ProbeSim: Scalable Single-Source and Top-k SimRank Computations on Dynamic Graphs

Yu Liu[1], Bolong Zheng[2], Xiaodong He[1], Zhewei Wei[1] *, Xiaokui Xiao[3], Kai Zheng[4], Jiaheng Lu[5]
[1]School of Information, Renmin University of China, China
[2]School of Data and Computer Science, Sun Yat-sen University
[3]School of Computer Science and Engineering, Nanyang Technological University, Singapore
[4]School of Computer Science and Engineering and Big Data Research Center, University of Electronic Science and Technology of China
[5] Department of Computer Science, University of Helsinki
[1]{yu.liu, hexiaodong_1993, zhewei}@ruc.edu.cn [2]zblchris@gmail.com
[3]xkxiao@ntu.edu.sg [4]zhengkai@uestc.edu.cn [5]jiahenglu@gmail.com

ABSTRACT

Single-source and top-k SimRank queries are two important types of similarity search in graphs with numerous applications in web mining, social network analysis, spam detection, etc. A plethora of techniques have been proposed for these two types of queries, but very few can efficiently support similarity search over large dynamic graphs, due to either significant preprocessing time or large space overheads.

This paper presents *ProbeSim*, an *index-free* algorithm for single-source and top-k SimRank queries that provides a non-trivial theoretical guarantee in the absolute error of query results. *ProbeSim* estimates SimRank similarities without precomputing any indexing structures, and thus can naturally support *real-time* SimRank queries on *dynamic* graphs. Besides the theoretical guarantee, *ProbeSim* also offers satisfying practical efficiency and effectiveness due to non-trivial optimizations. We conduct extensive experiments on a number of benchmark datasets, which demonstrate that our solutions outperform the existing methods in terms of efficiency and effectiveness. Notably, our experiments include the first empirical study that evaluates the effectiveness of SimRank algorithms on graphs with billion edges, using the idea of *pooling*.

PVLDB Reference Format:
Yu Liu, Bolong Zheng, Xiaodong He, Zhewei Wei, Xiaokui Xiao, Kai Zheng, Jiaheng Lu. ProbeSim: Scalable Single-Source and Top- k SimRank Computations on Dynamic Graphs. *PVLDB*, 11(1): 14 - 26, 2017.
DOI: https://doi.org/10.14778/3136610.3136612

1. INTRODUCTION

SimRank [11] is a classic measure of the similarities of graph nodes, and it has been adopted in numerous applications such as web mining [12], social network analysis [17], and spam detection [25]. The formulation of SimRank is based on two intuitive statements: (i) a node is most similar to itself, and (ii) two nodes are similar if their neighbors are similar. Specifically, given two nodes u and v in a graph G, the SimRank similarity of u and v, denoted as $s(u,v)$, is defined as:

$$s(u,v) = \begin{cases} 1, & \text{if } u = v \\ \dfrac{c}{|I(u)| \cdot |I(v)|} \displaystyle\sum_{x \in I(u), y \in I(v)} s(x,y), & \text{otherwise.} \end{cases} \quad (1)$$

where $I(u)$ denotes the set of in-neighbors of u, and $c \in (0,1)$ is a decay factor typically set to 0.6 or 0.8 [11,19].

Computing SimRank efficiently is a non-trivial problem that has been studied extensively in the past decade. The early effort [11] focuses on computing the SimRank similarities of all pairs of nodes in G, but the proposed *Power Method* algorithm incurs prohibitive overheads when the number n of nodes in G is large, as there exists $O(n^2)$ node pairs in G. To avoid the inherent $O(n^2)$ costs in all-pair SimRank computation, the majority of the subsequent work considers two types of SimRank queries instead

- Single-source SimRank query: given a query node u, return $s(u,v)$ for every node v in G;
- Top-k SimRank query: given a query node u and a parameter $k \geq 1$, return the k nodes v with the largest $s(u,v)$.

Existing techniques [7, 13, 14, 16, 20, 24, 27, 31] for these two types of queries, however, suffer from two major deficiencies. First, most methods [14, 16, 20, 24, 31] fail to provide any worst-case guarantee in terms of the accuracy of query results, as they either rely on heuristics or adopt an incorrect formulation of SimRank. Second, the existing solutions [7, 16, 27] with theoretical accuracy guarantees, all require constructing index structures on the input graphs with a preprocessing phase, which incurs significant space and pre-computation overheads. The only exception is the Monte Carlo method in [7] which provides an index-free solution with theoretical accuracy guarantees. While pioneering, unfortunately, this method entails considerable query overheads, as shown in [24].

Motivations. In this paper, we aim to develop an *index-free* solution for single-source and top-k SimRank queries with provable accuracy guarantees. Our motivation for devising algorithms without preprocessing is two-fold. First, index-based SimRank meth-

*Corresponding author

Proceedings of the VLDB Endowment, Vol. 11, No. 1
Copyright 2017 VLDB Endowment 2150-8097/17/09... $ 10.00.
DOI: https://doi.org/10.14778/3136610.3136612

ods often have difficulties handling *dynamic* graphs. For example, *SLING* [27], which is the state-of-art indexing-based SimRank algorithm for static graphs, requires its index structure to be rebuilt from scratch whenever the input graph is updated, and its index construction requires several hours even on medium-size graphs with 1 million nodes. This renders it infeasible for real-time queries on dynamic graphs. In contrast, index-free techniques can naturally support real-time SimRank queries on graphs with frequent updates. To the best of our knowledge, the *TSF* method [24] is the only indexing approach that allows efficient update. However, *TSF* is unable to provide any worst-case guarantees in terms of the accuracy of the SimRank estimations, which leads to unsatisfying empirical effectiveness, as shown in [34] and in our experiments.

Our second motivation is that index-based SimRank methods often fail to scale to large graphs due to their space overheads. For example, *TSF* requires an index space that is two to three orders of magnitude larger than the input graph size, and our empirical study shows that it runs out of 64GB memory for graphs over 1GB in our experiments. This renders it only applicable on small- to medium-size datasets. Further, if one considers to move the large index to the external memory, this idea would incur expensive preprocessing and query costs, as shown in our empirical study. In contrast, an index-free solution proposed in this paper does not increase the size of an original graph.

Our contributions. This paper presents an in-depth study on single-source and top-k SimRank queries, and makes the following contributions. First, for single-source and top-k SimRank queries, we propose an algorithm with provable approximation guarantees. In particular, given two constants ε_a and δ, our algorithm ensures that, with at least $1-\delta$ probability, each SimRank similarity returned has at most ε_a absolute error. The algorithm runs in $O(\frac{n}{{\varepsilon_a}^2}\log\frac{n}{\delta})$ expected time, and it does not require any index structure to be pre-computed on the input graph. Our algorithm matches the state-of-the-art index-free solution in terms of time complexity, but it offers much higher practical efficiency due to an improved algorithm design and several non-trivial optimizations.

Our second contribution is a large set of experiments that evaluate the proposed solutions with the state of the art on benchmark datasets. Most notably, we present the first empirical study that evaluates the effectiveness of SimRank algorithms on graphs with billion edges, using the idea of *pooling* borrowed from the information retrieval community. The results demonstrate that our solutions significantly outperform the existing methods in terms of both efficiency and effectiveness. In addition, our solutions are more scalable than the state-of-the-art index-based techniques, in that they can handle large graphs on which the existing solutions require excessive space and time costs in preprocessing.

2. PRELIMINARIES

2.1 Problem Definition

Table 1 shows the notations that are frequently used in the remainder of the paper. Let $G=(V,E)$ be a directed simple graph with $|V|=n$ and $|E|=m$. We aim to answer *approximate* single-source and top-k SimRank queries, defined as follows:

DEFINITION 1 (APPROXIMATE SINGLE-SOURCE QUERIES). *Given a node u in G, an absolute error threshold ε_a, and a failure probability δ, an approximate single-source SimRank query returns an estimated value $\tilde{s}(u,v)$ for each node v in G, such that*

$$|\tilde{s}(u,v)-s(u,v)|\le\varepsilon_a$$

holds for any v with at least $1-\delta$ probability. □

DEFINITION 2 (APPROXIMATE TOP-k QUERIES). *Given a node u in G, a positive integer $k<n$, an error threshold ε_a, and a failure probability δ, an approximate top-k SimRank query returns a sequence of k nodes $v_1,v_2,\dots,v_k$ and an estimated value $\tilde{s}(u,v_i)$ for each v_i, such that the following equations hold with at least $1-\delta$ probability for any $i\in[1,k]$:*

$$s(u,v_i)\ge s(u,v_i')-\varepsilon_a$$

where v_i' is the node in G whose SimRank similarity to u is the i-th largest. □

Essentially, the approximate top-k query for node u returns k nodes $v_1,\dots,v_k$ such that their *actual* SimRank similarities with respect to u are ε-close to those of the actual top-k nodes. It is easy to see that an approximate single-source algorithm can be extended to answer the approximate top-k queries, by sorting the SimRank estimations $\{\tilde{s}(u,v)\mid v\in V\}$ and output the top-k results. Therefore, our main focus is on designing efficient and scalable algorithms that answer approximate single-source queries with ε_a guarantee.

2.2 SimRank with Random Walks

In the seminal paper [11] that proposes SimRank, Jeh and Widom show that there is an interesting connection between SimRank similarities and random walks. In particular, let u and v be two nodes in G, and $W(u)$ (resp. $W(v)$) be a random walk from u that follows a randomly selected incoming edge at each step. Let t be the smallest positive integer i such that the i-th nodes of W_u and W_v are identical. Then, we have

$$s(u,v)=\mathbb{E}[c^{t-1}],\tag{2}$$

where c is the decay factor in the definition of SimRank (see Equation 1).

Subsequently, it is shown in [27] that Equation 2 can be simplified based on the concept of $\sqrt{c}$*-walks*, defined as follows.

DEFINITION 3 ($\sqrt{c}$-WALKS). *Given a node u in G, an $\sqrt{c}$-walk from u is a random walk that follows the incoming edges of each node and stops at each step with $1-\sqrt{c}$ probability.* □

A $\sqrt{c}$-walk from u can be generated as follows. Starting from $v=u$, when visiting node v, we generate a random number r in $[0,1]$ and check whether $r\le 1-\sqrt{c}$. If so, we terminate the walk at v; otherwise, we select one of the in-neighbors of v uniformly at random and proceed to that node.

Let $W'(u)$ and $W'(v)$ be two $\sqrt{c}$-walks from two nodes u and v, respectively. We say that two $\sqrt{c}$-walks *meet*, if there exists a positive integer i such that the i-th nodes of $W'(u)$ and $W'(v)$ are the same. Then, according to [27],

$$s(u,v)=\Pr\left[W'(u)\text{ and }W'(v)\text{ meet}\right].\tag{3}$$

Based on Equation 3, one may estimate $s(u,v)$ using a Monte Carlo approach [7,27] as follows. First, we generate r pairs of $\sqrt{c}$-walks, such that the first and second walks in each pair are from u and v, respectively. Let r_{meet} be the number of $\sqrt{c}$-walk pairs that meet. Then, we use r_{meet}/r as an estimation of $s(u,v)$. By the Chernoff bound, it can be shown that when $r\ge\frac{1}{2{\varepsilon_a}^2}\log\frac{1}{\delta}$, with at least $1-\delta$ probability we have $\left|\frac{r_{meet}}{r}-s(u,v)\right|\le\varepsilon_a$. In addition, the expected time required to generate r $\sqrt{c}$-walks is $O(r)$, since each $\sqrt{c}$-walk has $\frac{1}{1-\sqrt{c}}$ nodes in expectation.

The above Monte Carlo approach can also be straightforwardly adopted to answer any approximate single-source SimRank query from a node u. In particular, we can generate r $\sqrt{c}$-walks from each node, and then use them to estimate $s(u,v)$ for every node

v in G. This approach is simple and does not require any pre-computation, but it incurs considerable query overheads, since it requires generating a large number of $\sqrt{c}$-walks from each node.

2.3 Competitors

The TopSim based algorithms. To address the drawbacks of the Monte Carlo approach, Lee et al. [14] propose *TopSim-SM*, an index-free algorithm that answers top-k SimRank queries by enumerating all short random walks from the query node. More precisely, given a query node u and a number T, *TopSim-SM* enumerates all the vertices that reach u by at most T hops, and treat them as potential meeting points. Then, *TopSim-SM* enumerates, for each meeting point w, the vertices that are reachable from w within T hops. Lee et al. [14] also propose two variants of *TopSim-SM*, named *Trun-TopSim-SM* and *Prio-TopSim-SM*, which trade accuracy for efficiency. In particular, *Trun-TopSim-SM* omits the meeting points with large degrees, while *Prio-TopSim-SM* prioritizes the meeting points in a more sophisticated manner and explore only the high-priority ones.

For each node v returned, *TopSim-SM* provides an estimated SimRank $s_T(u, v)$ that equals the SimRank value approximated using the *Power Method* [11] with T iterations. When T is sufficiently large, $s_T(u, v)$ can be an accurate approximation of $s(u, v)$. However, Lee et al. [14] show that the query complexity of *TopSim-SM* is $O(d^{2T})$ time, where d is the average in-degree of the graph. As a consequence, Lee et al. [14] suggests setting $T = 3$ to achieve reasonable efficiency, in which case the absolute error in each SimRank score can be as large as c^3, where c is the decay factor in the definition of SimRank 1. Meanwhile, *Trun-TopSim-SM* and *Prio-TopSim-SM* does not provide any approximation guarantees even if T is set to a large value, due to the heuristics that they apply to reduce the number of meeting points explored.

The *TSF* algorithm. Very recently, Shao et al. [24] propose a two-stage random-walk sampling framework (*TSF*) for top-k SimRank queries on dynamic graphs. Given a parameter R_g, *TSF* starts by building R_g *one-way graphs* as an index structure. Each one-way graph is constructed by uniformly sampling *one* in-neighbor from each vertex's in-coming edges. The one-way graphs are then used to simulate random walks during query processing.

To achieve high efficiency, *TSF* approximates the SimRank score of two nodes u and v as

$$\sum_i \Pr[\text{two } \sqrt{c}\text{-walks from } u \text{ and } v \text{ meet at the } i\text{-th step}],$$

which is an over estimation of the actual SimRank. (See Section 3.3 in [24].) Furthermore, *TSF* assumes that every random walk in a one-way graph would not contain any cycle, which does not always hold in practice, especially for undirected graphs. (See Section 3.2 in [24].) As a consequence, the SimRank value returned by *TSF* does not provide any theoretical error assurance.

3. PROBESIM ALGORITHM

In this section, we present *ProbeSim*, an index-free algorithm for approximate single-source and top-k SimRank queries on large graphs. Recall that an approximate single-source algorithm can be extended to answer the approximate top-k queries, by sorting the SimRank estimations $\{\tilde{s}(u, v) \mid v \in V\}$ and output the top-k results. Therefore, the *ProbeSim* algorithm described in this section focuses on approximate single-source queries with ε_a guarantee. Before diving into the details of the algorithm, we first give some high-level ideas of the algorithm.

Table 1: Table of notations.

Notation	**Description**
G	the input graph
n, m	the numbers of nodes and edges in G
$I(v)$	the set of in-neighbors of a node v in G
$s(u, v)$	the SimRank similarity of two nodes u and v in G
$\tilde{s}(u, v)$	an estimation of $s(u, v)$
$W(u)$	a $\sqrt{c}$-walk from a node u
c	the decay factor in the definition of SimRank
ε_a	the maximum absolute error allowed in SimRank computation
δ	the failure probability of a Monte Carlo algorithm

3.1 Rationale

Let $W(u)$ and $W(v)$ be two $\sqrt{c}$-walks from two nodes u and v, respectively. Let u_i be the i-th node in $W(u)$. (Note that $u_1 = u$.) By Equation 3,

$$\begin{aligned} s(u, v) &= \Pr[W(u) \text{ and } W(v) \text{ meet}] \\ &= \sum_i \Pr[W(u) \text{ and } W(v) \text{ first meet at } u_i]. \quad (4) \end{aligned}$$

In other words, for a given $W(u) = (u_1, u_2, \ldots)$, if we can estimate the probability that an $\sqrt{c}$-walk from v first meets $W(u)$ at u_i, then we can take the sum of the estimated probabilities over all u_i as an estimation of $s(u, v)$. Towards this end, a naive approach is to generate a large number of $\sqrt{c}$-walks from v, and then check the fraction of walks that first meet $W(u)$ at u_i. However, if $s(u, v)$ is small, then most of the $\sqrt{c}$-walks is "wasted" since they would not meet $W(u)$. To address this issue, our idea is as follows: instead of sampling $\sqrt{c}$-walks from each v to see if they can "hit" any u_i, we start a graph traversal from each u_i to identify any node v that has a non-negligible probability to "walk" to u_i. Intuitively, this significantly reduces the computation cost since it may enable us to omit the nodes whose SimRank similarities to u are smaller than a given threshold ε_a.

In what follows, we first explain the details of the traversal-based algorithm mentioned above, and then analyze its approximation guarantee and time complexity. For convenience, we formalize the concept of *first-meeting probability* as follows.

DEFINITION 4 (FIRST-MEETING PROBABILITY). *Given a reverse path $\mathcal{P} = (u_1, \ldots, u_i)$ and a node $v \in V$, $v \neq u_1$, the first-meeting probability of v with respect to $\mathcal{P}$ is defined to be*

$$P(v, \mathcal{P}) = \Pr_{W(v)}[v_i = u_i, v_{i-1} \neq u_{i-1}, \ldots, v_1 \neq u_1],$$

where $W(v) = (v_1, \ldots, v_i, \ldots)$ is a random $\sqrt{c}$-walk that starts at $v_1 = v$.

Here, the subscript in $\Pr_{W(v)}$ indicates that the randomness arises from the choices of $\sqrt{c}$-walk $W(v)$. In the remainder of the paper, we will omit this subscript when the context is clear.

3.2 Basic algorithm

We describe our basic algorithm for the *ProbeSim* algorithm. Given a node $u \in V$, a *sampling error* parameter ε and a failure probability δ, the algorithm returns $\mathcal{R}$, an hash_set of $n - 1$ nodes in $V \setminus \{u\}$ and their SimRank estimations. For *EVERY* node $v \in V$, $v \neq u$, algorithm 1 returns an estimated SimRank $\tilde{s}(u, v)$ to the actual SimRank $s(u, v)$ with guarantee $\Pr[|\tilde{s}(u, v) - s(u, v)| \leq \varepsilon] \geq 1 - \delta$. Note that the basic algorithm

Algorithm 1: Basic *ProbeSim* algorithm

```
Input: Directed graph $G = (V, E)$; $u \in V$; Error $\varepsilon$ and failure probability $\delta$
Output: $\mathcal{R} = \{(v, \tilde{s}(u, v)) \mid v \in V\}$, a hash_set of size $n$ that maintains the SimRank estimations for each node $v \in V$
1  $n_r \leftarrow \frac{3c}{\varepsilon^2} \log \frac{n}{\delta}$;
2  for $k = 1$ to $n_r$ do
3      Generate $\sqrt{c}$-walk $W_k(u) = (u = u_1, \ldots, u_\ell)$;
4      Initialize hash_set $\mathcal{H}$;
5      for $i = 2, \ldots, \ell$ do
6          Set hash_set $\mathcal{S} \leftarrow$ PROBE($(u_1, \ldots, u_i)$);
7          for each $(v, Score(v)) \in \mathcal{S}$ do
8              if $(v, \tilde{s}_k(u, v)) \in \mathcal{H}$ then
9                  $\tilde{s}_k(u, v) \leftarrow \tilde{s}_k(u, v) + Score(v)$;
10             else
11                 Insert $(v, Score(v))$ to $\mathcal{H}$;
12     for each $(v, \tilde{s}(u, v)) \in \mathcal{R}$ do
13         if $\exists (v, \tilde{s}_k(u, v)) \in \mathcal{H}$ then
14             $\tilde{s}(u, v) \leftarrow \tilde{s}(u, v) \cdot \frac{k-1}{k} + \tilde{s}_k(u, v) \cdot \frac{1}{k}$;
15         else
16             $\tilde{s}(u, v) \leftarrow \tilde{s}(u, v) \cdot \frac{k-1}{k}$;
17 return $\mathcal{R}$;
```

uses unbiased sampling to produce the estimators, thus we can set $\varepsilon_a = \varepsilon$.

The pseudo-code for the basic *ProbeSim* algorithm is illustrated in Algorithm 1. The algorithm runs $n_r = \frac{3c}{\varepsilon^2} \log \frac{n}{\delta}$ independent trials (Line 1). For the k-th trial, the algorithm generates a $\sqrt{c}$-walk $W_k(u) = (u = u_1, \ldots, u_\ell)$ (Lines 2-3), and invokes the PROBE algorithm on partial $\sqrt{c}$-walk $W_k(u, i) = (u_1, \ldots, u_i)$ for $i = 2, \ldots, \ell$ (Lines 5-6). The PROBE algorithm computes a $Score(v)$ for each node $v \in V$. As we shall see later, $Score(v)$ is equal to $P(v, W_k(u, i))$, the first-meeting probability of v with respect to partial walk $W_k(u, i)$. Let $Score_i(v)$ denote the score computed by the PROBE algorithm on partial $\sqrt{c}$-walk $W_k(u, i)$, for $i = 2, \ldots, \ell$. The algorithm sums up all scores to form the estimator $\tilde{s}_k(u, v) = \sum_{i=2}^{\ell} Score_i(v)$ (Lines 7-11).

Algorithm 2: Deterministic PROBE algorithm

```
Input: A partial $\sqrt{c}$-walk $(u = u_1, \ldots, u_i)$
Output: $\mathcal{S} = \{(v, Score(v)) \mid v \in V\}$, a hash_set of nodes and their scores w.r.t. partial walk $W(u, i)$
1  Initialize hash_set $\mathcal{H}_j$ for $j = 0, \ldots, i-1$;
2  Insert $(u_i, 1)$ to $\mathcal{H}_0$;
3  for $j = 0$ to $i - 2$ do
4      for each $(x, Score(x)) \in \mathcal{H}_j$ do
5          for each $v \in \mathcal{O}(x)$ and $v \neq u_{i-j-1}$ do
6              if $(v, Score(v)) \in \mathcal{H}_{j+1}$ then
7                  $Score(v) \leftarrow Score(v) + \frac{\sqrt{c}}{|I(v)|} \cdot Score(x)$;
8              else
9                  Insert $(v, \frac{\sqrt{c}}{|I(v)|} \cdot Score(x))$ to $\mathcal{H}_{j+1}$;
10 return $\mathcal{S} = \mathcal{H}_{i-1}$;
```

Finally, for each node v, we take the average over the n_r independent estimators to form the final estimator $\tilde{s}(u, v) = \frac{1}{n_r} \sum_{k=1}^{n_r} \tilde{s}_k(u, v)$. Note that if we take the average after all n_r trials finish, it would require $\Omega(n_r \cdot n)$ space to store all the $\tilde{s}_k(u, v)$ values. Thus, we dynamically update the average estimator $\tilde{s}(u, v) \in \mathcal{R}$ for each trial (Lines 12-16). After all n_r trials finishes, we return $\mathcal{R}$ as the SimRank estimators for each node $v \in V$ (Line 17).

	a	b	c	d	e	f	g	h
$s(a, *)$	1.0	0.0096	0.049	0.131	0.070	0.041	0.051	0.051

Table 2: SimRank similarities with respect to node a.

Deterministic PROBE Algorithm. We now give a simple deterministic PROBE algorithm for computing the scores in Algorithm 1. Given a partial $\sqrt{c}$-walk $W(u, i) = (u_1, \ldots, u_i)$ that starts at $u = u_1$, the PROBE algorithm outputs $\mathcal{S} = \{(v, Score(v)) \mid v \neq u \in V\}$, a hash_set of nodes and their first-meeting probability with respect to reverse path $W(u, i)$.

The pseudo-code for the algorithm is shown in Algorithm 2. The algorithm initializes $i - 1$ hash tables $\mathcal{H}_0, \ldots, \mathcal{H}_{i-1}$ (Line 1) and adds $(u_i, 1)$ to $\mathcal{H}_0$ (Line 2). In the j-th iteration, for each node x in $\mathcal{H}_j$, the algorithm finds each out-neighbour $v \in \mathcal{O}(x)$, and checks if $(v, Score(v)) \in \mathcal{H}_{j+1}$ (Lines 3-5). If so, the algorithm adds $\frac{\sqrt{c}}{|I(v)|} \cdot Score(x)$ to $Score(v)$ (Lines 6-7). Otherwise, it adds $(v, \frac{\sqrt{c}}{|I(v)|} \cdot Score(x))$ to $\mathcal{H}_{j+1}$ (Lines 8-9). We note that in this iteration, no score is added to u_{i-j-1}, which ensures that the walk $W(v)$ avoids u_{i-j-1} at v_{i-j-1} (Line 5).

The intuition of the PROBE algorithm is as follows. For the ease of presentation, we let $Score(v, j)$ denote the score computed on the j-th iteration for node $v \in V$. One can show that $Score(v, j)$ is in fact equal to $P(v, (u_{i-j-1}, \ldots, u_i))$, the first-meeting probability of each node v with respect to reverse path $(u_{i-j-1}, \ldots, u_i)$. Consequently, after the $(i-1)$-th iteration, the algorithm computes $Score(v, i-1) = P(v, (u_1, \ldots, u_i))$, the first-meeting probability of each node v with respect to reverse path $W(u, i)$. We will make this argument rigorous in the analysis.

Running Example for Algorithm 1 and 2. Throughout the paper, we will use a toy graph in Figure 1 to illustrate our algorithms and pruning rules. For ease of presentation, we set the decay factor $c' = 0.25$ so that $\sqrt{c'} = 0.5$. The SimRank values of each node to a is listed in Table 2, which are computed by the Power Method within 10^{-5} error.

Suppose at the k-th trial, a random $\sqrt{c}$-walk $W(a) = (a_1, a_2, a_3, a_4) = (a, b, a, b)$ is generated. Figure 2 illustrates the traverse process of the deterministic PROBE algorithm. For simplicity, we only demonstrate the traverse process for partial walk $W(a, 4) = (a_1, a_2, a_3, a_4) = (a, b, a, b)$, which is represented by the right-most tree in Figure 2. The algorithm first inserts $(b, 1)$ to $\mathcal{H}_0$. Following the out-edges of $a_4 = b$, the algorithm finds a and omits it as $a_3 = a$. Next, the algorithm finds c, computes $Score(c, 1) = Score(b, 0) \cdot \frac{\sqrt{c'}}{|I(c)|} = 1 \cdot \frac{0.5}{3} = 0.167$, and insert $(c, 0.167)$ to $\mathcal{H}_1$. Similarly, the algorithm finds d and e, and inserts $Score(d, 1) = \frac{0.5}{1} = 0.5$ and $Score(e, 1) = \frac{0.5}{2} = 0.25$ to $\mathcal{H}_1$. For the next iteration, we find a, f, g and h from the out-neighbours of c, d and e. Note that b is omitted due to the fact that $a_2 = b$. The score of f at this iteration is computed by $Score(f, 2) = (Score(c, 1) + Score(d, 1) + Score(e, 1)) \cdot \frac{\sqrt{c'}}{|I(f)|} = (0.167 + 0.5 + 0.25) \cdot \frac{0.5}{4} = 0.115$.

Similarly, the algorithm computes $Score(a, 2) = 0.042$, $Score(g, 2) = 0.153$ and $Score(h, 2) = 0.153$, and insert $(a, 0.042)$, $(f, 0.115)$, $(g, 0.153)$ and $(h, 0.153)$ to $\mathcal{H}_2$. Finally, for the last iteration, the algorithm computes $Score(b, 3) = 0.011, Score(c, 3) = 0.033, Score(e, 3) = 0.038$ and $Score(f, 3) = 0.019$, and returns $\mathcal{H}_3 = \{(b, 0.011), (c, 0.033), (e, 0.038), (f, 0.019)\}$ as the results.

To get an estimation from $\sqrt{c}$-walk $W(a) = (a_1, a_2, a_3, a_4) = (a, b, a, b)$, Algorithm 1 will invoke PROBE for $W(u, 2) = (a, b)$,

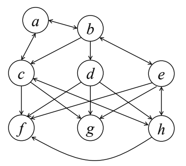


Figure 1: Toy graph.

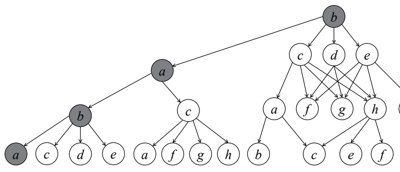


Figure 2: The probing tree.

$W(u,3) = (a,b,a)$ and $W(u,4) = (a,b,a,b)$. Each probe gives score set: $\mathcal{S}_2 = \{(c,0.167),(d,0.5),(e,0.25))\}$, $\mathcal{S}_3 = \{(f,0.021),(g,0.028),(h,0.028)\}$ and $\mathcal{S}_4 = \{(b,0.011),(c,0.033),(e,0.038),(f,0.019)\}$. As an example, the estimator $\tilde{s}(a,c)$ is computed by summing up all scores for c, which equals to $0.167 + 0.033 = 0.2$. By summing all scores up from different nodes in $W(a)$, the returned estimation of SimRank scores are $\tilde{s}(a,b) = 0.011, \tilde{s}(a,c) = 0.2, \tilde{s}(a,d) = 0.5, \tilde{s}(a,e) = 0.2877, \tilde{s}(a,f) = 0.04, \tilde{s}(a,g) = 0.028$ and $\tilde{s}(a,h) = 0.028$.

3.3 Analysis

Time Complexity. We notice that in each iteration of the PROBE algorithm, each edge in the graph is traversed at most once. Thus the time complexity of the PROBE algorithm is $O(m \cdot i)$, where i is the length of the partial $\sqrt{c}$-walk $W(u,i)$. Consequently, the expected time complexity of probing a single $\sqrt{c}$-walk in Algorithm 1 is bounded by $O(\sum_{i=1}^{\ell} i \cdot m) = O(\ell^2 m)$, where ℓ is the length of the $\sqrt{c}$-walk $W(u)$. We notice that each step in the $\sqrt{c}$-walk terminates with probability at least $1-\sqrt{c}$, so ℓ is bounded by a geometric distributed random variable X with successful probability $p = 1 - \sqrt{c}$ (here "success" means the termination of the $\sqrt{c}$-walk). It follows that

$$\mathrm{E}[\ell^2] \le E[X^2] = \mathrm{Var}(X) + \mathrm{E}[X]^2 = \frac{1-p}{p^2} + \frac{1}{p^2}$$
$$= \frac{2-p}{p^2} = \frac{1+\sqrt{c}}{(1-\sqrt{c})^2} = O(1).$$

Therefore, the expected running time of Algorithm 1 on a single $\sqrt{c}$-walk is $O(m)$. Summing up for n_r walks follows that the expected running time of Algorithm 1 is bounded by $O(mn_r) = O(\frac{m}{\varepsilon^2}\log\frac{n}{\delta})$.

Correctness. We now show that Algorithm 1 indeed gives an good estimation to the SimRank values $s(u,v)$ for each $v \in V$, $v \neq u$. The following Lemma states that each trial in Algorithm 1 gives an unbiased estimator for the SimRank value $s(u,v)$.

LEMMA 1. *For any $v \in V$ and $v \neq u$, Algorithm 1 gives an estimator $\tilde{s}(u,v)$ such that* $\mathrm{E}[\tilde{s}(u,v)] = s(u,v)$.

We need the following Lemma, which states that if we start a $\sqrt{c}$-walk $W(v) = (v_1, v_2, \ldots)$, then the score computed by the PROBE algorithm on partial walk $W(u,i) = (u_1, \ldots, u_i)$ is exactly the probability that $W(u)$ and $W(v)$ first meet at $v_i = u_i$.

LEMMA 2. *For any node $v \in V$, $v \neq u$, after the $(i-1)$-iteration, $Score(v,i)$ is equal to* $\Pr[v, W(u,i)]$*, the first-meeting probability of v with respect to partial $\sqrt{c}$-walk $W(u,i)$.*

PROOF. We prove the following claim: Let $Score(v,j)$ denote the score of v after the j-th iteration. Fix a node $v \in V$, $v \neq u$. After the j-th iteration in Algorithm 2, we have $Score(v,j) = P(v,(u_{i-j},\ldots,u_i))$, the first-meeting probability of v with respect to reverse path $(u_{i-j},\ldots,u_i)$. Recall that

$$P(v,(u_{i-j},\ldots,u_i)) = \Pr_{W(v)}[v_{j+1} = u_i, v_j \neq u_{i-1},\ldots,v_1 \neq u_{i-j}],$$

where $W(v) = (v_1,\ldots,v_{j+1},\ldots)$ is a random $\sqrt{c}$-walk that starts at $v_1 = v$.

Note that if above claim is true, then after the $(i-1)$-th iteration, we have $Score(v) = Score(v,i-1) = P(v,(u_1,\ldots,u_i)) = P(v,W(u,i))$, and the Lemma will follow. We prove the claim by induction. After the 0-th iteration, we have $Score(u_i,0) = 1$ and $Score(v,0) = 0$ for $v \neq u_i$, so the claim holds. Assume the claim holds for the $(j-1)$-th iteration. After the j-th iteration, for each $v \in V$, $v \neq u_{i-j-1}$, the algorithm set $Score(v,j+1)$ by equation

$$Score(v,j+1) = \sum_{x \in I(v)} \frac{\sqrt{c}}{|I|} \cdot Score(x,j). \tag{5}$$

By the induction hypothesis we have $Score(x,j) = P(x,(u_{i-j},\ldots,u_i))$, and thus

$$Score(v,j+1) = \sum_{\substack{x\in I(v)\\ x\neq u_{i-j+1}}} \frac{\sqrt{c}}{|I|} \cdot P(x,(u_{i-j+1},\ldots,u_i))$$
$$= \sum_{\substack{x\in I(v)\\ x\neq u_{i-j+1}}} \Pr[v_2 = x] \cdot P(x,(u_{i-j+1},\ldots,u_i)). \tag{6}$$

Here $\Pr[v_2 = x]$ denotes the probability that $W(v)$ selects x at the first step. On the other hand, $P(v,(u_{i-j},\ldots,u_i))$ can be expressed the summation of probabilities that $W(v)$ first select a node $x \in I(v)$ that is not u_{i-j+1}, and then select a reverse $\sqrt{c}$-walk from x to u_i of length $j-2$ and avoid u_{i-j+k} at k-th step. It follows that

$$P(v,(u_{i-j},\ldots,u_i))$$
$$= \sum_{\substack{x\in I(v)\\ x\neq u_{i-j+1}}} \Pr[v_2 = x] \cdot \Pr[v_{j+1} = u_i,\ldots,v_2 \neq u_{i-j+1}]$$
$$= \sum_{\substack{x\in I(v)\\ x\neq u_{i-j+1}}} \Pr[v_2 = x] \cdot P(x,(u_{i-j+1},\ldots,u_i)). \tag{7}$$

Combining equations (6) and (7) proves the claim, and the Lemma follows. $\square$

With the help of Lemma 2, we can prove Lemma 1:

PROOF OF LEMMA 1. Let $\mathbf{W}(u)$ denote the set of all possible $\sqrt{c}$-walks that starts at u. Fix a $\sqrt{c}$-walk $W(u) = (u_1,\ldots,u_\ell)$. By Lemma 2, the estimated SimRank can be expressed as $\tilde{s}(u,v) = \sum_{i=2}^{\ell} P(v,W(u,i))$. Thus we can compute the expectation of this estimation by

$$\mathrm{E}[\tilde{s}(u,v)] = \sum_{W(u)\in\mathbf{W}(u)} \Pr[W(u)] \cdot \sum_{i=2}^{\ell} P(v,W(u,i))$$
$$= \sum_{W(u)\in\mathbf{W}(u)} \sum_{i=2}^{\ell} \Pr[W(u)] \cdot P(v,W(u,i)), \tag{8}$$

where $\Pr[W(u)]$ is the probability of walk $W(u)$. Recall that $P(v,W(u,i))$ is the probability that a random $\sqrt{c}$-walk $W(v) = (v_1,\ldots,v_i,\ldots)$ first meet $W(u)$ at $u_i = v_i$. For the ease of presentation, let $I(W(u),W(v),i)$ denote that indicator variable that two $\sqrt{c}$-walk $W(v) = (v_1,\ldots,v_i,\ldots)$ and $W(u) = (u_1,\ldots,u_i,\ldots)$ first meet at $u_i = v_i$. In other word,

$I(W(u), W(v), i) = 1$ if $W(u)$ and $W(v)$ first meet at $u_i = v_i$, and $I(W(u), W(v), i) = 0$ if otherwise. We have

$$P(v, W(u, i)) = \sum_{W(v) \in \mathbf{W}(v)} \Pr[W(v)] \cdot I(W(u), W(v), i). \quad (9)$$

Combining equation (8) and (9), it follows that

$$\begin{aligned} \mathrm{E}[\tilde{s}(u, v)] &= \sum_{\substack{W(u) \in \mathbf{W}(u) \\ W(v) \in \mathbf{W}(v)}} \sum_{i=2}^{\ell} \Pr[W(u)] \cdot \Pr[W(v)] \cdot I(W(u), W(v), i) \\ &= \sum_{i=2} \Pr[W(u) \text{ and } W(v) \text{ first meet at } i] \\ &= \Pr[W(u) \text{ and } W(v) \text{ meet}] \end{aligned}$$

Note that $s(u, v)$ is the probability that $W(u)$ and $W(v)$ meet, and the Lemma follows. □

By Lemma 1 and Chernoff bound, we have the following Theorem that states by performing $n_r = \frac{3c}{\varepsilon} \log \frac{n}{\delta}$ independent trials, the error of the estimator $\tilde{s}(u, v)$ provided by Algorithm 1 can be bounded with high probability.

THEOREM 1. *For every node $v \in V$, $v \neq u$, Algorithm 1 returns an estimation $\tilde{s}(u, v)$ for $s(u, v)$ such that* $\Pr[\forall v \in V, |\tilde{s}(u, v) - s(u, v)| \leq \varepsilon] \geq 1 - \delta$.

We need the following form of Chernoff bound:

LEMMA 3 (CHERNOFF BOUND [5]). *For any set $\{x_i\}$ ($i \in [1, n_x]$) of i.i.d. random variables with mean μ and $x_i \in [0, 1]$,*

$$\Pr\left\{ \left| \sum_{i=1}^{n_x} x_i - n_x \mu \right| \geq n_x \varepsilon \right\} \leq \exp\left(-\frac{n_x \cdot \varepsilon^2}{\frac{2}{3}\varepsilon + 2\mu} \right).$$

PROOF OF THEOREM 1. We first note that in each trial k, the estimator $\tilde{s}_k(u, v)$ is a value in $[0, 1]$. It is obvious that $\tilde{s}_k(u, v) \geq 0$. To see that $\tilde{s}_k(u, v) \leq 1$, notice that $\tilde{s}_k(u, v) = \sum_{i=2}^{\ell} P(v, W(u, i))$ is a probability. More precisely, it is the probability that a $\sqrt{c}$-walk $W(v)$ meets with $\sqrt{c}$-walk $W(u)$ using the same steps.

Thus, the final estimator $\tilde{s}(u, v) = \frac{1}{n_r} \sum_{k=1}^{n_r} \tilde{s}_k(u, v)$ is the average of n_r i.i.d. random variables whos values lie in the range $[0, 1]$. Thus, we can apply Chernoff bound:

$$\Pr[|\tilde{s}(u, v) - s(u, v)| \geq \varepsilon] \leq \exp(-\varepsilon^2 n_r / (3s(u, v))).$$

Recall that $n_r = \frac{3c}{\varepsilon^2} \log \frac{n}{\delta}$, and notice that $s(u, v) \leq c$, it follows that

$$\Pr[|\tilde{s}(u, v) - s(u, v)| \geq \varepsilon] \leq \exp\left(-\log \frac{n}{\delta} \right) = \frac{\delta}{n}.$$

Taking union bound over all nodes $v \in V$ follows that

$$\Pr[\forall v \in V, |\tilde{s}(u, v) - s(u, v)| \geq \varepsilon] \leq \delta,$$

and the Theorem follows. □

4. OPTIMIZATIONS

We present three different optimization techniques to speed up our basic *ProbeSim* algorithm. The *pruning rules* eliminates unnecessary traversals in the PROBE algorithm, so that a single trial can be performed more efficiently. The *batch* algorithm builds a reachability tree to maintain all n_r c-walks, such that we do not have to perform duplicated PROBE operations in multiple trials. The randomized PROBE algorithm reduces the worst-case time complexity of our algorithm to $O(\frac{n}{\varepsilon^2} \log \frac{n}{\delta})$ in expectation.

4.1 Pruning

Although the expected steps of a $\sqrt{c}$-walks is $O(1)$, we may still find some long walks during a large number of trials. To avoid this overhead, we add the following pruning rule:

PRUNING RULE 1. *Let ε_t be the termination parameter to be determined later. In Algorithm 1, truncate all $\sqrt{c}$-walks at step $\ell_t = \log \varepsilon_t / \log \sqrt{c}$.*

We explain the intuition of this pruning rule as follows. Let $W(u) = (u_1, \ldots, u_\ell)$ denote the $\sqrt{c}$ walk, and u_i denote a node on the walk with $i > \ell_t$. For each node $v \in V$, $v \neq u$, the probability that a $\sqrt{c}$ walk $W(v)$ meets $W(u)$ at $u_i = v_i$ is at most $(\sqrt{c})^i = (\sqrt{c})^{i-\ell_t-1} \cdot (\sqrt{c})^{\ell_t+1} \leq (\sqrt{c})^{i-\ell_t-1}\varepsilon_t$, which means that u_i will contribute at most $(\sqrt{c})^{i-\ell_t-1}\varepsilon_t$ to the SimRank $s(u, v)$. Summing up over $i = \ell_t + 1, \ldots, \ell$ results in an error of $\frac{1}{1-\sqrt{c}} \cdot \varepsilon_t$. As we shall see in Theorem 2, more elaborated analysis would show that the error contributed by this pruning rule is in fact bounded by ε_t. We further notice that it is one-sided error, we can add $\varepsilon_t/2$ to each estimator, which will reduce the pruning error by a factor of 2.

The next pruning rule is inspired by the fact that the PROBE algorithm may traverse many vertice with small scores, which can be ignored for the estimation:

PRUNING RULE 2. *Let ε_p denote the pruning parameter to be determined later. In Algorithm 2, after computing all $(v, Score(v))$ in $\mathcal{H}_j$ and before descending to $\mathcal{H}_{j+1}$, we remove $(x, Score(x))$ from $\mathcal{H}_j$ if $Score(x) \cdot (\sqrt{c})^{i-j-1} \leq \varepsilon_p$.*

The intuition of pruning rule 2 is that after $i - j - 1$ more iterations in Algorithm 2, the scores computed from $Score(x, j)$ will drop down to $Score(x) \cdot \sqrt{c}^{i-j-1} \leq \varepsilon_p$. One might think that a node v may get multiple error contributions from different pruned nodes; However, the key insight is that the probabilities of the walks from v to these nodes sum up to at most 1, which implies that the error introduced by a single probe is at most ε_p. We will make this argument rigorous in Theorem 2.

Running Example for the Pruning Rules. Consider $\sqrt{c}$-walk $W(a) = (a, b, a, b, e)$, and set the termination and pruning parameters to be $\varepsilon_t = \varepsilon_p = 0.05$. We first note that the length of $W(a)$ is $\ell = 5$, and $(\sqrt{c})^{\ell_t} < 0.05$, so we truncate $W(a)$ to (a, b, a, b).

Now consider the PROBE algorithm on $W(a, 4) = (a, b, a, b)$. In the second iteration in Figure 2, recall that we have $Score(c, 1) = 0.167$. There are still two more iterations to go, and yet we have $Score(c, 1) \cdot (\sqrt{c})^2 = 0.042 < \varepsilon_p$. Thus pruning rule 2 takes place, and the algorithm does not have to descend to the subtree of c.

Correctness. For ease of presentation, we refer to the sampling error parameter in algorithm 1 as ε and the maximum allowed error as ε_a. Recall that ε_t and ε_p denote the termination parameter and pruning parameter, respectively. The following Theorem shows how these parameters affect the final error. Essentially, the error introduced by the two pruning rules is roughly the same as the sampling error. The proof of the Theorem can be found in the full version [1] of the paper.

THEOREM 2. *Assume ε, ε_t and ε_p satisfies the following inequality $\varepsilon + \frac{1+\varepsilon}{1-\sqrt{c}} \cdot \varepsilon_p + \frac{1}{2} \cdot \varepsilon_t \leq \varepsilon_a$, then Algorithm 1 achieves* $\Pr[\forall v \in V, |\tilde{s}(u, v) - s(u, v)| \leq \varepsilon_a] \geq 1 - \delta$.

Algorithm 3: Batch algorithm

```
Input: Directed graph $G = (V, E)$; $u \in V$; Error $\varepsilon_a$ and failure
       probability $\delta$;
Output: $\mathcal{R} = \{(v, \tilde{s}(u, v)) \mid v \in V\}$, a hash_set of size $n$ that
        maintains the SimRank estimations for each node $v \in V$
1  $n_r \leftarrow \frac{3c}{\varepsilon^2} \log \frac{n}{\delta}$;
2  Initialize reverse reachability tree $T$ rooted by $r$, with $r.node = u$ and
   $r.weight = 0$;
3  for $k = 1$ to $n_r$ do
4      Generate $\sqrt{c}$-walk $W_k(u) = (u_1, u_2, ..., u_l)$;
5      $r_1 \leftarrow r$;
6      for $i = 2$ to $\ell$ do
7          if $\exists r_i \in r_{i-1}.children$ and $r_i.node = u_i$ then
8              $r_i.weight \leftarrow r_i.weight + 1$;
9          else
10             Add $r_i$ as a child to $r_{i-1}$, with $r_i.node = u_i$ and
               $r_i.weight = 1$;
11 for each root-to-node path $(r = r_1, r_2, \dots, r_q)$ in $T$ do
12     $\mathcal{S} \leftarrow$ PROBE$((r_1.node, \dots, r_q.node))$;
13     for each $(v, Score(v)) \in \mathcal{S}$ do
14         $\tilde{s}(u, v) \leftarrow \tilde{s}(u, v) + \frac{r_q.weight}{n_r} \cdot Score(v)$;
15 return $\mathcal{R}$;
```

4.2 Batching Up $\sqrt{c}$-walks

A simple observation is that if two $\sqrt{c}$-walks $W_1(u)$ and $W_2(u)$ share the same partial walk $W(u, i) = (u_1, u_2, \dots, u_i)$, then we can perform a single probe on $W(u, i)$ to return the scores for both $\sqrt{c}$-walks. Meanwhile, we expect many $\sqrt{c}$-walks to share a common partial walk when the number of trial n_r is large. Based on this observation, we propose to batch up $\sqrt{c}$-walks before we perform the PROBE algorithm.

More precisely, we use a *reverse reachability tree* T to compactly store all n_r $\sqrt{c}$-walks. Each tree node r_q in T maintains a graph node $r_q.node$ and a weight integer $r_q.weight$. Let $(r = r_1, \dots, r_q)$ denote the tree path from root r to r_q, then $(r = r_1.node, \dots, r_q.node)$ corresponds to a partial $\sqrt{c}$-walk that starts at u, and $r_q.weight$ is set to be the number of $\sqrt{c}$-walks that shares this partial walk. In particular, the root r maintains $r.node = u$ and $r.weight = n_r$.

Algorithm 3 illustrates the pseudo-code of the batch method. The algorithm generates $n_r = \frac{3c}{\varepsilon^2} \log \frac{n}{\delta}$ random $\sqrt{c}$-walks, where ε is a constant that satisfies Theorem 2. To insert a $\sqrt{c}$-walk $W_i(u) = (u_1, u_2, ..., u_\ell)$ to T, the algorithm starts from the root $r_1 = r$, and recursively move down to the lower level. After r_{i-1} is processed, it checks if there is a child of r_{i-1}, denoted r_i, such that $r_i.node = u_i$ (Lines 5-7). If so, we know that partial $\sqrt{c}$-walk $(u_1, u_2, ..., u_i)$ is already recorded by $r_1, \dots, r_i$, so the algorithm increases the weight of r_i by 1 (Lines 8-9). Otherwise, it adds a new child r_i to r_{i-1}, with $r_i.node = u_i$ and $r_i.weight = 1$ (Lines 10-11). After all n_r walks are inserted to T, the algorithm starts the probe processes. For each root-to-node path $(r = r_1, r_2, \dots, r_q)$ in T, the algorithm invokes the PROBE algorithm on $(r_1.node, \dots, r_q.node)$ to get the score set $\mathcal{S}$ (Lines 11-12). Note that we can apply BFS traversal to T to enumerate all root-to-node paths. Since the number of $\sqrt{c}$-walks that share $(r_1.node, \dots, r_q.node)$ as their partial $\sqrt{c}$-walk is $r_q.weight$, the algorithm adds $\frac{r_q.weight}{n_r} \cdot Score(v)$ to the estimation of SimRank $\tilde{s}(u, v)$, for each $(v, Score(v)) \in \mathcal{S}$ (Lines 13-14). Finally, the algorithm returns $\mathcal{R}$ as the SimRank estimators (Line 15).

A running example for the batch algorithm. Suppose we have a reverse reachability tree T shown in 3(a), which records two $\sqrt{c}$-walks (a, b, c) and (a, c, a). To insert $W = (a, b, a)$ to T, the algorithm starts with the root r_1 and increase $r_1.weight$ by 1. Then it finds the child r_2 such that $r_2.node = b$, and increase $r_2.weight$ by 1. Finally, since there is no child node of r_2 with node equal to a, the algorithm inserts a new node r_6 as a child node of r_6, with $r_6.node = a$, and $r_6.weight = 1$.

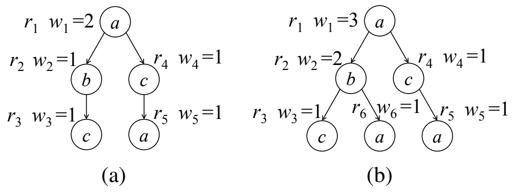


Figure 3: An example of reverse reachability tree

To estimate the SimRank values using the reverse reachability tree T, we probe partial walk represented by each tree node and sum up the scores according to the weights of nodes. For example, let $Score(b, r_j)$ denote the score computed by the PROBE algorithm for node b on the partial walk represented by r_j, for $j = 2, 3, 4, 5, 6$. Since the weights of r_2, r_3, r_4, r_5, r_6 are $2, 1, 1, 1, 1$, the final estimator $\tilde{s}(a, b)$ will be set to $\frac{1}{3} \cdot Score(b, r_2) + \frac{1}{6} \cdot Score(b, r_3) + \frac{1}{6} \cdot Score(b, r_4) + \frac{1}{6} \cdot Score(b, r_5) + \frac{1}{6} \cdot Score(b, r_6)$.

4.3 Randomized PROBE Algorithm

Recall that the running time of Algorithm 1 is $O(\frac{m}{\varepsilon_a^2} \log \frac{n}{\delta})$. The m factor comes from the PROBE algorithm, which runs in $O(m)$ time. To overcome this worst-case complexity, we present a randomized version of the PROBE algorithm. This algorithm runs in $O(n)$ time in expectation. The intuition is that for each iteration, instead of deterministically probing each out-neighbours of the nodes and computing corresponding scores, we simply sample the in-neighbours of EVERY nodes in the graph to determine if it should be put into the next iteration. We delicate the sampling process such that the probability that v gets selected by j-th iteration is exactly the score of v computed by the deterministic PROBE algorithm at j-th iteration. Since an iteration touches each node $v \in V$ exactly once, and there are constant number of iterations in expectation, the expected running time is bounded by $O(n)$.

Algorithm 4 shows the pseudo-code of the randomized PROBE algorithm. Given a partial $\sqrt{c}$-walk $W(u, i) = (u_1, \dots, u_i)$ that starts at $u = u_1$, the randomized PROBE algorithm outputs $\mathcal{S} = \{(v, Score(v)) \mid v \neq u \in V\}$, a hash_set of nodes and their first-meeting probability with respect to reverse path $W(u, i)$. Similar to its deterministic sibling, the algorithm initializes $i - 1$ hash tables $\mathcal{H}_0, \dots, \mathcal{H}_{i-1}$ (Line 1), and adds u_i to $\mathcal{H}_0$ (Line 2). In the j-th iteration, the algorithm first checks if the sum of out-degrees of the nodes in $\mathcal{H}_j$ is below n (Lines 3-4). If so, the algorithm sets the candidate set U to the union of the out-neighbours (Line 5); otherwise, it simply sets U as V (Lines 6-7). After that, the algorithm samples each node $v \in U$ with the following procedure. First, it uniformly selects an incoming edge (x, v) from the in-neighbour set $I(v)$ of v (Line 9). If $x \in \mathcal{H}_j$ (i.e., x is selected in iteration $j - 1$), the algorithm selects v into $\mathcal{H}_{j+1}$ with probability $\sqrt{c}$ (Lines 10-11). After i iterations, the algorithm returns all nodes in $\mathcal{H}_{i-1}$ with their scores set to be 1 (Line 12).

Time complexity. Each iteration in Algorithm 4 runs $O(n)$ time, so the randomized PROBE algorithm for a partial $\sqrt{c}$-walk $W(u, i)$ of length $i - 1$ runs in $O(i \cdot n)$ time. If we use randomized PROBE in Algorithm 1, the running time for a single walk is bounded by $O(\sum_{i=1}^{\ell} i \cdot n) = O(\ell^2 n)$. As proved in Section 3.3 the expectation of ℓ^2 is a constant, thus the ex-

Algorithm 4: Randomized PROBE algorithm

```
Input: A partial $\sqrt{c}$-walk $(u = u_1, \ldots, u_i)$
Output: $\mathcal{S} = \{(v, Score(v)) \mid v \in V\}$, a hash_set of nodes and their
        scores w.r.t. partial walk $W(u, i)$
1  Initialize hash_set $\mathcal{H}_j$ for $j = 0, \ldots, i-1$;
2  Insert $(u_i, 1)$ to $\mathcal{H}_0$;
3  for $j = 0$ to $i - 2$ do
4      if $\sum_{v \in \mathcal{H}_j} |\mathcal{O}(v)| \le n$ then
5          $U \leftarrow \bigcup_{v \in \mathcal{H}_j} \mathcal{O}(v)$;
6      else
7          $U \leftarrow V$;
8      for each $x \in U$, $x \ne u_{i-j-1}$ do
9          Uniformly sample an edge $(v, x)$ from $I(x)$;
10         if $v \in \mathcal{H}_j$ then
11             Insert $x$ to $\mathcal{H}_{j+1}$ with probability $\sqrt{c}$;
12 return $\mathcal{S} = \{(v, 1) \mid v \in \mathcal{H}_{i-1}\}$;
```

pected running for a single walk with randomized PROBE is bounded by $O(n)$. By setting the number of $\sqrt{c}$-walks to be $n_r = O(\frac{1}{\varepsilon^2} \log \frac{n}{\delta}) = O(\frac{1}{{\varepsilon_a}^2} \log \frac{n}{\delta})$, the time complexity of our single source SimRank algorithm with randomized PROBE is at bounded by $O(\frac{n}{{\varepsilon_a}^2} \log \frac{n}{\delta})$. We also note that in practice, the randomized PROBE algorithm tends to only visit the nodes that can be reached by u_i with non-negligible probabilities, which is few in number for real-world graphs that follow the power-law distribution.

Correctness. The following Theorem shows that the randomized PROBE algorithm gives an unbiased Bernoulli estimators for the scores computed by the deterministic PROBE algorithm. The proof of the Theorem can be found in the full version [1] of the paper.

THEOREM 3. *For EVERY node $v \in V$, $v \ne u$, Algorithm 1 with randomized PROBE returns an estimation $\tilde{s}(u, v)$ for $s(u, v)$ such that* $\Pr[\forall v \in V, |\tilde{s}(u, v) - s(u, v)| \le \varepsilon_a] \ge 1 - \delta$.

4.4 Best of both worlds

Although the randomized PROBE algorithm achieves better worst-case time complexity, it still suffers from one drawback: the sampling processes cannot be batched up. Recall that in the batch algorithm, each partial $\sqrt{c}$-walk $W(u, i)$ with weight w is probed with the deterministic PROBE algorithm exactly once, regardless of what w is. If we switch to the randomized PROBE algorithm, however, we have to perform w independent probes and take the average to get an unbiased estimator for each node. This means that batching up the $\sqrt{c}$-walks does not reduce the running time of our single source SimRank algorithm, if we use the randomized PROBE algorithm.

Now we have a deterministic PROBE algorithm that can be batched up, and a randomized PROBE algorithm that achieves $O(n)$ time complexity. To get the best of both world, we can combine the two algorithms to cope with the batch algorithm. The idea is very simple: Let r_q be a tree node with weight w, and consider a probe for the partial $\sqrt{c}$-walk for $r_1.node, \ldots, r_q.node$. After each iteration (say j-th) in the deterministic PROBE algorithm, we check if the summation of the out-degrees exceeds c_0wn for some constant c_0. If so, we know that the deterministic PROBE algorithm is going to incur a time complexity of at least c_0wn, and is no longer suitable for this partial path. Thus, we will switch to the randomized PROBE algorithm, which runs in $O(wn)$ time for a single probe.

The intuition of this combination can be explained as follow. If the partial $\sqrt{c}$-walk $W(u) = (u_1, \ldots, u_i)$ is short, then it is likely that the weight w of this partial walk is large. In the mean time, the number of nodes that are can reversely reach u_i using $\le i-1$ steps is likely to be small, so we can afford to use the deterministic PROBE algorithm to calculate the scores exactly and save a factor of w. On the other hand, if the partial $\sqrt{c}$-walk is very long, then it is likely that the weight w is small, and thus performing the randomized PROBE algorithm independently w times is affordable.

5. RELATED WORK

The first method for SimRank computation [11], referred to as the *Power Method*, is designed for deriving the SimRank similarities of all node pairs in the input graph G. It utilizes the following matrix formulation of SimRank [13]:

$$S = (cP^\top SP) \vee I, \tag{10}$$

where S is an $n \times n$ matrix such that $S(i, j)$ equals the SimRank similarities between the i-th and j-th nodes, I is an $n \times n$ identity matrix, c is the decay factor in the definition of SimRank, P is a *transition matrix* defined by the edges in G, and $\vee$ is the element-wise maximum operator. The power method starts by setting $S = I$, and then it iteratively updates all elements in S based on Equation 10, until the values of all elements converge. This method is subsequently improved in [19,30,33] in terms of either efficiency or accuracy. However, all methods proposed in [11,19,30,33] incur $O(n^2)$ space overheads, which is prohibitively expensive for large graphs.

To mitigate the inefficiency of the power method, a line of research [8,9,13,15,28,29,31] has proposed to utilize an alternative formulation of SimRank that makes it easier to compute:

$$S = cP^\top SP + (1 - c) \cdot I. \tag{11}$$

This formulation is claimed to be equivalent to that in Equation 10 [8, 9, 15, 28, 29, 31]. As pointed out in [13], however, the two formulations are rather different, due to which the techniques in [8,9,15,28,29,31] do not always return the correct SimRank similarities of node pairs.

Among the existing solutions that adopt the correct formulation of SimRank (in Equations 1 and 10), the ones most related to ours are proposed in [7, 14, 27], since they can answer approximate single-source and top-k SimRank queries with non-trivial absolute error guarantees. In particular, Fogaras and Rácz [7] propose a Monte Carlo approach that is similar to the method discussed in Section 2.2, except that it uses conventional random walks instead of $\sqrt{c}$-walks. They also present an index structure that stores pre-computed random walks to accelerate query processing. As shown in [13, 27], however, the index structure incurs tremendous space and preprocessing overheads, which makes it inapplicable on sizable graphs. Lee et al. [14] propose an index-free algorithm for top-k SimRank queries that is claimed to provide absolute error guarantees. Nevertheless, Lee et al.'s algorithm may return erroneous query results due to limited steps of random walks, as we discuss in Section 1 and 6. Tian and Xiao [27] present *SLING*, an index structure that answers any single-source SimRank query in $O(m \log \frac{1}{\varepsilon_a})$ time and requires $O(n/\varepsilon_a)$ space. *SLING* is shown to outperform the state of the art in terms of both query efficiency and accuracy, but its space consumption is more than an order of magnitude larger the size of G. Furthermore, it cannot handle updates to the input graph, and its absolute error guarantee cannot be changed after the preprocessing procedure.

In addition, there exist two other index structures [20, 24] for top-k SimRank queries, but neither of them is able to provide any

Table 3: Datasets.

Dataset	Type	n	m
Wiki-Vote	directed	7,155	103,689
HepTh	undirected	9,877	25,998
AS	directed	26,475	106,762
HepPh	directed	34,546	421,578
LiveJournal	directed	4,847,571	68,993,773
It-2004	directed	41,291,594	1,150,725,436
Twitter	directed	41,652,230	1,468,365,182
Friendster	directed	68,349,466	2,586,147,869

worst-case error guarantee, since they reply on heuristic assumptions about G that do not always hold. We discuss the technique in [24] in Section 1, and we refer interested readers to [27] for explanations of the limitation of [20]. Furthermore, Li et al. [16] propose an distributed version of the Monte Carlo approach in [7] and show that it can scale to a billion-node graph, albeit requiring 110 hours of preprocessing time and using 10 machines with 3.77TB total memory. Last but not least, there is existing work on *SimRank similarity join* [21,26,36] and variants of SimRank [4,6,18,32,35], but the solutions therein cannot be applied to address top-k and single-source SimRank queries.

6. EXPERIMENTS

This section experimentally evaluates the proposed solutions against the state of the art. All experiments are conducted on a machine with a Xeon(R) CPU E5-2620@2.10GHz CPU and 96GB memory. All algorithms are implemented in C++ and compiled by g++ 4.8.4 with the -O3 option.

Methods. We evaluate six algorithms: *ProbeSim*, *MC* [6], *TSF* [24], *TopSim* [14], *Trun-TopSim* [14] and *Prio-TopSim* [14]. As mentioned in Section 5, the three TopSim based algorithms are the state-of-the-art index-free approaches for single-source and top-k SimRank queries, and *TSF* is the state-of-the-art index structure for SimRank computations on dynamic graphs.

Datasets. We use 4 small graphs and 4 large graphs, as shown in Table 3. All datasets are obtained from public sources [2, 3].

6.1 Experiments on Small Graphs

We first evaluate the algorithms on the four small graphs, where the ground truth of the SimRank similarities can be obtained by the *Power Method*. On each dataset, we select 100 nodes uniformly at random from those with nonzero in-degrees. We generate single-source and top-k SimRank queries from each node to evaluate the algorithms.

Parameters. Following previous work [19, 20, 29, 31, 32], we set the decay factor c of SimRank to 0.6. *TSF* has two internal parameters, R_g and R_q, where R_g is the number of one-way graphs stored in the index of *TSF*, and R_q is number of times each one-way graph is reused in the query stage. In accordance with the settings in [34] and [24], we set $R_g = 300$ and $R_q = 40$. The TopSim based algorithms (i.e., *TopSim*, *Trunc-TopSim* and *Prio-TomSim*) has a common internal parameter T, which is the depth of the random walks. *Trun-TopSim* has two additional parameters h and η, where $1/h$ is the minimal degree threshold used to identify a high degree node and η is similarity threshold for trimming a random walk. *Prio-TopSim* has an extra parameter H, which is the number of random walks to be expanded at each level. We set $T = 3$, $1/h = 100$, $\eta = 0.001$, and $H = 100$, according to [34] and [14]. For *ProbeSim*, we apply all optimizations presented in Sections 4.1 and 4.3. We vary the parameter ε_a so that the overall absolute error guarantee varies from 0.0125 to 0.025, 0.05, and 0.1, so as to examine the tradeoff between the query efficiency and accuracy of *ProbeSim* in comparison to the other algorithms.

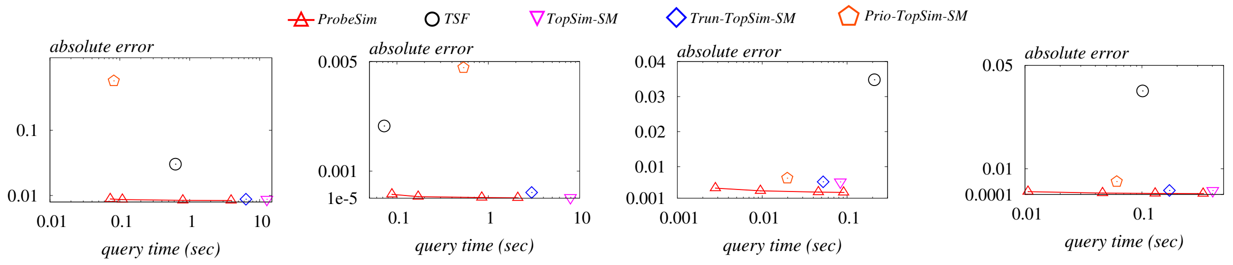
Metrics. On each of the four small graphs, we use the power method [11] with 55 iterations to compute the ground-truth SimRank similarity of each node pair. This ensures that each ground-truth value has at most 10^{-12} absolute error. Then, for each SimRank similarity returned by a method, we compute its *absolute error (AbsError)* with respect to the ground truth.

For each single-source SimRank query from a node u, we define the absolute error of the query as $AbsError = \max_{v \in V, v \neq u} |s(u,v) - \tilde{s}(u,v)|$, which is the maximum absolute error incurred by the method in computing the SimRank between u and any other node. After that, we take the average of the absolute error over 100 single-source SimRank queries and over 10 runs. Figure 4 shows the average absolute error of each method as a function of its average query costs.

For Top-k queries, we invoke the six algorithms to answer 100 top-k SimRank queries, with $k = 50$. We use *Precision@k*, the *Normalized Discounted Cumulative Gain (NDCG@k)* [10], and the Kendall Tau difference τ_k [23] to evaluate the accuracy of each algorithm. More precisely, given a query node u, let $V_k = \{v_1, \dots, v_k\}$ denote the top-k node list returned by the algorithm to be evaluated, and $V'_k = \{v'_1, \dots, v'_k\}$ to be the ground truth of the top-k results. *Precision@k* measures the fraction of answers that are among the ground-truth top-k results, which is formally defined as $Precision@k = \frac{|V_k \cap V'_k|}{k}$. *NDCG@k* measures the usefulness of a node based on its position in the result list, which is formally defined as $NDCG@k = \frac{1}{Z_k}\sum_{i=1}^{k}\frac{2^{s(u,v_i)}-1}{\log(i+1)}$, where $Z_k = \sum_{i=1}^{k}\frac{2^{s(u,v'_i)-1}}{\log(i+1)}$ is the discounted cumulative gain obtained by the ground truth of the top-k results. Recall that $s(u, v_i)$ is the actual SimRank similarity between u and v_i. Kendall Tau difference τ_k measures the accuracy of the ranking of the top-k list, which is defined as $\tau_k = \frac{\#(\text{concordant pairs}) - \#(\text{discordant pairs})}{k(k-1)/2}$. Figures 5, 6, and 7 show the average *Precision@k*, *NDCG@k*, and τ_k of each method, respectively, as functions of its average query cost.

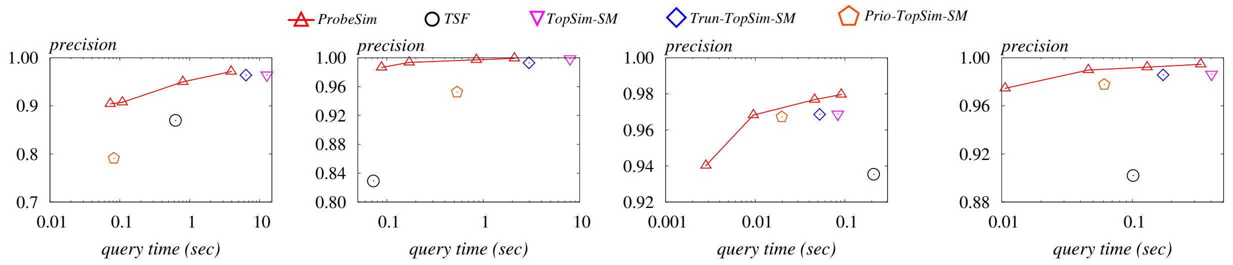
Comparisons with TopSim based algorithms. Our first observation from Figure 4 is that *ProbSim* can achieve lower *AbsError* than the TopSim based algorithms, even when its query cost is much lower than those of the latter. For example, *ProbeSim* yields an *AbsError* of 0.008 using 0.8 seconds on AS, while *Trun-TopSim* and *TopSim* achieve the same level of accuracy using 6.2 seconds and 13.5 seconds. This is mainly due to the fact that *ProbeSim* is able to estimate the SimRank value up to any given precision, while the TopSim based algorithms have a level of accuracy that is equivalent to the *Power Method* with only $T = 3$ iterations. Among the TopSim family, the *AbsError* of *Prio-TopSim* and *Trun-TopSim* are higher than that of *TopSim*, which concurs with the fact that the formal two algorithms use heuristics that trade accuracy for efficiency.

For top-k queries, Figures 5 show that the query time for *ProbeSim* is 2 to 4 times smaller than those of the TopSim based algorithms, when providing a similar level of precision. Take the Wiki-Vote dataset for example. *ProbeSim* takes less than 2 seconds to achieve a precision of 99.99%, while *TopSim* requires 8.76 seconds. In addition, *ProbeSim* achieves a precision of 99% in less than 0.08 seconds, while *Prio-TopSim* only yields a precision of 95.5% in 0.8 seconds. Meanwhile, Figure 6 and 7 show that *ProbeSim* also achieves better *NDCG@k* and Kendall Tau difference than the TopSim based algorithms do, which suggests that the ranking of the top-k results returns by *ProbeSim* is superior to those of the TopSim based algorithms.



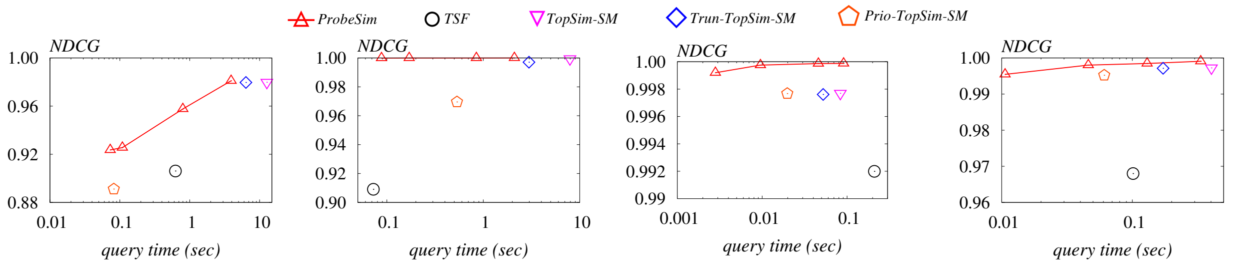
(a) AS (b) Wiki-Vote (c) HepTh (d) HepPh

Figure 4: Absolute error in answering single-source SimRank queries on small graphs



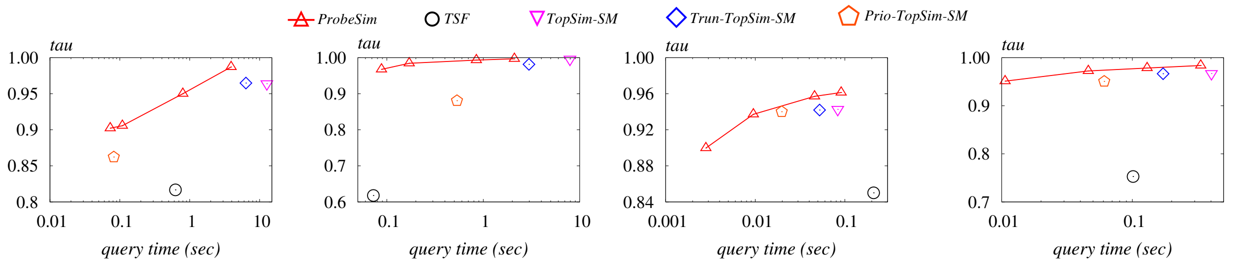
(a) AS (b) Wiki-Vote (c) HepTh (d) HepPh

Figure 5: *Precision@k* vs. query time for top-*k* SimRank queries on small graphs



(a) AS (b) Wiki-Vote (c) HepTh (d) HepPh

Figure 6: *NDCG@k* vs. query time for top-*k* SimRank queries on small graphs



(a) AS (b) Wiki-Vote (c) HepTh (d) HepPh

Figure 7: τ_k vs. query time for top-*k* SimRank queries on small graphs

Comparisons with *TSF*. To compare *ProbeSim* with *TSF*, we first observe from Figure 4 that the absolute error of *ProbeSim* is significantly lower than that of *TSF*. There are two possible reasons for *TSF*'s relatively inferior performance. First, the number of one-way graphs R_g used by *TSF* is only 300, which means that the number of random walks used in the query stage is limited, leading to inaccurate SimRank estimations. In contrast, *ProbeSim* generates much more random walks, which enables it to achieve a much better precision. Second, as we discuss in Section 2.3, *TSF* adopts two heuristics that make it unable to provide worst-case accuracy guarantee, which could contribute to its relatively large query error.

From Figures 5, we observe that *ProbeSim* dominates *TSF* for top-k queries, as it is able to achieve higher *Precision@k*, *NDCG@k* and τ_k while incurring the same or a smaller computation overhead than *TSF* does. The only exception is on Wiki-Vote, where *ProbeSim* achieves a much higher precision (99% vs. 83%) than *TSF* but incurs a slightly higher query cost (0.08 seconds vs. 0.06 seconds). Similar phenomenons can be observed for *NDCG@k* and τ_k on Wiki-Vote from Figure 6 and 7. This is due to the fact that the *AbsError* of *ProbeSim* is smaller than that of *TSF*.

Table 4: Space overheads and preprocessing costs comparison on large graphs.

Dataset	Query Time (Seconds)					Space Overhead (GBs)					Graph Size
	ProbeSim	*TopSim*	*Trun-TopSim*	*Prio-TopSim*	*TSF*	*ProbeSim*	*TopSim*	*Trun-TopSim*	*Prio-TopSim*	*TSF*	
LiveJournal	0.4	554.73	13.31	1.26	0.52	0.05	17.2	10.1	0.09	10.8	0.88 GB
IT-2004	0.006	4.91	2.34	0.21	0.93	0.4	1.7	0.9	1.0	83.7	10.9 GB
Twitter	13.2	N/A	N/A	6220	175.34	0.6	N/A	N/A	1.2	79.1	14.3 GB
Friendster	3.2	N/A	N/A	58.4	1036	0.9	N/A	N/A	1.9	99.2	23.4 GB

An interesting observation for Wiki-Vote is that the *AbsError* of *TSF* is lower than that of *Prio-TopSim*, while the *Precision@k*, *NCDG@k* and τ_k of *Prio-TopSim* are higher than those of *TSF*. One possible explanation is that Wiki-Vote is "locally dense" graph, in that more than 60% of its nodes have zero in-degree, while the remaining ones form a dense subgraph. Therefore, by setting $H = 100$, *Prio-TopSim* may omit some nodes with small SimRank values, which leads to large *AbsError*. However, *Prio-TopSim* may still examine most of the nodes with large SimRank values, thus achieving a relatively high precision.

6.2 Experiments on Large Graphs

Next, we evaluate the algorithms on the four large graphs with up to 2.59 billions of edges. Previous work ignores the accuracy comparisons on such graphs, as the ground truth of top-k results on these graphs are unavailable due to the high computational cost of the *Power Method*. To the best of our knowledge, we are the first to empirically evaluate both accuracy and efficiency of SimRank algorithms on billion-edge graphs.

Pooling. As *Power Method* only works for small graphs, we need an alternative approach to evaluate the accuracy of SimRank algorithms on large graphs. Towards this end, we use *pooling* [22], which is a standard approach for evaluating top-k documents ranking quality in Information Retrieval (IR) systems when the ground-truth ranking scores of all documents are difficult to obtain. The basic idea of pooling is as follows. Suppose that we are to evaluate ℓ IR systems, $A_1, \ldots, A_\ell$, each of which aims to return the top-k documents that are most relevant to a certain query. We first take the top-k documents returned by each system, and we merge them into a pool, with duplicates removed. Then, we present the results in the pool to experts for evaluation. Based on the relevance scores provided by the experts, we pick the best k documents from the pool, and use them as the ground truth for evaluating the top-k results returned by $A_1, \ldots, A_\ell$.

In the scenario of evaluating SimRank algorithms, we use single-pair Monte Carlo algorithm as the "expert" for gauging the results in the pool. More precisely, for each query node u, we retrieve the top-k nodes returned by each algorithm, remove the duplicates, and merge them into a pool. For each node v in the pool, we estimate $s(u, v)$ using the Monte Carlo algorithm. We set the parameters of the Monte Carlo algorithms such that it incurs an error less than 0.0001 with a confidence over 99.999%. Then we take the k nodes with the highest estimated SimRank scores from the pool as the ground truth. Essentially, these k results are the best possible k nodes that can be obtained by any of the algorithms considered.

Parameters and setups. We compare *ProbeSim*, *TopSim*, *Trun-TopSim*, *Prio-TopSim* and *TSF* using the pooling approach. On each dataset, we select 20 nodes uniformly at random from the nodes with nonzero in-degrees, and we generate top-k SimRank queries from each node. We generate top-k SimRank queries from each node to evaluate the algorithms. For *TSF* and the TopSim based algorithms, we use the same parameters as in the experiments on small graphs. For *ProbeSim* , however, we can no longer vary the error parameter ε_a, since changing its parameters may result in a different "ground truth" top-k nodes in the pool, rendering it difficult to compare different algorithms. Therefore, we fix $\varepsilon_a = 0.1$ for *ProbeSim* in this set of experiments.

Table 4 shows the average query time of each algorithm, while Figure 8, 9, and 10 show the *Precision@k*, *NDCG@k*, and τ_k, respectively, of each algorithm. We exclude *TopSim* and *Trun-TopSim* from the experiments on Twitter and Friendster, because for some queries they either run out of memory or require more than 24 hours. In addition, on Friendster, the index size of *TSF* exceeds the size of the main memory (i.e., 96GB), due to which we move 100 one-way graphs in *TSF* to the disk, as suggested in [24].

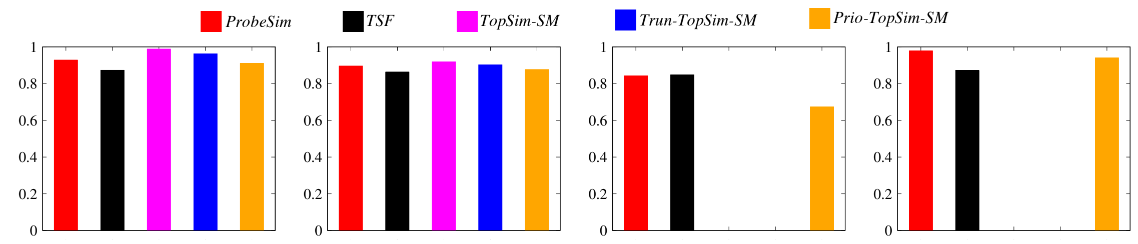
Comparisons with TopSim based algorithms. Our first observation from Table 4 is that the query cost of *ProbeSim* is significantly lower than those of the TopSim based algorithms on all four graphs. In particular, on the Twitter dataset, *TopSim* and *Trun-TopSim* could not finish query processing in 24 hours, while *Prio-TopSim* takes 2 hours on average to answer a query. On the other hand, *ProbeSim* is able to answer a query within 13 seconds on average.

It has been observed in [34] that the running time of TopSim based algorithms is sensitive to the subgraph structure and density around the query node. For example, the running time of *Prio-TopSim* on "locally dense" graphs, such as Twitter and Friendster, are higher than those on "locally sparse" graphs, such as IT-2004. Table 4 suggests that *ProbeSim* shares the same property, as its query cost on Twitter is significantly higher than those on other graphs. In contrast, the query cost of ProbeSim is less sensitive to the local subgraph structure and density.

Figure 8, 9, and 10 show that in terms of accuracy, *ProbeSim* outperforms the TopSim based algorithms on Twitter and Friendster. For example, *ProbeSim* achieves an average precision of 84% on Twitter, while *Prio-TopSim* only achieves an average precision of 67%. On LiveJournal and IT-2004, *TopSim* and *Trun-TopSim* offer a slightly better accuracy than *ProbeSim* does, at the cost of significantly higher cost. For example, *TopSim* yields a precision of 91.7% on IT-2004, while *ProbeSim* achieves a precision of 89.5%; however, the running times of *TopSim* is 600 times higher than that of *ProbeSim* on IT-2004. Figures 9 shows that *NDCG@k* of *TopSim* and *ProbeSim* are essentially the same on IT-2004, which indicates that the accuracy of the two methods are highly comparable.

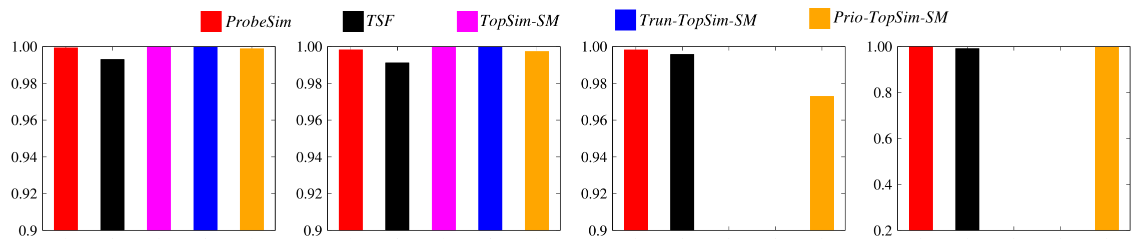
Comparisons with *TSF*. From Table 4, we first observe that *ProbeSim* achieves better query time comparing to *TSF*. For example, it takes 180 seconds for *TSF* to answer a query on Twitter, while *ProbeSim* only requires 12 seconds. Furthermore, *TSF* runs out of 96 GB memory on Friendster, even though the dataset itself is only 23 GB in size. Consequently, *TSF* has to store some of the one-way graphs on the disk, which leads to severe degradation of query efficiency. On the other hand, *ProbeSim* is able to handle a query on Friendster within 3.2 seconds on average.

LiveJournal is the only dataset on which *ProbeSim* and *TSF* use the same amount of time to answer a query. The main reason is that (i) LiveJournal (with only 4 million nodes) is a relatively small graph comparing to the other three large graphs, and (ii) the query



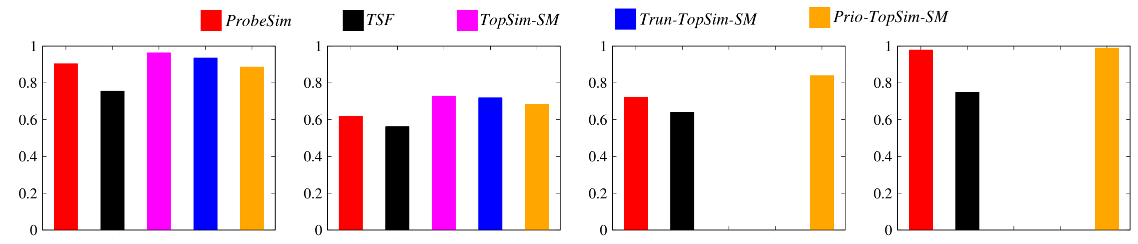
(a) LiveJournal (b) IT-2004 (c) Twitter (d) Friendster

Figure 8: *Precision@k* for top-*k* SimRank queries on large graphs



(a) LiveJournal (b) IT-2004 (c) Twitter (d) Friendster

Figure 9: *NDCG@k* for top-*k* SimRank queries on large graphs



(a) LiveJournal (b) IT-2004 (c) Twitter (d) Friendster

Figure 10: τ_k for top-*k* SimRank queries on large graphs

cost of *TSF* tends to decrease when the graph size reduces. In contrast, *ProbeSim* is less sensitive to the graph size.

In terms of query accuracy, Figure 8, 9, and 10 show that *ProbeSim* is able to provide more accurate results than *TSF* does on LiveJournal,IT-2004, and Friendster. For example, on Friendster, *ProbeSim* achieves a *Precision@k* of 98% and an *NDCG@k* of 0.9998, while *TSF* yields a *Precision@k* of 87% and an *NDCG@k* of 0.9914. On Twitter, *ProbeSim* and *TSF* offer almost the same *Precision@k* and *NDCG@k*, but *ProbeSim* outperforms *TSF* on the Kendall Tau difference. This suggests that the ranking accuracy of *ProbeSim* is better than that of *TSF*. We also observe that the precision of all three algorithms are relatively low on Twitter, which implies that SimRank on Twitter is still a difficult problem to solve.

An interesting observation is that *TSF* outperforms *Prio-TopSim* in terms of accuracy on Twitter, which contrasts the case on small graphs. As mentioned in [24], *Prio-TopSim* only expands H random walks at each level, and hence, its performance heavily rely on the random walks chosen. Because Twitter has very denser local structures, $H = 100$ may not be not sufficient for *Prio-TopSim* to explore all possible candidates. In contrast, the random sample framework of *TSF* treats each node equally, and thus, it gives relatively stable performance across social graphs and web graphs.

Finally, Table 4 shows that *TSF* incurs significant overheads in terms of space. In particular, the index size of *TSF* is one to two orders of magnitude larger than the size of the input graph G. For example, the index of *TSF* for IT-2004 is 85 GB, while the graph size is only 8 GB. Furthermore, *TSF* runs out of memory when preprocessing Friendster, which leads to performance degradation. In contrast, *ProbeSim* is able to efficiently handle queries on all large graphs without any preprocessing.

7. CONCLUSIONS

This paper presents *ProbeSim*, an algorithm for single-source and top-k SimRank computation without preprocessing. *ProbeSim* answers any single-source SimRank query in $O(\frac{n}{{\varepsilon_a}^2}\log\frac{n}{\delta})$ expected time, and it ensures that, with $1-\delta$ probability, all SimRank similarities returned have at most ε_a absolute error. Our experiments show that the algorithm significantly outperforms the existing approaches in terms of query efficiency, and they are more scalable than the existing index-based methods, as they are able to handle graphs that are too large for the latter to preprocess. For future work, we plan to study lightweight indexing approaches for SimRank that provide higher effectiveness than our current algorithms on large graphs (such as Twitter) without incurring significant space and time in computation.

8. ACKNOWLEDGMENTS

This work was partly supported by the National Natural Science Foundation of China (No. 61502503, No. 61532018, No. 61502324 and No. 61472427), by the National Key Basic Research Program (973 Program) of China (No. 2014CB340403, No. 2012CB316205), by Academy of Finland (310321), by the DSAIR center at the Nanyang Technological University, by a gift grant from Microsoft Research Asia, and by Grant MOE2015-T2-2-069 from MOE, Singapore.

9. REFERENCES

[1] https://arxiv.org/abs/1709.06955.

[2] http://snap.stanford.edu/data/index.html.

[3] http://law.di.unimi.it/datasets.php.

[4] I. Antonellis, H. G. Molina, and C. C. Chang. Simrank++: query rewriting through link analysis of the click graph. *PVLDB*, 1(1):408–421, 2008.

[5] F. R. K. Chung and L. Lu. Concentration inequalities and martingale inequalities: A survey. *Internet Mathematics*, 3(1):79–127, 2006.

[6] D. Fogaras and B. Rácz. Scaling link-based similarity search. In *WWW*, pages 641–650, 2005.

[7] D. Fogaras, B. Rácz, K. Csalogány, and T. Sarlós. Towards scaling fully personalized pagerank: Algorithms, lower bounds, and experiments. *Internet Mathematics*, 2(3):333–358, 2005.

[8] Y. Fujiwara, M. Nakatsuji, H. Shiokawa, and M. Onizuka. Efficient search algorithm for simrank. In *ICDE*, pages 589–600, 2013.

[9] G. He, H. Feng, C. Li, and H. Chen. Parallel simrank computation on large graphs with iterative aggregation. In *KDD*, pages 543–552, 2010.

[10] K. Järvelin and J. Kekäläinen. Ir evaluation methods for retrieving highly relevant documents. In *Proceedings of the 23rd annual international ACM SIGIR conference on Research and development in information retrieval*, pages 41–48. ACM, 2000.

[11] G. Jeh and J. Widom. Simrank: a measure of structural-context similarity. In *SIGKDD*, pages 538–543, 2002.

[12] R. Jin, V. E. Lee, and H. Hong. Axiomatic ranking of network role similarity. In *KDD*, pages 922–930, 2011.

[13] M. Kusumoto, T. Maehara, and K. Kawarabayashi. Scalable similarity search for simrank. In *SIGMOD*, pages 325–336, 2014.

[14] P. Lee, L. V. S. Lakshmanan, and J. X. Yu. On top-k structural similarity search. In *ICDE*, pages 774–785, 2012.

[15] C. Li, J. Han, G. He, X. Jin, Y. Sun, Y. Yu, and T. Wu. Fast computation of simrank for static and dynamic information networks. In *EDBT*, pages 465–476, 2010.

[16] Z. Li, Y. Fang, Q. Liu, J. Cheng, R. Cheng, and J. Lui. Walking in the cloud: Parallel simrank at scale. *PVLDB*, 9(1):24–35, 2015.

[17] D. Liben-Nowell and J. M. Kleinberg. The link-prediction problem for social networks. *JASIST*, 58(7):1019–1031, 2007.

[18] Z. Lin, M. R. Lyu, and I. King. Matchsim: a novel similarity measure based on maximum neighborhood matching. *KAIS*, 32(1):141–166, 2012.

[19] D. Lizorkin, P. Velikhov, M. N. Grinev, and D. Turdakov. Accuracy estimate and optimization techniques for simrank computation. *VLDB J.*, 19(1):45–66, 2010.

[20] T. Maehara, M. Kusumoto, and K. Kawarabayashi. Efficient simrank computation via linearization. *CoRR*, abs/1411.7228, 2014.

[21] T. Maehara, M. Kusumoto, and K. Kawarabayashi. Scalable simrank join algorithm. In *ICDE*, pages 603–614, 2015.

[22] C. D. Manning, P. Raghavan, H. Schütze, et al. *Introduction to information retrieval*, volume 1. Cambridge university press Cambridge, 2008.

[23] R. Nelsen. Kendall tau metric. *Encyclopaedia of Mathematics*, 3:226–227, 2001.

[24] Y. Shao, B. Cui, L. Chen, M. Liu, and X. Xie. An efficient similarity search framework for simrank over large dynamic graphs. *PVLDB*, 8(8):838–849, 2015.

[25] N. Spirin and J. Han. Survey on web spam detection: principles and algorithms. *SIGKDD Explorations*, 13(2):50–64, 2011.

[26] W. Tao, M. Yu, and G. Li. Efficient top-k simrank-based similarity join. *PVLDB*, 8(3):317–328, 2014.

[27] B. Tian and X. Xiao. SLING: A near-optimal index structure for simrank. In *SIGMOD*, pages 1859–1874, 2016.

[28] W. Yu, X. Lin, and W. Zhang. Fast incremental simrank on link-evolving graphs. In *ICDE*, pages 304–315, 2014.

[29] W. Yu, X. Lin, W. Zhang, L. Chang, and J. Pei. More is simpler: Effectively and efficiently assessing node-pair similarities based on hyperlinks. *PVLDB*, 7(1):13–24, 2013.

[30] W. Yu and J. McCann. Gauging correct relative rankings for similarity search. In *CIKM*, pages 1791–1794, 2015.

[31] W. Yu and J. A. McCann. Efficient partial-pairs simrank search for large networks. *PVLDB*, 8(5):569–580, 2015.

[32] W. Yu and J. A. McCann. High quality graph-based similarity search. In *SIGIR*, pages 83–92, 2015.

[33] W. Yu, W. Zhang, X. Lin, Q. Zhang, and J. Le. A space and time efficient algorithm for simrank computation. *World Wide Web*, 15(3):327–353, 2012.

[34] Z. Zhang, Y. Shao, B. Cui, and C. Zhang. An experimental evaluation of simrank-based similarity search algorithms. *PVLDB*, 10(5):601–612, 2017.

[35] P. Zhao, J. Han, and Y. Sun. P-rank: a comprehensive structural similarity measure over information networks. In *CIKM*, pages 553–562, 2009.

[36] W. Zheng, L. Zou, Y. Feng, L. Chen, and D. Zhao. Efficient simrank-based similarity join over large graphs. *PVLDB*, 6(7):493–504, 2013.