SLING: A Near-Optimal Index Structure for SimRank

Boyu Tian
Shanghai Jiao Tong University
China
bytian@umich.edu

Xiaokui Xiao
Nanyang Technological University
Singapore
xkxiao@ntu.edu.sg

ABSTRACT

SimRank is a similarity measure for graph nodes that has numerous applications in practice. Scalable SimRank computation has been the subject of extensive research for more than a decade, and yet, none of the existing solutions can efficiently derive SimRank scores on large graphs with provable accuracy guarantees. In particular, the state-of-the-art solution requires up to a few seconds to compute a SimRank score in million-node graphs, and does not offer any worst-case assurance in terms of the query error.

This paper presents *SLING*, an efficient index structure for SimRank computation. *SLING* guarantees that each SimRank score returned has at most ε additive error, and it answers any single-pair and single-source SimRank queries in $O(1/\varepsilon)$ and $O(n/\varepsilon)$ time, respectively. These time complexities are *near-optimal*, and are significantly better than the asymptotic bounds of the most recent approach. Furthermore, *SLING* requires only $O(n/\varepsilon)$ space (which is also near-optimal in an asymptotic sense) and $O(m/\varepsilon + n \log \frac{n}{\delta}/\varepsilon^2)$ pre-computation time, where δ is the failure probability of the preprocessing algorithm. We experimentally evaluate *SLING* with a variety of real-world graphs with up to several millions of nodes. Our results demonstrate that *SLING* is up to 10000 times (resp. 110 times) faster than competing methods for single-pair (resp. single-source) SimRank queries, at the cost of higher space overheads.

1. INTRODUCTION

Assessing the similarity of nodes based on graph topology is an important problem with numerous applications, including social network analysis [21], web mining [16], collaborative filtering [5], natural language processing [26], and spam detection [27]. A number of similarity measures have been proposed, among which *SimRank* [14] is one of the most well-adopted. The formulation of SimRank is based on two intuitive arguments:

- A node should have the maximum similarity to itself;
- The similarity between two different nodes can be measured by the average similarity between the two nodes' neighbors.

Formally, the SimRank score of two nodes v_i and v_j is defined as:

$$s(v_i, v_j) = \begin{cases} 1, & \text{if } v_i = v_j \\ \dfrac{c}{|I(v_i)| \cdot |I(v_j)|} \displaystyle\sum_{a \in I(v_i), b \in I(v_j)} s(a, b), & \text{otherwise} \end{cases} \quad (1)$$

where $I(v)$ denotes the set of in-neighbors of a node v, and $c \in (0, 1)$ is a decay factor typically set to 0.6 or 0.8 [14, 23]. Previous work [5, 8, 16, 21, 22, 26, 27, 33, 35] has applied SimRank (and its variants) to various problem domains, and has demonstrated that it often provides high-quality measurements of node similarity.

1.1 Motivation

Despite of the effectiveness of SimRank, computing SimRank scores efficiently on large graphs is a challenging task, and has been the subject of extensive research for more than a decade. In particular, Jeh and Widom [14] propose the first SimRank algorithm, which returns the SimRank scores of all pairs of nodes in the input graph G. The algorithm incurs prohibitive costs: it requires $O(n^2)$ space and $O(m^2 \log \frac{1}{\varepsilon})$ time, where n and m denote the numbers of nodes and edges in G, respectively, and ε is the maximum additive error allowed in any SimRank score. Subsequently, Lizorkin et al. [23] improve the time complexity of the algorithm to $O(\log \frac{1}{\varepsilon} \cdot \min\{nm, n^3/\log n\})$, which is further improved to $O(\log \frac{1}{\varepsilon} \cdot \min\{nm, n^\omega\})$ by Yu et al. [34], where $\omega \approx 2.373$. However, the space complexity of the algorithm remains $O(n^2)$, as is inherent in any algorithm that computes *all-pair* SimRank scores.

Fogaras and Rácz [8] present the first study on *single-pair* SimRank computation, and propose a Monte-Carlo method that requires $O(n \log \frac{1}{\delta}/\varepsilon^2)$ pre-computation time and space. The method returns the SimRank score of any node pair in $O(\log \frac{1}{\delta}/\varepsilon^2)$ time, where δ is the failure probability of the Monte-Carlo method. Subsequently, Li et al. [20] propose a deterministic algorithm for single-pair SimRank queries; it has the same time complexity with Jeh and Widom's solution [14], but provides much better practical efficiency. However, existing work [24] show that neither Li et al.'s [20] nor Fogaras and Rácz's solution [8] is able to handle million-node graphs in reasonable time and space. There is a line of research [10, 13, 19, 30–32] that attempts to mitigate this efficiency issue based on an alternative formulation of SimRank, but the formulation is shown to be *incorrect* [17], in that it does not return the same SimRank scores as defined in Equation (1).

The most recent approach to SimRank computation is the *linearization* technique [24] by Maehara et al., which is shown to considerably outperform existing solutions in terms of efficiency and scalability. Nevertheless, it still requires up to a few seconds to answer a single-pair SimRank query on sizable graphs, which is inadequate for large-scale applications. More importantly, the

SIGMOD'16, June 26-July 01, 2016, San Francisco, CA, USA

ISBN 978-1-4503-3531-7/16/06…$15.00

DOI: http://dx.doi.org/10.1145/2882903.2915243

Table 1: Comparison of SimRank computation methods with at most ε additive error and at least $1-\delta$ success probability.

Algorithm	Query Time		Space Overhead	Preprocessing Time
	Single Pair	Single Source		
Fogaras and Rácz [8]	$O\big(\log \frac{1}{\varepsilon} \log \frac{n}{\delta}/\varepsilon^2\big)$	$O\big(n \log \frac{1}{\varepsilon} \log \frac{n}{\delta}/\varepsilon^2\big)$	$O\big(n \log \frac{1}{\varepsilon} \log \frac{n}{\delta}/\varepsilon^2\big)$	$O\big(n \log \frac{1}{\varepsilon} \log \frac{n}{\delta}/\varepsilon^2\big)$
Maehara et al. [24] (under heuristic assumptions)	$O\big(m \log \frac{1}{\varepsilon}\big)$	$O\big(m \log^2 \frac{1}{\varepsilon}\big)$	$O(n+m)$	no formal result
this paper	$O(1/\varepsilon)$	$O(n/\varepsilon)$ (Algorithm 3)	$O(n/\varepsilon)$	$O\big(m/\varepsilon + n \log \frac{n}{\delta}/\varepsilon^2\big)$
		$O\big(m \log^2 \frac{1}{\varepsilon}\big)$ (Algorithm 6)		
lower bound	$\Omega(1)$	$\Omega(n)$	$\Omega(n)$	-

technique is unable to provide any worst-case guarantee in terms of query accuracy. In particular, the technique has a preprocessing step that requires solving a system L of linear equations; assuming that the solution to L is *exact*, Maehara et al. [24] show that the technique can ensure ε worst-case query error, and can answer any single-pair and single-source SimRank queries in $O\big(m \log \frac{1}{\varepsilon}\big)$ and $O\big(m \log^2 \frac{1}{\varepsilon}\big)$ time, respectively. (A single-source SimRank query from a node v_i asks for the SimRank score between v_i and every other node.) Unfortunately, as we discuss in Section 3.3, the linearization technique cannot precisely solve L, nor can it offer non-trivial guarantees in terms of the query errors incurred by the imprecision of L's solution. Consequently, the technique in [24] only provides heuristic solutions to SimRank computation. In summary, after more than tens years of research on SimRank, there is still no solution for efficient SimRank computation on large graphs with provable accuracy guarantees.

1.2 Contributions and Organization

This paper presents *SLING* (*SimRank via Local Updates and Sampling*), an efficient index structure for SimRank computation. *SLING* guarantees that each SimRank score returned has at most ε additive error, and answers any single-pair and single-source SimRank queries in $O(1/\varepsilon)$ and $O(n/\varepsilon)$ time, respectively. These time complexities are *near-optimal*, since any SimRank method requires $\Omega(1)$ (resp. $\Omega(n)$) time to output the result of any single-pair (resp. single-source) query. In addition, they are significantly better than the asymptotic bounds of the states of the art (including Maehara et al.'s technique [24] under their heuristic assumptions), as we show in Table 1. Furthermore, *SLING* requires only $O(n/\varepsilon)$ space (which is also near-optimal in an asymptotic sense) and $O(m/\varepsilon + n \log \frac{n}{\delta})$ pre-computation time, where δ is the failure probability of the preprocessing algorithm.

Apart from its superior asymptotic bounds, *SLING* also incorporates several optimization techniques to enhance its practical performance. In particular, we show that its preprocessing algorithm can be improved with a technique that estimates the expectation of a Bernoulli variable using an *asymptotically optimal* number of samples. Additionally, its space consumption can be heuristically reduced without affecting its theoretical guarantees, while its empirical efficiency for single-source SimRank queries can be considerably improved, at the cost of a slight increase in its query time complexity. Last but not least, its construction algorithms can be easily parallelized, and it can efficiently process queries even when its index structure does not fit in the main memory.

We experimentally evaluate *SLING* with a variety of real-world graphs with up to several millions of nodes, and show that it significantly outperforms the the states of the art in terms of query efficiency. Specifically, *SLING* requires at most 2.3 milliseconds to process a single-pair SimRank query on our datasets, and is up to 10000 times faster than the linearization method [24]. To our knowledge, this is the first result in the literature that demonstrates millisecond-scale query time for single-pair SimRank computation on million-node graphs. For single-source SimRank queries, *SLING* is up to 110 times more efficient than the linearization method. As a tradeoff, *SLING* incurs larger space overheads than the linearization method, but it is a still much more favorable choice in the common scenario where query time and accuracy (instead of space consumption) are the main concern.

The remainder of the paper is organized as follows. Section 2 defines the problem that we study. Section 3 discusses the major existing methods for SimRank computation. Section 4 presents the *SLING* index, with a focus on single-pair queries. Section 5 proposes techniques to optimize the practical performance of *SLING*. Section 6 details how *SLING* supports single-source queries. Section 7 experimentally evaluates *SLING* against the stats of the art

2. PRELIMINARIES

Let G be a directed and unweighted graph with n nodes and m edges. We aim to construct an index structure on G to support *single-pair* and *single-source* SimRank queries, which are defined as follows:

- A single-pair SimRank query takes as input two nodes u and v in G, and returns their SimRank score $s(u, v)$ (see Equation 1).
- A single-source SimRank query takes as input a node u, and returns $s(u, v)$ for each node v in G.

Following previous work [8, 23, 24, 33], we allow an additive error of at most $\varepsilon \in (0, 1)$ in each SimRank score returned for any SimRank query.

For ease of exposition, we focus on single-pair SimRank queries in Sections 3-5, and then discuss single-source queries in Section 6. Table 2 shows the notations frequently used in the paper. Unless otherwise specified, all logarithms in this paper are to base e.

3. ANALYSIS OF EXISTING METHODS

This section revisits the three major approaches to SimRank computation: the *power method* [14], the *Monte Carlo method* [8], and the *linearization method* [17, 24, 25, 33]. The asymptotic performance of the Monte Carlo method and the linearization method has been studied in literature, but to our knowledge, there is no formal analysis regarding their space and time complexities when ensuring ε worst-case errors. We remedy this issue with detailed discussions on each method's asymptotic bounds and limitations.

3.1 The Power Method

The power method [14] is an iterative method for computing the SimRank scores of all pairs of nodes in an input graph. The method

Table 2: Table of notations.

Notation	Description
G	the input graph
n, m	the numbers of nodes and edges in G
v_i	the i-th node in G
$I(v)$	the set of in-neighbors of a node v in G
$s(v_i, v_j)$	the SimRank score of two nodes v_i and v_j in G
c	the decay factor in the definition of SimRank
ε	the maximum additive error allowed in a SimRank score
δ	the failure probability of a Monte-Carlo algorithm
$M(i, j)$	the entry on the i-th row and j-th column of a matrix M
d_k	the correction factor for node v_k
$h^\ell(v_i, v_j)$	the hitting probability (HP) from node v_i to node v_j at step ℓ (see Section 4.2)

uses a $n \times n$ matrix S, where the element $S(i, j)$ on the i-th row and j-th column ($i, j \in [1, n]$) denotes the SimRank score of the i-th node v_i and j-th node v_j. Initially, the method sets

$$S(i,j) = \begin{cases} 1, & \text{if } i = j \\ 0, & \text{otherwise} \end{cases}$$

After that, in the t-th ($t \geq 1$) iteration, the method updates S based on the following equation:

$$S(i,j) = \begin{cases} 1, & \text{if } i = j \\ \dfrac{c}{|I(v_i)||I(v_j)|} \displaystyle\sum_{v_k \in I(v_i), v_\ell \in I(v_j)} S(k,\ell), & \text{otherwise} \end{cases}$$

Let $S^{(t)}$ denote the version of S right after the t-th iteration. Lizorkin et al. [23] establish the following connection between t and the errors in the SimRank scores in $S^{(t)}$:

LEMMA 1 ([23]). *If $t \geq \log_c(\varepsilon \cdot (1-c)) - 1$, then for any $i, j \in [1, n]$, we have $\left|S^{(t)}(i,j) - s(v_i, v_j)\right| \leq \varepsilon$.* □

Based on Lemma 1 and the fact that each iteration of the power method takes $O(m^2)$ time, we conclude that the power method runs in $O\big(m^2 \log \frac{1}{\varepsilon}\big)$ time when ensuring ε worst-case error. In addition, it requires $O(n^2)$ space (caused by S). These large complexities in time and space make the power method only applicable on small graphs.

3.2 The Monte Carlo Method

The Monte Carlo method [8] is motivated by an alternative definition of SimRank scores [14] that utilizes the concept of *reverse random walks*. Given a node w_0 in G, a reverse random walk from w_0 is a sequence of nodes $W = \langle w_0, w_1, w_2, \ldots \rangle$, such that w_{i+1} ($i \geq 0$) is selected uniformly at random from the in-neighbors of w_i. We refer to w_i as the i-th *step* of W.

Suppose that we have two reverse random walks W_i and W_j that start from two nodes v_i and v_j, respectively, and they *first meet* at the τ-th step. That is, the τ-th steps of W_i and W_j are identical, but for any $\ell \in [0, \tau)$, the ℓ-th step of W_i differs from the ℓ-th step of W_j. Jeh and Widom [14] establishes the following connection between τ and the SimRank score of v_i and v_j:

$$s(v_i, v_j) = \mathbb{E}[c^\tau], \tag{2}$$

where $\mathbb{E}[\cdot]$ denotes the expectation of a random variable.

Based on Equation (2), the Monte Carlo method [8] precomputes a set $\mathcal{W}_i$ of reverse random walks from each node v_i in G, such that (i) each set $\mathcal{W}_i$ has the same number n_w of walks, and (ii) each walk in $\mathcal{W}_i$ is truncated at step t, i.e., the nodes after the t-th step are omitted. (This truncation is necessary to ensure that the walk is computed efficiently.) Then, given two nodes v_i and v_j, the method estimates their SimRank score as

$$\hat{s}(v_i, v_j) = \frac{1}{n_w} \sum_{\ell=0}^{n_w} c^{\tau_\ell},$$

where τ_ℓ denotes the step at which the ℓ-th walk in $\mathcal{W}_i$ first meets with the ℓ-th walk in $\mathcal{W}_j$. Fogaras and Rácz [8] show that, with at least $1 - 2\exp(-\frac{6}{7} n_w \varepsilon^2)$ probability,

$$\left|\hat{s}(v_i, v_j) - \mathbb{E}\left[\hat{s}(v_i, v_j)\right]\right| \leq \varepsilon. \tag{3}$$

However, we note that $\mathbb{E}\left[\hat{s}(v_i, v_j)\right] \neq s(v_i, v_j)$, due to the truncation imposed on the reverse random walks in W_i and W_j. To address this issue, we present the following inequality:

$$\begin{aligned} \left|\mathbb{E}\left[s(v_i, v_j)\right] - \hat{s}(v_i, v_j)\right| &= \left|\mathbb{E}\big[c^\tau\big] - \Pr[\tau \leq t] \cdot \mathbb{E}\big[c^\tau \mid \tau \leq t\big]\right| \\ &= \left|Pr[\tau > t] \cdot \mathbb{E}[c^\tau \mid \tau > t]\right| \\ &\leq c^{t+1} \end{aligned} \tag{4}$$

By Equations (3) and (4) and the union bound, it can be verified that when $t > \log_c \frac{\varepsilon}{2}$ and $n_w \geq \frac{14}{3\varepsilon^2}\left(\log \frac{2}{\delta} + 2 \log n\right)$,

$$\left|\hat{s}(v_i, v_j) - s(v_i, v_j)\right| \leq \varepsilon$$

holds for all pairs of v_i and v_j with at least $1 - \delta$ probability. In that case, the space and preprocessing time complexities of the Monte Carlo method are both $O(n_w \cdot t) = O\big(\frac{n}{\varepsilon^2} \log \frac{1}{\varepsilon} \log \frac{n}{\delta}\big)$. In addition, the method takes $O\big(\frac{1}{\varepsilon^2} \log \frac{1}{\varepsilon} \log \frac{n}{\delta}\big)$ time to answer a single-pair SimRank query, and $O\big(\frac{n}{\varepsilon^2} \log \frac{1}{\varepsilon} \log \frac{n}{\delta}\big)$ time to process a single-source SimRank query. These space and time complexities are rather unfavorable under typical settings of ε in practice (e.g., $\varepsilon = 0.01$). Fogaras and Rácz [8] alleviate this issue with a *coupling technique*, which improves the practical performance of the Monte Carlo method in terms of pre-computation time and space consumption. Nevertheless, the method still incurs significant overheads, due to which it is unable to handle graphs with over one million nodes, as we show in Section 7.

3.3 The Linearization Method

Let S and P be two $n \times n$ matrices, with $S(i, j) = s(v_i, v_j)$ and

$$P(i,j) = \begin{cases} 1/|I(v_j)|, & \text{if } v_i \in I(v_j) \\ 0, & \text{otherwise} \end{cases} \tag{5}$$

Yu et al. [34] show that Equation (1) (i.e., the definition of SimRank) can be rewritten as

$$S = (cP^\top SP) \vee I, \tag{6}$$

where I is an $n \times n$ identity matrix, $P^\top$ is the transpose of P, and $\vee$ is the *element-wise maximum* operator, i.e., $(A \vee B)(i, j) = \max\{A(i, j), B(i, j)\}$ for any two matrices A and B and any i, j.

Maehara et al. [24] point out that solving Equation (6) is difficult since it is a *non-linear* problem due to the $\vee$ operator. To circumvent this difficulty, they prove that there exists a $n \times n$ diagonal matrix D (referred to as the *diagonal correction matrix*), such that

$$S = cP^\top SP + D. \tag{7}$$

Furthermore, once D is given, one can uniquely derive S based on the following lemma by Maehara et al. [24]:

LEMMA 2 ([24]). *Given the diagonal correction matrix D,*

$$S = \sum_{\ell=0}^{+\infty} c^{\ell} \left(P^{\ell}\right)^{\top} D P^{\ell}, \tag{8}$$

where P^{ℓ} denotes the ℓ-th power of P. □

Given Lemma 2, Maehara et al. [24] propose the linearization method, which pre-computes D and then uses it to answer SimRank queries based on Equation (8). In particular, for any two nodes v_i and v_j, Equation (8) leads to

$$s(v_i, v_j) = \sum_{\ell=0}^{+\infty} c^{\ell} \left(P^{\ell} \cdot \vec{e_i}\right)^{\top} D \left(P^{\ell} \cdot \vec{e_j}\right), \tag{9}$$

where $\vec{e_k}$ denotes a n-element column vector where the k-th element equals 1 and all other elements equal 0. To avoid the infinite series in Equation (9), the linearization method approximates $s(v_i, v_j)$ with

$$\tilde{s}(v_i, v_j) = \sum_{\ell=0}^{t} c^{\ell} \left(P^{\ell} \cdot \vec{e_i}\right)^{\top} D \left(P^{\ell} \cdot \vec{e_j}\right), \tag{10}$$

which can be computed in $O(m \cdot t)$ time. It can be shown that if D is precise and $t \geq \log_c(\varepsilon \cdot (1-c)) - 1$, then

$$\left|\tilde{s}(v_i, v_j) - s(v_i, v_j)\right| \leq \varepsilon. \tag{11}$$

Therefore, given an exact D, the linearization method answers any single-pair SimRank query in $O(m \log \frac{1}{\varepsilon})$ time. With a slight modification of Equation 10, the method can also process any single-source SimRank query in $O(m \log^2 \frac{1}{\varepsilon})$ time.

Unfortunately, the linearization method do not precisely derive D, due to which the above time complexities does not hold in general. Specifically, Maehara et al. [24] formulate D as the solution to a linear system, and propose to solve an *approximate* version of the system to derive an estimation $\widetilde{D}$ of D. However, there is no formal analysis on the errors in $\widetilde{D}$ and their effects on the accuracy of SimRank computation. In addition, the technique used to solve the approximate linear system does not guarantee to *converge*, i.e., it may not return $\widetilde{D}$ in bounded time. Furthermore, even if the technique does converge, its time complexity relies on a parameter that is unknown in advance, and may even dominate n, m, and $1/\varepsilon$. This makes it rather difficult to analyze the pre-computation time of the linearization method. We refer interested readers to Appendix A for detailed discussions on these issues.

In summary, the linearization method by Maehara et al. [24] does not guarantee ε worst-case error in each SimRank score returned, and there is no non-trivial bound on its preprocessing time. This problem is partially addressed in recent work [33] by Yu and McCann, who propose a variant of the linearization method that does not pre-compute the diagonal correction matrix D, but implicitly derives D during query processing. Yu and McCann's technique is able to ensure ε worst-case error in SimRank computation, but as a trade-off, it requires $O\big(mn \log \frac{1}{\varepsilon}\big)$ time to answer a single-pair SimRank query, which renders it inapplicable on any sizable graph.

4. OUR SOLUTION

This section presents our *SLING* index for SimRank queries. *SLING* is based on a new interpretation of SimRank scores, which we clarify in Section 4.1. After that, Sections 4.3-4.5 provide details of *SLING* and analyze its theoretical guarantees.

4.1 New Interpretation of SimRank

Let c be the decay factor in the definition of SimRank (see Equation (1)). Suppose that we perform a reverse random walk from any node u in G, such that

- At each step of the walk, we stop with $1 - \sqrt{c}$ probability;
- With the other $\sqrt{c}$ probability, we inspect the in-neighbors of the node at the current step, and select one of them uniformly at random as the next step.

We refer to such a reverse random walk as a *$\sqrt{c}$-walk* from u. In addition, we say that two $\sqrt{c}$-walks *meet*, if for a certain $\ell \geq 0$, the ℓ-th steps of the two walks are identical. (Note the 0-th step of a $\sqrt{c}$-walk is its starting node.) The following lemma shows an interesting connection between $\sqrt{c}$-walks and SimRank.

LEMMA 3. *Let W_i and W_j be two $\sqrt{c}$-walks from two nodes v_i and v_j, respectively. Then, $s(v_i, v_j)$ equals the probability that W_i and W_j meet.* □

The above formulation of SimRank is similar in spirit to the one used in the Monte Carlo method [8] (see Section 3.2), but differs in one crucial aspect: each $\sqrt{c}$-walk in our formulation has an expected length of $\frac{1}{1-\sqrt{c}}$, whereas each reverse random walk in the previous formulation is infinite. As a consequence, if we are to estimate $s(v_i, v_j)$ using a sample set of $\sqrt{c}$-walks from v_i and v_j, we do not need to truncate any $\sqrt{c}$-walk for efficiency; in contrast, the Monte Carlo method [8] must trim each reverse random walk to trade estimation accuracy for bounded computation time. In fact, if we incorporate $\sqrt{c}$-walks into the Monte Carlo method, then its query time complexities are immediately improved by a factor of $\log \frac{1}{\varepsilon}$. Nonetheless, the space and time overheads of this revised method still leave much room for improvement, since it requires $O(\log \frac{n}{\delta}/\varepsilon^2)$ $\sqrt{c}$-walks for each node, where δ is the upper bound on the method's failure probability. This motivates us to develop the *SLING* method for more efficient SimRank computation, which we elaborate in the following sections.

4.2 Key Idea of *SLING*

Let $h^{(\ell)}(v_a, v_b)$ denote the probability that a $\sqrt{c}$-walk from v_a arrives at v_b in its ℓ-th step. We refer to $h^{(\ell)}(v_a, v_b)$ as the *hitting probability (**HP**)* from v_a to v_b at step ℓ. Observe that, for any two $\sqrt{c}$-walks W_i and W_j from two nodes v_i and v_j, respectively, the probability that they meet at v_k at the ℓ-th step is

$$h^{(\ell)}(v_i, v_k) \cdot h^{(\ell)}(v_j, v_k).$$

Since $s(v_i, v_j)$ equals the probability that W_i and W_j meet, one may attempt to compute $s(v_i, v_j)$ by taking the the probability that W_i and W_j meet over all combinations of meeting nodes and meeting steps, i.e.,

$$s^*(v_i, v_j) = \sum_{\ell=0}^{+\infty} \sum_{k=1}^{n} \left(h^{(\ell)}(v_i, v_k) \cdot h^{(\ell)}(v_j, v_k)\right). \tag{12}$$

However, this formulation is incorrect, because the events that "W_i and W_j meet at node v_x at step ℓ" and "W_i and W_j meet at node v_y at step $\ell' > \ell$" are *not* mutually exclusive. For example, assume that $v_i = v_j$, and v_i has only in-neighbor v_k. In that case, W_i and W_j have 100% probability to meet at v_i at the 0-th step, and a non-zero probability to meet at v_k at the first step. This leads to $s^*(v_i, v_j) > 1$, whereas $s(v_i, v_j) = 1$ by definition.

Interestingly, Equation (12) can be fixed if we substitute $h^{(\ell)}(v_i, v_k) \cdot h^{(\ell)}(v_j, v_k)$ with the probability of the event that "W_i and W_j meet at v_k at step ℓ, *but never meet again afterwards*".

To explain this, observe that the above event indicates that W_i and W_j *last meet* at v_k at step ℓ. If we change v_k (resp. ℓ) in the event, then W_i and W_j should last meet at a different node (resp. step), in which case the changed event and the original one are mutually exclusive. Based on this observation, the following lemma presents a remedy to Equaiton (12).

LEMMA 4. *Let d_k be the probability that two $\sqrt{c}$-walks from node v_k do not meet each other after the 0-th step. Then, for any two nodes v_i and v_j,*

$$s(v_i, v_j) = \sum_{\ell=0}^{\infty} \sum_{k=1}^{n} \left(h^{(\ell)}(v_i, v_k) \cdot d_k \cdot h^{(\ell)}(v_j, v_k) \right). \quad (13)$$

In what follows, we refer to d_k as the *correction factor* for v_k.

Based on Lemma 4, we propose to pre-compute approximate versions of d_k and HPs $h^{(\ell)}(v_i, v_k)$, and then use them to estimate SimRank scores based on Equation (4). The immediate problem here is that there exists an infinite number of HPs $h^{(\ell)}(v_i, v_k)$ to approximate, since we need to consider all $\ell \geq 0$. However, we observe that if we allow an additive error in the approximate values, then most of the HPs can be estimated as zero and be omitted. In particular, we have the following observation:

OBSERVATION 1. *For any node v_i and $\ell \geq 0$, there exist at most $(\sqrt{c})^{\ell}/\varepsilon_h$ nodes v_k such that $h^{(\ell)}(v_j, v_k) \geq \varepsilon_h$.* □

To understand this, recall that each $\sqrt{c}$-walk has only $(\sqrt{c})^{\ell}$ probability to *not* stop before the ℓ-th step, i.e.,

$$\sum_{k=1}^{n} h^{(\ell)}(v_j, v_k) = (\sqrt{c})^{\ell}.$$

Therefore, at most $(\sqrt{c})^{\ell}/\varepsilon_h$ of the HPs at step ℓ can be larger than ε_h. Even if we take into account all $\ell \geq 0$, the total number of HPs above ε_h is only

$$\sum_{\ell=0}^{+\infty} (\sqrt{c})^{\ell}/\varepsilon_h = O(1/\varepsilon_h).$$

In other words, we only need to retain a constant number of HPs for each node, if we permit a constant additive error in each HP.

Based on the above analysis, we propose the *SLING* index, which pre-computes an approximate version $\tilde{d}_k$ of each correction factor d_k, as well as a constant-size set $H(v_i)$ of approximate HPs for each node v_i. To derive the SimRank score of two nodes v_i and v_j, *SLING* first retrieves $\tilde{d}_k$, $H(v_i)$, and $H(v_j)$, and then estimates $s(v_i, v_j)$ in constant time based on an approximate version of Equation (13). The challenge in the design of *SLING* is threefold. First, how can we derive an accurate estimation of $\tilde{d}_k$? Second, how can we efficiently construct $H(v_i)$ without iterating over all HPs? Third, how do we ensure that all $\tilde{d}_k$ and $H(v_i)$ can jointly guarantee ε worst-case error in each SimRank score computed? In Sections 4.3-4.5, we elaborate how we address these challenges.

Before we proceed, we note that there is an interesting connection between Lemmas 2 and 4:

LEMMA 5. *Let P and D be as in Lemma 2, and d_k and $h^{(\ell)}(v_i, v_k)$ be as in Lemma 4. For any $i, k \in [1, n]$, $h^{(\ell)}(v_i, v_k) = (\sqrt{c})^{\ell} \cdot P(k, i)$, and d_k equals the k-th diagonal element in D.* □

In other words, $h^{(\ell)}(v_i, v_k)$ (resp. d_k) can be regarded as a random-walk-based interpretation of the entries in P (resp. diagonal elements in D). Therefore, Lemmas 2 and 4 are different interpretations of the same result. The main advantage of our new interpretation is that it gives a physical meaning to d_k which, as we show in Section 4.3, enables us to devise a simple and rigorous algorithm to estimate d_k to any desired precision. In contrast, the only existing method for approximating D [24] fails to provide any non-trivial guarantees in terms of accuracy and efficiency, as we discuss in Section 3.3.

Algorithm 1: *A sampling method for estimating d_k*

Input: a node v_k, an error bound ε_d, and a failure probability δ_d
Output: an estimation version $\tilde{d}_k$ of d_k with at most ε_d error, with at least $1 - \delta_d$ probability

```
Let n_r = (2c^2 + c·ε_d)/ε_d^2 · log(2/δ_d);
Let cnt = 0;
for x = 1, 2, ··· , n_r do
    Select two nodes v_i and v_j from I(v_k) uniformly at random;
    if v_i ≠ v_j then
        Generate two √c-walks from v_i and v_j, respectively;
        if the two √c-walks meet then
            cnt = cnt + 1;
return d̃_k = 1 − c/|I(v_i)| − c · cnt/n_r;
```

4.3 Estimation of d_k

Let W and W' be two $\sqrt{c}$-walks from v_k. By definition, $1 - d_k$ is the probability that any of the following events occurs:

1. W and W' meet at the first step.
2. In the first step, W and W' arrive at two different nodes v_i and v_j, respectively; but sometime after the first step, W and W' meet.

Note that the above two events are mutually exclusive, and the first event occurs with $\frac{c}{|I(v_k)|}$ probability. For the second event, if we fix a pair of v_i and v_j, then the probability that W and W' meet after the first step equals the probability that a $\sqrt{c}$-walk from v_i meets a $\sqrt{c}$-walk from v_j; by Lemma 3, this probability is exactly $s(v_i, v_j)$. Therefore, we have

$$d_k = 1 - \frac{c}{|I(v_k)|} - \frac{c}{|I(v_k)|^2} \sum_{\substack{v_i, v_j \in I(v_k) \\ v_i \neq v_j}} s(v_i, v_j). \quad (14)$$

Equation (14) indicates that, if we are to estimate d_k, it suffices to derive an estimation of

$$\mu = \frac{1}{|I(v_k)|^2} \sum_{v_i, v_j \in I(v_k) \wedge v_i \neq v_j} s(v_i, v_j) \quad (15)$$

by sampling $\sqrt{c}$-walks from v_i and v_j. In particular, as long as μ is estimated with an error no more than ε_d/c, the resulting estimation of d_k would have at most ε_d error. Motivated by this, we propose a sampling method for approximating d_k, as shown in Algorithm 1.

In a nutshell, Algorithm 1 generates n_r pairs of $\sqrt{c}$-walks, such that each walk starts from a randomly selected node in $I(v_k)$; after that, the algorithm counts the number cnt of pairs that meet at or after the first step; finally, it returns $\tilde{d}_k = 1 - \frac{c}{|I(v_i)|} - c \cdot \frac{cnt}{n_r}$ as an estimation of d_k. By the Chernoff bound (see Appendix D) and the properties of $\sqrt{c}$-walks, we have the following lemma on the theoretical guarantees of Algorithm 1.

LEMMA 6. *Algorithm 1 runs in $O\left(\frac{1}{\varepsilon_d^2} \log \frac{1}{\delta_d}\right)$ expected time, and returns $\tilde{d}_k$ such that $|\tilde{d}_k - d| \leq \varepsilon_d$ holds with at least $1 - \delta_d$ probability.* □

Algorithm 2: *A local update method for constructing* $H(v_i)$

Input: G and a threshold θ
Output: A set $H(v_i)$ of approximate HPs for each node v_i in G

1 Initialize $H(v_i) = \emptyset$ for each node v_i;
2 **for** *each node* v_k *in* G **do**
3 Initialize a set $R_k = \emptyset$ for storing approximate HPs;
4 Insert $\tilde{h}^{(0)}(v_k, v_k) = 1$ into R_k;
5 **for** $\ell = 0, 1, 2, \ldots$ **do**
6 **for** *each* $\tilde{h}^{(\ell)}(v_x, v_k) \in R_k$ **do**
7 **if** $\tilde{h}^{(\ell)}(v_x, v_k) \leq \theta$ **then**
8 remove $\tilde{h}^{(\ell)}(v_x, v_k)$ from R_k;
9 **continue**;
10 **for** *each out-neighbor* v_i *of* v_x **do**
11 **if** $\tilde{h}^{(\ell)}(v_i, v_k) \notin R_k$ **then**
12 Insert $\tilde{h}^{(\ell+1)}(v_i, v_k) = \sqrt{c} \cdot \frac{\tilde{h}^{(\ell)}(v_x, v_k)}{|I(v_i)|}$ into R_k;
13 **else**
14 Increase $\tilde{h}^{(\ell+1)}(v_i, v_k)$ by $\sqrt{c} \cdot \frac{\tilde{h}^{(\ell)}(v_x, v_k)}{|I(v_i)|}$;
15 **if** R_k *does not contain any HP at step* $\ell + 1$ **then**
16 **break**;
17 **for** *each* $\tilde{h}^{(\ell)}(v_i, v_k) \in R_k$ **do**
18 Insert $\tilde{h}^{(\ell)}(v_i, v_k)$ into $H(v_i)$;

4.4 Construction of $H(v_i)$

As mentioned in Section 4.2, we aim to construct a constant-size set $H(v_i)$ for each node v_i, such that $H(v_i)$ contains an approximate version $\tilde{h}^{(\ell)}(v_i, v_x)$ of each HP $h^{(\ell)}(v_i, v_x)$ that is sufficiently large. Towards this end, a relatively straightforward solution is to sample a set $\mathcal{W}_i$ of $\sqrt{c}$-walks from each v_i, and then use $\mathcal{W}_i$ to derive approximate HPs. This solution, however, requires $O(1/{\varepsilon_h}^2)$ walks in $\mathcal{W}_i$ to ensure that the additive error in each $\tilde{h}^{(\ell)}(v_i, v_x)$ is at most ε_h, which leads to considerable computation costs when ε_h is small.

Instead of sampling $\sqrt{c}$-walks, we devise a deterministic method for constructing all $H(v_i)$ in $O(m/\varepsilon_h)$ time while allowing at most ε_h additive error in each approximate HP. The key idea of our method is to utilize the following equation on HPs:

$$h^{(\ell+1)}(v_i, v_k) = \frac{\sqrt{c}}{|I(v_i)|} \sum_{v_x \in I(v_i)} h^{(\ell)}(v_x, v_k), \qquad (16)$$

for any $\ell \geq 0$. Intuitively, Equation (16) indicates that once we have derived the HPs to v_k at step ℓ, then we can compute the HPs to v_k at step $\ell + 1$. Based on this intuition, our method generates approximate HPs to v_k by processing the steps ℓ in ascending order of ℓ. We note that our method is similar in spirit to the *local update* algorithm [4,9,15] for estimating *personalized PageRanks* [15], and we refer interested readers to Appendix B for a discussion on the connections between our method and those in [4, 9, 15].

Algorithm 2 shows the pseudo-code of our method. Given G and a threshold θ, the algorithm first initializes $H(v_i) = \emptyset$ for each node v_i (Line 1). After that, for each node v_k, the algorithm performs a graph traversal from v_k to generates approximate HPs from other nodes to v_k. Specifically, for each v_k, it first initializes a set $R_k = \emptyset$, and then inserts an HP $\tilde{h}^{(0)}(v_k, v_k) = 1$ into R_k, which captures the fact that every $\sqrt{c}$-walk from v_k has 100% probability to hit v_k itself at the 0-th step (Lines 3-4). Then, the algorithm enters an iterative process, such that the ℓ-th iteration ($\ell \geq 0$) processes the HPs to v_k at step ℓ that have been inserted into R_k.

Algorithm 3: *An algorithm for single-pair SimRank queries*

Input: $\tilde{d}_k$, $H(v_k)$, and two nodes v_i and v_j
Output: An approximate SimRank score $\tilde{s}(v_i, v_j)$

1 Let $\tilde{s}(v_i, v_j) = 0$;
2 **for** *each* $\tilde{h}^{(\ell)}(v_i, v_k) \in H(v_i)$ **do**
3 **if** *there exists* $\tilde{h}^{(\ell)}(v_j, v_k) \in H(v_j)$ **then**
4 $\tilde{s}(v_i, v_j) = \tilde{s}(v_i, v_j) + \tilde{h}^{(\ell)}(v_i, v_k) \cdot \tilde{d}_k \cdot \tilde{h}^{(\ell)}(v_j, v_k)$;
5 **return** $\tilde{s}(v_i, v_j)$;

In particular, in the ℓ-the iteration, the algorithm first identifies the approximate HPs $\tilde{h}^{(\ell)}(v_x, v_k)$ in R_k that are at step ℓ, and processes each of them in turn (Lines 6-16). If $\tilde{h}^{(\ell)}(v_x, v_k) \leq \theta$, then it is removed from R_k, i.e., the algorithm omits an approximate HP if it is sufficiently small. Meanwhile, if $\tilde{h}^{(\ell)}(v_x, v_k) > \theta$, then the algorithm inspects each out-neighbor v_i of v_x, and updates the approximate HP from v_i to v_k at step $\ell + 1$, according to Equation (16). After all approximate HPs at step ℓ are processed, the algorithm terminates the iterative process on ℓ. Finally, the algorithm inserts each $\tilde{h}^{(\ell)}(v_i, v_k) \in R$ into $H(v_i)$, after which it proceeds to the next node v_{k+1}.

The following lemma states the guarantees of Algorithm 2.

LEMMA 7. *Algorithm 2 runs in $O(m/\theta)$ time, and constructs a set $H(v_i)$ of approximate HPs for each node v_i, such that $|H(v_i)| = O(1/\theta)$. In addition, for each $\tilde{h}^{(\ell)}(v_i, v_k) \in H(v_i)$, we have*

$$0 \geq \tilde{h}^{(\ell)}(v_i, v_k) - h^{(\ell)}(v_i, v_k) \geq -\frac{1 - (\sqrt{c})^\ell}{1 - \sqrt{c}} \cdot \theta.$$

4.5 Query Method and Complexity Analysis

Given an approximate correction factor $\tilde{d}_k$ and a set $H(v_k)$ of approximate HPs for each node v_k, we estimate the SimRank score between any two nodes v_i and v_j according to a revised version of Equation (13):

$$\tilde{s}(v_i, v_j) = \sum_{\ell=0}^{\infty} \sum_{k=1}^{n} \left(\tilde{h}^{(\ell)}(v_i, v_k) \cdot \tilde{d}_k \cdot \tilde{h}^{(\ell)}(v_j, v_k) \right). \qquad (17)$$

Algorithm 3 shows the details of our query processing method.

To analyze the accuracy guarantee of Algorithm 3, we first present a lemma that quantifies the error in $\tilde{s}(v_i, v_j)$ based on the errors in $\tilde{d}_k$ and $H(v_k)$.

LEMMA 8. *Suppose that $\left|\tilde{d}_k - d_k\right| \leq \varepsilon_d$ for any k, and*

$$0 \geq \tilde{h}^{(\ell)}(v_k, v_x) - h^{(\ell)}(v_k, v_x) \geq -\varepsilon_h^{(\ell)},$$

for any k, x, ℓ. Then, we have $|\tilde{s}(v_i, v_j) - s(v_i, v_j)| \leq \varepsilon$ if

$$\frac{\varepsilon_d}{1 - c} + 2 \sum_{\ell=0}^{+\infty} \left((\sqrt{c})^\ell \cdot \varepsilon_h^{(\ell)} \right) \leq \varepsilon.$$

Combining Lemmas 6, 7, and 8, we have the following theorem.

THEOREM 1. *Suppose that we derive each $\tilde{d}_k$ using Algorithm 1 with input ε_d and δ_d, and we construct each $H(v_k)$ using Algorithm 2 with input θ. If $\delta_d \leq \delta/n$ and*

$$\frac{\varepsilon_d}{1 - c} + \frac{2\sqrt{c}}{(1 - \sqrt{c})(1 - c)} \theta \leq \varepsilon,$$

then Algorithm 3 incurs an additive error at most ε in each SimRank score returned, with at least $1 - \delta$ probability. □

By Theorem 1, we can ensure ε worst-case error in each SimRank score by setting $\varepsilon_d = O(\varepsilon)$, $\theta = O(\varepsilon)$, and $\delta_d = \delta/n$. In that case, our *SLING* index requires $O(m/\varepsilon + n \log \frac{n}{\delta})$ precomputation time and $O(n/\varepsilon)$ space, and it answers any single-pair SimRank query in $O(1/\varepsilon)$ time. The space (resp. query time) complexity of *SLING* is only $O(1/\varepsilon)$ times larger than the optimal value, since any SimRank method (that ensures ε worst-case error) requires $\Omega(n)$ space for storing the information about all nodes, and takes at least $\Omega(1)$ time to output the result of a single-pair SimRank query.

5. OPTIMIZATIONS

This section presents optimization techniques to (i) improve the efficiency of estimating each correction factors d_k (Section 5.1), (ii) reduce the space consumption of *SLING* (Section 5.2), (iii) enhance the accuracy of *SLING* (Section 5.3), and (iv) incorporate parallel and out-of-core computation into *SLING*'s index construction algorithm (Section 5.4).

5.1 Improved Estimation of d_k

As discussed in Section 4.3, Algorithm 1 generates an approximate correction factor $\tilde{d}_k$ in $O\left(\varepsilon_d^{-2} \log \delta_d^{-1}\right)$ expected time, where ε_d is the maximum error allowed in $\tilde{d}_k$, and δ_d is the failure probability. As the algorithm's time complexity is quadratic to $1/\varepsilon_d$, it is not particularly efficient when ε_d is small. This relative inefficiency is caused by the fact the algorithm requires $O\left(\varepsilon_d^{-2} \log \delta_d^{-1}\right)$ pairs of $\sqrt{c}$-walks to estimate the value μ (in Equation 15) with ε_d/c worst-case error.

However, we observe that we can often use a much smaller number of $\sqrt{c}$-walk pairs to derive an estimation of μ with at most ε_d/c error. Specifically, by the Chernoff bound (see Appendix D), we only need $O\left((\mu + \varepsilon_d) \cdot \varepsilon_d^{-2} \log \delta_d^{-1}\right)$ pairs of $\sqrt{c}$-walks to estimate μ. Apparently, this number is much smaller than $O\left(\varepsilon_d^{-2} \log \delta_d^{-1}\right)$ when $\mu \ll 1$ (which is often the case in practice). For example, if $\mu \leq \varepsilon_d$, then the number of $\sqrt{c}$-walk pairs required is only $O(\varepsilon_d^{-1} \log \delta_d^{-1})$. The main issue here is that we do not know μ in advance. Nevertheless, if we can derive an upper bound of μ, and we use it to decide an appropriate number of $\sqrt{c}$-walks needed.

Based on the above observation, we propose an improved algorithm for computing $\tilde{d}_k$, as shown in Algorithm 4. The algorithm first generates $n_r = O(\varepsilon_d^{-1} \log \delta_d^{-1})$ pairs of $\sqrt{c}$-walks from randomly selected nodes in $I(v_k)$, and counts the number cnt of pairs that meet (Lines 1-8). Then, it computes $\hat{\mu} = cnt/n_r$ as an estimation of μ. If $\hat{\mu} \leq \varepsilon_d$, then the algorithm determines that n_r pairs of $\sqrt{c}$-walks are sufficient for an accurate estimation of μ; in that case, it terminates and returns an estimation of d_k based on $\hat{\mu}$ (Lines 9-11).

On the other hand, if $\hat{\mu} > \varepsilon_d$, then the algorithm proceeds to generate a larger number of $\sqrt{c}$-walks to derive a more accurate estimation of μ. Towards this end, it first computes $\mu^* = \hat{\mu} + \sqrt{\hat{\mu} \cdot \varepsilon}$ as an upper bound of μ, and uses μ^* to decide the total number $n_r^* = O(\mu^* \varepsilon_d^{-2} \log \delta_d^{-1})$ of $\sqrt{c}$-walk pairs that are needed (Lines 12-13). After that, it increases the total number of $\sqrt{c}$-walk pairs to n_r^*, and recounts the number cnt of pairs that meet (Lines 14-19). Finally, it derives $\tilde{u} = cnt/n_r^*$ as an improved estimation of μ, and returns an approximate correction factor $\tilde{d}_k$ computed based on $\tilde{\mu}$ (Lines 20-21).

The following lemmas establish the asymptotic guarantees of Algorithm 4.

LEMMA 9. *With at least $1-\delta_d$ probability, Algorithm 4 returns $\tilde{d}_k$ such that $|\tilde{d}_k - d| \leq \varepsilon_d$ holds.* □

Algorithm 4: *An improved method for estimating d_k*

Input: a node v_k, an error bound ε_d, and a failure probability δ_d
Output: an estimation version $\tilde{d}_k$ of d_k with at most ε_d error, with at least $1 - \delta_d$ probability

1 Let $n_r = \frac{14c}{3\varepsilon_d} \log \frac{4}{\delta_d}$;
2 Let $cnt = 0$;
3 **for** $x = 1, 2, \cdots, n_r$ **do**
4 Select two nodes v_i and v_j from $I(v_k)$ uniformly at random;
5 **if** $v_i \neq v_j$ **then**
6 Generate two $\sqrt{c}$-walks from v_i and v_j, respectively;
7 **if** *the two $\sqrt{c}$-walks meet* **then**
8 $cnt = cnt + 1$;
9 Let $\hat{\mu} = cnt/n_r$;
10 **if** $\hat{\mu} \leq \varepsilon_d$ **then**
11 **return** $\tilde{d}_k = 1 - \frac{c}{|I(v_i)|} - c \cdot \hat{\mu}$;
12 Let $\mu^* = \hat{\mu} + \sqrt{\hat{\mu} \cdot \varepsilon_d}$;
13 Let $n_r^* = \frac{2c^2 \cdot \mu^* + \frac{2}{3}c \cdot \varepsilon_d}{\varepsilon_d^2} \log \frac{4}{\delta_d}$;
14 **for** $x = 1, 2, \cdots, n_r^* - n_r$ **do**
15 Select two nodes v_i and v_j from $I(v_k)$ uniformly at random;
16 **if** $v_i \neq v_j$ **then**
17 Generate two $\sqrt{c}$-walks from v_i and v_j, respectively;
18 **if** *the two $\sqrt{c}$-walks meet* **then**
19 $cnt = cnt + 1$;
20 $\tilde{\mu} = cnt/n_r^*$;
21 **return** $\tilde{d}_k = 1 - \frac{c}{|I(v_i)|} - c \cdot \tilde{\mu}$;

LEMMA 10. *Algorithm 4 generates $O(\frac{\mu+\varepsilon_d}{\varepsilon_d^2} \log \frac{1}{\delta_d})$ $\sqrt{c}$-walks in expectation, and runs in $O(\frac{\mu+\varepsilon_d}{\varepsilon_d^2} \log \frac{1}{\delta_d})$ expected time.* □

By Lemma 9, Algorithm 4 uses a number of $\sqrt{c}$-walks that is roughly $\max\{\mu, \varepsilon_d\}$ times the number in Algorithm 1, which leads to significantly improved efficiency. In addition, we note that Algorithm 4 can be easily revised into a general method that estimates the expectation μ_z of a Bernoulli distribution by taking $O(\frac{\mu_z+\varepsilon}{\varepsilon^2} \log \frac{1}{\delta})$ samples, while ensuring at most ε estimation error with at least $1 - \delta$ success probability. In particular, the only major change needed is to replace each $\sqrt{c}$-walk pair in Algorithm 4 with a sample from the Bernoulli distribution. In this context, we can prove that the number of samples used by Algorithm 4 is *asymptotically optimal*.

Specifically, let $z_1, z_2, \ldots$ be a sequence of i.i.d. Bernoulli random variables, and $\mu_Z = \mathbb{E}[z_i]$. Let $\mathcal{A}$ be an algorithm that inspects z_i in ascending order of i, and stops at a certain z_j before returning an estimation $\tilde{\mu_Z}$ of μ_Z. In addition, for any possible sequence of z_i, $\mathcal{A}$ runs in finite expected time, and ensures that $|\tilde{\mu_Z} - \mu_Z| \leq \varepsilon$ with at least $1 - \delta$ probability. It can be verified that the revised Algorithm 4 is an instance of $\mathcal{A}$. The following lemma shows that no other instance of $\mathcal{A}$ can be asymptotically more efficient than Algorithm 4.

LEMMA 11. *Any instance of $\mathcal{A}$ has $\Omega(\frac{\max\{\mu_z,\varepsilon\}}{\varepsilon^2} \log \frac{1}{\delta})$ expected time complexity when $\mu_z < 0.5$.* □

Our proof of Lemma 11 utilizes an important result by Dagum et al. [7] that establishes a lower bound of the expected time complexity of $\mathcal{A}$, when it provides a worst-case guarantee in terms of the relative error (instead of absolute error) in $\tilde{\mu_z}$. Dagum et al. [7] also provide a sampling algorithm whose time complexity matches

Algorithm 5: *An algorithm for constructing $H'(v_i)$*

Input: a node v_i
Output: A set $H'(v_i)$ of precise HPs from v_i at steps 1 and 2

```
Initialize H'(v_i) = ∅;
Insert h^(0)(v_i, v_i) = 1 into H'(v);
for each node v_x ∈ I(v_i) do
    Insert h^(1)(v_i, v_x) = c / |I(v_i)| into H'(v);
    for each node v_y ∈ I(v_x) do
        if h^(2)(v_i, v_y) ∉ H'(v_i) then
            Insert h^(2)(v_i, v_y) = √c · h^(1)(v_i, v_x) / |I(v_x)| into H'(v_i);
        else
            Increase h^(2)(v_i, v_y) by √c · h^(1)(v_i, v_x) / |I(v_x)| in H'(v_i);
return H'(v_i)
```

their lower bound, but the algorithm is inapplicable in our context, since it requires as input a relative error bound, which cannot be translated into an absolute error bound unless μ_z is known.

5.2 Reduction of Space Consumption

Recall that our *SLING* index pre-computes a set $H(v_i)$ of approximate HPs for each node v_i, such that each $\tilde{h}^{(\ell)}(v_i, v_k) \in H(v_i)$ is no smaller than a threshold $\theta = O(\varepsilon)$. The total size of all $H(v_i)$ is $O(n/\varepsilon)$, which is asymptotically near-optimal, but may still be costly from a practical perspective (especially when ε is small). To address this issue, we aim to reduce the size of $H(v_i)$ without affecting the time complexity of *SLING*.

We observe that, in each $H(v_i)$, a significant portion of the approximate HPs are in the form of $\tilde{h}^{(1)}(v_i, v_k)$ or $\tilde{h}^{(2)}(v_i, v_k)$, i.e., they concern the HPs from v_i to the nodes within two hops away from v_i. On the other hand, such HPs can be easily computed using a two-hop traversal from v_i, as we will show shortly. This leads to the following idea for space reduction: we remove from $H(v_i)$ all approximate HPs that are at steps 1 and 2, and we recompute those HPs *on the fly* during query processing. The re-computation may lead to slightly increased query cost, but as long as it takes $O(1/\varepsilon)$ time, it would not affect the asymptotic performance of *SLING*. In the following, we clarify how we implement this idea.

First, we present a simple and *precise* algorithm for computing the set $H'(v_i)$ of HPs from node v_i to other nodes at steps 1 and 2, as shown in Algorithm 5. The algorithm first initializes a set $H'(v_i) = \emptyset$ for storing HPs, and then inserts $h^{(0)}(v_i, v_i) = 1$ into $H'(v)$. After that, for each in-neighbor v_x of v_i, it sets $h^{(1)}(v_i, v_x) = \frac{\sqrt{c}}{|I(v_i)|}$, which is the exact probability that a $\sqrt{c}$-walk from v_i would hit v_x at step 1. In turn, for each in-neighbor v_y of v_x, the algorithm initializes $h^{(2)}(v_i, v_y) = \sqrt{c} \cdot \frac{h^{(1)}(v_i, v_x)}{|I(v_x)|}$ in $H'(v_i)$, if it is not yet inserted into $H'(v_i)$; otherwise, the algorithm increases $h^{(2)}(v_i, v_y)$ by $\sqrt{c} \cdot \frac{h^{(1)}(v_i, v_x)}{|I(v_x)|}$ in $H'(v_i)$. This reason is that if a $\sqrt{c}$-walk from v_i hits v_x at step 1, then it has $\frac{\sqrt{c}}{|I(v_x)|}$ probability to hit v_y at step 2. After all of v_i's in-neighbors are processed, the algorithm terminates and returns $H'(v_i)$.

Algorithm 5 runs in time linear to the total number $\eta(v_i)$ of incoming edges of v_i and its in-neighbors, i.e.,

$$\eta(v_i) = |I(v_i)| + \sum_{v_x \in I(v_i)} |I(v_x)|.$$

If $\eta(v_i) = O(1/\varepsilon)$, then we can omit all step-1 and step-2 approximate HPs in $H(v_i)$, and compute them with Algorithm 5 during query processing without degrading the time complexity of *SLING*; otherwise, we need to retain all approximate HPs in $H(v_i)$. In our implementation of *SLING*, we set a constant $\gamma = 10$, and we exclude step-1 and step-2 HPs from $H(v_i)$ whenever $\eta(v_i) \leq \gamma/\theta$, where $\theta = \Omega(\varepsilon)$ is the HP threshold used in the construction of $H(v_i)$ (see Algorithm 2). Notice that each $\eta(v_i)$ can be computed in $O(|I(v_i)|)$ time by inspecting v_i and all of its in-neighbors; therefore, the total computation cost of all $\eta(v_i)$ is $O(m)$, which does not affect *SLING*'s preprocessing time complexity. Furthermore, the on-the-fly computation of step-1 and step-2 HPs does not degrade *SLING*'s accuracy guarantee, since all HPs returned by Algorithm 5 are precise.

5.3 Enhancement of Accuracy

The approximation error of each $H(v_i)$ arises from the fact that it omits the HPs from v_i that are smaller than a threshold θ. A straightforward solution to reduce this error is to decrease θ, but it would degrade the space overhead of $H(v_i)$. Instead, we propose to generate *additional* HPs in $H(v_i)$ *on-the-fly* during query processing, to increase the accuracy of query results.

Specifically, for each node v_i, after $H(v_i)$ is constructed (with the space reduction procedure in Section 5.2 applied), we inspect the set of approximate HPs $\tilde{h}^{(\ell)}(v_i, v_j)$ in $H(v_i)$ such that v_j has no more than $1/\sqrt{\varepsilon}$ in-neighbors, and then *mark* the $1/\sqrt{\varepsilon}$ largest HPs in the set. After that, whenever a SimRank query requires utilizing $H(v_i)$, we substitute $H(v_i)$ with an enhanced version $H^*(v_i)$ constructed on-the-fly. In particular, we first set $H^*(v_i) = H(v_i)$. Then, for every marked HP $\tilde{h}^{(\ell)}(v_i, v_j)$ in $H(v_i)$, we process each in-neighbor v_k of v_j as follows:

- If there exists $\tilde{h}^{(\ell+1)}(v_i, v_k)$ in $H(v_i)$, then we omit v_k;
- If $\tilde{h}^{(\ell+1)}(v_i, v_k)$ is not in $H(v_i)$ and has not been inserted into $H^*(v_i)$, then we set $\tilde{h}^{(\ell+1)}(v_i, v_k) = \frac{\sqrt{c}}{|I(v_j)|} h^{(\ell)}(v_i, v_j)$, and insert it into $H^*(v_i)$;
- Otherwise, we update $\tilde{h}^{(\ell+1)}(v_i, v_k)$ in $H^*(v_i)$ as follows: $\tilde{h}^{(\ell+1)}(v_i, v_k) = \tilde{h}^{(\ell+1)}(v_i, v_k) + \frac{\sqrt{c}}{|I(v_j)|} h^{(\ell)}(v_i, v_j)$.

In other words, if $H(v_i)$ does not contain an approximate HP from v_i to v_k, then we generate $\tilde{h}^{(\ell+1)}(v_i, v_k)$ in $H^*(v_i)$.

It can be verified that $0 < \tilde{h}^{(\ell+1)}(v_i, v_k) \leq h^{(\ell+1)}(v_i, v_k)$, and hence, $H^*(v_i)$ provides higher accuracy than $H(v_i)$. In addition, the construction of $H^*(v_i)$ requires only $O(1/\varepsilon)$ time, and hence, it does not affect the $O(1/\varepsilon)$ query time complexity of *SLING*. Furthermore, marking HPs in all $H(v_i)$ requires only $O(n/\sqrt{\varepsilon})$ space and $O(n \log(1/\varepsilon)/\varepsilon)$ preprocessing time, which does not degrade the $O(n/\varepsilon)$ space and $O\big(m/\varepsilon + n \log \frac{n}{\delta}/\varepsilon^2\big)$ preprocessing time complexity of *SLING*.

5.4 Parallel and Out-of-Core Constructions

The preprocessing algorithms of *SLING* (i.e., Algorithms 1, 2, and 4) are *embarrassingly parallelizable*. In particular, Algorithm 1 (and Algorithm 4) can be simultaneously applied to multiple nodes v_k to compute the corresponding approximate correction factors $\tilde{d}_k$. Meanwhile, the main loop of Algorithm 2 (i.e., Lines 2-16) can be parallelized to construct the "reverse" HP sets R_k for multiple nodes v_k at the same time.

Furthermore, *SLING* does not require the complete index structure to fit in the main memory. Instead, we only need to keep all approximate correction factors $\tilde{v_k}$ ($k \in [1, n]$) in the memory, but can store the approximate HP set $H(v_x)$ for each node v_x on the disk. To process a single-pair SimRank query on two nodes v_i and

v_j, we retrieve $H(v_i)$ and $H(v_j)$ from the disk and combine them with $\tilde{v}_k$ to derive the query result, which incurs a constant I/O cost, since $H(v_i)$ and $H(v_j)$ takes only $O(1/\varepsilon)$ space. In addition, the index construction process of *SLING* does not require maintaining all HP sets $H(v_x)$ simultaneously in the memory. Specifically, in Algorithm 2, we can construct each "reverse" HP set R_k in turn and write them to the disk; after that, we can construct all approximate HP sets $H(v_x)$ in a batch, by using an external sorting algorithm to sort all HPs $\tilde{h}^{(\ell)}(v_x, v_k)$ by v_x. This process requires only $O(\frac{n}{\varepsilon} \log \frac{n}{\varepsilon})$ I/O accesses, since the total size of all $H(v_x)$ is $O(n/\varepsilon)$.

6. EXTENSION TO SINGLE-SOURCE QUERIES

Given the *SLING* index introduced in Sections 4, we can easily answer any single-source SimRank query from a node v_i, by invoking Algorithm 3 n times to compute $s(v_i, v_j)$ for each node v_j. This leads to a total query cost of $O(n/\varepsilon)$, which is near-optimal since any single-source SimRank method requires $\Omega(n)$ time to output the results. This straightforward algorithm, however, can be improved in terms of practical efficiency. To explain this, let us consider two nodes v_i and v_j, such that $H(v_i)$ and $H(v_j)$ do not contain any HPs to the same node at the same step, i.e.,

$$\nexists v_k, \ell, \;\; \tilde{h}^{(\ell)}(v_i, v_k) \in H(v_i) \wedge \tilde{h}^{(\ell)}(v_j, v_k) \in H(v_j).$$

Then, *SLING* would return $\tilde{s}(v_i, v_j) = 0$. We say that $H(v_i)$ and $H(v_j)$ do not *intersect* in this case. Intuitively, if we can avoid accessing those HP sets $H(v_j)$ that do not intersect with $H(v_i)$, then we can improve the efficiency of the single-source SimRank query from v_i. For this purpose, a straightforward approach is to maintain, for each combination of v_k and ℓ, an *inverted list* $L(v_k, \ell)$ that records the approximate HPs $\tilde{h}^{(\ell)}(v_x, v_k)$ from any node v_x to v_k. Then, to process a single-source SimRank query from node v_i, we first examine each approximate HP $\tilde{h}^{(\ell)}(v_i, v_k) \in H(v_i)$ and retrieve $L(v_k, \ell)$, based on which we compute $\tilde{s}(v_i, v_j)$ for any node v_j with $\tilde{s}(v_i, v_j) > 0$.

Although the inverted list approach improves efficiency for single-source SimRank queries, it doubles the space consumption of *SLING*, since the inverted lists have the same total size as the approximate HP sets $H(v_i)$. Furthermore, the approach cannot be combined with the space reduction technique in Section 5.2, because the former requires storing all approximate HPs in the inverted lists, whereas the latter aims to omit certain HPs to save space. To address this issue, we propose a single-source SimRank algorithm for *SLING* that finds a middle ground between the inverted list approach and the straightforward approach. The basic idea is that, given node v_i, we first retrieve all approximate HPs $\tilde{h}^{(\ell)}(v_i, v_k) \in H(v_i)$, and then apply a variant of Algorithm 2 to compute the HPs from other nodes to each v_k; after that, we combine all HPs obtained to derive the query results. In other words, we construct the inverted lists relevant for the single-source query *on the fly*, instead of pre-computing them in advance.

Algorithm 6 shows the details of our method. It takes as input a query node v_i and the threshold θ used in constructing $H(v_i)$ (see Algorithm 2), and returns an approximate SimRank score $\tilde{s}(v_i, v_j)$ for each node v_j. The algorithm starts by initializing $\tilde{s}(v_i, v_j) = 0$ for all v_j (Line 1). Then, it identifies the steps ℓ such that there is at least one step-ℓ approximate HP in $H(v_i)$; after that, it processes each of those steps in turn (Lines 2-10). The general idea of processing is as follows. By Equation 13, if v_i has a positive HP to a node v_k at step ℓ, then for any other node v_j with a positive HP to v_k at step ℓ, we have $s(v_i, v_j) > 0$. To identify such nodes v_j and their SimRank scores with v_i, we can apply the local update approach in Algorithm 2 to traverse ℓ steps from v_k; however, the local update procedure needs to be slightly modified to deal with the fact that we may need to traverse from multiple v_k simultaneously, i.e., when v_i have positive HPs to multiple nodes at step ℓ.

Algorithm 6: *An algorithm for single-source SimRank queries*

Input: query node v_i and threshold θ
Output: an approximate SimRank score $\tilde{s}(v_i, v_j)$ for each node v_j

1 Initialize $\tilde{s}(v_i, v_j) = 0$ for all v_j;
2 **for** *each* ℓ *such that* $H(v_i)$ *contains some approximate HP at step* ℓ **do**
3 **for** *each node* v_k *such that* $\tilde{h}^{(\ell)}(v_i, v_k) \in H(v_i)$ **do**
4 Initialize $\rho^{(0)}(v_k) = \tilde{h}^{(\ell)}(v_i, v_k) \cdot d_k$;
5 **for** $t = 1$ *to* ℓ **do**
6 **for** *each node* v_x *such that* $\rho^{(t-1)}(v_x) > (\sqrt{c})^{\ell} \cdot \theta$ **do**
7 **for** *each out-neighbor* v_y *of* v_x **do**
8 **if** $\rho^{(t)}(v_y)$ *does not exist* **then**
9 $\rho^{(t)}(v_y) = \frac{\sqrt{c}}{|I(v_y)|} \cdot \rho^{(t-1)}(v_x)$;
10 **else**
11 $\rho^{(t)}(v_y) = \rho^{(t)}(v_y) + \frac{\sqrt{c}}{|I(v_y)|} \cdot \rho^{(t-1)}(v_x)$;
12 **for** *each* v_j *such that* $\rho^{(\ell)}(v_j) > 0$ **do**
13 $\tilde{s}(v_i, v_j) = \tilde{s}(v_i, v_j) + \rho^{(\ell)}(v_j)$;
14 **return** $\tilde{s}(v_i, v_j)$ for each node v_j;

Specifically, for each particular ℓ, Algorithm 6 first identifies each node v_k such that $\tilde{h}^{(\ell)}(v_i, v_k) \in H(v_i)$, and initializes a temporary score $\rho^{(0)}(v_k) = \tilde{h}^{(\ell)}(v_i, v_k)$ for v_k (Line 3). After that, it traverses ℓ steps from all v_k simultaneously (Lines 5-8). In the t-th step ($t \in [1, \ell]$), it inspects the temporary scores created in the $(t-1)$-th step, and omit those scores that are no larger than $(\sqrt{c})^{\ell} \cdot \theta$ (Line 6). This omission is similar to the pruning of HPs applied in Algorithm 2, except that the threshold used here is $(\sqrt{c})^{\ell}$ times smaller than the threshold θ used in Algorithm 2. The reason is that the local update procedure in Algorithm 2 starts from a node whose approximate HP equals 1, whereas the procedure in Algorithm 6 begins from a node whose temporary score $\rho^{(0)}(v_k) \leq (\sqrt{c})^{\ell}$, due to which we need to scale down the threshold to ensure accuracy.

For each temporary score $\rho^{(t-1)}(v_x)$ that is above the threshold, Algorithm 2 examines each out-neighbor v_y of v_x, and checks whether the temporary score of v_y at step t (denoted as $\rho^{(t)}(v_y)$) exists. If it does not exist, then the algorithm initializes it as $\rho^{(t)}(v_y) = \frac{\sqrt{c}}{|I(v_y)|} \cdot \rho^{(t-1)}(v_x)$; otherwise, the algorithm increases it by $\frac{\sqrt{c}}{|I(v_y)|} \cdot \rho^{(t-1)}(v_x)$ (Lines 7-11). (Observe that this update rule is identical to that in Algorithm 2.) Finally, after the ℓ-step traversal is finished, the algorithm adds each temporary score $\rho^{(\ell)}(v_j)$ at step ℓ into $\tilde{s}(v_i, v_j)$, and then proceeds to consider the next ℓ (Lines 12-14). Once all steps ℓ are processed, the algorithm returns each $\tilde{s}(v_i, v_j)$ as the final result.

We have the following lemma regarding the theoretical guarantees of Algorithm 6.

LEMMA 12. *Algorithm 6 runs in* $O\left(m \log^2 \frac{1}{\varepsilon}\right)$ *time, and ensures that each SimRank score returned has* ε *worst-case error.* □

The time complexity of Algorithm 6 is not as attractive as those of the inverted list approach and the straightforward approach, but is roughly comparable to the latter when $m = O(n/\varepsilon)$ (as is often the case in practice). In addition, we note that the time complexity of Algorithm 6 matches that of the more recent method for single-source SimRank queries [24], even though the latter relies on heuristic assumptions that do not hold in general (see Section 3.3).

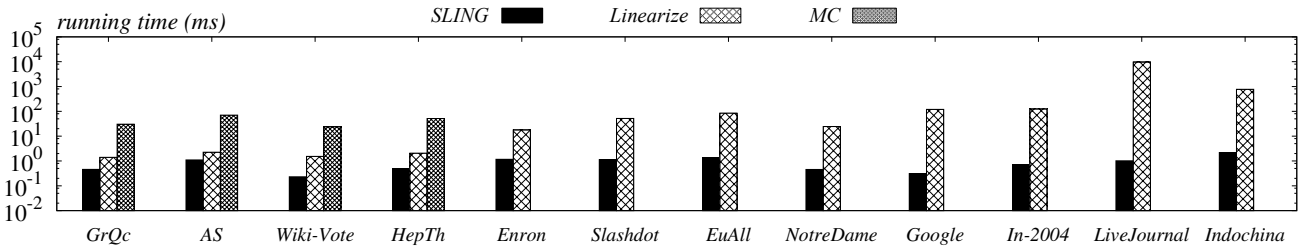


Figure 1: Average query costs for single-pair SimRank queries.

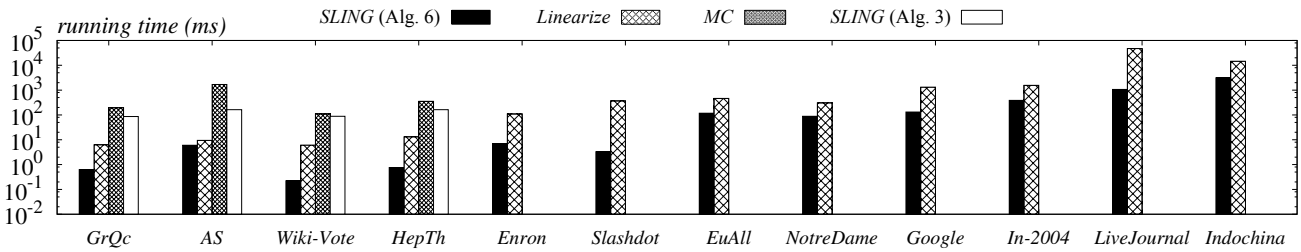


Figure 2: Average query costs for single-source SimRank queries.

Table 3: Datasets.

Dataset	Type	n	m
GrQc	undirected	5,242	14,496
AS	undirected	6,474	13,895
Wiki-Vote	directed	7,155	103,689
HepTh	undirected	9,877	25,998
Enron	undirected	36,692	183,831
Slashdot	directed	77,360	905,468
EuAll	directed	265,214	400,045
NotreDame	directed	325,728	1,497,134
Google	directed	875,713	5,105,049
In-2004	directed	1,382,908	17,917,053
LiveJournal	directed	4,847,571	68,993,773
Indochina	directed	7,414,866	194,109,311

7. EXPERIMENTS

This section experimentally evaluates *SLING*. Section 7.1 clarifies the experimental settings, and Section 7.2 presents the experimental results.

7.1 Experimental Settings

Datasets and Environment. We use twelve graph datasets that are publicly available from [1, 2] and are commonly used in the literature. Table 3 shows the statistics of each graph. We conduct all of experiments on a Linux machine with a 2.6GHz CPU and 64GB memory. All methods tested are implemented in C++. (Our code is available at [3].)

Methods and Parameters. We compare *SLING* against two state-of-the-art methods for SimRank computation: the linearization method [24, 25] (referred to as *Linearize*) and the Monte Carlo method [8] (referred to as *MC*). *Linearize* has three parameters T, R, and L. Following the recommendations in [24], we set $T = 11$, $R = 100$, and $L = 3$. In addition, we set the decay factor c in the SimRank model to 0.6, as suggested in previous work [23, 24, 31–33]. Under this setting, *Linearize* ensures a worst-case error $\varepsilon = c^T/(1-c) \approx 0.01$ in each SimRank score, *if it is able to derive an exact diagonal correction matrix D*. However, as we discuss in Section 3.3, *Linearize* utilizes an approximate version of D that provides no quality assurance, due to which the above error bound does not hold.

For *SLING*, we set its maximum error $\varepsilon = 0.025$, which is roughly comparable to the quality assurance of the linearization method given a precise D. Towards this end, we set $\varepsilon_d = 0.005$ and $\theta = 0.000725$, which ensures $\varepsilon < 0.025$ by Theorem 1. In addition, we set $\delta_d = 1/n^2$, which guarantees that the preprocessing algorithm of *SLING* succeeds with at least $1 - 1/n$ probability. For *MC*, we set $\varepsilon = 0.025$, as in *SLING*.

7.2 Experimental Results

In the first set of experiments, we randomly generate 1000 single-pair SimRank queries on each dataset, and evaluate the average computation time of each method in answering the queries. Figure 1 shows the results. We omit *MC* on all but the four smallest datasets, since its index size exceeds 64GB on the large graphs. Observe that the query time of *SLING* is at most 2.2ms in all cases, and is often several orders of magnitude smaller than that of *Linearize*. In particular, on *LiveJournal*, *SLING* is around 10000 times faster than *Linearize*. This is consistent with the fact that *SLING* and *Linearize* has $O(1/\varepsilon)$ and $O(m \log \frac{1}{\varepsilon})$ query time complexities, respectively. Meanwhile, *Linearize* incurs a smaller query cost than *MC* on the four smallest datasets, which is also observed in previous work [24].

Our second set of experiments evaluates the average computation cost of each method in answering 500 random single-source SimRank queries. For *SLING*, we consider two different methods: one that directly uses Algorithm 6, and another one that invokes Algorithm 3 once for each node. Figure 2 illustrates the results. Notice that the method that applies Algorithm 3 is significantly slower than Algorithm 6, even though the former (resp. latter) runs in $O(n/\varepsilon)$ time (resp. $O(m \log^2 \frac{1}{\varepsilon})$ time). This is in accordance with our analysis in Section 6, which shows that adopting Algorithm 3 for single-source queries would incur unnecessary overheads and lead to inferior query time. Since the method that employs Algorithm 3 is not competitive, we omit it on all but the four smallest datasets.

Among all methods for single-source SimRank queries, *SLING* (with Algorithm 6) achieves the best performance, but its improvement over *Linearize* is less pronounced when compared with the case of single-pair queries. This, as we mention in Section 6, is because the local update procedure in Algorithm 6 incurs super-linear overheads, due to which the algorihtm's time complexity is the same as *Linearize*'s. Nonetheless, *SLING* is still at least 9 times

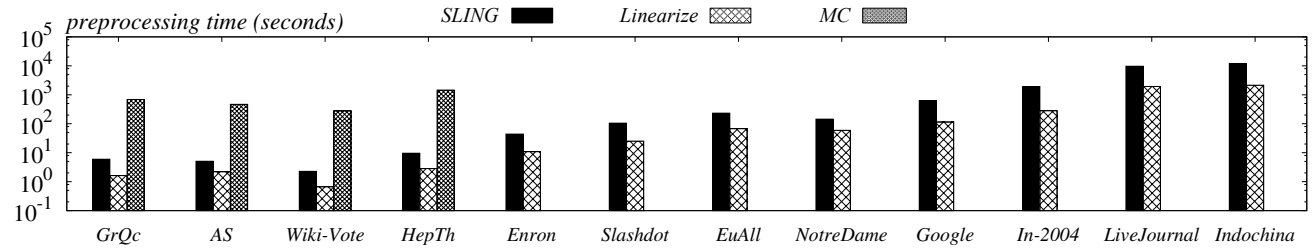


Figure 3: Preprocessing cost of each method.

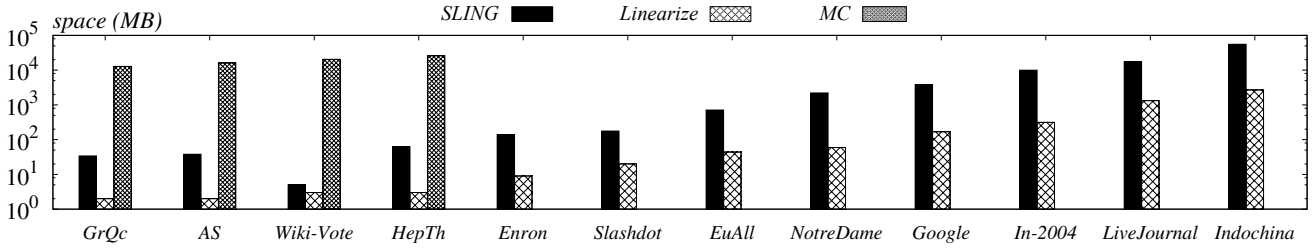
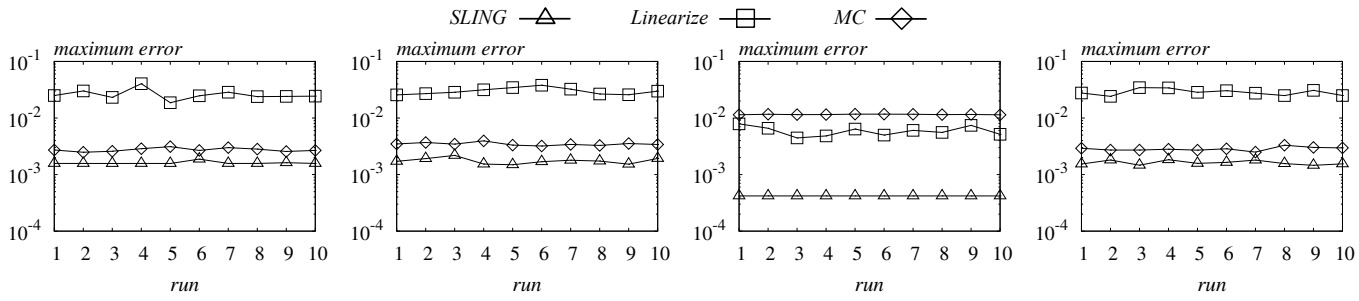


Figure 4: Space consumption of each method.



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

Figure 5: Maximum SimRank error of each method measured in 10 different runs.

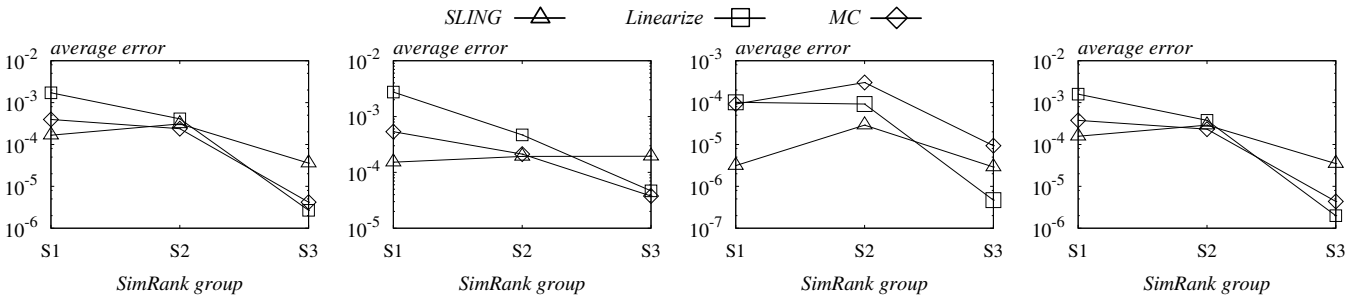
faster than *Linearize* on 7 out of the 12 datasets, and is 110 times more efficient on *Slashdot*. Meanwhile, *MC* is consistently outperformed by *Linearize*.

Next, we plot the the preprocessing cost (resp. space consumption) of each method in Figure 3 (resp. Figure 4). *Linearize* incurs a smaller pre-computation cost than *SLING* does; in turn, *SLING* is more efficient than *MC* in terms of pre-computation. The index size of *SLING* is considerably larger than *Linearize*, since *SLING* has an $O(n/\varepsilon)$ space complexity, while *Linearize* only incurs $O(n+m)$ space overhead. Nevertheless, *SLING* outperforms *MC* in terms of space efficiency. Overall, *SLING* is inferior to *Linearize* in terms of space overheads and preprocessing costs, but this is justified by the fact that *SLING* offers superior query efficiency and rigorous accuracy guarantee, whereas *Linearize* incurs significantly larger query costs and does not offer non-trivial bounds on its query errors. Furthermore, the pre-computation algorithm of *SLING* can be easily parallelized, as we discuss in Section 5.4 and demonstrate in Appendix C.

Our last three experiments focus on the query accuracy of each method. We first apply the power method (see Section 3.1) on each of the four smallest graphs to compute the SimRank score of each node pair, setting the number of iterations in the method to 50 (which results in a worst-case error below 10^{-11}). We take the SimRank scores thus obtained as the ground truth, and use them to gauge the error of each method computing all-pair SimRank scores. We do not repeat this experiment on larger graphs, due to the tremendous overheads in computing all-pair SimRank results.

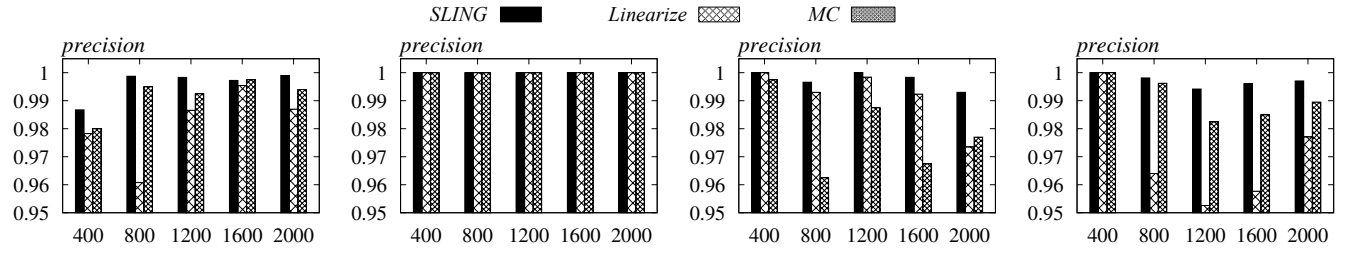
Figure 5 illustrates the maximum query error incurred by each method in all-pair SimRank computation over 10 different runs, where each run rebuilds the index of each method from scratch. Observe that the maximum error of *SLING* is always below 0.0025, which is considerably smaller than the stipulated error bound $\varepsilon = 0.025$. *MC*'s maximum error is also below $\varepsilon = 0.025$, but is consistently larger than that of *SLING*, and is over 0.01 on *Wiki-Vote*. In contrast, the maximum error of *Linearize* is above 0.025 in most runs on *GrQc*, *AS*, and *Hepth*, which is consistent with our analysis that *Linearize* does not offer any worst-case guarantee in terms of query accuracy.

To further assess each method's query accuracy, we divide the ground-truth SimRank scores into three groups S_1, S_2, and S_3, such that S_1 (resp. S_2) contains SimRank scores in the range of $[0.1, 1]$ (resp. $[0.01, 0.1]$), while S_3 concerns SimRank scores smaller than 0.01. Intuitively, the scores in S_1 and S_2 are more important than those in S_3, since the former correspond to node pairs that are highly similar. Figure 6 shows the average query errors of each method for S_1, S_2, and S_3. Observe that, compared with *Linearize*, *SLING* incurs much smaller (resp. slightly smaller) errors on S_1 (resp. S_2). This indicates that *SLING* is more effective than *Linearize* in measuring the similarity of important node pairs. Meanwhile, *MC* is less accurate than *SLING* on S_1, and is considerably outperformed by both *SLING* and *Linearize* on *Wiki-Vote*.



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

Figure 6: Average SimRank error vs. SimRank group.



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

Figure 7: Precision of the top-k SimRank pairs returned by each method.

Finally, we use the all-pair SimRank scores computed by each method to identify the k node pairs with the highest SimRank scores[1], and we measure the *precision* of those k pairs, i.e., the fraction of them among the ground-truth top-k pairs. Figure 7 illustrates the results when k varies from 400 to 2000. The precision of *SLING* is never worse than that of *Linearize*, and is up to 4% higher than the latter in many cases. This is consistent with our results in Figure 6 that, for node pairs with large SimRank scores, *SLING* provides much higher accuracy than *Linearize* does. Meanwhile, *MC* yields lower accuracy than *SLING* does, and is significantly outperformed by both *SLING* and *Linearize* on *Wiki-Vote*. These results are also in agreement with those in Figure 6.

8. OTHER RELATED WORK

The previous sections have discussed the existing techniques that are most relevant to ours. In what follows, we survey other related work on SimRank computation. First, there is a line of research [10, 13, 19, 30–32] on SimRank queries based on the following formulation of SimRank:

$$S = cP^\top SP + (1-c)I,$$

where S, P, and T are $n \times n$ matrices such that $S(i,j) = s(v_i, v_j)$ for any i, j, P is as defined in Equation 5, and I is an identity matrix. However, as point out by Kusumoto et al. [17], the above formulation is *incorrect* since it assumes that $(1-c)I$ equals the diagonal correction matrix D (see Equation 7), which does not hold in general. As a consequence, the methods in [10, 13, 19, 30–32] fail to offer any guarantees in terms of the accuracy of SimRank scores, due to which we do not consider them in this paper.

Second, several variants [5, 8, 22, 33, 35] of SimRank have been proposed to enhance the quality of similarity measure and mitigate certain limitations of SimRank. Antonellis et al. [5] present *SimRank++*, which extends SimRank by taking into account the weights of edges and prior knowledge of node similarities. Jin et al. [16] introduce *RoleSim*, which guarantees to recognize automorphically or structurally equivalent nodes. Fogaras and Rácz [8] propose *PSimRank*, which improves the quality of SimRank by allowing random walks that are close to each other to have a higher probability to meet. Yu and McCann [33] present *SimRank$^\#$*, which defines the similarity between two nodes based on the *consine similarity* of their neighbors. Zhao et al. [35] introduce *P-Rank*, which consider both in-neighbors and out-neighbors of two nodes when measuring their similarity.

Finally, there is existing work [10, 17, 18, 25, 28, 36] that studies *top-k SimRank queries* and *SimRank similarity joins*. In particular, a top-k SimRank queries takes as input a node v_i, and asks for the k nodes v_j with the largest SimRank score $s(v_i, v_j)$. Meanwhile, a SimRank similarity join asks for all pairs of nodes whose SimRank scores are among the largest k, or are larger than a predefined threshold. Techniques designed for these two types of queries are generally inapplicable for single-pair and single-source SimRank queries.

9. CONCLUSIONS

This paper presents the *SLING* index for answering single-pair and single-source SimRank queries with ε worst-case error in each SimRank score. *SLING* requires $O(n/\varepsilon)$ space and $O(m/\varepsilon + n \log \frac{n}{\delta}/\varepsilon^2)$ pre-computation time, and it handles any single-pair (resp. single-source) query in $O(1/\varepsilon)$ (resp. $O(n/\varepsilon)$) time. The space and query time complexities of *SLING* are near-optimal, and are significantly better than those of the existing solutions. In addition, *SLING* incorporates several optimization techniques that considerably improves its practical performance. Our experiments show that *SLING* provides superior query efficiency against the states of the art. For future work, we plan to (i) investigate techniques to reduce the index size of *SLING*, and (ii) extend *SLING* to handle other similarity measures for graphs.

[1]Note that we ignore any node pair containing two nodes that are identical.

10. REFERENCES

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

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

[3] https://sourceforge.net/projects/slingsimrank/.

[4] R. Andersen, F. R. K. Chung, and K. J. Lang. Local graph partitioning using pagerank vectors. In *FOCS*, pages 475–486, 2006.

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

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

[7] P. Dagum, R. M. Karp, M. Luby, and S. M. Ross. An optimal algorithm for monte carlo estimation. *SIAM J. Comput.*, 29(5):1484–1496, 2000.

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

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

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

[11] G. H. Golub and C. F. Van Loan. *Matrix Computations*. Johns Hopkins University Press, 3 edition, 2012.

[12] P. Gupta, A. Goel, J. Lin, A. Sharma, D. Wang, and R. Zadeh. WTF: the who to follow service at twitter. In *WWW*, pages 505–514, 2013.

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

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

[15] G. Jeh and J. Widom. Scaling personalized web search. In *WWW*, pages 271–279, 2003.

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

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

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

[19] 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.

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

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

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

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

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

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

[26] S. Rothe and H. Schütze. Cosimrank: A flexible & efficient graph-theoretic similarity measure. In *ACL*, pages 1392–1402, 2014.

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

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

[29] B. Tian and X. Xiao. SLING: A near-optimal index structure for simrank. *CoRR*, abs/1604.04185, 2016.

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

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

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

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

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

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

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

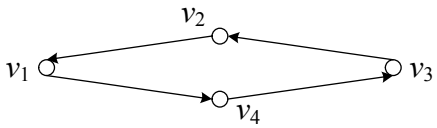


Figure 8: An adversarial case for the linearization method.

APPENDIX

A. LIMITATIONS OF THE LINEARIZATION METHOD

Recall that the linearization method [24] requires pre-computing the diagonal correction matrix D. Maehara et al. [24] prove that the diagonal elements in D satisfy the following linear system:

$$\text{for all } k \in [1, n], \qquad \sum_{\ell=0}^{\infty} \sum_{i=1}^{n} c^{\ell} \left(p_{k,i}^{(\ell)}\right)^2 D(i,i) = 1, \qquad (18)$$

where $p_{k,i}^{(\ell)}$ is the probability that v_i is the ℓ-th step of a reverse random walk from v_k. Based on this, the linearization method estimates $p_{k,i}^{(\ell)}$ with a set of reverse random walks, and then incorporates the estimated values into a truncated version of Equation (18):

$$\text{for all } k \in [1, n], \qquad \sum_{\ell=0}^{t} \sum_{i=1}^{n} c^{\ell} \left(\tilde{p}_{k,i}^{(\ell)}\right)^2 D(i,i) = 1, \qquad (19)$$

where $\tilde{p}_{k,i}^{(\ell)}$ denotes the estimated version of $p_{k,i}^{(\ell)}$. After that, it applies the Gauss-Seidel technique [11] to solve Equation (19), and obtains an $n \times n$ diagonal matrix $\widetilde{D}$ that approximates D.

The above approach for deriving $\widetilde{D}$ is interesting, but it fails to provide any worst-case guarantee in terms of the pre-computation time and the accuracy of SimRank queries, due to the following reasons. First, because of the sampling error in $\tilde{p}_{k,i}^{(\ell)}$ and the truncation applied in Equation (19), $\widetilde{D}$ could differ considerably from D, which may in turn lead to significant errors in SimRank computation. There is no formal result on how large the error in $\widetilde{D}$ could be. Instead, Maehara et al. [24] only show that the error in $\tilde{p}_{k,i}^{(\ell)}$ can be bounded by using a sufficiently large sample set of reverse random walks; however, it does not translate into any accuracy guarantee on $\widetilde{D}$.

Second, even if $\tilde{p}_{k,i}^{(\ell)} = p_{k,i}^{(\ell)}$ and Equation (19) is not truncated, the Gauss-Seidel technique [11] used by the linearization method to solve Equation (19) may not converge. In particular, if we define an $n \times n$ matrix M as

$$M(k,i) = \sum_{\ell=0}^{\infty} \sum_{i=1}^{n} c^{\ell} \left(p_{k,i}^{(\ell)}\right)^2,$$

then the linearization method requires that M should be diagonally dominant, i.e., for any i, $|M(i,i)| \geq \sum_{j \neq i} |M(i,j)|$. However, this requirement is not always satisfied. For example, consider the graph in Figure 8. The linear system corresponding to the graph is

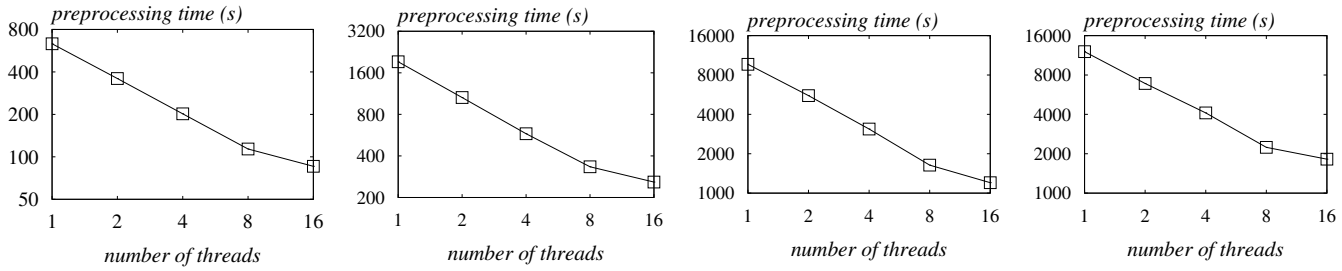


Figure 9: Preprocessing time vs. number of threads.

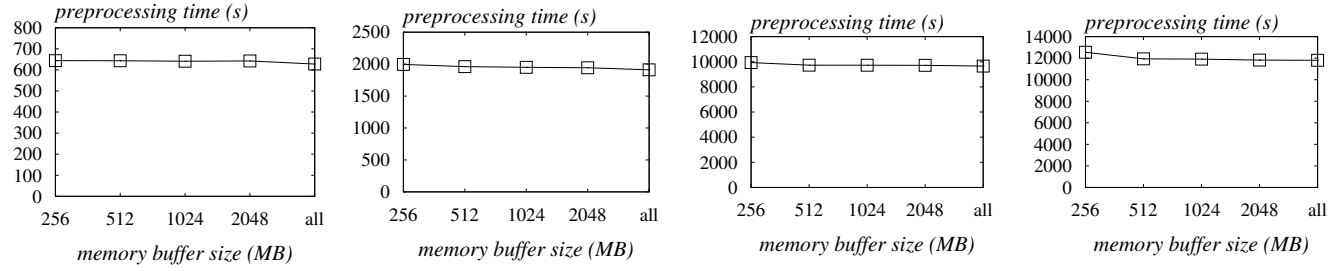


Figure 10: Preprocessing time vs. memory buffer size.

$$\frac{1}{1-c^4}\begin{pmatrix} 1 & c & c^2 & c^3 \\ c^3 & 1 & c & c^2 \\ c^2 & c^3 & 1 & c \\ c & c^2 & c^3 & 1 \end{pmatrix}\begin{pmatrix} D(1,1) \\ D(2,2) \\ D(3,3) \\ D(4,4) \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \end{pmatrix}$$

It can be verified the matrix 4×4 matrix M on the left hand side is not diagonally dominant when $c = 0.6$.

Finally, the number of iterations required by the Gauss-Seidel method is $O(\log \varepsilon^* / \log \rho)$, where ε^* is the maximum error allowed in the solution to the linear system, and ρ is the spectral radius of the iteration matrix used by the method [11]. The value of ρ depends on the input graph, and might be very close to 1, in which case $\log \varepsilon^* / \log \rho$ can be an extremely large number.

B. HITTING PROBABILITIES VS. PERSONALIZED PAGERANKS

Suppose that we start a random walk from a node v_i following the outgoing edges of each node, with $1 - c_p$ probability to stop at each step. The probability that the walk stops at a node v_j is referred to as the *personalized PageRank (PPR)* [15] from v_i to v_j. PPR is well-adopted as a metric for measuring the *relevance* of nodes with respect to the input node v_i, and it has important applications in web search [15] and social network analysis [12].

Our notion of hitting probabilities (HP) bears similarity to PPR, but differs in the following aspect:

1. HP concerns the probability that the random walk reaches node v_j at a particular step ℓ, but disregards whether the random walk stops at v_j;
2. PPR only concerns the endpoint v_j of the random walk, and disregards all nodes before it.

Our Algorithm 2 for computing approximate HPs is inspired by the *local update* algorithm [4, 9, 15] proposed for computing approximate PPRs. Specifically, given a node v_j and an error bound ε, the local update algorithm returns an approximate version of the PPRs from other nodes to v_j, with ε worst-case errors. The algorithm starts by assigning a *residual* 1 to v_j, and 0 to any other node. Subsequently, the algorithm iteratively propagates the residual of each node to its in-neighbors, during which it computes the approximate PPR from each node to v_j. When the largest residual in all nodes is smaller than ε, the algorithm terminates. This algorithm is similar in spirit to our Algorithm 2, but it cannot be directly applied in our context, due to the inherent differences between PPRs and HPs.

C. ADDITIONAL EXPERIMENTS

In this section, we evaluate the parallel and out-of-core algorithms for constructing the index structures of *SLING* (presented in Section 5.4), using the four largest datasets in Table 3. First, we implement a multi-threaded version of *SLING*'s pre-computation algorithm, and measure its running time when the number of threads varies from 1 to 16 and all 64GB main memory on our machine is available. (The total number of CPU cores on our machine is 16.) Figure 9 illustrates the results. Observe that the algorithm achieves a near-linear speed-up as the number of threads increases, which is consistent with our analysis (in Section 5.4) that *SLING* preprocessing algorithm is embarrassingly parallelizable.

Next, we implement an I/O-efficient version of *SLING*'s preprocessing algorithm, based on our discussions in Section 5.4. Then, we measure the running time of the algorithm when it uses one CPU core along with a memory buffer of a pre-defined size. (We assume that the input graph is memory-resident, and we exclude it when calculating the memory buffer size.) Figure 10 shows the processing time of the algorithm as the buffer size varies. Observe that the algorithm can efficiently process all tested graphs even when the buffer size is as small as 256MB. In addition, the overhead of the algorithm does not increase significantly when the buffer size decreases, since the algorithm is CPU-bound. In particular, its only I/O cost is incurred by (i) writing each entry in the index once to the disk, and (ii) performing an external sort on the entries.

D. CONCENTRATION INEQUALITIES

This section introduces the concentration inequalities used in our proofs. We start from the classic Chernoff bound.

LEMMA 13 (CHERNOFF BOUND [6]). *For any set $\{x_i\}$ ($i \in [1, n_x]$) of i.i.d. random variables with mean μ and $x_i \in [0, 1]$,*

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

Our proofs also use a concentration bound on *martingales*, as detailed in the following.

DEFINITION 1 (MARTINGALE). *A sequence of random variables $y_1, y_2, y_3, \cdots$ is a martingale if and only if $\mathbb{E}[y_i] < +\infty$ and $\mathbb{E}[y_{i+1}|y_1, y_2, \cdots, y_i] = y_i$ for any i.* □

LEMMA 14 ([6]). *Let $y_1, y_2, y_3, \cdots$ be a martingale, such that $|y_1| \leq a$, $|y_{j+1} - y_j| \leq a$ for any $j \in [1, i-1]$, and*

$$Var[y_1] + \sum_{j=2}^{i} Var[y_j \mid y_1, y_2, \cdots, y_j - 1] \leq b_i,$$

where $Var[\cdot]$ denotes the variance of a random variable. Then, for any $\lambda > 0$,

$$\Pr\{y_i - \mathbb{E}[y_i] \geq \lambda\} \leq \exp\left(-\frac{\lambda^2}{\frac{2}{3}a\lambda + 2b_i}\right)$$

E. PROOFS

This section presents the proofs of the theorems and lemmas in the paper. Due to space constraints, we omit the proofs of Lemmas 9-12 but include them in our technical report [29].

Proof of Lemma 3. Let $s'(v_i, v_j)$ be the probability that W_i and W_j meet. If $v_i = v_j$, then $s'(v_i, v_j) = 1$, since W_i and W_j always meet at the first step. Suppose that $v_i \neq v_j$. Then, $s'(v_i, v_j)$ is the probability that W_i and W_j meet at or after the second step. Assume without loss of generality that the second steps of W_i and W_j are v_k and v_ℓ, respectively. By definition, $s'(v_k, v_\ell)$ equals the probability that W_i and W_j meet at or after v_k and v_ℓ. Taking into account all possible second steps of W_i and W_j, we have

$$\begin{aligned} s'(v_i, v_j) &= \sum_{v_k \in I(v_i), v_\ell \in I(v_j)} \frac{\sqrt{c}}{|I(v_i)|} \cdot \frac{\sqrt{c}}{|I(v_j)|} \cdot s'(v_k, v_\ell) \\ &= \frac{c}{|I(v_i)| \cdot |I(v_j)|} \sum_{v_k \in I(v_i), v_\ell \in I(v_j)} s'(v_k, v_\ell). \end{aligned}$$

As such, $s'(v_i, v_j)$ have the same definition as $s(v_i, v_j)$ (see Equation (1)), which indicates that $s'(v_i, v_j) = s(v_i, v_j)$. □

Proof of Lemma 4. First, we define the following events:

- $E(v_i, v_j)$: Two $\sqrt{c}$-walks starting from v_i and v_j, respectively, meet each other.
- $L(v_i, v_j, v_k, \ell)$: Two $\sqrt{c}$-walks starting from v_i and v_j, respectively, *last meet* each other at the ℓ-th step at v_k.

As we discuss in Section 4.2, two different events $L(v_i, v_j, v_k, \ell)$ and $L(v_i, v_j, v'_k, \ell')$ are mutually exclusive whenever $v_k \neq v'_k$ or $\ell \neq \ell'$. Therefore,

$$\Pr\{E(v_i, v_j)\} = \sum_{\ell=0}^{+\infty} \sum_{k=1}^{n} \Pr\{L(v_i, v_j, v_k, \ell)\}$$

Observe that the probability of $L(v_i, v_j, v_k, \ell)$ can be computed by multiplying the following two probabilities:

1. The probability that two $\sqrt{c}$-walks W_i and W_j from v_i and v_j, respectively, meet at v_k at step ℓ.
2. Given that W_i and W_j meet at v_k step ℓ, the probability that they do not meet at steps $\ell + 1, \ell + 2, \ldots$.

The first probability equals $h^{(\ell)}(v_i, v_k) \cdot h^{(\ell)}(v_i, v_k)$. Meanwhile, since the $(x+1)$-th step of any $\sqrt{c}$-walk depends only on its x-th step, the second probability should equal the probability that two $\sqrt{c}$-walks from v_k never meet after the 0-th step, which in turn equals d_k. Hence, we have

$$\begin{aligned} s(v_i, v_j) = \Pr\{E(v_i, v_j)\} &= \sum_{\ell=0}^{+\infty} \sum_{k=1}^{n} \Pr\{L(v_i, v_j, v_k, \ell)\} \\ &= \sum_{\ell=0}^{+\infty} \sum_{k=1}^{n} \left(h^{(\ell)}(v_i, v_k) \cdot d_k \cdot h^{(\ell)}(v_j, v_k)\right), \end{aligned}$$

which completes the proof. □

Proof of Lemma 5. Let $R = \sqrt{c} \cdot P$, and R^ℓ be the ℓ-th power of R. We have

$$R^0(k, i) = \begin{cases} 1 & \text{if } i = k \\ 0 & \text{otherwise} \end{cases}$$

Hence, $R^0(k, i) = h^{(0)}(v_i, v_k)$ for all v_i and v_k. Assume that for a certain ℓ, we have $R^\ell(k, i) = h^{(\ell)}(v_i, v_k)$ for all v_i and v_k. Then,

$$\begin{aligned} R^{\ell+1}(k, i) &= \left(\sqrt{c} \cdot P \cdot R^\ell\right)(k, i) \\ &= \sum_{j=1}^{n} \left(\sqrt{c} \cdot P(k, j) \cdot R^\ell(j, i)\right) \\ &= \sum_{\text{each out-neighbor } v_j \text{ of } v_k} \left(\frac{\sqrt{c}}{|I(v_j)|} \cdot h^{(\ell)}(v_i, v_j)\right) \\ &= h^{(\ell+1)}(v_i, v_k). \end{aligned}$$

Therefore, $R^\ell(k, i) = h^{(\ell)}(v_i, v_k)$ for all v_i, v_k, and ℓ.

Let $\tilde{D}$ be the $n \times n$ diagonal matrix whose k-th diagonal element is d_k. Then, Equation (13) can be written as:

$$S = \sum_{\ell=0}^{+\infty} \left(\left(R^\ell\right)^\top \tilde{D} R^\ell\right)$$

By multiplying $R^\top$ and R on the left and right, respectively, on both side of the equation, we have

$$S = R^\top S R + \tilde{D} = cP^\top S P + \tilde{D}.$$

This indicates that $\tilde{D}$ is a diagonal correction matrix. Since the diagonal correction matrix is unique [24], we have $\tilde{D} = D$. □

Proof of Lemma 6. By the Chernoff Bound in Lemma 13,

$$\begin{aligned} \Pr\left\{\left|\frac{cnt}{n_r} - \mu\right| \geq \varepsilon/c\right\} &\leq 2\exp\left(-\frac{n_r(\varepsilon/c)^2}{\frac{2}{3}\varepsilon/c + 2\mu}\right) \\ &= 2\exp\left(-\frac{\frac{2}{3}\varepsilon/c + 2}{\frac{2}{3}\varepsilon/c + 2\mu}\log\frac{2}{\delta_d}\right) \\ &= \delta_d \end{aligned}$$

Therefore, $|\tilde{d_k} - d_k| = c \cdot |\frac{cnt}{n_r} - \mu| \leq \varepsilon$ with at least $1 - \delta_d$ probability. □

Proof of Lemma 7. According to Algorithm 2, for all $\tilde{h}^{(\ell)}(v_i, v_j) \in H(v_i)$, we have

$$\theta \le \tilde{h}^{(\ell)}(v_i, v_j) \le h^{(\ell)}(v_i, v_j).$$

Then, for each node v_i and each step ℓ,

$$\sum_{v_j \in V} \tilde{h}^{(\ell)}(v_i, v_j) \le \sum_{v_j \in V} h^{(\ell)}(v_i, v_j) = \sqrt{c}^{\ell}$$

Therefore, there are at most $(\sqrt{c})^{\ell}/\theta$ nodes v_j such that $\tilde{h}^{(\ell)}(v_i, v_j) \in H(v_i)$. Therefore, the size of $H(v_i)$ is

$$|H(v_i)| \ \le \ \sum_{\ell=0}^{+\infty} \frac{(\sqrt{c})^{\ell}}{\theta} \ = \ O(1/\theta)$$

Let $\bar{d}$ be the average out-degree of the G. Since a local update is performed on each entry $\tilde{h}^{(\ell)}(v_i, v_j) \in H(v_i)$, the running time of the algorithm is $O(\bar{d}n/\theta) = O(m/\theta)$.

Let ε_ℓ be the upper bound of $|\tilde{h}^{(\ell)}(v_i, v_j) - h^{(\ell)}(v_i, v_j)|$ for all nodes v_i and v_j at step ℓ. When $\ell = 0$, we have $\varepsilon_0 \le \frac{1-(\sqrt{c})^0}{1-\sqrt{c}}\theta$. Assume that $\varepsilon_\ell \le \frac{1-(\sqrt{c})^{\ell}}{1-\sqrt{c}}\theta$ holds for a certain ℓ. Then,

$$\varepsilon_{\ell+1} \le \sqrt{c} \cdot \varepsilon_\ell + \theta = \frac{1-\sqrt{c}^{\ell+1}}{1-\sqrt{c}}\theta.$$

Thus, the lemma is proved. □

Proof of Lemma 8. Given that $\left|\tilde{d}_k - d_k\right| \le \varepsilon_d$ and $-\varepsilon_h^{(\ell)} \le \tilde{h}^{(\ell)}(v_k, v_x) - h^{(\ell)}(v_k, v_x) \le 0$ for any k, x, ℓ, we have

$$\begin{aligned}
&\tilde{h}^{(\ell)}(v_i, v_k) \cdot \tilde{d} \cdot \tilde{h}^{(\ell)}(v_j, v_k) - h^{(\ell)}(v_i, v_k) \cdot d_k \cdot h^{(\ell)}(v_j, v_k) \\
&\le h^{(\ell)}(v_i, v_k) \cdot \tilde{d}_k \cdot h^{(\ell)}(v_j, v_k) - h^{(\ell)}(v_i, v_k) \cdot d_k \cdot h^{(\ell)}(v_j, v_k) \\
&\le h^{(\ell)}(v_i, v_k) \cdot \varepsilon_d \cdot h^{(\ell)}(v_j, v_k).
\end{aligned}$$

Therefore,

$$\begin{aligned}
\tilde{s}(v_i, v_j) - s(v_i, v_j) &\le \sum_{\ell=0}^{+\infty} \sum_{k=1}^{n} h^{(\ell)}(v_i, v_k) \cdot \varepsilon_d \cdot h^{(\ell)}(v_j, v_k) \\
&\le \sum_{\ell=0}^{+\infty} \varepsilon_d \cdot (\sqrt{c})^{\ell} \cdot (\sqrt{c}^{\ell}) \quad = \frac{\varepsilon_d}{1-c}.
\end{aligned}$$

Meanwhile,

$$\begin{aligned}
&h^{(\ell)}(v_i, v_k) \cdot d_k \cdot h^{(\ell)}(v_j, v_k) - \tilde{h}^{(\ell)}(v_i, v_k) \cdot \tilde{d} \cdot \tilde{h}^{(\ell)}(v_j, v_k) \\
&\le h^{(\ell)}(v_i, v_k) \cdot d_k \cdot h^{(\ell)}(v_j, v_k) \\
&\quad - \left(h^{(\ell)}(v_i, v_k) - \varepsilon_h^{(\ell)}\right) \cdot (d_k - \varepsilon_d) \cdot \left(h^{(\ell)}(v_j, v_k) - \varepsilon_h^{(\ell)}\right) \\
&= \left(h^{(\ell)}(v_i, v_k) - \varepsilon_h^{(\ell)}\right) \cdot \varepsilon_d \cdot \left(h^{(\ell)}(v_j, v_k) - \varepsilon_h^{(\ell)}\right) \\
&\quad + \varepsilon_h^{(\ell)} \cdot d_k \cdot \left(h^{(\ell)}(v_j, v_k) - \varepsilon_h^{(\ell)}\right) + \varepsilon_d \cdot \varepsilon_h^{(\ell)} \cdot h^{(\ell)}(v_i, v_k)
\end{aligned}$$

Hence,

$$\begin{aligned}
&s(v_i, v_j) - \tilde{s}(v_i, v_j) \\
&= \sum_{\ell=0}^{+\infty} \sum_{k=1}^{n} \Bigg(\left(h^{(\ell)}(v_i, v_k) - \varepsilon_h^{(\ell)}\right) \cdot \varepsilon_d \cdot \left(h^{(\ell)}(v_j, v_k) - \varepsilon_h^{(\ell)}\right) \\
&\quad + \varepsilon_h^{(\ell)} \cdot d_k \cdot \left(h^{(\ell)}(v_j, v_k) - \varepsilon_h^{(\ell)}\right) + \varepsilon_d \cdot \varepsilon_h^{(\ell)} \cdot h^{(\ell)}(v_i, v_k) \Bigg) \\
&\le \sum_{\ell=0}^{+\infty} \left((\sqrt{c})^{\ell} \cdot \varepsilon_d \cdot (\sqrt{c}^{\ell}) + 2 \cdot \varepsilon_h^{(\ell)} \cdot (\sqrt{c})^{\ell} \right) \\
&= \frac{\varepsilon_d}{1-c} + 2 \sum_{\ell=0}^{+\infty} \left((\sqrt{c}^{\ell}) \cdot \varepsilon_h^{(\ell)} \right).
\end{aligned}$$

This completes the proof. □

Proof of Theorem 1. By Lemma 7, for all k, x, l,

$$-\frac{1-(\sqrt{c})^{\ell}}{1-\sqrt{c}}\theta \le \tilde{h}^{(\ell)}(v_k, v_x) - h^{(\ell)}(v_k, v_x) \le 0.$$

Then,

$$\begin{aligned}
&\frac{\varepsilon_d}{1-c} + 2 \sum_{\ell=0}^{+\infty} \left((\sqrt{c})^{\ell} \cdot \varepsilon_h^{(\ell)} \right) \\
&= \frac{\varepsilon_d}{1-c} + 2 \sum_{\ell=0}^{+\infty} \left((\sqrt{c})^{\ell} \cdot \frac{1-(\sqrt{c})^{\ell}}{1-\sqrt{c}}\theta \right) \\
&= \frac{\varepsilon_d}{1-c} + \frac{2\sqrt{c}}{(1-c)(1-\sqrt{c})}\theta.
\end{aligned}$$

By Lemma 6, $\left|\tilde{d}_k - d_k\right| \le \varepsilon_d$ holds with at least $1-\delta_d$ probability. Since $\delta_d \le \delta/n$, with at least $1-\delta$ probability,

$$\left|\tilde{d}_k - d_k\right| < \varepsilon_d \text{ , for all } k$$

Therefore, By Lemma 8, $|\tilde{s}(v_i, v_j) - s(v_i, v_j)| \le \varepsilon$ holds with at least $1-\delta$ probability. □