An Efficient Similarity Search Framework for SimRank over Large Dynamic Graphs

Yingxia Shao♯ Bin Cui♯ Lei Chen§ Mingming Liu♯ Xing Xie♮
♯Key Lab of High Confidence Software Technologies (MOE), School of EECS, Peking University
§Department of Computer Science and Engineering, HKUST
♮Microsoft Research
♯{simon0227, bin.cui, minglotus}@pku.edu.cn
§leichen@cse.ust.hk ♮xingx@microsoft.com

ABSTRACT

SimRank is an important measure of vertex-pair similarity according to the structure of graphs. The similarity search based on SimRank is an important operation for identifying similar vertices in a graph and has been employed in many data analysis applications. Nowadays, graphs in the real world become much larger and more dynamic. The existing solutions for similarity search are expensive in terms of time and space cost. None of them can efficiently support similarity search over large dynamic graphs. In this paper, we propose a novel two-stage random-walk sampling framework (TSF) for SimRank-based similarity search (e.g., top-k search). In the preprocessing stage, TSF samples a set of one-way graphs to index raw random walks in a novel manner within $\mathcal{O}(NR_g)$ time and space, where N is the number of vertices and R_g is the number of one-way graphs. The one-way graph can be efficiently updated in accordance with the graph modification, thus TSF is well suited to dynamic graphs. During the query stage, TSF can search similar vertices fast by naturally pruning unqualified vertices based on the connectivity of one-way graphs. Furthermore, with additional R_q samples, TSF can estimate the SimRank score with probability $1 - 2e^{-2\epsilon^2 \frac{R_g R_q}{(1-c)^2}}$ if the error of approximation is bounded by $1 - \epsilon$. Finally, to guarantee the scalability of TSF, the one-way graphs can also be compactly stored on the disk when the memory is limited. Extensive experiments have demonstrated that TSF can handle dynamic billion-edge graphs with high performance.

1. INTRODUCTION

SimRank [14] is a general link-based similarity measure. Its definition is inspired by the observation that is "objects related to similar objects are also similar". The similarity scores of objects are computed based on the objects' structural context which is modeled as a directed graph. SimRank has been successfully employed in many data analysis applications, such as sponsored search [1], web spam detection [5], schema matching [22] and many other web applications [25, 31, 8]. Among these applications, a common operation is *top-k* similarity search which is defined as follows.

Proceedings of the VLDB Endowment, Vol. 8, No. 8
Copyright 2015 VLDB Endowment 2150-8097/15/04.

- **Problem Definition** *(Top-k search)*: given a query vertex v of a graph G and the number of similar vertices k, top-k search finds k vertices with the highest SimRank scores with respect to query vertex v in the graph G. This type of search is denoted by $q(v, k)$.

A straightforward approach for the *top-k* search is first computing the SimRank scores of all-pairs off-line by using available solutions of SimRank computation, then ranking the scores with respect to the query vertex, and lastly outputting the top-k vertices. This approach is extremely fast for online querying. However, the computation of SimRank scores of all-pairs is prohibitively expensive in terms of both time and space costs. For example, the solution of computing SimRank scores in [21] costs $\mathcal{O}(N^3)$ time and $\mathcal{O}(N^2)$ space. As a consequence, the straightforward approach cannot process large graphs.

Since the order of vertices, not the exact SimRank scores, really matters in the top-k search, many researches resort to compute the approximate scores to improve the performance and capability of solutions. One good strategy is derived from the matrix formula of SimRank. Several approximation algorithms [18, 10] use low-rank approximation techniques [23] (e.g., Singular Value Decomposition) to approximately represent the original matrix by several low-rank matrices, then top-k search can be completed in linear time. However, to obtain the low-rank representation, it requires quadratic time [18]. These algorithms incur expensive preprocessing costs when handling large graphs.

A more promising solution for similarity search on large-scale graphs is to design approximation algorithms based on the random surfer-pairs model. In the model, a single-pair SimRank score is interpreted as the expected first-meeting time of two random surfers. Fogaras and Rácz [9] adopted Monte-Carlo simulation technique to estimate first-meeting time through sampling random walks. The estimated values are indexed by a data structure called a "fingerprint". The fingerprints can be built in $\mathcal{O}(NR)$ time, where R is the number of samples; then top-k search is efficiently executed based on the fingerprints. Most recently, Kusumoto et al. [15] developed another random-walk based algorithm. The algorithm computes upper bounds for each vertex's SimRank score through random-walk sampling in $\mathcal{O}(NRT)$ time, where T is the length of a random walk, and then uses the bound to filter unqualified vertices for speeding up query processing. Both random-walk based approaches are able to handle large graphs since the preprocessing and query execution can be completed in linear time.

On the other hand, graph structure evolution is a typical characteristic in the real world graph applications. Recent decades have witnessed the rapid emergence of large dynamic graphs in various areas, hence the similarity search on large dynamic graphs becomes important. However, the existing solutions [18, 29] for dynamic graphs are based on all-pairs SimRank scores so that they cannot process large dynamic graphs. Furthermore, the aforementioned random-walk based solutions cannot handle dynamic graphs efficiently either. The reason is that the indexes (i.e., fingerprints and bounds) are built by aggregating some properties from the sampled random walks, and they requires to be reconstructed from scratch when the original graph is (even slightly) modified. Due to the expensive cost of rebuilding, similarity search fails to be aware of the graph modifications immediately. Taking the twitter graph dataset as an example, Kusumoto's solution [15] takes about 7.7 hours to build the index. Thus the search results only reflect the structure of graph at least 7.7 hours ago, and cannot meet the real-time analysis requirements. In summary, none of existing solutions can perfectly support similarity search on large dynamic graphs.

To efficiently solve top-k search on both static and dynamic graphs, we propose a novel two-stage random-walk sampling framework (TSF). TSF organizes raw random walks in a novel fashion; it indexes the walks by one-way graphs in $\mathcal{O}(NR_g)$ time and space, where R_g is the number of one-way graphs. The **one-way graph** is a special subgraph of the reversed original graph and contains exactly one random walk for each vertex. For a given top-k search query $q(u,k)$, TSF first samples R_q new random walks for vertex u, then searches the similar vertices based on R_q random walks and R_g one-way graphs in $\mathcal{O}(R_qR_gT^2)$ time, where T is the length of a random walk. The high performance of query stage is guaranteed by the property of one-way graph's connectivity, which helps prune unqualified vertices without any cost overhead, and an approximation random model, which speeds up the similarity estimation. Further, with the additional R_q samples, TSF can estimate the SimRank score with probability $1-2e^{-2\epsilon^2\frac{R_gR_q}{(1-c)^2}}$ when the error of approximation is bounded by $1-\epsilon$.

Unlike other random-walk based approaches, TSF is also well suited to the dynamic graph scenario. This is because raw random walks are directly indexed in one-way graphs, and the one-way graph can be locally updated when the original graph is changed. In practice, we implement the update semantics via a simple but efficient log-based approach. Finally, to guarantee the scalability of TSF, the one-way graphs can further be compactly stored on the disk when the memory is limited.

To summarize, we have made the following contributions.

- We propose an approximation random surfer-pairs model to speed up the SimRank score estimation.
- We propose a novel two-stage random-walk sampling framework for SimRank-based similarity search, called TSF.
- We introduce the one-way graph to directly and compactly encode a set of random walks in TSF.
- We extend our solution onto dynamic graphs by efficiently updating one-way graph with a log-based approach.
- We design an external storage format for one-way graphs to support similarity search on large-scale graphs when the available memory is limited.

Organization. We introduce the background material in the next section. The details of TSF and top-k similarity search are elaborated in Section 3. The miscellaneous features of TSF are described in Section 4. The results of experimental evaluation are presented in Section 5. Finally, we present the related work and conclude this paper in Section 6 and Section 7.

2. BACKGROUND

Here we review key concepts of SimRank and the basic Monte-Carlo based approach for estimating SimRank scores. In this paper, a directed graph is denoted by $G=(V,E)$, where V is the vertex set and E is the directed edge set. The reversed G is denoted by G_r, which uses the same vertex set V with all of the edges reversed compared to the orientation of corresponding edges in G.

2.1 SimRank

SimRank, proposed by Jeh and Windom [14], is a domain independent similarity measure and is computed on a directed graph G which models objects as vertices and relationships as edges. According to the intuitive observation mentioned in Section 1, the SimRank score iteratively aggregates the similarity from incoming neighbor pairs. Let $s(u,v)$ denote the similarity between vertices u and v ($u,v\in V$), then it can be written as

$$s(u,v)=\begin{cases}1 & if\ u=v;\\ \frac{c}{|N_I(u)||N_I(v)|}\sum\limits_{u'\in N_I(u)}\sum\limits_{v'\in N_I(v)}s(u',v') & if\ u\neq v.\end{cases}$$

where $c\in(0,1)$ is the decay factor and $N_I(u)$ represents the incoming neighbors of vertex u. Table 1 lists the notations frequently used in this paper.

Symbols	Description
G, G_r	a directed graph and the reversed one
N	the number of vertices
$N_O(u)$ $N_I(u)$	the outgoing / incoming neighbors of vertex u
$s(u,v)$, $s_a(u,v)$	similarity of a vertex pair (u,v)
R R_g R_q	the number of samples the number of one-way graphs in TSF the number of samples at query time in TSF
T	the number of iterations/steps
W_v, $w_v(v_0,...,v_k)$	a random walk that starting from vertex $v_0(=v)$ with k steps
G_{owg}, G_{rowg}	a one-way graph and the reversed one

Table 1: Notations

Jeh and Windom further proposed a random surfer-pairs model to interpret the $s(u,v)$ as the expected first-meeting time of two random surfers who respectively start from vertices u and v in the reversed graph G_r [14]. Let T be the length of a random walk and I_w be the initial similarity of a single vertex w, according to the random surfer-pairs model $s(u,v)$ can be rewritten as

$$s(u,v)=\sum_{t=1}^{T}\sum_{w\in V}p_{ft}(u,v,w)c^tI_w, \quad (1)$$

where $p_{ft}(u,v,w)$ is the first-meeting probability that both random surfers u and v reach w at time t. Usually I_w is one.

2.2 Monte-Carlo Based Estimation

Equation 1 expresses $s(u,v)$ as the expected value which is related to the first-meeting vertices between random walks. The computation of exact $s(u,v)$ is expensive, since we need to enumerate all the random walks. Monte-Carlo simulation [24] is a general process to efficiently estimate a numerical value by random sampling. Here an approximation SimRank score can be computed by sampling random walks for each related vertex. The procedure of Monte-Carlo based estimation is listed in Algorithm 1.

To estimate $s(u,v)$, the algorithm conducts R random samples. In each sample, two random walks W_{ui} and W_{vi} are sampled starting from vertices u and v in G_r, then the delta SimRank score $\delta_{u,v}$ is computed by finding the first-meeting vertex between W_{ui} and W_{vi} (Line 5), finally it finishes this sample by increasing $s(u,v)$

Algorithm 1 Basic Monte-Carlo based estimation of $s(u, v)$

```
Input: reversed graph $G_r$, vertices $u, v$
Output: $s(u, v)$
1: $s(u, v) \leftarrow 0$;
2: for $i \leftarrow 0$ to $R - 1$ do
3:     sample a $W_{ui}$=$w_u(u_0, u_1, ..., u_T)$ from $G_r$;
4:     sample a $W_{vi}$=$w_v(v_0, v_1, ..., v_T)$ from $G_r$;
5:     $\delta_{u,v} = c^t, t = \min\{t|u_t = v_t, 1 \le t \le T\}$
6:     $s(u, v) \leftarrow s(u, v) + \delta_{u,v}$;
7: end for
8: return $\frac{s(u,v)}{R}$;
```

with $\delta_{u,v}$. Theorem 1 guarantees the accuracy of estimated SimRank score with respect to the number of samples.

THEOREM 1. *Let $s'(u, v)$ be the estimated score based on Algorithm 1 and $s(u, v)$ is the exact score, then*

$$Pr\{|s'(u, v) - s(u, v)| > \epsilon\} \le 2e^{-2\epsilon^2 \frac{R}{(B-c)^2}},$$

where ϵ is the error bound and B is $\min\{1, \frac{c}{1-c}\}$.

PROOF. The random variable $\delta_{u,v}$ is in the range $[c, \min\{1, \frac{c}{1-c}\}]$ under the random surfer-pairs model. According to the Hoeffding's inequality [13], the result can be derived. □

Similar to other random-walk based solutions [9, 15] for similarity search, the Monte-Carlo simulation is the basic technique in TSF as well.

3. A NOVEL FRAMEWORK FOR SIMRANK BASED SIMILARITY SEARCH

For the random-walk based SimRank estimation, the essence is to sample a set of random walks and find the meeting vertices among them. Our novel framework directly maintains a set of raw random walks for each vertex and computes the meeting vertices online efficiently. Through indexing the raw random walks, the new solution can handle dynamic graphs with high performance as well, which will be presented in Section 4. However the straightforward method for storing random walks consumes a large amount of memory, so we further design a special graph, **one-way graph**, to compactly maintain random walks. On top of the one-way graph, the top-k similarity search can be efficiently executed.

In the following subsections, we first introduce the novel SimRank estimation framework, which is called two-stage random-walk sampling framework. Then we present the basic concepts and properties of one-way graph. To reduce the cost of finding meeting vertices, we introduce an approximation random surfer-pairs model followed by the details of top-k similarity search based on the one-way graph. In the end, we analyze the error of approximation in TSF.

3.1 Overview of TSF

Two-stage random-walk sampling framework (TSF for short) is a new Monte-Carlo based approach for SimRank estimation. The core idea of TSF is to maintain a set of random walks for each vertex. These walks approximately represent each vertex's complete random-walk tree (Figure 2). At query time, TSF estimates SimRank via the indexed walks and a lot of redundant sampling can be avoided. Moreover, the indexed raw random walks can be updated efficiently when the original graph is modified.

Specifically, TSF applies the random-walk sampling technique in two stages, the preprocessing stage and query processing stage. The details of the two stages are as follows.

Preprocessing stage. In this stage, TSF aims at sampling R random walks for each vertex to represent the vertex's complete

Algorithm 2 Estimation of $s(u, v)$ in TSF

```
Input: reversed graph $G_r$, vertex pair $u$,$v$
Output: $s(u, v)$
1: for $i \leftarrow 0$ to $R_g - 1$ do
2:     $G^i_{owg} \leftarrow$ load the $i^{th}$ one-way graph;
3:     $w_v(v_0, v_1, ..., v_T) \leftarrow$ retrieve vertex $v$'s random walk from $G^i_{owg}$
       with $T$ steps;
4:     for $j \leftarrow 0$ to $R_q - 1$ do
5:         $u_e \leftarrow u$
6:         for $t \leftarrow 1$ to $T$ do
7:             $u_e \leftarrow$ randomly select a vertex from $N_O(u_e)$ in $G_r$
8:             /*Approximation Random Model*/
9:             if $u_e = v_t$ then
10:                $s(u, v) \leftarrow s(u, v) + c^t$
11:            end if
12:        end for
13:    end for
14: end for
15: return $\frac{s(u,v)}{R_g R_q}$
```

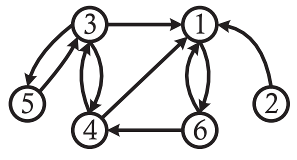
random-walk tree. However, it is expensive in both time and space if we generate and store all vertices' random walks separately. For instance, assuming a directed graph contains N vertices and each vertex samples R random walks of T steps, the total space cost will be $\mathcal{O}(NTR)$. In TSF, we use **one-way graph**, a novel compact data structure, as index. Instead of sampling random walks separately, TSF randomly samples R one-way graphs, where each graph compactly contains N coupled random walks and exactly one random walk for each vertex. The number of sampled one-way graphs is denoted by R_g and the details of one-way graph are described in the next subsection.

Query processing stage. Due to the compression of one-way graph, the random walks of different vertices in a one-way graph are coupled and dependent (refer to the sampling process of one-way graph in Section 3.2). To avoid the influence of this dependence and improve the accuracy of estimation, at query time in TSF, two query vertices u and v of a single pair are processed unequally. One vertex v simply uses the indexed random walks from one-way graphs while the other one u resamples some new random walks from the reversed graph G_r. For each random walk of vertex v, we sample R_q new random walks of vertex u to find the meeting vertices. The extra R_q samples help improve the accuracy of estimation. Moreover, sampling extra R_q random walks is an indispensable operation for top-k similarity search and helps distinguish close vertices more explicitly. The detailed approximation error of TSF will be described in Section 3.5.

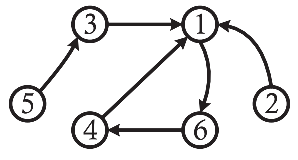
Algorithm 2 illustrates the detailed procedure of the query processing stage. At the beginning, TSF retrieves random walks of vertex v from R_g one-way graphs (Lines 2-3). For each random walk of vertex v, TSF samples R_q random walks for the other vertex u (Lines 4-13). TSF estimates the SimRank score $s(u, v)$ based on vertex v's indexed random walks and vertex u's newly sampled random walks by detecting the meeting vertices. To efficiently find all the meeting vertices at the same time, we further introduce an approximation random surfer-pairs model (Section 3.3).

3.2 One-Way Graph

In TSF, a one-way graph should contain a random walk for every vertex of the reversed graph G_r. The **one-way graph** is actually a graph satisfying the one out-degree restriction, which means each vertex has at most one outgoing edge. Figure 1(b) shows a one-way graph sampled from Figure 1(a). Given the one out-degree restriction, it easily figures out that each vertex has exactly one random walk which starts from that vertex. Therefore we can uniformly



(a) a reversed graph G_r



(b) a sampled one-way graph

Figure 1: An illustration of a G_r and a one-way graph.

sample one-way graphs instead of sampling random walks separately.

To sample a one-way graph, each vertex in the reversed graph randomly selects an outgoing edge from its neighbors and the N such edges compose a one-way graph. Through repeating above procedure R_g times, we can get R_g sampled one-way graphs, thus each vertex of G_r has R_g random walks. The procedure of building one-way graphs is described in Algorithm 3. The total time complexity of preprocessing is $\mathcal{O}(NR_g)$ and the space complexity is $\mathcal{O}(NR_g)$ as well.

Properties of one-way graph

Here we first introduce two properties related to the weakly connected component (WCC) in the one-way graph. Then the characteristics of random walks in a one-way graph are discussed.

Graphs	#WCC	W_{MS}	$\frac{N}{W_{MS}}$
livejournal	607,126	38,757	125.08
it-2004	5,044,878	29,664	1391.95
wikipedia	15,987,009	134,537	192.83
twitter	6,513,092	3,610,459	11.54

Table 2: Average number of WCC and average maximum WCC size (W_{MS}) from ten sampled one-way graphs in four real-world graphs are listed. The last column shows the ratio between the size of graph and W_{MS}.

Table 2 shows the statistics of weakly connected component of one-way graphs in four real-world graphs. It clearly shows that a one-way graph sampled from a real-world power-law graph consists of many weakly connected components. In twitter dataset, a one-way graph contains average 6,513,092 weakly connected components, and the average maximum size of WCC is around 3,610,459 which is an order smaller than the size of graph. This phenomenon concludes Property 1 and the property helps TSF filter unqualified vertices without any overhead (Section 3.4).

PROPERTY 1. *A one-way graph sampled from a real-world power-law graph is highly disconnected.*

Furthermore, considering the one out-degree restriction, the weakly connected component of a one-way graph also has Property 2.

PROPERTY 2. *For each weakly connected component in a one-way graph, there exists at most one cycle.*

Figure 1(b) is a weakly connected component with cycle consisting of vertices 1, 4 and 6. The cycle in a weakly connected component will incur dependence for a single random walk when the walk visits the same vertex more than once. However, in the SimRank estimation situation, this dependence has little influence on the results.

First, when a random walk obtained from a one-way graph did not visit any same vertex more than once, Lemma 1 guarantees that the walk is a randomly sampled walk in the reversed graph G_r and the walk is true independent.

LEMMA 1. *Given a walk $w_{v_1}(v_1, v_2, ..., v_k, v_{k+1})$ of a one-way graph, if $\forall v_i, v_j$, $i \neq j, 1 \leq i, j \leq k+1$, $v_i \neq v_j$ holds, then the marginal probability of the walk is*

Algorithm 3 Sampling one-way graphs (indexing)

```
Input: reversed graph $G_r$
1: for $i \leftarrow 0$ to $R_g - 1$ do
2:     $G^i_{owg} \leftarrow Empty$;
3:     for $v \in G_r$ do
4:         $v_r \leftarrow$ randomly select a vertex from $N_O(v)$ in $G_r$;
5:         $G^i_{owg}$.insertEdge($v$, $v_r$);
6:     end for
7:     store $G^i_{owg}$;
8: end for
```

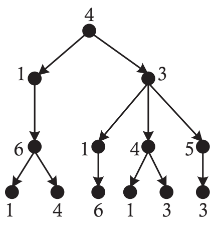
$$p(w_{v_1}(v_1, v_2, ..., v_k, v_{k+1})) = \sum_{i=1}^{k} \frac{1}{|N_O(v_k)|}.$$

PROOF. According to the one-way graph indexing process, where each vertex independently samples an outgoing edge from its neighbors, the marginal probability can be inferred directly. □

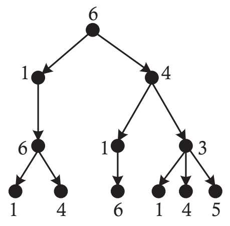
Second, let L be the length of a cycle in a weakly connected component and T be the length of a random walk required for SimRank estimation. If L is larger than T, each random walk in a one-way graph will not visit the same vertex more than once and they are true independent by Lemma 1. If L is smaller than T, some random walks are affected by the cycle. On basis of Property 2, only one cycle exists in a WCC. So in the worst case, at most $L + L * (T - L - 1)$ vertices have random walks influenced by the cycle. However, in the SimRank computation, T is usually small (∼10) [14, 21, 9, 15], therefore only a few vertices (50∼60) are influenced. The influence is further reduced by resampling R_q new random walks for one vertex at query time. In summary, the dependence caused by the cycle is negligible for the SimRank estimation in TSF.

According to the above discussion, it is reasonable to assume a random walk of a one-way graph is independent in TSF. The remained analysis in this paper is based on this assumption.

3.3 Approximation Random Model



(a) start from vertex 4



(b) start from vertex 6

Figure 2: Two random-walk trees within three steps in the reversed graph G_r.

In the random surfer-pairs model, all the possible random walks traversed by a random surfer u over G_r form a random-walk tree whose root is u. Figure 2 shows two random-walk trees within three steps. To (approximately) compute SimRank score, random-walk based solutions always enumerate or sample random walks from the tree, calculate the probability and find the first-meeting vertices between random walks. However, it is expensive to find the first-meeting vertices among multiple random walks simultaneously because all the walks need to be stored [20]. Therefore, we propose an approximation random surfer-pairs model which relaxes the constraints on meeting vertices. From now on, the **approximation random model** is short for the approximation random surfer-pairs model, and the **classic random model** refers to the original random surfer-pairs model.

The approximation random model no longer requires the *first-meeting* condition. This means two random surfers can meet more than once on their walks. For two walks $w_4(4,3,1,6)$ and $w_6(6,4,1,6)$ in Figure 2, only meeting vertex 1 counts in the classic random model while both meeting vertices 1 and 6 are considered in the approximation random model. Without the first-meeting constraint, the meeting vertices can be efficiently detected. The approximation random model is formalized as follows,

$$s_a(u,v) = \sum_{t=1}^{T} \sum_{w \in V} p_t(u,w) p_t(v,w) c^t I_w, \tag{2}$$

where $p_t(u,w)$ is the probability that random surfer u reaches vertex w after t steps.

Comparison of approximation and classic models

Theorem 2 shows that the SimRank score in the approximation random model is larger than the one in the classic random model, but the difference is bounded and will not be larger than $\frac{c}{1-c}\frac{I_{max}}{I_{min}}$ times of $s(u,v)$. The bound is estimated under the worst case assumption and is hardly reached for the real-world datasets in average. Moreover, both the upper bound $(1+\frac{c}{1-c}\frac{I_{max}}{I_{min}})s(u,v)$ and Equation 3 imply that the larger $s(u,v)$ the larger $s_a(u,v)$. This observation helps distinguish some close SimRank scores more explicitly in the approximation random model.

THEOREM 2. *Assuming $s_a(u,v)$ and $s(u,v)$ are the SimRank scores of a single vertex pair (u,v) in the approximation random model and the classic random model respectively, we have*

$$0 \le s_a(u,v) - s(u,v) \le \frac{c}{1-c}\frac{I_{max}}{I_{min}} s(u,v),$$

where $I_{max} = \max_{w \in V}\{I_w\}$ and $I_{min} = \min_{w \in V}\{I_w\}$.

PROOF. Let $s_{ft}(u,v,w)$ be the score of two random surfers, who start from vertices u and v respectively, first meeting at vertex w with t steps, i.e., $s_{ft}(u,v,w) = p_{ft}(u,v,w)c^t I_w$. In the approximation random model, after first meeting at vertex w, the surfers u and v will contribute extra similarity score $\delta_{T'}(w)$, where T' is $T-t$. $\delta_{T'}(w)$ is as follows,

$$\delta_{T'}(w) = \frac{s_{ft}(u,v,w)}{I_w} \sum_{i=1}^{T'} c^i \sum_{x \in V} p_i^2(w,x) I_x. \tag{3}$$

The total extra similarity of a single pair (u,v) in the approximation random model is

$$s_a(u,v) - s(u,v) = \Delta(u,v) = \sum_{t \le T; w \in V} \delta_{T-t}(w).$$

First, $\Delta(u,v) \ge 0$ because of $\delta_{T-t}(w) \ge 0$.

Second, because $\sum_{x \in V} p_i^2(w,x) \le 1$, we have

$$\delta_{T-t}(w) \le \frac{s_{ft}(u,v,w)}{I_{min}} \sum_{i=1}^{T-t} c^i I_{max} \le \frac{c}{1-c}\frac{I_{max}}{I_{min}} s_{ft}(u,v,w).$$

Finally,

$$\Delta(u,v) \le \sum_{t \le T; w \in V} \frac{c}{1-c}\frac{I_{max}}{I_{min}} s_{ft}(u,v,w)$$
$$= \frac{c}{1-c}\frac{I_{max}}{I_{min}} s(u,v).$$

□

Algorithm 4 Top-k similarity search in TSF

Input: reversed graph G_r, $q(v,k)$
Output: k vertices with the highest SimRank scores

```
1: for $i \leftarrow 0$ to $R_g - 1$ do
2:     $G^i_{rowg} \leftarrow$ load the reversed version of $i^{th}$ one-way graph;
3:     /*$tsMap$ is a map to record possible meeting vertices, where key is
       the vertex id and value is a list of meeting times ($tsList$).*/
4:     $tsMap \leftarrow$ Empty;
5:     $sArray \leftarrow 0$ // store the SimRank score for each vertex
6:     for $j \leftarrow 0$ to $R_q - 1$ do
7:         $u_e \leftarrow v$
8:         for $t \leftarrow 1$ to $T$ do
9:             $u_e \leftarrow$ randomly select a vertex from $N_O(u_e)$ in $G_r$.
10:            $tsList_{u_e} \leftarrow tsMap[u_e]$
11:            $tsList_{u_e}.add(t)$ //merge meeting time
12:        end for
13:    end for
14:    for $w \in tsMap.keySet()$ do
15:        updateSimRank($G^i_{rowg}$, $w$, $tsList_w$, $sArray$)
16:    end for
17: end for
18: return top-$k$ vertices in $sArray$
```

```
1: updateSimRank method
2: /*traverse $G^i_{rowg}$ starting from $w$*/
3: $t \leftarrow 0$
4: $queue \leftarrow w$
5: sort $tsList_w$ in ascending order
6: while $t < T$ do
7:     $t \leftarrow t + 1$
8:     $newQueue \leftarrow$ expanding all vertices in $queue$ one step
9:     if $t$ exists in $tsList_w$ then
10:        for vertex $u$ in $newQueue$ do
11:            $sArray[u] \leftarrow sArray[u] + c^t$
12:        end for
13:    end if
14:    $queue \leftarrow newQueue$
15: end while
```

3.4 Efficient Top-k Similarity Search

A simple approach of running top-k search query $q(v,k)$ in TSF is using Algorithm 2 as a basic procedure and iteratively calling the procedure to estimate N vertex pairs. The simple approach entails $\mathcal{O}(NR_qR_gT)$ time to execute a top-k search query and is really inefficient.

The properties of one-way graph open up opportunities to optimize the search algorithm. First, because of the one out-degree restriction, random walks of different starting vertices are highly overlapped in a single one-way graph. Thus we can compute different similarity pairs together instead of computing them separately. Second, if $s(v,u)$ wants to have a non-zero similarity gain from a one-way graph, vertex u, which uses the random walk in the one-way graph, and the meeting vertex must be in the same WCC of the one-way graph. According to Property 1, a one-way graph is highly disconnected in real-world graphs. Thus the connectivity of one-way graph can help the search algorithm prune unqualified vertices naturally and efficiently.

Based on above two observations, the new top-k search algorithm utilizes the connectivity of a one-way graph to efficiently compute SimRank scores of different vertex pairs in a single traversal. The algorithm first extracts the possible meeting vertices from R_q sampled random walks of query vertex v. Then for each possible meeting vertex w, it traverses on the reversed one-way graph to find other vertices u who meet with query vertex v at vertex w. Because of the traversal, vertices u and w are restricted in the same WCC, and the search algorithm successfully prunes the unqualified vertex pairs which are not in the same WCC with w. Finally, it increases SimRank scores of each vertex u. Note that, a meeting vertex with different meeting times from the R_q random walks can be efficiently processed in a single traverse as well. This is because there is no constraints on the meeting vertices in the approximation random model.

The details of top-k search procedure in TSF are illustrated by Algorithm 4. Figure 3 shows a concrete example which runs $q(4, 2)$ (i.e., $v = 4, k = 2$) on the G_r of Figure 1(a) with setting $R_g = 1$, $R_q = 2$ and $T = 2$. The one sampled one-way graph is shown in Figure 1(b) and the reversed one is shown in Figure 3. The algorithm first samples two random walks with two steps from G_r, $W_4(4, 3, 1)$ and $W_4(4, 1, 6)$, and records possible meeting vertices $1, 3, 6$ in $tsMap$ (Lines 6-13). In the $tsMap$, each meeting vertex is associated with its all possible meeting times, like vertex 1 can be met at times 1 or 2 based on the two sampled random walks. Next, each possible meeting vertex w traverses on the reversed one-way graph to find the valid vertex u that meets with the query vertex $v(= 4)$ at the possible meeting vertex w (Lines 14-16). Figure 3 illustrates the traversal of possible meeting vertex 1. We find vertices 2,3,5,6 are all the valid vertices for u. Finally, we increase the corresponding vertices' SimRank scores based on the meeting time. For example, vertex 4 and vertex 6 meet at vertex 1 at time 2, so $s(4, 6)$ is increased by c^2.

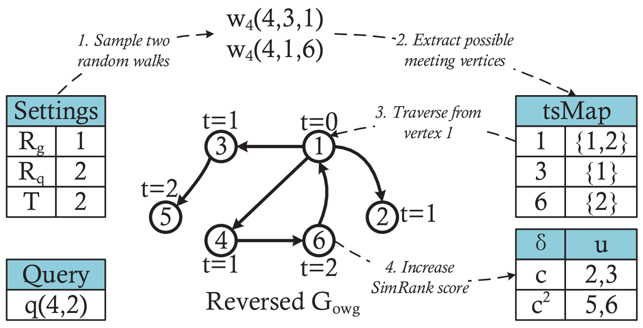


Figure 3: An example of top-k search

Complexity analysis. The top-k search in TSF is very efficient. Theorem 3 shows that TSF can averagely estimate N-pairs SimRank scores in $\mathcal{O}(R_qR_gT^2)$ time which is independent of the size of graph.

THEOREM 3. *In TSF, the average computation time of estimating N-pairs SimRank scores for a top-k search query is bounded by $\mathcal{O}(R_qR_gT^2)$. The query time is bounded by $\mathcal{O}(R_qR_gT^2 + Nlog(k))$.*

PROOF. First, let's analyze the cost of **updateSimRank** method. The method traverses over a reversed one-way graph within T steps. Assume the average out-degree in the reversed one-way graph is $\bar{d}$, then the total number of visited vertices is $\sum_{i=1}^{i=T} \bar{d}^i$. Based on the one out-degree restriction of one-way graph, we know that $\bar{d} = 1$. Therefore, the cost of **updateSimRank** method is $\mathcal{O}(T)$.

For each single one-way graph, the algorithm takes $\mathcal{O}(R_qT)$ times to sample R_q random walks and generate $tsMap$. In the worst case, $tsMap$ contains R_qT different vertices, thus the **updateSimRank** will be called at most R_qT times. So the cost of processing a single one-way graph is $\mathcal{O}(R_qT^2)$.

Therefore, the total cost of computing N-pairs SimRank socres is $\mathcal{O}(R_gR_qT^2)$. Finally, it entails extra $\mathcal{O}(Nlog(k))$ time to find the top k vertices based the SimRank scores. The query is finished in $\mathcal{O}(R_qR_gT^2 + Nlog(k))$ time. □

3.5 Error of Approximation

In TSF, parameters R_g and R_q have crucial influences on the performance. The size of index, the preprocessing time and the query time are all linear to these two parameters. We next formally analyze the R_g and R_q needed for a given approximation error bound. The theorems show that small R_g and R_q are sufficient to achieve a reasonable accuracy. Furthermore, R_g determines the global approximation bound while R_q helps improve the accuracy further only with a slight overhead.

TSF involves two stages of random-walk sampling. In the following analysis, the sampling in the preprocessing stage is called first-stage sampling while the one in the query stage is second-stage sampling.

Let $s_b(u, v)$ be the most accurate similarity which can be obtained from a given set of one-way graphs. Once the index (one-way graphs) is built in the preprocessing stage, the $s_b(u, v)$ is determined as well. Let $\phi_{u,v}(R_g)$ be the difference between $s_b(u, v)$ and $s(u, v)$, i.e., $\phi_{u,v}(R_g) = s(u, v) - s_b(u, v)$. Then $\phi_{u,v}(R_g)$ is the best approximation error. It bounds the accuracy of query processing. We call $\phi_{u,v}(R_g)$ the global accuracy.

THEOREM 4. *For a given error bound ϵ, the probability that global accuracy $\phi_{u,v}(R_g)$ exceeds ϵ is bounded as follows.*

$$Pr\{|\phi_{u,v}(R_g)| > \epsilon\} \le 2e^{-2\epsilon^2 \frac{R_g}{(1-c)^2}}.$$

PROOF. Suppose X_i is the estimated delta SimRank score brought by the random walk W_{ui} in i^{th} one-way graph. The most accurate X_i can be calculated by finding all the meeting vertices between W_{ui} and the complete random-walk tree of vertex v. The random variable X_i is still limited by range $[c, 1]$. Therefore, the above inequality can be derived via Hoeffding's inequality. □

Theorem 4 shows the probability that $|\phi_{u,v}(R_g)|$ exceeds ϵ is bounded. Increasing R_g helps enhance the global accuracy. The global accuracy is only related to the first-stage sampling, and the preprocessing determines the global accuracy of the SimRank score estimation.

As mentioned before, the computation of $s_b(u, v)$ depends on the complete random-walk tree of vertex v. Thereupon, in order to approximate the global accuracy bound, we bring in the second-stage sampling which is responsible for sampling random walks from the vertex v's random walk tree. Theorem 5 guarantees we can achieve better approximation by simply increasing R_q, but it cannot get the accurate result even if increasing R_q to the positive infinity since $\phi_{u,v}(R_g)$ is not influenced by R_q.

THEOREM 5. *Let $s_b^{'}(u, v)$ be the estimated $s_b(u, v)$, then*

$$Pr\{|s_b^{'}(u, v) - s(u, v) + \phi_{u,v}(R_g)| > \epsilon\} \le 2e^{-2\epsilon^2 \frac{R_gR_q}{(1-c)^2}}$$

PROOF. Assume that random variable X_{ij} $(1 \le i \le R_g; 1 \le j \le R_q)$ is the SimRank score computed between vertex u's i^{th} random walk and vertex v's j^{th} random walk which is sampled during processing the i^{th} one-way graph, and $X_{ij} \in [c, 1]$, then

$$s_b^{'}(u, v) = \sum_{i=1}^{R_g} \sum_{j=1}^{R_q} X_{ij}.$$

According to Hoeffding's inequality

$$Pr\{|s_b^{'}(u, v) - s_b(u, v)| > \epsilon\} \le 2e^{-2\epsilon^2 \frac{R_gR_q}{(1-c)^2}}. \quad (4)$$

Finally, the result can be derived by replacing $s_b(u, v) = s(u, v) - \phi_{u,v}(R_g)$. □

Based on Theorem 5, increasing R_q helps improve the accuracy of top-k search. In one-way graphs, random walks of different vertices are highly coupled, therefore many vertices have close SimRank score when R_q is small. With the increasing of R_q, each vertex has a more accurate estimation and distinguishes itself from others. A large R_q but still much smaller than R_g can improve the accuracy of top-k similarity search, see the experimental results in Section 5.3.

Above two theorems are analyzed under the classic random model. The results can be directly extended to the approximation random model by using the range of random variable determined in Theorem 2.

4. TSF ON LARGE DYNAMIC GRAPHS

One-way graph is the core component in TSF. During the query processing, TSF will traverse one-way graphs to retrieve random walks and find meeting vertices. The one-way graph, first of all, should be well organized in the memory for fast traversal. So TSF applies a simple version of CRS format[1] in memory. It sorts the directed edges in a one-way graph by their source id in ascending order and then stores the starting position for each source id as an index array. The process can be finished in $\mathcal{O}(N)$ time with counting sort [7], because the number of edges will not be larger than the number of vertices in a one-way graph.

Moreover, given the fact that the real-world graphs are dynamic and large-scale, one-way graphs should be not only efficiently organized in memory, but also adaptive to the graph modification and compactly stored on the disk when R_g one-way graphs are too large to fit in the memory. Thereby, TSF truly has the ability to handle large dynamic graphs.

In the following subsections, we first elaborate the updating semantics on one-way graph followed by its log-based implementation, and then we discuss how to compactly store one-way graphs on disk to save I/Os.

4.1 Semantics of One-way Graph Updating

Unlike other random-walk based solutions [9, 15], the one-way graph in TSF can be efficiently updated when the reversed graph G_r is modified. This benefits from two factors. First, one-way graph directly indexes the raw random walks, so that the relation between the one-way graph and the G_r is explicitly preserved. Second, one-way graph is generated through independent vertex sampling. As a result, whenever a vertex or an edge of G_r is modified, only one related vertex and its neighbors need to be updated in the one-way graph. We next explain the semantics of one-way graph updating when an edge or a vertex is modified in the reversed graph G_r.

Edge modification in G_r. Every time a directed edge (v, u) is inserted into or deleted from G_r, only the changes of vertex v's outgoing edges in G_r affect the sampled one-way graphs. This is because the number of outgoing edges influences the probability of sampling. Therefore, to update the vertex v in one-way graphs, we only need to resample an edge from the updated outgoing edges of vertex v. The time complexity for updating R_g one-way graphs is $\mathcal{O}(R_g)$. Figure 4(a) shows an example where an edge $(4, 5)$ is inserted into G_r. In Figure 4(b), vertex 4 in the one-way graph resamples an edge $(4, 3)$ from the updated outgoing edges of vertex 4 of G_r, and updates the one-way graph by deleting edge $(4, 1)$ and inserting edge $(4, 3)$.

[1]Compressed Row Storage (CRS), http://netlib.org/linalg/html_templates/node91.html

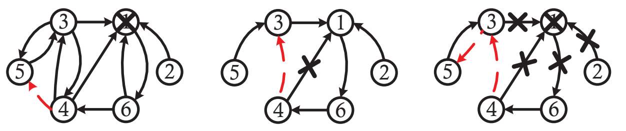

(a) update $\mathbf{G}_r$ (b) result of $(4, 5)$ inserted in $\mathbf{G}_r$ (c) result of vertex 1 deleted in $\mathbf{G}_r$

Figure 4: (a) Edge $(4, 5)$ is inserted and vertex 1 is deleted in reversed graph G_r; (b) Edge $(4, 1)$ is deleted and $(4, 3)$ is inserted in the one-way graph after vertex 4 resamples its new outgoing edge. (c) Related vertices and edges change in the one-way graph after vertex 1 is deleted.

Vertex modification in G_r. A single vertex modification can be divided into a series of edge modifications. When one vertex v is modified, the overall influences can be interpreted as follows. Each source vertex of vertex v's incoming edges has an edge modification and the vertex v has successive $|N_O(v)|$ edge modifications. To update a one-way graph, all the source vertices of vertex v's incoming edges require to resample outgoing edges, and vertex v is deleted as well as its outgoing edge. We explain the above operations through an example. In Figure 4(a), when vertex 1 is deleted in G_r, first vertex 1 and its outgoing edge $(1, 6)$ in the one-way graph are deleted. Then vertices $2, 3, 4, 6$ resample their outgoing edges. A possible result is illustrated in Figure 4(c) where the outgoing edge of vertex 6 remains unchanged after resampling.

4.2 Log-based Implementation

As mentioned at the beginning of this section, the static one-way graph is organized in the CRS format. Whenever the one-way graph is updated, it is expensive to reorganize the one-way graph repeatedly. Therefore, we develop a log-based solution to implement aforementioned semantics of one-way graph modification. The core idea of log-based solution is similar to the log buffer tree [2]. It maintains logs in memory with buffers, and reorganizes the one-way graph through the counting sort when the number of logs exceeds a predefined threshold.

Definition of Logs

We first define three types of logs that are used to record the modification of one-way graph.

- `L(u, v)`: the log that vertex u has a new outgoing edge (u, v) in a one-way graph. v is `EMPTY` when vertex u has zero outgoing edge.
- `L(u, DELETE)`: the log of deleting vertex u.
- `L(u, INSERT)`: the log of adding a new vertex u.

Based on the definition of logs, for a one-way graph, each edge modification corresponds to a single `L(u,v)`. Since the vertex in a one-way graph has exactly one outgoing edge, the `L(u,v)` also implies that the existing edge (u, v') is out of date. We do not need additional log to delete the existing edge. In the vertex (u) modification case, it generates a `L(u, DELETE)` or `L(u, INSERT)` with a set of `L(w,v)`s where vertex w is the source of incoming edges adhered to vertex u. In summary, the above three logs are sufficient for recording the one-way graph update in accordance with the graph modification.

Log Management

We proceed to discuss the organization of logs in memory. We use a log buffer to store raw logs. Given the one out-degree restriction in one-way graph, logs for the same vertex u can be overwritten ahead according to the updating-time order in the log buffer. This means the log buffer always stores the latest log for a single vertex.

Since the top-k similarity search relies on the reversed one-way graph (updateSimRank method in Algorithm 4), we also need to maintain the reversed logs (e.g., `L(v,u)` is the reversed version

Algorithm 5 Explore neighbors on a updated one-way graph

```
Input: vertex $v$, $G_{rowg}$, log buffer, reversed log buffer
Output: vertex $v$'s outgoing neighbors $nSet$
 1: $nSet \leftarrow Empty$
 2: if L(v, DELETE) exists then
 3:     return $nSet$;
 4: end if
 5: $nSet \leftarrow$ retrieve neighbors of vertex $v$ from $G_{rowg}$
 6: $nSet \leftarrow nSet \cup$ {neighbors of vertex $v$ in reversed log buffer}
 7: for $u \in nSet$ do
 8:     if (L(u,v') exists and $v' \neq v$) or (L(u, DELETE) exists) then
 9:         delete $u$ from $nSet$
10:     end if
11: end for
```

of L(u,v)). Then the updated reversed one-way graph can be explored efficiently. We bring in a reversed log buffer to store the reversed logs. For the reversed logs, the one outgoing degree restriction does not exist any more. In order to logically cluster the reversed logs related to the same vertex together, the reversed log buffer stores the reversed logs of a vertex as a link-list.

Finally, with the help of log buffer and reversed log buffer, TSF can explore neighbors of a vertex v on the updated reversed one-way graph as Algorithm 5 depicts.

4.3 External Storage Format of One-way Graph

Although the memory cost of a single one-way graph is $\mathcal{O}(N)$, when R_g and the size of graph increase, it is still expensive to store all the R_g one-way graphs in a memory-limited environment. For the twitter dataset, a single one-way graph consumes about 296MB, but for 100 and 1000 one-way graphs, they will occupy 29GB and 290GB memory respectively.

According to the top-k similarity search algorithm, at query time, TSF processes one-way graphs one by one. Hence TSF only needs to load a single one-way graph into memory for computation, and other one-way graphs can be stored on disk and loaded on demand. In order to save I/Os, we compactly store each one-way graph on the disk through variable-length quantity [3] and run-length encoding [4]. These two techniques help compress one-way graph efficiently with retaining high decompressing performance.

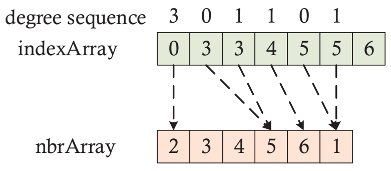


Figure 5: In-memory representation of a reversed one-way graph

The in-memory compact format of reversed one-way graph consists of two arrays, which are indexArray and nbrArray. The indexArray records the starting index of a vertex's outgoing neighbors which are stored in nbrArray. Figure 5 shows the in-memory compact format of the reversed one-way graph in Figure 3. In fact, the indexArray can be implicitly represented by vertices' outdegrees.

Given the fact that real-world graph has a power-law degree distribution, the reversed one-way graph also has a power-law outgoing degree distribution. The distribution guarantees that most vertices have a small degree and a lot of vertices have the same degree with successive vertex id. Hence we can efficiently compress the indexArray through the corresponding degree sequence. First the degree sequence is extracted from the indexArray, then we encode the degree sequence by run-length encoding, and at last the small numbers in run-length encoding are further compressed by the variable-length quantity. The nbrArray is not compressed since each element is totally different. Furthermore, the nbrArray usually has a smaller size than the indexArray since G_r has many zero outdegree vertices.

5. EXPERIMENTAL EVALUATION

We present the evaluation results of TSF in this section. The advantages of TSF, such as efficient preprocessing, fast query processing, dynamic graph supporting and external storage supporting, are demonstrated through extensive experimental study.

5.1 Experiment Environment

The experiments are evaluated on a commodity computer which has two Xeon(R) E5530@2.40GHz CPUs and 96GB memory. The operation system is Ubuntu 12.04.4 LTS. All the algorithms are implemented in C++ and compiled by g++ 4.6.3 with -O3 option.

Scale	Graphs	#vertex	#edge
Small	ca-grqc (CG)	5,242	28,980
	ca-hepth (CH)	9,877	51,971
	wiki-vote (WV)	7,115	103,689
Medium	web-google (WG)	875,713	5,105,039
	web-berkstan (WB)	685,230	7,600,595
	dblp-2011 (DB)	986,324	6,707,236
	livejournal (LJ)	4,847,571	68,993,773
Large	wikipedia (WK)	25,942,246	601,038,301
	it-2004 (IT)	41,290,682	1,150,725,436
	twitter (TW)	41,652,230	1,468,365,182

Table 3: Statistics of graphs used in the experiments.

Datasets. We conduct our experiments on ten real-world graphs, whose statistics are described in Table 3. All graphs are available on SNAP, KNOECT and LWA websites.

Baselines. We compare our proposal with three state-of-the-art baselines which are random-walk based solutions of SimRank similarity search. Two of the baselines are Monte-Carlo based algorithms, FR algorithm [9] and KM algorithm [15]. FR algorithm, introduced by Fogaras and Rácz, runs the similarity search based on fingerprints. KM algorithm pre-computes similarity bounds for speeding up query processing. The third baseline is TopSim algorithm [16]. The TopSim algorithm enumerates all the random walks with heuristically pruning the walks. The three baselines are all in-memory solutions. In this paper, our TSF has two variants, **MEM-TSF** and **EXT-TSF**. The MEM-TSF holds all the index in memory while EXT-TSF stores indexes on disk and loads the index on demand. Furthermore, the TSF uses the approximation random model to estimate the SimRank score.

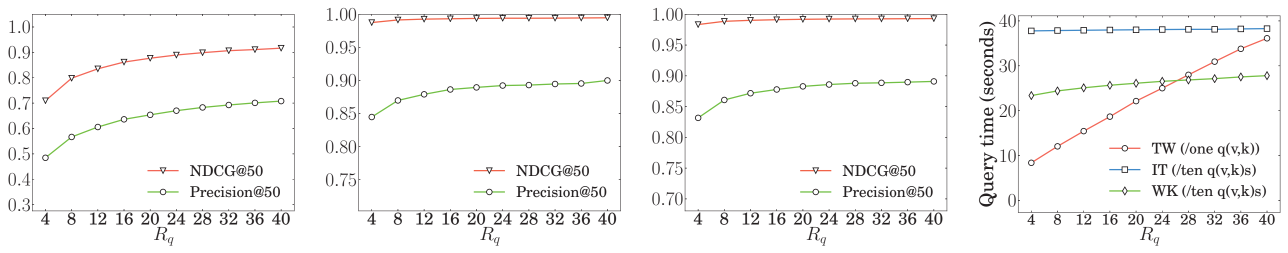
Performance metrics. We evaluate both efficiency and effectiveness of different approaches in the experiments. In terms of efficiency, we report the time cost of index building and query processing, as well as the space cost of index. On the other hand, we use precision and Normalized Discounted Cumulative Gain (NDCG) as effectiveness metrics to evaluate the quality of returned top-k vertices. The precision is defined as follows,

$$precision@k = \frac{|\{retrieved\ topk\} \cap \{exact\ topk\}|}{k},$$

where $\{retrieved\ topk\}$ and $\{exact\ topk\}$ are k vertices returned by an approximation algorithm and the exact algorithm respectively. The NDCG [18] is

$$NDCG@k = \frac{1}{Z_k} \sum_{i=1}^{k} \frac{2^{s_i} - 1}{log_2(i+1)},$$

where s_i is the exact SimRank score of vertex at rank i and Z_k is a normalization factor to ensure the exact ranking generate $NDCG@k$ equals 1.



(a) Effectiveness on wiki-vote (b) Effectiveness on ca-grqc (c) Effectiveness on ca-hepth (d) Query time

Figure 6: The performance of TSF with respect to different R_q.

SimRank				KM and FR Alg.	TSF	
c	T	I_w	k	R	R_g	R_q
0.6	10	1.0	50	100	100	20

Table 4: Parameter settings.

Parameter settings. FR algorithm, KM algorithm and our TSF are all based on Monte-Carlo sampling technique. The parameters R and R_g are equivalent. We set R_g and R as 100, which is suggested by [15]. R_q is a unique parameter of TSF and it is set to 20 by making a trade off between the effectiveness and efficiency of TSF. All the parameters used in most of the experiments are listed in Table 4. Some experiment-specific parameters will be presented in the corresponding experiments.

5.2 Selection of R_q

Theorem 5 shows a larger R_q leads to a more accurate result, but the R_q directly derived from Equation 4 for a given error bound is still too loose. The larger R_q will also cause TSF to spend more time on query execution. To determine a reasonable R_q, we conduct the following experiments.

Three small datasets, ca-grqc, ca-hepth and wiki-vote, are used in the experiments of effectiveness evaluation, since any exact solution only works well on small datasets. For each dataset, we use the exact solution (i.e., Lizorkin's algorithm [21]) to generate the perfect top-k results, where the smallest SimRank score is larger than 0.0001.

Figures 6(a), 6(b) and 6(c) illustrate the variation of precision@50 and NDCG@50 with respect to the R_q. With the increasing R_q, the precision and NDCG are both improved. But the real improvement is related to the characteristics of datasets. In ca-grqc and ca-hepth datasets, when R_q is four for the top-50 query, both NDCGs are about 98% while the precisions are 84% and 83%. This indicates that the two datasets have many similar vertices. Furthermore, in Figure 6(d), we show the query time of MEM-TSF for different R_q over twitter, wikipedia and it-2004 datasets. It is easy to observe that the query time is linear to the R_q. But the overhead of increasing R_q is dependent on the size of WCC in one-way graphs (Table 2). The larger WCC incurs the more overhead. In twitter datasets, changing R_q from 4 to 40, each query spends extra 28 seconds, while it only takes 4 more seconds in wikipedia for every ten queries.

By weighing the quality against the efficiency, we choose R_q to be 20, when R_g is 100. At this point, TSF obtains top-k vertices in proper quality with efficient query processing.

5.3 Comparison with FR and KM Algorithms

In this subsection, we compare MEM-TSF with FR and KM algorithms. All the three algorithms use Monte-Carlo technique to estimate SimRank scores. The experiments demonstrate that MEM-TSF can efficiently process billion-edge graphs. Compared with FR and KM algorithms, MEM-TSF not only exhibits the highest efficiency in terms of preprocessing and querying, but also has

Graphs	MEM-TSF		FR Alg.		KM Alg.	
	Index	Query	Index	Query	Index	Query
WB	4.89	0.074	30.10	0.069	64.92	0.400
WG	7.07	0.067	39.79	0.090	68.52	0.204
DB	8.27	0.091	49.59	0.096	139.43	0.187
LJ	69.22	0.48	306.79	0.58	999.39	2.69
WK	294.06	2.57	992.82	2.98	779.76	50.32
IT	519.65	3.80	2163.49	4.33	4733.67	40.46
TW	830.17	23.03	2641.32	5.32	10089.09	159.64

Table 5: The performance (seconds) of MEM-TSF, FR and KM algorithms on graphs in different sizes.

a better precision and NDCG of the top-k vertices than the other two methods.

Preprocessing efficiency. First, the results of different algorithms' preprocessing efficiency are presented. The experiments focus on the medium and large graphs. Figures 7(a) and 7(b) show the time ratio and space ratio of preprocessing. The ratio is computed by normalizing the cost to the minimal cost on the same dataset. The actual time for building index of three algorithms is summarized in Table 5.

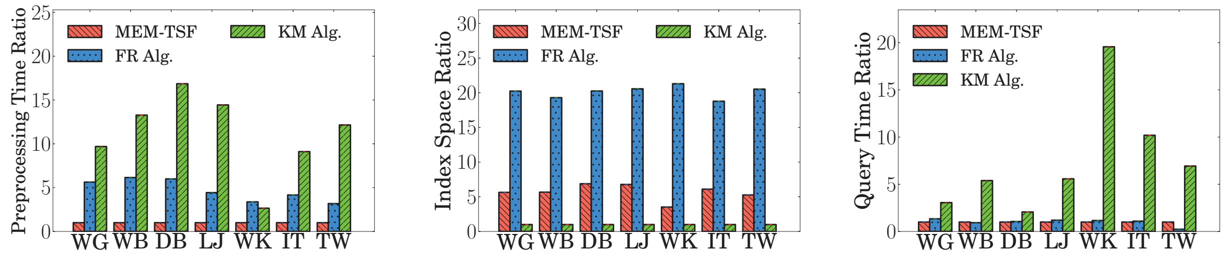
From both figures, we can clearly see that the speed of building index by TSF is constantly an order faster than KM algorithm and five times faster than FR algorithm. On twitter dataset, TSF only takes 830.17 seconds while KM and FR algorithms need 2.8 and 0.7 hours respectively. This is because the time complexity of preprocessing in TSF is $\mathcal{O}(R_gN)$ while it is around $\mathcal{O}(RNT)$ in KM. Although the FR algorithm has close time complexity to TSF, it has a much larger constant. FR algorithm spends time on extra computation like identifying each vertex's fingerprint id.

At the aspect of space cost, TSF has a middle performance among the three algorithms. It occupies approximately five times larger space than KM, but four times smaller space than FR on various datasets. On the twitter dataset, KM generates 3.8G indexes while the indexes of TSF are 20G. This is because KM highly aggregates the properties from random walks while TSF directly indexes the random walks. However, through directly indexing random walks, TSF empowers itself to efficiently handle large dynamic graphs.

To summary, the preprocessing in TSF is efficient, and TSF is also scalable to process graphs in different sizes.

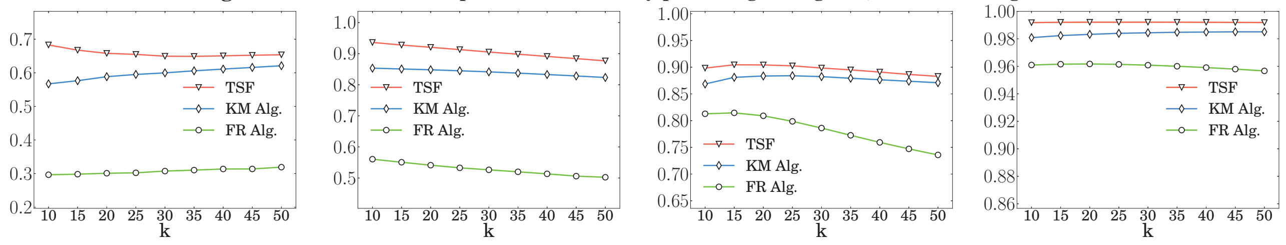
Query efficiency. Next we evaluate the query efficiency of the aforementioned three similarity search algorithms based on their indexes. For each graph, we generate a set of top-50 queries. The vertices in the queries are selected according to the in-degree distributions of the graph. More specifically, assume Q_n queries need to be generated, let V_n be the total number of non-zero in-degree vertices and N_d is the number of vertices whose in-degree is d, then there are around $\frac{N_d}{V_n}$ of Q_n vertices with in-degree d which is uniformly selected from N_d vertices. The query performance is measured by the average query time of all the queries.

Table 5 show the query performance on seven datasets. We also present the query time ratio, which is calculated by normalizing the time to the one of MEM-TSF, in Figure 7(c). First, all the three algorithms are efficient to execute similarity search on large graphs.



(a) Time of building index (b) Space of indexes (c) Performance of queries

Figure 7: Performance comparison of in-memory processing among TSF, KM and FR algorithms.



(a) Precision on wiki-vote (b) NDCG on wiki-vote (c) Precision on ca-hepth (d) NDCG on ca-hepth

Figure 8: Effectiveness comparison among TSF, KM and FR algorithms.

Second, we notice that the query efficiency of FR algorithm and MEM-TSF surpasses the KM algorithm. This is because both FR algorithm and TSF filter the unqualified vertices by the connectivity of indexes (one-way graphs and fingerprints), while KM algorithm prunes vertices by several bounds which is under assumption that the returned vertices have a high SimRank score. For the top-50 similarity search, the SimRank score of the lowest rank vertex is small, therefore, KM algorithm loses its efficiency. On the twitter dataset, MEM-TSF is slower than FR algorithm, where the average query time of two algorithms are 23 seconds and 5 seconds respectively. This is because twitter has a large WCC in one-way graphs (refer to Table 2) and MEM-TSF requires an additional R_q (=20) samples to compute on WCCs for improving the quality of results while FR algorithm does not.

Effectiveness of TSF. Finally, we evaluate the effectiveness of TSF by comparing to the KM and FR algorithms through running top-k queries with different k. The experimental results show that TSF has a better precision and NDCG than the KM and FR algorithms for the various top-k queries. Since the exact top-k results only can be obtained in small graphs, three small datasets, ca-grqc, ca-hepth and wiki-vote, are used in the following experiments. The vertices for queries are the same with the one in experiments of Section 5.2. The parameters of each algorithm are listed in Table 4.

Figure 8 only visualizes the average precision and NDCG on ca-hepth and wiki-vote datasets for figure's clarity, and the results on ca-grqc is similar to the one on ca-hepth. From the figures, we clearly see that the quality (i.e., precision and NDCG) of top-k vertices returned by TSF is constantly higher than the other two by changing k from 10 to 50. The superiority of TSF benefits from its two-stage random-walk sampling framework. With the extra R_q(=20) samples, TSF can obtain a better precision and NDCG without incurring significant overhead. For KM and FR algorithms, the only way to improve the quality of results is to increase R, which brings about heavy overhead.

5.4 Comparison with TopSim Algorithms

TopSim based algorithms [16] enumerate all the random walks of a certain length to find meeting vertices. To improve the performance, the authors proposed two approximation algorithms, Trun-TopSim-SM and Prio-TopSim-SM. The Trun-TopSim-SM prunes the random walk which contributes small delta SimRank score. While the Prio-TopSim-SM only selects and expands top-H (e.g., H=100) random walks at each level.

Since TopSim based algorithms and TSF use the different framework, they have different parameters. We first conduct experiments, which are similar to the experiments in Section 5.2, on small graphs to determine the parameters of TopSim-based algorithms and TSF. The length of random walk (T) in TopSim is set as 4. In TSF, R_g and R_q are 300 and 40 respectively. With these parameters, two algorithms have a similar precision and NDCG. On the ca-hepth dataset, both algorithms achieve about 94%∼98% precision and 99% NDCG with different k.

Method	WG	WB	DB	LJ	WK	IT	TW
Trun-TopSim-SM	0.117	0.712	15.61	355.0	442.5	248.2	OOM[2]
Proi-TopSim-SM	**0.059**	**0.094**	1.641	38.05	**2.039**	**1.802**	41213.2
MEM-TSF	0.074	0.148	**0.199**	**0.463**	2.261	3.054	**187.6**

Table 6: The performance (seconds) comparison of MEM-TSF and TopSim based algorithms.

Table 6 lists the performance comparison of MEM-TSF and TopSim based algorithms with previously determined parameters on large graphs. We clearly see that MEM-TSF outperforms Trun-TopSim-SM on different graphs. Compared to Proi-TopSim-SM, TSF performs significantly better on social graphs (e.g., DB, LJ and TW datasets), and yields comparable performance on web graphs. As mentioned before, Proi-TopSim-SM only expands H random walks at each level, thus its performance is heavily determined by choosing the H random walks at each level. Because social graphs always have denser local structures than web graphs, Prio-TopSim-SM spends much more time to generate the top-H random walks when processing social graphs. However, TSF samples a fixed number ($R_g = 300$, $R_q = 40$) of random walks, the local structure has little influence on it. TSF has relatively stable performance across social graphs and web graphs.

5.5 Performance of TSF on Dynamic Graph

To show the advantages of TSF in processing large dynamic graphs, we conduct the following experiments on four graph datasets,

[2]OOM is Out of Memory error.

i.e., livejournal, wikipedia, it-2004 and twitter. The results demonstrate that TSF can efficiently update the indexes when the original graph is modified, and retain the high performance of running top-k queries on dynamic graphs.

Since the node modification can be divided into a serial of edge modifications (details in Section 4.1), the experiments are concentrated on the edge modification. We set the size of log buffer to 1000 for the experiments. To generate updating logs, we randomly modify a certain number of edges for each real-world graph. Given the fact that in the real-world graphs, like social networks, the number of insertions are more than the one of deletions, so the edge modification consists of 80% edge insertions and 20% edge deletions in our experiments. Considering that the log buffer is full when 1K edge modifications are executed, during the experiments, TSF repeatedly executes the following procedure. It first successively runs 1K updating queries and then tests the query performance of TSF on the dynamic graph with the full log buffer. At last the logs are merged into one-way graphs. The log-based implementation of TSF is denoted by L-TSF.

Method		LJ	WK	IT	TW
L-TSF	$C_u/10^3$	0.101	0.168	0.198	0.174
	$C_m/10^2$	50.62	197.76	391.77	524.14
TSF (rebuild)		69.22	294.06	519.65	830.17
FR Alg. (rebuild)		306.79	992.82	2163.50	2641.32
KM Alg. (rebuild)		999.38	779.76	4733.67	10089.09

Table 7: Updating cost (seconds) comparison between log-based TSF and other solutions with building the index from scratch. $C_u/10^3$ is the cost of executing 1000 edge modifications and $C_m/10^2$ is the cost of merging 100 one-way graphs' full log buffers.

Table 7 shows the updating performance of L-TSF by comparing with the performance of rebuilding index from scratch over four large graphs. We clearly see that a single edge modification can be updated in about 0.2 millisecond, since this operation can be finished in $\mathcal{O}(1)$ time. Including the merge operation, the amortized updating cost of L-TSF for 100 one-way graphs is less than six seconds for processing one-billion edge graphs, like twitter and it-2004. Furthermore, the merge operation can be executed faster than any solutions of building index from scratch as well.

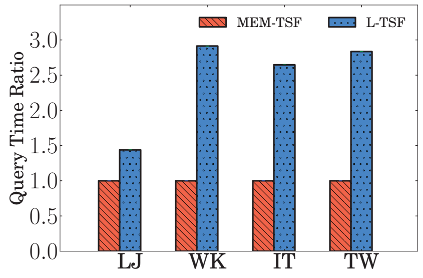


Figure 9: Query efficiency comparison between MEM-TSF and L-TSF.

Figure 9 describes the query efficiency between MEM-TSF and L-TSF. Here the query time only includes the cost of computing SimRank scores (Lines 6-17 in Algorithm 4) and the cost of sorting scores are omitted. The query time is also normalized for the clarity of figure. The L-TSF runs on the graph with logs while MEM-TSF runs over the graph which is obtained by merging the logs. From the figure, we notice that L-TSF is only 2∼3 times slower than MEM-TSF over different large graphs. In twitter dataset, L-TSF runs a query in average 50 seconds while MEM-TSF takes about 17 seconds. The extra cost is caused by searching vertex's neighbors in the logs (Algorithm 5) during the **updateSimRank** process.

5.6 Performance of External TSF

We next show that one-way graphs can be compactly stored on disk with the simple compression solution described in Section 4.3. Based on the compact form of one-way graphs, TSF can still handle large graphs when the available memory is limited.

Graph	Raw Storage	Compact Storage	Compression Ratio
LJ	3.5G	3G	1.17:1
WK	14G	8.1G	1.73:1
IT	31G	25G	1.24:1
TW	29G	20G	1.45:1

Table 8: The storage of 100 one-way graphs.

We first present the four large graphs' storages of the raw one-way graphs and the compacted ones in Table 8. From the table, we can figure out that the compression ratio of our proposal is around 1.5. For the wikipedia dataset, the ratio reaches 1.73. The different ratios among datasets are caused by the characteristics of corresponding degree sequence.

Method	LJ	WK	IT	TW
MEM-TSF	0.48	2.57	3.80	23.03
EXT-TSF (Raw)	33.19	129.47	282.60	289.33
EXT-TSF (Compact)	30.44	83.13	247.30	221.37
EXT-TSF Impr. Ratio	8.3%	35.8%	12.5%	26.2%

Table 9: Query time (seconds) of EXT-TSF with different external storage format. MEM-TSF shows the in-memory processing cost.

We further compare the performance of EXT-TSF with different external storage format. The results are summarized in Table 9. Compared to MEM-TSF, the query time of EXT-TSF is dominated by the cost of loading one-way graphs in memory. By using the compact external storage format, EXT-TSF achieves about 10% to 30% improvement of the query performance.

6. RELATED WORK

Since SimRank was first introduced by Jen and Widom [14] in 2002, there have been a lot of research works [21, 19, 26, 18, 30, 28, 10, 12, 6, 27] to optimize the computation of SimRank. The proposed solutions can be classified into three categories. They are exact solution of fix-point iteration, low-rank approximation solution and random-walk based approximation solution.

The authors [14] introduced the first exact solution of fix-point iteration. The solution computes the all-pairs SimRank scores with $\mathcal{O}(kd^2N^2)$ time in k iterations, where d is the average in-degree. Later, Lizorkin et al. [21] improved the original solution via partial sum memorization to $\mathcal{O}(kdN^2)$ time. Yu et al. [26] used fast matrix multiplication to speed up the all-pairs SimRank computation as well. Recently, Yu et al. [28] further enhanced the SimRank computation to $\mathcal{O}(kd'N^2)$ time (with $d' < d$) through fine-grained memorization. For computing a single-pair SimRank score, Li et al. [20] developed a method by iteratively computing the position matrix in $\mathcal{O}(kd^2\min\{d^k, N^2\})$ time. All these solutions require $\mathcal{O}(N^2)$ space. Due to the high time and space complexity, they cannot support similarity search on large graphs.

Low-rank approximation techniques are one of good strategies to approximately compute the SimRank scores for similarity search. Li et al. [18] proposed a novel non-iterative matrix formulation for SimRank and pre-compute four auxiliary matrices via Kronecker product [11] and SVD in $\mathcal{O}(r^4N^2)$ time, where r is the target rank of transpose matrix of the original graph. Then the similarities with respect to the given vertex can be computed in $\mathcal{O}(Nr^4)$ time. Of late Fujiwara et al. [10] improved the performance of similarity search to $\mathcal{O}(Nr)$ via the Sylvester equation. Yu et al. [27] developed an efficient SimRank computation algorithm on undirected graphs with eigenvalue decomposition technique. But low-rank approximation solutions are not feasible to process large graphs as well, because their performance is limited by the expensive cost of preprocessing. For example, the SVD algorithm requires $\mathcal{O}(rN^2)$ time to decompose a matrix of size N.

Another branch of approximation solutions are based on the random walks. Fogaras and Rácz [9] proposed the first random-walk based algorithm for fast SimRank search. The algorithm precomputes independent sets of fingerprints which index the first-meeting times through Monte-Carlo sampling in $\mathcal{O}(NR)$ time and space. The similarity can be approximated from the fingerprints at query time. Recently Kusumoto et al. [15] formalized the SimRank as linear recursive formula and obtained upper bounds via Cauchy-Schwartz inequality. Then they used random-walk sampling to estimate a single-pair SimRank score as well as the upper bound. The preprocessing cost is $\mathcal{O}(NRT)$ time, where T is the number of steps and R is the number of samples. Unlike the aforementioned two solutions, Lee et al. [16] proposed TopSim-based algorithms which compute SimRank scores by enumerating all the similarity paths (random walks). To improve the performance of TopSim-based algorithms, Lee also designed heuristic rules to prune random walks during the computation. Our TSF is also a random-walk based approximation solution. But TSF is the first solution which indexes the raw random walks with low cost meanwhile the indexes can be updated efficiently when graph changes.

Next we discuss the works about computing SimRank scores on dynamic graphs. Li et al. [18] proposed the first incremental algorithm for SimRank updating. The solution decomposes the transpose matrix $\mathbf{W}$ of the original graph into $\mathbf{U}\Sigma\mathbf{V}$ via SVD first, and then incrementally updated the matrices $\mathbf{U}, \Sigma, \mathbf{V}$ by further decomposing the difference matrix $\mathbf{\Delta W}$ according to the graph updating. The solution requires $\mathcal{O}(N^2)$ time to update the all vertex pairs. Recent Yu et al. [29] improved the solution by using rank-one Sylvester matrix equation, thus updating all vertex pairs costs $\mathcal{O}(K(nd + |AFF|))$ time, where $|AFF|$ is the size of affected areas when the graph is updated. Since these incremental solutions depend on the old all-pairs SimRank scores, which entails $\mathcal{O}(N^2)$ space complexity, they cannot be trivially extended onto large dynamic graphs. In TSF, it only needs to maintain one-way graphs whose space overhead is linear to the size of original graph.

Finally, we briefly discuss the graph sampling [17] which is related to our one-way graph. The purpose of graph sampling is to generate a representative sample from original graph while retaining some interesting measures. One-way graph is a sampled subgraph to preserve the probability of arbitrary walks which has never been mentioned before to the best of our knowledge. Additionally, the sampling method in TSF is different from the traditional ones [17].

7. CONCLUSION

In this paper, we proposed a two-stage random-walk sampling framework, TSF, to efficiently execute top-k similarity search on large dynamic graphs. TSF utilizes the one-way graph to directly index random walks in a novel manner, and uses the indexed random walks to process top-k queries efficiently. The advantage of TSF not only lies in the fact that the one-way graph can be quickly built by preprocessing, but also can be updated efficiently when the original graph changes. Our comprehensive experimental studies have clearly demonstrated the superiority of TSF over the state-of-the-art algorithms.

ACKNOWLEDGMENTS

The research is supported by the National Natural Science Foundation of China under Grant No. 61272155 and 61472009, Beijing Natural Science Foundation(4152023), and Microsoft Research Asia Collaborative Research Award. Lei Chen's work is supported in part by the Hong Kong RGC/NSFC Project N_HKUST 637/13, National Grand Fundamental Research 973 Program of China under Grant 2012-CB316200, and Google Faculty Award 2013.

8. REFERENCES

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

[2] L. Arge. The buffer tree: A new technique for optimal i/o-algorithms. In *WADS*, volume 955, pages 334–345, 1995.

[3] T. I. M. Association. Standard midi-file format spec. 1.1.

[4] B. Balkenhol and Y. M. Shtarkov. One attempt of a compression algorithm using the bwt, 1999.

[5] A. A. Benczúr, K. Csalogány, and T. Sarlós. Link-based similarity search to fight web spam. In *AIRWEB*, pages 9–16, 2006.

[6] L. Cao, B. Cho, H. D. Kim, Z. Li, M.-H. Tsai, and I. Gupta. Delta-simrank computing on mapreduce. In *BigMine*, pages 28–35, 2012.

[7] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson. *Introduction to Algorithms*. 2nd edition, 2001.

[8] B. Cui, H. Mei, and B. C. Ooi. Big data: the driver for innovation in databases. *National Science Review*, 1(1):27–30, 2014.

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

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

[11] D. A. Harville. *Matrix Algebra From a Statistician's Perspective*. Springer, corrected edition, Nov. 2000.

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

[13] W. Hoeffding. Probability inequalities for sums of bounded random variables, 1962.

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

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

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

[17] J. Leskovec and C. Faloutsos. Sampling from large graphs. In *KDD*, pages 631–636, 2006.

[18] 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.

[19] P. Li, Y. Cai, H. Liu, J. He, and X. Du. Exploiting the block structure of link graph for efficient similarity computation. In *PAKDD*, pages 389–400, 2009.

[20] P. Li, H. Liu, J. Xu, Y. Jun, and H. X. Du. Fast single-pair simrank computation. In *SDM*, pages 571–582, 2010.

[21] D. Lizorkin, P. Velikhov, M. Grinev, and D. Turdakov. Accuracy estimate and optimization techniques for simrank computation. In *PVLDB*, pages 45–66, 2010.

[22] S. Melnik, H. Garcia-Molina, and E. Rahm. Similarity flooding: A versatile graph matching algorithm and its application to schema matching. In *ICDE*, pages 117–128, 2002.

[23] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. Numerical recipes 3rd edition: The art of scientific computing. 2007.

[24] C. P. Robert and G. Casella. Monte carlo statistical methods (springer texts in statistics). 2005.

[25] W. Xi, E. A. Fox, W. Fan, B. Zhang, Z. Chen, J. Yan, and D. Zhuang. Simfusion: Measuring similarity using unified relationship matrix. In *SIGIR*, pages 130–137, 2005.

[26] W. Yu, X. Lin, and J. Le. A space and time efficient algorithm for simrank computation. In *APWEB*, pages 164–170, 2010.

[27] W. Yu, X. Lin, and J. Le. Taming computational complexity: Efficient and parallel simrank optimizations on undirected graphs. In *WAIM*, pages 280–296, 2010.

[28] W. Yu, X. Lin, and W. Zhang. Towards efficient simrank computation on large networks. In *ICDE*, pages 601–612, 2013.

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

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

[31] W. Zheng, L. Zou, Y. Feng, L. Chen, and D. Zhao. Efficient simrank-based similarity join over large graphs. In *PVLDB*, pages 493–504, 2013.