READS: A Random Walk Approach for Efficient and Accurate Dynamic SimRank

Minhao Jiang[†], Ada Wai-Chee Fu[‡], Raymond Chi-Wing Wong[†]
[†]The Hong Kong University of Science and Technology [‡]The Chinese University of Hong Kong
{mjiangac, raywong}@cse.ust.hk, adafu@cse.cuhk.edu.hk

ABSTRACT

Similarity among entities in graphs plays a key role in data analysis and mining. SimRank is a widely used and popular measurement to evaluate the similarity among the vertices. In real-life applications, graphs do not only grow in size, requiring fast and precise SimRank computation for large graphs, but also change and evolve continuously over time, demanding an efficient maintenance process to handle dynamic updates. In this paper, we propose a random walk based indexing scheme to compute SimRank efficiently and accurately over large dynamic graphs. We show that our algorithm outperforms the state-of-the-art static and dynamic SimRank algorithms.

1. INTRODUCTION

SimRank has been proposed as a measure of similarity between two vertices in a graph based on links among vertices. The idea behind is that similar objects are linked to objects that are similar. This is a recursive concept, since the neighboring similar objects are themselves linked to other similar objects, and so on. The base case is that an object is most similar to itself. SimRank has attracted much attention since its introduction [2, 5, 6, 7, 10, 11, 12, 13, 15, 17, 22, 24, 25, 27, 30].

SimRank is useful in applications with object-to-object relationships. Many applications require a measure of "similarity" between objects. For example, within a citation network we may want to find publications similar to a given publication, or we may ask for the most similar journals to a given journal. For collaborative filtering in a recommender system, an important process is to cluster objects, so that similar users or items are grouped together based on a similarity measure [1]. A recent study shows that SimRank can produce good results for a recommender system over web data [19]. SimRank is also found to be useful in a multitude of other applications, such as sponsored search [2], natural language processing [18, 20], link prediction [14], graph clustering [31], and web spam detection [3]. Since most data graphs are dynamic in nature, it is worth noting that many such applications are in need of a dynamically adaptive SimRank algorithm. A recommender system or web spam detection requires spontaneous responses on top of fast-evolving datasets. However, handling dynamic updates is challenging, and amid the vast amount of work on SimRank, only [21] is scalable and efficient.

Proceedings of the VLDB Endowment, Vol. 10, No. 9
Copyright 2017 VLDB Endowment 2150-8097/17/05.

Given a directed and unweighted graph $G = (V, E)$, where V is the vertex set and E is the directed edge set, the SimRank value $sim(u, v)$ for vertices u and v in graph G is

$$sim(u,v) = \begin{cases} 1 & u = v \\ \displaystyle\sum_{u' \in In(u), v' \in In(v)} \frac{c \times sim(u', v')}{|In(u)||In(v)|} & u \neq v \end{cases} \quad (1)$$

where c is a constant value in $[0, 1]$, commonly set to 0.6 or 0.8, and $In(u)$ is the set of incoming neighbours of vertex u [9]. We consider the computation of SimRank values among the vertices in V. The following shows some of the main variations of the problem. [*Single Pair (One-to-one)*]: Given a pair of query vertices u and v, return the SimRank value $sim(u, v)$. [*Single Source (One-to-all)*]: Given a query vertex u, return the SimRank value between u and every vertex in V. [*Top-k*]: Given a query vertex u, return the top k SimRank values of $sim(u, v)$ between u and other vertices $v \in V$. [*All Pairs*]: Return the SimRank value $sim(u, v)$ between every pair of vertices u and v in G. [*Partial Pairs*]: Given two vertex subsets A and B, return the SimRank values for all vertex pairs (u, v) where $u \in A$ and $v \in B$.

1.1 Limitations of Existing Solutions

Although there are various studies on SimRank, most of them focus on static graphs, and only very few of them are capable of handling dynamic updates. Most dynamic algorithms, either [13, 24] which require an all-pair matrix leading to unaffordable $O(|V|^2)$ space cost, or [12] with $O(D^{2t})$ querying time, where t is the max depth, and relying on heuristic pruning without stable performance guarantee, fail to handle dynamic updates on large graphs efficiently.

The only scalable and efficient dynamic algorithm for large graphs is TSF, proposed by Shao et al. in [21]. TSF builds an index for top-k SimRank querying. The preprocessing step randomly samples R_g *one-way graphs*, where each graph contains the coupling of random walks with one random walk for each vertex. The one-way graphs serve as an index for the querying process. For each one-way graph, R_q new random walks of vertex u are sampled at query time. The query time is $O(R_q R_g t|V| + |V| \log k)$ (see Section 2 for explanation), where t is typically 10. The index requires

$O(R_g|V|)$ space. However, the probabilistic guarantee for the error bound is based on the assumption that no cycle in the given graph has a length shorter than t. This may not hold in general, and thus in some cases its precision is low.

The best-known method for the computation of SimRank is SLING by Tian and Xiao [23]. SLING consists of a near-optimal index structure. With this structure, the single source (one-to-all) query time is $O(|V|/\epsilon)$ or $O(|E|\log^2 1/\epsilon)$, with respect to two proposed querying algorithms, and the preprocessing time is $O(|E|/\epsilon + |V|\log(|V|/\delta)/\epsilon^2)$, where ϵ is a bound on the additive error, with at least $(1-\delta)$ success probability. The single-source algorithm can be adapted for computing top-k SimRank by selecting the top values with an additional $O(|V|\log k)$ time cost. However, the algorithm is static, while the given graphs are dynamic in many applications. Changes in the graph require rebuilding the indexing components, which is highly inefficient. Besides, from our empirical studies, the precision of SLING in top-k querying is not high in some cases, which may be due to the pruning process in the indexing.

1.2 Contributions

We propose a new method called READS (Randomized Efficient Accurate Dynamic SimRank computation) for the computation of SimRank for static and dynamic graphs. The main contributions are as follows:

1. Compared with the state-of-the-art dynamic algorithm TSF[21] and static algorithm SLING[23], READS makes no assumption about cycle lengths as in TSF, and unlike SLING, READS is dynamic. Our theoretical and empirical performances are also better. In experiments, our algorithms are typically one to two orders of magnitude faster than our most relevant competitor TSF in querying and updating while returning better accuracy.

2. Our algorithm is based on a new definition of random walks, we show that it is optimal under a condition noted for efficiency.

3. In indexing, efficient local search techniques are developed to boost the accuracy. In querying, an online sampling technique improves the querying accuracy significantly with only slightly additional cost.

4. While our discussion is based on single-source SimRank, our indexing also can support single-pair, top-k, all-pairs, and partial-pairs SimRank querying.

This paper is organized as follows. We summarize the related work in Section 2. In Section 3, we present some definitions and a baseline Monte Carlo method. In Section 4, we describe our main indexing and querying method based on sets. Section 5 introduces our boosting techniques based on local search. Section 6 is about dynamic update handling and an improvement by online walks. Our experimental results are reported in Section 7. We conclude in Section 8.

2. RELATED WORK

Jeh and Widom first introduced SimRank as a measure of structural and contextual similarity between two vertices in a graph [9]. Since its introduction, there has been much interest on the study of efficient and accurate computation for different variations of the problem.

Dynamic Algorithms

Li et al. [13] first studied the problem of SimRank computation on dynamic graphs. Their idea is to factorize the backward transition matrix of the given graph via SVD and to incrementally update the component matrices. Yu et al. [24] improve on this approach via a rank-one Sylvester matrix equation. A problem about the correctness of their SimRank formulation has been pointed out in [11, 23]. The scalability of these matrix approaches is limited since they require $O(|V|^2)$ space.

For top-k search, Lee et al. in [12] propose an index-free algorithm TopSim that is based on a random walk coupling mechanism with random walk pruning techniques. The complexity is $O(D^{2t})$, where D is the average degree and t is the max depth. Note that [12], being index-free, supports dynamic updates but the query efficiency is low.

The state-of-the-art index for dynamic updates is TSF proposed by Shao et al. in [21] by sampling one-way graphs to index raw random walks. The query time for top-k SimRank is $O(R_qR_gt|V| + |V|\log k)$ where t is the max depth. The $|V|$ factor is due to the time to traverse the reversed one-way graph, which depends on $\sum_{i=1}^{t} d_i$, where d_i is the total degree of the vertices visited at iteration i. In the worst case, all vertices are visited. Thus, the complexity for the traversal is $O(|V|)$. See Section 1.1 for more discussion.

Static Algorithms

Jeh and Widom propose an iterative deterministic computation in [9] for the all-pairs problem, which requires $O(|V|^2)$ space and $O(K|E|^2)$ time for K iterations. It is shown in [15] and [23] that if $K \geq O(\log(1/\epsilon))$, the worst-case error is ϵ, and the time complexity becomes $O(|E|^2\log(1/\epsilon))$. Lizorkin et al. [15] propose optimization techniques for reducing the computation of vertex pairs: eliminating vertex pairs with zero scores, caching partial simularities for later iterations, and the use of a similarity threshold. The time complexity is $O(k|V|^3)$ for k iterations and it requires $O(|V|^2)$ space. [28] proposes effective optimization techniques and improves the time to $O(\log(1/\epsilon)\min\{|E||V|, |V|^{\omega}\})$ where $\omega \approx 2.4$, and the space to $O(|V|^2)$. They are still steep for large graphs.

Fogaras and Racz first introduced a random walk based approximation algorithm [5]. Their indexing method is based on fingerprint trees, which can represent a set of reversed walks for each vertex with a size of $O(|V|)$. The query time for single pair SimRank is $O(rt)$, where r is the number of sets of random walks, and t is the length of a random walk. The space requirement is $O(|E|+r|V|)$. It is shown in [16, 23, 21] that the algorithm is not scalable and the precision is not competitive for the top-k SimRank problem.

Kusumoto et al. in [11] introduce a linear recursive formula for SimRank, and propose an efficient algorithm for top-k search, which involves search pruning based on upper bounds on the SimRank score. A Monte Carlo approach is used for the single-pair SimRank problem via a random walk interpretation of their formula. Their algorithm requires $O(|E|+|V|)$ indexing time and $O(rt|V|)$ space for r random walks of length t, single-pair querying requires $O(rt)$ time. A diagonal correction matrix is employed in some related work to reduce the computation cost for top-k search [13] [6], and [26]. For example, the use of SVD can be used to approximate the original matrix by low-ranked matrices. As noted in [11], though such a SimRank formulation deviates from the original definition, it can be effective for preserving the similarity ranking. However, these algorithms require quadratic time and space, which become prohibitive for large graphs. A linearization technique in [16] improves the query time for single pair to $O(|E|\log(1/\epsilon))$, and single source to $O(|E|\log^2(1/\epsilon))$, with at most ϵ additive error.

Tao et al. consider the problem of top-k similarity join based on SimRank [22], which is to find the k most similar pairs of vertices with the largest SimRank similarities among all possible pairs. They consider using the factor of $\sqrt{c}$ for each of two paths that meet in the computation of SimRank. Yu and McCann consider partial-pairs SimRank querying [26]. For vertex subsets A and B, their algorithm requires $O(w|E|\min\{|A|,|B|\})$ query time and $O(|E|+w|V|)$ space for w iterations.

The state-of-the-art method for static graphs is SLING, proposed by Tian and Xiao in [23]. SLING involves two preprocessing steps: (1) an estimation of multi-meeting probabilities of vertices, and (2) a deterministic computation to correct the multi-meeting probabilities to first-meeting probabilities. See Section 1.1 for more discussion.

An experimental evaluation of 10 algorithms from 2002 to 2015 is reported in [29]. It shows that despite the vast amount of work, the precision and runtime of known algorithms still leave much room for improvement. Our work advances the status quo in both aspects.

3. DEFINITION AND APPROXIMATION

The SimRank measurement is first defined by Equation (1), which is recursive. Here, we describe another definition of SimRank and a natural Monte Carlo method based on this definition. Some of our notations are as follows:

$G = (V, E)$	the given directed unweighted graph
$In(u)$	the set of in-neighbors of vertex u
c	the decay factor in the definition of SimRank
r	the number of simulations
$sim(u, v)$	the SimRank score of vertices u and v in G
t	the maximum length of a random walk
$s(u, v)$	SimRank score based on length t random walks
π_u	a reversed random walk from u of length t
$first(\pi_u, \pi_v)$	the first meeting point of π_u and π_v
$f(\pi_u, \pi_v)$	the first meeting point of π_u and π_v within t steps

Given a directed unweighted graph $G = (V, E)$, where V is the vertex set and E is the directed edge set. For $u \in V$, let $In(u)$ be the set of in-neighbors of u. Another interpretation of SimRank is based on random walks [9]. Let t be an integer parameter which is the maximum length of a random walk. We call a path $\pi_u = (u_0 \leftarrow u_1 \leftarrow ... \leftarrow u_t)$ a **reversed random walk** from u if the following conditions are met:

(C1)	$u_0 = u$
(C2)	$u_i \in V$ or $u_i = null$ for $0 \leq i \leq t$
(C3)	$u_{i+1} = null$ if $\|In(u_i)\| = 0$ or $u_i = null$
(C4)	$\Pr(u_{i+1} = v) = 0$ if $v \notin In(u_i)$ for $0 \leq i < t$
(C5)	$\Pr(u_{i+1} = v) = \frac{1}{\|In(u_i)\|}$ if $v \in In(u_i)$ and $0 \leq i < t$

In the above, we assume that $In(null) = \emptyset$. The position of the first-meeting-point of π_u and π_v is denoted by $first(\pi_u, \pi_v)$, i.e., (1) for $i < first(\pi_u, \pi_v)$, $u_i \neq v_i$ and (2) for $i = first(\pi_u, \pi_v)$, $u_i = v_i \neq null$. If $u_i \neq v_i$ for $0 \leq i \leq t$, then $first(\pi_u, \pi_v) = \infty$. SimRank can be rewritten based on the first meeting point of reversed random walks [9]:

$$sim(u, v) = \sum_{i=1}^{\infty} \Pr(first(\pi_u, \pi_v) = i) \times c^i = E[c^{first(\pi_u, \pi_v)}]$$

The above equation takes all the paths up to length ∞ into consideration, which may be computationally expensive. A practical approximation is to only count short paths. Given the parameter t, the approximated $first(\pi_u, \pi_v)$, denoted by $f(\pi_u, \pi_v)$, only takes early meeting-points that are within t steps into consideration. The value of t is set as 10 in [9, 5, 15, 11, 21].

$$f(\pi_u, \pi_v) = \begin{cases} first(\pi_u, \pi_v) & \text{if } first(\pi_u, \pi_v) \leq t \\ \infty & \text{if } first(\pi_u, \pi_v) > t \end{cases} \quad (2)$$

The approximated $sim(u, v)$ denoted by $s(u, v)$ is naturally given based on $f(\pi_u, \pi_v)$.

$$s(u, v) = \sum_{i=1}^{t} \Pr(f(\pi_u, \pi_v) = i) \times c^i = E[c^{f(\pi_u, \pi_v)}] \quad (3)$$

Since $sim(u, v) - s(u, v) = \sum_{i=t+1}^{\infty} \Pr(first(\pi_u, \pi_v) = i) \times c^i \leq c^{t+1}$, the error of $s(u, v)$ is bounded within a small gap:

THEOREM 1. *$sim(u, v) - s(u, v) \in [0, c^{t+1}]$*

A Monte Carlo Method: In Equation (3), we need to estimate $\Pr(f(\pi_u, \pi_v) = i)$, and a natural approach is a Monte Carlo method. Given a graph G, and vertices u and v, we generate a set of r reversed random walks with length up to t from $u = \{\pi_u^1, \pi_u^2, ..., \pi_u^r\}$ and another set from $v = \{\pi_v^1, \pi_v^2, ..., \pi_v^r\}$. The meeting-point of π_u^j and π_v^j is the outcome of $f(\pi_u^j, \pi_v^j)$, so that the r independent tests serve as an estimator $\widetilde{\Pr}(f(\pi_u, \pi_v) = i)$. Formally,

$$\widetilde{\Pr}(f(\pi_u, \pi_v) = i) = |\{j \in [1, r] | f(\pi_u^j, \pi_v^j) = i\}|/r \quad (4)$$

Embedding $\widetilde{\Pr}(f(\pi_u, \pi_v) = i)$ into Equation (3) gives an estimated value of $s_0(u, v)$ for $s(u, v)$. Its accuracy depends on r. We can derive a guarantee on the error bound based on Hoeffding's inequality.

LEMMA 1 (HOEFFDING [8]). *Let $X_1, X_2, ..., X_r$ be independent random variables where X_i are strictly bounded by the interval $[a_i, b_i]$, let $\overline{X} = \frac{1}{r}(X_1 + ... + X_r)$. Then, for all $\epsilon > 0$: $\Pr(|\overline{X} - E[\overline{X}]| \geq \epsilon) \leq 2e^{-2r^2\epsilon^2/\sum_{i=1}^{r}(b_i - a_i)^2}$*

Since each meeting point contributes a value in $[0, c]$ towards $s_0(u, v)$, from Lemma 1, we derive the following.

THEOREM 2. *$\Pr(|s_0(u, v) - s(u, v)| \geq \epsilon) \leq 2e^{-2r\epsilon^2/c^2}$*

Our indexing methods are based on a similar strategy of generating r sets of random walks for each vertex, though the random walks are generated in a different way by revising Conditions (C1)-(C5). In the following, we shall refer to the r sets of random walks as r **simulations**.

4. SIMRANK INDEXING BY SETS

Based on the basic Monte Carlo method, we propose a new indexing scheme to efficiently answer various SimRank queries, including single-pair, single-source, and top-k querying. We call our method **READS**.

4.1 Generalization

We first generalize the definition of "reversed random walk". Given $G(V, E)$, a path $\pi_u = (u_0 \leftarrow u_1 \leftarrow ... \leftarrow u_t)$ is called **a generalized reversed random walk from** u if it satisfies Conditions (C1)-(C4), and Condition (C6) below

(C6)	$\sum_{v \in V} \Pr(u_{i+1} = v) \leq 1$ for $0 \leq i < t$

Clearly, reverse random walk is a special case of generalized reverse random walk, since Condition (C5) always implies Condition (C6). Thus, for a reverse random walk, if $|In(u_i)| > 0$, there always exists a non-null next vertex of u_{i+1}. Condition (C6) in the above definition relaxes this requirement, and allows a reverse random walk to stop at u_i (i.e., u_{i+1} becomes *null*), even if $|In(u_i)| > 0$. Note that Condition (C6) allows us to set $Pr(u_{i+1} = v)$ to be different from $Pr(u_{j+1} = v)$ for $i \neq j$ even if $u_i = u_j$. It means that the transition probability for a vertex can depend on its position in the random walk. Thus, such a random walk may not be a Markov chain with a vertex being a state. However, certain features of Markov chains can be useful, one of them is coupling.

We call two generalized reverse random walks $\pi_u = (u_0 \leftarrow u_1 \leftarrow ... \leftarrow u_t)$ and $\pi_v = (v_0 \leftarrow v_1 \leftarrow ... \leftarrow v_t)$ **mergeable** if given $u_i = v_i$, $\Pr(u_{i+1} = v) = \Pr(v_{i+1} = v)$ for each $v \in V$ holds. Clearly, any two reverse random walks in a graph are mergeable. The definition of mergeable walk guarantees that if two paths π_u and π_v meet at a point, say at the i-th step with $u_i = v_i$, then by setting $u_j = v_j$ for $j \geq i$, we get another path π_u which is also a valid generalized reverse random walk from u. As a result, it is possible to merge a set of walks into a tree. E.g., given graph G_1 in Figure 1, two walks starting from v_3 and v_2 meet at v_0, then we only need to keep one of the two successor vertices v_1 and v_2. As shown in Section 6, the resulting tree structures can support very simple and efficient dynamic updates.

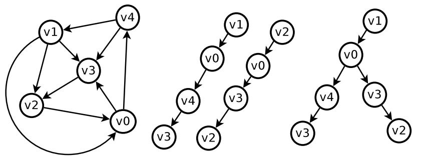

Figure 1: Graph G_1, Walks, and Merged Walks

The merging or coupling of random walks, which is a useful concept in the theory of random walks, has been utilized in [5], the algorithms in [5] are not competitive as shown in [21, 23]. Our purpose in generalizing the concept of reverse random walk is that it offers us a more flexible interpretation of SimRank from the perspective of "random walks". Our first idea is to design a set of sampling probabilities to refine Condition (6) of the generalized walk, which brings about some desirable properties. Secondly, we introduce optimization techniques to further boost the accuracy guarantee in Section 5. These strategies together result in a scheme that is highly efficient and accurate.

4.2 SimRank-Aware Random Walks

There are many ways to generate a set of mergeable reverse generalized random walks for all vertices in a graph. We discuss in the following which sets should be generated to optimize SimRank querying (see Section 4.3).

We specialize the generalized reverse random walk by specifying the transition probability setting for the steps in the walk. Let $(u_0 \leftarrow u_1 ... \leftarrow u_t)$ be a generalized reverse random walk. Let $u_0 = u$. We randomly assign the next vertex u_1 to be one of its in-neighbors with a probability of $1/|In(u)|$. For each of the remaining steps, at u_i, $i > 0$, with a probability of $\sqrt{c}$, we randomly assign one of the in-neighbors of u_i to be u_{i+1}. There is a probability of $1 - \sqrt{c}$ that u_{i+1} becomes *null* so that the walk ends. Thus, we replace Condition (C6) with the following 2 conditions:

(C6a)	$Pr(u_1 = v) = \frac{1}{\lvert In(u_0)\rvert}$ for $v \in In(u_0)$
(C6b)	$Pr(u_{i+1} = v) = \frac{\sqrt{c}}{\lvert In(u_i)\rvert}$ for $v \in In(u_i), 0 < i < t$

We call a generalized reversed random walk that follows Conditions (C1)-(C4), (C6a) and (C6b) an **SA random walk** (SimRank-Aware random walk). Clearly, SA random walks are mergeable.

Based on standard reversed random walks, from Equation (3), we have $s(u, v) = \sum_{i=1}^{t} \Pr(f(\pi_u, \pi_v) = i) \times c^i$.

With the newly defined SA random walks, we denote the probability that $(f(\pi_u, \pi_v) = i)$ by $Pr^{SA}(f(\pi_u, \pi_v) = i)$ and we can prove $Pr^{SA}(f(\pi_u, \pi_v) = i) = \Pr(f(\pi_u, \pi_v) = i) \times c^{i-1}$ by induction. Therefore, we can rewrite $s(u, v)$ as:

$$s(u, v) = \sum_{i=1}^{t} Pr^{SA}(f(\pi_u, \pi_v) = i) \times c \qquad (5)$$

As discussed in Section 3, we can adopt a Monte Carlo method that gives an estimation to $\Pr^{SA}(f(\pi_u, \pi_v) = i)$ by means of r simulations of sampling SA random walk pairs π_u and π_v that follow Conditions (C6a) and (C6b).

$$\widetilde{Pr}^{SA}(f(\pi_u, \pi_v) = i) = |\{j \in [1, r] | f(\pi_u^j, \pi_v^j) = i\}|/r \qquad (6)$$

Let $s_1(u, v)$ be the estimated value of $s(u, v)$ based on this approach. As we collect the occurrences of pairs of random walks that first meet within t steps, according to Equation (5), each occurrence as such contributes an additive value of c/r to $s_1(u, v)$. It is easy to show that a similar guarantee as Theorem 2 holds, substituting $s_1(u, v)$ for $s_0(u, v)$.

A uniform contribution from each pair of meeting walks: From Equations (5) and (6), the additive contribution of each pair of meeting walks in the simulations is uniformly c/r. This is an important difference from the baseline case in Section 3, where the contribution, namely c^i, depends on the depth i of the meeting point. The advantage of having such a uniform value is that in order to derive the SimRank approximation value of $s_1(u, v)$, we can simply count the number of simulations in which the paths from u and v meet. That is, we compute the following for approximating $s(u, v)$:

$$\sum_{i=1}^{t} \widetilde{Pr}^{SA}(f(\pi_u, \pi_v) \neq \infty) = |\{j \leq r | f(\pi_u^j, \pi_v^j) \neq \infty\}|/r$$

We shall see that for any vertex u, we can store in a set S all other vertices with SA random walks that meet with the SA random walk from u in a simulation. To compute the single source SimRank $s_1(u, *)$, we only need to increase by c/r the SimRank value $s(u, v)$ for each $v \in S$ for each simulation. This greatly reduces the computation cost and the storage requirement as compared to the baseline approach.

4.3 Optimality of SA Random Walks

From the previous subsection, we see that a uniform weight contribution for each pair of SA random walks that meet is a key to the effectiveness of the set-based algorithm. More precisely, since the sets for different simulations are handled separately, a desirable feature of such an indexing scheme is that for each of the r simulations, we have a unique uniform weight contribution for each meeting pair of random walks. Here, we show that the SA random walk induces an optimal error bound within such a context.

In our estimation of $s(u, v)$, we carry out r simulations, where each simulation essentially consists of a random walk starting from each vertex in the graph. In the generalized reversed random walk model, at each step of the random walk, we can set an additional weight factor on the transition probability to the in-neighbors of the current vertex. A general way to set this factor is to assign a value at each distance from the starting vertex. Thus, for the i-th simulation, let the factor be given by $\sqrt{a_j^i}$ for the sampling at the j-th step of a random walk. Thus, for a path $\pi_u = (u = u_0 \leftarrow u_1 ... \leftarrow u_j)$, the transition probability to an in-neighbor of u_{j-1} at step j is given by $\frac{\sqrt{a_j^i}}{|In(u_{j-1})|}$. We call it **c-walk** (Canonical random walk).

Let $f^C(u, v, \ell_1)$ be the first meeting point of vertices u and v, where u and v are at the ℓ-th step of two c-walks, respectively, and the two walks have not met up to the ℓ_1-th step. Let $Pr_i^C(f^C(u, v, \ell_1) = \ell_2)$, where $\ell_2 > \ell_1$, be the probability that in the i-th simulation, $f^C(u, v, \ell_1) = \ell_2$. We first derive a relationship between this probability and a counterpart probability of a reversed random walk.

LEMMA 2. *Let ℓ_1 and ℓ_2 be two integers and $\ell_2 > \ell_1 > 0$.* $Pr_i^C(f^C(u, v, \ell_1) = \ell_2) = \Pr(f(\pi_u, \pi_v) = \ell_2 - \ell_1) \prod_{j=\ell_1+1}^{\ell_2} a_j^i$

PROOF. We prove by induction on $\ell_2 - \ell_1$. The base case is when $\ell_2 = \ell_1 + 1$. With one more step, the sampling probability of a c-walk from u is $\sqrt{a_{\ell_2}^i}$ times that of a reversed random walk, being $\frac{\sqrt{a_{\ell_2}^i}}{|In(u)|}$. Similarly for v. Thus,

$$Pr^C(f^C(\pi_u, \pi_v, \ell_1) = \ell_2) = \Pr(f(\pi_u, \pi_v) = 1) \times a_{\ell_2}^i$$

Assume it holds for $\ell_2 - \ell_1 \leq j$, so when $\ell_2 = \ell_1 + j + 1$,

$$\begin{aligned} & Pr_i^C(f(u, v, \ell_1) = \ell_2) \\ &= \frac{\sqrt{a_{\ell_1+1}^i}}{|In(u)|} \frac{\sqrt{a_{\ell_1+1}^i}}{|In(v)|} \sum_{\substack{u' \in In(u) \\ v' \in In(v)}} Pr_i^C(f(u', v', \ell_1 + 1) = \ell_2) \\ &= \frac{a_{\ell_1+1}^i}{|In(u)||In(v)|} \sum_{\substack{u' \in In(u) \\ v' \in In(v)}} Pr(f(\pi_{u'}, \pi_{v'})) = j) \prod_{j=\ell+2}^{\ell_2} a_j^i \\ &= \Pr(f(\pi_u, \pi_v) = j + 1) \times \prod_{j=\ell+1}^{\ell_2} a_j^i \end{aligned}$$

□

COROLLARY 1. *Let ℓ be a positive integer.*

$$Pr_i^C(f^C(u, v, 0) = \ell) = \Pr(f(\pi_u, \pi_v) = \ell) \times \prod_{j=1}^{\ell} a_j^i \quad (7)$$

We aim to find a_j^i, $1 \leq i \leq r$, $0 \leq j \leq t$, so that

$$s(u, v) = \frac{1}{r} \sum_{i=1}^{r} \sum_{j=1}^{t} Pr_i^C(f^C(u, v, 0) = j) \times a_0^i \quad (8)$$

If Equality (8) holds, then we can apply a Monte Carlo approach so that each pair of meeting random walks in the i-th simulation contributes the same weight of a_0^i/r to the estimation of $s(u, v)$. The value of each random variable X_i in the corresponding Hoeffding bound in Theorem 1 is in the range of $[0, a_0^i]$, since X_i is either 0 or a_0^i. Next, we derive the setting to optimize a bound on the estimation error.

Let $P_i = \Pr(f(\pi_u, \pi_v) = i)$. Substituting Equation (7) into Equation (8), we get

$$\sum_{j=1}^{t} P_j \times c^j = \frac{1}{r} \sum_{j=1}^{t} P_j \times \sum_{i=1}^{r} \prod_{h=0}^{j} a_h^i$$

Equating the similar terms for each P_j, we get a set of equations, where $a_j^i \in [0, 1]$ for $1 \leq j \leq t$, $1 \leq i \leq r$.

$$\frac{1}{r} \sum_{i=1}^{r} a_0^i a_1^i a_2^i ... a_j^i \quad = \quad c^j \quad (9)$$

With r simulations to collect random walks that first meet at different lengths, we can bound the error by Hoeffding's Inequality (see Lemma 1). In the i-th simulation, the value of $\sum_{j=1}^{t} Pr_i^C(f^C(u, v, 0) = j)$ can be approximated by examining the existence of a meeting point for the c-walks of u and v. The expected value of X_i is given by $\sum_{j=1}^{t} Pr_i^C(f^C(u, v, 0) = j) \times a_0^i$. Let $s_C(u, v)$ be the estimated SimRank for u,v, from Inequality (1), we get

$$\Pr(|s_C(u, v) - s(u, v)| \geq \epsilon) \leq 2e^{-2r^2\epsilon^2 / \sum_{i=1}^{r} (a_0^i)^2}$$

To optimize the bound, we minimize $\sum_{i=1}^{r} (a_0^i/r)^2$. Given the condition of Equation (9), it is easy to see that an optimal setting is $a_0^i = c$, $a_1^i = 1$, $a_2^i = c$, ..., $a_t^i = c$.

This matches exactly the setting of Conditions (C6a) and (C6b) for SA random walks.

4.4 Set based Indexing and Querying

Given the probabilistic settings to generate the sample sets in the previous subsections, we describe the process of indexing and querying.

For the indexing, r sets of samples are generated in r independent simulations. In each simulation, we first create $|V|$ trees with a single (leaf) node in each tree. Each vertex $v \in V$ is stored in one of the leaf nodes. If $v \in V$ is stored in tree node a, we say that $vertex(a) = v$, and we say that a leaf node a is at level 0, or $level(a) = 0$. Higher level nodes in the trees are generated by reversed random walks level-by-level, and during this process, two intermediate trees are merged whenever their roots contain the same $v \in V$. The growth stops at level t. Thus, a forest is built, and we call it an **SA forest**. The trees in the forest are called SA trees.

We use a queue Q to facilitate the level-by-level growth of the trees. Initially, the first $|V|$ leaf nodes created are enqueued. For each popped tree node a, where $vertex(a) = u$ and $level(a) = \ell$, we attempt to assign one of the in-neighbors of u to be stored in its father. If a is a leaf node, i.e., $\ell = 0$, we select each in-neighbor with a probability of $\frac{1}{|In(u)|}$. If u is at level 1 or above, we select each in-neighbor with probability $\frac{\sqrt{c}}{|In(u)|}$. The selected vertex, v, if there is one, is then entered into the queue Q if it has not been pushed as a node at level $\ell + 1$. If v already exists at level $\ell + 1$, merging of trees is triggered.

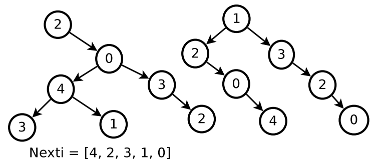

Figure 2: Indexing the Merged Walks

After Q becomes empty, a set of SA trees is constructed. The vertices stored at the leaf nodes of an SA tree forms a set, which we call an **SA set**. To store the SA sets, an array $Next_i$ is created for the i-th simulation. Each element of $Next_i$ is a vertex ID, where $Next_i[v]$ is the right sibling of vertex v in simulation i. For the rightmost leaf node v of a tree, $Next_i[v]$ is set to the leftmost leaf node in this tree. Figure 2 is an example of indexing the merged walks generated by a simulation. In this figure, the vertex IDs are 0 to 4. From $Next_i[0] = 4$, we go to $Next_i[4] = 0$, so we detect a loop and this means we have collected all leaf nodes of one tree, namely 0 and 4.

Algorithm 1: Basic Indexing

```
Input: $G$, $r$, $t$
Output: index $Next$
for $i = 1 \to r$ do
    Queue $Q \leftarrow \emptyset$;
    for each $u \in V$ do
        Create a tree $T$ with only one leaf node $a$ with $vertex(a) = u$ and $level(a) = 0$;
        $enqueue(Q, a)$
    while $Q \neq \emptyset$ do
        $a \leftarrow dequeue(Q)$;
        if $level(a) < t$ then
            if $level(a) = 0$ then $\theta = 1$ else $\theta = \sqrt{c}$;
            With probability $\theta$, select a vertex $v$ from $In(vertex(a))$ uniformly at random and
            if $v$ exists at node $b$ at $level(a) + 1$ in a tree $T'$ then
                Link $a$ as a child of $b$ in $T'$ ... merging
            else
                Create a father node $f$ of $a$ with $vertex(f) = v$; $enqueue(Q, f)$
    for each tree $T$ do
        for each leaf node $v$ in $T$ do
            if $v$ is the right most leaf node of $T$ then
                $Next_i[v] \leftarrow$ the left most leaf node of $T$ ;
            else
                $Next_i[v] \leftarrow$ the right sibling of $v$ ;
return $Next = \{Next_1, ..., Next_r, \}$;
```

In the implementation, although a forest of SA trees is built in each simulation, keeping the leaf nodes (SA set) only for each tree is sufficient to support the basic querying process. This is because given a query vertex u, only the leaf nodes located at the same SA tree as the leaf node for u are relevant, and they can be visited through $Next_i$.

Given r sets of $Next_i$, it is easy to see how querying goes. For one-to-all/top-k querying, to compute the SimRank value from u to all the others vertices, we enumerate i, visit $Next_i[u]$, followed by its right sibling $Next_i[Next_i[u]]$, etc, until we meet u again. When v is scanned, $s_1[v]$ is incremented by c/r. For one-to-one querying, $Next_i[v]$ stores the tree ID that v belongs to in simulation i. Counting the number of times $Next_i[u] = Next_i[v]$ gives $SimRank(u, v)$.

Algorithm 2: Basic One-to-all Querying

```
Input: $Next$, $u$
Output: $s_1(u, *)$
for $i = 1 \to r$ do
    $v \leftarrow Next_i[u]$;
    while $v \neq u$ do
        $s_1(u, v) += c/r$;
        $v \leftarrow Next_i[v]$;
$s_1(u, u) \leftarrow 1$;
return $s_1(u, *)$;
```

We only show the one-to-all querying in Algorithm 2 since the other variations are straightforward. When a coexisting occurrence of u and v is encountered, a weight of c/r is added to the SimRank estimation. Therefore, Hoeffding's inequality leads to the same error bound as Theorem 2.

[Index size and Indexing Time]: The expected size of the forest is $O(|V|r)$, because the number of nodes in each level is expected to be $\sqrt{c}$ times less than the previous level, leading to $O(|V|)$ nodes in the trees generated in a simulation. In practice, the $\sqrt{c}$ effect together with the merging effect reduce the number of nodes very rapidly when level grows. Thus, the expected indexing time is $O(|V|r)$. For each simulation, array $Next_i$ takes $O(|V|)$ space, and the total index size is $O(|V|r)$. Single source querying time is $O(|V|r)$. We shall discuss the setting of r after we introduce our optimization steps in the next section.

4.5 Comparison with previous work

The factor $\sqrt{c}$ is also used in the computation of SimRank in [22] and [23]. However, the way in which this factor is used is entirely different. In both [22] and [23], the $\sqrt{c}$ factor is used in a deterministically computation of SimRank. In contrast, we sample SA walks to form SA forests, and introduce techniques for high accuracy and efficiency. Another important difference is that the first step of a SA walk has a zero stopping probability, due to an optimality study in Section 4.3. In the following section, we will show that the second step is also assigned with a zero stopping probability when using local search. This position-dependent probability and the optimality are not shared by [22] and [23].

There is coupling of random walks in the fingerprint trees in [5], as is commonly used in the theory of random walks. However, there are significant differences with the SA trees. Firstly, unlike SA walks, there is no probabilistic stopping at each step in [5]. Secondly, with SA forests, querying checks whether two nodes are in the same tree, while [5] computes the depths of two nodes in the same tree. Thus, we only need $O(r)$ time for computing a single pair SimRank value, while $O(rt)$ time is needed in [5].

5. OPTIMIZATION

It is possible to further boost precision(from Theorem 2 to 3) by incorporating our basic querying with local search. Two local search boosters are introduced in this section, with which we can give a better bound for the approximation error, which in turn improves the query time and indexing space while preserving the approximation guarantee.

5.1 Neighborhood Average

We call the first technique **Neighborhood Average**. Let us rewrite the SimRank definition and analyze the relationship between the querying vertices and the neighborhood. Let us first introduce a term $s_{t-1}(u, v)$ which has the same definition as $s(u, v)$ except that we replace t by $t - 1$. If $u \neq v$, let us define $s_u(v')$ as:

$$s_u(v') = \sum_{u' \in In(u)} \frac{s_{t-1}(u', v')}{|In(u)|} \tag{10}$$

Then, we can rewrite $s(u, v)$ as:

$$s(u, v) = c \times \textstyle\sum_{v' \in In(v)} \frac{s_u(v')}{|In(v)|} \tag{11}$$

The high-level idea of using the neighborhood average is that when the neighbours of u and v are known, we use $\overline{s_u}(v')$, the average of r simulations (with random walks of length $t - 1$), to approximate the term $s_u(v')$. In each simulation, we randomly choose a $u' \in In(u)$, and visit the vertices v' that coexist with u' in the same SA set to update $\overline{s_u}(v')$. We search the out-going neighbours v of each v' with $\overline{s_u}(v') > 0$ to update the SimRank value of (u, v) by aggregating the weights. The details are given in Algorithm 3, integrating neighborhood average with neighborhood exact search.

5.2 Neighborhood Exact Search

The second local search booster aims to search the neighborhood of u and v to obtain the exact value of $\Pr(f(\pi_u, \pi_v) = 1)$. We call it **Neighborhood Exact Search**. In Algorithm 2, the sum of weights increases by either 0 or c in each simulation. Reducing the range $[0, c]$ could improve the precision according to Hoeffding's inequality. To this end, we define $s'(u, v)$ to rewrite SimRank for $u \neq v$.

$s'(u,v) = \Pr(f(\pi_u, \pi_v) = 1)c^2 + \sum_{i=2}^{t} \Pr(f(\pi_u, \pi_v) = i)c^i$
$s(u,v) = s'(u,v) + \Pr(f(\pi_u, \pi_v) = 1) \times (c - c^2)$

In $s'(u, v)$, each term is associated with a weight of at most c^2, rather than c in $s(u, v)$. The index structure can be modified to capture $s'(u, v)$, so that the weights from the simulations become either 0 or c^2, narrowing the range from $[0, c]$ to $[0, c^2]$. The gap between $s(u, v)$ and $s'(u, v)$ is $\Pr(f(\pi_u, \pi_v) = 1) \times (c - c^2)$, but when neighborhood search is applied, we get the exact value of $\Pr(f(\pi_u, \pi_v) = 1)$ easily.

When indexing, we replace Conditions (C6a) and (C6b) with the following Conditions (C7a) and (C7b), so that the walks generated by Conditions (C1)-(C4), (C7a) and (C7b) are used to build forests as index to capture $s'(u, v)$.

(C7a)	$\Pr(u_{i+1} = v) = \frac{1}{\lvert In(u_i)\rvert}$ for $v \in In(u_i), 0 \leq i \leq 1$
(C7b)	$\Pr(u_{i+1} = v) = \frac{\sqrt{c}}{\lvert In(u_i)\rvert}$ for $v \in In(u_i), 1 < i < t$

In this way, the probability that two nodes u and v being in a same tree is $\Pr(f(\pi_u, \pi_v) = 1) + \sum_{i=2}^{t} \Pr(f(\pi_u, \pi_v) = i) \times c^{i-2} = s'(u, v)/c^2$, so giving a coexisting occurrence a weight of c^2 can capture $s'(u, v)$.

In order to combine the neighborhood exact search booster with the neighbourhood average technique, when approximating $s(u, v)$, we consider the relationship between u and v' where $v' \in In(v)$. We define $s'_u(v')$ and $p_u(v')$ as follows:

$s'_u(v') = \sum_{u' \in In(u)} \frac{s'_{t-1}(u',v')}{|In(u)|}$
$p_u(v') = \sum_{u' \in In(u)} \frac{\Pr(f(\pi_{u'},\pi_{v'})=1) \times (c-c^2) + \mathbb{1}_{u'=v'}}{|In(u)|}$

where $s'_{t-1}(u', v')$ has the same definition as $s'(u', v')$ except that t is replaced by $t-1$. Then, these terms are embedded into $s(u', v')$ in Equation (10), $s_u(v') = s'_u(v') + p_u(v')$, and Equation (11) becomes

$$s(u,v) = c \times \sum_{v' \in In(v)} \frac{s'_u(v')}{|In(v)|} + c \times \sum_{v' \in In(v)} \frac{p_u(v')}{|In(v)|} \quad (12)$$

In Algorithm 3, we obtain $p_u(v')$ by local search, and $\overline{s'_u}(v')$ is the average of r simulations to approximate $s'_u(v')$, then they are used to compute $s_2(u, v)$ by

$$s_2(u,v) = c \times \sum_{v' \in In(v)} \frac{\overline{s'_u}(v')}{|In(v)|} + c \times \sum_{v' \in In(v)} \frac{p_u(v')}{|In(v)|} \quad (13)$$

The worst-case runtime of Algorithm 3 is $O(r|V|+|E|)$, since the time for local search is $O(|E|)$ and that to check coexisting nodes is $O(r|V|)$. The index size becomes $O(r|V|+|E|)$ due to extra $O(E)$ size to keep the original graph for local search. Theorem 3 shows the accuracy improvement after local search is adopted. Although local search is also used in previous work [23], the processes are very different, and our local search boosters based on different derivations of SimRank formula give rise to significantly improved theoretical guarantees, which is not the case for [23].

Algorithm 3: One-to-all Querying with Local Search

```
Input: $Next_i$, $u$, $G$
Output: $s_2(u,*)$
for $u' \in In(u)$ do
    for $w \in In(u')$ do
        for $v' \in Out(w)$ do
            if $u' \neq v'$ then
                $p_u(v') += (c - c^2)/|In(u')|/|In(v')|/|In(u)|$;
    $p_u(u') += 1/|In(u)|$;
for $i = 1 \to r$ do
    $u' \leftarrow$ a random node in $In(u)$;
    $v' \leftarrow Next_i[u']$;
    while $v' \neq u'$ do
        $\overline{s'_u}(v') += c^2/r$;
        $v' \leftarrow Next_i[v']$;
for $v' \in V$ with $\overline{s'_u}(v') + p_u(v') > 0$ do
    for $v \in Out(v')$ do
        $s_2(u,v) += c \times (\overline{s'_u}(v') + p_u(v'))/|In(v)|$;
$s_2(u,u) \leftarrow 1$;
return $s_2(u,*)$;
```

THEOREM 3.

$$\Pr(|s_2(u,v) - s(u,v)| > \epsilon) < 2e^{-2r\epsilon^2/c^6} \quad (14)$$

PROOF. In each simulation, $s'_u(v')$ grows by 0 or c^2, which means the increase of $\sum_{v' \in In(v)} \frac{s'_u(v')}{|In(v)|}$ is in $[0, c^2]$. From Hoeffding's inequality, we can prove that $\Pr(|\sum_{v' \in In(v)} \frac{s'_u(v')}{|In(v)|} - \sum_{v' \in In(v)} \frac{\overline{s'_u}(v')}{|In(v)|}| > \epsilon) < 2e^{-2r\epsilon^2/c^4}$
$\Pr(|c\sum_{v' \in In(v)} \frac{s'_u(v')}{|In(v)|} - c\sum_{v' \in In(v)} \frac{\overline{s'_u}(v')}{|In(v)|}| > \epsilon) < 2e^{-2r\epsilon^2/c^6}$
In addition, the exact value of $c \times \sum_{v' \in In(v)} \frac{p_u(v')}{|In(v)|}$ can be obtained by local search. Based on Equation (12), (13) and the above inequality, we can show $\Pr(|s_2(u,v) - s(u,v)| > \epsilon) < 2e^{-2r\epsilon^2/c^6}$ □

6. HANDLING DYNAMIC UPDATES

In many real life applications, graphs are not only large but also dynamic. READS can readily support efficient incremental updates, as described below. We first discuss the natural adaption from the static method and its maintenance,then an online reversed random walk enhancement for accuracy improvement(from Theorem 3 to 4) and index size reduction is introduced next.

6.1 SA Forests for Update Maintenance

Updates to a graph can be in the form of edge or vertex insertions and deletions. For a static graph, the leaf nodes of the simulated SA trees are sufficient in supporting SimRank querying. However, updating the graph may result in changes in the non-leaf levels as tree merging or splitting can be triggered. Thus, our index is upgraded to a forest form to capture dynamic updates on trees. The SA trees generated during indexing are kept in the index.

Since each tree node keeps exactly one vertex, and each vertex appears at most once in any given level of all the trees in one simulation, the vertex ID can serve as an identifier among nodes in the same level of the same simulation.

In the indexing phase, all the tree nodes at the same level of all the trees in a simulation are grouped together. In the bottom level of the trees in the i-simulation, we use an array $\mathbb{L}_i$ with n elements to keep the n leaf nodes, each with an ID as the key and a tuple ($father\_ID$, $left\_sibling\_ID$, $right\_sibling\_ID$) as the value, in a form of

$$\mathbb{L}_i[ID] = (\text{father\_ID, left\_sibling\_ID, right\_sibling\_ID})$$

The left sibling of the leftmost leaf node of a tree is set to $NULL$, and so is the right sibling of the rightmost leaf node. Nodes in higher levels become sparse, because of the merging effect of nodes in lower levels and the $\sqrt{c}$ probability in the sampling of a father node. Therefore, we adopt a hashing scheme for effective storage of the inner nodes and efficient look-up. All non-leaf nodes at the j-th level of the i-th simulation are grouped in a hash table h_{ij}, using its node ID as key and its affiliated information as value, thus,

$$h_{ij}[ID] = (\text{father\_ID, leftmost\_leaf\_ID, rightmost\_leaf\_ID})$$

In the forest form, visiting the left and right siblings of a leaf node u one-by-one serves the same purpose of visiting $Next_i$ in Algorithms 2 and 3, so these querying algorithms can be directly applied to dynamic cases.

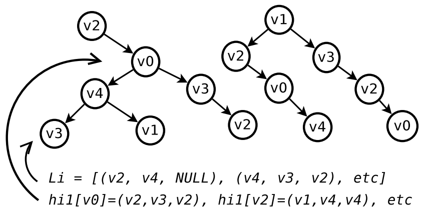
v2
v0
v4
v3
v1
v3
v2
v1
v2
v3
v0
v2
v4
v0
Li = [(v2, v4, NULL), (v4, v3, v2), etc]
hi1[v0]=(v2,v3,v2), hi1[v2]=(v1,v4,v4), etc

Figure 3: Forest Form of Dynamic Index

An example is given in Figure 3. For the bottom level, the elements of the array are $\mathbb{L}_i[v_0] = (v_2, v_4, NULL), \mathbb{L}_i[v1] = (v_4, v_3, v_2)$, etc. The level above the bottom level is kept in h_{i0}, and its upper level is h_{i1} as shown, containing $h_{i1}[v_0] = (v_2, v_3, v_2), h_{i1}[v2] = (v_1, v_4, v_4)$, etc.

6.2 Edge Modification

Since vertex modification can be treated as a trivial case, we discuss how to handle edge insertion and deletion here.

Inserting an edge ($u \leftarrow v$) into the graph may cause changes in every occurrence of u in an SA forest. Let us refer to the node at level i in the SA forest that contains vertex v as $\mathbb{N}_i(v)$. In the following, we may denote $\mathbb{N}_i(v)$ by $\mathbb{N}(v)$ for clarity. Note that only one level i node may contain any vertex $v \in V$. Let $father(a)$ be the father node of a in the SA forest, if one exists. We enumerate the occurrences of ($\mathbb{N}(u) \leftarrow father(\mathbb{N}(u))$ one by one, and for each occurrence, we replace the old father by a node $\mathbb{N}(v)$ with a probability $1/new\_degree(u)$. If such a replacement takes place, the replacement is to firstly detach the subtree rooted at u, and then insert this subtree to the tree node with vertex ID v one level above u. If no such tree node exists prior to the update, a path is grown from v until level t is reached or a successful insertion takes place. The process for edge removal is similar. Deleting edge ($u \leftarrow v$) in the graph leads to removing each ($\mathbb{N}(u) \leftarrow \mathbb{N}(v)$) occurrence in the forest, finding a new father for $\mathbb{N}(u)$ and then inserting the subtree rooted at $\mathbb{N}(u)$.

The analysis at the end of Section 4.4 shows that the expected size of the dynamic index is $O(|V|r)$, the same as the static case. The indexing time and querying time is also the same as the static case. For each edge update, $O(rt)$ time is used to find the occurrences of the modified edge, and $O(t)$ time is used to deal with each triggered subtree modification. For an edge insertion, the starting vertex appears $O(|V|r/|V|) = O(r)$ times in the forest, and $|V|/|E|$ of them triggers a subtree modification. While for an edge deletion, an edge appears $O(|V|r/|E|)$ times in the forest. Therefore, the average update time is $O(rt|V|/|E|+rt) = O(rt)$.

6.3 Online Reversed Random Walk

In previous algorithms, a forest generated in a simulation is used only once to evaluate the similarity between a vertex (u in Algorithm 2, or u' in Algorithm 3) and other vertices. The structure of our dynamic index makes it possible to use a forest multiple times to increase the accuracy. The main idea is to generate multiple reversed random walks online, and each of them is matched with a forest in the index.

With the help of online reversed random walk, SA forests can be modified to improve accuracy and reduce the index size. We replace Condition (C7b) by Condition (C7c), and generate walks by Conditions (C1)-(C4), (C7a) and (C7c). Such a walk is called **c-SA walk**. The forests formed by these walks are called **c-SA Forests**. We store c-SA forests in the same format as SA forests by $\mathbb{L}_i$ and h_{ij}.

(C7c)	$\Pr(u_{i+1} = v) = \frac{c}{\lvert In(u_i)\rvert}$ for $v \in In(u_i), 1 < i < t$

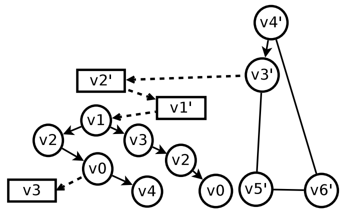

Figure 4: Online Sampling from v_3

The following example shows how an online reversed random walk is generated and how it is matched with a c-SA forest in the index.

Example 1. *Suppose the c-SA forest in the index is formed by the tree nodes in circles and solid line segments in Figure 4, and we want to obtain the similarity between v_3 and other vertices. Tree nodes in squares and dotted line segments are generated on the fly. Together with some tree nodes and edges in the index, they form an online reversed random walk. The walk ($v_3 \leftarrow v_0 \leftarrow v_2 \leftarrow v_1 \leftarrow v'_1 \leftarrow v'_2 \leftarrow v'_3 \leftarrow v'_4$) is an online reversed random walk. It is matched with leaf nodes $v_4, v_0, v'_5, ..., v'_6$.*

For each c-SA forest, we generate r_q reversed random walks online. In general, an online reversed random walk starts from a query node u' (equivalent with u' in line 7 Algorithm 3) at leaf level, and it randomly selects an incoming neighbor as a tree node at level 1. In the remaining steps, if a tree node at a particular level can be found in the c-SA forest and it has a father, then the edge to its father is used as the next edge on the walk. Otherwise, it randomly selects an incoming neighbor as the next tree node in a higher level. The walk continues until its length reaches depth t or when the last node has no incoming neighbor. The similarity between u' and each leaf node of each crossed tree in a c-SA forest increases by $c^2/r/r_q$ if there are r c-SA forests and r_q online reversed random walks for each c-SA forests. In our implementation, we use a hashmap *cnt* to count the number of times that each root node is visited, so that after r_q walks there is only one weight update of its leaf nodes.

Although an online reversed random walk is not explicitly generated by Conditions (C1)-(C5) as in the definition of reversed random walk, it also satisfies Conditions (C1)-(C5). For a tree node with its father absent in the current c-SA forest, clearly, selecting a random incoming neighbor satisfies Condition (C5). In the case when a tree node and its father are found in the current c-SA forest according to Condition (C7c), and based on the condition that it has a father, the probability of each incoming neighbor v as its father u_{i+1} is $\Pr(u_{i+1} = v | u_{i+1} \neq null) = \Pr(u_{i+1} = v, u_{i+1} \neq null)/\Pr(u_{i+1} \neq null) = \Pr(u_{i+1} = v)/c = 1/|In(u_i)|$. Condition (C5) also holds in this case. Clearly, Conditions (C1)-(C4) always hold. Therefore, an online reversed random walk generated as the above is a reversed random walk.

Algorithm 4: Online Reversed Random Walk Querying

```
   Input: $\mathbb{L}_i$ and $h_{ij}$, $u$, $G$, $r_q$
   Output: $s_3(u,*)$
1  Line 1-6 in Algorithm 3
2  for $i = 1 \rightarrow r$ do
3      $cnt \leftarrow$ an empty hashmap;
4      for $j = 1 \rightarrow r_q$ do
5          $u' \leftarrow$ a random node in $In(u)$;
6          if $In(u') = \emptyset$ then
7              continue;
8          Queue $q_0 \leftarrow$ a random node in $In(u')$;
9          for $k = 0 \rightarrow t-2$ and $In(q_k) \neq \emptyset$ do
10             if $h_{ik}[q_k] \neq \emptyset$ then
11                 if $h_{ik}[q_k].father\_ID = NULL$ then
12                     $cnt_k[q_k]++$;
13                     $q_{k+1} \leftarrow$ a random node in $In(q_k)$;
14                 else
15                     $q_{k+1} \leftarrow h_{ik}[q_k].father\_ID$ ;
16             else
17                 $q_{k+1} \leftarrow$ a random node in $In(q_k)$;
18     for $v \in cnt_j[v]$ with $cnt_j[v] > 0$ do
19         for $v' \in h_{ij}[v].leaf\_nodes$ do
20             $\overline{s'_u}(v') += cnt_j[v] \times c^2/r/r_q$;
21 for $v' \in V$ with $\overline{s'_u}(v') + p(v') > 0$ do
22     for $v \in Out(v')$ do
23         $s_3(u,v) += c \times (\overline{s'_u}(v') + p_u(v'))/|In(v)|$;
24 $s_3(u,u) \leftarrow 1$;
25 return $s_3(u,*)$;
```

Algorithm 4 upgrades Algorithm 3 with the use of r c-SA forests and r_q online reversed random walks for each forest. Denote the approximated SimRank value as s_3. Its querying time is $O(rr_qt + r|V|+|E|)$. Next, we show that online sampling approach improves the theoretical guarantee without increasing the index size. Our experiments show that the r_q factor contributes a lot to accuracy in practice with very little overhead on the querying time.

LEMMA 3 (BERNSTEIN [4]). *Let $X_1, X_2, ..., X_r$ be i.i.d random variables where X_i are strictly bounded by the interval $[a,b]$, let $\overline{X} = \frac{1}{r}(X_1 + ... + X_r)$, $Var[X]$ is the variance of X_i. $\forall \epsilon > 0$, Then, for all $\epsilon > 0$:* $\Pr(|\overline{X} - E[\overline{X}]| \geq \epsilon) \leq 2e^{\frac{-r\epsilon^2}{2Var[X]+2\epsilon(b-a)/3}}$.

THEOREM 4. $\Pr(|s_3(u,v) - s(u,v)| > \epsilon) < 2e^{\frac{-r\epsilon^2}{\frac{c^6}{2r_q}+\frac{2\epsilon c^3}{3}}}$

PROOF. In the j-th online sampling of the i-th simulation, we denote the increase of $s'_u(v')$ by $s'_{ijv'}$. In the i-th simulation, we denote the overall increase of $s'_u(v')$ by $s'_{iv'} = \sum_{j=1}^{r_q} s'_{ijv'}/r_q$, and the overall increase of $\sum_{v' \in In(v)} \frac{s'_u(v')}{|In(v)|}$ by $X_i = \sum_{v' \in In(v)} \frac{s'_{iv'}}{|In(v)|}$. Since each $s'_{ijv'} \in \{0, c^2\}$, $s'_{iv'} \in [0, c^2]$, we can conclude that $X_i \in [0, c^2]$. As the r simulations are independent of each other, we can consider r as the number of independent variables X_i, and $[0, c^2]$ as the value range in Bernstein's inequality.

Regarding the variance, $s'_{ijv'} \in \{0, c^2\}$ implies that $Var[s'_{ijv'}] \leq c^4/4$. For each i and $j \neq j'$, $s'_{ijv'}$ and $s'_{ij'v'}$ are independent because they are from two independent online reversed random walks. It means $Var[s'_{iv'}] \leq c^4/4r_q$. Although X_i is the average of several $s'_{iv'}$ that may be correlated, its largest value is bounded by $Var[X_i] \leq c^4/4r_q$. Adopting Bernstein's inequality for X_i,

$$\Pr\left(\left|\sum_{v' \in In(v)} \frac{s'_u(v')}{|In(v)|} - \sum_{v' \in In(v)} \frac{\overline{s'_u}(v')}{|In(v)|}\right| > \epsilon\right) < 2e^{\frac{-r\epsilon^2}{\frac{c^4}{2r_q}+\frac{2\epsilon c^2}{3}}}$$

$$\Pr\left(\left|c \times \sum_{v' \in In(v)} \frac{s'_u(v')}{|In(v)|} - c \times \sum_{v' \in In(v)} \frac{\overline{s'_u}(v')}{|In(v)|}\right| > \epsilon\right) < 2e^{\frac{-r\epsilon^2}{\frac{c^6}{2r_q}+\frac{2\epsilon c^3}{3}}}$$

Since s_3 is also computed based on Equation (13), with Equation (12) and the exact value of $p_u(v')$ obtained by local search, we get $\Pr(|s_3(u,v) - s(u,v)| > \epsilon) < 2e^{\frac{-r\epsilon^2}{\frac{c^6}{2r_q}+\frac{2\epsilon c^3}{3}}}$. □

The above proof implies that multiple online walks create a more stable $s'_u(v')$, so that the final aggregated SimRank value is more accurate. Apart from accuracy improvement, adopting online reversed random walk also reduces the index size, as the stopping probability of walks in the index is increased from $1 - \sqrt{c}$ to $1 - c$.

We are aware of the use of R_q online walks in [21]. However, our method is very different from [21], and the overhead in query time of our method is very small, which is $O(rr_qt)$, versus an overall increase of a factor of R_q in [21].

7. EXPERIMENTS

We show the strength of our proposed algorithms with experiments in this section. The experiments are conducted on a PC with 2.3 GHz CPU and 196 GB memory running Linux complied by G++. We implemented our **READS** (Algorithm 3 static), **READS-D** (Algorithm 3 dynamic) and **READS-Rq** (dynamic Algorithm 4 with online walks). The major competitors are the dynamic solutions **TopSim** [12], **TSF**, and **L-TSF** [21], which is a log-based implementation of TSF supporting dynamic updates. They are implemented by the authors of [21]. We also compare with static algorithms **FR-SR** [5] and Algorithm 6 of **SLING** [23], which are implemented by the authors of [23]. We do not compare with [16] and [11] since their performance is dominated by SLING and TSF, as shown in [23] and [21]. We set $r = 100$, $r_q = 10$, and $t = 10$ for our algorithms. We set $R_g = 100$, $R_q = 20$ for TSF/L-TSF, and $T = 10$, $R = 100$ for FR-SR as suggested by [21] and [5]. We set $T = 3$ and adopt the PrioTopSim strategy for TopSim as previous work [29]. We set ϵ in SLING as 0.25 to generate indices that are of comparable sizes as with the others.

Datasets We use 14 datasets, listed in the following table, and downloaded from SNAP [1] or KONECT [2]. We have

[1]https://snap.stanford.edu/data/
[2]http://konect.uni-koblenz.de/networks/

selected graphs from different categories. The first 4 graphs are of small sizes, so that a brute force method computing the exact SimRank value is applicable. We summarize the nature of each graph here. HP is a network for human protein, CA is a collaboration network, AD is a social network, WV is a network about elections, WS is a network from the Stanford webpage, WG is a graph on Google webpage, DB is a network for entities, UP is a citation network, WP is an article network, LJ is another social network, and WD is an article network, WF is a communication network, WL is a hyperlink network, and DL is another hyperlink network.

Graph	$\|V\|$	$\|E\|$	**Description**
HumanProtein(**HP**)	3,133	6,726	undirected
ca-GrQc(**CA**)	5,242	14,496	undirected
Advogato(**AD**)	6,541	51,127	directed
Wiki-Vote(**WV**)	7,115	103,689	directed
web-Stanford(**WS**)	281,903	2,312,497	directed
web-Google(**WG**)	875,713	5,105,039	directed
WikiPolish(**WP**)	1,033,050	25,026,208	directed
WikiDe(**WD**)	2,166,669	86,337,879	directed
WikiFrench(**WF**)	3,023,165	102,382,410	directed
US-Patents(**UP**)	3,774,768	16,518,947	directed
DBpedia(**DB**)	3,966,924	13,820,853	directed
live-journal(**LJ**)	4,847,571	68,475,391	directed
WikiLink(**WL**)	12,150,976	378,142,420	directed
DbpediaLink(**DL**)	18,268,992	172,183,984	directed

7.1 Precision

In top-k querying, we measured the percentage of approximated top-k nodes among the ground-truth top-k nodes as precision to evaluate the quality of different algorithms.

$$Precision = \frac{|approximated\_top\_k\_set \cap exact\_top\_k\_set|}{k}$$

The brute-force algorithm in [9] serves as the ground truth to obtain pairwise SimRank values, and we only compare the precisions on the first 4 small datasets.

We derive top-k results by selecting the top k SimRank values from the single source solution set computed by the algorithms. Since the results of READS and READS-D are identical, we only show READS in Figure 5, and likewise for the case of TSF and L-TSF. The results show that READS and READS-Rq dominate the existing two methods SLING and TSF in all datasets. With the increase of k, the precision of SLING falls rapidly. This can be explained by the indexing algorithm of SLING, which deterministically prunes search space with contribution below its additive error allowance. The accuracy is high for computing high rank values, but the result quality may deteriorate when smaller rank values appear in the true top-k result, which will be the case when k is large. Although TSF can achieve similar precision as READS in some datasets, its performance is not stable, with rather low precision in some datasets, e.g. AD and WV. This is because the probabilistic guarantee for the error bound for TSF is based on the assumption that no cycle in the given graph has a length shorter than t. There can be cases where the assumption and the final guarantee do not hold. The theoretical error bound of FR-SR is worse than ours, and the empirical results verify the differences in precision. TopSim uses a heuristic strategy to prune paths among all paths in a small region, but the imprecise pruning and only counting paths limited in a small region lead to a low precision in experiments.

READS-Rq is more accurate than READS due to the variance reduction via r_q online reversed random walks, as proved in Theorem 4. Comparing READS with No-Local, which is Algorithm 2 without using local search boosters, the effect of switching from the guarantee in Theorem 2 to that of Theorem 3 is very significant. Non-Opt is a version of Algorithm 2 with $1 - \sqrt{c}$ stopping probability at every step, which is not optimal as proven in Section 4.3. The improvement from Non-Opt to No-Local shows the significance of using a zero stopping probability at Step 1.

7.2 Querying Efficiency

In testing querying efficiency, we randomly generated 1,000 query nodes as in [21]. In Figure 6, the average single-source query times of all algorithms are presented as a histogram, and the standard deviations of the query times are shown as curves. The algorithms are grouped into two categories; dynamic algorithms and static algorithms.

In particular, READS-D, READS-Rq, L-TSF and TopSim are dynamic. The difference between the first four bars in each dataset shows that READS-D and READS-Rq are faster than L-TSF and TopSim by 1 to 3 orders of magnitude. READS-Rq is slightly slower than READS-D but the overhead of extra $O(rr_qt)$ running time is not too much. Its improvement on accuracy shows that it is a good trade-off between querying cost and accuracy. The standard deviations show that READS and READS-Rq are much more stable than L-TFS and TopSim. Among the static datasets, READS, SLING and FR-SR are much more efficient than TSF in small graphs, while READS dominates the others in large graphs, being several times faster compared to TSF, SLING and FR-SR. As main competitors of our algorithms, either TSF or L-TSF answers queries much slower than our algorithms READS, READS-D and READS-Rq.

The dynamic version L-TSF is typically several times slower than the static version of TSF. In order to support dynamic updates efficiently, a log-based implementation was employed in L-TSF which groups updates together, leading to a slower querying efficiency. In contrast, the gap between READS-D and READS is very small, showing that our approach supports dynamic updates more naturally.

Moreover, the real running times of 10 random queries on live-journal are shown in Figure 11. In both dynamic and static cases, our algorithms support very efficient querying.

7.3 Dynamic Update Cost

For datasets WP and WD with real timestamps on edges, we insert and remove the last 1000 edges to the graph to measure the cost of real dynamic updates. For the others, we follow the setting in [21], randomly choosing 1,000 edges, and considering 80% of them as insertions and 20% as deletions. As TopSim is an index-free method, it is not list in this experiment, similar for the case of indexing cost. The average time of READS-D, READS-Rq index and L-TSF index for each update is reported in Figure 7. In most graphs, the efficiencies of READS-D and READS-Rq are much better than L-TSF, faster by about 1 order of magnitude. READS-Rq is slightly faster than READS-D due to its smaller index size and higher stopping probability.

Without any mechanism supporting dynamic updates, we have to rebuild the index from scratch with highly expensive indexing cost, as shown in Figure 9. For example, it takes thousands of seconds to rebuild the index for live-journal, and the rebuilding must be carried out repeatedly with updates. With READS-D and READS-Rq, each update is han-

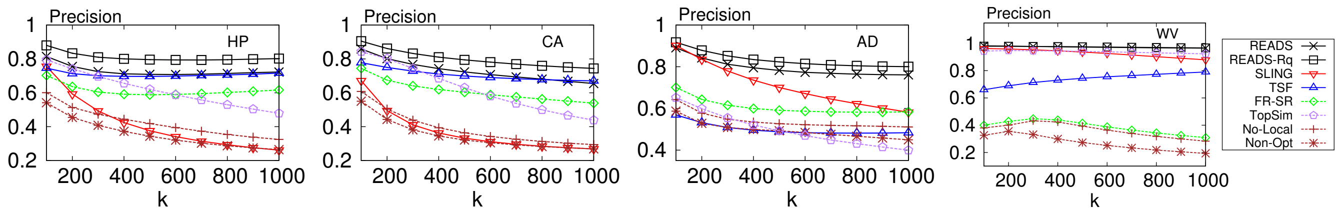

Figure 5: Top-k Querying Precision

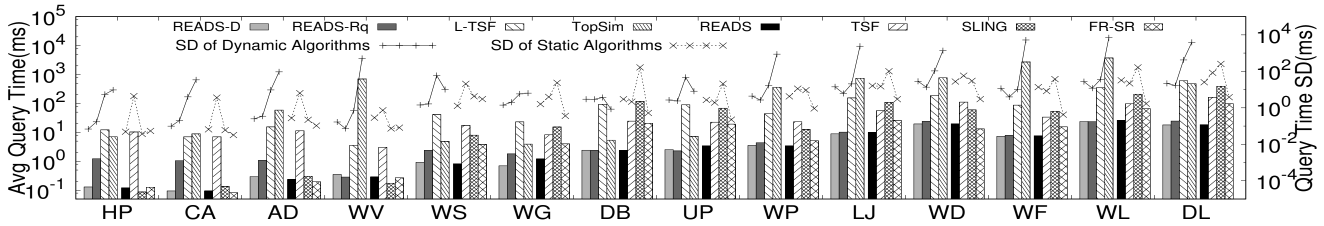

Figure 6: Query Time : the bars show the average query time(left y-axis) and the line curves show the query time standard deviation(right y-axis)

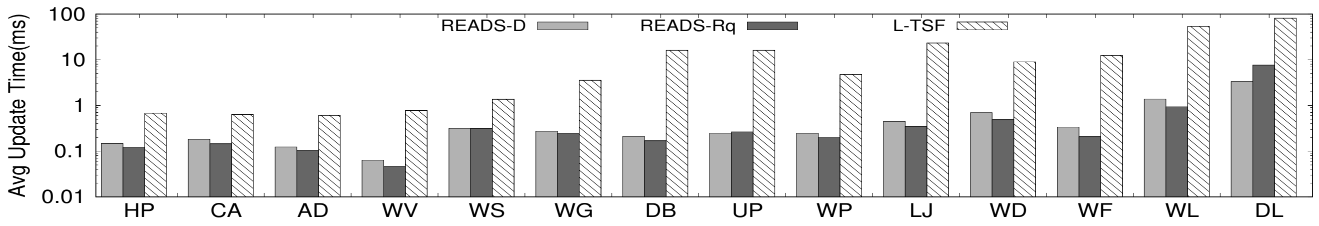

Figure 7: Update Time per Edge Insertion/Deletion

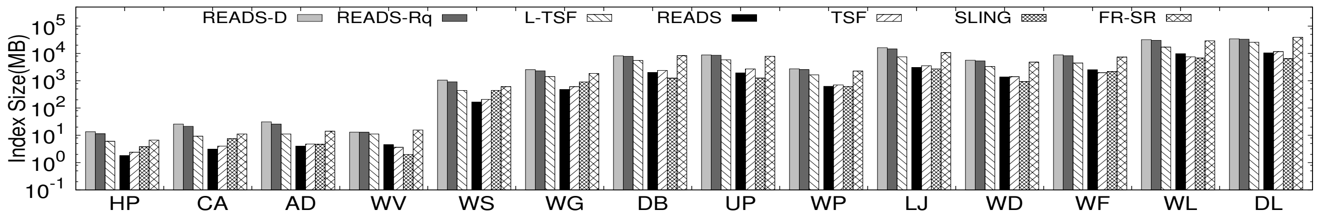

Figure 8: Index Size

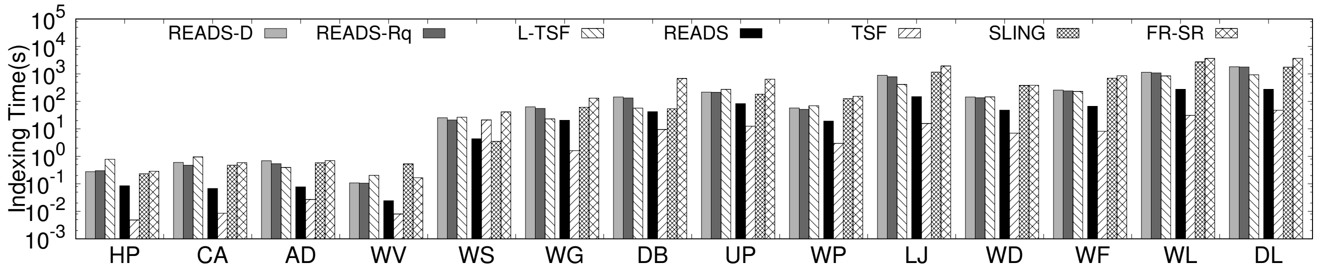

Figure 9: Indexing Time

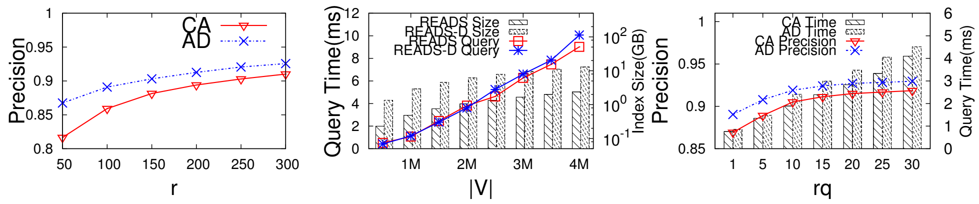

Figure 10: Scalability Varying r, Graph Size, and r_q

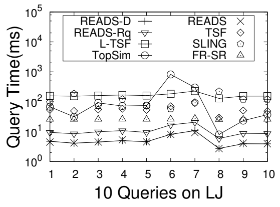

Figure 11: Querying on LJ

dled in about 1 millisecond. The almost 10 times speedup in update efficiency over the state-of-the art dynamic L-TSF illustrates the strength of our method.

7.4 Indexing Cost

The index sizes in Figure 8 are also grouped into dynamic and static categories. In each category, the index sizes of different algorithms are similar. The sizes of READS-D and READS-Rq are slightly larger than L-TSF, but they are still small enough to be put in the main memory of a commodity machine for million-vertex graphs. Among the static algorithms, READS produces the smallest index in 6 of the 14 datasets. Figure 9 shows that all algorithms can generate an index with an acceptable preprocessing time. Although TSF is faster in preprocessing, its querying efficiency is poor, which is an undesirable trade-off.

7.5 Scalability

For scalability testing, we vary r in READS/READS-D and show the trend of precision of top-k querying on ca-GrQc and Advogato when $k = 100$. In order to evaluate the performance with the increase of graph size, we also randomly sample subgraphs of live-journal from $|V| = 0.5M$ to $|V| = 4M$ for our scalability test. From the results in Figure 10, the precision of top-k queries grows with r, but the increase becomes slower for larger r values. The query time and index size increase linearly with $|V|$, reflecting the time and space complexity of $O(|V|r)$. The index size gap between READS and READS-D is caused by the additional tree structure for dynamic updates, but with the increase of graph size, the gap remains a constant, implying that the dynamic version is stable and scalable. The effect of varying r_q in READS-Rq in Figure 10 implies that the improvement of r_q is significant when r_q is small, and the trade-off on querying time does not increase too much.

8. CONCLUSION

We propose READS, an indexing scheme for SimRank computation on a large dynamic graph. The scheme makes use of a new generalized notion of random walk sampling. Two optimization strategies further boost the accuracy of READS. Our experiments show that READS outperforms the state-of-the-art algorithms significantly in terms of precision, query time, and update time.

Acknowledgements This research is supported in part by the Hong Kong RGC/GRF research grant 412313 Project ID 2150758. The research of Minhao Jiang and Raymond Chi-Wing Wong is supported by HKRGC GRF 16219816.

9. REFERENCES

[1] Z. Abbassi and V. S. Mirrokni. A recommender system based on local random walks and spectral methods. *WebKDD/SNA-KDD, Lecture Notes in Computer Science*, 5439:139–153, 2007.

[2] I. Antonellis, H. Garcia-Molina, C.-C. T. Chang, and X. Xiao. SimRank++: Query rewriting through link analysis of the click graph. In *PVLDB*, 2008.

[3] A. Benczur, K. Csalogany, and T. Sarlos. Link-based similarity search to fight web spam. In *AIRWeb*, 2006.

[4] S. Bernstein. *The theory of probabilities*. Gastehizdat Publishing House, Moscow, 1946.

[5] D. Fogars and B. Racz. Scaling link-based similarity search. In *WWW*, pages 641–650. ACM, May 2005.

[6] Y. Fugiwara, M. Nakatsuji, H. Shiokawa, and M. Onizuka. Efficient search algorithm for simrank. In *ICDE*, 2013.

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

[8] W. Hoeffding. Probability inequalities for sums of bounded random variables. *Journal of the American Statistical Association*, 58(301):13–30, 1963.

[9] G. Jeh and J. Widom. SimRank: A measure of structural-context similarity. In *KDD*, 2002.

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

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

[12] P. Lee, L. Lakshmannan, and J. Yu. On top-k structural similarity search. In *ICDE*, pages 774–785. IEEE, 2012.

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

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

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

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

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

[18] R. Mihalcea and D. Radev. *Graphbased natural language processing and information retrieval*. Cambridge University Press, 2011.

[19] P. T. Nguyen, P. Tomeo, T. D. Noia, and E. D. Sciascio. An evaluation of SimRank and personalized pagerank to build a recommender system for the web of data. In *WWW*, 2015.

[20] C. Scheible, F. Laws, L. Michelbacher, and H. Schutze. Sentiment translation through multi-edge graphs. In *In Proceedings of the 23rd International Conference on Computational Linguistics: COLING*, 2010.

[21] Y. Shao, B. Cui, L. Chen, M. Liu, and X. Xie. An efficient similarity search framework for simrank over large dynamic graphs. In *VLDB*, 2015.

[22] W. Tao, M. Yu, and G. Li. Efficient top-k simrank-based similarity join. In *VLDB*, pages 317–328, 2014.

[23] B. Tian and X. Xiao. Sling: A near-optimal index structure for SimRank. In *SIGMOD*. ACM, 2016.

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

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

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

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

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

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

[30] W. Zheng, L. Zou, Y. Feng, and D. Zhao. Efficient SimRank-based similarity join over large graphs. *PVLDB*, 6(7), 2013.

[31] Y. Zhou, H. Cheng, and J. X. Yu. Graph clustering based on structural/attribute similarities. *PVLDB*, 2(1), 2009.