Improving Efficiency of Maximizing Spread in the Flow Authority Model for Large Sparse Networks

Philip K. Chan
School of Computing
Florida Institute of Technology
Melbourne, Florida 32901
Email: pkc@cs.fit.edu

Ebad Ahmadzadeh
School of Computing
Florida Institute of Technology
Melbourne, Florida 32901
Email: mahmadzadehe2012@my.fit.edu

***Abstract*—Given a network, finding a set of nodes that maximizes their spread of influence has a number of applications. Based on the Flow Authority model, we propose the VSM (Vector-based Spread Maximization) algorithm that estimates the SSS (Steady-state Spread) objective function for multiple seeds based on SSS values of individual seeds to reduce computation. Based on three real-world large sparse networks, our empirical results indicate that VSM is more effective than two existing algorithms, and two orders of magnitude more efficient than one of them.**

***Index Terms*—spread maximization; flow authority model; large sparse networks; efficiency**

I. Introduction

Given a network, a number of researchers in different areas have studied how to find the set of most "important" nodes in the network. In social networks, important nodes are influential people [1]. For marketing purposes, the problem is to decide whom to market first, who in turn influence others, so that the spread of influence is maximized. In detecting water contaminants, important nodes are sensor locations in the water distribution network [2]. The problem is to identify a set of sensor locations that minimizes the total cost of detecting contaminants in the network. Other applications include finding a small set of "leaders" who could coordinate a network of distributed agents/robots.

Aggarwal et al. [3] introduce the Flow Authority (FA) model, which specifies how information flows from nodes to their neighbors. Given a graph, the main question is how to efficiently find a set of nodes that initially has the information and maximizes the expected number of nodes that will assimilate the information. The authors define Steady-state Spread (SSS) as the objective function and propose RankedReplace as an algorithm to maximize SSS. RankedReplace repeatedly calls the objective function to guide its search for the top set of seeds, however, each SSS call can be time consuming.

We propose VSM that leverages spread information in the initial SSS calls and estimates the SSS value for future calls to reduce computation. Our main contributions include:

- an efficient method to estimate SSS of multiple seeds from SSS of individual seeds,
- our proposed VSM algorithm is more effective than two existing algorithms and two orders of magnitude more efficient than RankedReplace in 3 large real-world datasets, and
- a four orders of magnitude more efficient SSS algorithm.

We discuss related work in Sec. II. Sec. III provides the problem statement and more background on the RankedReplace algorithm and the SSS function. Sec. IV introduces our VSM algorithm. Sec. V discusses a more efficient SSS algorithm for large sparse graphs. We evaluate our algorithms in Sec. VI and conclude in Sec. VII.

II. Related Work

Given a graph, Kempe et al. [1] discuss two diffusion models: Independent Cascade (IC) and Linear Threshold (LT). In the IC model, each node has only one chance to influence its neighbors. In the LT model, each node has an activation threshold; a node is active when the total influence from its active neighbors exceeds the threshold. Aggarwal et al. [3] introduce the Flow Authority model. Different from the two models above, the expected number of active nodes is directly estimated, instead of running (e.g. 10,000 [1]) Monte Carlo simulations, which could be computationally expensive. The key question of these three models is how to find the initial seed set that maximizes the expected number of active nodes.

The methods used in related work can be categorized into two general approaches. The first general approach uses an evaluation function for a node or a set of nodes to find the top set of nodes. Different sets of nodes are generated and the evaluation function guides the selection. For example, the Greedy [1] algorithm starts with sets, each containing only one node, successively generates sets with one additional node, and selects the set that maximizes the evaluation function, which is the objective function for the problem. Kimura et al. [4] use the Greedy algorithm and bond percolation as the evaluation function. Based on the submodularity property of the objective function, CELF [2] improves the Greedy algorithm with a "lazy forward" evaluation technique, which prunes nodes that cannot improve the set. Degree Discount [5] is similar except the evaluation function is a heuristic based on the neighbors of the node. PMIA [6] uses a tree of nodes with maximum influence to construct a heuristic as the evaluation function. CGA [7] first identifies communities in the graph and then

greedily selects nodes from the communities. SIMPATH [8] estimates the spread from a set by exploring paths from the set up to a threshold as the evaluation function. Jiang et al. [9] initially select a set of random seeds then use simulated annealing to evaluate neighboring seed sets by replacing one of the seeds. IPA [10] evaluates each candidate by calculating the spread from the current set and the candidate to the descendants of the candidate. They store influence paths for each node, but they limit the number of nodes to reduce memory usage. Borgs et al. [11] propose a nearly-optimal-time algorithm for the IC model that chooses a random set of initial nodes and finds their ancestors, which are seed candidates. The evaluation function of a seed candidate is the number of times it is an ancestor. Recognizing the large constant in the time complexity, TIM+ [12] improves Borgs et al.' algorithm by bounding the number of initial nodes to a smaller number. IMM [13] further reduces the number of initial nodes by a Martingale approach, which allows some dependency between successive runs of finding ancestors to determine the number of initial nodes. Based on the Flow Authority model, Aggarwal et al. [3] propose Steady State Spread (SSS) as the objective and evaluation functions for a node in their algorithms. RankedReplace selects individual nodes with the highest SSS as the initial set and successively attempts to replace one in the set by one outside the set. Using the DBLP data set, they illustrated that the top set of authors found by their algorithms is more recognizable than sets found by two other algorithms.

The second general approach propagates values according to the graph structure and selects the top nodes with the highest values. Bayes Traceback [3] starts with equal probability for each node and back propagates probabilities to its in-neighbors. At each iteration, a fraction of the nodes with the lowest probabilities are removed and the probabilities of the remaining nodes are redistributed.

In the first general approach, many algorithms greedily add a node to the set and do not attempt to change previously added nodes. The exceptions are the simulated annealing approach [9] and RankedReplace[3]. Both approaches iteratively evaluate the neighboring seed sets, by replacing a node in the seed set with another not in the seed set. If the replacement improves the objective function, both approaches keep the replacement. However, simulated annealing keeps a replacement that does not improve with some probability. In simulated annealing, the initial seed set is randomly selected. However, RankedReplace creates the initial set by selecting nodes based on their individual values of the objective function in the Rank step. Since RankedReplace chooses the initial seeds independently without considering their interactions, more replacements might be needed in the Replace step. Also, each attempt of replacement invokes the objective function SSS, which could be expensive.

Algorithm 1 SSS(S, P)

1: $\forall i \in S, \quad q^0(i) \leftarrow 1$
2: $\forall i \notin S, \quad q^0(i) \leftarrow 0$
3: $t \leftarrow 0$
4: **repeat**
5: $\quad \forall i \in S, \quad q^{t+1}(i) \leftarrow 1$
6: $\quad \forall i \notin S, \quad q^{t+1}(i) \leftarrow 1 - \prod_{j \in N(i)} (1 - P_{ji} \cdot q^t(j))$
7: $\quad C_{t+1} \leftarrow \sum_{i \notin S} |q^{t+1}(i) - q^t(i)|$
8: $\quad t \leftarrow t + 1$
9: **until** $C_t < 0.01 \cdot C_1$
10: **return** $\sum_{i \notin S} q^t(i)$

III. Problem Statement and Background

Flow authorities are the nodes that cause maximum spread of information in social networks. We use the same formulation as Aggarwal et al. [3]. Consider a directed network $G = (V, E)$, where V is a set of nodes and E is a set of edges. Each edge $e = (i, j)$ of the network is associated with a propagation probability P_{ij} which specifies the probability by which the information propagated by node *i* is absorbed at the destination node *j*. This is based on the assumption that if node *i* has the information, all its neighbors are automatically exposed to the information, and the assimilation probability is P_{ij} for each neighbor node *j*. Given a set *S* of *k* nodes, we define $\pi(i)$ to be the steady-state probability that node *i* assimilates the information. The expected number of nodes, or Steady-State Spread (SSS), which assimilates the information is: $SSS(S) = \sum_{i \in V} \pi(i)$. The goal is to find *S* of size *k* such that $SSS(S)$ is maximized.

A. Steady-state Spread and RankedReplace

Aggarwal et al. [3] proposed RankedReplace to find S such that the objective function $SSS(S)$ is maximized. They first calculate $\pi(i)$, which is the steady-state probability that node i assimilates the information (is activated). The basic idea is that node i is activated if it receives the information from at least one of its neighbors. Then, $\pi(i)$ is calculated as:

$$\pi(i) = 1 - \prod_{j \in N(i)} (1 - \pi(j) p_{ji}), \tag{1}$$

where $N(i)$ is the set of in-neighbors of node i, and p_{ji} is the propagation probability from node j to i. Given an initial set S and a propagation probability matrix P, Alg. 1 calculates *SSS*. $q^t(i)$ is the estimate of the steady state probability of node i having the information at time t. Initially, the value of $q(i)$ is set to 1 where $i \in S$, and 0 where $i \notin S$. Then $q^t(i)$ is iteratively updated by calculating the probability that at least one of i's neighbors spreads the information to i (line 6) until the total spread converges. The Rank step in RankedReplace performs SSS for each node in the graph and the top k nodes form the initial S. In the Replace step, a node in S is replaced with a node in $V \backslash S$ if the SSS value improves. The algorithm stops if no replacement was made after r trials.

Algorithm 2 Greedy(f, k)

```
S ← ∅
for i ← 1 to k do
    u ← argmax_{c∈V\S}(f(S ∪ {c}) − f(S))        ▷ gain
    S ← S ∪ {u}
return S
```

IV. Vector-based Spread Maximization

The Replace step in RankedReplace calls the expensive SSS (Alg. 1) for different seed sets. Also, the Rank step does not consider seed interactions to create the initial seed set, which could result in more replacements in the Replace step. Our VSM (Vector-based Spread Maximization) algorithm efficiently estimates SSS and considers seed interactions.

A. Greedy Algorithm

For the IC and LT models, Kempe et. al. [1] show that the problem of finding a seed set S of size k that maximizes the total spread is NP-Hard. They also prove that if the objective function f is non-negative, monotone, and submodular, a general greedy approach, shown in Alg 2, guarantees a solution to be at least 1 - 1/e (63%) of the optimal solution. It iteratively finds new seed nodes that yield the highest spread gain, and it stops when k such seed nodes are found. Given the seed set *S*, spread function *f*, and a candidate node c, the gain is calculated as: $f(S \cup \{c\}) - f(S)$. SSS (Alg. 1) is an option for *f*, however, it is relatively computationally expensive. To improve efficiency, we propose estimating SSS without running Alg. 1 for seed sets with more than one seed.

B. Estimating SSS

To estimate SSS from multiple seeds, we store and use the last vector q^t from SSS (Algorithm 1) with seed set of size 1. This vector, which we call SSS-vector, contains the influence spread value of the given seed set on every node in the graph. Consider v is a seed, we denote the SSS-vector of v as q_v and $q_v(i)$ as the spread from v to i. By storing the SSS-vectors, we can calculate the estimated spread, called eSSS, much faster. For example, for a set of nodes $v_1, v_2, ..., v_n$, if we calculate and store the SSS-vectors for each of them, then in order to calculate $SSS(v_1, v_2)$ we do not need to run SSS. Instead, we could find the estimated SSS (eSSS) by aggregating the stored SSS-Vectors of v_1 and v_2. To aggregate the SSS-vectors (or "vectors" when the context is clear) from two seeds, we calculate the probability that the information is spread to *i* from the first seed or the second seed. This probability is the probability complement of neither the first seed nor the second seed spread the information to *i*. Consider q_x is the vector from seed x, q_y is the vector from seed y, and $q_x(i)$ and $q_y(i)$ are the spread probabilities at node i from the two seeds. We use $\oplus$ to denote the aggregate operator for two vectors:

$$\begin{aligned} q_{\{x,y\}} &= q_x \oplus q_y \\ q_{\{x,y\}}(i) &= 1 - (1 - q_x(i)) \times (1 - q_y(i)), \\ &= q_x(i) + q_y(i) - q_x(i)q_y(i), \end{aligned} \tag{2}$$

where vector $q_{\{x,y\}}$ is the result of aggregating vectors q_x and q_y, and $q_{\{x,y\}}(i)$ is the aggregated spread to node i. The general case for aggregating vectors is:

$$\begin{aligned} q_S &= q_{v_1} \oplus q_{v_2} \oplus ... \oplus q_{v_k} \\ q_S(i) &= 1 - \prod_j^k (1 - q_{v_j}(i)), \end{aligned} \tag{3}$$

where $S = \bigcup_j^k \{v_j\}$. The aggregate operator is "cumulative:"

$$q_{S \cup \{c\}} = q_S \oplus q_c$$

since:

$$\begin{aligned} q_{S\cup\{c\}}(i) &= 1 - \left(\prod_j^k (1 - q_{v_j}(i))\right)(1 - q_c(i)) \\ &= 1 - \left(1 - \left(1 - \prod_j^k (1 - q_{v_j}(i))\right)\right)(1 - q_c(i)) \\ &= 1 - (1 - q_S(i))(1 - q_c(i)) \end{aligned}$$

That is, we can aggregate the SSS-vector of candidate c and the SSS-Vector of S without using Eq.3. We note that Eq.2 (or similarly Eq.3) is only needed when $q_x(i)$ and $q_y(i)$ are both positive, which means that both *x* and *y* influence *i*. Otherwise, if one of them is zero, $q_{S\cup\{c\}}(i)$ is updated to be the non-zero value (which is mathematically equivalent to Eq.2). Similarly, if both $q_x(i)$ and $q_y(i)$ are zero, $q_{S\cup\{c\}}(i)$ is not updated. We next discuss seed interactions (common paths and blocked seeds) and techniques that improve the estimation.

1) Multiple Seeds via a Common Path: One source of error in the estimation of SSS is when multiple seeds influence a node via a common path. We show this issue with an example illustrated in Fig. 1(A) where two seeds *a* and *b* share a common path to influence node *y*. In this case, $SSS(\{a, b\})$ to node *y* is $\alpha\gamma + \beta\gamma - \alpha\beta\gamma$. Because, according to Alg. 1, at the first iteration, $SSS(\{a, b\})$ is zero on *y*, but it is $\alpha+\beta-\alpha\beta$ on *x*. At the second iteration $SSS(\{a, b\})$ becomes $\alpha\gamma + \beta\gamma - \alpha\beta\gamma$ to *y*. However, the estimation, using Eq.2, would yield $\alpha\gamma + \beta\gamma - \alpha\beta\gamma^2$. Because $q_a(y) = \alpha\gamma$ and $q_b(y) = \beta\gamma$. This estimation could be corrected by dividing the third term by γ. Our second example shown in Fig. 1(B) depicts a longer common path from the two seeds to node *z*. In this case, similar to the previous one, $SSS(a, b)$ to node *z* is $\alpha\gamma\lambda + \beta\gamma\lambda - \alpha\beta\gamma\lambda$. However, the estimation using Eq.2 is $\alpha\gamma\lambda + \beta\gamma\lambda - \alpha\beta\gamma^2\lambda^2$. The weights γ and λ in the common path should be discounted from the third term. Generally, the estimation could be corrected by dividing the third term by the product of the weights on the common path.

Our last example illustrated in Fig. 1(C), shows a case where more than one common path exist from the seeds to a node like *z*. However, in this case, we cannot correct the estimation based on the vector entries and the common path weights, as the terms are not easy to decompose unless we store more information. So, we only consider to address situations like cases A and B. Another reason for this decision is that finding all such common paths can be computationally expensive. Calculating a single common path seems to be an appropriate balance between accuracy **and** efficiency.

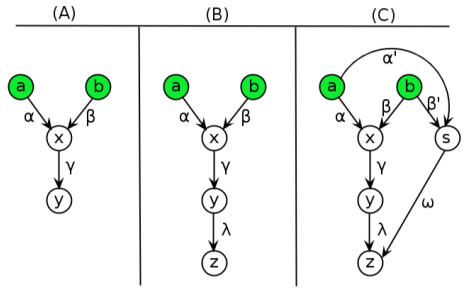

Fig. 1. 3 Examples showing common paths in which 2 seeds flow (Seeds are green, other nodes are white).

Hence, Eq. 2 is updated as:

$$q_{\{x,y\}}(i) = q_x(i) + q_y(i) - [q_x(i)q_y(i)]/cpw_{xy}(i), \quad (4)$$

where $cpw_{xy}(i)$ is the product of common path weights from x and y to i. To find a common path in aggregating two vectors, we use depth-first search as shown in Alg. 3. The algorithm finds a common path from the seed set *S* and candidate *c* to *targetNode*, where *targetNode* is the starting node for the depth-first search. q_c is the SSS vector for the candidate *c*, and q_S is a vector representing all seed vectors aggregated. We note that q_S can represent one or more seeds, where *S* in q_S denotes the set of seeds. *hopLimit* is the maximum hop parameter. The output is the product of the weights on a common path from the seeds to the target node, or 1.0 if a common path does not exist. The algorithm starts by getting the in-neighbors (N_{in}) of the target node (e.g. node z in Fig. 1). It does not consider a neighbor that is not a descendant of both *c* and nodes in *S* (line 7). Otherwise, if the neighbor is a descendant of both, *currentNode* is updated by the node ID of the neighbor (line 9). The while loop has two stopping criteria (line 4). First, the number of hops is limited to *hopLimit*, which we discuss further in Section V. Second, we stop when the candidate node *c* is reached because the effect of previous seeds on the common path has already been calculated. The *break* statement (line 10) limits the search to finding only a single common path to balance accuracy and efficiency.

As we discussed in Section IV-B, when *targetNode* (node *i*) is not influenced by either or both *c* (node *x*) and *S* (node *y*), we do not need Eq.2 and Alg. 3 is not called to adjust Eq.2. If *targetNode* is influenced by **both** *c* and *S*, adjustment to Eq.2 might be needed and Alg. 3 is called. When a common path is not found, the algorithm returns 1.0 and adjustment is not applied to the estimation.

2) Ancestor Checking for Blocked Seeds: Before we aggregate SSS-vectors, we need to check whether any seed is *blocked* by another seed and update the SSS-vectors if necessary. PMIA [6] and IPA [10] for the IC model also consider blocked seeds. Node v is an ancestor of node u if there exists a path from v to u in the graph. Similarly, u is considered a descendant of v. Since if a path from node v to node u exists, the spread from v to u is larger than zero. Hence, to check if v is an ancestor of u, we check if $q_v(u)$ is

Algorithm 3 CommonPathWeights($targetNode$, c, q_c, q_S, $hopLimit$)

1: $hopCount \leftarrow 0$
2: $cpw \leftarrow 1.0$
3: $currentNode \leftarrow targetNode$
4: **while** $hopCount < hopLimit$ **and** $c \notin N_{in}(currentNode)$ **do**
5: $\quad hopCount \leftarrow hopCount + 1$
6: $\quad$ **for** $u, w \in N_{in}(currentNode)$ **do** ▷ w is the weight from an in-neighbor to currentNode
7: $\quad\quad$ **if** $q_c(u) > 0$ **and** $q_S(u) > 0$ **then** ▷ u is a descendant of c and the seeds in S
8: $\quad\quad\quad cpw \leftarrow cpw * w$
9: $\quad\quad\quad currentNode \leftarrow u$
10: $\quad\quad\quad break$
return cpw

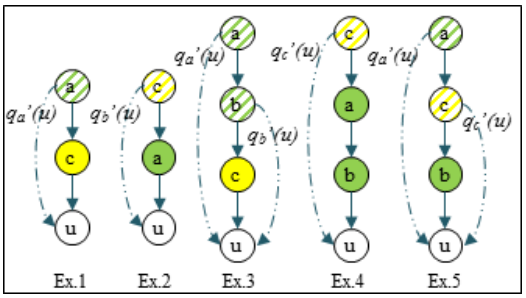

Fig. 2. Examples of Blocked seeds (seeds in green, candidates in yellow, blocked nodes in stripes, modified spread of blocked nodes in dashed paths)

positive. Given a seed s, a candidate seed c, and a non-seed node v, s is considered blocked by c with respect to v if s is an ancestor of c, and there exists a path from s to v that passes through c. Similarly, c is considered blocked by s with respect to v if c is an ancestor of s, and there exists a path from c to v that passes through s.

When a seed s is blocked with respect to a node v, the spread to v from s is reduced. Not considering the blocked nodes can lead to eSSS overestimating the actual SSS. We discuss how to update the SSS-vectors of blocked nodes with five examples depicted in Fig. 2.

Example 1: Consider S contains a single seed a, and candidate c is being added. Here we aim to calculate $eSSS(\{a, c\})$. Suppose that a is blocked by c (w.r.t u). Since c becomes a seed (c contains the information), a cannot influence c and a cannot influence u via c any more. However, a can influence u via other paths not containing c, which is a discounted spread from a to c. That is, we need to update the spread to u in the SSS-Vector of a to the discounted spread.

Before c becomes a seed, let ω be the spread from a to u, which is $q_a(u)$. ω is the aggregate of two components: spread from a via c and spread from a *not* via c. Let α be the spread from a to c, which is $q_a(c)$, and β be the spread from c to u, which is $q_c(u)$. We estimate the first component by $\alpha\beta$. Let γ be the second component, which is the discounted spread when c becomes a seed. Figure 2 depicts this example. Since

ω is the aggregate spread of the two components, from Eq. 2:

$$\omega = 1 - (1 - \alpha\beta) \times (1 - \gamma) = \alpha\beta + \gamma - \alpha\beta\gamma$$
$$\gamma = \frac{\omega - \alpha\beta}{1 - \alpha\beta} \quad (5)$$

That is, we update $q_a(u)$ to the discounted spread γ when c becomes a seed. Since the transmission probabilities are usually less than 1, α and β are generally less than 1 and the denominator in Eq. 5 generally cannot be zero. If α and β are both 1, the value of $q_a(u)$ is not important since $q_c(u)$ (spread from c to u) is 1 and the total spread from all nodes to u cannot exceed 1.

To generalize the calculation, ω represents the spread from a blocked seed to a node *u* before considering candidate *c*. α denotes the spread from the blocked seed to the blocking seed and β the spread from the blocking seed to *v*. γ is the spread from the blocked seed to *u* via paths not involving the blocking seed and is hence the updated (discounted) spread from the blocked seed to *u* after considering candidate *c*.

Example 2: Seed *a* is an ancestor of candidate *c* in Example 1, we now consider the opposite case when *c* is an ancestor of *a*. That is, *c* is blocked by *a* and the SSS-vector of *c* need to be updated. The blocked node is *c* and the blocking node is *a*. Hence, ω is $q_c(u)$, α is $q_c(a)$, β is $q_a(u)$, and $q_c(u)$ is updated to be γ in Eq. 5.

Example 3: A more complicated case is when there are multiple seeds in S, and c can be ancestor/descendant of multiple of them. Consider set S containing nodes a, b and candidate node c. Also, a was added to *S* before *b*. Here we aim to calculate $eSSS(\{a, b, c\})$. Suppose that *a* is an ancestor of *b* and *b* is an ancestor of *c*. That is, a is blocked by b, and b is blocked by c (w.r.t u). Since *a* is blocked by *b*, the spread from *a* to *u* has been updated to the spread via paths not involving *b* when *b* was a candidate previously. When *c* becomes a candidate, we need to further update the spread of *a* to *u* via paths not involving *c* — ω is $q_a(u)$, α is $q_a(c)$, β is $q_c(u)$, and $q_a(u)$ is updated to be γ in Eq. 5. Similarly, we need to update the spread of *b* to *u* via paths not involving *c* — ω is $q_b(u)$, α is $q_b(c)$, β is $q_c(u)$, and $q_b(u)$ is updated to be γ in Eq. 5.

Example 4: Consider *a* is an ancestor of *b* as in Example 3, but now *c* is an ancestor of *a*. That is, *c* is blocked by *a* and *a* is blocked by *b*. Since *a* is blocked by *b*, the spread from *a* to *u* has been updated to the spread via paths not involving *b* when *b* was a candidate previously. When *c* becomes a candidate, we need to update the spread of *c* to *u* via paths not involving *a* — ω is $q_c(u)$, α is $q_c(a)$, β is $q_a(u)$, and $q_c(u)$ is updated to be γ in Eq. 5. Though *a* is an ancestor of *b*, *c* might have paths to *u* via *b* but not *a*. Hence, we also need to update the spread of *c* to *u* via paths not involving *b* — ω is $q_c(u)$, α is $q_c(b)$, β is $q_b(u)$, and $q_c(u)$ is updated to be γ in Eq. 5.

Example 5: Consider *a* is an ancestor of *b* as in Example 3, but now *c* is "between" *a* and *b*. That is, *a* is blocked by *c* and *c* is blocked by *b*. Similar to Examples 3 and 4, the spread from *a* to *v* has been updated to the spread via paths not involving *b* when *b* was a candidate previously. When *c* becomes a candidate, there might be paths from *a* to *u* involving *c*, so we need to update the spread of *a* to *u* via paths not involving *c* — ω is $q_a(u)$, α is $q_a(c)$, β is $q_c(u)$, and $q_a(u)$ is updated to be γ in Eq. 5. Also, we need to update the spread of *c* to *u* via paths not involving *b* — ω is $q_c(u)$, α is $q_c(b)$, β is $q_b(u)$, and $q_c(u)$ is updated to be γ in Eq. 5. We update $q_a(u)$ and $q_c(u)$, but $q_c(u)$ is used to update $q_a(u)$. Based on some experiments, we choose to update $q_a(u)$ using the original $q_c(u)$, not the updated $q_c(u)$, to reduce error.

The above five examples help illustrate the general case, where we check if candidate *c* is an ancestor or descendant of each seed in *S*. If so, we use Eq. 5 to update the SSS-Vector of each blocked seed. This process helps increase the accuracy of eSSS. Alg. 4 illustrates how SSS-vectors are modified when blocking exists between the seeds and candidate. Parameter *c* is the candidate, *S* is the seed set, q_S is the aggregated vector for *S*, and *q* has the vector of each node in the graph. If the candidate is blocked by a seed, we update the vector for the candidate according to Eq. 5 (lines 4-8). If a seed is an ancestor of the candidate, we update the vector for the seed similarly (lines 9-13). Note that a cycle could exist that involves c and v, so we check for ancestors both ways in lines 4 and 9. When a seed or a candidate is not blocked, their vectors need not be modified (lines 15 and 17). Vectors for the seeds and candidate are returned.

In summary, our approach to estimating SSS is to first find blocked seeds and then update the SSS-Vectors of the blocked seeds (Eq. 5). Next, the vectors are aggregated (Eq. 3), with adjustments from common path weights (Eq. 4). The resulting SSS-Vector (q_S) is summed over all elements to obtain the eSSS value: $eSSS(S) = \sum_i^n q_S(i)$.

We note that *eSSS* might not estimate SSS perfectly. We check ancestors and update vectors only once before aggregating two vectors. However, in case of cycles involving seeds, the vectors should be updated multiple times until convergence. However, the likelihood of cycles is small within 3 hops. Also, we find the weights of only one common path, but multiple common paths might exist. Although additional computation can yield a more accurate estimation, we aim at a more efficient and relatively accurate estimation.

C. Improving efficiency of the Greedy Algorithm

The greedy algorithm calculates $eSSS(S \cup \{c\})$ for each candidate c and the gain efficiently: $gain(c, S) = eSSS(S \cup \{c\}) - eSSS(S)$. Because of the submodularity property of eSSS, we can improve the efficiency by not updating the gain of every candidate [2]. The submodularity property states: $gain\,(c, S \cup \{x\}) \leq gain(c, S)$, where x is an additional seed (the newly added seed in our case) and *c* is a candidate. That is, the gain for candidate c cannot be larger than the gain obtained from the previous level(s). (We consider the greedy algorithm conducts a tree-like search and selects one seed at each level.)

Theorem 1. *eSSS is submodular.*

Algorithm 4 BlockedVec(c, S, q_S, q)

```
1: c.blocked ← false
2: for v ∈ S do
3:     v.blocked ← false
4:     if c is an ancestor of v then                ▷ c is blocked by v
5:         c.blocked ← true
6:         for u ∈ {i|q_c(i) > 0} do
7:             αβ ← q_c(v) · q_v(u)
8:             q'_c(u) ← (q_c(u) − αβ)/(1 − αβ)
9:     if v is an ancestor of c then                ▷ v is blocked by c
10:        v.blocked ← true
11:        for u ∈ {i|q_v(i) > 0} do
12:            αβ ← q_v(c) · q_c(u)
13:            q'_v(u) ← (q_v(u) − αβ)/(1 − αβ)
14:    if v.blocked = false then
15:        q'_v ← q_v
16: if c.blocked = false then
17:    q'_c ← q_c
18: return [q'_{v_1}, ...q'_{v_|S|}, q'_c]
```

Proof. First, the additional seed in S can block other seeds, and hence can reduce the spread in their SSS-vectors. Second, from Eq.3, the gain of adding c to S at node i is:

$$gain(c,S,i) = \left(1-(1-q_c(i))\prod_j^k\left(1-q_{v_j}(i)\right)\right) - \left(1-\prod_j^k(1-q_{v_j}(i))\right) = q_c(i)\prod_j^k(1-q_{v_j}(i)).$$

Similarly, the gain of adding c to $S\cup\{x\}$ at node i is:

$$gain(c,S\cup\{x\},i) = q_c(i)(1-q_x(i))\prod_j^k(1-q_{v_j}(i))$$

Since $0 \le (1-q_x(i)) \le 1$:

$$gain(c,S\cup\{x\},i) \le gain(c,S,i).$$

After summing the gain at each node i:

$$gain(c,S\cup\{x\}) \le gain(c,S).$$

□

To utilize the submodularity property of eSSS for reducing computation, we use a priority queue to store the gain of each candidate and the level number when the gain was updated [14]. If the largest *updated* gain at the current level is larger than the largest *non-updated* gain, we prune the gain updates for the rest of the candidates, which cannot yield a larger updated gain. Hence, if the candidate at the head of the priority queue has been updated at the current level, we select it and add it to the seed set.

D. VSM Algorithm

Our VSM (Vector-based Spread Maximization) algorithm uses the Greedy algorithm (Alg. 2) with eSSS as the evaluation function f. Alg. 5 illustrates our VSM algorithm. VSM finds the SSS values for each vertex using Alg. 6 (an improved version of SSS, which is discussed in Sec. V), saves the SSS-vectors, and populates the priority queue (lines 1-4). We initialize the seed set, aggregated SSS-vector of the seed set, and eSSS value of the seed set (lines 5-8). While the candidate's level is less than the current level, its gain is not up to date (line 13). If the candidate is an ancestor or descendant of any seed in the seed set, the SSS-vectors are modified and the aggregated vector is updated (line 15-18). Otherwise, vectors of the seeds and candidate are not modified, we update the aggregated vector by aggregating existing aggregated vector of the seed set and vector of the candidate (line 20). We update the gain and level of the candidate in the priority queue and heapify the priority queue (lines 21-23). If the candidate is still at the head of the priority queue, the updated candidate is the best candidate and we do not use the previously saved vectors (lines 24-25). Otherwise, if the candidate's gain is larger than the largest gain so far, we save the modified vectors so that we do not need to recalculate them if the candidate eventually becomes the best candidate (lines 26-28). We remove the best candidate from the priority queue, update the vectors from the modified versions if needed, update the eSSS value of the seed sets, and add the best candidate to the seed set (lines 30-38).

1) Improving space and time: Alg. 5 generates a vector for each node (lines 1 - 4). For a graph with n nodes, the space complexity is $O(n^2)$, which could be prohibitive for large graphs. Based on initial experiments, we observe that many of the vectors are not used because of pruning in Sec. IV-C. Generally, fewer than $2k$ (k is the seed set size) vectors are used. Hence, an improved VSM generates vectors for only $2k$ nodes initially to reduce space and time. To select the initial $2k$ nodes, we find the nodes with the highest spread to their immediate out-neighbors. If needed (when the priority queue is exhausted), vectors for additional nodes are generated in the order of spread to their immediate neighbors. IPA [10] for the IC model similarly reduces space by limiting the priority queue to be of size $3k$. However, their approach does not allow further expansion of the priority queue if additional candidate nodes can improve gain. Moreover, IPA stores paths from each node, which requires more space than spread values from each node in VSM.

To calculate $q_{S'}$ for the updated S, line 18 of Alg. 5 aggregates all the seed vectors. However, not all seed vectors are updated since some seeds are not blocked by another seed. If the number of updated seed vectors is small, we can reduce computation. We use $\ominus$ to denote the deaggregate operator and follow Eq 2:

$$\begin{aligned} q_x &= q_{\{x,y\}} \ominus q_y \\ q_x(i) &= (q_{\{x,y\}}(i) - q_y(i))/(1-q_y(i)). \end{aligned} \tag{6}$$

Since the transmission probabilities are usually less than 1, $q_y(i)$ is generally less than 1 and the denominator in Eq. 6 generally cannot be zero. If $q_y(i)$ is 1, the value of $q_x(i)$ is not important since $q_y(i)$ (spread from y to i) is 1 and the total spread from all nodes to i cannot exceed 1.

Consider only seed v is blocked by candidate c, q_v is the vector before adding c, and q'_v is the updated vector afterwards. We can calculate $q_{S'}$ as: $q_S \ominus q_v \oplus q'_v \oplus q_c$, instead of aggregating all the seed vectors from scratch. Let b be

Algorithm 5 VSM(V, P, k)

1: **for** $v \in V$ **do**
2: $[q_v, v.gain] \leftarrow SSS2(\{v\}, P)$ ▷ Alg. 6
3: $v.level \leftarrow 1$
4: $PQ.insert(v)$ ▷ priority queue wrt gain
5: $v \leftarrow PQ.remove()$
6: $S \leftarrow \{v\}$ ▷ seed set
7: $q_S \leftarrow q_v$ ▷ aggregated vector of seed set
8: $eSSSofS \leftarrow v.gain$ ▷ eSSS value of seed set
9: **for** $level \leftarrow 2$ to k **do** ▷ k seeds
10: $bestGain \leftarrow 0$
11: $bestVec \leftarrow \emptyset$
12: $c \leftarrow PQ.head()$ ▷ candidate c
13: **while** $c.level < level$ **do** ▷ c's gain is not up to date
14: $c.blockedSeeds \leftarrow false$
15: **if** c is ancestor/descendant of $v, v \in S$ **then**
16: $c.blockedSeeds \leftarrow true$
17: $[q'_{v_1} ... q'_{v_{|S|}}, q'_c] \leftarrow BlockedVec(c, S, q_S, q)$
18: $q_{S'} \leftarrow q'_{v_1} \oplus ... \oplus q'_{v_{|S|}} \oplus q'_c$
19: **else**
20: $q_{S'} \leftarrow q_S \oplus q_c$
21: $c.gain \leftarrow eSSS(q_{S'}) - eSSSofS$
22: $c.level \leftarrow level$
23: $PQ.heapify()$
24: **if** $PQ.head() = c$ **then** ▷ updated gain of c is best
25: $bestVec \leftarrow \emptyset$
26: **else if** $c.gain > bestGain$ **then**
27: $bestGain \leftarrow c.gain$
28: $bestVec \leftarrow [q'_{v_1}, ..., q'_{v_{|S|}}, q'_c, q_{S'}]$
29: $c \leftarrow PQ.head()$
30: $c \leftarrow PQ.remove()$ ▷ best candidate is found
31: **if** $bestVec \neq \emptyset$ **then**
32: $[q_{v_1}, ..., q_{v_{|S|}}, q_c, q_S] \leftarrow bestVec$
33: **else if** $c.blockedSeeds = true$ **then**
34: $[q_{v_1}, ..., q_{v_{|S|}}, q_c, q_S] \leftarrow [q'_{v_1}, ..., q'_{v_{|S|}}, q'_c, q_{S'}]$
35: **else**
36: $q_S \leftarrow q_{S'}$
37: $eSSSofS \leftarrow eSSSofS + c.gain$
38: $S \leftarrow S \cup c$
39: **return** S

Algorithm 6 SSS2(S, $P, hoplimit$)

1: $\forall i \in S \quad q^0(i) \leftarrow 1$
2: $\forall i \notin S \quad q^0(i) \leftarrow 0$
3: $t \leftarrow 0$
4: $A \leftarrow outNeighbors(S)$ ▷ activated nodes
5: $A_{new} \leftarrow A$ ▷ newly activated nodes
6: $A_{old} \leftarrow \emptyset$ ▷ activated in previous iteration
7: **repeat**
8: $\forall i \in S, \quad q^{t+1}(i) \leftarrow 1$
9: **if** $t > 0$ **then**
10: $A_{old} \leftarrow A$
11: $A \leftarrow (A_{old} \cup outNeighbors(A_{new}))\backslash S$
12: $A_{new} \leftarrow A\backslash A_{old}$
13: $\forall i \in A, \quad q^{t+1}(i) \leftarrow 1 - \prod_{j \in N(i)} (1 - P_{ji} \cdot q^t(j))$
14: $C_{t+1} \leftarrow \sum_{i \in A} |q^{t+1}(i) - q^t(i)|$
15: $t \leftarrow t + 1$
16: **until** $C_t < 0.01 \cdot C_1$ or $t \geq hoplimit$
17: **return** $\sum_{i \in A} q^t(i)$

the number of blocked seeds. Aggregating the seed vectors from scratch needs $|S| - 1$ aggregations. Deaggregating and aggregating the updated seed vectors needs b deaggregations and b aggregations. If $|S| - 1 < 2b$, VSM aggregates vectors from scratch; otherwise, it de/aggregates updated vectors.

E. Optimality, Time, and Space Complexity of VSM

Since VSM uses the Greedy algorithm (Alg. 2) with eSSS, which is non-negative, monotone, and submodular (Theorem 1), VSM guarantees that the found solution is at least $1-1/e$ (63%) of the optimal solution based on eSSS (Theorem 2.1 in [1]). Though eSSS is an estimate of SSS, which is the objective function, our empirical results indicate that eSSS is within 0.12% of SSS (Sec. VI-H).

To select k nodes from a graph of n nodes ($k \ll n$), the nested loop starting on line 9 dominates VSM's time—outer loop runs $O(k)$ times and inner loop runs $O(n)$ time [but $O(k)$ in practice due to pruning (Sec.IV-C)]. At each iteration of the inner loop, $O(kn)$ for $BlockedVec()$, $O(kn)$ for aggregating vectors, $O(\log n)$ for $heapify()$, and $O(kn)$ for copying into $savedVec$. Hence, VSM's time complexity is $O(k \cdot n \cdot kn)$ or $O(k^2 n^2)$ [but $O(k^3 n)$ in practice]. Since VSM stores vectors for $2k$ nodes (the dominant data structure), the space complexity is $O(kn)$.

V. More efficient SSS and GreedySSS

The SSS method in Alg. 1 updates the q value for all nodes except the seeds, however, many of them will remain zero, particularly in a large sparse graph. To improve the efficiency of SSS for large sparse graphs, we only update nodes that will have positive spread. We call these nodes "activated" nodes. We use the out-going edges of the activated nodes of the previous iteration to find the activated nodes of the current iteration. Also, we keep track of newly activated nodes in the previous iteration so that we only need to add their out-neighbors as activated nodes in the current iteration, otherwise we unnecessarily add out-neighbors that have been added in the previous iterations. We only consider activated nodes for updating q, total spread, and change in total spread. Moreover, the number of activated nodes can grow exponentially. To prevent finding a large number of activated nodes and not using them in the last iteration, we find activated nodes at the beginning of the loop for the current iteration rather than at the bottom of the loop for the next iteration.

Alg. 6 shows the improved algorithm called SSS2. We initialize the sets for activated nodes, newly activated nodes and old ones (lines 4-6). The activated nodes are updated to be the union of the old ones and out-neighbors of the newly activated nodes from the previous iteration. We then exclude the seeds and find the newly activated nodes (lines 10-12). We update q and C, and return the total spread considering only the activated nodes (lines 13, 14 and 17). [For further efficiency, we initialize q on line 2 based on the activated nodes in the previous SSS2 call on a subset of S (not included in Alg. 6 for simplicity).]

From seed s to a node i, the more hops there are between s and i, the smaller the spread is from s to i because of the

TABLE I
RUNNING TIME OF SSS2 VS. SSS (SECONDS) ON 100,000 NODES

Alg.	DBLP	LAST.FM	Twitter
SSS (Alg. 1)	16524.74	18512.06	12006.22
SSS2 (Alg. 6)	0.87	1.38	0.57

discount from propagation probability in each hop. Goyal et al. [8] observe that much of the spread is within 3 or 4 hops from the seeds. For efficiency, we stop updating q if the hop limit is exceeded (line 19). Note that when there is a hop limit, the return value might not closely estimate the steady state value since the convergence criterion of less than 1% change might not have met. Hence, when SSS is used as the objective function for evaluating and comparing different algorithms, we do *not* use a hop limit.

Since SSS2 is faster than SSS, we propose GreedySSS, which is the same as VSM, except that it calls SSS2 (Alg. 6) instead of estimating SSS from SSS-vectors. We would like to see if GreedySSS is more effective (but slower) than VSM because SSS values are not estimated.

VI. EXPERIMENTAL EVALUATION

A. Experimental Criteria

The main evaluation criterion is effectiveness as measured by SSS2 (Alg. 6) *without a hop limit*. To evaluate efficiency, we measure the CPU running time. To evaluate the accuracy of eSSS, we measure the % difference, which is $(eSSS - SSS)/SSS * 100\%$. To evaluate the amount of storage for the SSS-vectors, we measure the number of (positive) entries in the SSS-vectors.

B. Experimental Data and Procedures

We use three datasets: DBLP, Last.fm, and Twitter from Aggarwal el. al [3]. DBLP has 684,911 authors and 7,764,604 edges. Last.fm has 818,800 users and 3,340,954 friendships. Twitter has 1,994,092 users and 6,450,193 edges.

We evaluate our proposed VSM (Sec. IV) and GreedySSS (Sec. V), and compare them with RankedReplace [3] and Bayes Traceback [3]. For VSM, we evaluate two versions: with or without ancestor checking for blocked seeds. We varied k from 20 to 100, with an increment of 20 as in [3]. VSM, GreedySSS and RankedReplace need to calculate SSS and we use our faster SSS2 algorithm (Alg. 6) for a comparison that focus on differences not contributed by the improvement due to SSS2. The hop limit for SSS2 is 3. The replacement factor r is 10 for RankedReplace. The discard fraction f for Bayes Traceback is 0.25, 0.2, and 0.3 for DBLP, Last.fm and Twitter respectively (the parameters were selected to maximize effectiveness). The algorithms were implemented in Python. The implementations were run on a 128GB, 8-core virtual machine running on ESXi 6.0.0 on a Dell PowerEdge M620 containing 2x Intel Xeon E5-2630 V2 @ 2.6 GHz with Ubuntu Linux 14.04.

TABLE II
HOP LIMIT VS. SSS VALUES (30K NODES)

Hop	AC	k=20	k=40	k=60	k=80	k=100
DBLP						
1	false	36.51	66.94	95.52	122.8	149.2
2	false	36.54	67.04	95.47	122.8	149.2
3	false	36.54	67.04	95.46	122.8	149.1
no limit	false	36.54	67.04	95.46	122.8	149.1
1	true	36.51	66.81	95.52	122.7	149.2
2	true	36.56	67.04	95.58	122.8	149.2
3	true	36.56	67.04	95.57	122.8	149.2
no limit	true	36.56	67.04	95.57	122.8	149.2
LAST.FM						
1	false	61.8	101.7	136.9	170.6	203.5
2	false	61.8	101.8	137.2	171.0	203.7
3	false	61.8	101.8	137.2	171.0	203.7
no limit	false	61.8	101.8	137.2	171.0	203.7
1	true	61.8	101.7	136.9	170.6	203.5
2	true	61.8	101.8	137.2	171.0	203.8
3	true	61.8	101.8	137.2	171.0	203.8
no limit	true	61.8	101.8	137.2	171.0	203.8
Twitter						
1	false	33.86	62.10	89.54	116.1	141.9
2	false	33.87	62.23	89.57	116.1	142.0
3	false	33.87	62.23	89.57	116.1	142.0
no limit	false	33.87	62.23	89.57	116.1	142.0
1	true	33.86	62.10	89.54	116.1	141.9
2	true	33.87	62.23	89.57	116.2	142.0
3	true	33.87	62.23	89.57	116.2	142.0
no limit	true	33.87	62.23	89.57	116.2	142.0

C. Efficiency of SSS2

To compare the efficiency of our improved SSS2 (Alg. 6) with SSS (Alg. 1), we sampled 100,000 nodes from the three datasets and measured the running time of the two algorithms calculating SSS (without a hop limit) for all vertices in the subsets. The results in Table I indicate that our proposed improvement is about 4 orders of magnitude faster.

D. Selecting Hop Limit for SSS2 with Smaller Datasets

Our experiments with smaller datasets of 30K nodes indicate that VSM with hop limits of 2 and 3 achieves the same SSS as VSM with no hop limits (Table II). Interestingly, with and without ancestor checking for blocked seeds also yield the same SSS when the hop limit is 2, 3, or none. In terms of running time (not shown due to space limitation), raising the hop limit increases computation. As k increases, computation grows faster with ancestor checking than without ancestor checking. Interestingly, the increase in computation from a hop limit of 3 to none is much smaller than the increase from a hop limit of 2 to 3. For Last.fm and Twitter, the increase in computation from a hop limit of 3 to none is quite small. This indicates that a hop limit of 3 is close to convergence, which is used as a stopping criterion when hop limit is none. In summary, a hop limit of 2 or 3, with less computation, yields the same effectiveness as no hop limit for datasets with 30K nodes. We conservatively choose 3 as the default hop limit.

E. Effectiveness of Algorithms

Table III displays the effectiveness of different algorithms. Generally, VSM with ancestor checking for blocked seeds is

TABLE III
SSS OF ALGORITHMS BOLD: HIGHEST, UNDERLINE: $\leq 0.1\%$ FROM HIGHEST)

Algorithm	k=20	k=40	k=60	k=80	k=100
DBLP					
RankedReplace	97.8	170.9	236.6	297.8	355.3
BayesTraceback	67.8	123.7	170.7	224.9	298.8
GreedySSS	97.8	171.0	236.6	297.9	355.6
VSM ac=false	97.7	170.2	236.4	298.2	355.9
VSM ac=true	**98.2**	**171.5**	**237.3**	**298.7**	**356.0**
LAST.FM					
RankedReplace	568.4	816.8	1024.2	1210.6	1377.9
BayesTraceback	332.6	545.8	708.1	816.2	952.1
GreedySSS	568.4	816.8	1024.2	1210.6	1377.9
VSM ac=false	**571.6**	**822.4**	**1033.0**	**1221.5**	1389.7
VSM ac=true	**571.6**	**822.4**	1031.6	**1221.5**	**1390.1**
Twitter					
RankedReplace	314.9	489.0	625.0	742.9	847.6
BayesTracekback	189.7	311.0	414.6	522.5	595.9
GreedySSS	314.9	489.0	625.1	742.9	847.6
VSM ac=false	315.2	**489.7**	625.9	743.8	848.7
VSM ac=true	**315.3**	**489.7**	**626.0**	**744.0**	**849.7**

more effective than without ancestor checking. For DBLP, VSM with ancestor checking outperforms the other algorithms consistently. For Twitter, VSM with ancestor checking outperforms the other algorithms. For Last.fm, VSM with ancestor checking outperforms the others, except when k=60. Interestingly, GreedySSS is generally less effective than VSM with ancestor checking, even though VSM estimates SSS, and GreedySSS measures SSS. One reason might be SSS with a low hop limit has not converged, while eSSS is more accurate in adjusting SSS-vectors for blocked seeds. Overall, VSM with ancestor checking is more effective than the other algorithms across the three datasets.

Some of the SSS values are similar to the highest value—at most 0.1% difference from the highest value. For DBLP, when k=100 VSM without ancestor checking is similar to the most effective algorithm. For the Twitter dataset, VSM without ancestor checking has similar SSS values as the most effective algorithm at k= 20, 60, 80. For Last.fm, the two versions of VSM are similar to the most effective algorithm, except for VSM with ancestor checking at k=60. Overall, compared to VSM, BayesTraceback is significantly less effective, while the other algorithms are within 1% difference in effectiveness across the three datasets.

F. Efficiency of Algorithms

Figure 3 plots the running time of different algorithms. VSM with ancestor checking (hop=3) is about an order of magnitude faster than RankedReplace and VSM without ancestor checking (hop=3) is about 2 orders of magnitude faster. VSM without ancestor checking (hop=3) is generally faster (and more effective) than Bayes Traceback. Since GreedySSS measures SSS instead of estimating SSS, it is generally slower than VSM as expected. Note that we use our proposed SSS2 algorithm (Alg. 6) in RankedReplace in all our experiments. If we use the original SSS (Alg. 1), the original RankedReplace will be much slower (Sec. VI-C) [we did not wait for the original RankedReplace to complete after a few days].

TABLE IV
SSS VS HOP LIMIT AND ANCESTOR CHECKING

Hop	AC	k=20	k=40	k=60	k=80	k=100
DBLP						
2	false	97.7	170.7	236.5	298.0	354.8
3	false	97.7	170.2	236.4	298.2	355.9
2	true	98.0	171.5	237.2	297.4	355.5
3	true	98.2	171.5	237.3	298.7	356.0
LAST.FM						
2	false	571.6	822.0	1030.8	1220.6	1388.5
3	false	571.6	822.4	1033.0	1221.5	1389.7
2	true	571.6	821.9	1030.8	1220.0	1388.5
3	true	571.6	822.4	1031.6	1221.5	1390.1
Twitter						
2	false	315.2	489.7	625.5	743.7	848.4
3	false	315.2	489.7	625.9	743.8	848.7
2	true	315.3	489.7	626.0	743.7	848.7
3	true	315.3	489.7	626.0	744.0	849.7

G. Effectiveness, Efficiency, and Space in VSM

The effectiveness of VSM with hop limits from 2 to 3 is displayed in Table IV. Generally, increasing the hop limit increases the effectiveness. The difference between hop limits of 2 and 3 is at most 0.2% and sometimes non-existent, which is similar to our earlier experiments with smaller data sets. Checking ancestors for blocked seeds generally yields higher SSS. However, the improvement is at most 0.1% for Twitter and Last.fm. For DBLP, the improvement can be as high as 0.8%. This relatively small improvement is unexpected because checking ancestors for blocked seeds should improve the accuracy of eSSS. However, with small hop limits, the SSS vectors are less accurate, which might degrade the effectiveness of ancestor checking and adjusting the SSS-vectors for blocked seeds.

Figure 3 plots the CPU times. Computation grows with k and hop limit. Generally, increasing the hop limit by one could increase the computation by an order of magnitude due to more (positive) entries in the vectors (Table V) governed by out-degrees. Checking ancestors for blocked seeds could be 1 to 4 times slower. We also observe that the number of blocked seeds are generally small (data not shown) and deaggregating/aggregating updated vectors when appropriate, instead of always aggregating vectors, reduces computation (Sec. IV-D1). For $k > 80$ GreedySSS runs faster than VSM on DBLP, and for $k > 100$ it is expected to be slightly faster on Twitter and LAST.FM. However, its accuracy is consistently lower than VSM on all three data sets.

Table V displays the number of (positive) entries in SSS vectors that VSM stores for calculating eSSS with k=100. The needed memory is less than 1 GB, demonstrating the effectiveness of space reduction in Sec. IV-D1. When the hop limit increases, the number of entries grows rapidly due to the space complexity of $O(b^h)$, where b is the branching factor of a node and h is the hop limit. However, as we discussed above, we do not need a hop limit beyond 3. Table VI displays

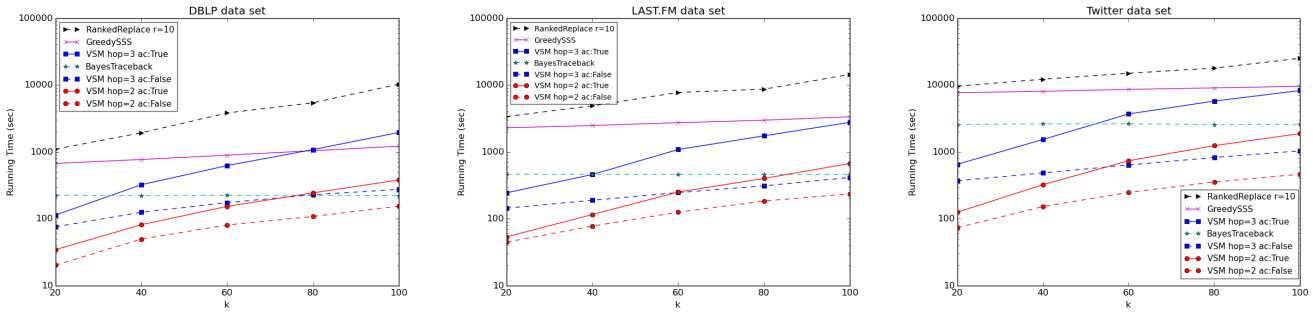

Fig. 3. Efficiency versus k

TABLE V
NUMBER OF ENTRIES ($\times 10^6$) IN VECTORS (k=100)

Hop limit	DBLP	LAST.FM	Twitter
2	0.65	1.40	2.97
3	5.30	10.23	26.59

TABLE VI
NUMBER OF UNIQUE NODES EVALUATED FOR SEED SET

Dataset	k=20	k=40	k=60	k=80	k=100
DBLP	25	48	73	102	133
LAST.FM	21	41	68	81	112
Twitter	22	43	66	84	116

the number of unique nodes VSM (with ancestor checking) evaluates for the seed set. Generally, VSM evaluates a small number of nodes beyond k, which make space reduction in Sec. IV-D1 and pruning in Sec. IV-C effective.

In summary, our results indicate that a hop limit of 2 or 3 and storing vectors for $2k$ nodes are reasonable. However, a hop limit of 2 is preferable to improve speed with a small loss of effectiveness. The additional computation for ancestor checking with a small hop limit might not be worthwhile for some datasets.

H. Accuracy of eSSS in VSM

Table VII shows the error rates of eSSS of the found seed set. When the hop limit increases, the SSS-vectors are more accurate and the error generally decreases. When we check ancestors for blocked seeds, the error generally decreases. Overall, eSSS with a hop limit of 3 and ancestor checking is within 0.12% of SSS.

VII. CONCLUDING REMARKS

We propose estimating SSS (eSSS) from SSS-vectors without running the more expensive *SSS* algorithm (Alg. 1 or 6) in our VSM algorithm. eSSS allows us to efficiently evaluate interactions among seeds and hence effectively select seeds. Also, eSSS is non-negative, monotone, and submodular, which allows VSM to guarantee $(1 - 1/e)$ optimality with respect to eSSS. We further propose considering only the top $2k$ candidates to reduce time and space, without affecting VSM's effectiveness for 3 real-world datasets. Our empirical results

TABLE VII
eSSS ERROR IN PERCENTAGE

Hop	AC	k=20	k=40	k=60	k=80	k=100	Avg. of Abs(err)
DBLP							
2	false	-2.14	-2.71	-3.03	-2.58	-1.88	2.47
3	false	1.70	2.41	2.01	1.80	1.06	1.80
2	true	-3.22	-1.84	-1.09	-1.80	-1.12	1.81
3	true	-0.17	-0.04	-0.13	0.13	0.11	0.12
LAST.FM							
2	false	-2.44	-1.67	-1.23	-1.45	-2.11	1.78
3	false	-0.09	-0.08	-0.24	-0.11	0.03	0.11
2	true	-1.10	-1.10	-1.39	-1.56	-1.21	1.27
3	true	-0.06	-0.08	-0.05	-0.21	-0.09	0.10
Twitter							
2	false	-0.71	-0.45	-0.41	-0.38	-0.64	0.52
3	false	0.33	0.08	0.21	0.21	0.27	0.22
2	true	-0.40	-0.37	-0.26	-0.51	-0.54	0.42
3	true	-0.05	0.00	0.00	0.00	0.00	0.01

on the datasets indicate that VSM is more effective than existing algorithms, but about two orders of magnitude more efficient than RankedReplace with our SSS2. Without SSS2, the original RankedReplace is much slower.

REFERENCES

[1] D. Kempe, J. Kleinberg, and E. Tardos, "Maximizing the spread of influence through a social network," in *Proc. SIGKDD*, 2003, pp. 137–146.
[2] J. Leskovec, A. Krause, C. Guestrin, C. Faloutsos, J. VanBriesen, and N. Glance, "Cost-effective outbreak detection in networks." in *Proc. SIGKDD*, 2007, pp. 420–429.
[3] C. Aggarwal, A. Khan, and X. Yan, "On flow authority discovery in social networks," in *Proc. SDM*, 2011, pp. 522–533.
[4] M. Kimura, K. Saito, and R. Nakano, "Extracting influential nodes for information diffusion on a social network," in *Proc. AAAI*, 2007, pp. 1371–1376.
[5] W. Chen, Y. Wang, and S.Yang, "Efficient influence maximization in social networks," in *Proc. SIGKDD*, 2009, pp. 199–208.
[6] W. Chen, C. Wang, and Y. Wang, "Scalable influence maximization for prevalent viral marketing in large-scale social networks," in *Proc. SIGKDD*, 2010, pp. 1029–1038.
[7] Y. Wang, G. Cong, G. Song, and K. Xie, "Community-based greedy algorithm for mining top-k influential nodes in mobile social networks," in *Proc. SIGKDD*, 2010, pp. 1039–1048.
[8] A. Goyal, W. Lu, and L. Lakshmanan, "Simpath: An efficient algorithm for influence maximization under the linear threshold model," in *Proc. ICDM*, 2011, pp. 211–220.
[9] Q. Jiang, G. Song, C. Gao, Y. Wang, W. Si, and K. Xie, "Simulated annealing based influence maximization in social networks," in *Twenty-Fifth AAAI Conference on Artificial Intelligence*, 2011, pp. 127–132.

[10] J. Kim, S. Kim, and H. Yu, "Scalable and parallelizable processing of influence maximization for large-scale social networks," in *Proc. ICDE*, 2013, pp. 266–277.
[11] C. Borgs, M. Brautbar, J. Chayes, and B. Lucier, "Maximizing social influence in nearly optimal time," in *Proc. Symp. Discrete Alg.*, 2014, pp. 946–957.
[12] Y. Tang, X. Xiao, and Y. Shi, "Influence maximization: Near-optimal time complexity meets practical efficiency," in *Proc. SIGMOD*, 2014, pp. 75–86.
[13] Y. Tang, Y. Shi, and X. Xiao, "Influence maximization in near-linear time: A martingale approach," in *Proc. SIGMOD*, 2015, pp. 1539–1554.
[14] A. Goyal, W. Lu, and L. Lakshmanan, "CELF++: optimizing the greedy algorithm for influence maximization in social networks," in *Intl. conf. companion on World wide web*, 2011, pp. 47–48.