because $\mathcal{X}$ is a valid sequence for G, it follows that $\mathcal{X}'$ is also a valid sequence for G'. Hence, $\mathcal{I}'$ is a valid distribution for G'. □

LEMMA 5.4. *Let G be a graph and G' be the graph obtained from G by adding a new vertex $v \notin V(G)$. If $\mathcal{I}$ is a valid distribution for G, then $\mathcal{I}' = \text{ADDVERTEX}(\mathcal{I}, v)$ is a valid distribution for G'.*

PROOF. Let $\mathcal{X}$ and $\mathcal{X}'$ be the distributions of the pair sequences corresponding to $\mathcal{I}$ and $\mathcal{I}'$, respectively. Then, we can obtain $\mathcal{X}'$ from $\mathcal{X}$ as follows. Let $(z_1, x_1), (z_2, x_2), \ldots$ be a sequence sampled from $\mathcal{X}$. Then, we replace each of z_i by v with probability $1 - 1/|V(G')|$. We can observe that $\mathcal{X}'$ is a valid sequence for G', and it follows that $\mathcal{I}'$ is a valid distribution for G'. □

LEMMA 5.5. *Let G be a graph and G' be the graph obtained from G by removing a vertex $v \in V(G)$. If $\mathcal{I}$ is a valid distribution for G, then $\mathcal{I}' = \text{DELETEVERTEX}(\mathcal{I}, v)$ is a valid distribution for G'.*

PROOF. Let $\mathcal{X}$ and $\mathcal{X}'$ be the distributions of the pair sequences corresponding to $\mathcal{I}$ and $\mathcal{I}'$, respectively. Then, we can obtain $\mathcal{X}'$ from $\mathcal{X}$ as follows. Let $(z_1, x_1), (z_2, x_2), \ldots$ be a sequence sampled from $\mathcal{X}$. If $z_i = v$, then we again replace z_i from $V(G) - v$ uniformly at random again. We can observe that $\mathcal{X}'$ is a valid distribution for G', and it follows that $\mathcal{I}'$ is a valid distribution for G'. □

Similarly, we obtain the following two lemmas. As the proofs are quite similar to the ones of Lemmas 5.4 and 5.5, we omit them due to the lack of space.

LEMMA 5.6. *Let G be a graph and G' be a graph obtained from G by adding a new edge $uv \notin E(G)$. If $\mathcal{I}$ is a valid distribution for G, then $\mathcal{I}' = \text{ADDEDGE}(\mathcal{I}, uv)$ is a valid distribution for G'.*

LEMMA 5.7. *Let G be a graph and G' be the graph obtained from G by removing an edge $uv \in E(G)$. If $\mathcal{I}$ is a valid distribution for G, then $\mathcal{I}' = \text{DELETEEDGE}(\mathcal{I}, uv)$ is a valid distribution for G'.*

THEOREM 5.8. $\mathcal{I}_W^{\text{dyn}}(G) = \mathcal{I}_W(G)$.

PROOF. From Lemmas 5.4, 5.5, 5.6, and 5.7, we have that the distribution of indices obtained by our dynamic update procedures is always a valid distribution of tuple sequences for the current graph. Because we apply ADJUST at the end of each update, the distribution of indices is given exactly by $\mathcal{I}_W(G)$ by Lemma 5.1. □

By Theorems 5.8, 4.1, and 4.2, we obtain the following.

THEOREM 5.9. *Let $W = \Theta(\frac{1}{\epsilon^3}(n+m)\log n)$, and let I be the index obtained by a sequence of dynamic updates. Then, $\text{ESTIMATEINFLUENCE}(I, S)$ estimates $\sigma(S)$ with an additive error of ϵ with probability at least $1 - \frac{1}{n}$ over the choice of z_i's and x_i's.*

THEOREM 5.10. *Let $W = \Theta(\frac{1}{\epsilon^3}(n+m)\log n)$, and let I be the index obtained by a sequence of dynamic updates. $\text{MAXIMIZEINFLUENCE}(I, k)$ returns a set S of size k, such that $\sigma(S) \geq (1 - 1/e - \epsilon)\sigma(S^*)$ with probability at least $1 - \frac{1}{n}$ over the choice of z_i's and x_i's, where $S^* = \text{argmax}_{S \subseteq V : |S| = k} \sigma(S)$.*

5.2 Time complexity

Now, we turn our focus to analyzing the time complexity of our indexing method. We note that it is difficult to precisely bound the time complexity of dynamic update operations because it depends on the sizes of cascades and hence the structure of the input graph. Instead, we will analyze the number of sketches examined in each update operation.

To this end, we will apply the following lemma, the correctness of which is obvious from the construction of H_i.

LEMMA 5.11. *For a vertex set S and a randomly sampled sketch (z_i, x_i, H_i), it holds that $\Pr[V(H_i) \cap S \neq \emptyset] = \frac{\sigma(S)}{n}$, where the probability is over the choice of z_i and x_i.*

THEOREM 5.12. ADDVERTEX *examines $|I|$ sketches.* DELETEVERTEX, CHANGE, ADDEDGE, *and* DELETEEDGE *examine $\sigma(\{v\})/n \cdot |I|$ sketches on average.*

PROOF. The first claim is obvious, as the number of sketches is $|I|$. Suppose that $\text{DELETEVERTEX}(I, v)$, $\text{CHANGE}(I, uv, p)$, $\text{ADDEDGE}(I, uv, p)$, or $\text{DELETEEDGE}(I, uv, p)$ was called. Then, we only examine the sketches containing v and thus the expected number of examined sketches is $\frac{\sigma(\{v\})}{n}|I|$, following from Lemma 5.11. □

6. TECHNIQUES FOR IMPROVING THE PERFORMANCE

Here, we introduce several techniques to improve the performance of our indexing method.

6.1 A reachability-tree-based pruning technique

Although we have demonstrated that the number of updated sketches resulting from vertex deletions and update operations on edges is small, naive implementations of these are computationally expensive, because a whole subgraph is scanned for each relevant sketch in the SHRINK procedure. In this section, we will address this issue by introducing a technique, called the *reachability-tree-based technique*.

A key observation is that the removal of a single edge or vertex rarely effects the reachability among the vertices in a sketch, because real-world social networks are highly connected. To exploit this observation, for each sketch we will store a *directed reachability tree* T_i in H_i rooted at z_i. Therefore, each sketch is now a tuple (z_i, x_i, H_i, T_i).

6.1.1 Definition of reachability trees

Let us begin with the definition of the reachability tree. The *reachability tree* for a sketch (z_i, x_i, H_i) is a directed tree on the vertex set $V(H_i)$ towards the root z_i, constructed by using active (directed) edges in $E(H_i)$ with regard to x_i. Each vertex in $V(H_i)$ is able to reach z_i along with T_i.

6.1.2 *Speeding up the* SHRINK *procedure*

Next, we will explain how we can apply the reachability tree to speed up the SHRINK procedure.

Suppose that an edge uv is inactivated in the CHANGE procedure. Then, we need to update each sketch (z_i, x_i, H_i, T_i) with $uv \in E(H_i)$ by calling the procedure SHRINK(I, i) (Line 10). Previously, we updated H_i by performing a reverse BFS from z_i. Because we now have T_i, we can update H_i more efficiently. We present the details as follows.

If $uv \notin E(T_i)$, then u can still reach z_i along with T_i after the removal of uv. Thus, we do nothing. Otherwise, the vertices in $V(T_i(u))$ are the only candidates that may no longer reach z_i, where $T_i(u)$ denotes the subtree rooted at u. Thus, we check whether the vertices in $V(T_i(u))$ can reach $V(H_i) \setminus V(T_i(u))$. With this aim, we first explicitly compute the set $V(T_i(u))$. Then, we compute the set R of vertices in $V(T_i(u))$ adjacent to $V(H_i) \setminus V(T_i(u))$. Finally, we compute the set of vertices in $V(T_i(u))$ that can reach R, using a reverse BFS from R. Then, we remove the vertices that can no longer reach R from $V(H_i)$ and update H_i and T_i. The pseudocode is presented as SHRINK-AFTER-EDGE-REMOVAL in Algorithm 5.

In addition, we can speed up SHRINK(I, i) at Line 14 in the DELETEVERTEX procedure in a similar manner. The pseudocode is presented as SHRINK-AFTER-VERTEX-REMOVAL in Algorithm 5.

6.1.3 *Maintaining reachability trees*

Now, we discuss how we maintain reachability trees. First, when creating a new sketch (z_i, x_i, H_i, T_i), T_i is a tree consisting of a single vertex z_i. When updating T_i in an existing sketch (z_i, x_i, H_i, T_i), we have the following four cases.

Suppose that an edge uv with $uv \in E(H_i)$ is inactivated. Let H_i' be the new subgraph computed by calling SHRINK-AFTER-EDGE-REMOVAL(I, i, uv). Note that the vertices in $R' = V(H_i') \cap V(T_i(u))$ are obtained by performing a reverse BFS from R (see SHRINK-AFTER-EDGE-REMOVAL for the definitions of R and R'). Hence, by removing the subtree $V(T_i(u))$ from T_i and concatenating the tree formed by this reverse BFS, we obtain a new reachability tree for H_i'. Suppose that an edge uv with $v \in V(H_i)$ is activated. Then, we perform a reverse BFS from u, and add the edge uv and the obtained subtree rooted at u to T_i. When deleting a vertex v with $v \in V(H_i)$, we can update T_i, in a similar manner as in the case where an edge is inactivated. When adding a vertex, we are not required to do anything.

6.2 A skipping method for vertex addition

When adding a vertex, we are required to change the target vertex of each sketch with probability $\frac{1}{n+1}$. However, running through all of the sketches in I is a costly procedure. We can avoid this issue by applying the following technique. Let k be the first index for which we change the target vertex z_k. Then, for each positive integer t, we have $\Pr[k = t] = (1-\alpha)^{t-1}\alpha$, where $\alpha = \frac{1}{n+1}$. Hence, we can sample k by first sampling $x \in [0, 1]$ uniformly at random, and then taking the minimum k such that $\sum_{t \in [k]} (1-\alpha)^{t-1}\alpha \geq x$. This is equivalent to $k \geq \log \frac{1}{1-x} / \log \frac{1}{1-\alpha}$. We choose minimum such k with these properties, and change the target vertex of the k-th sketch to v. Then, we repeat the same procedure for the remainder of the index. See ADDVERTEX' in Algorithm 5 for complete details.

Algorithm 5 Improved operations.

1: **procedure** SHRINK-AFTER-EDGE-REMOVAL(I, i, uv)
2: **if** $uv \notin E(T_i)$ **then return**
3: $T' \leftarrow$ the subtree rooted at u.
4: $R \leftarrow$ vertices in T' adjacent to $V(H_i) \setminus T'$.
5: $R' \leftarrow$ the set of vertices in T' that can reach R.
6: **for** $w \in T' \setminus R'$ **do**
7: Remove w and edges entering w from H_i and T_i, resp.
8: **procedure** SHRINK-AFTER-VERTEX-REMOVAL(I, i, v)
9: $T' \leftarrow$ the subtree rooted at v.
10: $R \leftarrow$ vertices in T' adjacent to $V(H_i) \setminus T'$.
11: $R' \leftarrow$ the set of vertices in T' that can reach R.
12: **for** $w \in T' \setminus R'$ **do**
13: Remove w and edges entering w from H_i and T_i, resp.
14: **procedure** ADDVERTEX'(I, v)
15: $\alpha \leftarrow \frac{1}{n+1}$, $i \leftarrow 0$.
16: **while** $i < |I|$ **do**
17: $x \leftarrow_R [0, 1]$, $k \leftarrow \lceil \log \frac{1}{1-x} / \log \frac{1}{1-\alpha} \rceil$, $i \leftarrow i + k$.
18: **if** $i \leq |I|$ **then**
19: $H_i \leftarrow \emptyset$, $z_i \leftarrow v$.
20: EXPAND(I, i, z_i).
21: ADJUST(I).

Because we are only required to seeing sketches for which we change the target vertex, the expected number of sketches we look at is $\frac{|I|}{n+1}$, which is much smaller than $|I|$.

6.3 Random number generators

We have assumed that each sketch (z_i, x_i, H_i, T_i) stores a function $x_i : E \to [0, 1]$. This is undesirable because storing x_i requires $O(m)$ space.

To address this issue, we use a pseudorandom generator called Random123 [27]. Random123 is *counter-based*, that is, it maps an integer to a (pseudo)random value. Instead of explicitly storing $x_i(uv)$ for each $i \in [|I|]$ and $uv \in E$, for each time that we need $x_i(uv)$, we call Random123 with an integer representation of the pair (i, uv), and we use the obtained value after normalizing it to be in $[0, 1]$.

7. EXPERIMENTS

In this section, we will demonstrate the efficiency and effectiveness of our indexing method by performing experiments on real-world networks. We conducted the experiments on a Linux server, with Intel Xeon E5-2690 2.90GHz CPU and 256GB memory. All algorithms were implemented in C++ and compiled using g++v4.6.3 with the -O2 option.

7.1 Experimental settings

Datasets: We selected seven real-world dynamic networks with timestamps, from the Koblenz Network Collection.[2] We ordered edges in ascending time order. The basic information regarding each dataset is presented in Table 1 and the degree distribution of each dataset is plotted in Figure 3.

Probability settings: As these networks do not contain information concerning the propagation probabilities of edges, we assign an edge probability p_{uv} for each edge uv to a probability chosen from $\{0.1, 0.01, 0.001\}$ uniformly at random under the *trivalency* (TR) model [4], or $1/d^-(v)$ under the *weighted cascade* (WC) model [15].

[2] http://konect.uni-koblenz.de/networks/

Table 1: Datasets.

Dataset	n	m	Type
Digg	30,398	85,247	Communication (directed)
Enron	87,273	320,154	Communication (directed)
Epinions	131,828	840,799	Social (directed)
Facebook	63,731	1,634,070	Social (undirected)
DBLP	1,314,050	10,724,828	Coauthorship (undirected)
YouTube	3,223,585	18,750,748	Social (undirected)
Flickr	2,302,925	33,140,017	Social (directed)

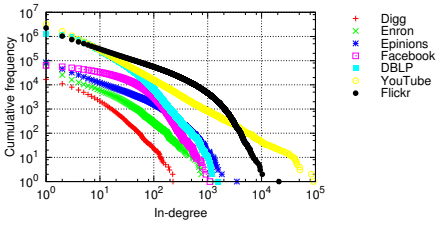


Figure 3: Degree distribution of each dataset.

Algorithms: Our method is parameterized by W, which represents the threshold of the total weight. In order to apply the same parameter for various networks, we introduce the parameter β, which determines W in a manner such that $W = \beta(n+m)\log n$. Unless otherwise specified, we set $\beta = 32$. This choice will be justified in Section 7.4 in terms of accuracy.

For the purpose of our comparison, we use the following algorithms in our experiments.

- RIS [3]: A reverse influence sampling method on which our method is based. We set the total number of traversed edges to be $32(n+m)\log n$.
- TIM^{+}[3] [29]: A sketch-based method with the two-phase strategy. We set the parameters as $\ell = 1$ and $\epsilon = 0.5$.
- IMM[4] [28]: A sketch-based method that uses martingales. We set the parameters as $\ell = 1$ and $\epsilon = 0.5$.
- PMC[5] [24]: A simulation-based method using pruned Monte-Carlo simulations for the IC model. The number of subgraphs is set to 200, as in [24].
- IRIE[6] [14]: A heuristic-based method that uses a linear system. We set the parameters as $\alpha = 0.7$ and $\theta = 1/320$.
- Degree: A baseline method that chooses the topmost k vertices in decreasing order of degrees.
- MC [15]: A simulation-based influence estimation method that simulates the diffusion process 10,000 times, and calculates the average number of activated vertices.

7.2 Index construction

For each network, we constructed our index using the entire network. The indexing times and index sizes are presented in Table 2. From this, we can observe the scalability and efficiency of our method. For the larger three networks, which incorporate tens of millions of edges, it only requires a few hours to construct the index. However, we note that without our dynamic method, this amount of time is required to estimate or maximize the influence spread. We here note that the difference in the index size under the WC model comes from the differences in the degree distribution. For example, YouTube has more vertices with high in-degree than DBLP. Thus, for each sketch construction, the total weight of the index for YouTube increases faster than DBLP. As a result, the obtained index for YouTube has a small number of sketches (1 million), which requires only 3.9GB, while the index for DBLP has a larger number of sketches (37 millions) and consumes 34.8GB. Table 3 reports the index sizes both with and without counter-based random number generators introduced in Section 6.3. Note that without random number generators, we need to store all the edges incident to vertices in each of H_i's. With random number generators, the space consumption is reduced by a factor from 4 to 50. Figure 4 depicts the change of the indexing time and index size with the value of β from 2 to 2048. Both of these values are scaled to β.

7.3 Dynamic updates

In this section, we evaluate the efficiency of our methods in terms of dynamic updates. Specifically, we have measured the running time of each operation as follows.

- Vertex additions: The average running time for adding 1,000 new isolated vertices to the index constructed using the whole network.
- Vertex deletions: The average running time for deleting 1,000 uniformly chosen vertices from the index constructed using the whole network.
- Propagation probability updates: The average running time for updating the propagation probabilities of 1,000 uniformly chosen edges of the index constructed using the whole network. The update is conducted as follows. Suppose that an edge e with propagation probability p_e is chosen. When using the TR model, we randomly choose a probability from $\{0.1, 0.01, 0.001\} \setminus \{p_e\}$ as the new edge probability of e. When using the WC model, we randomly assign $p_e \times 2$ or $p_e/2$ to e.
- Edge additions: The average running time for adding the final 1,000 edges to the index constructed using all of the edges except for the final 1,000 edges.
- Edge deletions: The average running time for deleting 1,000 edges in the reverse of the order that they were added, from the index constructed using the whole network.

Table 2 presents the average running times of the dynamic update operations. Each vertex addition and edge deletion is processed within a few milliseconds. An edge addition requires several ten milliseconds for the three largest networks, under the TR setting, and requires a few milliseconds under the WC setting. The reason for this is that under the TR model, sketches are likely to expand more following the addition of edges in comparison with the WC model. Vertex deletion exhibits a similar tendency to edge addition, where it becomes slower for the probability setting of TR. Figure 4 suggests that the average processing times of dynamic operations are roughly proportional to the value of β, because the expected size of I is proportional to β.

[3] http://sourceforge.net/projects/timplus/
[4] http://sourceforge.net/projects/im-imm/
[5] https://github.com/todo314/pruned-monte-carlo/
[6] The code is provided by Kyomin Jung, an author of [14].

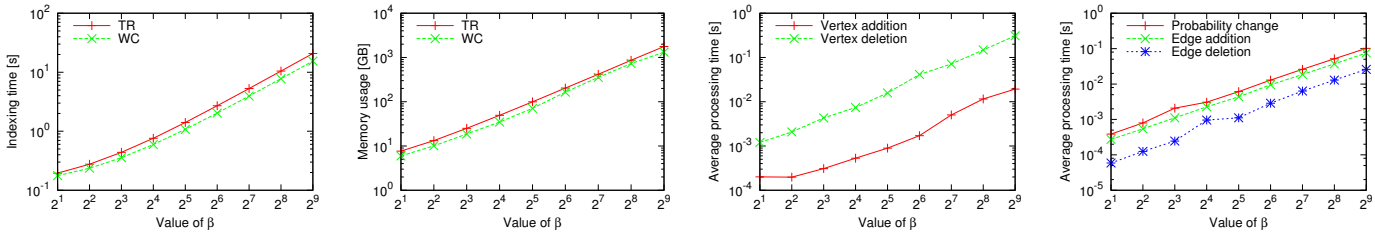
Table 2: Indexing time, index size, and average processing times of dynamic updates.

Dataset	Model	Indexing time	Index size	Vertex addition	Vertex deletion	Probability change	Edge addition	Edge deletion
Digg	TR	36.3 s	3.0 GB	2.6 ms	2.9 ms	0.59 ms	0.32 ms	0.39 ms
	WC	20.3 s	1.4 GB	1.6 ms	4.0 ms	0.92 ms	0.81 ms	1.3 ms
Enron	TR	27.3 s	0.6 GB	0.46 ms	8.1 ms	3.4 ms	1.2 ms	1.2 ms
	WC	16.5 s	0.6 GB	0.33 ms	4.6 ms	1.4 ms	0.39 ms	0.94 ms
Epinions	TR	89.1 s	1.4 GB	0.80 ms	14.8 ms	5.8 ms	4.1 ms	1.0 ms
	WC	62.2 s	1.1 GB	0.69 ms	8.3 ms	1.7 ms	1.0 ms	1.8 ms
Facebook	TR	165.0 s	2.1 GB	2.7 ms	95.8 ms	11.1 ms	6.9 ms	0.75 ms
	WC	135.2 s	1.6 GB	2.7 ms	61.0 ms	3.6 ms	1.2 ms	2.3 ms
DBLP	TR	2,965.9 s	27.7 GB	5.2 ms	124.0 ms	85.7 ms	42.5 ms	2.1 ms
	WC	3,395.7 s	34.8 GB	4.0 ms	18.1 ms	2.7 ms	0.87 ms	1.3 ms
YouTube	TR	5,000.4 s	44.6 GB	0.01 ms	92.2 ms	236.2 ms	31.8 ms	0.26 ms
	WC	1,985.5 s	3.9 GB	0.65 ms	5.7 ms	1.5 ms	0.14 ms	0.04 ms
Flickr	TR	5,467.6 s	31.3 GB	0.00 ms	459.0 ms	125.2 ms	89.6 ms	2.4 ms
	WC	4,253.7 s	12.2 GB	2.1 ms	53.8 ms	4.8 ms	0.19 ms	0.08 ms

Table 3: Effectiveness of the proposed techniques compared to naive implementations.

		Edge deletion		Vertex deletion		Vertex addition		Index size	
Dataset	Model	Sec. 6.1	Naive	Sec. 6.1	Naive	Sec. 6.2	Naive	Sec. 6.3	Naive
Epinions	TR	1.0 ms	163.1 ms	14.8 ms	575.3 ms	0.80 ms	1.6 ms	1.4 GB	5.5 GB
	WC	1.8 ms	1.4 ms	8.3 ms	7.4 ms	0.69 ms	6.4 ms	1.1 GB	7.0 GB
YouTube	TR	0.26 ms	597.9 ms	92.2 ms	$>$ 10,000.0 ms	0.01 ms	5.6 ms	44.6 GB	250.0 GB
	WC	0.04 ms	0.05 ms	5.7 ms	4.4 ms	0.65 ms	5.9 ms	3.9 GB	179.7 GB
Flickr	TR	2.4 ms	1,705.5 ms	459.0 ms	$>$ 10,000.0 ms	0.00 ms	6.5 ms	31.3 GB	* $\approx$ 282.0 GB
	WC	0.08 ms	0.10 ms	53.8 ms	41.2 ms	2.1 ms	33.6 ms	12.2 GB	* $\approx$ 292.1 GB

* We report twice the index size for $\beta = 16$ as an approximation of that for $\beta = 32$.



(a) Indexing time (b) Index size (c) Vertex operations (TR) (d) Edge operations (TR)

Figure 4: The change of indexing time, index size, and average processing times of dynamic updates with the increase of β on Epinions.

Next, we analyze the effectiveness of the proposed speed-up techniques introduced in Section 6.1 and 6.2. Table 3 presents the average running times of dynamic updates both with and without the proposed pruning techniques. The proposed techniques improve the performances of edge deletion, vertex deletion, and vertex addition, making them up to 2,000, 100, and 500 times faster, respectively. Notice that our techniques for deletion operations have more effectiveness for larger networks.

7.4 Influence estimation

Now, we will show that our method efficiently and accurately estimates the influence spread using the index constructed from a given graph. As the exact computation of the influence spread is #P-hard, we regard the estimate given using MC as the ground truth of the influence spread.

First, we focus on the influence estimation for a single vertex. For each network, we randomly sampled 1,000 vertices, and then estimated the influence spread for each vertex. Table 4 presents the average estimation time for each method. The average query time for our method is of the order of a few microseconds, which is several orders of magnitude faster than both RIS and MC. Therefore, once we construct our index, we can perform efficient tracking of influential vertices. Figure 5 indicates that the average estimation time is robust against changes in β, because it requires only constant time, as described in Section 4.3.1.

Figure 7 illustrates the accuracy of our method, where each point corresponds to a seed set consisting of a single vertex, and the x and y coordinates represent the influence

Table 4: Average running time for estimating the influence spread of a single vertex.

Dataset	Model	This work		Static methods	
		Indexing	Query	MC	RIS
Epinions	TR	89.1 s	0.97 µs	6.3 s	8.7 s
	WC	62.2 s	0.96 µs	0.01 s	9.3 s
DBLP	TR	2,965.9 s	1.62 µs	48.0 s	267.1 s
	WC	3,395.7 s	1.41 µs	0.02 s	298.1 s

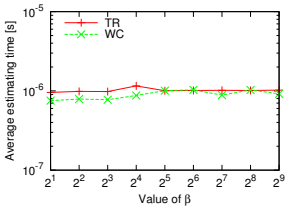


Figure 5: Average times for estimating influence of a single vertex with β on Epinions.

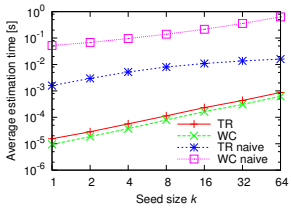
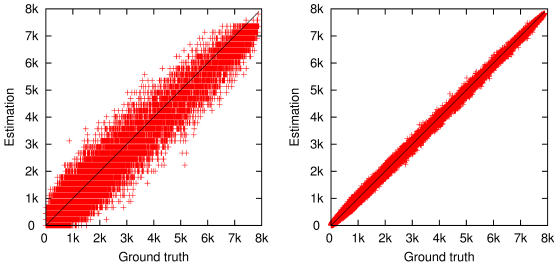


Figure 6: Average times for estimating influence of a vertex set of various sizes on Epinions.

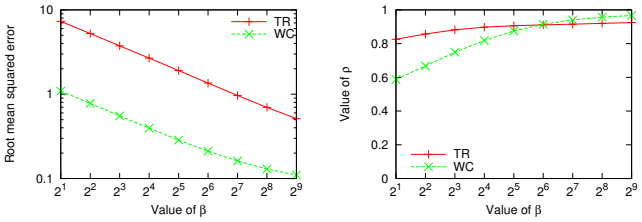


(a) Epinions (TR, $\beta = 1$) (b) Epinions (TR, $\beta = 32$)

Figure 7: Correlation between the ground truth and influence estimation calculated by our method.

spreads computed by MC and our method, respectively. It can be seen that as β increases, our method becomes more accurate. Even when $\beta = 1$, our method is stable, in the sense that all the points are close to the diagonal (i.e., the line of $y = x$) and we do not have any outliers. This property is desirable, because such outliers would result in huge errors when we want to maximize the influence spread. We here justify our choice of parameter β. We plotted the root mean squared error and Spearman's rank correlation coefficient between the influence estimations of our method with various values of β and those from MC, as presented in Figure 8. Higher values of β yield more accurate influence estimations, but the improvements are limited when $\beta > 32$. Because the efficiency of index construction and dynamic updates depends on the value of β, we adopted $\beta = 32$ as a sweet spot between accuracy and efficiency.

Next, we evaluate the efficiency of the influence estimation for a set of multiple vertices. We randomly generated 1,000 vertex sets of a specific size, and then estimated the influence spread for each vertex set. Figure 6 presents the average estimation times for a vertex set of sizes ranging from 1 to 64. The estimation times are scaled to the seed set size, and under one millisecond is required for a seed set of size 64, which is 10–100 times faster than the required times without our speed-up technique.



(a) Root mean squared error (b) Rank correlation coefficient

Figure 8: Accuracy improvements of influence estimation with the increase of β on Epinions.

7.5 Influence maximization

Finally, we will demonstrate that our method processes influence maximization queries efficiently and accurately using the index constructed from a given graph.

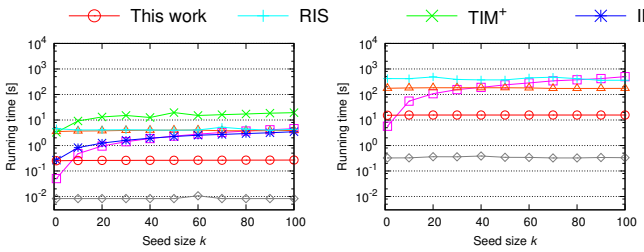
Figure 9 summarizes the running times required to compute seed sets of sizes $1, 10, 20, \ldots, 100$ after reflecting all of the edges in each network. Note that the running times do not include the times needed to read the input graph from a secondary storage location. Both TIM$^+$ and IMM did not finish within two hours on Flickr (TR, $k \geq 1$). Our method returns a seed set within 20 seconds for all of the settings, whereas other state-of-the-art static methods require a time of at least one order of magnitude longer. It should be noted that finding a seed set of the same quality that our method delivers from scratch requires ten times longer, as the performance of RIS demonstrates. We can also observe the robustness of our method against the seed size k, while TIM$^+$, IMM, and IRIE become slower as k increases.

Figure 10 presents the influence spread for seed sets of sizes $1, 10, 20, \ldots, 100$, as computed using each method. As we can see, our method and RIS deliver almost the same quality. This is not a coincidence, because we have a theoretical guarantee that both our method and RIS generate indices sampled from the same distribution. TIM$^+$, IMM, and PMC also gave seed sets of a similar quality to our method, and this is because they also have accuracy guarantees. IRIE and Degree perform comparatively badly on Enron (TR).

8. CONCLUSIONS

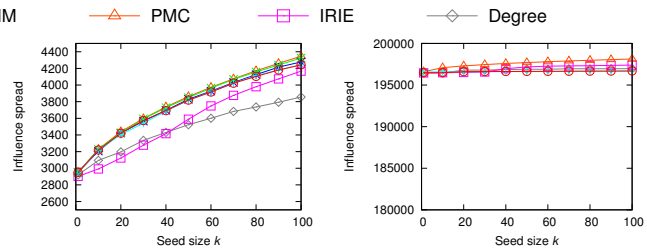
In this paper, we proposed the first fully-dynamic index data structure for influence analysis on an evolving network. Our indexing method can instantly incorporate graph updates of any kind, and it can efficiently answer the two kinds of queries, influence estimation and influence maximization, on the latest graph snapshot. Our thorough theoretical analysis guarantees the non-degeneracy of our update algorithms and solution accuracy for our query algorithms. Moreover, we developed several speed-up techniques that drastically reduce both the time and space consumptions.

Experimental results on real dynamic networks demonstrated that our method is able to update the index several orders of magnitude faster than reconstructing an index from scratch owing to the proposed speed-up techniques. Furthermore, comparing with existing static methods for influence estimation and influence maximization, we verified the efficiency and accuracy of our query algorithms.



(a) Enron (TR) (b) Flickr (TR)

Figure 9: Running times for extracting a seed set of size from 1 to 100 for each algorithm.



(a) Enron (TR) (b) Flickr (TR)

Figure 10: The influence spreads of a seed set of size from 1 to 100 computed by each algorithm.

Acknowledgments. N. O. was supported by JSPS Grant-in-Aid for JSPS Research Fellow (No. 16J09440). T. A. was supported by JSPS Grant-in-Aid for Research Activity Startup (No. 15H06828) and JST, PRESTO. Y. Y. was supported by JSPS Grant-in-Aid for Young Scientists (B) (No. 26730009), MEXT Grant-in-Aid for Scientific Research on Innovative Areas (No. 24106003), and JST, ERATO, Kawarabayashi Large Graph Project.

9. REFERENCES

[1] E. Adar and L. A. Adamic. Tracking information epidemics in blogspace. In *WI*, pages 207–214, 2005.

[2] A. Anagnostopoulos, R. Kumar, and M. Mahdian. Influence and correlation in social networks. In *KDD*, pages 7–15, 2008.

[3] C. Borgs, M. Brautbar, J. Chayes, and B. Lucier. Maximizing social influence in nearly optimal time. In *SODA*, pages 946–957, 2014.

[4] W. Chen, C. Wang, and Y. Wang. Scalable influence maximization for prevalent viral marketing in large-scale social networks. In *KDD*, pages 1029–1038, 2010.

[5] W. Chen, Y. Wang, and S. Yang. Efficient influence maximization in social networks. In *KDD*, pages 199–208, 2009.

[6] X. Chen, G. Song, X. He, and K. Xie. On influential nodes tracking in dynamic social networks. In *SDM*, pages 613–621, 2015.

[7] J. Cheng, L. Adamic, P. A. Dow, J. M. Kleinberg, and J. Leskovec. Can cascades be predicted? In *WWW*, pages 925–936, 2014.

[8] S. Cheng, H. Shen, J. Huang, G. Zhang, and X. Cheng. StaticGreedy: Solving the scalability-accuracy dilemma in influence maximization. In *CIKM*, pages 509–518, 2013.

[9] P. Domingos and M. Richardson. Mining the network value of customers. In *KDD*, pages 57–66, 2001.

[10] N. Du, L. Song, M. Gomez-Rodriguez, and H. Zha. Scalable influence estimation in continuous-time diffusion networks. In *NIPS*, pages 3147–3155, 2013.

[11] J. Goldenberg, B. Libai, and E. Muller. Talk of the network: A complex systems look at the underlying process of word-of-mouth. *Marketing Letters*, 12(3):211–223, 2001.

[12] J. Goldenberg, B. Libai, and E. Muller. Using complex systems analysis to advance marketing theory development: Modeling heterogeneity effects on new product growth through stochastic cellular automata. *Academy of Marketing Science Review*, 9(3):1–18, 2001.

[13] A. Goyal, F. Bonchi, and L. V. Lakshmanan. Learning influence probabilities in social networks. In *WSDM*, pages 241–250, 2010.

[14] K. Jung, W. Heo, and W. Chen. IRIE: Scalable and robust influence maximization in social networks. In *ICDM*, pages 918–923, 2012.

[15] D. Kempe, J. Kleinberg, and É. Tardos. Maximizing the spread of influence through a social network. In *KDD*, pages 137–146, 2003.

[16] M. Kimura, K. Saito, and R. Nakano. Extracting influential nodes for information diffusion on a social network. In *AAAI*, pages 1371–1376, 2007.

[17] R. Kumar, J. Novak, and A. Tomkins. Structure and evolution of online social networks. In *Link Mining: Models, Algorithms, and Applications*, pages 337–357. Springer, 2010.

[18] J. Leskovec, J. Kleinberg, and C. Faloutsos. Graph evolution: Densification and shrinking diameters. *ACM Transactions on Knowledge Discovery from Data*, 1(1):2, 2007.

[19] J. Leskovec, A. Krause, C. Guestrin, C. Faloutsos, J. VanBriesen, and N. Glance. Cost-effective outbreak detection in networks. In *KDD*, pages 420–429, 2007.

[20] B. Lucier, J. Oren, and Y. Singer. Influence at scale: Distributed computation of complex contagion in networks. In *KDD*, pages 735–744, 2015.

[21] M. Minoux. Accelerated greedy algorithms for maximizing submodular set functions. *Optimization Techniques*, 7:234–243, 1978.

[22] S. A. Myers, C. Zhu, and J. Leskovec. Information diffusion and external influence in networks. In *KDD*, pages 33–41, 2012.

[23] G. Nemhauser, L. Wolsey, and M. Fisher. An analysis of the approximations for maximizing submodular set functions. *Mathematical Programming*, 14:265–294, 1978.

[24] N. Ohsaka, T. Akiba, Y. Yoshida, and K. Kawarabayashi. Fast and accurate influence maximization on large networks with pruned monte-carlo simulations. In *AAAI*, pages 138–144, 2014.

[25] M. Richardson and P. Domingos. Mining knowledge-sharing sites for viral marketing. In *KDD*, pages 61–70, 2002.

[26] K. Saito, R. Nakano, and M. Kimura. Prediction of information diffusion probabilities for independent cascade model. In *KES*, pages 67–75, 2008.

[27] J. K. Salmon, M. A. Moraes, R. O. Dror, and D. E. Shaw. Parallel random numbers: as easy as $1, 2, 3$. In *SC*, pages 1–12, 2011.

[28] Y. Tang, Y. Shi, and X. Xiao. Influence maximization in near-linear time: A martingale approach. In *SIGMOD*, pages 1539–1554, 2015.

[29] Y. Tang, X. Xiao, and Y. Shi. Influence maximization: Near-optimal time complexity meets practical efficiency. In *SIGMOD*, pages 75–86, 2014.

[30] Y. Wang, G. Cong, G. Song, and K. Xie. Community-based greedy algorithm for mining top-k influential nodes in mobile social networks. In *KDD*, pages 1039–1048, 2010.

[31] H. Zhuang, Y. Sun, J. Tang, J. Zhang, and X. Sun. Influence maximization in dynamic social networks. In *ICDM*, pages 1313–1318, 2013.