ForestStream: Accurate Measurement of Cascades in Online Social Networks

Long Gong[§], Lanxi Huang[§], Paul Tune[†], Jinyoung Han[¶], Chen-Nee Chuah[‡], Matthew Roughan[†], and Jun Xu[§]
[§]Georgia Institute of Technology, [†]University of Adelaide, [¶]Hanyang University, [‡]University of California Davis
{gonglong,lanxih}@gatech.edu {paul.tune,matthew.roughan}@adelaide.edu.au jinyounghan@hanyang.ac.kr
chuah@ucdavis.edu jx@cc.gatech.edu

***Abstract*—Various Online Social Network (OSN) based applications depend on the interactions between users to disseminate information and recruit more users. The temporal evolution of adoption or *cascade* process of new products, applications or ideas is important to advertisers, OSN operators and application developers. Interactions between users are represented by massive directed graphs, so graph sampling methods were proposed to capture their properties. Existing graph sampling methods, such as a simple random walk, however, are ill-suited for capturing this and other dynamic properties of the graph.**

We propose *ForestStream*, a measurement method that relies on a combination of sampling and streaming with the goal of capturing the statistical properties of cascades in OSN graphs. We demonstrate our method's accuracy over existing methods in inferring the cascade statistics, with a low memory usage.

I. Introduction

Online Social Networks (OSNs), such as Facebook, Instagram and Pinterest, are undoubtedly a dominant presence on the Internet. Many OSN-based applications depend on the interactions between users to disseminate information, recruit more users [1], or facilitate the propagation or *adoption*, of new products, applications or ideas. Hence, understanding the dynamics and evolutions of such interactions is of great interest to researchers, advertisers, OSN operators and the application developers themselves.

There has been considerable measurement studies on the relationship graphs formed by OSNs [2–5]. These studies are focused mostly on measuring the statistical properties of these graphs. Note that friendship or relationship graphs are relatively static in nature; once a link has been established between users, it often exists for a long period of time except for those cases when ties are severed. Consequently, the properties of interest, such as the node degree distribution, remain relatively unchanged over time. However, OSN relationship graphs are invariably massive, so these measurement studies all involve *sampling* a portion of the OSN graph in various ways. The involved sampling techniques range in simplicity, from a random walk [4], to the more sophisticated Forest Fire Sampling [3]. These sampling techniques are quite accurate in estimating the statistics of interest thanks to the mostly static nature of the relationship graphs.

The focus of this work, however, is on measuring a different type of OSN graphs. More specifically, we are interested in studying user interactions in an OSN, which are often modeled as a dynamic graph, or a dynamic process over the (mostly static) OSN relationship graph, that evolves over time. In such a graph, each interaction between two users (nodes) corresponds to an edge, which is usually directed (*e.g.,* an invitation from one user to another); the graph is uniquely defined by the stream of edges that arrive over time. For instance, in a number of OSN applications, the interactions between application users form a *cascade* graph [1, 6], as information propagates outwards from one or more sources to an increasing number of users. Cascade propagation is of great interest to an application developer because understanding what triggers large cascades, and how to maintain a large cascade, would lead to more revenue for the developer.

The (dynamic) graphs that capture user interactions or activities over OSNs, which often involve multiple streams of "arriving" edges from various users, result in even more massive data sets than the (static) user relationship graphs. It is often one or more orders of magnitude larger than the amount of computer memory we have to process it. For instance, a popular Facebook gifting application, iHeart, was reported to have more than 110 million user activities within a week during its peak[1] [7]. As of April 2016, there are over 1.65 billion monthly active Facebook users, with 9 million applications and websites integrated with Facebook. Massive streams of user interaction data is observed at OSN platforms such as Facebook each day. How to process a massive graph data stream typically in one pass, using a limited amount of computer memory, to answer certain queries about the graph is an active area of research called graph streaming [8]. The key challenge in graph streaming is how to digest and remember, using a limited amount of memory, important information about the graph that is pertinent to the query, when the edges of the graph "stream by".

Measuring the aforementioned cascade graph, using a limited amount of memory, is known to be a challenging problem. It was shown in [9] that the sampling-based techniques perform poorly in capturing the statistics of the cascade graph in a popular Facebook gifting application, iHeart. More specifically, the estimates of the cascade statistics, such as the cascade size distribution, from samples of these methods are highly inaccurate. The large errors point to the lack of preservation of the properties from the samples, *i.e.,* there is

[1]At its peak in 2009, iHeart was ranked 3rd most popular application at Facebook, and was installed by 76 million users by August 2010.

978-1-5090-2991-4/17/$31.00 ©2017 IEEE

little information that could be extracted from the samples. The poor performance necessitates a better graph measurement method to preserve the OSN graph's dynamic properties.

In this work, we propose *ForestStream*, a hybrid graph streaming-sampling algorithm for accurate measurements of OSN cascade statistics, using limited memory. ForestStream departs from the sampling-only paradigm, by combining graph streaming and sampling to capture more relevant information about the cascades. Our experiments on real-world data show that, under the same memory constraint, ForestStream produces more accurate estimations of the cascade statistics than the aforementioned sampling-based techniques.

A summary of our main contributions is as follows:

- We developed a hybrid sampling-streaming method to measure the dynamic properties of the OSN graph.
- We derived an inference procedure based on the Expectation-Maximization algorithm.
- We performed rigorous comparisons between our method and existing sampling-based methods using real user activity data from a popular Facebook gifting application.

To the best of our knowledge, ForestStream is the first to meld sampling and streaming methods to measure dynamic graph properties.

II. Problem Statement

Representing the cascade graph in an accurate yet concise manner is challenging due to complex temporal orderings of user interaction events. An in-memory representation of the entire graph typically would not fit in memory. We want a representation that has a low memory requirement, yet can answer queries about the cascade graph properties, such as the number of cascades above a certain size accurately (and preferably quickly). With limited memory, however, there is a tradeoff: we have to resort to a lower-resolution representation at the cost of lower accuracies in answering queries. This is exactly what the graph sampling methods try to do by sampling only a portion of the cascade graph in the hope that the error on statistics of interest is low. Unfortunately, this does not turn out to be the case [9] for existing approaches. Our goal is to develop a method that meets the low memory constraint and preserves, to the extent possible, the statistics of the cascade graph.

A. Cascade Definition and Formulation

User interactions can be described as a directed graph $G(V, E)$, where V and E are the sets of users (nodes) and invitations (edges) respectively. A directed edge between users i and j (nodes V_i and V_j) denotes that (sender) user i *invites* (receiver) user j to use the application. In iHeart [6], for instance, this involves a user sending a gift, which are themed images or items.

A user can have many invitations (activation requests) from other users before adopting the application, *i.e.,* becoming *activated*. In iHeart, a user is activated once the user accepts a gift from another user [6]. Only after a user installs the application and accepts an invite can he/she send invites to others. Each user can invite any other user in the community even if they do not have a prior relationship.

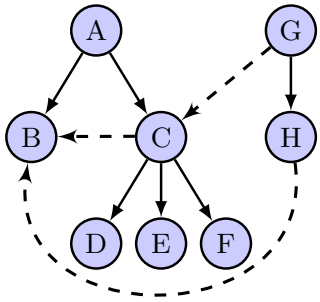


Fig. 1: Example of the first parent definition.

We enforce a relationship between inviters and activated users by defining a single parent for each activated user. The purpose of this is to identify the influential users in a cascade, so as to find the users that are responsible for triggering a cascade. Some users will adopt the application voluntarily without receiving an invite first: these important users are the *seed users*. By this definition, each cascade is a tree-like structure, with the seed user as the root.

We need a way to define which user is the parent. There are four heuristics to do so: the first parent, last parent, highest out-degree parent and random parent [6]. As there was little difference between these heuristics [6], for simplicity, we define the parent of a user j to be the first user i (in chronological order) that sent an invitation to user j, *i.e.,* the first parent heuristic.

Figure 1 demonstrates the first parent heuristic. Here, we have two cascades: (A, B, C, D, E, F) and (G, H). A directed solid-edge represents the first invite from a sender to a receiver. The dashed-lines represent invites that arrive later. For example, user B receives an invite first from user A, and subsequently from users C and H, so A is defined as B's parent in cascade formation.

Once we enforce this relationship on $G(V, E)$, it results in a *forest*, denoted as $\mathcal{F}(V, E)$, consisting of many trees. Each tree, rooted in the seed user, is a *cascade*, as seen in Figure 2. The relationship elucidates the sequence of invites and adoptions, showing the propagation of the application starting from the seed user. Leaves in the cascade are activated users who have never sent out invites or are ineffective in recruiting more users. Note that throughout the paper, we will be using the terms cascade and tree, and user and node interchangeably.

Three common properties of a cascade are:

- *size*: the number of nodes in a cascade,
- *depth*: the maximum number of levels of the cascade (a cascade with only a seed user has depth 0), and
- *width*: the maximum number of nodes amongst all levels of the cascade.

Figure 2 presents an example of a cascade. Here, the size, depth and width of the cascade are 12, 3 and 6 respectively.

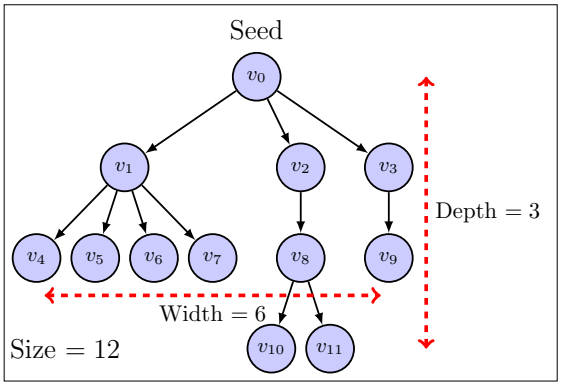


Fig. 2: Example for definition of cascade size, depth and width.

B. Cascade Size Distribution

For simplicity, we first focus on the problem of estimating the distribution of the sizes of the cascades. We will later discuss how ForestStream can be used to measure the other properties (see §III-D).

In a measurement interval, we have a total of N cascades. We assume that all cascades within the interval are *whole i.e.,* no cascade gets artificially truncated due to the finite measurement interval. Cascades can be of size 1, *i.e.,* the cascade comprises only of the seed user.

Let N_k be the number of cascades of size k. Equivalently, let $\theta_k = N_k/N$, $k = 1, \cdots, W$ be the proportion of cascades with size k, where W is the largest cascade size. Note that

$$\theta_k \geq 0, \ \forall k, \ \text{and} \ \sum_{k=1}^{W} \theta_k = 1. \tag{1}$$

Collectively, the column vector $\boldsymbol{\theta} = [\theta_1, \cdots, \theta_W]^{\mathrm{T}}$ denotes the *cascade-size distribution*, and this is what we aim to infer from our observations. The parameter set $(\boldsymbol{\theta}, N)$ is equivalent to the parameter set $\{N_k\}_{k=1}^{W}$, but we choose the former because estimation of $\boldsymbol{\theta}$ directly estimates the cascade-size distribution.

When a measurement method is applied to the cascades, the observables are subtrees of the cascades. The quality of these observables in retaining information about $\boldsymbol{\theta}$ will determine the accuracy of a measurement method. A measurement method is constrained by the amount of memory allocated to it. Our goal then is to capture as much information about the cascade-size distribution given a memory constraint.

The problem is reminiscent of the recovery of the traffic flow-size distribution [10]. Our problem is far more challenging because of the complex dependency structure between users within a cascade. If a node in a cascade is not sampled, it is conceivable that a subtree may be missing, leading to an inaccurate inference of the true size of the cascade. Also, same sized flows are structurally similar but trees of the same size can differ greatly. Moreover, more space is required to store each cascade, compared to a flow, where only the latter's key and packet/byte count is stored, so our memory constraint is more stringent. These issues guide our measurement method, which we describe next.

III. The ForestStream Method

ForestStream consists of four components:

- *sampling unit*: cascades from the OSN graph are streamed and sampled uniformly at random, and are stored in their entirety in a hash table, subject to memory constraints. During measurement, cascades stored in the hash table may be evicted if there is a lack of memory allocated to the hash table;
- *tracking unit*: each time a cascade has been evicted from the hash table, its nodes are stored in a Bloom filter [11]. This ensures that nodes belonging to evicted cascades will not be inserted back into the hash table;
- *eviction registry unit*: records, for each cascade (tree) evicted from the hash table due to lack of memory, its identifier and the number of nodes in it (*i.e.,* size) at the time of eviction, in a log $F_{\mathcal{D}}$. This information is critical for the statistical recovery of the cascade size distribution;
- *inference procedure*: based on the sampled cascades in the hash table and the size counts of evicted cascades from the eviction registry unit, an Expectation-Maximization algorithm (EM) [12] infers the statistics of interest (*e.g.,* cascade size distribution).

We detail these in the following.

A. Measurement Module

The measurement module consists of the sampling unit (the hash table), the tracking unit (Bloom filter), and the eviction registry unit. We are limited by a memory pool that allows us to only store a small portion of $G(V, E)$ (perhaps, say 10% of its total number of nodes). The eviction registry unit is tiny compared to the other two units, since each record (of an evicted tree) takes only a few bytes to store. So we consider our memory pool to be shared between the hash table and the Bloom filter.

1) The "main" program: The measurement module streams a set of user interaction logs, generated by an OSN's systems. In practice, such logs could be collected through an application programming interface (API), *e.g.,* Twitter's API [13][2]. Each log is described by the unique IDs of a sender of an invite and its receiver, with a corresponding timestamp. These logs describe the directed edges of $G(V, E)$ at each snapshot of time, thereby describing the evolution of $G(V, E)$ over time.

As user interactions are processed, the incoming nodes are inserted into a hash table based on their unique IDs for fast look-up operations (insertion and deletion). Each hash table entry is a node of a tree. Nodes of each tree are also "threaded" together using a separate first-child/next-sibling tree data structure, so that all nodes of the tree can be located in a straightforward manner when the tree needs to be evicted (to be described shortly). Each new tree is assigned a number m if it is the m-th tree processed so far by counting the number of seed user IDs seen so far.

Algorithm 1 shows the "main" program of processing user interaction logs. Each user interaction log entry, in the form

[2]For evaluation, we have a dataset donated by an application provider.

Input: $\mathcal{E}$, user interaction logs
Output: $F_{\mathcal{D}}$, list of evicted tree IDs and their features

```
// Require HT, hash table
// Require BF, Bloom filter
1  Procedure ForestStream(E)
       // V_i sends invite to V_j at time t_{i,j}
2      foreach invite event {V_i, V_j, t_{i,j}} ∈ E do
3          T ← ∅; // Default: insert nothing
4          if V_i ∉ HT and V_j ∉ HT then
               // Store sender, receiver, edge
5              if V_i ∉ BF and V_j ∉ BF then
6                  T ← ({V_i, V_j}, {V_i V_j});
               // Store sender only
7              else if V_i ∉ BF and V_j ∈ BF then
8                  T ← ({V_i}, ∅);
9              else
10                 Insert V_j into BF;
11             end
12         else if V_i ∈ HT and V_j ∉ HT then
               // Store receiver only
13             if V_j ∉ BF then
14                 T ← ({V_j}, ∅);
15             end
16         else if V_i ∉ HT and V_j ∈ HT then
               // Store sender only
17             if V_i ∉ BF then
18                 T ← ({V_i}, ∅);
19             end
20         end
           // Insert subtree; see Algorithm 2
21         InsertSubTree(T)
22     end
```

Algorithm 1: ForestStream Procedure.

$(V_i, V_j, t_{i,j})$ where $t_{i,j}$ is the timestamp when V_i sends an invite to V_j, is processed as follows. There are several cases to consider depending on whether V_i and/or V_j is currently in the hash table or has been evicted to the Bloom filter before. For example, the first case, shown in lines 5 through 6, is that neither V_i nor V_j is currently in the hash table or has been evicted to the Bloom filter before. Here, V_i is a seed user (*i.e.,* root of a new tree) and, by the first parent heuristic, the parent of V_j. Thus both vertices V_i and V_j are inserted into the hash table as hash nodes (call `InsertSubTree(`T`)` in line 21), with the parent-child relationship (*i.e.,* edge V_iV_j) reflected in the corresponding fields in both hash nodes (line 6 of Algorithm 2). Since both insertion into the hash table and querying the Bloom filter takes $O(1)$ time, it takes $O(1)$ to process each user interaction log entry, unless a tree needs to be evicted due to lack of memory.

2) Insertion of new trees: Algorithm 2 details how new trees are inserted into the hash table and, when necessary, calls Algorithm 3 on line 9 which performs the eviction of trees to free memory in the hash table. In the next section, we explain when such an eviction will be triggered and how it is carried out in an "equal opportunity" manner.

Input: T, subtree to be inserted

```
1  Procedure InsertSubTree(T)
2      i ← find index of tree that T belongs to;
3      m ← number of trees observed so far;
4      if i is not found then
5          Assign T with identifier m + 1;
6          Insert T into hash table;
7      else
8          Update tree i with T;
9      end
10     if size of hash table > τ_max then
11         EvictTrees(); // See Algorithm 3
12     end
```

Algorithm 2: Insertion routine into hash table.

In both our implementation and other competing schemes, each hash table entry (node) takes 20 bytes of memory to store (*i.e.,* 4 bytes each for the IDs of the node, its next sibling, its first child, next node along the hash chain, and the root of the tree). In our scheme alone, however, for each node in the hash table, an extra byte is required for recording the quantized (hence approximate) false positive rate of the Bloom filter when the node is inserted into the hash table; we have accounted for this extra byte when comparing the efficacy of our scheme with those of other schemes. The false positive rate information will be useful later on to estimate the size of evicted trees, as seen below.

3) Eviction of trees: The hash table's memory is limited. At some point, some cascade trees must be evicted because of memory restrictions. We also want to ensure that by the end of the measurement interval, all cascade trees were sampled with the same probability $0 < p < 1$ (*i.e.,* in the aforementioned "equal opportunity" manner).

Thus, the measurement is performed as follows. Incoming nodes are inserted into the hash table until the predefined memory allocation threshold $\tau_{\max}$ is reached. Once this happens, we select a tree *uniformly at random* among trees seen so far, including those that have already been evicted. This selection process continues until a not-yet-evicted tree is chosen. This unlucky tree, including the seed user (root) and all its descendants, will have their corresponding nodes evicted from the hash table. Multiple trees could be evicted from the hash table at a time to bring the total hash table memory used below the threshold $\tau_{\max}$. Before a tree is evicted, its root (seed node) ID and its compensated size (see Algorithm 4) is exported to the aforementioned eviction registry unit.

This process, depicted in Algorithm 3, of selecting unlucky trees to evict is a nonstandard implementation of reservoir sampling [14], where the hash table acts as a *reservoir*. The standard reservoir sampling process, through admittance sampling of newly arrived trees and resampling of surviving trees, is more computationally efficient than the depicted process (which is statistically equivalent), but is too onerous to present here.

The threshold $\tau_{\max}$ is determined by the following param-

```
1  Procedure EvictTrees()
2      m ← number of trees observed so far;
3      while size of hash table > τ_max do
4          t ← uniform{1, 2, ··· , m};
5          T_delete ← tree with identifier t;
6          if T_delete ∈ hash table then
               // See Algorithm 4
7              Compensation(T_delete);
8              Record root ID and compensated size of
                T_delete to F_D;
9              Delete T_delete from hash table;
10             Insert T_delete into Bloom filter;
11         end
12     end
```

Algorithm 3: Eviction routine of hash table.

eters:

- total memory budget for the measurement study, in terms of what would cost to store a certain percentage of nodes from $G(V, E)$ (*e.g.,* 10% the nodes of $G(V, E)$),
- the proportion of memory allocated to the hash table (*e.g.,* 50% for the hash table and the other 50% for the Bloom filter), and
- the cost of storing each node in memory.

The probability p with which a tree survives in the hash table cannot be determined in advance because it depends on the parameters of the measurement module and the order in which nodes and edges are read from $\mathcal{F}(V, E)$. However, by the properties of reservoir sampling, it is guaranteed that each tree is stored in the hash table with approximately equal probability [14]. We will provide an estimation of p later on when we are recovering the cascade size distribution.

4) The tracking (Bloom filter) unit: Later incoming nodes may belong to an evicted tree, however, so we need to ensure these nodes are not inserted back into the hash table as descendants of existing or future trees. Hence in Algorithm 1 (lines $5, 7, 13$ and 17), we check if an incoming node belongs to an evicted tree, by looking it up in a Bloom filter. More specifically, each time a tree is evicted, all of its nodes are inserted into the Bloom filter and its features (size, depth, width) are recorded in the eviction registry unit, seen in line 10 of Algorithm 3. The interaction between the "main" program and the eviction registry unit allows us to track (the sizes of) the deleted trees.

At the end of the measurement interval, the eviction registry unit outputs a log file $F_{\mathcal{D}}$, a record of trees that were evicted from the hash table throughout the measurement process. $F_{\mathcal{D}}$ is processed offline to obtain $\mathcal{D}$, the set of the number of counts of trees truncated to size j, $j = 1, \cdots, W$, because of eviction, and will be use in the inference procedure.

The tradeoff of less memory usage is that a Bloom filter has a *false positive error*: the filter may report that an incoming node belongs to an evicted tree when in fact it does not, resulting in misclassification. The false positive error is

```
   Input: T_delete, evicted tree
1  Procedure Compensation(T_delete)
2      L ← leaf nodes of T_delete;
3      Initialize: S^(0) ← L, Size ← 0;
       // Iterate while not at cascade seed
4      while S^(j) ≠ ∅ do
5          foreach V_k ∈ S^(j) do
6              if V_k ∈ L then // Leaf node
7                  w_{V_k} ← 1/(1−f_k);
8              else
9                  w_{V_k} ← (Σ_{i∈C(V_k)} w_i)/(1−f_k);
10             end
11             Size ← Size + w_{V_k};
12         end
13         S^(j+1) ← {P(V)}_{V∈S^(j)} ; // Update
14     end
15     Size ← Size + 1;
16     return Size;
```

Algorithm 4: Compensation Procedure.

exacerbated as more trees are inserted into the Bloom filter. The error has two consequences. First, some legitimate seed nodes are misclassified, so some of these trees are lost. Second, nodes belonging to trees in the reservoir are misclassified as belonging to that of an evicted tree, misrepresenting the size of trees stored in the reservoir.

We cannot compensate for the former error (aside from increasing memory allocation to the Bloom filter to lower the false positive error), but we can do something about the latter. This is the reason we have `Compensation(`T_{delete}`)` on line 7 of Algorithm 3. When a node is inserted into the hash table, the false positive rate at the time is also recorded in the aforementioned extra byte in its hash table entry. Whenever a tree is selected for eviction, line 7 is performed to estimate its size (and other features) based on the false positive error (see Algorithm 4).

5) Compensation Procedure: Algorithm 4 presents the compensation procedure. We reconstruct the sizes of the subtrees starting from the leaf nodes to estimate the full size of the cascade. Suppose V_k is a leaf node and let f_k be the Bloom filter false positive rate that it fortunately did not "succumb to" (*i.e.,* survived). Let $\mathcal{P}(V_k)$ denote the parent of V_k.

Intuitively, each such "survivor" V_k statistically represents $w_{V_k} = 1/(1-f_k)$ nodes that include $f_k/(1-f_k)$ non-survivors and itself. With any non-leaf node V_ℓ, we propagate estimates from the bottom up, *i.e.,* from the leaf nodes all the way to the seed node, given by

$$w_{V_\ell} = \frac{\sum_{i \in \mathcal{C}(V_\ell)} w_i}{1 - f_\ell},$$

where $\mathcal{C}(V_\ell)$ are the set of children of V_ℓ. This is an unbiased estimate of the true size of the evicted tree:

Theorem 1: Algorithm 4 unbiasedly estimates the true size of T_{delete}.

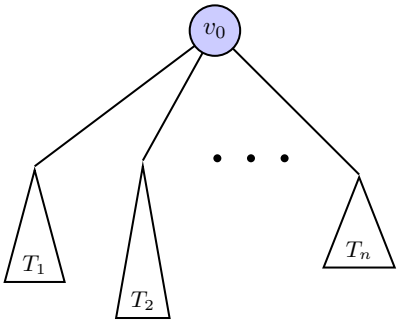


Fig. 3: Illustration of observed tree T with n subtrees T_i, $i = 1, 2, \cdots, n$, rooted at the children of the root v_0.

Proof: We prove this by induction. Let $T^\star$, T_{delete} be the true and observed trees respectively, and let $Size^\star$, $Size$ and $\widehat{Size}$ be the true, observed, and estimated (compensated) size from Algorithm 4. Let the false positive rate at the root v_0 be f_0.

For $Size^\star = 1$, the only node present is the root v_0. Note that $\Pr(Size = 1) = 1 - f_0$ and clearly $\Pr(Size = 0) = f_0$. The statement is true since we have $\widehat{Size} = \frac{Size}{1-f_0}$, and

$$\mathbb{E}[\widehat{Size}] = (1 - f_0) \cdot \frac{1}{1 - f_0} + f_0 \cdot 0 = 1.$$

Now, assume that unbiasedness holds for estimates of trees with $Size^\star \leq k$. Consider the case $Size^\star = k + 1$. Let v_0 be the root node of a tree $T^\star$ with true size $k + 1$. Let $T_1, T_2, \cdots, T_n$ be the n subtrees rooted at its children, of the observed tree T_{delete}, as shown in Figure 3. Note that any child of v_0 in T_{delete} was not misclassified.

Algorithm 4 estimates the size of each subtree T_i as $\widehat{Size_i}$, and so the estimate for of the size of $T^\star$ is

$$\widehat{Size} = \min\left\{0, \frac{1 + \sum_{i=1}^{n} \widehat{Size_i}}{1 - f_0}\right\},$$

since the estimated size can only be non-zero if v_0 was not misclassified. Since the true size of all these subtree is less than or equal to k, so $\widehat{Size_i} = Size_i^\star$, therefore,

$$\begin{aligned}\mathbb{E}[\widehat{Size}] &= (1 - f_0) \cdot \frac{1 + \sum_{i=1}^{n} \mathbb{E}[\widehat{Size_i}]}{1 - f_0} + f_0 \cdot 0 \\ &= 1 + \sum_{i=1}^{n} \mathbb{E}[\widehat{Size_i}] = 1 + \sum_{i=1}^{n} Size_i^\star \\ &= Size^\star.\end{aligned}$$

This proves unbiasedness. ∎

In practice, this is implemented as a recursive function. Starting from the seed node, the procedure traverses all the way to the leaf nodes and estimates the size of subtrees. The procedure has $O(M + e)$ time complexity for a tree with M nodes and e directed edges.

The accuracy of the compensation procedure depends on the false positive rate at each node. When the memory allocated to the Bloom filter is low relative to the number of nodes inserted, the accuracy degrades.

Input: W, maximum cascade size
Input: p, probability tree in the hash table
Input: $\mathcal{H} = \{H_i\}_{i=1}^{W}$, hash table tree sizes
Input: $\mathcal{D} = \{D_j\}_{j=1}^{W}$, evicted tree sizes
Input: ϵ, error threshold for loop termination
Input: C, reliable size threshold
Output: $\widehat{\boldsymbol{\theta}}$, estimated cascade-size distribution

$\widehat{\boldsymbol{\theta}}^{(0)} \leftarrow \mathbf{0}_W, \widehat{\boldsymbol{\theta}}^{(1)} \leftarrow \boldsymbol{\theta}_{\text{init}}, \ell \leftarrow 1;$
$\widehat{N}_k \leftarrow h_k/p$, for $k = 1, \cdots, C;$
// While termination condition not met
while $\|\widehat{\boldsymbol{\theta}}^{(\ell)} - \widehat{\boldsymbol{\theta}}^{(\ell-1)}\|_2 \geq \epsilon$ **do**
  // **E-step**: Decode each evicted cascade
  for $k := 1$ **to** W **do**
    for $j := 1$ **to** k **do**
      $\widehat{N}_k^{(\ell+1)} = \widehat{N}_k^{(\ell)} + \frac{k^{-1}\widehat{\theta}_k^{(\ell)}}{\sum_{i=j}^{W} i^{-1}\widehat{\theta}_i^{(\ell)}} D_j;$
    end
  end
  // **M-step**: Update $\widehat{\boldsymbol{\theta}}$ sequence
  for $k := C$ **to** W **do**
    $\widehat{\theta}_k^{(\ell+1)} = \frac{\widehat{N}_k^{(\ell+1)}}{\sum_{k'=1}^{C-1} \widehat{N}_{k'} + \sum_{\ell=C}^{W} \widehat{N}_\ell};$
  end
  $\ell \leftarrow \ell + 1;$
end

Algorithm 5: EM for cascade-size distribution recovery.

B. Inference Procedure

Once the measurements are collected, we have to infer the cascade-size distribution. Let $h_i \in \mathcal{H}$ be the number of cascades of size i (after the false positive compensation) observed in the hash table. These cascades are the survivors of the hash table's eviction process. There is less ambiguity for these trees than the evicted trees since more information about them are known. Let $d_j \in \mathcal{D}$ be the observed number of evicted cascades of size j. Since a cascade can be evicted from the hash table before it grows to its true size, we know that the true size of the cascade is at least size j.

Previously, it was mentioned that p can only be determined once measurement is over, since p depends on the dataset. Its value is important to recovery, and can be approximated as

$$p = \frac{\sum_{i=1}^{W} h_i}{\sum_{i=1}^{W} h_i + \sum_{j=1}^{W} d_j}.$$

It is an approximation since some seed nodes may be misclassified. However, with a low enough false positive rate, this estimate is quite accurate.

For inference, we implement an EM algorithm, presented in Algorithm 5 [12]. We omit the derivation from the paper due to space constraints, but derivation is straightforward.

We have to choose an initial guess $\boldsymbol{\theta}_{\text{init}}$ so we use

$$\boldsymbol{\theta}_{\text{init}} = \left[\frac{h_1}{\sum_{k=1}^{W} h_k}, \cdots, \frac{h_W}{\sum_{k=1}^{W} h_k}\right]^{\mathrm{T}}. \tag{2}$$

This can be shown to be the MLE if only the hash table observations were used. The estimate will also form the basis of the estimator for ForestStream's observations.

Lines 4 to 8 implement the Expectation or E-step. Here, the E-step computes an estimate of the number of trees of size $k = 1, \cdots, W$. Line 6,

$$\frac{k^{-1}\widehat{\theta}_k^{(\ell)}}{\sum_{i=j}^{W} i^{-1}\widehat{\theta}_i^{(\ell)}},$$

is an estimate of the probability that a cascade of size k is truncated to a cascade of size j due to eviction. This a consequence of the following (conditioned on the observations from the hash table), for $j = C, \cdots, W$,

$$\mathbf{d} = \mathbf{B}\boldsymbol{\theta},$$

where $\mathbf{d}$ is a vector with $d_j := \Pr(\text{truncated cascade size } j)$ and

$$\begin{aligned}[\mathbf{B}]_{j,k} &:= \Pr(\text{truncated cascade size } j \,|\, \text{cascade size } k) \\ &= \begin{cases} k^{-1}, & j \le k, \\ 0, & \text{otherwise}, \end{cases} \end{aligned} \tag{3}$$

for all $j = 1, \cdots, W$ and $k \ge C$. We used a parameter C, where we deliberately use only partial information from $\mathcal{D}$ is used, and we explain this below in §III-C. Our recovery procedure uses a specific choice of $\mathbf{B}$, where a uniform distribution is assumed for each column of $\mathbf{B}$.

Lines 9 to 11 implement the Maximization or M-step. This computes the estimate of θ_k subject to the constraints (1). The derivation involves a straightforward evaluation via Lagrangian multipliers. The recovery procedure then alternates between the E- and M-step until a convergence criterion (line 3) is met.

Computationally, our recovery procedure has $O(W^2)$ time complexity. Since it is an instance of an EM algorithm, its convergence speed can be improved several ways through acceleration, for instance [15]. Parallelization is also possible to speed up the procedure.

C. Refining Tail Estimates

The tail of the cascade size distribution is difficult of capture. We included a parameter C here for a strong reason. The estimate (2) from the hash table observations is an *empirical histogram*, so there are binning effects towards the tail-end in general due to small sample effects. Often, in OSN graph data, the tail-end of the cascade-size distribution has few observations, as the distributions follow a power-law type distribution, as seen in §IV.

The estimates for these sizes are often unreliable, so we resort to computing a reliable size threshold C and use the evicted tree information to fill in the gaps for sizes for C and above, via the uniform model for $\mathbf{B}$.

We exclude the sparser tail end by computing the cumulative sum of the histogram (2) and taking only a portion of the cumulative sum, say $1 - \alpha$, $\alpha > 0$ of the sum, typically $\alpha = 1\%$. The size at which we reach this sum will be the reliable size threshold C. This effectively assigns the estimates C to W as a single bin to reduce noise [16], on which the uniform model choice on $\mathbf{B}$ will be used.

On the other hand, we do not use all the information from the set $\mathcal{D}$, simply because it is difficult to come up with a good model for the "after-life" (*i.e.*, after-eviction) growth of the evicted trees, because such a model depends on specific information that we do not have such as when they first start growing and when they stop.

D. Extensions

ForestStream can be extended to other metrics such as the cascade-depth and outdegree distribution. Some modifications are required for these metrics.

For the cascade-depth distribution, just information from the evicted trees is unlikely to provide accurate estimates about the depth distribution. Eviction destroys a lot of information about the final depth of the tree, due to the non-linear structure of a tree. More information is required in the form of measurements on the arrival process of the cascades and their growth rates. This can be used to construct a rudimentary model of the evolution of the cascade process. If such information is unavailable, it would be better off to utilize just the observations from the hash table.

For the outdegree distribution, the memory for the Bloom filter has to be adequately provisioned to ensure that Algorithm 4 is reliable. As we shall see in our experiments later on, the compensation procedure is quite accurate when the false positive rate is low. We can then use a similar model to recover the outdegree distribution.

IV. Evaluation

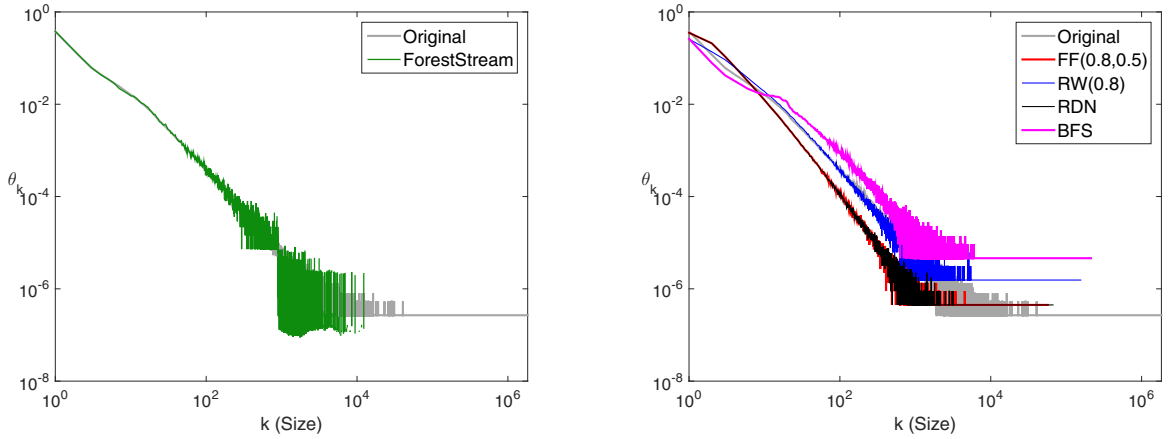
In this section, we evaluate ForestStream against well-known graph sampling methods.

A. Graph Sampling Methods

ForestStream is compared to 4 sampling methods:

- *Forest Fire (FF)*: a set of initial nodes are chosen each based on the weight of its combined degrees over all nodes in the dataset. For selected nodes their child and parent nodes are selected to visit based on the forward or backward burning probability respectively. The forward and backward burning probabilities are $p_f = 0.8$ and $p_b = 0.5$[3]. The method proceeds until all selected nodes are visited or the sample size has been reached.
- *Random Walk (RW)*: multiple random walks are started from a set of initial nodes chosen each based on the weight of its degree (in- plus out-degree) for all nodes. Starting from these seed nodes, the random walks traverse their parent and child nodes randomly proportional to the weight of their degrees. Each step of the sampling has probability $p_r = 0.2$ to return to the starting point of its random walk, and proceeds until all possible paths

[3]We use the definition in [17] which advocates the Geometric distribution rather than the Binomial distribution in [18] because the former paper defined Forest Fire more precisely.



(a) ForestStream against the ground truth

(b) The 4 graph sampling methods against the ground truth

Fig. 4: Performance comparison on the 10% sampling budget.

have been visited or the sample has reached its predefined maximum size.

- *Random Degree Node (RDN)*: nodes are sampled according to a weighted function of its combined degree, and the samples are vertex-induced subgraphs of the sampled nodes, and
- *Breadth First Sampling (BFS)*: a set of initial nodes are chosen uniformly at random over all nodes and nodes that are traversed in a breadth-wise fashion until it traverses all nodes (starting from the initial nodes) or the sample size is met.

The observations of these methods are the user ID of an observed node and its parent. These observations are then used to construct the sampled trees for estimation.

It is difficult to derive an optimal unbiased estimator for the 4 graph sampling methods because of their inherent dependence on the underlying structure of the OSN graph. It requires the development of the stochastic process of how the method traverses the OSN and a graph model of the OSN. For simplicity, the estimators are the empirical histogram of their observations. Let s_k, $k = 1, \cdots, W$ be the sampled trees of size k obtained from a graph sampling method. The estimator is then

$$\widehat{\theta_k} = \frac{s_k}{\sum_{\ell=1}^{W} s_\ell}, \ \forall k. \tag{4}$$

The optimal estimator of the cascade-size distribution for each method remains an open question.

B. Dataset

We evaluate all methods on a gifting application, iHeart, launched in June 2009 on Facebook by Manakki, LLC at the time of data collection. The dataset was anonymized and donated by Manakki, LLC, covering the major lifespan of the application (58 weeks since its initial launch till its decline in popularity). It contains more than $2,027,475,219$ entries of user activities generated by $189,988,908$ users. Each user activity record includes the sender's and recipient's anonymized Facebook user IDs as well as the timestamp when the sender sends an invitation to the recipient.

Here, $N = 3,734,967$, each rooted at a unique seed. Note that there are $1,409,508$ users who installed iHeart without prior invitations but have not recruited any users in our dataset. These are the cascades of size 1. The largest tree has size $W = 1,830,226$.

C. Setup

For fair comparison, all methods were given the same amount of memory. We set ForestStream's memory pool to be equal to that required to store 10% and 15% of the total users (nodes) in iHeart. We call these the 10% and 15% *sampling budget* cases. A single node sampled from the graph sampling methods requires 20 bytes. ForestStream has a slight additional overhead due to having to record the false positive rate. Compared to ForestStream, a graph sampling method can sample slightly more nodes: 10.53% and 15.63% respectively for the 10% and 15% sampling budget cases.

For the following, the memory pool for ForestStream is divided in half[4], with 50% allocated to the hash table and Bloom filter respectively. Note that, when saying 50%, we ignore the one extra byte for the false positive rate record. Therefore, the actual memory allocated to hash table is slightly larger than 50%. For both sampling budget cases, we use 5 hash functions per item in the Bloom filter, so the maximum false positive rate is 3.12%. The bits for the false positive rate record are the slight additional overhead we have accounted for in comparing ForestStream with the graph sampling methods.

For recovery, we plotted the cumulative sum of the empirical histogram from the hash table estimates (see formula (2)) and set $\alpha = 0.5\%$. We obtained $C = 991$ and $C = 1000$ as the reliable size threshold for the 10% and 15% sampling budget cases respectively. Here, the probability a cascade remains in the hash table by the end of the measurement interval is $p =$

[4]We also investigate other memory allocations, *e.g.,* 60% for hash table. The results are omitted due to the space constraints.

TABLE I: ℓ_2 error comparison between ForestStream against the 4 sampling methods for 10% and 15% sampling budgets.

Method	$\|\boldsymbol{\theta} - \widehat{\boldsymbol{\theta}}\|_2$ (%)	
	10% budget	**15% budget**
ForestStream	0.47	0.35
FF(0.8,0.5)	10.85	9.62
RW(0.8)	12.69	10.30
RDN	10.80	9.61
BFS	12.64	12.81

0.032 and 0.077 for the 10% and 15% scenarios respectively. These will be used in the estimation process.

Computationally, streaming all iHeart user interactions took 6 hours on a machine with a 3.5 GHz processor and 8 GB of RAM. Estimating the cascade size distribution took 9 hours on average. However, the measurement module and the recovery algorithm were not optimized, so further performance gains are possible.

D. Empirical Comparison Results

Figure 4 presents the estimates from ForestStream and the 4 graph sampling methods for the 10% sampling budget[5]. Already, we can clearly see that ForestStream's estimates are superior to the other graph sampling methods, as its estimates track the true cascade-size distribution very well.

For a clearer comparison, the error comparison is found in Table I. Performance is measured by the ℓ_2 error:

$$\|\boldsymbol{\theta} - \widehat{\boldsymbol{\theta}}\|_2 = \sqrt{\sum_{k=1}^{W} (\theta_k - \hat{\theta}_k)^2}.$$

Here, we can see that ForestStream outperforms all the other methods by a large margin with the same memory pool size. The chief advantage of ForestStream is the higher quality samples from the hash table that results in accurate estimation for most of the cascade-size distribution.

From Figure 4 we can see that the tail end (corresponding to large-tree counts) estimates of ForestStream has high variance, although the front end (corresponding to small-tree counts) is estimated very well. A contributing factor to this high variance at the tail end is that, due to the memory constraints, very large trees are unlikely to be stored in the hash table. For example, in our dataset, there is a single cascade of size $1,830,226$; though it was stored for while in the hash table, it ultimately was evicted at around size $250,000$ due to memory constraints. To the extent possible, estimates based on uniform model compensate for the loss of large-tree counts, smooth out the overall distribution and provide more intuition about how large a cascade can get.

Conceivably, if large trees could be sampled with higher probabilities than small trees, and the total memory footprint kept the same, we might be able to obtain better tail end estimates. Such a preferential sampling however is hard to achieve because large trees are difficult to spot in advance (without prior knowledge of their early growth statistics).

[5]Note that, we omit the results for 15% sampling budgets in the interest of space, since the story remains intact.

Although the other sampling methods do sample the large trees, they only sample portions of these trees. In Figure 4(b), we can see that none of the sampling methods obtain a good estimate of the large trees, because there is no way the missing portions of the large cascades can be estimated without some form of prior knowledge.

V. Conclusion

Measuring the dynamic properties of an OSN is a challenging task due to the massive size of these graphs. However, they are of importance to both researchers and application developers. In this paper, we propose ForestStream, a sampling-streaming hybrid measurement method to capture the dynamic properties of an OSN. We show empirically that ForestStream outperforms four existing graph sampling methods in capturing these information. Moreover, ForestStream is able to do this even when given the same amount of memory as the other graph sampling methods. Future work will focus on developing better statistical models to improve the inference accuracy for various dynamic functions of the OSN graph.

Acknowledgment: This work was supported in part by US NSF grant CNS-1302197.

References

[1] H. Liu, A. Nazir, J. Joung, and C.-N. Chuah, "Modeling/predicting the evolution trend of OSN-based applications," in *Proc. of WWW*, 2013.
[2] M. Kurant, A. Markopoulou, and P. Thiran, "Towards unbiased BFS sampling," *IEEE J. Sel. Areas Commun*, vol. 29, no. 9, pp. 1799–1809, Oct. 2011.
[3] J. Leskovec and C. Faloutsos, "Sampling from large graphs," in *Proc. of KDD*, August 2006.
[4] L. Lovász, "Random walks on graphs: A survey," *Combinatorics, Paul Erdös is Eighty*, vol. 2, no. 1, pp. 1–46, 1993.
[5] B. Ribeiro and D. Towsley, "Estimating and sampling graphs with multidimensional random walks," in *Proc. of IMC*, Nov. 2010.
[6] M. R. Rahman, P. A. Noël, C.-N. Chuah, B. Krishnamurthy, R. M. D'Souza, and S. F. Wu, "Peeking into the invitation-based adoption process of OSN-based applications," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 1, pp. 21–27, Jan. 2014.
[7] A. Nazir, A. Waagen, V. S. Vijayaraghavan, R. D. C-N. Chuah, and B. Krishnamurthy, "Beyond friendship: Modeling user activity graphs on social network based applications," in *Proc. of IMC*, Nov. 2012.
[8] A. McGregor, "Graph stream algorithms: A survey," *SIGMOD Rec.*, vol. 43, no. 1, pp. 9–20, May 2014.
[9] M. R. Rahman and C.-N. Chuah, "Can sampling preserve application adoption process over OSN graphs?" in *NetSci*, Jun. 2014.
[10] N. Duffield, C. Lund, and M. Thorup, "Estimating flow distributions from sampled flow statistics," *IEEE/ACM Trans. Net.*, vol. 13, no. 5, pp. 933–946, 2005.
[11] B. H. Bloom, "Space-time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, pp. 422–426, 1970.
[12] G. J. McLachlan and T. Krishnan, *The EM Algorithm and Extensions*, 2nd ed. Wiley Interscience, 2008.
[13] T. Inc., "The streaming APIs," https://goo.gl/GimVDR.
[14] J. S. Vitter, "Random sampling with a reservoir," *ACM Trans. Math. Software*, vol. 11, no. 1, pp. 37–57, 1985.
[15] G. J. McLachlan and T. Krishnan, *The EM Algorithm and Extensions*, ser. Wiley Series in Probability and Statistics. John Wiley and Sons, 1997.
[16] H. A. Sturges, "The choice of class interval," *J. Amer. Statist. Assoc.*, pp. 65–66, Mar. 1926.
[17] J. Leskovec, J. Kleinberg, and C. Faloutsos, "Graph evolution: Densification and shrinking diameters," *ACM Trans. Knowl. Discov. Data*, vol. 1, no. 1, pp. 1–41, Mar. 2007.
[18] J. Leskovec, J. Kleinberg, and C. Faloutsos, "Graphs over time: Densification laws, shrinking diameters and possible explanations," in *Proc. of KDD*, 2005.