Influence at Scale: Distributed Computation of Complex Contagion in Networks

Brendan Lucier
Microsoft Research
brlucier@microsoft.com

Joel Oren
University of Toronto
oren@cs.toronto.edu

Yaron Singer
Harvard University
yaron@seas.harvard.edu

ABSTRACT

We consider the task of evaluating the spread of influence in large networks in the well-studied independent cascade model. We describe a novel sampling approach that can be used to design scalable algorithms with provable performance guarantees. These algorithms can be implemented in distributed computation frameworks such as MapReduce. We complement these results with a lower bound on the query complexity of influence estimation in this model. We validate the performance of these algorithms through experiments that demonstrate the efficacy of our methods and related heuristics.

1. INTRODUCTION

For the past several decades there has been a growing interest in understanding the way information is adopted between individuals in a society [20, 21, 34, 30, 4]. In recent years, the surge of massive records of human interactions has brought a new, system-wide perspective on such processes. As interactions between individuals link across multiple steps in a network, patterns of cascades emerge in the data. These digital traces allow predicting future cascades in the network [18, 19, 11], recovering cascades from incomplete measurements [17, 8], and even engineering future cascades by selecting important individuals to promote a new product or social movement [14, 24].

Naturally, the availability of data at massive scale quickly becomes a double-edged sword. While more data can potentially make it easier to detect emerging patterns and improve predictions, processing large data sets is challenging. In cascades particularly, quantifying the impact of a chain reaction of individual interactions in a large network quickly becomes a difficult computational task.

Estimating the spread of a cascade. The primitive module of cascade prediction and optimization methods is the estimate of its expected spread in the network: given a mathematical model of influence that describes the way in which information is transmitted in the network, the module takes a set of infected nodes at time step t and returns the expected number of nodes that will be infected in some time step $t' > t$. Since the mathematical models that we tend to use are stochastic, these methods are estimated through sampling. While naive sampling is usually reasonable for small networks, already for networks with several thousands of nodes estimating influence becomes a daunting task. This naturally raises the following question.

Can cascade estimation be made scalable?

In this paper we address questions revolving around cascade estimation in large networks. We focus on the well-studied *independent cascade* model of influence as formulated in [24] and consider the question of designing scalable algorithms for estimating cascades in this model.

There are largely two aspects behind the role of scalable algorithms for estimating influence. The first, and perhaps more obvious, is the paradigmatic one: given the prominent role of cascade estimation in data mining, we are naturally interested in tools that can estimate cascades efficiently. The other aspect revolves around our goal to understand whether the *models* we use for describing cascades are appropriate. If the purpose of a mathematical model is to accurately predict cascades, we would like it to be computationally feasible to estimate cascades on large data sets.

The standard approach to dealing with large-scale data is through distributed computation frameworks such as MapReduce [12] or Hadoop [35]. The idea is to partition the data, perform local computations on the partitions, and aggregate the results. By their definition, cascades do not easily succumb to distributed computing paradigms. Since the local influence between any two nodes is affected by interactions with all other nodes in the network, distributing this computation becomes a serious challenge.

Main Results. We describe an algorithm that, for a prescribed set of nodes, provides with high probability an arbitrarily accurate estimate of the influence function under the independent cascade model. We show that this algorithm can be implemented in a fashion that is compatible with parallel frameworks such as MapReduce. At a high level, the idea is to estimate influence via a sampling approach that allows both parallelization and trading off between simulation cost and informativeness. We employ a probabilistic analysis to illustrate how an algorithm can choose parameters to navigate this tradeoff appropriately.

We complement this result by studying the information demands of computing the influence of a single node in a

KDD'15, August 10-13, 2015, Sydney, NSW, Australia.
© 2015 ACM. ISBN 978-1-4503-3664-2/15/08 ...$15.00.
DOI: http://dx.doi.org/10.1145/2783258.2783334.

social network. At its core, estimating a cascade involves simulations that iteratively make calls to an adjacency matrix encoding the edges of the network. Since a single machine cannot store a large network in memory, large graphs are stored in distributed hash tables that can be queried by computational nodes. The question becomes: how many queries does one need to perform in order to estimate influence well? We derive lower bounds on the number of queries to the networks' adjacency matrix required to estimate the influence of a prescribed node in the network when the topology of the network is a priori unknown.

1.1 Related work

The independent cascade model has been formulated by Kempe, Kleinberg, and Tardos in their seminal paper [24]. There is a rich body of literature on predicting cascades [18, 19, 11, 7, 15], and work on improving the sampling methods in influence maximization [6, 2], but we are not aware of work that addresses the problem of substantial reduction to the sampling required to estimate influence or how to distribute this computation.

In recent work, Cohen et al. study a related problem of *sketching* influence functions [10]. In their work, Cohen et al. perform preprocessing and construct oracles that approximate influence functions well by sampling realizations of the influence graphs. They show empirically that their methods work well in practice. However, in order to guarantee a constant-factor approximation in theory, the number of samples needs to be quadratic in the size of the graph. In contrast, we develop a scalable algorithm that yields both practical performance and theoretical guarantees.

There is a growing body of literature on parallel algorithms in the MapReduce computational paradigm. Karloff et al. [23] introduced the first theoretical model of computation that captures the main characteristics of the now-popular MapReduce framework [12]. They provide a number of MapReduce algorithms for graph-theoretic problems. Our estimation problem is tied to the reachability problem in directed graphs, about which little is known in distributed settings [37]. The literature on MapReduce algorithm design also includes works on combinatorial optimization problems such as (e.g., [9, 26, 25, 31]). A recurring theme in these papers, which also appears in our methods, is the need to estimate an objective value in a poly-log number of rounds.

The complexity of computing influence exactly is known to be #P hard [38]. Here, we show the communication complexity of approximating influence in a distributed setting.

2. PRELIMINARIES

We use the standard notation of a graph $G = (V, E)$, where $|V| = n$. At a high level, an influence spread process on a graph $G = (V, E)$ is defined to be a stochastic, discrete process, that admits a sequence of subsets of nodes $S_1, S_2 \ldots, S_n$. For $t \geq 1$, the subset of nodes $S_t \subseteq V$ is referred to as the set of infected nodes corresponding to step t. The point of origin of the process is given by a subset $S_0 \subseteq V$ of *seed* nodes. In this context, the *influence of a node* $v \in V$ for step t is defined to be the expected number of infected nodes at step t; i.e., $\mathbb{E}[|S_t|]$, if we were to fix $S_0 = \{v\}$. In this paper, we will focus on the widely studied independent cascade model [24], and assume $t = n$, though all our results hold for any value of $t > 0$.

The Independent Cascade model. We will use the definition of the Independent Cascade as formalized by Kempe, Kleinberg, and Tardos [24]. In this model, we are given a directed edge-weighted graph $G = (V, E, \mathbf{w})$, for $\mathbf{w} \in [0, 1]^{|E|}$. Given the set of initially infected nodes $S = S_0$, at each step $t > 1$, every node $u \in S_{t-1}$ that was infected at step $t - 1$, attempts to infect each of its out-neighbors v, and succeeds with probability w_{uv}. Once infected, a node *remains* infected throughout the process. Note that due to this monotonicity property of the process, it can take at most $n - 1$ steps. We will be interested in estimating the value of the function $h_G(S) = \mathbb{E}[|S_{n-1}||S_0 = S]$, defined to be the expected size of the set $S_{(n-1)}$, given the initial set S, where the expectation is taken over all of the realization of the process. Slightly abusing notation, for a specified node $u \in V$, we let $h_G(u) = h_G(\{u\})$.

The query model. In both our lower bound and upper bound results, we make use of the link-server model of communication, which has been previously used in the context of computing the PageRank of webpages (e.g., [1, 3]). Since the social networks may have billions of nodes, a graph is too large to store on individual machines. In this model, every query on a node v returns the set of incoming and outgoing neighborhoods of node v and the associated the edge probabilities. This model provides a convenient abstraction that enables us to study the information requirement of computing the influence of a given node, without having any dependencies on the way in which the information is of the network is stored. Our lower bounds will be on the number of queries to a centralized disk that stores the entire network. In the case of distributed algorithms, this model implies that all machines can communicate with this centralized server, as common in distributed computing (e.g. [36, 13, 33, 5]).

3. SAMPLING METHODS

We now turn to the task of estimating the influence of a given set of seed nodes, in the independent cascades model. A natural approach to this problem is to repeatedly sample the influence process. But we note that a straightforward Monte Carlo simulation, in which the influence process is executed repeatedly to completion until the estimation errors become small, can be prohibitively expensive. Our example below illustrates that a linear number of simulations are required to guarantee a constant approximation in the worst case. Since each simulation can involve traversing the entire network, the full simulation process can scale quadratically with network size. Moreover, this is a bound only on the number of nodes traversed, and says nothing of the additional overhead of implementation.

A simple example illustrates the issue. Consider a network consisting of a clique on $n-1$ nodes, plus one additional vertex u that is connected to a single vertex v of the clique. The weight of edge (u, v) is $\frac{c_0}{n-1}$ for some constant $c_0 > 0$, and the weight of every other edge is 1. The influence of seed set $S_0 = \{u\}$ is precisely $1 + c_0$, since u is always influenced and the remainder of the network is influenced precisely if influence spreads from node u to node v. However, to estimate this influence by sampling, one must take enough samples to observe the event that u influences v; this requires $\theta(n)$ samples. Motivated by this, one might think to take $\theta(n)$ samples when estimating the influence of any given seed set.

However, if we were estimating the influence of a node from the clique, say $S_0 = \{v\}$, then each sample would take $\theta(n)$ time and space, since each node of the clique is guaranteed to be influenced. Taking a linear number of samples would thus lead to quadratic time and space, which is prohibitively expensive.

The primary issue raised by this example is that simulation costs are heterogeneous. Estimating the influence of node u requires many samples, but these samples can (on average) be executed cheaply – they almost always finish in constant time. On the other hand, simulating the influence of node v is very costly, but very few samples are required to conclude that v has high influence. To implement a scalable estimation procedure, we will design a sampling protocol that accounts for this cost heterogeneity. The high-level approach is to first use a small number of (potentially expensive) samples, and determine whether the outcomes of these simulations suffice to generate a good influence estimate. If so then the algorithm terminates; if not, it will increase the number of samples and try again. We will show that the intuition from the example above holds in general: many samples are required for estimation precisely when samples have low average cost, so the iterative approach has low cost in aggregate. The formal proof requires some care in the implementation details. For instance, it will be useful to run simulations with a cap on the number of nodes traversed, to more cheaply estimate the probability that the influence value is above a given threshold.

The remainder of this section describes the modified sampling process, and establishes its improved query performance via probabilistic theory arguments. Then, in Section 4, we will show how to actually implement the sampling procedure in a highly parallelizable fashion, so that (a) the cost of simulating the influence function is approximately proportional to the number of nodes influenced, and (b) no significant computational or memory overhead is imposed, beyond the query complexity bounds we establish below.

3.1 Frugal estimation of influence

Given seed nodes S in a graph G, our goal is to estimate $h_G(S)$, the expected influence of S. If we write I for the random variable denoting the set of nodes influenced, then we can write $h_G(S) = \mathbb{E}[I] = \sum_{t=1}^{n} \mathbb{P}[I > t]$. We can then approximate this quantity using a standard Riemann sum:

$$\pi = \sum_{t \in \{1,(1+\epsilon),(1+\epsilon)^2,\ldots,n\}} \frac{\epsilon}{1+\epsilon} \cdot t \cdot \mathbb{P}[I > t]. \tag{1}$$

We then observe that π is a close approximation to $h_G(S)$: $h_G(S) \le \pi \le (1+\epsilon)h_G(S)$. Thus, from this point onwards, we will focus on approximating π rather than $h_G(S)$. The reason we choose to approximate (1) is that, in a simulation of influence, the event $[I > t]$ can be determined before the simulation completes; for example, one can imagine terminating the simulation after t nodes have been influenced. Thus, by separating π into its constituent terms, we can hope to estimate them more cheaply by appropriately capping the length of simulations.

Frugal sampling. Write $\pi_t = \mathbb{P}[I > t]$. To approximate π, it will suffice to approximate each π_t. We will approximate π_t by sampling graphs from $\mathcal{G}$. How many samples are needed? Intuitively, when t is small and π is large, it is likely that the influence will often be larger than t, and hence few samples are required to estimate π_t. On the other hand, if π is small but t is large, many samples may be required; however, because π is small, we expect most samples to generate few queries. We formalize this intuition in the following lemma. For a given $L > 0$ indicating a number of samples from $\mathcal{G}$, we will use $\pi_t(L)$ to denote the random variable indicating the fraction of sampled graphs in which the set reaches at least t nodes. The lemma shows that if L is large enough, then $\pi_t(L)$ is a good estimate of π_t.

LEMMA 1. *For any $t > 0$, $\tau \ge \pi$, and $L \ge \frac{8t\log^3 n}{\tau\epsilon^2}$,*

$$\mathbb{P}\Big[|\pi_t(L) - \pi_t| > \frac{\epsilon \cdot \tau}{t \cdot \log_{1+\epsilon} n}\Big] \le \frac{1}{n^2}. \tag{2}$$

PROOF. Define λ so that $\frac{\epsilon\tau}{t\log_{1+\epsilon} n} = \lambda\pi_t$. Note that $\pi_t \le \pi/t \le \tau/t$, from the definition of π.

Suppose that $\pi_t < \frac{\epsilon\tau}{t\log^2 n}$, so that $\lambda > 1$. By Chernoff bounds, the event in (2) occurs with probability at most

$$\exp\left(-\pi_t \cdot L \cdot \frac{\epsilon\tau}{t\pi_t \log n} \cdot \frac{1}{2}\right) < \frac{1}{n^2}.$$

Next suppose π_t lies between $\frac{\epsilon\tau}{t\log^2(n)}$ and $\frac{\tau}{t}$, so $\lambda \le 1$. By the multiplicative Chernoff bound, the probability of the event in (2) is at most

$$\exp\left(-\pi_t \cdot L \cdot \left(\frac{\epsilon\tau}{t\pi_t\log n}\right)^2 \frac{1}{4}\right) \le \exp\left(-\frac{2\tau\log n}{\pi_t t}\right) \le \frac{1}{n^2}$$

as required. □

Since Lemma 1 holds for each $t > 0$, a union bound implies that $\left|\sum_t \frac{\epsilon}{1+\epsilon} \cdot t \cdot \pi_t(L) - \pi\right| < \log_{1+\epsilon}(n) \cdot \frac{\epsilon\tau}{\log_{1+\epsilon} n} = \epsilon\tau$ with probability at least $1 - 1/n$. This yields the following corollary, which bounds the error of an empirical estimate of π that uses L simulations to estimate π_t for each t.

COROLLARY 2. *If $\tau \ge \pi$ and $L \ge \frac{8t\log^3 n}{\tau\epsilon^2}$ for all $t > 0$ then, w.h.p., $\left|\sum_t \frac{\epsilon}{1+\epsilon} \cdot t \cdot \pi_t(L) - \pi\right| \le \epsilon\tau$.*

3.2 The algorithm

We can now describe our simulation algorithm, which we call Influence Estimator (InfEst). The major component of the algorithm is a simple verifier: given a guess τ of the true influence of a set S, the verifier estimates the influence using $\sum_t \frac{\epsilon}{1+\epsilon} \cdot t \cdot \pi_t(L_t)$ with an appropriately chosen L_t, and accepts if this value is close to τ. An immediate consequence of Corollary 2 is that when the verifier receives π as a guess it accepts, with high probability. The InfEst algorithm then simply iterates over guesses of τ until the verifier accepts. We formally describe the algorithm below.

ALGORITHM VerifyGuess: The algorithm for verifying whether a guess of influence value is correct

```
Input: An edge weighted graph $G = (V, E, \mathbf{w})$, initial seed set
       $S \subseteq V$ and guess $\tau$
1 for $t \in \{\tau, (1+\epsilon)\tau, (1+\epsilon)^2\tau, \ldots, n\}$ do
2     Sample $L = \frac{8t\log^3 n}{\tau\epsilon^2}$ instances of the graph;
3 If $\sum_t \frac{\epsilon}{1+\epsilon} \cdot t \cdot \pi_t(L) \ge (1-2\epsilon)\tau$ return 1.
4 return 0
```

Note that InfEst iterates over guesses τ from large to small. This is crucial: our bound in Lemma 1 requires that

ALGORITHM InfEst: The approximation algorithm for estimating the spread for the independent cascade model

```
Input: An edge weighted graph $G = (V, E, \mathbf{w})$, initial seed set
       $S \subseteq V$, precision $\epsilon$
1 for $\tau \in \{n, n/(1+\epsilon), n/(1+\epsilon)^2, \ldots, |S|\}$ do
2     If VerifyGuess($G, S, \tau$) $= 1$ return $\tau$
3 return 1    // w.h.p. we don't reach this point.
```

$\tau \geq \pi$, and hence we can establish correctness as long as τ is an over-estimate of influence. Employing this reasoning leads to the following result.

THEOREM 3. *For any $\epsilon \in (0, \frac{1}{4})$,* INFEST *provides a $(1 + 8\epsilon)$ approximation to $h_G(S)$, with high probability.*

PROOF. Consider an iteration of INFEST with $\tau > \frac{1}{1-3\epsilon}\pi$. Then Corollary 2 implies

$$\sum_t \frac{\epsilon}{1+\epsilon} \cdot t \cdot \pi_t(L) < \pi + \epsilon\tau < (1-3\epsilon)\tau + \epsilon\tau < (1-2\epsilon)\tau.$$

So w.h.p. VERIFYGUESS will return 0 and INFEST will not terminate on this iteration. Next suppose $\pi \leq \tau \leq (1+\epsilon)\pi$. In this case, Corollary 2 implies that

$$\sum_t \frac{\epsilon}{1+\epsilon} \cdot t \cdot \pi_t(L) \geq \pi - \epsilon\tau \geq \frac{1}{1+\epsilon}\tau - \epsilon\tau \geq (1-2\epsilon)\tau.$$

So INFEST will terminate w.h.p. before the first iteration in which $\tau < \pi$. We conclude that the algorithm always terminates on an iteration in which τ is within a factor of $\frac{1}{1-3\epsilon}$ of π. The fact that π is within a factor of $(1+\epsilon)$ of $h_G(S)$, plus the fact that $\epsilon < \frac{1}{4}$, completes the proof. □

We next establish an upper bound on the sum of influences over all simulations executed by INFEST. We omit the proof, which is very similar to the proof of Proposition 6 in Section 4. The main idea is that each subsequent iteration of INFEST increases the number of simulations executed, but later iterations are only performed if π is small, in which case the observed influences will likely be low.

PROPOSITION 4. *For any $\epsilon \in (0, \frac{1}{4})$, the sum of observed influences over all simulations executed by* INFEST *is at most $\frac{8(1+\epsilon)n\log^5(n)}{\epsilon^2}$, with high probability.*

4. DISTRIBUTED INFLUENCE ESTIMATION

Theorem 3 and Proposition 4 establish that INFEST can obtain an ϵ-approximation to the influence of S, using a sequence of samples of total aggregate size $O(n \cdot polylog(n) \cdot \epsilon^{-2})$. How should these samples be collected? One option is a sequential implementation: perform the samples one at a time, and for each one simulate influence by way of a Breadth-First Search (BFS) crawl of the network. The total execution time and memory requirements would then closely match the bound from Proposition 4, but this cost would be suffered sequentially. Our goal is to describe a more parallelizable implementation that obtains similar bounds on memory and total computation cycles.

Parallelizing samples. We begin by making a simple observation. INFEST involves $\log_{1+\epsilon}(n)$ calls to `VerifyGuess`. Each call then repeatedly samples an instance of the graph and determines, for each sample, whether the number of nodes reachable from S is greater than a quantity t. Each of these samples in `VerifyGuess` could be taken in parallel, with outcomes aggregated in the summation from line 4 of `VerifyGuess`. Moreover, the multiple calls to `VerifyGuess` from INFEST can themselves be parallelized; aggregation involves determining the maximal τ for which `VerifyGuess` returns 1. In this way, INFEST can be implemented as $O(n\log^5(n)\epsilon^{-2})$ parallel calls to a subroutine that explores the component reachable from set S in a random graph realization. We can treat this subroutine as a black box; indeed, in some scenarios (such as when influence is almost always low) a sequential implementation of the exploration subroutine may be satisfying. For more general applicability, however, we will also show how to implement such a sampling oracle in a more parallelized fashion.

4.1 Sampling in MapReduce

We now describe an implementation of SAMPLEORACLE, a method for sampling of L instances of the independent cascade model. The oracle must return the fraction of these instances in which the number of nodes reachable from a seed set S is at least some threshold t. We note that there is a known algorithm for determining the size of a connected component in an *undirected* network, developed by Karloff et al. [23]. However, because networks are directed in our context, we cannot apply their approach directly.

Algorithm SAMPLEORACLE is based on the natural MapReduce algorithm for breadth-first search [29]. The algorithm takes place in "epochs." During epoch $d > 0$, the following operations are performed:

1. *Infection:* Each node that was newly infected in the previous epoch attempts to infect each of its out-neighbors.
2. *Node-level Aggregation:* For each node v that was successfully infected at least once, and had not yet been infected, record the node as having been infected.
3. *Sample-level Aggregation:* Increment a counter for the number of nodes infected within a sample. If the number of reached nodes is greater than t or if no new nodes were infected, terminate the sample.

The details of SAMPLEORACLE are listed below.

From the above description, one can observe that SAMPLEORACLE directly implements the discovery of L spanning trees, corresponding to L realizations of the network, up to the size of the component reachable from S (if less than t) or to a size that is at least t (otherwise). The correctness of the algorithm is then immediate.

PROPOSITION 5. *Algorithm* SAMPLEORACLE *returns the fraction of L graph realizations in which the component reachable from S has size at least t.*

Implementation and Memory Requirements. SAMPLEORACLE maintains sets of $\langle key; value\rangle$ tuples of two types:

1. *node tuples:* Correspond to the nodes infected thus far in a given sample. An infected node $v \in V$ in sample $\ell \in [L]$, will be represented by a tuple $\langle v; (\ell, new)\rangle$, where $new = True$ if v was first infected in the previous epoch and $new = False$ otherwise.

ALGORITHM SampleOracle: A BFS MapReduce Algorithm

```
Input: Seed set $S \subseteq V$; Number of samples $L$; threshold $t$;
       oracle query access: $Q(\cdot,\cdot)$, such that $Q(u,i) = (v, w_{uv})$
       where $v$ is the $i$'th out-neigbor of $u$.
/* Initialization:                                              */
1  for $\ell = 1, \ldots, L$ do
2      Set initial set of reached nodes: $R_0(\ell) = S$
3      Label sample $\ell$ "incomplete" Label each $u \in R_0(\ell)$ as "new"
4  $d \leftarrow 1$
5  while Not all samples complete do
6      foreach $\ell \in [L]$ s.t. sample $\ell$ is labelled "incomplete" do
7          $T_d(\ell) \leftarrow \emptyset$
8          foreach $u \in R_{d-1}(\ell)$ s.t. $u$ is labelled "new" do
9              for $i \leftarrow 1$ to $\min\{|N^-(u)|, t\}$ do
10                 $v, w_{uv} \leftarrow Q(u,i)$
11                 Infect $v$ w.p. $w_{uv}$.
12                 If successful, add a "new"-labeled $v$ to $T_d(\ell)$.
13                 Add an "old"-labelled $u$ to $T_d(\ell)$.
       /* Aggregation:                                              */
14     for $\ell \leftarrow 1$ to $L$ do
15         $R_d(\ell) \leftarrow \emptyset$
16         foreach unique $v \in T_d(\ell)$ do
17             If there is an "old" labelled $v$ in $T_d(\ell)$, add an "old"
               labelled $v$ to $R_d(\ell)$, otherwise, add a "new" labelled
               $v$ to $R_d(\ell)$.
           /* Sample-level aggregation                                  */
18         If $|R_t(\ell)| \geq t$ or all nodes in $R_t(\ell)$ are labelled "old",
           declare sample $\ell$ completed.
19     $d \leftarrow d + 1$
20 return $|\{\ell \in [L] : |R_t(\ell)| \geq t\}|/L$
```

2. *sample-counter tuples:* Correspond to completed samples. Each sample-counter tuple is of the form $\langle \ell, r_\ell \rangle$, where ℓ is the sample identifier, and r_ℓ is the number of nodes reached during that sample. These tuples are generated in line 18 of the algorithm.

A key property of SampleOracle is that it does not generate too many tuples in total over all L samples generated. To obtain the desired bounds, we must analyze the number of tuples generated as a function of the size of the generated spanning tree. This will connect its memory requirements to the lower bound from Proposition 4. A complication is that we cannot assume that the trees generated have size at most t, since a given sample can grow significantly larger than t during the epoch on which its size first exceeds t. We must therefore bound the number of tuples needed to implement a sample that spans an arbitrary fraction of the network. Recall the definitions of π, π_t, and $\pi_t(L)$ from Section 3.

Proposition 6. *Suppose* SampleOracle *is invoked with $L \leq \frac{8t\log^3 n}{\epsilon^2 \tau}$ for some $\tau \geq \pi$. Then with probability at least $1-1/n$, the total number of tuples generated by the algorithm is at most $8\left(\frac{1+\epsilon}{\epsilon^2}\right) m^{3/2}\log^4 m$.*

Proof. We will actually prove the result assuming $L = \frac{8n\log^3 n}{\epsilon^2\pi}$, which can only be greater than the bound imposed in the statement of the proposition. Since the number of tuples generated by the algorithm stochastically increases with L, this will imply the proposition.

Lemma 1 implies that for any $k \in [1, n]$,

$$\Pr\left[|\pi_k(L) - \pi_k| > \frac{\epsilon\pi}{k\log_{1+\epsilon} n}\right] < \frac{1}{n^2}.$$

In other words, with probability at least $1 - \frac{1}{n^2}$, we have

$$\pi_k(L)\cdot L \leq \pi_k \cdot L + \frac{\epsilon\pi}{k}\cdot L \leq 8\left(\frac{n}{k}\right)(\log^3 n)\left(\frac{1+\epsilon}{\epsilon^2}\right)$$

where in the second inequality we used that $\pi \geq k\cdot\pi_k$, which follows from the definition of π. Taking a union bound over choices of k, we have that

$$\pi_k(L)\cdot L \leq 8\left(\frac{n}{k}\right)(\log^3 n)\left(\frac{1+\epsilon}{\epsilon^2}\right) \qquad (3)$$

for all $k \in \{1, \ldots, n\}$, with probability at least $1 - 1/n$.

Write $Y(\ell)$ for the number of tuples generated over the course of sample ℓ. We must have $Y(\ell) \leq m$, since each edge of the graph can have at most one associated tuple. Suppose that $k = |R_t(\ell)|$ nodes are reached by the spanning tree generated in sample ℓ. Then we must have $Y(\ell) \leq k^2$, since each tuple uniquely corresponds to an attempted infection of some node $v \in R_t(\ell)$ by some other node $u \in R_t(\ell)$. In summary, we have $Y(\ell) \leq \min\{m, k^2\}$.

Counting up over all L samples, and using $\pi_k(L)$ as an upper bound on the fraction of spanning trees of size exactly k, the total number of tuples generated is at most

$$\sum_{k\in\{1,(1+\epsilon),(1+\epsilon)^2,\ldots,n\}} \pi_k(L)\cdot L\cdot\min\{m,k^2\}$$
$$\leq 8n\log^3 n\left(\frac{1+\epsilon}{\epsilon^2}\right)\sum_k \min\{m/k, k\}$$
$$\leq 8m^{3/2}\log^4 m\left(\frac{1+\epsilon}{\epsilon^2}\right)$$

as required. □

We note that our breadth-first traversal method gives an exact estimate of the number of nodes infected in each sample. There are known methods for decreasing the number of tuples generated (e.g., [22]), at the expense of adding an additional multiplicative factor to the approximation ratio. If the tradeoff is deemed desireable in a given setting, our methodology is compatible with such implementations.

4.2 The MRC Framework

The algorithm SampleOracle fits within a theoretical class of distributed algorithms known as MRC, which was developed to formalize MapReduce algorithms [23]. An algorithm in the MRC class is composed of sequence of rounds. In each round, a procedure called a *mapper* traverses the set of tuples, one by one, and produces a new multiset of $\langle key, value\rangle$ tuples. Then, a module called a *shuffler* dispatches each set of tuples with the same key k to a separate *algorithmic* component called a *reducer*, which can process these keys in a sequential manner. Each reducer runs in parallel on its set of tuples, and then outputs a new set of keys that would be use in the next map-reduce round. The output of the algorithm is required to be correct with probability at least 2/3. There are three further requirements: first, the number of map-reduce rounds in polylogarithmic in the size of the input m. Second, although the reducers may run for polynomial time, their alotted space should be sublinear in m, and third, at most a sublinear number of reducers are allowed to execute in parallel. The latter two restrictions imply that the total space taken by the tuples output by the reducers is $O(m^{2-2\epsilon})$ for some $\epsilon > 0$.

SampleOracle satisfies the total space requirement of the MRC class, by Proposition 6. Moreover, one can imploy random hashing methods to map (node,sample) pairs to reducers in such a way that a sublinear number of reducers is required, and each receives a sublinear number of tuples. This hashing method was employed in [23]; we elaborate on the details in the full version of the paper. Finally, if every realization of $\mathcal{G}$ has polylogarithmic diameter, then it follows that SampleOracle will complete in a polylogarithmic number of rounds. While this property is not guaranteed to hold for arbitrary $\mathcal{G}$, it has been observed empirically that long chains of influence are rare [16], and hence it is natural to consider cases in which the tree of influence spread has low diameter. We obtain the following result.

Proposition 7. *If, in every realization of $\mathcal{G}$, the component of nodes reachable from S has polylogarithmic diameter, then algorithm* SampleOracle *invoked with with $\frac{8n \log^3 n}{\epsilon^2 \pi}$ falls within the class MRC.*

Our earlier discussion of the parallel implementation of InfEst implies a reduction: if SampleOracle can be implemented by an MRC algorithm, then InfEst can also be implemented as an MRC algorithm. This follows directly from the fact that InfEst is a polylogarithmic number of iterations over the sampling procedure. We note that this reduction does not immediately imply an MRC implementation of InfEst for the problem of sampling influence in general, as the polylogarithmic diameter requirement of Proposition 7 may not hold for all realizations of the independent cascades process on a given network. We leave the development of new MRC sampling methods for influence processes on general networks as a direction for future research.

5. THE COMPLEXITY OF CONTAGION

We devote this section to the complexity of contagion. We assume the network is stored on some hypothetical disk, and lower bound the number of queries to the disk required to estimate (within some specified precision) the influence of a given set. To establish these lower bounds we judiciously construct graphs that are a priori unknown to the algorithm designer, so that, for specific a node u, approximating its influence (the expected spread in G as a result of selecting it to be the single initial seed) beyond a certain approximation ratio requires a relatively high number of queries to the link server. We obtain these bounds using the following well-known result due to Nisan [32]:

Lemma 8. *Let $\varphi = \bigvee_{i=1}^m z_i$ be an OR function, given by the disjunction of m Boolean variables $z_1, \ldots, z_m$. Then (randomly) determining the value of φ with a confidence level of $1-\delta$ requires at least $m(1-2\delta)$ queries on the values.*

Our reductions from the OR problem will have the following overall structure. Given a formula $\varphi = \bigvee_{i=1}^m z_i$, and an assignment $\mathbf{a} \in \{0,1\}^m$, we will construct a graph $G_a = (V, E)$ with a designated node $u \in V$, where the structure of the graph will crucially depend on the particular assignment $\mathbf{a}$. As we will argue, for any two assignments to the formula $\mathbf{a}, \mathbf{b} \in \{0,1\}^m$, such that $\mathbf{a}$ satisfies the OR formula, whereas $\mathbf{b}$ does not (i.e., $\mathbf{b}$ is the all zeros vector), the resulting expected spread values in the graphs G_a and G_b would exhibit a large gap. We note that a similar approach was taken by Bar-Yossef and Mashiach [1] for the task of locally computing the PageRank of a webpage.

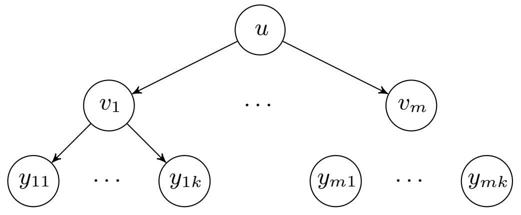


Figure 1: The construction for the lower bound.

Theorem 9. *Let $\epsilon \in [0, 1/3), \delta \in [0, 1/2)$, and consider the independent cascade influence model in directed graphs. Then for large enough n, there exists a graph $G = (V, E, \mathbf{w})$, and a distinguished node $u \in V$, for which estimating the influence function $h_G(\{u\})$ to a factor in the range $[(1-\epsilon), (1+\epsilon)]$ with probability (confidence) at least $1-\delta$ requires $\Omega((1-2\delta)\sqrt{n})$ queries to the link server.*

Proof. We create the rooted tree depicted in Figure 1. Formally, the graph G_φ contains the set of nodes: $V = \{u\} \cup \{v_i : i \in [m]\} \cup \{y_{ij} : i, j \in [m]\}$. Each node v_i, for $i = 1, \ldots, m$, corresponds to variable z_i. For each $i = 1, \ldots, m$ we draw an edge (u, v_i). Also, for each i such that $z_i = 1$, we add m edges: $(v_i, y_{i1}), \ldots, (v_i, y_{im})$. Finally, we set all of the edge weights to 1. The following observation then follows from the construction and the definition of the spread process, which gives the lower bound almost immediately:

Observation 10. *Let $\mathbf{a}$ be an assignment to the OR formula φ. Then, $h_{G_\varphi}(u) = 1 + m \cdot (|\mathbf{z}| + 1)$.*

We can now prove the theorem. Suppose by way of contradiction that the theorem is false, and there exists an algorithm A that, for any graph $G = (V, E)$ with n nodes, provides an estimate $\tilde{h}_G(v)$ with relative error at most ϵ, with probability at least $1-\delta$, that makes $o(\sqrt{n}) = o(m)$ queries to the link server. We now construct an algorithm B that computes the value of φ with $o(m)$ queries.

Given an assignment $\mathbf{a}$ to the formula $\varphi = \bigvee_{i=1}^m z_i$, construct the corresponding graph G_a as described above.[1] Algorithm B will then simulate an execution of algorithm A, by acting as a link server, as follows. Upon the query of node u by algorithm A, the set of nodes $\{v_1, \ldots, v_m\}$ will be returned to it. Upon the query of a node v_i, the simulation will query bit z_i, and will return the set $\{y_{i1}, \ldots, y_{im}\} \cup \{u\}$ if $z_i = 1$, and $\{u\}$ otherwise. Finally, if A queries the simulated link server on a node y_{ij}, the singleton $\{v_i\}$ will be returned. Letting $\tilde{h}_{G_a}(u)$ denote the resulting estimated expected influence of node u after t steps, algorithm B returns 0 if $\tilde{h}_{G_a}(u) \le (1-\epsilon)2m + 2$, and 1 otherwise.

Now, if $\mathbf{a}$ does not satisfy φ (the OR formula is not satisfied), by Observation 10, $h_{G_a}(u) = m+1$. And so with probability at least $1-\delta$, $\tilde{h}_{G_a}(u) \le (1+\epsilon)(m+1) < 4m/3 + 2$, in which case B will return 0, as required. Similarly, if $\mathbf{a}$ satisfies φ, with probability at least $1-\delta$, $\tilde{h}_{G_a}(u) \ge (1-\epsilon)h_{G_a}(u) > \frac{2}{3}(2m - o(1)) = \frac{4m}{3} - o(1)$, and so the algorithm would return 1 in that case, as required. Having shown the correct estimation of the value of φ with probability at least $1-\delta$, while using $o(\sqrt{n}) = o(m)$, results in a contradiction to Lemma 8. $\square$

[1]Note that our simulation does not need to construct the entire graph in advance; rather, it suffices to generate the adjacency lists on the fly, upon each query.

Network	n	m	Type	Avg. degree
wiki-Vote	7,115	103,689	Directed	14.6
Epinions	75,879	508,837	Directed	6.7
Slashdot	82,168	948,464	Directed	11.5
Youtube	1,134,890	2,987,624	Undirected	2.6

6. EXPERIMENTS

We present empirical validation of our methods and results from experiments on real and synthetic large network data sets. We experimented with a distributed implementation of InfEst as discussed in Section 3, as well as several heuristics based on this approach. We will be comparing against the benchmark of sampling from the influence process $\Theta(n \log n)$ times; we will refer to this benchmark algorithm as MonteCarlo. Our main conclusion from running the experiments is that InfEst can handle very large data sets efficiently while maintaining its theoretical guarantees.

6.1 Experimental setup

We tested our algorithms on real social networks, as well as on synthetic ones that are based on well-studied generative models. For all of datasets, we considered three methods for setting the edge weights[2]:

- Method **E1**: Each edge is assigned a weight drawn uniformly from $[0, 1]$;
- Method **E2**: Each edge is assigned a weight drawn uniformly from $\{0.1, 0.01\}$ (see [24]);
- Method **E3**: The edge probability of an edge (u, v) was set to the inverse of v's in-degree (as in [24]);

In our tests the seed sets were chosen uniformly from the vertex set V. We ran experiments on the following networks.

Real networks. In our experiments, we have made use of several well-studied real online social networks of varying size: wiki-Vote, Epinions, Slashdot, and the YouTube network. All these networks are obtained from the SNAP database [28]. We summarize statistics in Table 6.1.

Synthetic networks. To test the effects of network size and topology on the algorithm, we tested our methods on networks that were constructed based on standard generative models for social networks. We used the following generative models:

- *Small-world graphs:* We generated small-world networks [39] using a ring lattice of degree 200, then rewiring each edge with probability 0.3.
- *The Barabási-Albert model:* We constructed preferential attachment networks with out-degree 10.
- *Kronecker Graphs:* This generative model was proposed by Leskovec et al. [27]. We generated graphs by starting with 4 vertices and repeatedly applying the Kronecker product.
- *Configuration model* We employed the configuration model [40] using a power law degree sequence, matching those of our Barabási-Albert networks.

[2]For the graphs we experimented with that are undirected we set the edge weights separately for each direction.

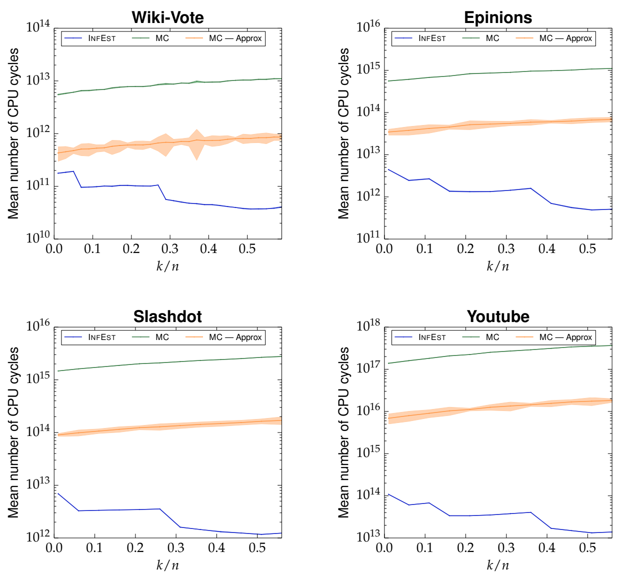


Figure 2: Running times of Algorithm InfEst and MonteCarlo on various datasets (y-axes are in logarithmic scale). The results depicted correspond to method **E1**; similar results were obtained for **E2**, and **E3**.

Computational setup. We ran our experiments on a Linux server equipped with two Intel E5-2697v2 CPUs, each with 24 cores, 30 MB of cache, and 128GB of RAM. Running time is measured in the number of CPU cycles needed, and not actual time units. This allows for flexibility in running experiments on multiple machines with different configurations. Experiments were implemented in Python 2.7.

6.2 Running time

Performance on real-world datasets. The goal of our first experiment was to compare between the running time of InfEst and MonteCarlo. In order to test the efficacy of InfEst in terms of running times, we carried out the following experiment. For a given network, and fixed value of k, we measured the number of CPU cycles needed to complete the execution of InfEst on a uniformly random sampled seed set S of size k. For the purpose of the experiments, we set the precision parameter ϵ of InfEst to be $\epsilon = 1/2$. Our goal was to compare the number of CPU cycles required by InfEst and MonteCarlo. On large graphs however, it is infeasible to take the $n \log n$ samples required by MonteCarlo. We therefore interpolated the necessary running time of the MonteCarlo algorithm by measuring the number of CPU cycles needed to sample $s = 1{,}000$ instances of the independent cascade processes, and scaling it by $n \log n/s$. We also considered a variant of MonteCarlo that takes n samples, for comparison purposes. We took varying values of k/n in the range $[0.01, 0.6]$, and five seed sets for each k. The results of this set of experiments are depicted in Figure 2.

The required running time for MonteCarlo is dramatically larger than InfEst: in the best case MonteCarlo requires 10 times the CPU cycles of InfEst (on Wiki-vote with the smallest value of k) and in the worst example (YouTube on large k) it requires as many as a $10,000$ times. Notice that whereas the running time of MonteCarlo monotonically increases with the size of the seed set, the running

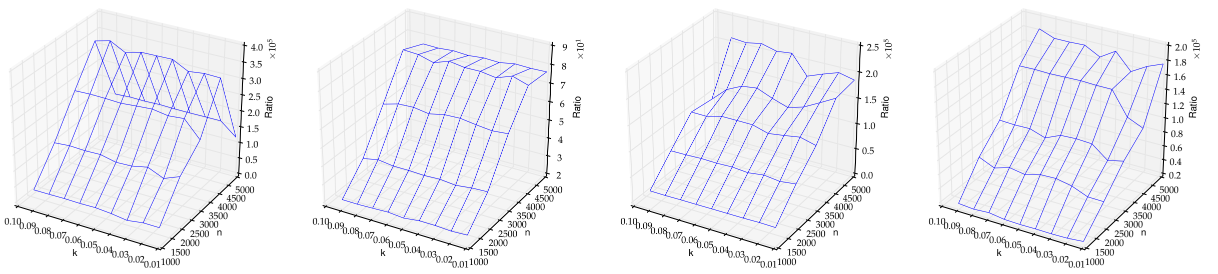

(a) Barabási-Albert (b) Kronecker (c) SmallWorld (d) Configuration Model

Figure 3: Ratios of the Running time of MonteCarlo to that of InfEst in synthetic datasets.

time of InfEst almost always decreases monotonically. This is due to the fact that for higher influence values, InfEst stops at earlier iterations of its outer loop.

Performance on synthetic datasets. To further reason about the running time of Algorithm InfEst, we studied its performance on random graphs when varying n *and* k. For each of the graph models, we created a single graph of size n, for $n \in \{1k, \ldots, 5k\}$. For each graph we sampled five initial seed sets of size k, where $k/n \in \{0.01, \ldots, 0.1\}$, and ran both InfEst and MonteCarlo algorithm (again, with interpolation). The results are depicted in Figure 3 (as before, we plot the results for **E1** and omit the very similar results obtained with the other methods).

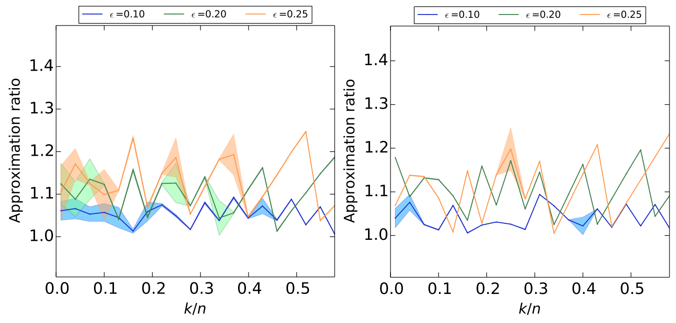
Almost all graphs display a monotonic increase in the ratio of the running times when increasing n. The only exception is the Barabási-Albert model in which there is a drop in the ratio when going from $n = 4k$ to $n = 5k$. In this model we checked larger values of n from $10k$ until $n = 25k$ and saw that there was a monotonic increase with n and k. To explain the general growing trend in all models, we have further investigated the mean influence function for various values of k, under these graph models, and have found that very high values are reached for even low values of k. This causes the running time of MonteCarlo to increases steadily as n grows. In contrast, the running time of InfEst, remains stable – as for the real datasets. Second, running time ratios are largely constant across varying values of k/n (fixing n); this can also be explained by the above observation about the value of the influence function.

6.3 Approximation

In the next set of experiments we investigated how the approximation ratio of InfEst is affected by the size of the seed set k. We ran InfEst on our small and medium datasets, with five samples of initial seed sets of varying sizes, and $\epsilon \in \{0.1, 0.2, 0.25\}$. We calculated the approximation ratio with respect to the *true value*, as computed by MonteCarlo.[3] Figure 4 depicts the results on method **E2**.

As can be seen in both of the plots in Figure 4, the approximation ratios fluctuate within the range $[1, 1.2]$, which suggests that in practice InfEst provides an estimate that is more accurate than what is predicted by our theoretical guarantee (Theorem 3). Moreover, the curves show that the approximation tends to monotonically decrease with ϵ.

[3]Note that due to its lack of scalability, we could not run the MonteCarlo algorithm on very large datasets (e.g., 100k nodes and above).



(a) Wiki-Vote (b) Epinions

Figure 4: Approximation ratios obtained by Alg. InfEst

6.4 Heuristics

Until this point we only discussed algorithms with provable guarantees. In practice one may be interested in heuristics that do not have guarantees but do well in practice. The biggest problem with estimating the performance of heuristics for computationally intensive problems is that it is often impossible to analyze the performance guarantee of the heuristic, since the optimal solution is infeasible. The provable guarantees of InfEst however, enable us to benchmark heuristics: since one cannot run the optimal number of samples required to estimate influence, an alternative would be to run InfEst and analyze the approximation of the heuristic against that. We performed several heuristics.

Convergence of influence. As a first step we examined the convergence of influence on a relatively small data set. We sampled a single random node, and estimated the value of its influence, by taking varying numbers of samples of the spread process. We ran this test on the Wiki-Vote dataset, with methods **E1**, **E2**, and **E3**. Results are in Figure 5.

Heuristic implementations. We explored heuristic implementations of InfEst. We tested three heuristics:

1. InfEst$'(y)$: A variant of InfEst in which we set $L = y$ in each iteration of VerifyGuess.

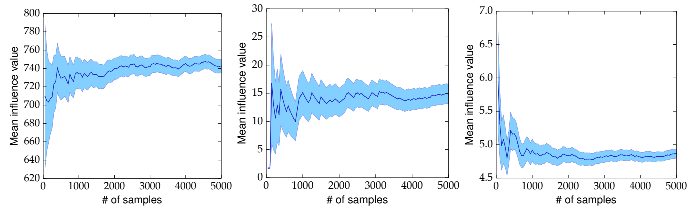

(a) E1 (b) E2 (c) E3

Figure 5: Estimates on the wiki-Votes of the function $h_G(v)$ for a random $v \in V$ as a function of the number of samples of the independent cascade process (y-axes given in logarithmic scale). Note that different nodes were selected for each test. The shaded region in the plot depicts the standard error.

2. InfEst$''(x, y)$: Set $L = y$ in VerifyGuess, and also start the outer loop of InfEst with τ set to the mean of x samples of the influence process (instead of n).

3. MonteCarlo$'(z)$: Monte Carlo with z samples.

In the experiments we set the scaling factor to 0.1. For the purpose of the experiment, we tested InfEst$'$ and InfEst$''$ with $x = y = 10$ and ran MonteCarlo$'$, with $z = 20$.

We used small and medium graphs to estimate the true value of the influence function with high precision using MonteCarlo, so that we can measure the actual estimation error of the heuristics. We ran the test on the Wiki-Vote dataset and Epinions data sets. For varying values of k, we took five random seed sets of size k, and ran all of the above algorithms for each them. We then calculated the approximation ratio and running time (in CPU cycles) of each heuristic with respect to the outcome of MonteCarlo.

Figure 6 depicts our results for the Wiki-Vote and Epinions datasets, when using the E1 edge probability method. As illustrated in the figure, both of the InfEst-based heuristics tend to give comparable approximation ratios to those MonteCarlo'(20). In particular, for the significantly larger Epinions dataset, MonteCarlo'(20) displayed an overall inferior performance, in terms of estimation, relative to the InfEst'(20) and InfEst"(20,20), which seemed to have given stable approximation ratios. This is likely due to the high variance in the spread process in this more massive network. This more noisy behavior of the spread process can also be seen in the more noticeable error bars.

Regarding running time, MonteCarlo'(20) outperformed the other two heuristics. This is to be expected: InfEst is designed to be parallelizable, and hence incurs overhead beyond a straight Monte Carlo simulation. However, in most cases the ratio of the running time of MonteCarlo'(20) to that of InfEst"(20,20) was around 10. We view this as an acceptable amount of overhead, given the added robustness in approximation ratio and the parallelizability of the InfEst methodology.

Acknowledgements

We thank the anonymous reviewers and meta-reviewers for their valuable comments which greatly contributed to the exposition of this work. This research has been supported by NSF grants CCF-1301976 and CAREER CCF-1452961, as well as a Google Research award.

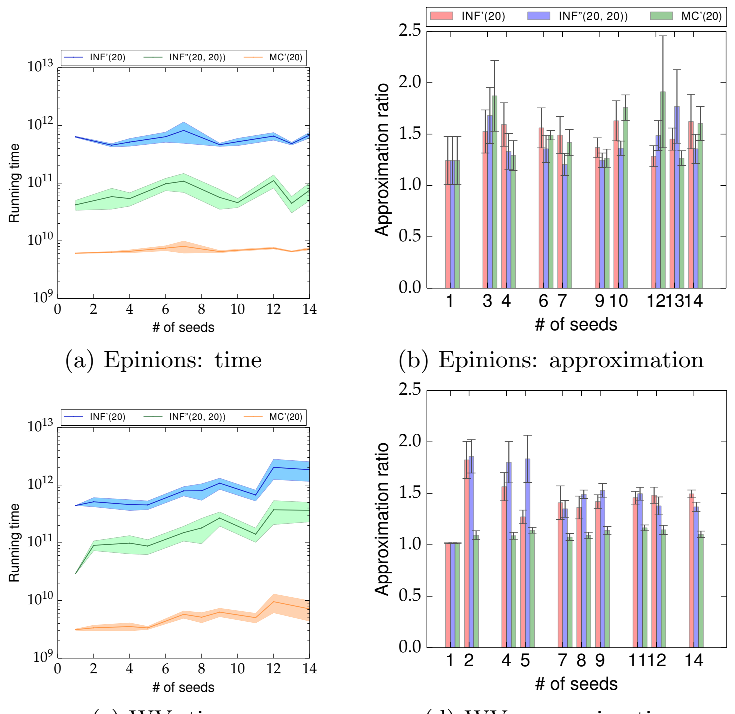

(a) Epinions: time (b) Epinions: approximation (c) WV: time (d) WV: approximation

Figure 6: Time (log scale) and approximation of heuristics.

7. REFERENCES

[1] Z. Bar-Yossef and L.-T. Mashiach. Local approximation of pagerank and reverse pagerank. In *CIKM*, pages 279–288, 2008.

[2] C. Borgs, M. Brautbar, J. T. Chayes, and B. Lucier. Maximizing social influence in nearly optimal time. In *SODA, Portland, Oregon, USA, January 5-7*, pages 946–957, 2014.

[3] M. Bressan, E. Peserico, and L. Pretto. Approximating pagerank locally with sublinear query complexity. *CoRR*, abs/1404.1864, 2014.

[4] J. J. Brown and P. H. Reingen. Social ties and word-of-mouth referral behavior. *Journal of Consumer Research*, 14(3):350–62, December 1987.

[5] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2), 2008.

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

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

[8] F. Chierichetti, J. M. Kleinberg, and D. Liben-Nowell. Reconstructing patterns of information diffusion from incomplete observations. In *NIPS*, pages 792–800, 2011.

[9] F. Chierichetti, R. Kumar, and A. Tomkins. Max-cover in map-reduce. WWW '10, pages 231–240, New York, NY, USA, 2010. ACM.

[10] E. Cohen, D. Delling, T. Pajor, and R. F. Werneck. Sketch-based influence maximization and computation: Scaling up with guarantees. In *Conference on Information and Knowledge Management, CIKM*, pages 629–638, 2014.

[11] H. Daneshmand, M. Gomez-Rodriguez, L. Song, and B. Schölkopf. Estimating diffusion network structures: Recovery conditions, sample complexity & soft-thresholding algorithm. In *ICML*, pages 793–801, 2014.

[12] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, Jan. 2008.

[13] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. In *SOSP 2007, Stevenson, Washington, USA, October 14-17, 2007*, pages 205–220, 2007.

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

[15] N. Du, Y. Liang, M. Balcan, and L. Song. Influence function learning in information diffusion networks. In *ICML 2014, Beijing, China, 21-26 June 2014*, pages 2016–2024, 2014.

[16] S. Goel, D. J. Watts, and D. G. Goldstein. The structure of online diffusion networks. EC '12, pages 623–638, New York, NY, USA, 2012. ACM.

[17] B. Golub and M. Jackson. Using selection bias to explain the observed structure of internet diffusions. *Proceedings of the National Academy of Sciences*, pages 40–47, 2010.

[18] M. Gomez-Rodriguez, J. Leskovec, and A. Krause. Inferring networks of diffusion and influence. In *KDD*, 2010.

[19] M. Gomez-Rodriguez, J. Leskovec, and B. Schölkopf. Modeling information propagation with survival theory. In *ICML*, 2013.

[20] M. Granovetter. Threshold models of collective behavior. *The American Journal of Sociology*, 83(6):1420–1443, 1978.

[21] M. Granovetter. The strength of weak ties: A network theory revisited. *Sociological Theory*, 1, 1983.

[22] U. Kang, C. E. Tsourakakis, A. P. Appel, C. Faloutsos, and J. Leskovec. Hadi: Mining radii of large graphs. *TKDD*, 5(2):8, 2011.

[23] H. J. Karloff, S. Suri, and S. Vassilvitskii. A model of computation for mapreduce. In *SODA*, pages 938–948, 2010.

[24] D. Kempe, J. Kleinberg, and E. Tardos. Maximizing the spread of influence through a social network. In *KDD*, pages 137–146, New York, NY, USA, 2003. ACM.

[25] R. Kumar, B. Moseley, S. Vassilvitskii, and A. Vattani. Fast greedy algorithms in mapreduce and streaming. SPAA '13, pages 1–10, New York, NY, USA, 2013. ACM.

[26] S. Lattanzi, B. Moseley, S. Suri, and S. Vassilvitskii. Filtering: A method for solving graph problems in mapreduce. SPAA '11, pages 85–94, New York, NY, USA, 2011. ACM.

[27] J. Leskovec, D. Chakrabarti, J. Kleinberg, and C. Faloutsos. Realistic, mathematically tractable graph generation and evolution, using kronecker multiplication. PKDD'05, pages 133–145, Berlin, Heidelberg, 2005. Springer-Verlag.

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

[29] J. Lin and C. Dyer. *Data-Intensive Text Processing with MapReduce*. Morgan and Claypool Publishers, 2010.

[30] M. W. Macy. Chains of cooperation: Threshold effects in collective action. *American Sociological Review*, 56, 1991.

[31] B. Mirzasoleiman, A. Karbasi, R. Sarkar, and A. Krause. Distributed submodular maximization: Identifying representative elements in massive data. In *NIPS*, pages 2049–2057, 2013.

[32] N. Nisan. Crew prams and decision trees. *SIAM J. Comput.*, 20(6):999–1007, 1991.

[33] S. C. Rhea, B. Godfrey, B. Karp, J. Kubiatowicz, S. Ratnasamy, S. Shenker, I. Stoica, and H. Yu. Opendht: a public DHT service and its uses. In *Proceedings of the ACM SIGCOMM 2005 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, Philadelphia, Pennsylvania, USA, August 22-26, 2005*, pages 73–84, 2005.

[34] T. C. Schelling. *Micromotives and Macrobehavior*. Norton, 1978.

[35] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. MSST '10, Washington, DC, USA, 2010. IEEE Computer Society.

[36] I. Stoica, R. Morris, D. R. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM*, pages 149–160, 2001.

[37] J. Ullman and M. Yannakakis. High-probability parallel transitive closure algorithms. SPAA '90, pages 200–209, New York, NY, USA, 1990. ACM.

[38] C. Wang, W. Chen, and Y. Wang. Scalable influence maximization for independent cascade model in large-scale social networks. *Data Min. Knowl. Discov.*, 25(3):545–576, 2012.

[39] D. J. Watts and S. H. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 393(6684):440–442, 1998.

[40] N. C. Wormald. Models of random regular graphs. *London Mathematical Society Lecture Note Series*, pages 239–298, 1999.