2011 11th IEEE International Conference on Data Mining

SIMPATH: An Efficient Algorithm for Influence Maximization under the Linear Threshold Model

Amit Goyal, Wei Lu, Laks V. S. Lakshmanan
University of British Columbia
Vancouver, BC
Email: {goyal, welu, laks}@cs.ubc.ca

***Abstract*—There is significant current interest in the problem of *influence maximization*: given a directed social network with influence weights on edges and a number k, find k seed nodes such that activating them leads to the maximum expected number of activated nodes, according to a propagation model. Kempe et al. [1] showed, among other things, that under the Linear Threshold model, the problem is NP-hard, and that a simple greedy algorithm guarantees the best possible approximation factor in PTIME. However, this algorithm suffers from various major performance drawbacks. In this paper, we propose SIMPATH, an efficient and effective algorithm for influence maximization under the linear threshold model that addresses these drawbacks by incorporating several clever optimizations. Through a comprehensive performance study on four real data sets, we show that SIMPATH consistently outperforms the state of the art w.r.t. running time, memory consumption and the quality of the seed set chosen, measured in terms of expected influence spread achieved.**

***Index Terms*—Social Networks; Influence Spread; Linear Threshold Model; Simple Path Enumeration; Viral Marketing.**

I. INTRODUCTION

The study of influence propagation in social networks has attracted a great deal of attention in recent years. One of the fundamental problems in this area is influence maximization whose motivation comes from viral marketing where the idea is to give free or price-discounted samples of a product to selected individuals in the hope that through the word of mouth effect, it can result in a large number of adoptions of the product. In addition, the study of influence propagation has been found to be useful for personalized recommendations [2], selecting informative blogs [3] and finding influential tweeters [4], [5].

In their seminal paper, Kempe et al. [1] formulated influence maximization as a discrete optimization problem: Given a directed social graph with users as nodes, edge weights reflecting influence between users, and a number k (called *budget*), find k users, called the *seed set*, such that by activating them, the expected spread of the influence (just *spread* for brevity) according to a propagation model is maximized. The influence diffusion process unfolds in discrete time steps, as captured by the propagation model.

Two classical propagation models discussed in [1] are the *Linear Threshold* (LT) Model and the *Independent Cascade* (IC) Model, both taken from mathematical sociology. In both models, at any time step, a user is either *active* (an adopter of the product) or *inactive*. In the IC model, when an inactive user becomes active at a time step t, it gets exactly one chance to independently activate its currently inactive neighbors at time $t+1$. In the LT model, the sum of incoming edge weights on any node is assumed to be at most 1 and every user chooses an activation threshold uniformly at random from $[0, 1]$. At any time step, if the sum of incoming influence (edge weights) from the active neighbors of an inactive user exceeds its threshold, it becomes active. In both models, influence propagates until no more users can become active.

Influence maximization under both IC and LT models is **NP**-hard and the spread function is monotone and submodular [1]. A set function $f : 2^U \rightarrow \mathbb{R}^+$ is monotone if $f(S) \leq f(T)$ whenever $S \subseteq T \subseteq U$. It is submodular if $f(S \cup \{w\}) - f(S) \geq f(T \cup \{w\}) - f(T)$ for all $S \subseteq T$ and $w \in U \setminus T$. Intuitively, submodularity says the marginal gain $f(S \cup \{w\}) - f(S)$ from adding a new node w shrinks as the seed set grows.

Exploiting these properties, Kempe et al. [1] presented a simple greedy algorithm which repeatedly picks the node with the maximum marginal gain and adds it to the seed set, until the budget k is reached. However, computing exact marginal gain (or exact expected spread) under both the IC and LT models is #**P**-hard [6], [7]. Hence, it is estimated by running Monte Carlo (MC) simulations. This greedy algorithm, referred to as *simple greedy* henceforth, approximates the problem within a factor of $(1 - 1/e - \epsilon)$ for any $\epsilon > 0$.

Unfortunately, the simple greedy algorithm suffers from the following severe performance drawbacks: (i) The MC simulations that are run sufficiently many times (typically 10,000) to obtain an accurate estimate of spread prove to be very expensive. (ii) The simple greedy algorithm makes $O(nk)$ calls to the spread estimation procedure (MC in this case) where n is the number of nodes in the graph and k is the size of the seed set to be picked.

Considerable work has been done to address the first limitation in the case of the IC model by developing heuristic solutions for seed selection [8], [9], [6]. By contrast, for the LT model, the only similar work to our knowledge is by Chen et al. [7]. They observed that while computing the spread is #**P**-hard in general graphs, it can be computed in linear time on directed acyclic graphs (DAGs). Moreover, they claim that the majority of the influence flows only within a small neighborhood and thus they construct one local DAG (LDAG) per node in the graph and assume that the influence flows to the node only through that LDAG. They experimentally show that

1550-4786/11 $26.00 © 2011 Crown Copyright
DOI 10.1109/ICDM.2011.132

this heuristic is significantly faster than the greedy algorithm and achieves high quality seed set selection, measured in terms of the spread achieved.

While their approach is interesting, the algorithm has the following limitations: First, it relies heavily on finding a high quality LDAG. However, finding an optimal LDAG is **NP**-hard [7] and a greedy heuristic is employed to discover a good LDAG. Recall, their algorithm is already a heuristic and using a greedy LDAG in place of the optimal one may introduce an additional level of loss in quality of the seed set chosen in terms of the spread achieved. Second, it assumes that influence flows to a node via paths within only one LDAG and ignores influence flows via other paths. As a result, if other "ignored" LDAGs together have a high influence flow, the quality of the seed set chosen may suffer. Finally, they store all LDAGs in memory, making the approach memory intensive. From here on, we refer to their algorithm as LDAG. In our experiments, we explore each of the limitations of LDAG mentioned above. It is worth noting that LDAG is the state of the art algorithm for influence maximization under the LT model.

Toward addressing the second limitation of the simple greedy algorithm, Leskovec et al. [3] proposed the CELF optimization that significantly reduces the number of calls made to the spread estimation procedure (MC simulation). Even though this improves the running time of the simple greedy algorithm by up to 700 times [3], it has still been found to be quite slow and definitely not scalable to very large graphs [7], [10]. In particular, the first iteration is very expensive as it makes n calls to the spread estimation procedure. It should be mentioned that LDAG was shown [7] to be significantly faster than even the simple greedy, optimized using CELF.

In this paper, we propose the SIMPATH algorithm for influence maximization under the LT model. SIMPATH incorporates three key novel ways of optimizing the computation and improving the quality of seed selection, where seed set quality is based on its spread of influence: the larger its spread, the higher its quality.

A. Contributions and Roadmap

In Section III, we establish a fundamental result on influence propagation in the LT model which says the spread of a set of nodes can be calculated as the sum of spreads of each node in the set on appropriate induced subgraphs. This paves the way for our SIMPATH algorithm. SIMPATH builds on the CELF optimization that iteratively selects seeds in a lazy forward manner. However, instead of using expensive MC simulations to estimate the spread, we show that under the LT model, the spread can be computed by enumerating the simple paths starting from the seed nodes. It is known that the problem of enumerating simple paths is $\#\mathbf{P}$-hard [11]. However, the majority of the influence flows within a small neighborhood, since probabilities of paths diminish rapidly as they get longer. Thus, the spread can be computed accurately by enumerating paths within a small neighborhood. We propose a parameter η to control the size of the neighborhood that represents a direct trade-off between accuracy of spread estimation and running time. Our spread estimation algorithm which we call SIMPATH-SPREAD is given in Section IV.

We propose two novel optimizations to reduce the number of spread estimation calls made by SIMPATH. The first one, which we call VERTEX COVER OPTIMIZATION, cuts down on the number of calls made in the first iteration (Section V.A), thus addressing a key weakness of the simple greedy algorithm that is *not* addressed by CELF. We show that the spread of a node can be computed directly using the spread of its out-neighbors. Thus, in the first iteration, we first construct a vertex cover of the graph and obtain the spread only for these nodes using the spread estimation procedure. The spread of the rest of the nodes is derived from this. This significantly reduces the running time of the first iteration.

Next, we observe that as the size of the seed set grows in subsequent iterations, the spread estimation process slows down considerably. In Section V.B, we provide another optimization called LOOK AHEAD OPTIMIZATION which addresses this issue and keeps the running time of subsequent iterations small. Specifically, using a parameter ℓ, it picks the top-ℓ most promising seed candidates in the start of an iteration and shares the marginal gain computation of those candidates.

In Section VI, we show through extensive experiments on four real datasets that our SIMPATH algorithm is more efficient, consumes less memory and produces seed sets with larger spread of influence than LDAG, the current state of art. Indeed, among all the settings we tried, the seed selection quality of SIMPATH is quite close to the simple greedy algorithm.

Related work is discussed in Section II, while Section VII summarizes the paper and discusses promising directions for further research.

II. RELATED WORK

Influence maximization from a data mining perspective was first studied by Domingos and Richardson [12]. Later, Kempe et al. [1] modeled the problem as the discrete optimization problem described in the introduction. A major limitation of their approach is the inefficiency.

Significant amount of work has been done to find efficient solutions to influence maximization. Leskovec et al. [3] exploited submodularity and proposed a "lazy-forward" optimization (called CELF) to the simple greedy algorithm. The idea is that the marginal gain of a node in the current iteration cannot be more than its marginal gain in previous iterations and thus the number of spread estimation calls can be reduced significantly. Goyal et al. [13] proposed CELF++, an extension to CELF that further reduces the number of spread estimation calls. The key idea behind CELF++ is that in any iteration, whenever the marginal gain of a node u is computed w.r.t. the current seed set S, the algorithm also computes the marginal gain of u w.r.t. $S \cup \{x\}$ where x is the node that has the maximum marginal gain among all the nodes examined in the current iteration until now. Thus, if x is chosen as the seed node at the end of the current iteration, there is no need to recompute the marginal gain of u in the next iteration. Clearly, the algorithm performs well when it is possible to compute the marginal gain of a

$b_{u,v}$	Influence weight on edge (u, v)
$\Upsilon_{S,v}$	Prob. that v activates if S is the initial seed set
$\sigma(S)$	Expected spread of influence achieved by seed set S
$N^{in}(v)$	Set of in-neighbors of v
$N^{out}(v)$	Set of out-neighbors of v
$\mathsf{P} = \langle v_1, ..., v_m \rangle$	A simple path from v_1 to v_m
$\mathcal{P}(u, v)$	Set of all simple paths from node u to v
η	Pruning threshold (see Section IV)
ℓ	Look-ahead value (see Section V.B)

Fig. 1. Notations used. Terms Υ, σ and $\mathcal{P}(\cdot)$ can have superscripts implying that these terms are evaluated on the corresponding subgraph. E.g., $\sigma^W(S)$ is the influence spread achieved by the seed set S on the subgraph induced by nodes in W.

node u w.r.t. S and $S \cup \{x\}$ simultaneously without much overhead. The authors of [13] report that CELF++ is found to be approximately 35-55% faster than CELF. A natural question that arises is how the SIMPATH algorithm compares with the simple greedy algorithm leveraging CELF++ instead of CELF. We will address this question in Section VI.

A substantial amount of work has been done to get round the MC simulations for seed selection, particularly for the IC model [9], [8], [6]. By contrast, much less work has been done to improve the efficiency of seed selection under the LT model. A notable exception is the scalable LDAG algorithm of Chen et al. [7]. The key differences between LDAG and SIMPATH were discussed in the introduction. An extensive empirical comparison appears in Section VI.

Goyal et al. [10] proposed an alternative approach to influence maximization which, instead of assuming influence probabilities are given as input, directly uses the past available data. The complementary problem of learning influence probabilities from the available data is studied in [14] and [15].

III. PROPERTIES OF LINEAR THRESHOLD MODEL

Consider a digraph $G = (V, E, b)$ with edges $(u, v) \in E$ labeled with influence weights $b_{u,v}$ that represent the influence weight of node u on v. Kempe et al. [1] showed that the LT model is equivalent to the "live-edge" model where a node $v \in V$ picks at most one incoming edge with a probability equal to the edge weight. If an edge is selected, it is considered live. Otherwise, it is considered dead/blocked. This results in a distribution over possible worlds. Then the spread is the expected number of nodes reachable from the initial seed set S over the possible worlds. Let X be any possible world and $\sigma_X(S)$ denote the number of nodes reachable from S via live paths in (the deterministic graph) X where a live path is a path made of live edges. Then the spread of S, by definition, is $\sigma(S) = \sum_X Pr[X] \cdot \sigma_X(S)$ where $\sigma_X(S) = \sum_{v \in V} I(S, v, X)$. Here, $I(S, v, X)$ is the indicator function which is 1 if there is a "live" path in X from any node in S to v and 0 otherwise. Hence,

$$\sigma(S) = \sum_{v \in V} \sum_X Pr[X] \cdot I(S, v, X) = \sum_{v \in V} \Upsilon_{S,v}, \quad (1)$$

where $\Upsilon_{S,v}$ is the probability that v activates if S is the initial seed set. We call it the total influence weight of S on v. We consider only directed paths. Let $\mathsf{P} = \langle v_1, v_2, ...v_m \rangle$ be a path. We write $(v_i, v_j) \in \mathsf{P}$ to indicate that the edge (v_i, v_j) belongs to path P. A simple path is a path in which no nodes are repeated. Henceforth by paths we mean directed simple paths. Clearly, $Pr[\mathsf{P}] = \prod_{(v_i,v_j) \in \mathsf{P}} b_{v_i,v_j}$. By definition of the "live-edge" model, we have[1],

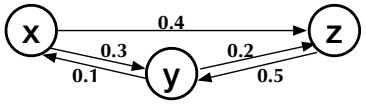


Fig. 2. Example

$$\Upsilon_{u,v} = \sum_{\mathsf{P} \in \mathcal{P}(u,v)} Pr[\mathsf{P}] \quad (2)$$

where $Pr[\mathsf{P}]$ is the probability of a path P being live and $\mathcal{P}(u, v)$ is the set of all paths from node u to v. As an example, consider the influence graph shown in Fig. 2. The influence of node x on z is $\Upsilon_{x,z} = 0.3 \cdot 0.2 + 0.4 = 0.46$.

With superscripts, we denote the corresponding subgraph considered. For example, $\Upsilon^{V-S}_{u,v}$ denotes the total influence weight of u on v in the subgraph induced by $V - S$. For simplicity, we write $V - S$ for $V \setminus S$ and $V - S + u$ to denote $((V \setminus S) \cup \{u\})$. When there is no superscript, the whole graph G is considered. Fig. 1 summarizes the notations used in the paper. SIMPATH builds on the following key result.

Theorem 1. *In the LT model, the spread of a set S is the sum of the spread of each node $u \in S$ on subgraphs induced by $V - S + u$. That is,*

$$\sigma(S) = \sum_{u \in S} \sigma^{V-S+u}(u)$$

We first illustrate the theorem using the example shown in Fig. 2. Let $S = \{x, y\}$, then according to the claim, $\sigma(S) = \sigma^{V-y}(x) + \sigma^{V-x}(y) = 1 + 0.4 + 1 + 0.2 = 2.6$ which is the correct spread of the set $\{x, y\}$.

Proof of theorem 1: We claim that $\Upsilon_{S,v} = \sum_{u \in S} \Upsilon^{V-S+u}_{u,v}$. The theorem follows from this, upon taking the sum over all nodes $v \in V$ (see Eq. (1)). We next prove the claim. Intuitively, it says that the influence weight of a set S on node v is the sum of influence weights of each node $u \in S$ on v on subgraphs induced by $V - S + u$. We prove the claim by induction on path lengths. Specifically, let $\Upsilon_{S,v}(t)$ denote the total influence weight of S on v via all paths of length t. Clearly, $\Upsilon_{S,v} = \sum_t \Upsilon_{S,v}(t)$. To prove the claim above, it suffices to show $\Upsilon_{S,v}(t) = \sum_{u \in S} \Upsilon^{V-S+u}_{u,v}(t), \forall t$.

Base Case: When the path length t is 0, we must have $v \in S$. Then the claim is trivial as $\Upsilon_{S,v}(0) = \Upsilon^{V-S+v}_{v,v}(0) = 1$ and $\Upsilon^{V-S+w}_{w,v}(0) = 0$, $\forall w \neq v$.

Induction Step: Assume the claim for t. Consider paths from S to v of length $t + 1$. A path of length $t + 1$ from S to v must contain a sub-path of length t from S to some in-neighbor w of v. Clearly, this sub-path cannot pass through v (since the path is simple). Thus, we have:

$$\Upsilon_{S,v}(t + 1) = \sum_{w \in N^{in}(v)} \Upsilon^{V-v}_{S,w}(t) \cdot b_{w,v} \quad (3)$$

[1]This definition is also used in [7].

where $N^{in}(v)$ denotes the set of in-neighbors of v. By expanding the term $\Upsilon^{V-v}_{S,w}(t)$ using induction hypothesis in Eq. 3, we have:

$$\Upsilon_{S,v}(t+1) = \sum_{w \in N^{in}(v)} \sum_{u \in S} \Upsilon^{V-v-S+u}_{u,w}(t) \cdot b_{w,v}$$

By switching the summations, we have $\Upsilon_{S,v}(t+1) = \sum_{u \in S} \sum_{w \in N^{in}(v)} \Upsilon^{V-v-S+u}_{u,w}(t) \cdot b_{w,v} = \sum_{u \in S} \Upsilon^{V-S+u}_{u,v}(t+1)$. This was to be shown. □

IV. Spread Estimation using Simpath-Spread

In this section, we develop Simpath-Spread, the spread estimation algorithm used by Simpath. Eq. 1 and 2 show that the spread of a node can be computed by summing the weights (i.e., probabilities) of all simple paths originating from it. For instance, consider the example shown in Fig. 2. The spread of the node x is $\Upsilon_{x,x} + \Upsilon_{x,y} + \Upsilon_{x,z} = 1 + (0.3 + 0.4 \cdot 0.5) + (0.4 + 0.3 \cdot 0.2) = 1.96$. Moreover, in Theorem 1, we show that the spread of a seed set S can be obtained as the sum of spreads of individual nodes u in S in subgraphs induced by $V - S + u$. Hence, the spread of a seed set can be computed by enumerating paths from nodes in S, although on different subgraphs. Not surprisingly, the problem of enumerating all simple paths is $\#\mathbf{P}$-hard [11]. However, we are interested only in path weight (that is, probability of a path being live), which decreases rapidly as the length of the path increases. Thus, the majority of the influence can be captured by exploring the paths within a small neighborhood, where the size of the neighborhood can be controlled by the error we can tolerate. This is the basis of our algorithm. We adapt the classical Backtrack algorithm [16], [17] to enumerate all simple paths as follows. We maintain a stack Q containing the nodes in the current path. Initially, Q contains only the current seed node. The algorithm calls the subroutine Forward which takes the $last$ element in Q and adds the nodes to it in a depth-first fashion until no more nodes can be added. Whenever a new node is added, the subroutine Forward ensures that it does not create a cycle and the path segment has not been explored before. If no more nodes can be added, the algorithm backtracks and the last node is removed from the stack Q. The subroutine Forward is called again to get a new path. The process is continued until all the paths are enumerated.

As argued above, the majority of the influence to a node flows in from a small neighborhood and can be captured by enumerating paths within that neighborhood. Thus, we prune a path once its weight reaches below a given *pruning threshold* η. Intuitively, η is the error we can tolerate. It represents the trade-off between accuracy of the estimated spread and efficiency of the algorithm (running time). If η is high, we can tolerate larger error and prune paths early, at the price of accuracy of our spread estimation. On the other hand, if it is low, we will explore more paths, leading to high accuracy but with increased running time. As a special case, if $\eta = 0$, then the spread obtained would be exact. We study the effect of η on the quality of seed set and running time in Section VI.

The spread estimation procedure Simpath-Spread is described in Algorithms 1, 2, 3. Given a set S and a pruning threshold η, Algorithm 1 exploits Theorem 1 to compute the spread of S. For each node $u \in S$, it calls Backtrack (line 3) which in turn takes a node u, the pruning threshold η, a given set of nodes $W \subseteq V$ and estimates $\sigma^W(u)$. In addition, the algorithm also takes a set $U \subseteq V$ which is needed to implement optimizations proposed in Section V. To give a preview, in addition to computing $\sigma^W(u)$, the algorithm also computes $\sigma^{W-v}(u)$ for all nodes $v \in U$. We don't need it right now and hence assume U to be an empty set. We will revisit to it in Section V.

Algorithm 1 Simpath-Spread

Input: S, η, U
1: $\sigma(S) = 0$.
2: **for** each $u \in S$ **do**
3: $\quad \sigma(S) \leftarrow \sigma(S) +$ Backtrack$(u, \eta, V - S + u, U)$.
4: return $\sigma(S)$.

Algorithm 2 Backtrack

Input: u, η, W, U
1: $Q \leftarrow \{u\}$; $spd \leftarrow 1$; $pp \leftarrow 1$; $D \leftarrow null$.
2: **while** $Q \neq \emptyset$ **do**
3: $\quad [Q, D, spd, pp] \leftarrow$ Forward$(Q, D, spd, pp, \eta, W, U)$.
4: $\quad u \leftarrow Q.last()$; $Q \leftarrow Q - u$; $delete\ D[u]$; $v \leftarrow Q.last()$.
5: $\quad pp \leftarrow pp / b_{v,u}$.
6: return spd.

The subroutine Backtrack (Algorithm 2) enumerates all simple paths starting from u. It uses a stack Q to maintain the current nodes on the path, pp to maintain the weight of the current path and spd to track the spread of node u in the subgraph induced by W. $D[x]$ maintains the out-neighbors of x that have been seen so far. Using this, we can keep track of the explored paths from a node efficiently. The variables are initialized in line 1. The subroutine Forward is called repeatedly which gives a new pruned path that is undiscovered until now (line 3). Then, the last node is removed from the path and the variable pp is set accordingly. Since u no longer exists in the current path, $D[u]$ is deleted (lines 4-5).

Algorithm 3 Forward

Input: $Q, D, spd, pp, \eta, W, U$
1: $x = Q.last()$.
2: **while** $\exists y \in N^{out}(x)$: $y \notin Q, y \notin D[x], y \in W$ **do**
3: $\quad$ **if** $pp \cdot b_{x,y} < \eta$ **then**
4: $\quad\quad D[x].insert(y)$.
5: $\quad$ **else**
6: $\quad\quad Q.add(y)$.
7: $\quad\quad pp \leftarrow pp \cdot b_{x,y}$; $spd \leftarrow spd + pp$.
8: $\quad\quad D[x].insert(y)$; $x \leftarrow Q.last()$.
9: $\quad\quad$ **for** each $v \in U$ such that $v \notin Q$ **do**
10: $\quad\quad\quad spd^{W-v} \leftarrow spd^{W-v} + pp$.
11: return $[Q, D, spd, pp]$.

Next, we explain the Forward algorithm (Algorithm 3). It takes the $last$ element as x on the path (line 1) and extends it as much as possible in a depth-first fashion. A new node y added to the path must be an out-neighbor of x which is not

yet explored ($y \notin D[x]$), should not create a cycle ($y \notin Q$) and should be in the graph considered ($y \in W$) (line 2). Lines 3-4 prune the path if its weight becomes less than the pruning threshold η and the node y is added to the seen neighbors of x. If a node y is added to the path, the weight of the path pp is updated accordingly and the spread is updated (lines 5-7). Finally, the node y is added to the seen neighbors of x (line 8). The process is continued until no new nodes can be added to the path. Lines 9-10 are used only for the optimizations proposed in section V and can be ignored for now.

V. Assembling and Optimizing Simpath

In the previous section, we proposed Simpath-Spread to compute the spread of a seed set. Now, we can plug it in the greedy algorithm (optimized with Celf) to obtain an algorithm for influence maximization. This forms the core of the Simpath algorithm. We propose two optimizations to further improve Simpath which make it highly efficient.

A. Vertex Cover Optimization

Even with the Celf optimization, in the first iteration, the spread estimation needs to be done for all nodes in the graph, resulting in $|V|$ calls to the Simpath-Spread subroutine, where V is the set of nodes in the input social graph. This makes the selection process of the first seed particularly slow. In this section, we propose the Vertex Cover Optimization that reduces the number of calls to Simpath-Spread significantly. It leverages the Theorem 2, proved below, which says for any node v, if we have the spread values of all its out-neighbors in the subgraph induced by $V - v$, then we can directly compute the spread of v without making any call to Simpath-Spread. Using this insight, in the first iteration, we first split the set V of all nodes into two disjoint subsets C and $V - C$ such that C is a vertex cover of the underlying undirected graph G' of the directed social graph G.[2] Then, we *simultaneously* compute for each node $u \in C$, the spread of u in G and in the subgraphs induced by $V - v$, for each in-neighbor v of u that is present in $V - C$. In Algorithm 1, this can be achieved by setting $U = (V - C) \cap N^{in}(u)$ (see Algorithm 4 for details). The Backtrack subroutine remains unchanged and in the Forward subroutine, in addition to updating spd (that is, $\sigma(u)$), lines 9-10 update the variable spd^{V-v} (which represents $\sigma^{V-v}(u)$) for all $v \in (V - C) \cap N^{in}(u)$ properly. Note that in the first iteration, $W = V$. Clearly, v must not be in the current path ($v \notin Q$). Once this is done, we can use Theorem 2 to compute the spread of every node in $V - C$ directly.

Theorem 2. *In the LT model, the spread of a node linearly depends on the spread of its out-neighbors as follows.*

$$\sigma(v) = 1 + \sum_{u \in N^{out}(v)} b_{v,u} \cdot \sigma^{V-v}(u)$$

[2]The underlying undirected graph G' of a digraph G is obtained by ignoring the directions of edges and deleting any duplicate edges. A vertex cover of a graph is a set of vertices such that each edge of the graph is incident to at least one vertex in the set.

As an example, consider the graph shown in Fig. 2. Spread of node x can be computed as $\sigma(x) = 1 + b_{x,y} \cdot \sigma^{V-x}(y) + b_{x,z} \cdot \sigma^{V-x}(z) = 1 + 0.3 \cdot (1 + 0.2) + 0.4 \cdot (1 + 0.5) = 1.96$.

Proof of theorem 2: Consider $\Upsilon_{v,y}$, the influence of v on an arbitrary node y. Recall that it is the sum of probabilities of simple paths from v to y. If $y \equiv v$, $\Upsilon_{v,y} = 1$. For $y \in V - v$, any path from v to y must pass through some out-neighbor u of v. Let $\mathsf{P} = \langle v, u, ..., y \rangle$ be such a path. Clearly, $Pr[\mathsf{P}] = b_{v,u} \cdot Pr[\mathsf{P}']$, where $\mathsf{P}' = \langle u, ..., y \rangle$. From this, we have

$$\Upsilon_{v,y} = \sum_{u \in N^{out}(v)} \sum_{\mathsf{P}' \in \mathcal{P}(u,y)} b_{v,u} \cdot Pr[\mathsf{P}']$$

where $N^{out}(v)$ is the set of out-neighbors of v. Since path P does not have any cycles, P' must not contain v and thus, we can rewrite the above equation as

$$\begin{aligned} \Upsilon_{v,y} &= \sum_{u \in N^{out}(v)} \sum_{\mathsf{P}' \in \mathcal{P}^{V-v}(u,y)} b_{v,u} \cdot Pr[\mathsf{P}'] \\ &= \sum_{u \in N^{out}(v)} b_{v,u} \cdot \Upsilon^{V-v}_{u,y} \end{aligned}$$

where $\mathcal{P}^{V-v}(u, y)$ is the set of (simple) paths from u to y in the subgraph induced by $V - v$. Taking the sum over all $y \in V$, we get the theorem. □

Finding Vertex Cover. Minimizing the number of calls to Simpath-Spread in the first iteration making use of Theorem 2, requires finding a minimum vertex cover (that is, a vertex cover of minimum size). However, this is an **NP**-complete problem. Fortunately, this problem is approximable and several PTIME approximation algorithms are known (e.g., see [18] and [19]). However, it is important to keep in mind that the overhead in finding an approximate minimum vertex cover should not overshadow the benefits coming from it, by means of saved spread estimation calls.

In our experiments, we observed that when using the approximation algorithms, the amount of savings achieved is dominated by the running time of the algorithms, so we settle for a heuristic solution. We use a maximum degree heuristic that repeatedly picks a node of maximum degree and adds it into the vertex cover C whenever at least one of its incident edges is not yet covered. It does this until all edges in the (underlying undirected) graph G' are covered, i.e., incident to at least one vertex in C. It is worth pointing out that the approximation quality of the vertex cover does not affect the output quality (i.e., seed selection) of our algorithm. We employ it only to improve the efficiency of our algorithm. Thus, we recommend a fast heuristic such as the maximum degree heuristic for Vertex Cover Optimization.

B. Look Ahead Optimization

As the seed set grows, the time spent on the spread estimation process increases. An important reason is that the number of calls to the Backtrack subroutine in an iteration increases hurting the performance of Simpath. For instance, if S_i is the seed set after the i-th iteration ($|S_i| = i$), then for each seed candidate x, we need to compute $\sigma(S_i + x)$ to

obtain its marginal gain $\sigma(S_i + x) - \sigma(S_i)$, where $\sigma(S_i)$ is known from the previous iteration.

It takes $i + 1$ calls to BACKTRACK to compute $\sigma(S_i + x)$, resulting in $j \cdot (i+1)$ calls in total, where j is the index of the seed node to be picked in the CELF queue. In other words, j is the number of nodes the algorithm ends up examining before picking the seed in the iteration. In each BACKTRACK call, we compute the spread of an individual node $v \in S_i + x$ on the subgraph induced by $V - S + v$. Except for the node x, the work for all other nodes in S_i is largely repeated, although on slightly different graphs. In this section, we propose LOOK AHEAD OPTIMIZATION, which intelligently avoids this repetition and as a result, significantly reduces the number of BACKTRACK calls. We manipulate Theorem 1 as follows.

$$\sigma(S_i + x) = \sum_{u \in S_i + x} \sigma^{V - S_i - x + u}(u) \quad (4)$$

$$= \sigma^{V - S_i}(x) + \sum_{u \in S_i} \sigma^{V - S_i - x + u}(u) \quad (5)$$

$$= \sigma^{V - S_i}(x) + \sigma^{V - x}(S_i) \quad (6)$$

Thus, if we have a set U of the most promising seed candidates to use from an iteration, say $i+1$, we can compute $\sigma^{V-x}(S_i)$ for all $x \in U$ in the start of the iteration $i+1$. Then, for each x, we call BACKTRACK to compute $\sigma^{V - S_i}(x)$.

Let ℓ be the *look-ahead value*, that is, the number of most promising seed candidates in an iteration. At the beginning of the $i + 1$-th iteration, we select the top-ℓ nodes as one batch from the CELF queue as U, and estimate $\sigma(S_i + x)$ based on Eq. (7). If the algorithm fails to find a new seed from the current batch, we take the next top-ℓ nodes from the CELF queue as U. The process is repeated until a seed is selected (see Algorithm 4). Consequently, the optimization reduces the number of BACKTRACK calls to $i \cdot \lceil j/\ell \rceil + j$ where $\lceil j/l \rceil$ is the number of batches we end up processing. The look-ahead value ℓ represents the trade-off between the number of batches we process and the overhead of computing $\sigma^{V-x}(S_i)$ for all $x \in U$. A large value of ℓ would ensure that the seed node is picked in the first batch itself ($\lceil j/\ell \rceil = 1$). However, that would be inefficient as the overhead of computing $\sigma^{V-x}(S_i)$ for all $x \in U$ for a large U is high. On the other hand, a small value of ℓ would result in too many batches that we end up processing. As a special case, $\ell = 1$ is equivalent to the optimization not being applied. We study the effect of ℓ on SIMPATH's efficiency in Section VI.

C. SIMPATH: *Putting the pieces together*

Algorithm 4 shows the complete SIMPATH algorithm. Lines 1-8 implement the VERTEX COVER OPTIMIZATION. First, the algorithm finds a vertex cover C (line 1), then for every node $u \in C$, its spread is computed on required subgraphs needed for the optimization (lines 2-4). This is done in a single call to SIMPATH-SPREAD. Next, for the nodes that are not in the vertex cover, the spread is computed using Theorem 2 (lines 6-7). The CELF queue (sorted in the decreasing order of marginal gains) is built accordingly (lines 5 and 8).

Lines 9-20 select the seed set in a lazy forward (CELF) fashion using LOOK AHEAD OPTIMIZATION. The spread of the seed set S is maintained using the variable spd. At a time, we take a batch of top-ℓ nodes, call it U, from the CELF queue (line 11). In a single call to SIMPATH-SPREAD, the spread of S is computed on required subgraphs needed for the optimization (lines 12). For a node $x \in U$, if it is processed before in the same iteration, then it is added in the seed set as it implies that x has the maximum marginal gain w.r.t. S (lines 13-16). Recall that the CELF queue is maintained in decreasing order of the marginal gains and thus, no other node can have a larger marginal gain. If x is not seen before, we need to recompute its marginal gain, which is done in lines 17-19. The subroutine BACKTRACK is called to compute $\sigma^{V-S}(x)$ and Eq. 6 is applied to get the spread of the set $S + x$. The CELF queue is updated accordingly (line 20).

Algorithm 4 SIMPATH

Input: $G = (V, E, b), \eta, \ell$

1: Find the vertex cover of input graph G. Call it C.
2: **for** each $u \in C$ **do**
3: $U \leftarrow (V - C) \cap N^{in}(u)$.
4: Compute $\sigma(u)$ and $\sigma^{V-v}(u)$, $\forall v \in U$ in a single call to Algorithm 1: SIMPATH-SPREAD(u, η, U).
5: Add u to CELF queue.
6: **for** each $v \in V - C$ **do**
7: Compute $\sigma(v)$ using Theorem 2.
8: Add v to CELF queue.
9: $S \leftarrow \emptyset$. $spd \leftarrow 0$.
10: **while** $|S| < k$ **do**
11: $U \leftarrow$ top-ℓ nodes in CELF queue.
12: Compute $\sigma^{V-x}(S)$, $\forall x \in U$, in a single call to Algorithm 1: SIMPATH-SPREAD(S, η, U).
13: **for** each $x \in U$ **do**
14: **if** x is previously examined in the current iteration **then**
15: $S \leftarrow S + x$; Update spd.
16: Remove x from CELF queue. Break out of the loop.
17: Call BACKTRACK$(x, \eta, V - S, \emptyset)$ to compute $\sigma^{V-S}(x)$.
18: Compute $\sigma(S + x)$ using Eq. 6.
19: Compute marginal gain of u as $\sigma(S + x) - spd$.
20: Re-insert u in CELF queue such that its order is maintained.
21: return S.

VI. EXPERIMENTS

We conduct extensive experiments on four real-world datasets ranging from small to large scale, to evaluate the performance of SIMPATH and compare it with well-known influence maximization algorithms on various aspects such as efficiency, memory consumption and quality of the seed set.

The code is written in C++ using the Standard Template Library (STL), and all the experiments are run on a Linux (OpenSuSE 11.3) machine with a 2.93GHz Intel Xeon CPU and 64GB memory. The code is available for download at http://people.cs.ubc.ca/~welu/downloads.html. It includes the implementation of all the algorithms listed in this section.

A. Datasets

We use four real-world graph data sets whose statistics are summarized in Table I. To obtain influence weights on edges,

TABLE I
STATISTICS OF DATASETS.

Dataset	NetHEPT	Last.fm	Flixster	DBLP
#Nodes	15K	61K	99K	914K
#Directed Edges	62K	584K	978K	6.6M
Avg. Degree	4.1	9.6	9.9	7.2
Maximum Out-degree	64	1073	428	950
#Connected Components	1781	370	1217	41.5K
Avg. Component Size	8.55	164	81.4	22
Largest Component Size	6794	59.7K	96.5K	789K

we adopt the methods in [15] and [1] and learn them. More precisely, we assign the weight on an edge (u, v) as $b_{u,v} = A(u, v)/N(v)$ where $A(u, v)$ is the number of actions both u and v perform, and $N(v)$ is a normalization factor to ensure that the sum of incoming weights to v is 1, i.e., $N(v) = \sum_{u \in N^{in}(v)} A(u, v)$. The details are as follows:

NetHEPT. Frequently used in previous works [1], [8], [6], [7] in this area, NetHEPT is a collaboration network taken from the "High Energy Physics (Theory)" section of arXiv[3] with nodes representing authors and edges representing co-authorship. Here, an action is a user publishing a paper, and $A(u, v)$ is the number of papers co-authored by u and v. The numbers of nodes and directed edges in the graph are 15K and 62K respectively[4].

Last.fm. Taken from a popular music and online radio website with social networking features[5], this network data contains 100K users, 3.15M edges, 66K groups, and 1M subscriptions [20]. We consider "joining a group" as an action, and thus, $A(u, v)$ is the number of groups both u and v have joined. After excluding users who did not join any group, we are left with 60K users with 584K directed edges.

Flixster. Taken from a social movie site[6] allowing users to share movie ratings, the raw data contains 1M users with 28M edges. There are 8.2M ratings distributed among 49K distinct movies [21][7]. Here, an action is a user rating a movie and thus, $A(u, v)$ is the number of movies rated by both u and v. Users who do not rate any movies are excluded. The resulting graph contains 99K users and 978K directed edges.

DBLP. A collaboration network from the DBLP Computer Science Bibliography, much larger compared to NetHEPT. We obtain all the latest bibliographic records (snapshot on June 7, 2011) available from the website. Similar to NetHEPT, $A(u, v)$ is the number of manuscripts both u and v have co-authored. The graph consists of 914K nodes and 6.6M edges.

B. Algorithms Compared

Besides LDAG, which is the state of art, we include several other algorithms and some generic model independent heuristics (listed below) in our empirical evaluation.

[3]http://arxiv.org/

[4]The dataset is publicly available at http://research.microsoft.com/en-us/people/weic/graphdata.zip

[5]http://www.last.fm/

[6]http://www.flixster.com/

[7]The dataset is available at http://www.cs.sfu.ca/∼sja25/personal/datasets

HIGH-DEGREE. A heuristic based on the notion of "degree centrality", considering high-degree nodes as influential [1]. The seeds are the nodes with the k highest out-degrees.

PAGERANK. A link analysis algorithm to rank the importance of pages in a Web graph [22]. We implement the power method with a damping factor of 0.85 and pick the k highest-ranked nodes as seeds. We stop when the score vectors from two consecutive iterations differ by at most 10^{-6} as per L_1-norm.

MC-CELF. The greedy algorithm with CELF optimization [3]. Following the literature, we run $10,000$ Monte Carlo (MC) simulations to estimate the spread of a seed set.

LDAG. The algorithm proposed in [7]. As recommended by the authors, we use the influence parameter $\theta = 1/320$ to control the size of the local DAG constructed for each node.

SIMPATH. Our proposed algorithm described in Sections IV and V (Algorithm 4). Unless otherwise noted, the pruning threshold η used in SIMPATH is set to 10^{-3} and the look-ahead value ℓ is set to 4. These values were chosen based on empirically observing the performance.

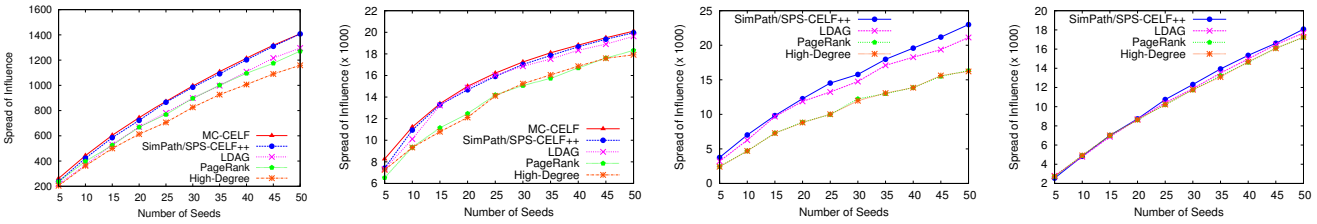
SPS-CELF++. As mentioned in Section II, an improvement to CELF, called CELF++, was recently proposed in [13]. Thus, a natural question is whether we can obtain an algorithm better than SIMPATH by using CELF++ in place of CELF. We investigate this question next. More concretely, define SPS-CELF++ to be the greedy algorithm with CELF++ (instead of CELF) used to select seeds and SIMPATH-SPREAD (instead of MC) used for spread estimation. SPS-CELF++ also leverages VERTEX COVER OPTIMIZATION for the first iteration (Section V.A). As recommended by [13], we apply CELF++ starting from the second iteration. For fairness of comparison, we set η to the same value 10^{-3} that we used in SIMPATH.

As described in Section II, CELF++ works well when it is possible to compute marginal gain of a node u w.r.t. the current seed set S_i and $S_i + x$ simultaneously where x is the node having the maximum marginal gain among all nodes examined in the current iteration. Next, we show how to do this within SIMPATH architecture. From Theorem 1, using an algebraic manipulation similar to that in Eq. (6), we can obtain

$$\sigma(S_i + x + u) = \sigma^{V-x}(S_i + u) + \sigma^{V-S_i-u}(x)$$

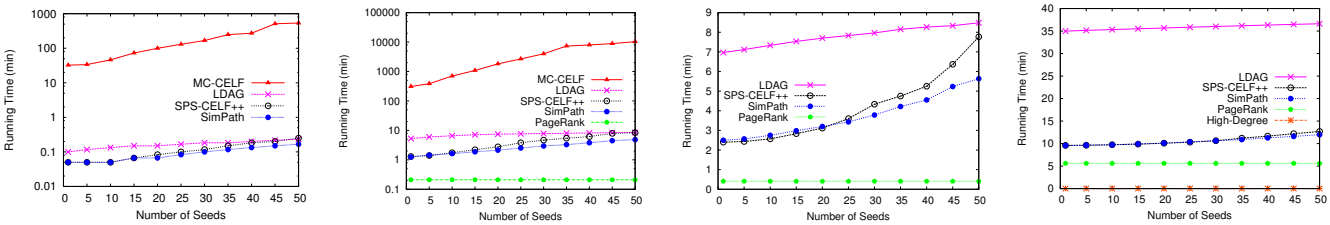
Thus, in the i-th iteration, while computing $\sigma(S_i+u)$, we also compute $\sigma^{V-x}(S_i + u)$ by setting $U = \{x\}$ in Algorithm 1. Finally, if x is selected as seed, we just compute $\sigma^{V-S_i-u}(x)$ to obtain $\sigma(S_i + x + u)$.

However, CELF++ cannot be applied in conjunction with the LOOK AHEAD OPTIMIZATION as CELF++ requires to compute $\sigma^{V-x}(S_i + u)$, where x is the previous best node (having maximum marginal gain), but when we compute the spread of S_i on different subgraphs at the beginning of an iteration, this x is unknown. As a result, in SPS-CELF++, we can incorporate VERTEX COVER OPTIMIZATION but not LOOK AHEAD OPTIMIZATION. We include SPS-CELF++ in our evaluation. Since it is more efficient than simple greedy using CELF++, we do not evaluate the latter separately.



(a) NetHEPT (b) Last.fm (c) Flixster (d) DBLP

Fig. 3. Influence spread achieved by various algorithms.



(a) NetHEPT (b) Last.fm (c) Flixster (d) DBLP

Fig. 4. Efficiency: Comparisons of running time. Running times below 0.01 minutes are not shown.

TABLE II
SIMPATH'S IMPROVEMENT OVER LDAG

Dataset	Improvement in		
	Spread	Running Time	Memory
NetHEPT	8.7%	21.7%	62.9%
Last.fm	1.7%	42.9%	86.5%
Flixster	8.9%	33.6%	87.5%
DBLP	2.3%	67.2%	87.1%

C. Experimental Results

We compare the performance of the various algorithms on the following metrics: quality of seed sets, running time, memory usage, and scalability. We also study the effectiveness of our two optimizations: the VERTEX COVER OPTIMIZATION and LOOK AHEAD OPTIMIZATION.

Table II gives an overall summary of the relative performance of LDAG and SIMPATH observed in our experiments. It shows the percentage improvement registered by SIMPATH over LDAG on quality of seed set (measured in spread), running time, and memory consumption. On all counts, it can be seen that SIMPATH outperforms LDAG, the state of the art. We will drill down into the details in the next subsections.

Due to MC-CELF's lack of efficiency and scalability, its results are only reported for NetHEPT and Last.fm, the two datasets on which it can finish in a reasonable amount of time.

On Quality of Seed Sets. The quality of the seed sets obtained from different algorithms is evaluated based on the expected spread of influence. Higher the spread, better the quality. For fair comparisons, we run MC simulations $10,000$ times to obtain the "ground truth" spread of the seeds sets obtained by all algorithms. Fig. 3 shows the spread achieved against the size of the seed set. We show the spread of the seed sets chosen by SIMPATH and SPS-CELF++ as one plot because they both use Algorithm 1 (SIMPATH-SPREAD) to estimate the spread and hence, the seed sets produced by both the algorithms are exactly the same.

The seed sets output by SIMPATH are quite competitive in quality with those of MC-CELF. For instance, both have the spread 1408 on NetHEPT, while on Last.fm, SIMPATH is only 0.7% lower than MC-CELF in spread achieved. Note that MC-CELF is too slow to complete on Flixster and DBLP.

Except for MC-CELF, on all datasets, SIMPATH is able to produce seed sets of the highest quality. The biggest differences between SIMPATH and LDAG are seen on Flixster and NetHEPT, where the seed set of SIMPATH has 8.9% and 8.7% larger spread than that of LDAG, respectively. PAGERANK and HIGH-DEGREE have similar performances, both being worse than SIMPATH, e.g., PAGERANK is 9.7%, 8.2%, 27.5%, and 4.7% lower than SIMPATH in spread achieved on NetHEPT, Last.fm, Flixster, DBLP, respectively.

On Efficiency and Memory Consumption. We evaluate the efficiency and scalability on two aspects: running time and memory usage. Fig. 4 reports the time taken by various algorithms against the size of the seed set. Note that the plots for NetHEPT and Last.fm have a *logarithmic scale* on the y-axis. MC-CELF takes 9 hours to finish on NetHEPT and 7 days on Last.fm while it fails to complete in a reasonable amount of time on Flixster and DBLP.

HIGH-DEGREE and PAGERANK finish almost instantly in most cases[8]. Except for them, SIMPATH is the fastest among other algorithms on all datasets. It is not only highly efficient on moderate datasets such as NetHEPT (10 seconds), but also scalable to large data, finishing in 5.6 min. on Flixster and in 12.0 min. on DBLP to select 50 seeds. LDAG is approximately twice and 3 times slower on Flixster and DBLP, respectively.

On the other hand, SPS-CELF++ is less scalable than SIMPATH on large datasets. This is because that SPS-CELF++ is not compatible with the LOOK AHEAD OPTIMIZATION, so it makes a much larger number of calls to BACKTRACK (Algorithm 2) to compute the marginal gains of seed candidates, especially when $|S|$ becomes large. Recall that

[8]Running time below 0.01 min. are not shown on the plots.

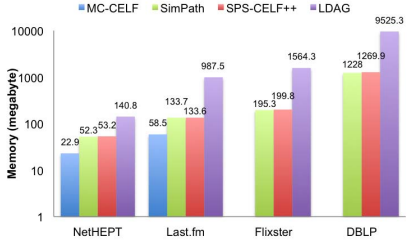


Fig. 5. Comparison of memory usages by MC-CELF, SIMPATH, and LDAG (logarithmic scale).

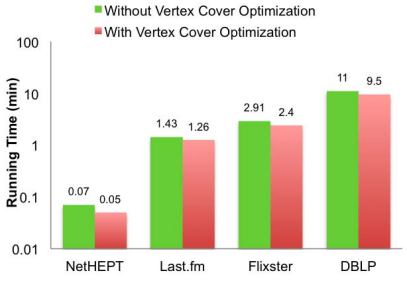


Fig. 6. Effects of Vertex Cover Optimization on the running time of SIMPATH's 1st iteration (logarithmic scale).

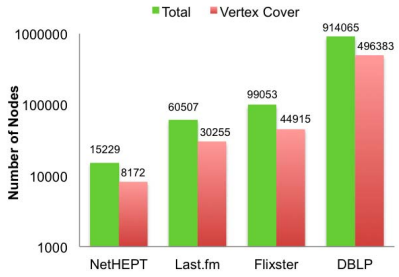
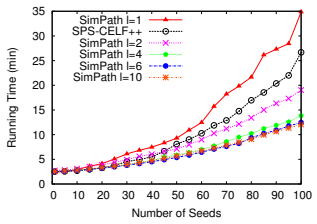
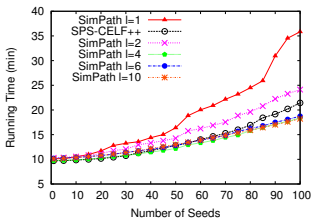


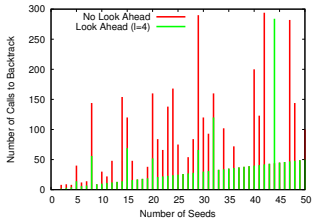
Fig. 7. Size of Vertex Covers for four datasets (logarithmic scale).



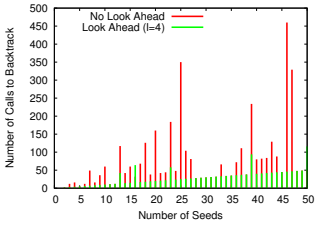
(a) Flixster



(b) DBLP



(c) Flixster



(d) DBLP

Fig. 8. LOOK AHEAD OPTIMIZATION on Flixster and DBLP: (a) and (b) show the effects of different look-ahead values ℓ on running time; (c) and (d) show the number of BACKTRACK calls reduced by LOOK AHEAD OPTIMIZATION.

the VERTEX COVER OPTIMIZATION is indeed applicable to SPS-CELF++, indicating that LOOK AHEAD OPTIMIZATION makes a substantial difference in running time between SIMPATH and SPS-CELF++ (more on this later).

Next, we compare the memory consumption of MC-CELF, SIMPATH, SPS-CELF++, and LDAG in Fig. 5 (*logarithmic scale*). LDAG consumes the most memory among the three algorithms, as it maintains a local DAG for every single node in the graph. Relatively, the memory usage by SIMPATH is much less. For instance, on the largest dataset DBLP, LDAG consumes 9.3GB while SIMPATH uses only 1.2GB. Indeed, Table II shows that SIMPATH can save up to 87% of the memory footprint that is used by LDAG.

On Vertex Cover Optimization. Recall that VERTEX COVER OPTIMIZATION aims to reduce the number of spread estimation calls in the first iteration, thus addressing a limitation of CELF. We show its effectiveness in Fig. 6 and 7 (both have *logarithmic scale* on Y-axis). In Fig. 6, we compare the running time of the first iteration of SIMPATH with and without the VERTEX COVER OPTIMIZATION. With the optimization turned on, the first iteration is 28.6%, 11.9%, 17.5%, and 13.6% faster on NetHEPT, Last.fm, Flixster, and DBLP, respectively, than without the optimization. This is because with this optimization, the number of calls to SIMPATH-SPREAD is only for a fraction of nodes in the graph (i.e., the vertex cover).

Next, in Fig. 7, we report the size of the vertex cover found by the maximum degree heuristic for each dataset. On an average, VERTEX COVER OPTIMIZATION reduces the number of calls to SIMPATH-SPREAD by approximately 50%.

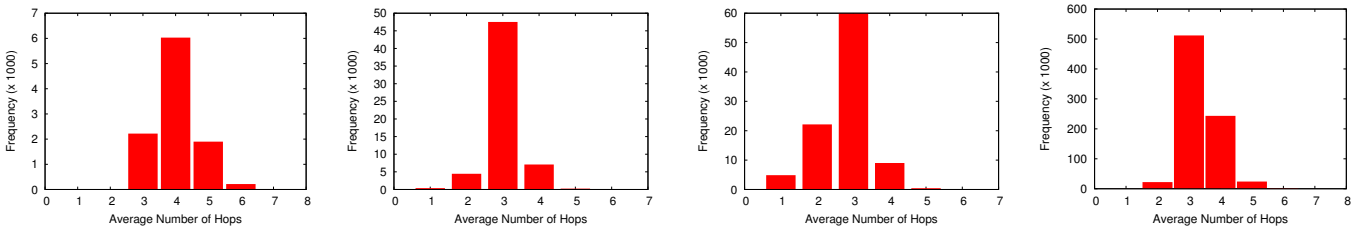
On Look Ahead Optimization. We show the effectiveness of LOOK AHEAD OPTIMIZATION on Flixster and DBLP (results are similar on other two datasets). First, we choose five values to compare running time: 1 (equivalent to no look ahead), 2, 4, 6, and 10 (Fig. 8 (a)-(b)). We also include SPS-CELF++ (which is not compatible with this optimization) as baseline. In both cases we select 100 seeds. On both Flixster and DBLP, SIMPATH with small ℓ (1 or 2) performs poorly and is slower than SPS-CELF++. For DBLP, $\ell = 4$ is the best choice, while $\ell = 6$ is the best for Flixster. When ℓ increases to 10, the running time goes up slightly on both datasets, suggesting that to take a batch of 10 candidates introduces more overhead than the benefit brought by the optimization (see Section V.B).

In Fig. 8 (c)-(d), we show the number of BACKTRACK calls reduced by using the LOOK AHEAD OPTIMIZATION. On both datasets, without look-ahead, the number of BACKTRACK calls grows drastically and fluctuates when $|S|$ increases, while with look-ahead, it grows gently and steadily, being mostly around the value of $|S|$.

On Pruning Threshold of SIMPATH. To study how effectively the pruning threshold η represents a trade-off between efficiency and quality of seed set, we run SIMPATH with different values of η on one moderate dataset (NetHEPT) and one large (DBLP). The look-ahead value ℓ is set to 4.

The results in Table III clearly show that on both datasets, as η decreases, the running time of SIMPATH rises, while the estimated spread of influence $\sigma(S)$ is improving. For instance, when we decrease η from 10^{-3} to 10^{-4}, the running time increases 7.7 folds while the spread is increased only by 0.41%. Other datasets follow similar behavior. It suggests that 0.001 is indeed a good choice for η. It is worth mentioning that when η is relatively small (e.g., 10^{-4} and 10^{-5}), SIMPATH can even produce seed sets with larger spread than those of MC-CELF with $10,000$ simulations (which give a spread of 1408 on NetHEPT).

On Number of Hops. The next analysis we perform is how the choice of the pruning threshold is related to the average



(a) NetHEPT (b) Last.fm (c) Flixster (d) DBLP

Fig. 9. Frequency Distribution of Average Number of Hops.

TABLE III
PRUNING THRESHOLD η AS A TRADE-OFF BETWEEN QUALITY OF SEED SETS AND EFFICIENCY. $|S| = 50$. RUNNING TIME IN MINUTES.

η	NetHEPT		DBLP	
	$\sigma(S)$	Time	$\sigma(S)$	Time
10^{-1}	1160	0.02	16592	1.02
10^{-2}	1362	0.03	17868	2.02
10^{-3}	1408	0.18	18076	12.0
10^{-4}	1414	1.3	18151	104.3
10^{-5}	1416	9.9	18350	927.2

number of hops explored. Fig. 9 shows the distribution. On all the four datasets, the distribution follows a bell curve with the mean of either 3 or 4. For instance, on Flixster, for 60K users, the influence decays below 10^{-3} in 3 hops on an average. We also found that for any user, on all four datasets, the influence decays below 10^{-3} in a maximum of 8 hops. These statistics support our conjecture that the majority of the influence flows in a small neighborhood, and thus, even though enumerating simple paths in $\#\mathbf{P}$-hard in general, computation of influence spread by enumerating simple paths can be done efficiently and accurately by focusing on a small neighborhood.

To summarize, our experiments demonstrate that SIMPATH consistently outperforms other influence maximization algorithms and heuristics. It is able to produce seed sets with quality comparable to those produced by MC-CELF, but is far more efficient and scalable. Also, SIMPATH is shown to have higher seed set quality, lower memory footprint, and better scalability than other well-established heuristics and the state of the art LDAG.

VII. SUMMARY AND FUTURE WORK

Designing a scalable algorithm delivering high quality seeds for influence maximization problem under the LT model is the main goal of this paper. The simple greedy algorithm is known to produce the best possible seed sets (in terms of influence spread) in PTIME but suffers from severe performance issues. The CELF optimization [3] helps reduce the number of spread estimation calls significantly, except in the first iteration. On the other hand, the LDAG heuristic proposed by Chen et al. [7], the current state of art, is shown to be significantly faster than the greedy algorithm and is often found to generate a seed set of high quality. We propose an alternative algorithm SIMPATH that computes the spread by exploring simple paths in the neighborhood. Using a parameter η, we can strike a balance between running time and desired quality (of the seed set). SIMPATH leverages two optimizations. The VERTEX COVER OPTIMIZATION cuts down the spread estimation calls in the first iteration, thus addressing a key limitation of CELF, while the LOOK AHEAD OPTIMIZATION improves the efficiency in subsequent iterations. Through extensive experimentation on four real data sets, we show that SIMPATH outperforms LDAG, in terms of running time, memory consumption and the quality of the seed sets. Currently, we are investigating extension of our techniques for other propagation models.

Acknowledgments. This research was partially supported by a strategic network grant from NSERC Canada on Business Intelligence Network.

REFERENCES

[1] D. Kempe, J. M. Kleinberg, and É. Tardos, "Maximizing the spread of influence through a social network," in *KDD 2003*.
[2] X. Song, B. L. Tseng, C.-Y. Lin, and M.-T. Sun, "Personalized recommendation driven by information flow," in *SIGIR 2006*.
[3] J. Leskovec et al., "Cost-effective outbreak detection in networks," in *KDD 2007*.
[4] J. Weng, E.-P. Lim, J. Jiang, and Q. He, "Twitterrank: finding topic-sensitive influential twitterers," in *WSDM 2010*.
[5] E. Bakshy, J. M. Hofman, W. A. Mason, and D. J. Watts, "Everyone's an influencer: quantifying influence on twitter," in *WSDM 2011*.
[6] W. Chen, C. Wang, and Y. Wang, "Scalable influence maximization for prevalent viral marketing in large-scale social networks," in *KDD 2010*.
[7] W. Chen, Y. Yuan, and L. Zhang, "Scalable influence maximization in social networks under the linear threshold model," in *ICDM 2010*.
[8] W. Chen, Y. Wang, and S. Yang, "Efficient influence maximization in social networks," in *KDD 2009*.
[9] M. Kimura and K. Saito, "Tractable models for information diffusion in social networks," in *ECML PKDD 2006*.
[10] A. Goyal, F. Bonchi, and L. V. Lakshmanan, "A data-based approach to social influence maximization," in *PVLDB 2012*.
[11] L. G. Valiant, "The complexity of enumeration and reliability problems," *SIAM Journal on Computing, vol. 8, no. 3, pp. 410421*, 1979.
[12] P. Domingos and M. Richardson, "Mining the network value of customers," in *KDD 2001*.
[13] A. Goyal, W. Lu, and L. V. S. Lakshmanan, "CELF++: Optimizing the greedy algorithm for influence maximization in social networks," in *WWW (Companion Volume) 2011*.
[14] K. Saito, R. Nakano, and M. Kimura, "Prediction of information diffusion probabilities for independent cascade model," in *KES 2008*.
[15] A. Goyal, F. Bonchi, and L. V. Lakshmanan, "Learning influence probabilities in social networks," in *WSDM 2010*.
[16] D. Kroft, "All paths through a maze," *Proc. IEEE 55*, 1967.
[17] D. B. Johnson, "Find all the elementary circuits of a directed graph," *J. SIAM, 4:77-84*, 1975.
[18] M. R. Garey and D. S. Johnson, *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., 1990.
[19] G. Karakostas, "A better approximation ratio for the vertex cover problem," *ACM Transactions on Algorithms*, vol. 5, no. 4, 2009.
[20] R. Schifanella et al., "Folks in folksonomies: social link prediction from shared metadata," in *WSDM 2010*.
[21] M. Jamali and M. Ester, "A matrix factorization technique with trust propagation for recommendation in social networks," in *RecSys*, 2010.
[22] S. Brin and L. Page, "The anatomy of a large-scale hypertextual web search engine," *Computer Networks*, vol. 30, no. 1-7, pp. 107–117, 1998.