An Experimental Evaluation of SimRank-based Similarity Search Algorithms

[Experiments and Analyses]

Zhipeng Zhang[†] Yingxia Shao[†] Bin Cui[†] Ce Zhang[‡]
Key Lab of High Confidence Software Technologies (MOE), School of EECS, Peking University
[†]{zhangzhipeng, shao.yingxia, bin.cui}@pku.edu.cn
[‡]Department of Computer Science, ETH Zurich, ce.zhang@inf.ethz.ch

ABSTRACT

Given a graph, SimRank is one of the most popular measures of the similarity between two vertices. We focus on efficiently calculating SimRank, which has been studied intensively over the last decade. This has led to many algorithms that efficiently calculate or approximate SimRank being proposed by researchers. Despite these abundant research efforts, there is no systematic comparison of these algorithms. In this paper, we conduct a study to compare these algorithms to understand their pros and cons.

We first introduce a taxonomy for different algorithms that calculate SimRank and classify each algorithm into one of the following three classes, namely, iterative-, non-iterative-, and random walk-based method. We implement ten algorithms published from 2002 to 2015, and compare them using both synthetic and real-world graphs. To ensure the fairness of our study, our implementations use the same data structure and execution framework, and we try our best to optimize each of these algorithms. Our study reveals that none of these algorithms dominates the others: algorithms based on iterative method often have higher accuracy while algorithms based on random walk can be more scalable. One non-iterative algorithm has good effectiveness and efficiency on graphs with medium size. Thus, depending on the requirements of different applications, the optimal choice of algorithms differs. This paper provides an empirical guideline for making such choices.

1. INTRODUCTION

In many applications, it is important to measure the similarity between two vertices in a graph. Notable examples include collaborative filtering in recommendation systems [6, 8, 19, 1], link prediction for web searches [16, 14], web spam detection [3], and so on [2, 21]. Not surprisingly, many different similarity measures have been introduced by researchers over the last decade. Among these similarity measures, SimRank is one of the most popular.

Calculating SimRank can be computationally expensive because the similarity between two vertices not only depends on themselves, but also potentially on all other vertices in the same connected component—the intuition is that two nodes are similar if their neighbors are also similar. Therefore, there is a large body of research that focuses on optimizing and approximating SimRank. However, there are no systematic studies of existing algorithms, and it is difficult for developers to choose an algorithm for a given dataset and target accuracy. This motivates us to compare these algorithms empirically and provide such a guideline for developers.

Taxonomy of Existing Work. Existing algorithms of computing SimRank can be classified into three classes, namely, (1) iterative method, which iterates to the fix point, (2) non-iterative method, which solves a linear system (often with a low-rank approximation), and (3) random walk, which models the computation as random walks over the graph. These classes are logically equivalent: (1) previous work has established the equivalency between iterative method and random walk [7], and (2) we show in this work, theoretically, that non-iterative method is equivalent to an *approximate version* of random walk (Section 3). This connection provides insights on understanding the SimRank algorithms and explaining our empirical result. Figure 1 shows this taxonomy.

Summary of Methodology. To compare different algorithms fairly, we re-implement all of them with the same data structure and execution framework, and our implementations are at least as fast as the state-of-the-art work [18]. We compare these algorithms with multiple metrics including efficiency, effectiveness, robustness, and scalability. We also study the impact of different graph structures on the performance of these algorithms.

Summary of Results. We find that, across twelve real-world datasets, iterative method, if it can finish execution, has the highest accuracy; however, this method often suffers from scalability issues on large graphs. Non-iterative method algorithms based on low-rank matrix decomposition do not support top-k query well. Moreover, their accuracy is often lower due to the lossy low-rank approximation. On the other hand, Par-SR, a non-iterative algorithm that does not adopt the low-rank approximation but uses the sparse matrix techniques instead, has good efficiency and effectiveness. Algorithms based on random walk can often scale to large graphs; however, their performance and accuracy are often sensitive to the structure of the graphs. For graphs with a local sparse structure (Section 5.1.1), TopSim-based solutions perform better; while for graphs with a local dense structure, algorithms based on Monte Carlo sampling, like FP-SR and TSF, often perform better.

Summary of Contributions. In this work, we compared the existing SimRank algorithms and conducted experiments to demonstrate their differences. Our contribution is as follows:

1. We introduced a taxonomy of different algorithms for computing SimRank and classified each existing algorithm into

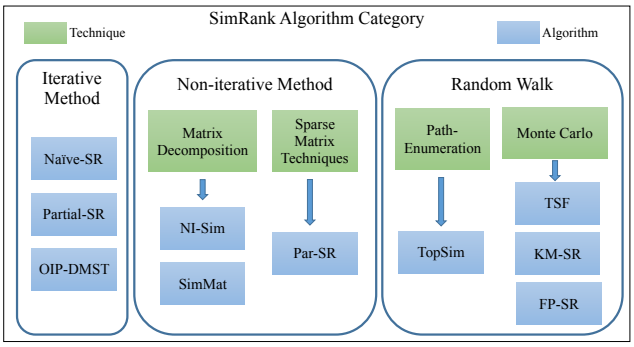


Figure 1: Taxonomy of SimRank algorithms.

one of the three categories.

2. We proved theoretically the relationship between non-iterative method and random walk and analyzed the loss of accuracy of non-iterative method.
3. We re-implemented all of the algorithms with the same framework and optimized each to give a fair comparison.
4. We empirically compared these algorithms on efficiency, effectiveness, robustness as well as scalability. Based on the result, we provided suggestions on how to choose an algorithm for a given dataset and target accuracy.

Overview. The rest of the paper is organized as follows: We introduce the preliminaries and the taxonomy of SimRank in Section 2 followed by the relationships of these categories in Section 3. In Section 4, we present different algorithms under the three categories and discuss their advantages and disadvantages. Experimental results are shown in Section 5 and related work is described in Section 6. Finally, we conclude the work in Section 7.

2. OVERVIEW OF SIMRANK

We first review the definition of SimRank and introduce the top-k query problem. We then describe the taxonomy of the different algorithms for computing SimRank. Table 1 summarizes some notation frequently used in this section.

2.1 Problem Definition

Given a graph $G(V, E)$, SimRank measures the similarity between two vertices using the graph structure. The intuition is that two vertices are similar if they are connected to neighbors that are also similar. Formally, the SimRank score between two vertices a and b is defined as follows:

$$S(a,b) = \begin{cases} 1 & \text{if } a = b, \\ \frac{c}{|I(a)||I(b)|} \sum_{i=1}^{|I(a)|} \sum_{j=1}^{|I(b)|} S(I_i(a), I_j(b)) & \text{if } a \neq b, \end{cases} \quad (1)$$

where c is a damping factor between 0 and 1 and $I(a)$ represents all the in-neighbors of vertex a. The SimRank score between a vertex and itself, i.e., $S(a,a)$, is always 1. If a vertex a has no in-neighbors, $\forall b \neq a, S(a,b) = 0$.

Let A be the adjacency matrix of the graph G, W the column-normalized matrix of A, and S the similarity matrix that we want to calculate. We can rewrite Equation 1 as

$$S = (c \cdot W'SW) \vee I, \quad (2)$$

where I is an identity matrix and W' is the transpose of W. The operator $\vee$ sets the diagonal elements of the left-hand side to the corresponding element of the right-hand side.

Symbols	Description
$G(V, E)$	Graph G with vertex set V and edge set E
$I(a)$	In-neighbors of vertex a
d	Average in-degree of a graph
n, m	Number of vertices and edges in a graph
$S(a, b)$	SimRank score between vertices a and b
W	Column-normalized matrix of the adjacency matrix of graph G
W'	Transpose matrix of W
I	Identity matrix
c	Damping factor
t, T	t^{th} step and iteration number
r	Rank

Table 1: Summary of notations.

For many applications, it is not necessary to compute the full matrix S, and many algorithms, as well as our study, focus on answering top-k queries [11, 10, 9, 4, 18, 5][1]:

DEFINITION 1. *(Top-k Query) Given a vertex u as the query vertex and a constant number k, the task of answering the top-k query returns k vertices that are most similar to u together with the corresponding scores.*

2.2 Taxonomy of SimRank Algorithms

Existing algorithms can be organized with the following taxonomy, namely (1) iterative method, (2) non-iterative method, and (3) random walk. We now describe these categories and leave the detailed discussion and description of each algorithm to Section 4.

Iterative Method. Iterative method answers the SimRank query in an iterative way following

$$S_t = (c \cdot W'S_{t-1}W) \vee I, \quad (3)$$

where t is the number of iterations. Algorithms in this category [7, 15, 22] follows Equation 3 by iterating to the fix point. To make each iteration faster, different algorithms exploit different optimizations to reduce the redundant computation for each iteration.

Non-iterative Method. The operation $\vee$ in iterative method makes it hard to apply the techniques in linear algebra, such as the low-rank approximation of a matrix, to calculate SimRank efficiently. As a result, non-iterative method [5, 11, 23] uses a correction matrix $(1-c)I$ to replace the operation $\vee$ as follows:

$$S = c \cdot W'SW + (1-c)I. \quad (4)$$

Because Equation 4 is a linear system, we can take advantage of decades of studies on solving linear systems to design efficient approximation algorithms. However, because $(1-c)I$ is only an approximation of $\vee$, it sometimes incurs a loss of accuracy in computing the true SimRank score, as we discuss in Section 3.2.

Random Walk. It is also possible to interpret SimRank with coupled random walks. The SimRank score $S(u,v)$ is equal to the expected-f meeting distance [7], which can be computed as:

$$S(u,v) = \sum_{t=0}^{\infty} c^t \cdot \sum_{w \in V} p_{ft}(u,v,w), \quad (5)$$

where c is the damping factor, V is the set of vertices, and $p_{ft}(u,v,w)$ is the probability of a pair of random walks L_u, L_v satisfying the following conditions: (1) L_u and L_v start from a and b, respectively; (2) The lengths of L_u and L_v are t; and (3) L_u and L_v meet at w and it is the first time that they meet.

[1]There are other tasks related to SimRank, e.g., all-pairs SimRank query [7, 15, 22], single-pair SimRank query [11, 5, 9, 4, 18, 13], range query [5], and similarity join [23, 24, 20]. In this paper, we focus on the top-k query because of its popularity.

Based on Equation 5, one can calculate the expected-f meeting distance for the SimRank computation, which treats random walks as building blocks. To calculate the expected-f meeting distance, different algorithms exploit different techniques, like enumerating all the possible coupled random walks [10] or sampling some of them for estimation [9, 4, 18] via efficient sampling techniques.

3. DISCUSSION OF TAXONOMY

The equivalency between iterative method and random walk was known in the original SimRank paper [7]. We now establish the relationship between non-iterative method and random walk.

3.1 Theoretical Analysis

From Equation 4 we have

$$S(u,v) = (1-c)\sum_{t=0}^{\infty} c^t \cdot e'_u (W')^t \cdot W^t e_v, \tag{6}$$

where e_u is an $n \times 1$ all-zero vector except that the uth element is 1, and similarly for e_v. We further expand Equation 6 as

$$S(u,v) = (1-c)\sum_{t=0}^{\infty} c^t \cdot \sum_{i=1}^{n} ((W^t e_u)')_i \cdot (W^t e_v)_i, \tag{7}$$

where $((W^t e_u)')_i$ and $(W^t e_v)_i$ denote the i^{th} elements of vectors $(W^t e_u)'$ and $W^t e_v$, respectively.

We can think of W as the transition matrix and e_u and e_v as the probability vectors of two random walks L_u and L_v starting from vertices u and v, respectively. Thus, $(W^t e_u)'_i$ represents the probability of random walk L_u ending at vertex i at the t^{th} step. A similar analysis holds for $(W^t e_v)_i$. As a consequence, $((W^t e_u)')_i \cdot (W^t e_v)_i$ is exactly the probability of two random walks meeting at vertex i at the t^{th} step. Then $\sum_{i=1}^{n}((W^t e_u)')_i \cdot (W^t e_v)_i$ is the probability of u and v meeting at all vertices at the t^{th} step.

Comparing with Equation 5, it is clear that the value of Equation 8 is exactly the expected-f meeting distance between u and v without the first-meeting restriction:

$$\sum_{t=0}^{\infty} c^t \cdot \sum_{i=1}^{n} ((W^t e_u)')_i \cdot (W^t e_v)_i \tag{8}$$

Comparing Equations 7 and 8, it can be inferred that the result of non-iterative method is scaled (by $(1-c)$) to that of random walk without guaranteeing first-meeting.

3.2 Implication on Accuracy

We just showed that non-iterative method is equivalent to random walk without the first-meeting constraint. We now analyze the impact of this relaxation on accuracy of non-iterative method.

THEOREM 1. *Assume that the accurate SimRank score between vertices u and v is $S(u,v)$, and that the SimRank score computed without the first-meeting constraint is $S'(u,v)$, then we have:*

$$1 \le \frac{S'(u,v)}{S(u,v)} \le \frac{1}{1-c}.$$

PROOF. *Let $f(u,v,w)$ denote the contribution of two random walks starting from u and v, respectively, and meeting at vertex w and only at vertex w. Then according to Equation 5, we have:*

$$f(u,v,w) = \sum_{t=0}^{\infty} c^t \cdot p_{ft}(u,v,w), \tag{9}$$

where c is the damping factor of SimRank and $p_{ft}(u,v,w)$ is defined in Equation 5.

Correspondingly, let $f'(u,v,w)$ represent the first-meeting contribution plus those of later meetings from vertex w, i.e.,

$$f'(u,v,w) = \sum_{t=0}^{\infty} \Bigg(c^t \cdot p_{ft}(a,b,w) + \sum_{l=1}^{\infty} c^{l+t} \cdot p_{ft}(a,b,w) \cdot p_{l:(w,w)\to(x,x)} \Bigg), \tag{10}$$

where $p_{l:(w,w)\to(x,x)}$ represents the probability of two random walks that both start from vertex w meeting at any vertex x at the l^{th} step.

Combining Equations 9 and 10, we have:

$$f'(u,v,w) = f(u,v,w) \cdot \left(1 + \sum_{l=1}^{\infty} c^l p_{l:(w,w)\to(x,x)}\right),$$

i.e.,

$$\frac{f'(u,v,w)}{f(u,v,w)} = 1 + \sum_{l=1}^{\infty} c^l p_{l:(w,w)\to(x,x)}. \tag{11}$$

Considering that $p_{l:(w,w)\to(x,x)} \le 1$, we have:

$$\frac{f'(u,v,w)}{f(u,v,w)} \le 1 + \sum_{l=1}^{\infty} c^l = \frac{1}{1-c}. \tag{12}$$

Furthermore, we have the following inequality straightforwardly following from Equation 11:

$$\frac{f'(u,v,w)}{f(u,v,w)} \ge 1. \tag{13}$$

According to the definition of $f(u,v,w)$ and $f'(u,v,w)$, we have:

$$\frac{S'(u,v)}{S(u,v)} = \frac{\sum_{w=1}^{n} f'(u,v,w)}{\sum_{w=1}^{n} f(u,v,w)}. \tag{14}$$

Combining Equations 12, 13, and 14, we have:

$$1 \le \frac{S'(u,v)}{S(u,v)} \le \frac{1}{1-c}.$$

□

4. ALGORITHM ANALYSES

In this section, we elaborate on the state-of-the-art algorithms that compute SimRank according to the taxonomy in Figure 1.

4.1 Iterative Method-based Algorithms

The key for algorithms based on iterative method is to reduce the repetitive computation during the iterations. The accuracy of iterative algorithms depends on the number of iterations. Theoretically, the result is accurate asymptotically.

4.1.1 Naive-SR

Naive-SR [7] is a naive iterative solution following Equation 3. It computes the similarity between all vertex pairs with $O(Td^2n^2)$ time and $O(n^2)$ space cost, where d is the average in-degree. To speed up, Naive-SR also provides some pruning skills by setting the SimRank score between two vertices far apart to be zero.

4.1.2 Partial-SR

Partial-SR [15] is an improved version of Naive-SR. Partial-SR stores the intermediate results, namely partial sums, to avoid some redundant computation.

Redundant computations exist in each iteration. As shown in

$$S(a,b) = \frac{1}{I(a)\cdot I(b)} \sum_{x\in I(a)} \underbrace{\sum_{y\in I(b)} S(x,y)}_{\text{Partial}_{I(b)}(x)}, \tag{15}$$

the part denoted by $\text{Partial}_{I(b)}(x)$ is computed repeatedly whenever $S(a,b)$ is computed, where vertex x is an in-neighbor of vertex a. For example, as shown in Figure 2, vertex a and vertex b have a common in-neighbor c. To get $S(a,p)$ and $S(b,p)$ for a given vertex p, $\text{Partial}_{I(p)}(c)$ is computed twice.

Partial-SR stores all the partial sums that have been computed in an iteration to accelerate the computation. Therefore, Partial-SR improves the computation cost of SimRank from $O(Td^2n^2)$ time to $O(Tdn^2)$ with additional $O(n)$ space cost.

Discussion. Two optimizations are further applied for a speed-up. First, only the upper triangular of the similarity matrix S is computed due to its symmetry. This reduces the processing time by half. Second, not all partial sums are computed in each iteration. When computing $S(a,*)$, we compute only part of the partial sums, i.e., $\text{Partial}_{I(a)}(y)$ with y belonging to $\{I(b)|b<a\}$. This also helps reduce the processing time.

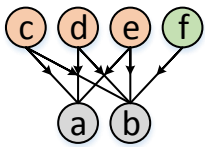


Figure 2: An example graph for Partial-SR and OIP-DMST.

4.1.3 OIP-DMST

OIP-DMST [22] eliminates duplicate computation in a finer granularity than Partial-SR. The basic idea is to change the computation order of partial sums and use partial sums of a vertex to compute the others if these two vertices share many common in-neighbors. Next, we discuss the repetitive computation in partial sums and introduce how to determine the computation order.

Duplicate summations exist when computing partial sums. For example, in Figure 2, vertex a and vertex b have three common in-neighbors, $\{c,d,e\}$. When computing $\text{Partial}_{I(a)}(p)$ and $\text{Partial}_{I(b)}(p)$ with respect to a given vertex p, Partial-SR computes them separately. However, OIP-DMST uses $\text{Partial}_{I(b)}(p)$ to compute $\text{Partial}_{I(a)}(p)$ as $\text{Partial}_{I(a)}(p) = \text{Partial}_{I(b)}(p) + S(f,p)$, which eliminates two additions.

A cost graph GS is constructed to determine the computation order of partial sums. Each vertex in GS denotes a partial sum and each edge represents the cost of using one partial sum to compute the other. Then the minimum spanning tree of GS describes the computation order of partial sums in OIP-DMST. As a result, OIP-DMST needs $O(dn^2)$ time to generate the cost graph and $O(n^2)$ time to find the minimum spanning tree.

Discussion. Like Partial-SR, only the upper triangular similarity matrix is computed in OIP-DMST. However, this cannot accelerate the computation as much as it does for Partial-SR. This is because the computations of partial sums and SimRank scores in OIP-DMST are strictly separated. All the partial sums for vertex u have to be computed before getting any SimRank score with respect to u. Furthermore, the partial sums are interrelated since some of them are needed to compute others. As a result, all the partial sums have to be computed.

4.1.4 Summary

The main optimization of iterative method lies in reusing intermediate results. Partial-SR and OIP-DMST calculate SimRank faster compared to Naive-SR by storing the partial sums. However, there still exist two shortcomings in algorithms based on iterative method. One is that the time complexity of these algorithms is at least quadratic and they cannot process large-scale graphs efficiently. The other is that these algorithms cannot get part of the SimRank scores without computing the whole similarity matrix. Thus, when handling a single-pair query or a top-k query, iterative algorithms have to compute all SimRank scores for indexing.

4.2 Non-Iterative Method-based Algorithms

The basic idea of non-iterative method-based algorithms is to use linear algebra techniques, such as the low-rank approximation of a matrix and sparse matrix multiplication, to solve Equation 4.

4.2.1 NI-Sim

NI-Sim [11] was the first non-iterative algorithm to compute SimRank. It uses two operators, the Kronecker product and the vectorization operator, to extract the similarity matrix and further, it uses the low-rank approximation to approximate SimRank.

Preprocessing phase. Based on the Kronecker product and the vectorization operator, Equation 4 can be rewritten as:

$$\text{vec}(S) = (1-c)(I - c(W'\otimes W'))^{-1}\,\text{vec}(I).$$

Moreover, NI-Sim performs singular value decomposition (SVD) on matrix W' to get an approximation. Let $W' = U\Sigma V$, according to the Woodbury formula,[2] then S can be computed as

$$\text{vec}(S) = (1-c)(\text{vec}(I) + cK_u\Lambda V_r),$$

where $K_u = U\otimes U$, $\Lambda = (K_\Sigma^{-1} - cK_{vu})^{-1}$, and $V_r = K_v\,\text{vec}(I)$. $K_{vu} = K_vK_u$. As a result, matrices K_u, Λ, and V_r are stored as the index for querying.

Querying phase. In this phase, NI-Sim uses the matrices stored in the index to answer the query as

$$S(i,j) = (1-c)(I(i,j) + cV_lV_r),$$

where V_l can be computed as $V_l = K_u((i-1)n + j,:)\Lambda$.

Discussion. NI-Sim computes a single-pair SimRank query in $O(r^4)$ time with $O(n^2r^2+r^4)$ space. The accuracy loss of NI-Sim comes from the low-rank approximation of a matrix and from ignoring the first-meeting constraint. However, there are three shortcomings in this algorithm. First, the expensive index space requirement and the high complexity of SVD have restricted the application scope of NI-Sim to small graphs. Second, SVD cannot preserve the relative order of SimRank scores because it focuses on minimizing the absolute error while SimRank scores are close to each other empirically. Third, there is not an error bound for NI-Sim for common cases.

4.2.2 SimMat

SimMat [5] is another non-iterative algorithm that computes SimRank. It does so based on the Sylvester equation[3] and low-rank approximation of a matrix.

Preprocessing phase. First, SimMat transforms Equation 4 into the form of the Sylvester equation by multiplying by W^{-1} on the left as:

$$\frac{1}{c}(W')^{-1}S - SW = \frac{1-c}{c}(W')^{-1}. \tag{16}$$

Then an eigen decomposition is performed on matrix W:

[2] http://mathworld.wolfram.com/WoodburyFormula.html
[3] https://en.wikipedia.org/wiki/Sylvester_equation

$$W = PDP^{-1},$$

where D is the diagonal matrix with eigenvalues as the diagonal elements and P is composed of the corresponding normalized eigenvectors. With some equivalence transformations, the result matrix S is computed as

$$S = (P')^{-1}XP^{-1},$$

where X can be computed via solving the following equation:

$$\frac{1}{c}D^{-1}X - XD = \frac{1-c}{c}D^{-1}P'P. \tag{17}$$

To improve the efficiency further, SimMat$_e$ and SimMat$_s$ are proposed via low-rank approximation techniques. For example, SimMat$_s$ performs SVD on matrix P^{-1} and stores the intermediate matrix as the index.

Querying phase. SimMat uses the index to compute the SimRank scores. Furthermore, SimMat uses efficient pruning through the Cauchy–Schwarz inequality for answering the top-k query.

Discussion. Likes NI-Sim, SimMat loses accuracy because of the low-rank approximation and not guaranteeing first-meeting.

SimMat can be applied to limited real-world graphs. This is because when transforming the initial SimRank formula into the form of the Sylvester equation in Equation 16 and computing the inverse matrix of D following Equation 17, matrices W and D are required to be reversible, which is hard to satisfy in real-world graphs.

4.2.3 Par-SR

Par-SR [23] interprets SimRank as a linearized formula following Equation 6. The most important part of Par-SR is that it answers a top-k query without computing the whole similarity matrix. SimRank scores with respect to a given vertex u are computed as:

$$[S_T]_{*,u} = (1-c)\sum_{t=0}^{T}(W')^t \cdot W^t e_u. \tag{18}$$

To reduce further the computational cost of Equation 18, Par-SR uses the seed germination model [23] and successfully reduces the number of duplicate computations in each iteration, and thus achieves $O(Tm)$ time and $O(Tn)$ space for answering a top-k query. Furthermore, PrunPar-SR can reduce unnecessary edge accesses. It uses sparse matrix multiplication technique to reduce the number of accessing irrelevant vertices and achieves $O(\min\{Tm, d^{2T}\})$ time efficiency, where d is the average in-degree. In the rest of the paper, Par-SR means PrunPar-SR for simplicity.

Discussion. Par-SR can be viewed as a special kind of random walk-based algorithms according to the analysis in Section 3. It uses matrix–vector multiplication to simulate the random walks and naturally groups the random walks with the same end vertices because a vector is inherently a hash map.

4.2.4 Summary

Non-iterative method uses $(1-c)I$ to approximate the diagonal correction matrix and breaks the holistic nature of the SimRank computation. However, SimMat, NI-Sim, and Par-SR all suffer from not guaranteeing first-meeting as a consequence. Moreover, different non-iterative algorithms lose accuracy from their own optimization techniques, such as the low-rank approximation of a matrix and the limited number of iterations.

4.3 Random Walk-based Algorithms

Algorithms based on random walk interpret the similarity as the expected-f meeting distance of two different vertices. These algorithms fall into two categories: (1) *Monte Carlo*, which samples some coupled random walks to estimate the SimRank scores by a Monte Carlo simulation, like KM-SR [9], FP-SR [4], and TSF [18], and (2) *path enumeration*, which enumerates all the possible coupled random walks, like TopSim [10].

4.3.1 KM-SR

KM-SR [9] is based on two observations. First, SimRank scores can be approximated by a linear form in Equation 6. Second, $W^t e_u$ can be thought of as the probability vector of vertex u ending at each vertex after randomly walking t steps. Furthermore, the probability vector can be computed via

$$W^t e_u = E[e_u(t)], \tag{19}$$

by a Monte Carlo simulation, where $E[e_u(t)]$ is an $n \times 1$ vector with each element representing the expected probability of u ending at the corresponding vertex at the t^{th} step.

Combining these two techniques, KM-SR handles the top-k query by a two-phase process.

Preprocessing phase. The preprocessing phase of KM-SR consists of two parts, *candidate selection* and *upper bound computation*. The intuition of *candidate selection* is that the vertices frequently reachable from vertex u are probably the points where u meets other vertices. Following this idea, KM-SR uses Monte Carlo simulation to select frequently reachable vertices for each vertex u as follows.

For each vertex u, repeat the following procedure P times:

1. Sample a random walk W_0 with length T from u, denoted by $\{w_{01}, w_{02}, \ldots, w_{0T}\}$.
2. Sample Q random walks from u with length T, namely, $W_1, \ldots, W_Q$.
3. For each vertex v in W_0, KM-SR adds it to the candidate list if there are at least two random walks ending at v at the corresponding step.

KM-SR stores all the frequently reachable vertices for each vertex as the index. The space and time needed are both $O(n)$. Furthermore, in the *upper bound computation* process, KM-SR stores the norm of the probability vector for each vertex to filter the SimRank scores when querying.

Querying phase. Given a top-k query with respect to vertex u, KM-SR first enumerates the vertices that have common frequently reachable vertices with u via the index. Then it computes the similarity between u and the enumerated vertices via Equations 6 and 19. Finally, the top-k similar vertices are picked as the answer.

Discussion. The *candidate selection* in the preprocessing phase is too harsh and there are few vertices in the candidate list. Suppose that the random walk starts from u and ends at vertex w_{0i} with probability p_i, then the probability of at least two random walks ending at w_{0i} can be computed as:

$$1 - (1-p_i)^Q - Q \cdot p_i \cdot (1-p_i)^{Q-1}.$$

Since there are T end vertices in random walk W_0 and the process is repeated P times, the expected number of candidates for vertex u is:

$$P\sum_{i=1}^{T}(1 - (1-p_i)^Q - Q \cdot p_i \cdot (1-p_i)^{Q-1}). \tag{20}$$

Furthermore, the probability of random walk W_0 ending at a certain vertex, p_i, is exponential to the step of the random walk, which can be approximated by the average in-degree of vertices on the path of random walk W_0 as

$$p_i \approx \left(\frac{1}{d}\right)^i,$$

where d is the average in-degree. When the number of steps increases, p_i converges to zero. As a result, the value of Equation 20 is small. With the parameters set in the original work, i.e., $P = 10$, $T = 11$, and $Q = 5$, only a few vertices are preserved, mostly less than ten for each vertex.

4.3.2 FP-SR

FP-SR [4] uses the FPG data structure to store compactly the random walks sampled via Monte Carlo simulation as well as the distance at which each two random walks meet for the first time. In the following, the FPG data structure is introduced and the detailed operations during the preprocessing and querying phases are provided.

FPG. An FPG organizes in a compact way one random walk with length T for each vertex together with the distance where each two random walks meet for the first time. FP-SR assumes that once two random walks meet, they walk together. A good property of FPG is that if the similarity of vertex pair (u, v) is not zero, then these two vertices must be in the same branch of an FPG, which significantly improves the search efficiency.

Preprocessing phase. FP-SR builds the FPGs as the index. When building an FPG, FP-SR generates one uniform edge e_i for each vertex independently and extends random walks that have the same last vertex i with edge e_i. This not only improves the indexing efficiency but also ensures the independence until the first meeting. The space needed is $O(nR_g)$.

Querying phase. Given a top-k query of vertex u, FP-SR needs to scan the branches in the FPGs that contain u and calculates SimRank scores with the stored distance where u meets others. The graph connectivity of FPGs helps to filter out vertices that rarely meet u.

Discussion. FP-SR answers a top-k query efficiently because it does not need to calculate the distance where the query vertex meets others when querying. The distance is also stored in FPGs. Moreover, the graph connectivity of a FPG helps to prune the vertices that never meet the query vertex.

FP-SR has two shortcomings. One is that it is hard for FP-SR to scale to large graphs for high accuracy since one FPG yields only one meeting chance for a vertex pair. If high accuracy is required, the space cost is expensive. The other one is that two random walks are dependent in a top-k query because FP-SR assumes that two random walks walk together after their first meeting. Although this reduces the index space, the error bound is hard to give.

4.3.3 TSF

TSF [18] employs Monte Carlo sampling techniques to estimate SimRank scores following random walk. It uses the novel one-way-graph data structure to store the random walks efficiently. In the following, we first describe the one-way-graph data structure and then introduce the preprocessing and querying phases of TSF.

One-way graph. A one-way graph is a novel compact data structure that contains a random walk for each vertex. It is a graph satisfying that each vertex has at most one outgoing edge, which naturally meets the need of random walks that each vertex randomly chooses an outgoing edge.

Preprocessing phase. TSF samples R_g one-way graphs as the index, where each one-way graph contains exactly n vertices and no more than n edges. The time and space needed are $O(nR_g)$.

Querying phase. When answering a top-k query with respect to vertex u, TSF repeats the following process for each indexed one-way graph: (1) TSF samples R_q random walks starting from u and extracts the meeting vertices and (2) for each meeting vertex v, it expands on that one-way graph to find vertex u' that meets u at vertex v. The SimRank scores are estimated from these meetings. The connectivity of one-way graphs also helps to filter out vertices that are rarely reached by vertex u.

Discussion. TSF uses the two-phase sampling technique to cope with a SimRank query. Each one-way graph is reused R_q times, thus, it can achieve higher accuracy with limited index space by re-sampling more random walks when querying. The graph connectivity of one-way graphs can help to filter out unnecessary vertices when querying top-k as FPGs do in FP-SR. However, the reuse of one-way graphs also leads to a dependence among the random walks for a top-k query.

4.3.4 TopSim

TopSim [10] avoids the global access of the graph because it computes SimRank via enumerating the random walks within a short distance.

For a top-k query of vertex v, TopSim repeats the following procedure T times: (1) It enumerates all the vertices that reach v at the $t^{th} \in [1, T]$ step and saves these vertices in the meeting point list U_t and (2) for each meeting point u in U_t, TopSim enumerates the vertices v' that are reachable from u at the t^{th} step. Intuitively, v' and v meet at vertex u in exactly t steps, which contributes to the similarity between vertex v and v'. After T steps, all the SimRank scores with respect to vertex v are computed.

During the traversal of similarity paths, there exist some repeated searches on the paths and TopSim-SM is proposed to merge such similarity paths. For example, path $\{a \to b \to c \to d\}$ and path $\{a \to e \to c \to d\}$ can be combined as $\{a \to b/e \to c \to d\}$. As a result, one traversal on edge $\{c \to d\}$ can be eliminated.

Furthermore, two heuristic solutions are provided for higher efficiency by sacrificing effectiveness, i.e., TrunTopSim and PrioTopSim. TrunTopSim sets two thresholds, h and η, to truncate the high-degree vertices in the set of meeting points U_t when expanding random walks. PrioTopSim applies a priority pool to select the top H meeting point in U_t for answering a top-k query in a certain time.

Discussion. TopSim-SM handles a top-k query in $O(d^{2T})$ time, where d is the average in-degree and T is the length of the random walk. As a result, TopSim-based algorithms can process large graphs, since the efficiency does not rely on the scale of graphs. However, they are sensitive to the graph structure. This is because TopSim-based solutions enumerate all or most of the random walks within a short distance. As a result, it runs fast on sparse graphs but slowly on dense graphs. TopSim-based algorithms loses accuracy because the length of random walks is short.

4.3.5 Summary

The key to Monte Carlo sampling-based algorithms, like FP-SR, KM-SR and TSF, is to sample efficiently these coupled random walks when querying or to store the random walks efficiently while preprocessing. They lose efficiency due to the limited samples used for estimating the expected-f meeting distance. Algorithms based on path enumeration, like TopSim, focus on how to reduce the number of random walks efficiently. The accuracy loss comes from the short length of the random walks.

5. EXPERIMENTAL ANALYSES

We analyze the pros and cons of each algorithm with extensive experiments. In particular, each algorithm is evaluated via effectiveness, efficiency, robustness, and scalability (defined in Section 5.1.2). In the following, we first introduce the experimental environment, then both the effectiveness and efficiency are compared according to the three categories; finally, the robustness and the scalability of these algorithms are reported.

5.1 Experimental Environment

All experiments are conducted on a machine powered by two Xeon(R) E5530@2.40GHz CPUs and 96GB memory, under Ubuntu 14.04.1 LTS. All the algorithms are implemented in C++ and compiled by g++ 4.8.4 with the -O3 option. The low-rank approximation of a matrix is computed via Armadillo [17], a C++ linear algebra library.

5.1.1 Datasets

Experiments are conducted on both synthetic and real-world datasets. The real-world datasets consist of small, medium, and large datasets as shown in Table 2. The synthetic datasets are generated via networkX[4].

	Graph	V	E	d
Tuning	BA1	1.0k	5.0k	5.0
	BA2	1.0k	10.0k	10.0
	BA3	5.0k	50.0k	10.0
	ER1	1.0k	5.0k	5.0
	ER2	5.0k	50.0k	10.0
Small	AirLine (AL)	0.5k	72.0k	157.8
	ODLIS.NET (OD)	2.9k	18.2k	6.3
	CA-GRQC (CG)	5.2k	29.0k	5.5
	P2p-Gnutella08 (PG)	6.3k	20.8k	3.3
	WikiVote (WV)	7.1k	103.7k	14.6
Medium	webNotreDame (ND)	0.3M	1.5M	4.6
	webBerkStan (BS)	0.7M	7.6M	11.1
	webGoogle (WG)	0.9M	5.1M	5.8
Large	LiveJournal (LJ)	4.8M	69.0M	14.2
	wikipedia-en (WP)	25.9M	601.0M	21.1
	it-2004 (IT)	41.3M	1150.7M	27.9
	twitter (TW)	41.7M	1468.4M	35.3

Table 2: Dataset statistics.

In this paper, we further classify these graphs into two types, local sparse and local dense ones. If a small proportion of the vertices in a graph take up most of the edges, then this graph is a local dense one. For example, WV, LJ, and TW are local dense graphs and the rest of the real-world datasets in Table 2 are local sparse graphs. Take WV as an example. More than 60% of the vertices have zero in-degree in WV, and as a result, the rest of the vertices form a dense subgraph. For simplicity, we use "dense" and "sparse" to represent "local dense" and "local sparse" in the rest of the paper.

5.1.2 Evaluation Metrics

Effectiveness, efficiency, robustness, and scalability are used to evaluate each algorithm.

Effectiveness. For effectiveness, Precision, normalized discounted cumulative gain (NDCG), and average difference (AvgDiff) are used to evaluate the quality of the returned top-k similar vertices in a top-k query. In the following, these criteria are introduced.

Suppose that the set of top-k similar vertices with respect to u returned by the accurate algorithm and the approximate algorithm are $\{TopK\}$ and $\{TopK'\}$, respectively. The corresponding scores are $S(u,v)$ and $S'(u,v)$. Precision@k is defined as

$$Precision@k = \frac{\#|\{TopK\} \cap \{TopK'\}|}{k},$$

where $\#|\{TopK\} \cap \{TopK'\}|$ denotes the number of elements in the set $\{TopK\} \cap \{TopK'\}$. NDCG@k is

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

where S_i is the exact SimRank score of vertex u at rank i in the returned top-k vertices and Z_k is a normalization factor which ensures that the NDCG@k of an accurate solution is 1. AvgDiff@k is defined as

$$AvgDiff@k = \frac{\sum_{v \in \{TopK\} \cap \{TopK'\}} |S(u,v) - S'(u,v)|}{\#|\{TopK\} \cap \{TopK'\}|},$$

where $|S(u,v) - S'(u,v)|$ denotes the absolute difference and $\#|\{TopK\} \cap \{TopK'\}|$ represents the number of elements in the intersection of the two sets.

Algorithms have good effectiveness if they have high Precision and NDCG and low AvgDiff. We set k as 50 for these criteria following the analysis in Section 5.2.2. For simplicity, Precision, NDCG, and AvgDiff are used to represent Precision@50, NDCG@50 and AvgDiff@50 in the rest of the paper, respectively.

Efficiency. Efficiency describes the speed of answering SimRank queries. In particular, the preprocessing time and query time for each algorithm are reported.

Robustness. Top-k queries for an algorithm with respect to vertices with different local structures have different performance. Robustness is used to describe how these queries differ in terms of efficiency. If an algorithm answers top-k queries on all vertices with the same time cost, this algorithm is the most robust.

Scalability. For scalability, graphs with a different number of vertices are used to demonstrate how the efficiency of these algorithms change when the graph scale increases.

5.1.3 Parameter Setting

Of the ten algorithms we studied, there are five algorithms (NI-Sim, SimMat, KM-SR, FP-SR and TSF) having hyperparameters. Different settings of the hyperparameters, as we will show in Section 5.2, often lead to different effectiveness and efficiency. For fairness, we conduct careful parameter tuning to choose the most appropriate parameters for each algorithm following the state-of-the-arts [18, 11]: varying each parameter on several small graphs and use it for large graphs. If the original paper hardcoded the value of a hyperparameter with justification, we follow their settings. For example, for TopSim, we set $T = 3$ [10]. For Monte Carlo sampling-based algorithms and NI-Sim, graphs with different local structures and different scales are used for tuning according to the analyses of algorithms in Section 4. Based on the tunning results in Section 5.2.1, we use the parameters for each algorithms shown in Table 3. For hyperparameters that are used to define the objective, we set them as $c = 0.8$ and $T = 10$.

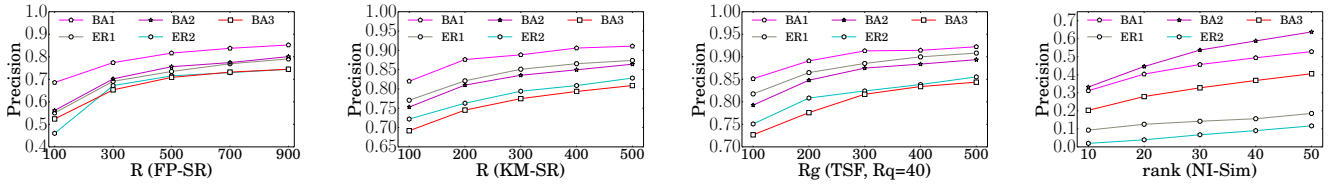
TSF	FP-SR	KM-SR	NI-Sim
$R_g = 300, R_q = 40$	$R = 500$	$R = 300$	$r = 30$

Table 3: Parameter values used in our study.

5.1.4 Experiment Clarification

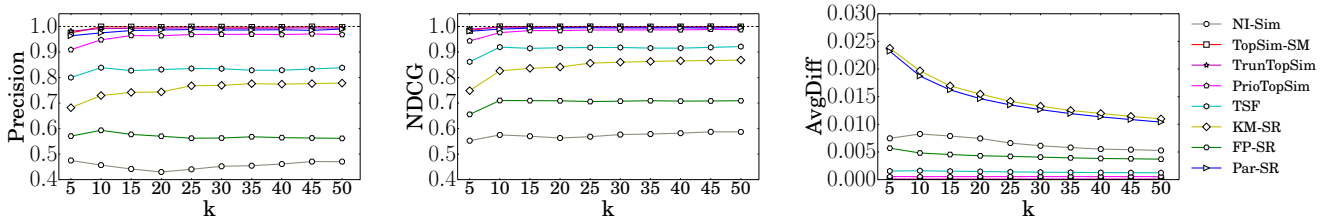
Some experiments could not be carried out due to the high time or space cost. The corresponding reasons for those experiments

[4] http://networkx.github.io/



(a) FP-SR (b) KM-SR (c) TSF (d) NI-Sim

Figure 3: Precision of FP-SR, KM-SR, TSF and NI-Sim with various parameter settings on five datasets.



(a) Precision (b) NDCG (c) AvgDiff

Figure 4: Effectiveness of different algorithms with various k for top-k queries on WikiVote.

Experiments not conducted	Reasons
Efficiency of iterative algorithms on medium and large datasets	Out of memory
Robustness of iterative algorithms	Robust by definition
Scalability of iterative algorithms	Out of memory
Effectiveness on medium and large datasets	Cannot be computed because iterative algorithms, which are used for computing effectiveness, run out of memory
NI-Sim on medium and large datasets	Out of memory
Robustness of NI-Sim	Robust by definition
Scalability of NI-Sim	Out of memory

Table 4: Experiments we omit in our study and the reasons.

not being conducted is presented in Table 4. For example, the robustness of iterative algorithms was not checked because they are robust by definition; the effectiveness on medium and large datasets was not evaluated since the accurate SimRank algorithms, which are used to compute effectiveness, ran out of memory, etc.

Furthermore, the reported query time and effectiveness of each algorithm on each dataset are estimated from 100 queries via stratified sampling according to the vertex in-degree if there is no specific clarifications.

5.2 Parameter Sensitivity

To ensure the fairness of our comparison among different algorithms, we present the sensitivity analyses of the parameters used in our experiment.

5.2.1 *Sensitivity of Parameters in Algorithms*

As discussed in Section 5.1.3, we use several graphs with different local structures and scales to tune FP-SR, KM-SR, TSF and NI-Sim. The graphs are shown in Table 2. In particular, ER1 and ER2 are generated based on the Erdos–Renyi model, in which all the pairs of vertices are connected randomly with a given probability. BA1, BA2, and BA3 are based on the Barabasi–Albert model, which has a similar characteristic to real-world social networks. The parameter sensitivity of these algorithms is shown in Figure 3. Only the result of Precision is presented but it is similar for NDCG. Note that there are two parameters for TSF (R_g and R_q) and we set R_q as 40 and vary R_g following the original paper [18].

Figure 3 shows the parameter sensitivity of FP-SR, KM-SR, TSF, and NI-Sim on five different graphs. Not surprisingly, all algorithms have higher Precision when the sampling number or the rank of NI-Sim increases. We choose the most appropriate parameters for each algorithm by a tradeoff between effectiveness and efficiency. For example, R for FP-SR is set as 500 since the Precision of FP-SR increases more slowly after 500. Similarly for KM-SR and TSF in Figure 3(b) and 3(c), we set R_g for TSF and R for KM-SR to 300. For NI-Sim, the rank is set to 30. It can be inferred that Precision increases more slowly after 30. Another reason not to choose a higher rank is that NI-Sim answers a top-k query in $O(nr^4)$ time. A higher rank incurs a higher efficiency loss. The resulting parameters from the experiments are shown in Table 3.

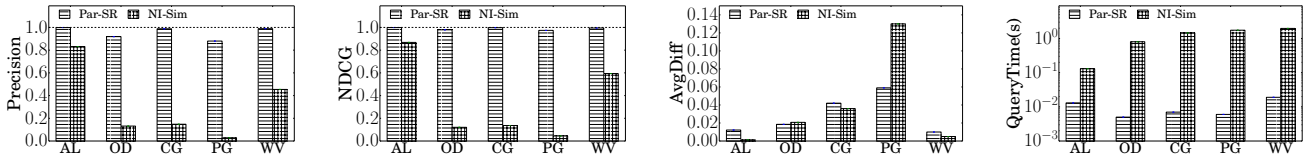
We find that, varying each hyperparameter has significant impact on Precision for all algorithms. Therefore, for all algorithms with hyperparameters, we recommend the users to tune them on small graphs with similar local structures. In the rest of the paper, we use the parameters tuned here by graphs with different local structures, which can serve as a common case for comparison.

5.2.2 *Sensitivity of k in Top-k Queries*

Previous works [10, 18, 23, 11] use different k values in metrics like Precision@k, NDCG@k, and AvgDiff@k to evaluate the effectivenss of a SimRank algorithm. For fairness, we design an experiment on two real-world datasets, WV and PG, to demonstrate how these metrics in the different algorithms that compute SimRank would change when k varies. We vary k from 5 to 50 in steps of 5. The result for WV is shown in Figure 4 while that for PG is not presented because it is similar.

Figure 4 indicates that the relative order of Precision, NDCG, and AvgDiff of each algorithm is well preserved as the parameter k increases. A special phenomenon is that the AvgDiff of Par-SR and KM-SR has a descending trend when k increases. This is because both Par-SR and KM-SR ignore the first-meeting constraint. In the following, we consider Par-SR in the explanation.

Given a top-k query with respect to vertex u, suppose that the set of top-k similar vertices returned by the accurate algorithm and Par-SR are $\{TopK\}$ and $\{TopK'\}$, respectively. The score re-



(a) Precision (b) NDCG (c) AvgDiff (d) Query Time

Figure 5: Results for non-iterative algorithms. Precision, NDCG, and AvgDiff describe the effectiveness while query time indicates the efficiency. The dotted lines represent the exact solutions. The AvgDiff of exact solutions is 0.

turned by the accurate solution is $S(u, v)$. Assume the accurate score computed without guaranteeing first-meeting is $S'(u, v)$, then the score returned by Par-SR can be represented as $(1 - c)S'(u, v)$ according to the discussion in Section 3.

With the above notations, AvgDiff@k of Par-SR is computed as

$$AvgDiff@k = \frac{\sum_{v \in \{TopK\} \cap \{TopK'\}} |(1-c)S'(u,v) - S(u,v)|}{\#|\{TopK\} \cap \{TopK'\}|}.$$

Furthermore, considering that $S'(u, v)$ and $S(u, v)$ are close to each other practically due to our empirical observation, AvgDiff@k of Par-SR can be further approximated by: $|(1 - c)S'(u, v) - S(u, v)| \approx c \cdot S_{(u,v)}$, i.e.,

$$AvgDiff@k = \frac{\sum_{v \in \{TopK\} \cap \{TopK'\}} c \cdot S(u,v)}{\#|\{TopK\} \cap \{TopK'\}|},$$

where $c \cdot S(u, v)$ means the multiplication of c and $S(u, v)$. When k increases, more vertices with smaller SimRank scores are included in the answer. However, AvgDiff@k is the average of these differences, so AvgDiff@k decreases.

5.3 Effectiveness and Efficiency

In this section, the algorithms in different categories are compared separately. Both efficiency and effectiveness are evaluated for each SimRank algorithm.

5.3.1 *Comparison of Iterative Algorithms*

The efficiency of Partial-SR and OIP-DMST, together with the corresponding optimized versions, namely, Opt-Partial-SR and Opt-OIP-DMST, which only compute the upper triangular part of the similarity matrix, are presented. The result of Naive-SR is not presented for simplicity since Naive-SR consumes more time than Partial-SR and OIP-DMST theoretically and experimentally. The results are shown in Figure 6, where the run time of an algorithm refers to the computation time for all-pairs SimRank. Effectiveness is not considered because iterative algorithms produce exact solutions.

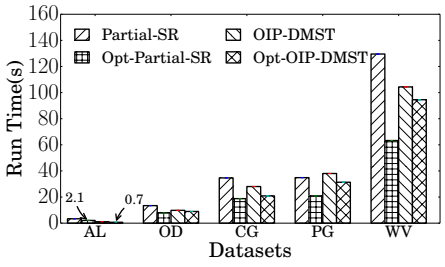


Figure 6: Run time (seconds) of iterative algorithms.

Figure 6 shows the efficiency of iterative algorithms on the five datasets. First, OIP-DMST performs better than Partial-SR in most cases since OIP-DMST reduces the amount of redundant computation in a finer granularity. For example, Partial-SR needs 130 seconds to finish the computation on WV while OIP-DMST needs 104 seconds. The reason why OIP-DMST is a little slower than Partial-SR on PG is that PG is a sparse graph and there is little redundant computation for partial sums, while OIP-DMST needs extra time to determine the computation order of partial sums. Second, Opt-Partial-SR is nearly twice as fast as Partial-SR because half of the computation of the partial sums in Opt-Partial-SR are avoided. On CG, for example, Partial-SR needs 34 seconds and Opt-Partial-SR needs 18 seconds. Third, Opt-OIP-DMST does not speed up as much as Opt-Partial-SR does. This is because the computation of partial sums and SimRank values are separated. All the partial sums have to be computed before getting any SimRank scores. Finally, the efficiency of Opt-OIP-DMST is not always better than Opt-Partial-SR due to the different degree of improvement from the optimization of computing half of the similarity matrix. For example, Opt-OIP-DMST is faster than Opt-Partial-SR on AL but slower on WV.

5.3.2 *Comparison of Non-iterative Algorithms*

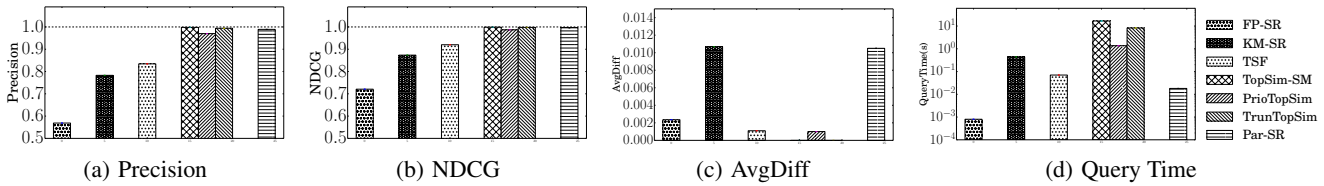
Effectiveness and efficiency of the non-iterative algorithms are evaluated with five small datasets. Specifically, comparisons between NI-Sim and Par-SR are reported since SimMat cannot run on these datasets due to the irreversibility of the transition matrix W. The results are shown in Figure 5.

Effectiveness. Figures 5(a) and 5(b) show that Par-SR outperforms NI-Sim in terms of Precision and NDCG. This is because the accuracy of Par-SR is influenced by the number of iterations and SimRank is accurate enough with ten iterations [9]. However, NI-Sim loses accuracy because it uses SVD to approximate the real SimRank scores but SVD fails to preserve the relative order of estimated values.

Figure 5(c) indicates that NI-Sim does not always perform better than Par-SR on AvgDiff. For example, AvgDiff of NI-Sim on WV and AL is better than that of Par-SR but worse on OD and CG. This is because, although both Par-SR and NI-Sim are based on non-iterative method, which ignores the first-meeting restriction, NI-Sim adopts SVD to compute SimRank. Thus, the NI-Sim result fluctuates around the "accurate" SimRank scores computed without guaranteeing first-meeting. As a result, the AvgDiff of NI-Sim is sometimes better than Par-SR, sometimes worse.

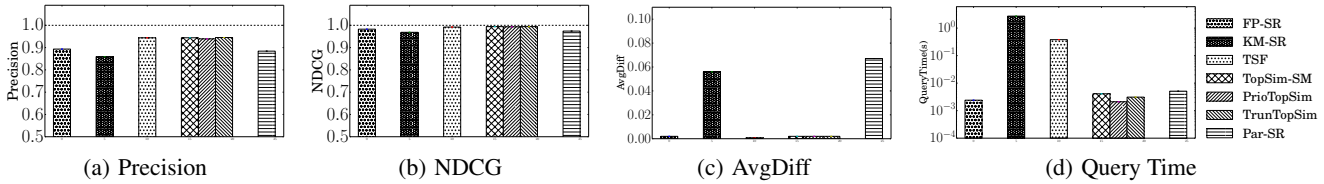
Efficiency. Figure 5(d) indicates that Par-SR is much faster than NI-Sim. For instance, Par-SR takes 0.005 seconds for a top-k query while NI-Sim needs 1.744 seconds on PG. This is because, although the time complexity for answering a top-k query in NI-Sim and Par-SR is linear for both, the constant factor of NI-Sim is much larger than that of Par-SR.

Summary. Par-SR outperforms NI-Sim in terms of effectiveness and efficiency. This also indicates that the techniques based on matrix decomposition do not work well with SimRank problems.



(a) Precision (b) NDCG (c) AvgDiff (d) Query Time

Figure 7: Results for algorithms based on random walk on WikiVote. Precision, NDCG, and AvgDiff describe the effectiveness while query time indicates the efficiency. The dotted lines represent the exact solutions. The AvgDiff of exact solutions is 0.



(a) Precision (b) NDCG (c) AvgDiff (d) Query Time

Figure 8: Results for algorithms based on random walk on P2p-Gnutella08. Precision, NDCG, and AvgDiff describe the effectiveness while query time indicates the efficiency. The dotted lines represent the exact solutions. The AvgDiff of exact solutions is 0.

5.3.3 *Comparison of Random Walk-Based Algorithms*

In this section, a comparison among algorithms which are based on random walk is conducted. Par-SR is also compared since it can be viewed as a special kind of algorithm based on random walk as discussed in Section 3 and it outperforms the other two non-iterative algorithms.

The datasets used here are five small datasets and three medium datasets. We run these algorithms on all of the small datasets but only the results for WV and PG are presented, as shown in Figures 7 and 8. For medium datasets, only the efficiency is presented in Tables 5 and 6. The effectiveness cannot be computed because accurate SimRank scores cannot be estimated for these medium graphs as we claimed in Table 4.

Effectiveness. TopSim-based algorithms perform better on effectiveness than Monte Carlo sampling-based algorithms with limited samples. For example, the Precision and the NDCG of TopSim based algorithms are higher than those of TSF, FP-SR, or KM-SR on WV, as shown in Figures 7(a) and 7(b). This is because the accuracy loss of TopSim-based algorithms comes from the limited length of the random walks and SimRank converges fast in a short distance. However, the accuracy of Monte Carlo sampling-based algorithms like TSF is restricted by the number of samples used.

Monte Carlo sampling-based algorithms perform better on local sparse graphs than on local dense graphs. For example, the Precision of FP-SR on WV is lower than that on PG, 0.57 and 0.89, respectively, as shown in Figures 7(a) and 8(a). WV is a local dense graph and it incurs more paths starting from the query vertex than PG. As a result, the original number of sampled random walks cannot characterize well all the paths in WV. The number of samples has to increase if a higher accuracy on WV is needed.

TSF performs better than FP-SR on dense graphs. For example, the Precision of TSF on WV is 0.83, much higher than that of FP-SR, 0.57, as shown in Figure 7(a). When answering a top-k query, FP-SR scans the indexed FPGs while TSF resamples R_q random walks for each indexed one-way graph. Thus, the number of coupled random walks used in TSF is much higher than that of FP-SR.

Algorithms not guaranteeing first-meeting have poor AvgDiff but good Precision and NDCG. For example, Par-SR has poor AvgDiff on WV and PG, about 0.01 and 0.06, respectively, but it has good Precision and NDCG, as shown in Figures 7 and 8. This is because the results for Par-SR are scaled to the accurate SimRank scores without guaranteeing first-meeting by a factor of $(1 - c)$, as discussed in Section 3. Therefore, the SimRank score is not accurate but the relative order is preserved.

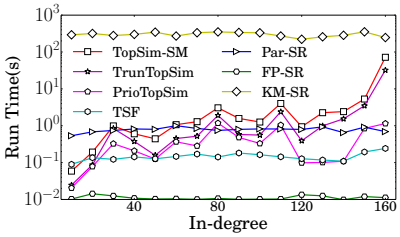
Efficiency. The efficiency of TopSim-based algorithms is sensitive to graph density. TopSim-SM performs poorly for dense graphs. For example, a single query for TopSim-SM on WV takes 16 seconds while it needs only 0.001 seconds on PG, as shown in Figures 7(d) and 8(d). This is because TopSim-based algorithms rely heavily on local graph structures. They enumerate all the random walks like TopSim-SM. TrunTopSim and PrioTopSim improve the efficiency by pruning random walks with small probabilities. However, this still does not ease the situation because the number of random walks is still exponential with the number of walking steps. For example, on WV, TrunTopSim takes 8.2 seconds to answer a top-k query, PrioTopSim takes 1.36 seconds while TSF needs only 0.06 seconds, as shown in Figure 7(d).

The efficiency of TSF is better than that of KM-SR, while it is worse than that of FP-SR. For example, when answering a top-k query on ND, TSF takes 0.67 seconds while FP-SR needs 0.11 seconds and KM-SR uses 28.2 seconds, as shown in Table 5. The efficiency of KM-SR is poor because all SimRank scores with respect to the given vertex have to be computed for picking the k most similar vertices as the answer. However, FP-SR and TSF use the graph connectivity of the index to avoid a global search of the top-k similar vertices. Furthermore, TSF is slower than FP-SR because TSF resamples R_q random walks and calculates the points where the sampled coupled random walks meet when querying, while FP-SR just needs to scan the indexed FPGs because the information about the meetings is also stored in the FPGs.

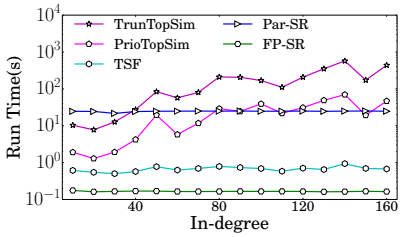
	ND	BS	WG
KM-SR	28.2	305	160
TopSim-SM	0.09	0.04	0.03
TrunTopSim	0.08	0.02	0.02
PrioTopSim	0.01	0.01	0.01
TSF	0.67	0.16	0.10
FP-SR	0.11	0.02	0.02
Par-SR	0.22	0.53	0.80

Table 5: Query time (seconds) on medium datasets.

Table 6 shows the index building time of Monte Carlo sampling-based algorithms. It can be concluded that FP-SR needs more time



(a) webBerkStan



(b) LiveJournal

Figure 9: Robustness of SimRank algorithms on webBerkStan and LiveJournal.

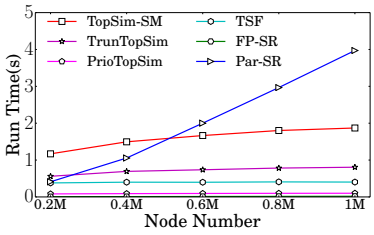


Figure 10: Scalability Results.

to build the index. For example, FP-SR needs 168.2 seconds while TSF needs only 18.8 seconds on BS. This is because FP-SR not only samples the random walks but also computes the points where each pair of random walks first meet when building the index.

	ND	BS	WG
KM-SR	32.7	47.6	69.8
TSF	8.9	18.8	28.4
FP-SR	79.5	168.2	216.6

Table 6: Index building time (seconds) on medium datasets.

Summary. Algorithms based on random walk have their own advantages and disadvantages. Path-enumeration-based algorithms, like TopSim-based solutions, have good accuracy but their efficiency is sensitive to graph density. FP-SR can answer a top-k query fast but the index cost is expensive. TSF achieves a balance between space and time cost with good accuracy. The efficiency of KM-SR is not acceptable.

5.4 Robustness Analyses

To demonstrate the robustness of each algorithm, two experiments on LJ and BS, a dense graph and a sparse graph, respectively, were conducted with the in-degree of vertices increasing from 10 to 160 in steps of 10. The results are shown in Figure 9. The query time for each in-degree is the average of ten queries. The results for TopSim-SM and KM-SR on LJ are not presented since they cannot finish in 20 hours. Iterative algorithms were not evaluated because they are robust by definition.

Figure 9 presents the robustness of each algorithm on BS and LJ. First, Monte Carlo sampling-based algorithms are robust. For example, the efficiency of TSF remains almost unchanged as the vertex in-degree increases on BS and LJ, as shown in Figures 9(a) and 9(b). This is guaranteed by the Monte Carlo sampling techniques. The fixed number of samples used prevents the efficiency of these algorithms from being affected by the graph structures. Second, Par-SR is robust as well. This is because the time complexity of Par-SR is linear with the number of the edges in the graph when the iteration number is set as ten. Finally, TopSim-based algorithms are heavily affected by the vertex in-degree. The query time increases wildly as the vertex in-degree increases, as shown in Figures 9(a) and 9(b). This is because the length of random walks in TopSim-based algorithms is fixed ($T = 3$) but they enumerate most of the random walks. As the in-degree of the query vertex increases, the local graph becomes denser and it incurs exponentially more random walks. Therefore, the TopSim-based solutions require more time to answer the query.

Summary. TSF, KM-SR, FP-SR, and Par-SR are robust and can answer a top-k query in a stable time. TopSim-based algorithms are sensitive to the graph structure and the query time increases sharply as the in-degree of the query vertex grows.

5.5 Scalability Analyses

We report how each algorithm performs for graphs with different scales. The datasets used here have various numbers of vertices from 0.2 million to 1 million, with the average in-degree set as ten. The result for KM-SR is not presented for clarity of the figure because KM-SR consumes 200× more time than other algorithms on these datasets. The scalability of iterative algorithms was not tested because they can only handle small graphs. Furthermore, extra large real-world datasets are used to demonstrate how each algorithm performs under large-scale graphs.

Figure 10 demonstrates how the efficiency of each algorithm changes when the scale of the graphs increases. First, the efficiency of FP-SR and TSF remains almost unchanged when the graph becomes bigger. This is because both FP-SR and TSF use a fixed number of random walks to approximate SimRank scores. The increased graph size causes the index building process to consume more time but has little effect on the in-memory query process. Second, TopSim-based algorithms are not influenced by the graph scale because they enumerate all the random walks in a local area ($T = 3$) to compute SimRank scores. When the scale of the graphs increases while the local structures remain unchanged, the efficiency of TopSim-based solutions is stable because the number of random walks used for computing SimRank remains unchanged. Finally, the query time of Par-SR increases linearly with respect to the graph scale. This is because the time complexity of Par-SR is linear with the graph size when the iteration number is set to ten.

Table 7 shows the efficiency of SimRank algorithms on four large real-world datasets. First, FP-SR cannot handle large graphs. For example, FP-SR on WP runs out of memory due to the high space cost for the index. Second, TSF can process large graphs, like IT and WP, efficiently. The reason why TSF needs more time on TW is that TW is so dense that fewer vertices are filtered by the connectivity of one-way graphs. Third, TopSim-based algorithms cannot answer a top-k query in large dense graphs, like TW. This is due to the extremely dense substructure of TW. Finally, Par-SR can handle large graphs, like LJ and IT, in a relatively stable time.

	LJ	WP	IT	TW
TopSim-SM	660.5	13.04	1.93	MLE
TrunTopSim	17.1	4.87	0.48	MLE
PrioTopSim	0.99	0.39	0.25	3603
TSF	0.63	2.29	1.31	174.7
FP-SR	0.12	MLE	MLE	MLE
Par-SR	22.11	68.03	116.6	751.2

Table 7: Query time (seconds) on large datasets. "MLE" stands for "memory limited error".

Summary. The efficiency of TSF and FP-SR does not change too much when the graph scale increases if the indices can be put in memory. TopSim-based solutions can scale to large graphs but their

efficiency relies on the local structure of the graphs. The efficiency cost of Par-SR increases linearly when the graph scale increases.

6. OTHER ALGORITHMS

Most of the SimRank algorithms are discussed and re-implemented in our paper. However, there are still some other algorithms that do not belong to the three classes or have poor performance for a top-k query theoretically. Now we introduce them as a complement to the SimRank topic.

FS-SR. Li et al. [13] proposed FS-SR to compute the single-pair SimRank in $O(Td^2 \cdot \min\{d^T, n^2\})$ time based on random walk. However, it is expensive for FS-SR to handle a top-k query since it has to calculate all SimRank scores with respect to the given vertex.

BlockSimRank. Li et al. [12] introduced BlockSimRank, an efficient similarity computation algorithm that exploits the block structure. BlockSimRank first partitions the graph into blocks. Then it computes similarity scores in each block iteratively. Finally, the similarity between vertices in different blocks is estimated. As a result, BlockSimRank achieves a performance improvement from $O(Tn^2d^2)$ to $O(Tn^{\frac{4}{3}}d^2)$ on average.

SRJ. Zheng et al. [24] introduced a new SimRank join-based query problem, which aims to find vertex pairs (u, v) from two vertex sets U and V ($u \in U, v \in V$) whose SimRank scores are larger than a given threshold. They proposed SRJ, which uses some stored SimRank scores to compute the unknown ones by solving linear equations. However, SRJ cannot scale to large graphs because it depends on the product graph ($G \times G$).

TSJ. Tao et al. [20] introduced the top-k SimRank-based similarity join problem, and proposed a solution, TSJ, which encodes each vertex as a vector and computes SimRank via a vector–vector multiplication. To cope with the top-k similarity join problem, TSJ identifies $2k$ candidates and uses a tree-based WAND algorithm to identify the answers based on the candidates. TSJ needs $O(nd^T)$ space to store the vectors via sparse representation, where d is the average in-degree of the graph.

7. CONCLUSION

In this paper, we discussed in depth the existing algorithms that compute SimRank and classified them into three different categories, i.e., iterative method, non-iterative method, and random walk. Furthermore, we set up a unified environment and comprehensively compared the algorithms via different metrics, including efficiency, effectiveness, robustness, and scalability. Another criteria is how easy an algorithm is to tune (e.g., users might prefer a parameter-free algorithm even if it is slightly slower or less accurate); however, as we will see, our guideline won't change even if we take the cost of hyperparamter tuning into consideration.

We provide the following high-level suggestions for choosing algorithms given a SimRank task.

1. For sparse graphs, TopSim-based solutions perform better; for small dense graphs, Par-SR is the most suitable solution. Moreover, Par-SR and TopSim-based have no hyperparamters to tune.
2. If the workload does not fit into the category above, Monte Carlo sampling-based algorithms are recommended. For example, we recommend TSF given large local dense graphs (e.g. Twitter in Table 7.).

Furthermore, with the theoretical analyses and experimental studies, we summarized a grade table that describes the pros and cons of each algorithm under different metrics in Table 8. Therefore, users can find a good option given a SimRank task.

Algorithms	Method	Efficiency	Effectiveness	Robustness	Scalability
Naive-SR	I	1	5	5	1
Partial-SR	I	1	5	5	1
OIP-DMST	I	1	5	5	1
SimMat	N	0	0	0	0
NI-Sim	N	1	1	5	2
Par-SR	N	3	5	5	4
KM-SR	R	2	4	5	2
TSF	R	4	4	5	5
FP-SR	R	5	3	5	4
TopSim-SM	R	4	5	2	5
TrunTopSim	R	4	5	2	5
PrioTopSim	R	5	4	3	5

Table 8: Grades of algorithms for different metrics. The grade varies from zero to five, and a higher grade indicates that the algorithm is better at the corresponding metric. "I," "N," and "R" stand for iterative method, non-iterative method, and random walk, respectively.

8. REFERENCES

[1] Z. Abbassi and V. S. Mirrokni. A recommender system based on local random walks and spectral methods. In *WebKDD/SNA-KDD*, pages 102-108, 2007.

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

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

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

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

[6] D. Goldberg, D. Nichols, B. M. Oki, and D. Terry. Using collaborative filtering to weave an information tapestry. *Commun. ACM*, 35(12):61–70, Dec. 1992.

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

[8] J. A. Konstan, B. N. Miller, D. Maltz, J. L. Herlocker, L. R. Gordon, and J. Riedl. Grouplens: Applying collaborative filtering to usenet news. *Commun. ACM*, 40(3):77–87, Mar. 1997.

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

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

[11] 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.

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

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

[14] D. Liben-Nowell and J. Kleinberg. The link prediction problem for social networks. In *CIKM*, pages 556-559, 2003.

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

[16] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web, 1998.

[17] C. Sanderson. Armadillo: An Open Source C++ Linear Algebra Library for Fast Prototyping and Computationally Intensive Experiments. Technical report, NICTA, Sept. 2010.

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

[19] U. Shardanand and P. Maes. Social information filtering: Algorithms for automating: Word of mouth. In *CHI*, pages 210-217, 1995.

[20] W. Tao and G. Li. Efficient top-k simrank-based similarity join. In *SIGMOD*, pages 1603-1604, 2014.

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

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

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

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