Towards Efficient SimRank Computation on Large Networks

Weiren Yu †1, Xuemin Lin †2, Wenjie Zhang †3

† *The University of New South Wales, Sydney, Australia*

1,2,3 {weirenyu, lxue, zhangw}@cse.unsw.edu.au

***Abstract*—SimRank has been a powerful model for assessing the similarity of pairs of vertices in a graph. It is based on the concept that two vertices are similar if they are referenced by similar vertices. Due to its self-referentiality, fast SimRank computation on large graphs poses significant challenges. The state-of-the-art work [16] exploits partial sums memorization for computing SimRank in $O(Kmn)$ time on a graph with n vertices and m edges, where K is the number of iterations. Partial sums memorizing can reduce repeated calculations by caching part of similarity summations for later reuse. However, we observe that computations among different partial sums may have duplicate redundancy. Besides, for a desired accuracy ϵ, the existing SimRank model requires $K = \lceil \log_C \epsilon \rceil$ iterations [16], where C is a damping factor. Nevertheless, such a geometric rate of convergence is slow in practice if a high accuracy is desirable.**

In this paper, we address these gaps. (1) We propose an adaptive clustering strategy to eliminate partial sums redundancy (*i.e.,* duplicate computations occurring in partial sums), and devise an efficient algorithm for speeding up the computation of SimRank to $O(Kd'n^2)$ time, where d' is typically much smaller than the average in-degree of a graph. (2) We also present a new notion of SimRank that is based on a differential equation and can be represented as an exponential sum of transition matrices, as opposed to the geometric sum of the conventional counterpart. This leads to a further speedup in the convergence rate of SimRank iterations. (3) Using real and synthetic data, we empirically verify that our approach of partial sums sharing outperforms the best known algorithm by up to one order of magnitude, and that our revised notion of SimRank further achieves a 5X speedup on large graphs while also fairly preserving the relative order of original SimRank scores.

I. Introduction

Identifying similar objects based on hyperlink structure is a fundamental operation for many web mining tasks. Examples include web page ranking [3], hypertext classification (KNN) [13], graph clustering (K-means) [5], and collaborative filtering [11], [17]. In the last decade, with the overwhelming number of objects on the Web, there is a growing need to be able to automatically and efficiently assess the similarity of these objects on large graphs. Indeed, the Web has huge dimensions and continues to grow rapidly— more than 5% of new objects are created per week [4]. As a result, similarity assessment on web objects is apt to become obsolete very shortly. In light of this, it is imperative for similarity assessment to get a fast computational speed on large graphs.

Amid the existing similarity metrics, SimRank [11] has emerged as a powerful tool for assessing structural similarities between objects. Similar to the well-known PageRank [3], SimRank scores depend merely on the link structure of the Web, independent of the textual content of objects. The main difference between the two models is the scoring mechanism. PageRank assigns an authority weight for each object, whereas SimRank assigns a similarity score between two objects. SimRank was first proposed by Jeh and Widom [11], and has gained enormous popularity for its success in many areas such as bibliometrics [14], top-K search [13], and recommender systems [1]. This reveals the importance of SimRank as an effective measure. The intuition underlying SimRank is a subtle recursion that "two vertices are similar if their incoming neighbors are similar", together with the base case that "every vertex is maximally similar to itself" [11]. Due to this self-referential concept, conventional algorithms for computing SimRank have an iterative nature. The sheer size of the Web has presented striking challenges to fast SimRank computing. The best known algorithm proposed by Lizorkin *et al.* [16] (hereafter referred to as psum-SR) requires $O(Kmn)$ time ($O(Kn^3)$ in the worst case) for K iterations, where n and m denote the number of vertices and edges, respectively.

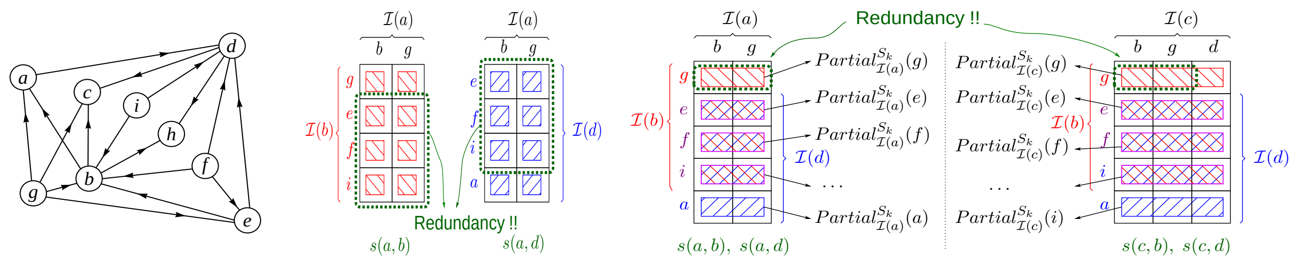
The beauty of psum-SR algorithm [16] resides in the following three observations. (1) *Essential nodes selection* may eliminate the computation of a fraction of node pairs with a-priori zero scores. (2) *Partial sums memorizing* can effectively reduce repeated calculations of the similarity among different node pairs by caching part of similarity summations for later reuse. (3) *A threshold setting* on the similarity enables a further reduction in the number of node pairs to be computed. Particularly, the second observation of *partial sums memorizing* plays a paramount role in greatly speeding up the computation of SimRank from the naive $O(Kd^2n^2)$ [11] to $O(Kdn^2)$, [1] where d is the number of average in-degrees in a graph.

Before shedding light on the limitations of psum-SR [16], let us first revisit the central idea of partial sums memorizing, as illustrated in the following example.

Example 1. Consider a paper citation network $\mathcal{G}$ in Fig. 1a, where each vertex represents a paper, and an edge a citation. For any vertex a, we denote by $\mathcal{I}(a)$ the set of in-neighbors of a. Individual element in $\mathcal{I}(a)$ is denoted as $\mathcal{I}_i(a)$. Let $s(a,b)$ be the SimRank similarity between vertices a and b. In what follows, we want to compute $s(a,b)$ and $s(a,d)$ in $\mathcal{G}$.

Before partial sums memorizing is introduced, a naive way is to sum up the similarities of all in-neighbors $(\mathcal{I}_i(a), \mathcal{I}_j(b))$ of (a,b) for computing $s(a,b)$, and to sum up the similarities

[1]The degree sum formula $n \cdot d = m$ implies that $O(Kmn)$ time in [16] is equivalent to $O(Kdn^2)$.


$\mathcal{I}(a)$
$\mathcal{I}(b)$
$\mathcal{I}(d)$
$\mathcal{I}(c)$
Redundancy !!
$s(a,b)$
$s(a,d)$
$s(a,b),\ s(a,d)$
$s(c,b),\ s(c,d)$
$Partial^{S_k}_{\mathcal{I}(a)}(g)$
$Partial^{S_k}_{\mathcal{I}(a)}(e)$
$Partial^{S_k}_{\mathcal{I}(a)}(f)$
$Partial^{S_k}_{\mathcal{I}(a)}(a)$
$Partial^{S_k}_{\mathcal{I}(c)}(g)$
$Partial^{S_k}_{\mathcal{I}(c)}(e)$
$Partial^{S_k}_{\mathcal{I}(c)}(f)$
$Partial^{S_k}_{\mathcal{I}(c)}(i)$

(a) A paper citation network (b) Naive method in [11] (c) *Partial sums memorizing* leads to redundancy among different partial sums

Fig. 1: Merit and demerit of *partial sums memorizing* for SimRank computation on a paper citation network

of all in-neighbors $(\mathcal{I}_i(a), \mathcal{I}_j(d))$ of (a,d) for computing $s(a,d)$, *independently*, as depicted in Fig. 1b. In contrast, psum-SR is based on the observation that $\mathcal{I}(b)$ and $\mathcal{I}(d)$ have three vertices $\{e,f,i\}$ in common. Thus, the three partial sums over $\mathcal{I}(a)$ (*i.e.,* $Partial^{s_k}_{\mathcal{I}(a)}(y)$ [2] with $y \in \{e,f,i\}$) can be computed only once, and reused for both $s(a,b)$ and $s(a,d)$ computation (see left part of Fig. 1c). Similarly, for computing $s(c,b)$ and $s(c,d)$, since $\mathcal{I}(b) \cap \mathcal{I}(d) = \{e,f,i\}$, the partial sums over $\mathcal{I}(c)$ (*i.e.,* $Partial^{s_k}_{\mathcal{I}(c)}(x)$ with $x \in \{e,f,i\}$) can be cached for later reuse (see right part of Fig. 1c). ■

Despite the aforementioned merits of psum-SR, existing work [16] on SimRank has the following limitations.

Firstly, we observe from Example 1 that computing partial sums [16] over different in-neighbor sets may have duplicate redundancy. For instance, $\mathcal{I}(a)$ and $\mathcal{I}(c)$ in Fig. 1c have two vertices $\{b,g\}$ in common, implying that the sub-summation $Partial^{s_k}_{\{b,g\}}(\star)$ is the common part shared between the partial sums $Partial^{s_k}_{\mathcal{I}(a)}(\star)$ and $Partial^{s_k}_{\mathcal{I}(c)}(\star)$. Thus, there is an opportunity to speed up the computation of SimRank by preprocessing the common sub-summation $Partial^{s_k}_{\{b,g\}}(\star)$ once, and caching it for both $Partial^{s_k}_{\mathcal{I}(a)}(\star)$ and $Partial^{s_k}_{\mathcal{I}(c)}(\star)$ computation. However, it is a big challenge to identify the well-tailored common parts for maximal sharing among the partial sums over different in-neighbor sets since there could be many irregularly and arbitrarily overlapped in-neighbor sets in a real graph. To address this issue, we propose optimization techniques to have such common parts memorized in a hierarchical clustering manner, and devise an efficient algorithm to eliminate such redundancy.

Secondly, the existing iterative paradigm [16] for computing SimRank has a geometric rate of convergence, which might be, in practice, rather slow when a high accuracy is attained. This is especially evident in *e.g.,* citation networks and web graphs [12]. For instance, our experiments on DBLP citation network shows that a desired accuracy of $\epsilon = 0.001$ may lead to more than 30 iterations of SimRank, for the damping factor $C = 0.8$. Lizorkin *et al.* [16] has proved theoretically in [16] that, for a desired accuracy ϵ, the number of iterations required for the conventional SimRank is $K = \lceil \log_C \epsilon \rceil$, which is due mainly to the geometric sum of the traditional representation of SimRank. This highlights the need for a revised SimRank model to speed up the geometric rate of convergence.

Contributions. Our main contributions are as follows.

- We propose an adaptive clustering strategy based on a minimum spanning tree to eliminate the partial sums redundancy [16] in a hierarchical fashion (Section III). By optimizing the sub-summations shared among the different partial sums, an efficient algorithm is devised for speeding up the computation of SimRank from $O(Kdn^2)$ [16] to $O(Kd'n^2)$ time, where d' $(\leq d)$ can, in general, be much smaller than the average in-degree d.
- We introduce a new notion of SimRank by using a matrix differential equation to further accelerate the convergence of SimRank iterations from the traditional geometric to exponential rate (Section IV). We show that the new notion of SimRank can be characterized as an exponential sum in terms of the transition matrix while fairly preserving the relative order of SimRank scores, as opposed to the conventional counterpart [16] as a geometric sum.
- We conduct extensive experiments on real and synthetic datasets (Section V), demonstrating that our approach of partial sum sharing on large graphs can be one order of magnitude faster than psum-SR. In addition, our revised notion of SimRank achieves up to a 5X further speedup against the conventional counterpart.

Related Work. The development of methods for efficiently computing SimRank is a vibrant research area [14], [16], [18] that is fundamental to *e.g.,* web mining and object ranking. Recent results on SimRank can be summarized as follows.

The earliest mention of SimRank dates back to Jeh and Widom [11] who suggested (i) an iterative approach to compute SimRank, which is in $O(Kd^2n^2)$ time, along with (ii) a heuristic pruning rule to set the similarity between far-apart vertices to be zero. Unfortunately, the naive iterative SimRank is rather costly to compute, and there is no provable guarantee on the accuracy of the pruning results. To overcome the limitations, a very appealing attempt was made by Lizorkin *et al.* [16] who (i) provided accuracy guarantees for SimRank iterations, *i.e.,* the number of iterations needed for a given accuracy ϵ is $K = \lceil \log_C \epsilon \rceil$, and (ii) proposed three excellent optimization approaches, *i.e.,* essential node-pair selection, partial sums memorization, and threshold-sieved similarities. Especially, partial sums memorizing serves as the cornerstone of their strategies, which significantly reduces the compu-

[2] Recall from [16] that *a partial sum* for a binary function $f : \mathcal{X} \times \mathcal{Y} \to \mathbb{R}$ over a set $\mathcal{D} = \{x_1, \cdots, x_n\} \subseteq \mathcal{X}$, denoted by $Partial^{f}_{\mathcal{D}}(\star)$, is defined as
$$Partial^{f}_{\mathcal{D}}(y) = \sum_{x_i \in \mathcal{D}} f(x_i, y), \quad (y \in \mathcal{Y}).$$

tation of SimRank to $O(Kdn^2)$ time. However, repeated sub-summations among different partial sums were largely overlooked by Lizorkin *et al.* . Our work differs from [16] in the following. (i) We put forward the phenomenon of partial sums redundancy in [16] that typically exists in real graphs. (ii) We accelerate the convergence of SimRank iterations from geometric [16] to exponential growth, by revising the existing SimRank model.

There has also been a flurry of research interests (*e.g.,* [1], [6], [10], [13]–[15], [19]) in the SimRank optimization problems. Li *et al.* [14] first based SimRank computation on the matrix representation. They developed very interesting SimRank approximation techniques on a low-rank graph, by leveraging the singular value decomposition and tensor product. However, (i) for digraphs, the upper bound of approximation error still remains unknown. (ii) The computational time in [14] would become $O(n^4)$ even when the rank of an adjacency matrix is relatively small, *e.g.,* $\lceil\sqrt{n}\rceil$ ($\ll n$). The pioneering work of He *et al.* [10] utilized the node-updating method on GPU for parallel SimRank computing. They deployed iterative aggregation techniques to accelerate the global convergence of parallel SimRank, in which the speed-up in the global convergence of SimRank is due mainly to the different local convergence rates on small matrix partitions. Recently, the new notions of weight- and evidence-based SimRank have been suggested in [1] to address the issue of query rewriting for sponsored search. Fogaras *et al.* [6] adopted a scalable Monte Carlo sampling approach to estimate SimRank by using the first meeting time of two random surfers. However, their algorithms are probabilistic in nature. Li *et al.* [15] employed an effective method for locally computing single-pair SimRank by breaking the holistic nature of the SimRank recursion. Zhao *et al.* [19] proposed a new ranking model, termed Penetrating Rank (P-Rank), by taking account of both in- and out-links. Since the iterative paradigms of SimRank and P-Rank are almost similar, our techniques for SimRank can be easily extended to P-Rank computation. Lee *et al.* [13] devised a top-K SimRank algorithm that only needs to access a small fraction of vertices in a graph.

II. Preliminaries

In this section, we review the two formulations of SimRank, *i.e.,* the iterative form [11], [16], and the matrix form [10], [14]. The consistency of two forms was pointed out in [14].

A. Iterative Form of SimRank

Formally, given a directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ with a vertex set $\mathcal{V}$ and an edge set $\mathcal{E}$, the SimRank similarity between vertices a and b, denoted by $s(a,b)$, is defined as (i) $s(a,a) = 1$; (ii) $s(a,b) = 0$, if $\mathcal{I}(a) = \varnothing$ or $\mathcal{I}(b) = \varnothing$; (iii) otherwise,

$$s(a,b) = \frac{C}{|\mathcal{I}(a)|\,|\mathcal{I}(b)|} \sum_{j\in\mathcal{I}(b)} \sum_{i\in\mathcal{I}(a)} s(i,j), \tag{1}$$

where $C \in (0,1)$ is a damping factor, $\mathcal{I}(a)$ is the in-neighbor set of a vertex a, and $|\mathcal{I}(a)|$ is the cardinality of $\mathcal{I}(a)$.

The above formulas naturally serve to introduce the iterative method by starting with $s_0(a,a) = 1$ and $s_0(a,b) = 0$ if $a \neq b$, and for $k = 0, 1, 2, \cdots$, setting (i) $s_{k+1}(a,a) = 1$; (ii) $s_{k+1}(a,b) = 0$, if $\mathcal{I}(a) = \varnothing$ or $\mathcal{I}(b) = \varnothing$; (iii) otherwise,

$$s_{k+1}(a,b) = \frac{C}{|\mathcal{I}(a)||\mathcal{I}(b)|} \sum_{j\in\mathcal{I}(b)} \sum_{i\in\mathcal{I}(a)} s_k(i,j). \tag{2}$$

The resultant sequence $\{s_k(a,b)\}_{k=0}^{\infty}$ converges to $s(a,b)$, the *exact* solution of Eq.(1).

B. Matrix Form of SimRank

In matrix notations, SimRank can be formulated as follows.

$$\mathbf{S} = C \cdot (\mathbf{Q} \cdot \mathbf{S} \cdot \mathbf{Q}^T) + (1 - C) \cdot \mathbf{I}_n, \tag{3}$$

where $\mathbf{S}$ is the similarity matrix whose entry $[\mathbf{S}]_{i,j}$ denotes the similarity score $s(i,j)$, $\mathbf{Q}$ is the backward transition matrix whose entry $[\mathbf{Q}]_{i,j} = 1/|\mathcal{I}(i)|$ if there is an edge from j to i, and 0 otherwise, and $\mathbf{I}_n$ is an $n \times n$ identity matrix.

III. Eliminating Partial Sums Redundancy

The existing method psum-SR [16] of performing Eq.(2) is to memorize the partial sums over $\mathcal{I}(a)$ first:

$$Partial_{\mathcal{I}(a)}^{s_k}(j) = \sum_{i\in\mathcal{I}(a)} s_k(i,j), \quad (j \in \mathcal{I}(b)) \tag{4}$$

and then iteratively compute $s_{k+1}(a,b)$ as follows:

$$s_{k+1}(a,b) = \frac{C}{|\mathcal{I}(a)||\mathcal{I}(b)|} \sum_{j\in\mathcal{I}(b)} Partial_{\mathcal{I}(a)}^{s_k}(j). \tag{5}$$

Consequently, the results of $Partial_{\mathcal{I}(a)}^{s_k}(j)$, $\forall j \in \mathcal{I}(b)$, can be reused later when we compute the similarities $s_{k+1}(a,\star)$ for a given vertex a as the first argument. However, we observe that the partial sums over different in-neighbor sets may share common sub-summations. For example in Fig. 1c, the partial sums $Partial_{\mathcal{I}(a)}^{s_k}(\star)$ and $Partial_{\mathcal{I}(c)}^{s_k}(\star)$ have the sub-summation $Partial_{\{b,g\}}^{s_k}(\star)$ in common. Based on this observation, we will discuss how to optimize sub-summations shared among different partial sums in this section.

A. Partition In-neighbor Sets for (Inner) Partial Sums Sharing

We first introduce the notion of *a set partition.*

Definition 1. A partition *of a set $\mathcal{D}$, denoted by $\mathscr{P}(\mathcal{D})$, is a family of disjoint subsets $\mathcal{D}_i$ of $\mathcal{D}$ whose union is $\mathcal{D}$, i.e.,*

$$\mathscr{P}(\mathcal{D}) = \{\mathcal{D}_1, \mathcal{D}_2, \cdots, \mathcal{D}_p\}, \text{ with } p = |\mathscr{P}(\mathcal{D})|,$$

where $\mathcal{D}_i \cap \mathcal{D}_j = \varnothing$ for $i \neq j$, and $\bigcup_{i=1}^{p} \mathcal{D}_i = \mathcal{D}$.

For instance, $\mathscr{P}(\mathcal{I}(b)) = \{\{f,g\},\{e,i\}\}$ is a partition of the in-neighbor set $\mathcal{I}(b) = \{f,g,e,i\}$ in Fig 1a.

The set partition is deployed for speeding up the computation of SimRank based on the following proposition.

Proposition 1. *For two distinct vertices a, b with $\mathcal{I}(a) \neq \varnothing$ and $\mathcal{I}(b) \neq \varnothing$, $s_{k+1}(a,b)$ can be iteratively calculated as*

$$s_{k+1}(a,b) = \frac{C}{|\mathcal{I}(a)||\mathcal{I}(b)|} \sum_{j\in\mathcal{I}(b)} \sum_{\Delta\in\mathscr{P}(\mathcal{I}(a))} Partial_{\Delta}^{s_k}(j), \tag{6}$$

where $Partial_{\Delta}^{s_k}(j)$ is defined as Eq.(4) *with $\mathcal{I}(a)$ replaced by Δ.*

Sketch of Proof: The proof follows immediately from (i) for two disjoint sets $\mathcal{A}$ and $\mathcal{B}$, $Partial^{s_k}_{\mathcal{A}}(j)+Partial^{s_k}_{\mathcal{B}}(j) = Partial^{s_k}_{\mathcal{A}\cup\mathcal{B}}(j)$, $\forall j$, and (ii) $\bigcup_{\Delta\in\mathscr{P}(\mathcal{I}(a))} = \mathcal{I}(a)$, $\forall a$. ■

The main idea in our approach is to share the common sub-summations among different partial sums, by precomputing the sub-summations $Partial^{s_k}_{\Delta}(\star)$ over $\Delta \in \mathscr{P}(\mathcal{I}(a))$ once, and caching them in a block fashion for later reuse, which can effectively avoid repeating duplicate sub-summations. As an example in Fig. 1c, when $\mathcal{I}(c)$ is partitioned as $\mathscr{P}(\mathcal{I}(c)) = \{\mathcal{I}(a), \{d\}\}$ with $\mathcal{I}(a) = \{b, g\}$, once computed, the sub-summations $Partial^{s_k}_{\mathcal{I}(a)}(\star)$ can be memorized and reused for computing $Partial^{s_k}_{\mathcal{I}(c)}(\star)$. In contrast, the existing method psum-SR [16] has to start from scratch to compute $Partial^{s_k}_{\mathcal{I}(a)}(\star)$ and $Partial^{s_k}_{\mathcal{I}(c)}(\star)$, independently, which is due to no reuse of common sub-summations.

The selection of a partition $\mathscr{P}(\mathcal{I}(a))$ for an in-neighbor set $\mathcal{I}(a)$ has a great impact on the performance of our approach. Troubles could be expected when a selected partition $\mathscr{P}(\mathcal{I}(a))$ is too coarse or too fine. For instance, if $\mathcal{I}(a)$ is taken to be a trivial partition of itself, *i.e.,* $\mathscr{P}(\mathcal{I}(a)) = \{\mathcal{I}(a)\}$ for every vertex a, Eq.(6) can be simplified to the conventional psum-SR iteration in Eq.(5). From this perspective, our approach is a generalization of psum-SR. On the other hand, if the partitions of $\mathcal{I}(a)$ get finer (*i.e.,* the size of $\Delta \in \mathscr{P}(\mathcal{I}(a))$ becomes smaller), there is a more likelihood of $Partial^{s_k}_{\Delta}(\star)$ with a high density of common sub-summations, but with a low cardinality on the similarity values to be clustered. An extreme example would be a discrete partition of $\mathcal{I}(a)$, *i.e.,* $\mathscr{P}(\mathcal{I}(a)) = \{\{x\}|x \in \mathcal{I}(a)\}$, where every block is a singleton vertex. In such a case, Eq.(6) would deteriorate to the naive iteration [11] in Eq.(2), which may be even worse than psum-SR. Thus, it is desirable to find the best partition $\mathscr{P}(\mathcal{I}(a))$ for each $\mathcal{I}(a)$ that has the largest and densest clumps of common vertices.

The problem of finding such optimal partitions to minimize the total cost of partial sums over different in-neighbor sets, referred to as *Optimal In-neighbors Partitioning* and denoted as OIP, can be formulated as follows:

Given a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, OIP is to find the optimal partition $\mathscr{P}(\mathcal{I}(a)) = \{\Delta^i_a \mid i = 1, \cdots, |\mathscr{P}(\mathcal{I}(a))|\}$ of each in-neighbor set $\mathcal{I}(a)$, $a \in \mathcal{V}$, for creating chunks Δ^i_a such that the total number of additions required for computing all the partial sums $Partial^{s_k}_{\mathcal{I}(a)}(\star)$ over every in-neighbor set $\mathcal{I}(a)$, $a \in \mathcal{V}$, is minimized by reusing the sub-summation results $Partial^{s_k}_{\Delta^i_a}(\star)$ over chunks Δ^i_a.

Proposition 2. *The OIP problem is NP-hard.*

Proof: We verify this by reducing the NP-complete *Ensemble Computation* (EC) problem [8, p.66] to a special case of the decision problem of OIP. The EC problem is defined as follows: Given a collection $\mathscr{C}$ of subsets of a finite set $\mathcal{A}$ and a positive integer J, EC is to decide whether there is a sequence $(z_1 = x_1 \cup y_1, \cdots, z_j = x_j \cup y_j)$ of $j \leq J$ union operations, where each x_i and y_i is either $\{a\}$ for some $a \in \mathcal{A}$ or z_p for some $p < i$, such that x_i and y_i are disjoint for $1 \leq i \leq j$ and such that for every subset $\mathcal{C} \in \mathscr{C}$ there is some z_i, $1 \leq i \leq j$, that is identical to $\mathcal{C}$. For each instance of EC, we construct the corresponding instance of the OIP decision problem by setting $\mathcal{A} = \{s_k(a, \star) \mid a \in \mathcal{V}\}$, $\mathscr{C} = \{Partial^{s_k}_{\mathcal{I}(a)}(\star) \mid a \in \mathcal{V}\}$, and an integer J to be the maximum number of required additions. Clearly, by converting union operations ($\cup$) of EC into additions ($+$), it follows that the OIP decision problem has a solution, *i.e.,* $\exists$ a sequence $(z_1 = x_1 + y_1, \cdots, z_j = x_j + y_j)$ of $j \leq J$ additions, if and only if there exists a sequence $(z_1 = x_1 \cup y_1, \cdots, z_j = x_j \cup y_j)$ of $j \leq J$ union operations for EC. Thus, the NP-completeness of the OIP decision problem follows immediately from the NP-completeness of EC. Also, the decision problem of OIP can be naturally converted into its corresponding optimization problem by imposing a bound on the number of additions to be optimized, namely, turning "whether there exists such a solution that can be done in fewer than J additions" into "minimize the number of additions". Hence, the OIP optimization problem is NP-hard due to the NP-completeness of its decision problem. ■

We next seek instead for a good heuristic method for OIP.

The basic idea is as follows. Consider a directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$. For every two in-neighbor sets $\mathcal{I}(a), \mathcal{I}(b)$ of vertices $a, b \in \mathcal{V}$, we first calculate *the transition cost* from $\mathcal{I}(a)$ to $\mathcal{I}(b)$, denoted by $\mathcal{TC}_{\mathcal{I}(a)\to\mathcal{I}(b)}$, as follows: [3]

$$\mathcal{TC}_{\mathcal{I}(a)\to\mathcal{I}(b)} \triangleq \min\{|\mathcal{I}(a) \ominus \mathcal{I}(b)|, |\mathcal{I}(b)| - 1\}, \qquad (7)$$

where $\ominus$ is the *symmetric difference* of two sets. [4] Thus, the value of $\mathcal{TC}_{\mathcal{I}(a)\to\mathcal{I}(b)}$ is actually the number of additions required to compute the partial sum $Partial^{s_k}_{\mathcal{I}(b)}(\star)$, given the partial sum $Partial^{s_k}_{\mathcal{I}(a)}(\star)$. Then, we construct a weighted digraph $\mathscr{G} = (\mathscr{V}, \mathscr{E})$ whose vertices correspond to the non-empty in-neighbor sets of $\mathcal{G}$, with an extra vertex corresponding to an empty set $\varnothing$, *i.e.,* $\mathscr{V} = \{\mathcal{I}(a) \mid a \in \mathcal{V}\} \cup \{\varnothing\}$. There is an edge from $\mathcal{I}(a)$ to $\mathcal{I}(b)$ in $\mathscr{G}$ if $|\mathcal{I}(a)| \leq |\mathcal{I}(b)|$. The weight of an edge $(\mathcal{I}(a), \mathcal{I}(b)) \in \mathscr{E}$ represents the transition cost $\mathcal{TC}_{\mathcal{I}(a)\to\mathcal{I}(b)}$. Finally, we find a minimum spanning tree of $\mathscr{G}$, denoted by $\mathscr{T}$, whose total transition cost is minimum. Henceforth, every edge $(\mathcal{I}(a), \mathcal{I}(b))$ in $\mathscr{T}$ implies the following: (i) $Partial^{s_k}_{\mathcal{I}(a)}(\star)$ should be computed prior to $Partial^{s_k}_{\mathcal{I}(b)}(\star)$ computation, which provides an optimized topological sort for efficiently computing all the partial sums. (ii) $\mathcal{I}(b)$ needs to be partitioned as $\mathcal{I}(b) \cap \mathcal{I}(a)$ and $\mathcal{I}(b) \backslash \mathcal{I}(a)$, meaning that the result of $Partial^{s_k}_{\mathcal{I}(a)}(\star)$ can be cached and shared with $Partial^{s_k}_{\mathcal{I}(b)}(\star)$ computation.

The following example illustrates how this idea works.

Example 2. Consider the network $\mathcal{G}$ in Fig. 1a, with the vertices and the corresponding non-empty in-neighbor sets

[3]Without loss of generality, only in the case of $|\mathcal{I}(a)| \leq |\mathcal{I}(b)|$, we need to compute $\mathcal{TC}_{\mathcal{I}(a)\to\mathcal{I}(b)}$. This is because we are interested only in the cost of computing $Partial^{s_k}_{\mathcal{I}(b)}(\star)$ by using the given $Partial^{s_k}_{\mathcal{I}(a)}(\star)$. Conversely, if utilizing the result of $Partial^{s_k}_{\mathcal{I}(b)}(\star)$ to compute $Partial^{s_k}_{\mathcal{I}(a)}(\star)$, for $|\mathcal{I}(a)| \leq |\mathcal{I}(b)|$, then we have to introduce the "subtraction" to undo the summation that we have already done, which is often an extra operation.

[4]The *symmetric difference* of two sets $\mathcal{A}$ and $\mathcal{B}$, denoted by $\mathcal{A} \ominus \mathcal{B}$, is the set of all elements of $\mathcal{A}$ or $\mathcal{B}$ which are not in both $\mathcal{A}$ and $\mathcal{B}$. Symbolically,

$$\mathcal{A} \ominus \mathcal{B} = (\mathcal{A}\backslash\mathcal{B}) \cup (\mathcal{B}\backslash\mathcal{A}).$$

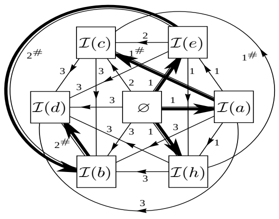
As an example in Fig 1c, given $\mathcal{I}(b) = \{g, e, f, i\}$ and $\mathcal{I}(d) = \{e, f, i, a\}$, we have $\mathcal{I}(b) \ominus \mathcal{I}(d) = \{g, a\}$.

vertex	$\mathcal{I}(\star)$
a	$\{b, g\}$
e	$\{f, g\}$
h	$\{b, d\}$
c	$\{b, d, g\}$
b	$\{f, g, e, i\}$
d	$\{f, a, e, i\}$

(a) In-neighbors in $\mathcal{G}$

	$\mathcal{I}(a)$	$\mathcal{I}(e)$	$\mathcal{I}(h)$	$\mathcal{I}(c)$	$\mathcal{I}(b)$	$\mathcal{I}(d)$
$\varnothing$	1	1	1	2	3	3
$\mathcal{I}(a)$		1	1	$1^{\#}$	3	3
$\mathcal{I}(e)$			1	2	$2^{\#}$	3
$\mathcal{I}(h)$				$1^{\#}$	3	3
$\mathcal{I}(c)$					3	3
$\mathcal{I}(b)$						$2^{\#}$

(b) Transition Costs (Edge Weights) in $\mathscr{G}$



(c) Minimum Spanning Tree $\mathscr{T}$ of $\mathscr{G}$

$$\begin{tikzcd}
① & ② & ③ \\
\varnothing \arrow[d, "1"] & \varnothing \arrow[d, "1"] & \varnothing \arrow[d, "1"] \\
\mathcal{I}(h) & \mathcal{I}(a) \arrow[d, "1^{\#}"] & \mathcal{I}(e) \arrow[d, "2^{\#}"] \\
 & \mathcal{I}(c) & \mathcal{I}(b) \arrow[d, "2^{\#}"] \\
 & & \mathcal{I}(d)
\end{tikzcd}$$

(d) Partial Sums Order

Fig. 2: Constructing a minimum spanning tree $\mathscr{T}$ to find an optimized topological sort for partial sums sharing in $\mathcal{G}$

depicted in Fig. 2a. We show how to find a decent ordering for partial sums computing and sharing in $\mathcal{G}$.

Firstly, we compute the transition cost of each pair of in-neighbor sets (along with an empty set $\varnothing$) in $\mathcal{G}$, by using Eq.(7). The results are shown in Fig. 2b, where each cell describes the transition cost from the in-neighbor set in the left most column to the in-neighbor set in the top line. For instance, the cell '$2^{\#}$' at row '$\mathcal{I}(e)$' column '$\mathcal{I}(b)$' shows that $\mathcal{TC}_{\mathcal{I}(e)\to\mathcal{I}(b)} = 2$. This cell is tagged with #, indicating that the partial sum $Partial^{s_k}_{\mathcal{I}(b)}(\star)$ can be computed from the memorized result of $Partial^{s_k}_{\mathcal{I}(e)}(\star)$ (rather than from scratch). This is because the transition cost 2 is, in essence, obtained from the 2 operations of symmetric difference (*i.e.*, $|\mathcal{I}(e) \ominus \mathcal{I}(b)| = |\{e, i\}| = 2$) in lieu of the 3 additions (*i.e.*, $|\mathcal{I}(b)| - 1 = 3$) *w.r.t.* Eq.(7). Note that the lower triangular part of the table in Fig. 2b remains empty since we are interested only in the cost $\mathcal{TC}_{\mathcal{I}(x)\to\mathcal{I}(y)}$ when $|\mathcal{I}(x)| \le |\mathcal{I}(y)|$.

Next, we build a weighted digraph $\mathscr{G}$ in Fig. 2c, with vertices corresponding to the non-empty in-neighbor sets (plus $\varnothing$) of $\mathcal{G}$ (which are in column '$\mathcal{I}(\star)$' of Fig. 2a), and edge weights to the transition costs. For instance, the weight of the edge ($\mathcal{I}(e)$,$\mathcal{I}(b)$) in $\mathscr{G}$ is associated with the cell '$2^{\#}$' at row '$\mathcal{I}(e)$' column '$\mathcal{I}(b)$' in Fig. 2b. Thus, every path in $\mathscr{G}$ yields a linear ordering of partial sums computation. More importantly, partial sums sharing may occur in the edges tagged with #. As an example, the path $\varnothing \xrightarrow{1} \mathcal{I}(e) \xrightarrow{2^{\#}} \mathcal{I}(b)$ in $\mathscr{G}$ shows that (i) $Partial^{s_k}_{\mathcal{I}(e)}(\star)$ is computed from scratch (from $\varnothing$) with 1 operation, and (ii) $Partial^{s_k}_{\mathcal{I}(b)}(\star)$ is obtained by reusing the result of $Partial^{s_k}_{\mathcal{I}(e)}(\star)$, involving 2 operations.

Finally, we find a directed minimum spanning tree $\mathscr{T}$ of $\mathscr{G}$, by starting from the vertex $\varnothing$, and choosing the cheapest path for partial sums computing and sharing, as depicted in bold edges in Fig. 2c. Consequently, using depth-first search (DFS), we can obtain 3 paths from $\mathscr{T}$ for partial sums optimization, as shown in Fig. 2d. ■

Using this idea, we can identify the moderate partitions of each in-neighbor set in $\mathcal{G}$, with large and dense chunks for sub-summations sharing. Such partitions are not optimal, but can, in practice, achieve better performances than psum-SR. Proposition 3 shows the correctness.

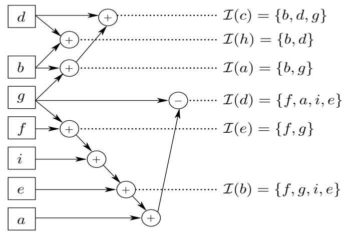
Proposition 3. *Given two distinct non-empty in-neighbor sets $\mathcal{I}(a)$ and $\mathcal{I}(b)$, and a partial sum $Partial^{s_k}_{\mathcal{I}(a)}(\star)$, if $|\mathcal{I}(a) \ominus \mathcal{I}(b)| < |\mathcal{I}(b)| - 1$, then we have the following:*

(i) $\mathcal{I}(b)$ can be partitioned as

$$\mathcal{I}(b) = (\mathcal{I}(b) \cap \mathcal{I}(a)) \cup (\mathcal{I}(b) \backslash \mathcal{I}(a)). \tag{8}$$

	$\mathscr{P}(\star)$
$\mathcal{I}(a)$	$\{\{b, g\}\}$
$\mathcal{I}(e)$	$\{\{f, g\}\}$
$\mathcal{I}(h)$	$\{\{b, d\}\}$
$\mathcal{I}(c)$	$\{\mathcal{I}(a), \{d\}\}$
$\mathcal{I}(b)$	$\{\mathcal{I}(e), \{e, i\}\}$
$\mathcal{I}(d)$	$\{\mathcal{I}(b)\backslash\{g\}, \{a\}\}$

(a) Partitions of $\mathcal{I}(\star)$ in $\mathcal{G}$


d
b
g
f
i
e
a
$\mathcal{I}(c) = \{b, d, g\}$
$\mathcal{I}(h) = \{b, d\}$
$\mathcal{I}(a) = \{b, g\}$
$\mathcal{I}(d) = \{f, a, i, e\}$
$\mathcal{I}(e) = \{f, g\}$
$\mathcal{I}(b) = \{f, g, i, e\}$

(b) Hierarchical Clustering Dendrogram

Fig. 3: In-neighbor sets partitioning and hierarchical clustering

(ii) The partial sum $Partial^{s_k}_{\mathcal{I}(b)}(\star)$ can be computed from the cached result of $Partial^{s_k}_{\mathcal{I}(a)}(\star)$ as follows

$$Partial^{s_k}_{\mathcal{I}(b)}(y) = Partial^{s_k}_{\mathcal{I}(a)}(y) - \sum_{x \in \mathcal{I}(a)\backslash\mathcal{I}(b)} s_k(x, y) + \sum_{x \in \mathcal{I}(b)\backslash\mathcal{I}(a)} s_k(x, y), \quad (y \in \mathcal{V}) \tag{9}$$

with $|\mathcal{I}(a) \ominus \mathcal{I}(b)|$ operations being performed.

Sketch of Proof: The proof of Eq.(8) is trivial, whereas the proof of Eq.(9) is based on (i) $\mathcal{B} = (\mathcal{A}\backslash(\mathcal{A}\backslash\mathcal{B})) \cup (\mathcal{B}\backslash\mathcal{A})$, (ii) $Partial^{s_k}_{\mathcal{A}\backslash\mathcal{B}}(j) = Partial^{s_k}_{\mathcal{A}}(j) - Partial^{s_k}_{\mathcal{B}\cap\mathcal{A}}(j)$, $\forall j$. ■

Example 3. Recall the network $\mathcal{G}$ in Fig. 1a, along with the optimized ordering of partial sums in Fig. 2d. We show how to identify the partition of each in-neighbor set in $\mathcal{G}$ for partial sums sharing. For instance, consider the path $\varnothing \xrightarrow{1} \mathcal{I}(a) \xrightarrow{1^{\#}} \mathcal{I}(c)$ in Fig. 2d. We have the following.

(i) The first edge $\varnothing \xrightarrow{1} \mathcal{I}(a)$ implies that $Partial^{s_k}_{\mathcal{I}(a)}(\star)$ need to be computed from scratch since the starting point of this edge is $\varnothing$. Thus, $\mathcal{I}(a)$ has only one partition of itself.

(ii) The second edge $\mathcal{I}(a) \xrightarrow{1^{\#}} \mathcal{I}(c)$ suggests that $\mathcal{I}(c)$ can be partitioned, by using Eq.(8), as

$$\mathcal{I}(c) = (\mathcal{I}(c) \cap \mathcal{I}(a)) \cup (\mathcal{I}(c)\backslash\mathcal{I}(a)) = \mathcal{I}(a) \cup \{d\}.$$

Hence, $Partial^{s_k}_{\mathcal{I}(c)}(\star)$ can be obtained from the memorized result of $Partial^{s_k}_{\mathcal{I}(a)}(\star)$ via Eq.(9) as follows:

$$Partial^{s_k}_{\mathcal{I}(c)}(x) = Partial^{s_k}_{\mathcal{I}(a)}(x) + s_k(d, x). \quad (x \in \mathcal{V})$$

We repeat these steps for the rest of two paths in Fig. 2d. Finally, we get all the partitions of in-neighbor sets in $\mathcal{G}$, as shown in Fig. 3a. Accordingly, the resultant accumulation of reusable partial sums is visualized in Fig. 3b, in which a letter with a box denotes a vertex, and a symbol with a circle an operator. For example, '$\boxed{d} \oplus \boxed{b} \cdots \mathcal{I}(h)$' means that $s_k(d, \star)$ and $s_k(b, \star)$ are added to yield $Partial^{s_k}_{\mathcal{I}(h)}(\star)$. ■

B. Use In-neighbor Set Partitions for Outer Sums Sharing

After the partitions of in-neighbor sets have been identified for *(inner)* partial sums sharing, optimization approaches in this subsection allow *outer* partial sums sharing for further speeding up the computation of SimRank.

To avoid ambiguity, we shall refer to the sums *w.r.t.* the index i in Eq.(4) as *(inner) partial sums*, and the sums *w.r.t.* the index j in Eq.(5) as *outer partial sums*.

Our key observation is as follows. Recall from Eq.(5) that, given the memorized results of partial sums $Partial^{s_k}_{\mathcal{I}(a)}(\star)$, the existing algorithm psum-SR for computing $s_k(a,b)$ is to sum up $Partial^{s_k}_{\mathcal{I}(a)}(y)$, one by one, over all $y \in \mathcal{I}(b)$. Such a process can be pictorially depicted in the left part of Fig. 1c, in which each horizontal bar represents a partial sum over $\mathcal{I}(a)$. In order to compute $s(a,b)$, we need to add up the horizontal bars (*i.e.,* the partial sums) in the first four rows. However, while computing $s(a,c)$ by adding up the horizontal bars in the last four rows, we observe that the three horizontal bars at rows 'e','f','i' may suffer from repetitive additions. As another example in the right part of Fig. 1c, for computing $s(b,c)$ and $s(d,c)$, the sum of the three horizontal bars at rows 'e','f','i' is again a repeated operation. As such, the major problem of Eq.(5) is the one-by-one fashion in which the partial sums $Partial^{s_k}_{\mathcal{I}(a)}(y)$ for $y \in \mathcal{I}(b)$ are added together.

Our main idea in optimizing Eq.(5) is to split $\mathcal{I}(b)$ into several chunks Δ^i_b first, such that

$$\mathscr{P}(\mathcal{I}(b)) = \{\Delta^i_b \mid i = 1, \cdots, |\mathscr{P}(\mathcal{I}(b))|\},$$

and then add up the cached results of partial sums in a chunk-by-chunk fashion for computing $s_{k+1}(a,b)$ as follows:

$$s_{k+1}(a,b) = \frac{C}{|\mathcal{I}(a)||\mathcal{I}(b)|} \sum_{\Delta^i_b \in \mathscr{P}(\mathcal{I}(b))} OuterPartial^{\mathcal{I}(a),s_k}_{\Delta^i_b} \quad (10)$$

with

$$OuterPartial^{\mathcal{I}(a),s_k}_{\Delta^i_b} \triangleq \sum_{j \in \Delta^i_b} Partial^{s_k}_{\mathcal{I}(a)}(j).$$

In contrast with Eq.(5), our method in Eq.(10) can eliminate the redundancy among different outer partial sums. Once computed, the outer partial sum $OuterPartial^{\mathcal{I}(a),s_k}_{\Delta^i_b}$ is memorized and can be reused later without recalculation again. As an example in Fig. 1c, suppose $\mathcal{I}(b)$ and $\mathcal{I}(d)$ are split into

$$\mathcal{I}(b) = \{g\} \cup \{e,f,i\}, \quad \mathcal{I}(d) = \{e,f,i\} \cup \{a\},$$

the outer partial sum $OuterPartial^{\mathcal{I}(a),s_k}_{\{e,f,i\}}$ is computed only once and can be reused in both $s_{k+1}(a,b)$ and $s_{k+1}(a,d)$ computation.

The problem of finding an ideal partition $\mathscr{P}(\mathcal{I}(b))$ of $\mathcal{I}(b)$ for maximal sharing outer partial sums is still NP-hard, and its proof is the same as that of OIP in Proposition 2. Thus, the partitioning techniques for (inner) partial sums sharing in Subsection III-A can be applied in a similar way to optimize outer partial sums sharing. In other words, the partitions of in-neighbor sets in Eq.(8) for (inner) partial sums sharing, once identified, can be reused later for outer partial sums sharing. The correctness is proved in Proposition 4.

vertex	$Partial^{s_k}_{\mathcal{I}(x)}(y)$			$OuterPartial^{\mathcal{I}(x),s_k}_{\mathcal{I}(z)}$		$s_{k+1}(x,z)$	
x	$y=b$	$y=g$	$y=d$	$z=a$	$z=c$	$z=a$	$z=c$
a	1	1	0.11	2	2.11	1	0.21
e	0	1	0	1	1	0.15	0.1
h	1.11	0	1.11	1.11	2.22	0.17	0.22
c	1.11	1	1.11	2.11	3.22	0.21	1
b	0.15	1	0.08	1.15	1.23	0.09	0.06
d	0.23	0	0.08	0.23	0.31	0.02	0.02

Fig. 4: Computing $s_{k+1}(x,a)$ and $s_{k+1}(x,c)$, $\forall x \in \mathcal{V}$, by using outer sums sharing $(k = 2$ and $C = 0.6)$

Proposition 4. *Given two non-empty in-neighbor sets $\mathcal{I}(b)$ and $\mathcal{I}(d)$, an outer partial sum $OuterPartial^{\mathcal{I}(a),s_k}_{\mathcal{I}(b)}$, and (inner) partial sums $Partial^{s_k}_{\mathcal{I}(a)}(\star)$, if $|\mathcal{I}(b) \ominus \mathcal{I}(d)| < |\mathcal{I}(d)| - 1$, then we have the following:*

(i) The outer partial sums $OuterPartial^{\mathcal{I}(a),s_k}_{\mathcal{I}(d)}$ can be computed from the memorized results of $OuterPartial^{\mathcal{I}(a),s_k}_{\mathcal{I}(b)}$, $\forall a \in \mathcal{V}$, as follows:

$$OuterPartial^{\mathcal{I}(a),s_k}_{\mathcal{I}(d)} = OuterPartial^{\mathcal{I}(a),s_k}_{\mathcal{I}(b)} - \\ - \sum_{x \in \mathcal{I}(b) \setminus \mathcal{I}(d)} Partial^{s_k}_{\mathcal{I}(a)}(x) + \sum_{x \in \mathcal{I}(d) \setminus \mathcal{I}(b)} Partial^{s_k}_{\mathcal{I}(a)}(x), \quad \forall a \in \mathcal{V}$$

with $|\mathcal{I}(b) \ominus \mathcal{I}(d)|$ operations being performed.

(ii) $s_{k+1}(a,d)$, $\forall a \in \mathcal{V} \setminus \{d\}$ can be computed as

$$s_{k+1}(a,d) = \tfrac{C}{|\mathcal{I}(a)||\mathcal{I}(d)|} OuterPartial^{\mathcal{I}(a),s_k}_{\mathcal{I}(d)}, \quad \forall a \in \mathcal{V} \setminus \{d\}. \quad (11)$$

(The proof is similar to Proposition 3. We omit it here.)

Example 4. Recall the network $\mathcal{G}$ in Fig. 1a, with the (inner) partial sums sharing dendrogram in Fig. 3b. Suppose $Partial^{s_k}_{\mathcal{I}(x)}(\star)$, $\forall x \in \mathcal{V}$, have been pre-computed based on Example 3, as depicted in part in the first four columns of Fig. 4. We show how to compute $s_{k+1}(x,a)$ and $s_{k+1}(x,c)$, $\forall x \in \mathcal{V}$, by using outer partial sums sharing.

Firstly, for each non-empty in-neighbor set $\mathcal{I}(x)$, we compute $OuterPartial^{\mathcal{I}(x),s_k}_{\mathcal{I}(a)}$ and $OuterPartial^{\mathcal{I}(x),s_k}_{\mathcal{I}(c)}$, $\forall x \in \mathcal{V}$, from the cached results of $Partial^{s_k}_{\mathcal{I}(x)}(\star)$. In light of the clustering dendrogram in Fig. 3b, we notice that the item '$\boxed{b} \oplus \boxed{g} \cdots \mathcal{I}(a)$', which, in the context of *outer* partial sums, can be reinterpreted as "adding up the (inner) partial sums $Partial^{s_k}_{\mathcal{I}(x)}(b)$ and $Partial^{s_k}_{\mathcal{I}(x)}(g)$ to yield the outer partial sums $OuterPartial^{\mathcal{I}(x),s_k}_{\mathcal{I}(a)}$, for all $x \in \mathcal{V}$". Thus, we have

$$OuterPartial^{\mathcal{I}(x),s_k}_{\mathcal{I}(a)} = \sum_{y \in \{b,g\}} Partial^{s_k}_{\mathcal{I}(x)}(y). \quad (\forall x \in \mathcal{V})$$

For instance, $OuterPartial^{\mathcal{I}(b),s_k}_{\mathcal{I}(a)} = 0.15 + 1 = 1.15$, for $x = b$, as illustrated in row 'b' of Fig. 4.

Similarly, the item '$\mathcal{I}(a) \oplus \boxed{d} \cdots \mathcal{I}(c)$' in Fig. 3b implies that $OuterPartial^{\mathcal{I}(x),s_k}_{\mathcal{I}(c)}$, $\forall x \in \mathcal{V}$, can be calculated from the cached results of $OuterPartial^{\mathcal{I}(x),s_k}_{\mathcal{I}(a)}$ via Eq.(10) as

$$OuterPartial^{\mathcal{I}(x),s_k}_{\mathcal{I}(c)} = OuterPartial^{\mathcal{I}(x),s_k}_{\mathcal{I}(a)} \\ + Partial^{s_k}_{\mathcal{I}(x)}(d), \quad (\forall x \in \mathcal{V})$$

e.g., $OuterPartial^{\mathcal{I}(b),s_k}_{\mathcal{I}(c)} = 1.15 + 0.08 = 1.23$, for $x = b$. The rest of the results are shown in columns 5-6 of Fig. 4.

```
Algorithm 1: OIP-SR $(\mathcal{G}, C, K)$
Input  : graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, damping factor $C$, iteration $K$.
Output: SimRank scores $s_K(\star,\star)$.
1  construct a transitional MST $\mathscr{T} \leftarrow$ DMST-Reduce $(\mathcal{G})$;
2  initialize $s_0(x,y) \leftarrow \{ {}^{1,\ x=y}_{0,\ x\neq y} \quad \forall x,y \in \mathcal{V}$
3  for $k \leftarrow 0,1,\cdots,K-1$ do
4      foreach vertex $u \in \mathcal{O}(\#)$ in the MST $\mathscr{T}$ do
5          foreach vertex $y \in \mathcal{V}$ in $\mathcal{G}$ do
6              $Partial^{s_k}_{\mathcal{I}(u)}(y) \leftarrow \sum_{x\in\mathcal{I}(u)} s_k(x,y)$ ;
7          $s_{k+1}(u,\star) \leftarrow$ OP $(\mathscr{T}, \mathcal{G}, u, C, k, Partial^{s_k}_{\mathcal{I}(u)}(\star))$ ;
8          while $\mathcal{O}(u) \neq \varnothing$ do
9              $v \leftarrow \mathcal{O}(u)$ ;
10             foreach vertex $y \in \mathcal{V}$ in $\mathcal{G}$ do
11                 $Partial^{s_k}_{\mathcal{I}(v)}(y) \leftarrow Partial^{s_k}_{\mathcal{I}(u)}(y) - \sum_{x\in\mathcal{I}(u)\setminus\mathcal{I}(v)} s_k(x,y) + \sum_{x\in\mathcal{I}(v)\setminus\mathcal{I}(u)} s_k(x,y)$ ;
12             $s_{k+1}(v,\star) \leftarrow$ OP$(\mathscr{T}, \mathcal{G}, v, C, k, Partial^{s_k}_{\mathcal{I}(v)}(\star))$;
13             $u \leftarrow v$ ;
14         foreach vertex $y \in \mathcal{V}$ in $\mathcal{G}$ do
15             free $Partial^{s_k}_{\mathcal{I}(u)}(y)$ ;
16             while $\mathcal{O}(u) \neq \varnothing$ do
17                 $v \leftarrow \mathcal{O}(u)$,  free $Partial^{s_k}_{\mathcal{I}(v)}(y)$,  $u \leftarrow v$ ;
18 return $s_K(\star,\star)$ ;
```

Then, using Eq.(11), we obtain $s_{k+1}(x,a)$ and $s_{k+1}(x,c)$, $\forall x \in \mathcal{V}$, from the cached results of $OuterPartial^{\mathcal{I}(x),s_k}_{\mathcal{I}(a)}$ and $OuterPartial^{\mathcal{I}(x),s_k}_{\mathcal{I}(c)}$. For example, in row 'b' of Fig. 4,

$$s_{k+1}(b,a) = \tfrac{0.6}{2\times 4} \times 1.15 = 0.09, \quad (x=b)$$

$$s_{k+1}(b,c) = \tfrac{0.6}{3\times 4} \times 1.23 = 0.06. \quad (x=b)$$

The remainder of the similarities are depicted in the last two columns of Fig. 4. ■

C. An Algorithm for Computing SimRank

We next present a complete algorithm for efficiently computing SimRank, by integrating the aforementioned techniques of inner and outer partial sums sharing.

The main result of this subsection is the following.

Proposition 5. *For any graph $\mathcal{G}$, it is in $O(dn^2 + Kd'n^2)$ time and $O(n)$ intermediate memory to compute SimRank similarities of all pairs of vertices for K iterations, where d is the average vertex in-degree of $\mathcal{G}$, and $d' \le d$.*

Note that d' is affected by the overlapped area size among different in-neighbor sets in $\mathcal{G}$. Typically, d' is much smaller than d as in-neighbor sets in $\mathcal{G}$ may have many vertices in common in real networks. That is, our approach of partial sums sharing can compute SimRank more efficiently than psum-SR in practice, as opposed to the $O(Kdn^2)$-time of the conventional counterpart via separate partial sums over each in-neighbour set in $\mathcal{G}$. Even in the extreme case when all in-neighbor sets in $\mathcal{G}$ are pair-wise disjoint, our method can retain the same complexity bound of psum-SR in the worst case.

```
Procedure DMST-Reduce($\mathcal{G}$)
Input  : graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$.
Output: transitional MST $\mathscr{T}$.
1  initialize $\mathscr{V} \leftarrow \mathcal{V} \cup \{\#\}$,  $\mathscr{E} \leftarrow \varnothing$ ;
2  sort the vertices of $\mathcal{G}$ into non-decreasing order by in-degree ;
3  initialize $\mathcal{U} \leftarrow \mathcal{V}$ ;
4  foreach vertex $a \in \mathcal{V}$ in $\mathcal{G}$, taken in sorted order do
5      $\mathcal{U} \leftarrow \mathcal{U} \backslash \{a\}$ ;
6      foreach vertex $b \in \mathcal{U}$ in $\mathcal{G}$, taken in sorted order do
7          $\mathscr{E} \leftarrow \mathscr{E} \cup \{(a,b)\}$ ;
8          assign a weight $w$ to the edge $(a,b)$ of $\mathscr{E}$ :
               $w(a,b) \leftarrow \min\{|\mathcal{I}(a) \ominus \mathcal{I}(b)|, |\mathcal{I}(b)|-1\}$ ;
9  find the MST $\mathscr{T}$ of the graph $\mathscr{G} = (\mathscr{V}, \mathscr{E}, w)$ :
       $\mathscr{T} \leftarrow$ Directed-MST $(\mathscr{G}, \#, w)$ ;
10 return $\mathscr{T}$ ;
```

We next prove Proposition 5 by providing an algorithm for SimRank computation, with the desired complexity bound.

Algorithm. The algorithm, referred to as OIP-SR, is shown in Algorithm 1. Given $\mathcal{G}$, a damping factor C, and the total iteration number K, it returns $s_K(\star,\star)$ of all pairs of vertices.

In the sequel, we shall abuse the notation $\mathcal{O}(v)$ to denote the out-neighbor set of vertex v.

The algorithm OIP-SR works as follows. (1) It first invokes procedure DMST-Reduce to identify the topological sort based on a minimum spanning tree $\mathscr{T}$ for computing partial sums (line 1). (2) For each iteration k, OIP-SR checks each path in $\mathscr{T}$, starting from the root node # as follows. (a) For the first edge $(\#, u)$ in each path, OIP-SR computes $Partial^{s_k}_{\mathcal{I}(u)}(\star)$ from scratch (lines 5-6), and then invokes procedure OP to compute $s_{k+1}(u,\star)$ by outer partial sums sharing (line 7). (b) For other edges (u,v) in each path, OIP-SR computes $Partial^{s_k}_{\mathcal{I}(v)}(\star)$ from the result of $Partial^{s_k}_{\mathcal{I}(u)}(\star)$ memorized earlier (lines 10-11), and gets $s_{k+1}(v,\star)$ by invoking procedure OP of outer partial sums sharing (line 12). This process repeats until all edges in every path have been traversed, and OIP-SR frees the memorized results of the partial sums generated from each path (lines 14-17). (3) The loop will continue to iterate until k reaches K, and OIP-SR returns all the similarities $s_K(\star,\star)$ (line 18).

Procedure DMST-Reduce. Given a graph $\mathcal{G}$, the procedure returns a minimum spanning tree $\mathscr{T}$ as a topological sort for computing partial sums. First, it builds a weighed graph $\mathscr{G}$, whose edge weights are the transition costs of all pairs of vertices (plus a special # denoting 'the root node') in $\mathcal{G}$ (lines 1-8). Then, it runs an algorithm [7] to find a directed MST $\mathscr{T}$ of $\mathscr{G}$ (starting from vertex #), which is returned as the final result (lines 9-10).

Procedure OP. This procedure adopts a similar paradigm of OIP-SR for outer partial sums sharing. OP takes as input a topological sort $\mathscr{T}$, a graph $\mathcal{G}$, a vertex u, a damping factor C, iteration k, and the cached partial sums $Partial^{s_k}_{\mathcal{I}(u)}(\star)$. It returns the similarities $s_{k+1}(u,\star)$.

The procedure OP runs in three phases for each path that starts from the root # of the tree $\mathscr{T}$. (a) For the first edge $(\#, w)$ of each path, OP needs to start from scratch to

Procedure OP$(\mathscr{T}, \mathcal{G}, u, C, k, Partial_{\mathcal{I}(u)}^{s_k}(\star))$

```
Input : transitional MST $\mathscr{T}$, graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$,
        vertex $u$, damping factor $C$,
        iteration $k$, partial sums $Partial_{\mathcal{I}(u)}^{s_k}(\star)$.
Output: SimRank scores $s_{k+1}(u, \star)$.
1  foreach vertex $w \in \mathcal{O}(\#)$ in the MST $\mathscr{T}$ do
2      $OuterPartial_{\mathcal{I}(w)}^{\mathcal{I}(u),s_k} \leftarrow \sum_{y \in \mathcal{I}(w)} Partial_{\mathcal{I}(u)}^{s_k}(y)$ ;
3      if $u = w$ then $s_{k+1}(u, w) \leftarrow 1$ ;
4      else if $\mathcal{I}(u) = \varnothing$ or $\mathcal{I}(w) = \varnothing$ then $s_{k+1}(u, w) \leftarrow 0$ ;
5      else $s_{k+1}(u, w) \leftarrow \frac{C}{|\mathcal{I}(u)||\mathcal{I}(w)|} OuterPartial_{\mathcal{I}(w)}^{\mathcal{I}(u),s_k}$ ;
6      while $\mathcal{O}(w) \neq \varnothing$ do
7          $z \leftarrow \mathcal{O}(w)$ ;
8          $OuterPartial_{\mathcal{I}(z)}^{\mathcal{I}(u),s_k} \leftarrow OuterPartial_{\mathcal{I}(w)}^{\mathcal{I}(u),s_k} - \sum_{y \in \mathcal{I}(w) \setminus \mathcal{I}(z)} Partial_{\mathcal{I}(u)}^{s_k}(y) + \sum_{y \in \mathcal{I}(z) \setminus \mathcal{I}(w)} Partial_{\mathcal{I}(u)}^{s_k}(y)$ ;
9          if $u = z$ then $s_{k+1}(u, z) \leftarrow 1$ ;
10         else if $\mathcal{I}(u) = \varnothing$ or $\mathcal{I}(z) = \varnothing$ then $s_{k+1}(u, z) \leftarrow 0$ ;
11         else $s_{k+1}(u, z) \leftarrow \frac{C}{|\mathcal{I}(u)||\mathcal{I}(z)|} OuterPartial_{\mathcal{I}(z)}^{\mathcal{I}(u),s_k}$ ;
12         $w \leftarrow z$ ;
13     free $OuterPartial_{\mathcal{I}(w)}^{\mathcal{I}(u),s_k}$ ;
14     while $\mathcal{O}(w) \neq \varnothing$ do
15         $z \leftarrow O(w)$ , free $OuterPartial_{\mathcal{I}(z)}^{\mathcal{I}(u),s_k}$ , $w \leftarrow z$ ;
16 return $s_{k+1}(u, \star)$ ;
```

calculate $OuterPartial_{\mathcal{I}(w)}^{\mathcal{I}(u),s_k}$ (line 2) and $s_{k+1}(u, w)$ (lines 3-5) from the memorized $Partial_{\mathcal{I}(u)}^{s_k}(\star)$. (b) For other edges (w, z) in each path, OP obtains $OuterPartial_{\mathcal{I}(z)}^{\mathcal{I}(u),s_k}$ from the cached result of $OuterPartial_{\mathcal{I}(w)}^{\mathcal{I}(u),s_k}$ (line 8), and then computes $s_{k+1}(u, z)$ (lines 9-11). The loop continues until all edges in the path have been visited. (c) OP releases the memorized results of all the outer partial sums which are generated by each path (lines 13-15). The whole process repeats until all the paths in $\mathscr{T}$ have been processed, and returns $s_{k+1}(u, \star)$ (line 16).

Correctness. (i) Algorithm OIP-SR correctly computes the similarities $s_k(u, v)$ in $\mathcal{G}$ for each vertex pair (u, v). One can verify that after the **foreach** loops (lines 5-6 and lines 10-11), for every $u \in \mathscr{T}$, $Partial_{\mathcal{I}(u)}^{s_k}(\star)$ and $OuterPartial_{\mathcal{I}(\star)}^{\mathcal{I}(u),s_k}$ are memorized, and the similarities $s_{k+1}(u, \star)$ are computed. (ii) The partial sums computed by our algorithm are indeed *optimized* because while computing $Partial_{\mathcal{I}(u)}^{s_k}(\star)$ and $OuterPartial_{\mathcal{I}(\star)}^{\mathcal{I}(u),s_k}$ for each vertex u, we allow the common parts of partial sums to be recomputed as fewer as possible by virtue of a minimum spanning tree $\mathscr{T}$; in particular, the partial sums sharing would definitely happen in every path of $\mathscr{T}$ for a graph with $|\bigcup_{v \in \mathcal{V}} \mathcal{I}(v)|$ less than $\sum_{v \in \mathcal{V}} |\mathcal{I}(v)|$.

Complexity. OIP-SR consists of two phases: (i) building an MST $\mathscr{T}$ (line 1), and (ii) computing similarities (lines 2-18).

(i) The procedure DMST-Reduce is used for finding a directed MST $\mathscr{T}$, which is bounded by $O(dn^2)$ time and $O(n)$ space. It includes (a) $O(n \log n)$ time and $O(n)$ space for sorting vertices in $\mathcal{G}$ by in-degree (line 2), (b) $O(d)$ time and $O(2d)$ space for computing the transitional cost for a single edge (a, b) in $\mathscr{E}$, being $O(\frac{dn^2}{2})$ time for all edges in $\mathscr{E}$ (lines 4-8), and (c) $O(m \log n)$ time and $O(n)$ space for finding the MST $\mathscr{T}$ of $\mathcal{G}$ [7].

(ii) For each iteration, OIP-SR uses $\mathscr{T}$ rooted at # to compute similarities in $\mathcal{G}$. Note that $|\mathcal{O}(\#)|$ paths in $\mathscr{T}$ are used for calculating partial sums over all in-neighbour sets of $\mathcal{G}$. Therefore, for completing a single path of average length $\frac{n}{|\mathcal{O}(\#)|}$, the complexity required for computing the partial sums, for the first edge of the path, is $O(nd)$ time and $O(n)$ space (lines 5-6); the complexity required, apart from the first edge of the path, is $O(\frac{n}{|\mathcal{O}(\#)|} \cdot n \cdot d_\ominus)$ time and $O(n)$ space, with $d_\ominus \triangleq \text{avg}_{(u,v) \in \mathscr{T}} |\mathcal{I}(u) \ominus \mathcal{I}(v)|$ (lines 8-13). It follows that the total complexity bound in this phase is $O(K(|\mathcal{O}(\#)| \cdot nd + n^2 \cdot d_\ominus))$ time and $O(n)$ space for K iterations. Since $d_\ominus \ll d$ and $|\mathcal{O}(\#)| \ll n$, such a time complexity bound is far less than $O(Kdn^2)$.

Combining (i) and (ii), the total complexity of OIP-SR is $O(dn^2 + K(|\mathcal{O}(\#)| \cdot nd + n^2 \cdot d_\ominus))$ time and $O(n)$ space.

IV. Exponential Rate of Convergence for SimRank Iterations

For a desired accuracy ϵ, the existing paradigm (via Eq.(2)) for computing SimRank needs $K = \lceil \log_C \epsilon \rceil$ iterations [16]. In this section, we introduce a new notion of SimRank that is based on a matrix differential equation, which can significantly reduce the number of iterations for attaining the accuracy ϵ while fairly preserving the relative order of SimRank.

The main idea in our approach is to replace the geometric sum of the conventional SimRank by an exponential sum that provides more rapid rate of convergence. We start by expanding the conventional SimRank matrix form (in Eq.(3))

$$\mathbf{S} = C \cdot (\mathbf{Q} \cdot \mathbf{S} \cdot \mathbf{Q}^T) + (1 - C) \cdot \mathbf{I}_n,$$

as a power series:

$$\mathbf{S} = (1 - C) \cdot \sum_{i=0}^{\infty} C^i \cdot \mathbf{Q}^i \cdot (\mathbf{Q}^T)^i, \tag{12}$$

where we notice that the coefficient for each term in the summation makes a geometric sequence $\{1, C, C^2, \cdots\}$. For this expansion form, the effect of damping factor C^i in the summation is to reduce the contribution of long paths relative to short ones. That is, the conventional SimRank measure considers two vertices to be more similar if they have more paths of short length between them. Following this intuition, we observe that there is an opportunity to speed up the asymptotic rate of convergence for SimRank iterations, if we allow a slight (and with hindsight sensible) modification of Eq.(12) as follows:

$$\hat{\mathbf{S}} = e^{-C} \cdot \sum_{i=0}^{\infty} \frac{C^i}{i!} \cdot \mathbf{Q}^i \cdot (\mathbf{Q}^T)^i, \tag{13}$$

Comparing Eq.(12) with Eq.(13), we notice that $\hat{\mathbf{S}}$ is just an exponential sum rather than $\mathbf{S}$ that is a geometric sum. Since the exponential sum converges more rapidly, such a modification can speed up the computation of SimRank. In addition, the modified coefficient for each term in the summation of Eq.(13) that yields the exponential sequence $\{1, \frac{C}{1!}, \frac{C^2}{2!}, \cdots\}$

still obeys the intuition of the conventional counterpart, that is, the efficacy of damping factor $\frac{C^i}{i!}$ is to reduce the contribution of long paths relative to short ones.

With the modified notion of SimRank in Eq.(13), we now need to define an Eq.(3)-like recurrence for $\hat{\mathbf{S}}$.

Definition 2. *Let $\hat{\mathbf{S}}(t)$ be a matrix function w.r.t. a scalar t. The* matrix differential form of SimRank *is defined to be $\hat{\mathbf{S}} \triangleq \hat{\mathbf{S}}(t)|_{t=C}$ such that $\hat{\mathbf{S}}(t)$ satisfies the following matrix differential equation:*

$$\frac{d\hat{\mathbf{S}}(t)}{dt} = \mathbf{Q} \cdot \hat{\mathbf{S}}(t) \cdot \mathbf{Q}^T, \qquad \hat{\mathbf{S}}(0) = e^{-C} \cdot \mathbf{I}_n. \tag{14}$$

Note that the solution of Eq.(14) is unique since the initial condition $\hat{\mathbf{S}}(0) = e^{-C} \cdot \mathbf{I}_n$ is specified. Based on Definition 2, it is crucial to verify that $\hat{\mathbf{S}}$ (in Eq.(13)) is the solution to Eq.(14). Proposition 6 shows the correctness.

Proposition 6. *The matrix differential form of SimRank in Eq.*(14) *has an exact solution $\hat{\mathbf{S}}$ given in Eq.*(13).

Proof: We shall prove this by plugging $\hat{\mathbf{S}}(t) = A \cdot (\mathbf{I}_n + \sum_{i=1}^{\infty} \frac{t^i}{i!} \cdot \mathbf{Q}^i \cdot (\mathbf{Q}^T)^i)$, with an arbitrary constant A, into the SimRank differential formula Eq.(14):

$$\begin{aligned}\frac{\mathrm{d}\hat{\mathbf{S}}(t)}{\mathrm{d}t} &= A \cdot \sum_{i=1}^{\infty} \frac{\mathrm{d}}{\mathrm{d}t} \left(\frac{t^i}{i!} \cdot \mathbf{Q}^i \cdot (\mathbf{Q}^T)^i \right) \\ &= A \cdot \sum_{i=1}^{\infty} \frac{t^{i-1}}{(i-1)!} \cdot \mathbf{Q}^i \cdot (\mathbf{Q}^T)^i = \mathbf{Q} \cdot \hat{\mathbf{S}}(t) \cdot \mathbf{Q}^T,\end{aligned}$$

where the first equality holds because $\left\| \frac{t^i}{i!} \cdot \mathbf{Q}^i \cdot (\mathbf{Q}^T)^i \right\|_{\max} \le \frac{t^i}{i!}$, and the series $\sum_{i=1}^{\infty} \frac{t^i}{i!}$ converges uniformly on $t \in [0, C]$.

Therefore, we have verified that the solution to Eq.(14) takes the form $\hat{\mathbf{S}}(t) = A \cdot \left(\mathbf{I}_n + \sum_{i=1}^{\infty} \frac{t^i}{i!} \cdot \mathbf{Q}^i \cdot (\mathbf{Q}^T)^i \right)$.

To find A, let $t = 0$ and $\hat{\mathbf{S}}(0) = e^{-C} \cdot \mathbf{I}_n$. Then we have $A \cdot \mathbf{I}_n = e^{-C} \cdot \mathbf{I}_n$, which implies that $A = e^{-C}$. Thus,

$$\hat{\mathbf{S}}(t) = e^{-C} \cdot \sum_{i=0}^{\infty} \frac{t^i}{i!} \cdot \mathbf{Q}^i \cdot (\mathbf{Q}^T)^i.$$

Setting $t = C$, we get $\hat{\mathbf{S}} \triangleq \hat{\mathbf{S}}(C)$ is the solution to Eq.(14). ■

To iteratively compute $\hat{\mathbf{S}}$, the conventional way is to use *the Euler method* [2] for approximating $\hat{\mathbf{S}}(t)$ at time $t = C$. Precisely, by choosing a value h for the step size, and setting $t_k = k \cdot h$, one step of the Euler method from t_k to t_{k+1} is

$$\hat{\mathbf{S}}_{k+1} = \hat{\mathbf{S}}_k + h \cdot \mathbf{Q} \cdot \hat{\mathbf{S}}_k \cdot \mathbf{Q}^T, \quad \hat{\mathbf{S}}_0 = \hat{\mathbf{S}}(0) = e^{-C} \cdot \mathbf{I}_n.$$

Subsequently, the value of $\hat{\mathbf{S}}_k$ is an approximation of the solution to Eq.(14) at time $t = t_k$, *i.e.,* $\hat{\mathbf{S}}_k \approx \hat{\mathbf{S}}(t_k)$. However, the approximation error of the Euler method hinges heavily on the choice of step size h, which is hard to determine since the small choice of h would entail huge computational cost for attaining high accuracy. To address this issue, we adopt the following iterative paradigm for computing $\hat{\mathbf{S}}$ by taking a finite sum of Eq.(13):

$$\left\{ \begin{array}{l} \mathbf{T}_{k+1} = \mathbf{Q} \cdot \mathbf{T}_k \cdot \mathbf{Q}^T \\ \hat{\mathbf{S}}_{k+1} = \hat{\mathbf{S}}_k + e^{-C} \cdot \frac{C^{k+1}}{(k+1)!} \cdot \mathbf{T}_{k+1} \end{array} \right. \text{with} \left\{ \begin{array}{l} \mathbf{T}_0 = \mathbf{I}_n \\ \hat{\mathbf{S}}_0 = e^{-C} \cdot \mathbf{I}_n \end{array} \right. \tag{15}$$

Note that the main difference in our approach, as compared to the Euler method, is that there is no need for the choice of a particular step size h to iteratively compute $\hat{\mathbf{S}}$. The correctness of our approach can be easily verified, by induction on k, that the value of $\hat{\mathbf{S}}_k$ in our iteration Eq.(15) equals the sum of the first k terms of the infinite series $\hat{\mathbf{S}}$ in Eq.(13).

Regarding the computational time, our iteration Eq.(15) of the matrix differential SimRank may retain the same complexity, in the worst case, as that of the conventional SimRank in Eq.(2). It is worth noticing that the matrix equation in Eq.(15) can be rewritten in the following component form:

$$\begin{aligned}[\mathbf{T}_{k+1}]_{(a,b)} &= [\mathbf{Q} \cdot \mathbf{T}_k \cdot \mathbf{Q}^T]_{(a,b)} = \sum_{i,j} [\mathbf{Q}]_{(a,i)} [\mathbf{Q}]_{(b,j)} [\mathbf{T}_k]_{(i,j)} \\ &= \frac{1}{|\mathcal{I}(a)||\mathcal{I}(b)|} \sum_{j \in \mathcal{I}(b)} \sum_{i \in \mathcal{I}(a)} [\mathbf{T}_k]_{(i,j)},\end{aligned}$$

which takes the same form as the conventional SimRank formula Eq.(2) except for the damping factor C. Thus, our prior optimization techniques of partial sums sharing in Section III for the conventional SimRank Eq.(2) can be applied in a similar way to Eq.(15), for achieving a better complexity. For the interest of space, we omit the detailed algorithm here.

Error Estimate. In the SimRank matrix differential model, the following estimate for the k-th iterative similarity matrix $\hat{\mathbf{S}}_k$ with respect to the exact one $\hat{\mathbf{S}}$ can be established.

Proposition 7. *For each iteration $k = 0, 1, 2, \cdots$, the difference between the k-th iterative and the exact similarity matrix in Eqs.*(13) *and* (15) *can be bounded as follows:*

$$\|\hat{\mathbf{S}}_k - \hat{\mathbf{S}}\|_{\max} \le \tfrac{C^{k+1}}{(k+1)!}, \tag{16}$$

where $\|\mathbf{X}\|_{\max} \triangleq \max_{i,j} |x_{i,j}|$ is the max norm.

Proof: Subtracting Eq.(13) from Eq.(15), we obtain

$$\hat{\mathbf{S}}_k - \hat{\mathbf{S}} = e^{-C} \cdot \sum_{i=k+1}^{\infty} \frac{C^i}{i!} \cdot \mathbf{Q}^i \cdot (\mathbf{Q}^T)^i.$$

Taking the matrix-to-vector operator $vec(\star)$ [14] on both sides, and then applying the Kronecker product property that $vec(\mathbf{AXB}) = (\mathbf{B}^T \otimes \mathbf{A}) \cdot vec(\mathbf{X})$ to the right-hand side gives

$$vec(\hat{\mathbf{S}}_k - \hat{\mathbf{S}}) = e^{-C} \cdot \sum_{i=k+1}^{\infty} \frac{C^i}{i!} \cdot (\mathbf{Q} \otimes \mathbf{Q})^i \cdot vec(\mathbf{I}_n),$$

Notice that $\mathbf{Q}$ is a transitional matrix, *i.e.,* the sum of each row in $\mathbf{Q}$ is less than 1, which implies that $\|\mathbf{Q} \otimes \mathbf{Q}\|_{\infty} \le 1$.

Take the matrix ∞-norm $\|\star\|_{\infty}$ on both sides, and apply $\|vec(\star)\|_{\infty} = \|\star\|_{\max}$ to the left-hand side, we have

$$\begin{aligned}\|\hat{\mathbf{S}}_k - \hat{\mathbf{S}}\|_{\max} &\le e^{-C} \cdot \sum_{i=k+1}^{\infty} \frac{C^i}{i!} \cdot \|(\mathbf{Q} \otimes \mathbf{Q})\|_{\infty}^i \cdot \|vec(\mathbf{I}_n)\|_{\infty} \\ &\le e^{-C} \cdot \sum_{i=k+1}^{\infty} \frac{C^i}{i!} \le \frac{C^{k+1}}{(k+1)!},\end{aligned}$$

where the last inequality holds since using the Lagrange remainder $\frac{f^{(k+1)}(\xi)}{(k+1)!} C^{k+1}$, $\xi \in (0, C)$, of Maclaurin series for

$f(C) = e^C$ yields $\sum_{i=k+1}^{\infty} \frac{C^i}{i!} = \frac{e^{\xi}}{(k+1)!} C^{k+1} \le \frac{e^C}{(k+1)!} C^{k+1}$. ■

For the SimRank differential model Eq.(13), Proposition 7 allows finding out the exact number of iterations needed for attaining a desired accuracy, based on the following corollary.

Corollary 1. *For a desired accuracy $\epsilon > 0$, the number of iterations required to perform Eq.(15) is*

$$K' \ge \lceil \frac{\ln \epsilon'}{W(\frac{1}{e \cdot C} \cdot \ln \epsilon')} \rceil, \text{ with } \epsilon' = (\sqrt{2\pi} \cdot \epsilon)^{-1}.$$

Here, $W(\star)$ is the Lambert W function [9].

Proof: Based on Eq.(16), $\forall \epsilon > 0$, we need to find an integer $K' > 0$ such that $\frac{C^{K'+1}}{(K'+1)!} \le \epsilon$.

We first use the Stirling's formula $(K'+1)! \ge \sqrt{2\pi} \cdot (\frac{K'+1}{e})^{K'+1}$ to obtain $(\frac{e \cdot C}{K'+1})^{K'+1} \le \sqrt{2\pi} \cdot \epsilon$.

Let $x = \frac{K'+1}{e \cdot C}$. It follows that $x^x \ge (\sqrt{2\pi} \cdot \epsilon)^{-\frac{1}{e \cdot C}}$. Using the Lambert W function, we have $x \ge \frac{\ln (\sqrt{2\pi} \cdot \epsilon)^{-\frac{1}{e \cdot C}}}{W(\ln (\sqrt{2\pi} \cdot \epsilon)^{-\frac{1}{e \cdot C}})}$. By substituting $x = \frac{K'+1}{e \cdot C}$ back into the inequality, we get the final result, which completes the proof. ■

Noting that $\ln(x) - \ln(\ln(x)) \le W(x) \le \ln(x)$, $\forall x > e$ [9], we have the following improved version of Corollary 1, which may avoid computing the Lambert W function.

Corollary 2. *For a desired accuracy $0 < \epsilon < \frac{1}{\sqrt{2\pi}} e^{-C \cdot e^2}$, the number of iterations needed to perform Eq.(15) is*

$$K' \ge \lceil \frac{-\ln(\sqrt{2\pi} \cdot \epsilon)}{\eta - \ln(\eta)} \rceil \text{ with } \eta = \ln(-\frac{1}{e \cdot C} \cdot \ln(\sqrt{2\pi} \cdot \epsilon)).$$

Comparing this with the conventional SimRank model that requires $K = \lceil \log_C \epsilon \rceil$ iterations [16] for a given accuracy ϵ, we see that our revision of the differential SimRank model in Eq.(14) can greatly speed up the convergence of SimRank iterations from the original geometric to exponential rate.

As an example, setting $C = 0.8$ and $\epsilon = 0.0001$, since $\frac{1}{\sqrt{2\pi}} e^{-0.8 \cdot e^2} = 0.0011 > 0.0001$, we can use Corollary 2 to find out the number of iterations K' in Eq.(15) necessary to our differential SimRank model Eq.(14) as follows:

$$\eta = \ln(-\frac{1}{e \cdot 0.8} \cdot \ln(\sqrt{2\pi} \cdot 0.0001)) = 1.3384,$$

$$K' \ge \lceil \frac{-\ln(\sqrt{2\pi} \cdot 0.0001)}{1.3384 - \ln(1.3384)} \rceil = \lceil \frac{8.2914}{1.0469} \rceil = 7.$$

In contrast, the conventional SimRank model Eq.(2) needs $K = \lceil \log_{0.8} 0.0001 \rceil = 41$ iterations.

For ranking purpose, our experimental results in Section V further show that the revised notion of SimRank in Eq.(14) not only drastically reduces the number of iterations for a desired accuracy, but can fairly maintain the relative order of vertices with respect to the conventional SimRank in [16].

V. Empirical Evaluation

We present an experimental study on both real and synthetic data to evaluate the efficacy of our proposed methods.

A. Experimental Setting

Datasets. We use the real data (BERKSTAN, PATENT, DBLP) to evaluate the efficiency of our approaches (see Fig. 5), and the synthetic data (SYN) to vary graph characteristics.

Dataset		Vertices	Edges	Avg Deg.
BERKSTAN		685,230	7,600,595	11.1
PATENT		3,774,768	16,518,948	4.4
DBLP	D02	5,982	15,985	2.7
	D05	9,342	22,427	2.4
	D08	13,736	37,685	2.7
	D11	19,371	51,146	2.6

Fig. 5: Real-life Dataset Details

(1) BERKSTAN. The first network is a Berkeley-Stanford web graph of 7.4M hyperlinks between 680K web pages (from `berkely.edu` and `stanford.edu` domains), downloaded from the Stanford Network Analysis Project (SNAP). [5]

(2) PATENT. This is a citation network among U.S. Patents, obtained from the National Bureau of Economic Research. [6] It is our largest dataset consisting of 3.2M U.S. patents (vertices) and 16.1M citations (edges), with a low average degree of 4.4.

(3) DBLP. This is a scientific publication network, derived from DBLP Computer Science Bibliography. [7] We selected the recent 12-year publications (from 2000 to 2011) in 8 major conferences (ICDE, VLDB, SIGMOD, PODS, CIKM, ICDM, SIGIR, SIGKDD), and then built 4 co-authorship graphs by choosing every 3 years as a time step.

(4) SYN. The synthetic data were produced by the well-known graph generator GTGraph [8], varying two parameters: the number of vertices, and the number of edges.

Compared Algorithms. We implement the following algorithms using Visual C++ 8.0. (1) OIP-DSR, our differential form of SimRank in conjunction with partial sums sharing. (2) OIP-SR, the conventional SimRank using partial sums sharing. (3) psum-SR [16], without partial sums sharing. (4) mtx-SR [14], a matrix-based SimRank via SVD factorization.

We set the following default parameters: $C = 0.6, \epsilon = 0.001$ (unless otherwise mentioned).

Evaluation Metrics. For evaluating ranking results on DBLP, we adopted *Normalized Discounted Cumulative Gain* (NDCG) [14]. The NDCG at a rank position p is defined as $\text{NDCG}_p = \frac{1}{\text{IDCG}_p} \sum_{i=1}^{p} (2^{\text{rank}_i} - 1)/\log_2(1+i)$, where rank_i is the graded relevance at position i, and IDCG_p is a normalization factor, ensuring the NDCG of an ideal ranking at position p is 1.

For ground truth, we invited ten independent evaluators from the database community, and used their final judgment, rendered by a majority vote, as the standard.

We used a machine powered by a Quad-Core Intel i5 CPU (3.10GHz) with 16GB RAM, running Windows 7. Each experiment was run 5 times, and the average is reported here.

B. Experimental Results

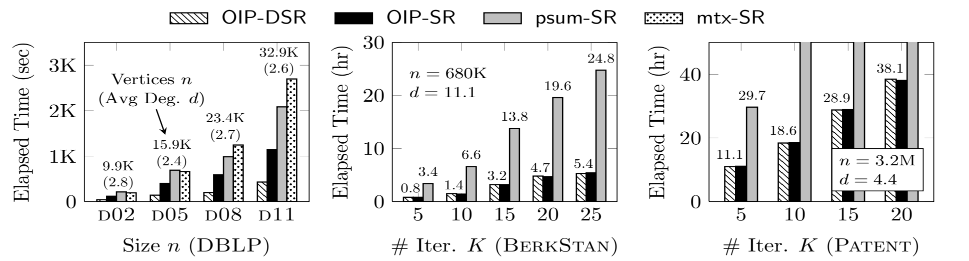
Exp-1: Time Efficiency. We first evaluate (1) the computational time of OIP-SR and OIP-DSR against psum-SR and mtx-SR over real data, and (2) the impact of graph density

[5]http://snap.stanford.edu/data/web-BerkStan.html

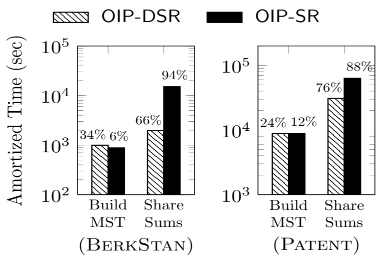
[6]http://data.nber.org/patents/

[7]http://dblp.uni-trier.de/˜ley/db/

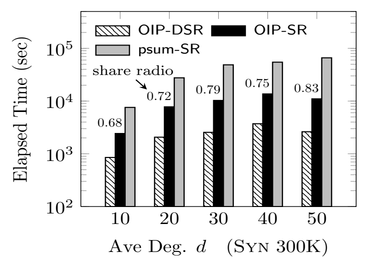
[8]http://www.cse.psu.edu/˜madduri/software/GTgraph/index.html



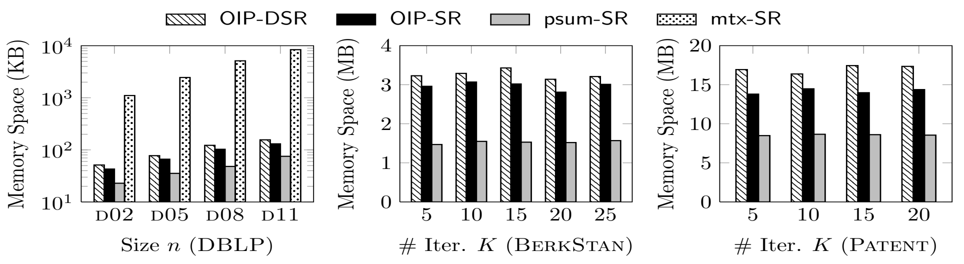
(a) Time Efficiency on Real Datasets



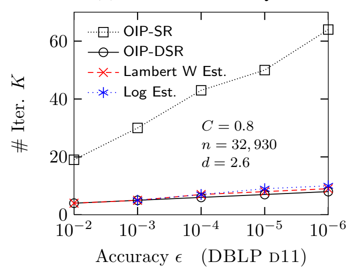
(b) Amortized Time on Real Data



(c) Effect of Density



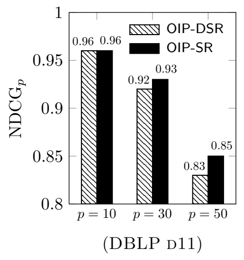
(d) Memory Space on Real Datasets



(e) Convergence Rate

Err ϵ	OIP-SR	OIP-DSR	LamW Est.	Log Est.
10^{-2}	19	4	4	-
10^{-3}	30	5	5	5
10^{-4}	43	6	7	7
10^{-5}	50	7	8	9
10^{-6}	64	8	9	10

(f) Lam W & Log Bound on K



(g) Relative Ordering

#	Co-authors	#	Co-authors
1	Hongjun Lu	16	Aoying Zhou
2	Lu Qin	17	Xiang Lian
3	Xuemin Lin	18	Cheqing Jin
4	Wei Wang	19	Baichen Chen
5	Lei Chen	20	Byron Choi
6	Lijun Chang	21	Wenfei Fan
7	Yiping Ke	22	Rong-Hua Li
8	Haifeng Jiang	23	**Hong Cheng ▼**
9	Philip S. Yu	24	**Jun Gao ▲**
10	Gabriel Pui Cheong Fung	25	Xiaofang Zhou
11	James Cheng	26	Ke Yi
12	Weifa Liang	27	Yufei Tao
13	Ying Zhang	28	Nan Tang
14	Bolin Ding	29	Jinsoo Lee
15	Haixun Wang	30	Kam-Fai Wong

(h) Top-30 Co-authors of "Jeffrey Xu Yu"

Fig. 6: Performance Evaluation of OIP-SR and OIP-DSR on Real and Synthetic Datasets

on CPU time, using synthetic data. To favor mtx-SR that only works on *low-rank graphs* (*i.e.,* graph with a small rank of the adjacency matrix), DBLP data are used although OIP-SR and OIP-DSR work pretty well on various graphs.

Fixing $\epsilon = .001$ for DBLP, varying K for BerkStan and Patent, we compare the CPU time of the four algorithms. The results are depicted in Figure 6a, telling us the following. (1) In all cases, OIP-SR consistently outperforms mtx-SR and psum-SR, *i.e.,* our partial sums sharing approach is effective. On BerkStan and Patent, the speedups of OIP-SR are on average 4.6X and 2.7X, respectively, better than psum-SR. On the large Patent, when $K \geq 8$, psum-SR takes too long to finish the computation in two days, which is practically unacceptable. In contrast, both OIP-SR and OIP-DSR just need about 18.6 hours for $K = 10$. (2) OIP-DSR always runs up to 5.2X faster than psum-SR, and 3X faster than OIP-SR on DBLP, for the desired $\epsilon = .001$. This is because the differential matrix form of OIP-DSR increases the rate of convergence, which enables fewer iterations for attaining the prescribed ϵ, as expected. (3) The speedups of OIP-SR and OIP-DSR on BerkStan (4.6X) are more pronounced than those on DBLP (1.8X) and Patent (2.7X), which is due to the high degree of BerkStan ($d = 11.1$) that may potentially increase the overlapped area for common in-neighbor sets, and thus provides more opportunities for partial sums sharing.

Figure 6b further shows the amortized time for each phase of OIP-SR and OIP-DSR on BerkStan and Patent data (given $\epsilon = .001$), in which x-axis represents different stages. From the results, we can discern that (1) for OIP-SR, the time taken for "Building MST" is far less than the time taken for "Share Sums". This confirms our complexity analysis in Proposition 5. (2) "Building MST" always takes up larger portions (34% on BerkStan, and 24% on Patent) in the total time of OIP-DSR, than those (6% on BerkStan, and 12% on Patent) in the total time of OIP-SR. This becomes more evident on various datasets because OIP-SR and OIP-DSR takes (almost) the same time for "Building MST", whereas, for "Sharing Sums", OIP-DSR enables less time (4.5X on BerkStan, and 2.5X on Patent) than OIP-SR, due to the speedup in the convergence rate of OIP-DSR.

Fixing $n = 300$K and varying m from 3M to 15M on the synthetic data, Figure 6c reports the impact of graph density (ave. in-degree) on CPU time, where y-axis is in the log scale. The results show that (1) for $\epsilon = .001$, OIP-DSR significantly outperforms psum-SR by at least one order of magnitude as m increases. In all the cases, OIP-SR achieves 0.5 order of magnitude speedups on average. (2) Interestingly, the speedups of OIP-DSR are sensitive to graph density (ave. in-degree d) The larger the d is, the higher the likelihood of overlapping in-neighbors is for partial sums sharing, as expected. The biggest speedups are observed for larger d (higher density) — with nearly 2 orders of magnitude speedup for $d = 50$.

Exp-2: Memory Space. We next evaluate the space efficiency of OIP-DSR and OIP-SR against psum-SR and mtx-SR, using real data. Note that we only use mtx-SR on small DBLP as a baseline; for large BERKSTAN and PATENT, the memory space of mtx-SR will explode since the SVD method of mtx-SR destroys the sparsity of a graph.

Figure 6d shows the results on space. We observe that (1) on DBLP, OIP-DSR and OIP-SR have much less space than mtx-SR by at least one order of magnitude, as expected. (2) In all the cases, the space cost of OIP-DSR and OIP-SR fairly retains the same order of magnitude as psum-SR. Indeed, both OIP-DSR and OIP-DSR merely need about 1.8X, 1.9X, 1.6X space of psum-SR on DBLP, BERKSTAN, PATENT, respectively, for outer partial sums sharing. This confirms our complexity analysis in Section III, suggesting that OIP-DSR and OIP-DSR do not require too much extra space for caching outer partial sums. Moreover, OIP-DSR needs a bit more space than OIP-SR due to the memoization of the auxiliary $\mathbf{T}_k$ in Eq.(15) per iteration. (3) On BERKSTAN and PATENT, the space costs of OIP-DSR and OIP-SR are stabilized as K increases. This is because the memorized partial sums are released immediately after each iteration, thus maintaining the same space costs during the iterations.

Exp-3: Convergence Rate. We next compare the convergence rate of OIP-DSR and OIP-SR, using real and synthetic data. For the interest of space, below we only report the results on DBLP D11 $(C = 0.8)$. The trends on other data are similar.

By varying ϵ from 10^{-2} to 10^{-6}, Fig. 6e and 6g show that (1) OIP-DSR needs far fewer iterations than OIP-SR (also psum-SR), for a given accuracy. Even for a small $\epsilon = 10^{-6}$, OIP-DSR only requires 8 iterations, whereas the convergence of OIP-SR in this case becomes sluggish, yielding over 60 iterations. This confirms our observation in Proposition 7 that OIP-DSR has an exponential rate of convergence. (2) The two curves labeled "Lambert W Est." and "Log Est." (dashed line) visualize our apriori estimates of K' derived from Corollaries 1 and 2, respectively. We can see that these dashed curves are close to the actual number iterations of OIP-DSR, suggesting that our estimates of K' for OIP-DSR are fairly precise.

Exp-4: Relative Order. To analyze the relative order of the similarity scores obtained from OIP-DSR and OIP-SR, we use DBLP D11, a co-authorship graph with ground truth. Fixing the vertex a as a given query (author), we compute the NDCG$_p$ values of OIP-DSR and OIP-SR via the similarities $s(a, \star)$ from the top-p query perspective. We issue the three queries $a =$ "Jeffrey Xu Yu", "Philip S. Yu", "Jian Pei". For each query, Figure 6g compares the average NDCG$_p$ values of OIP-DSR with its counterparts of OIP-SR, for $p = 10, 30, 50$. The result shows that OIP-DSR can perfectly maintain the relative order of the similarity scores produced by OIP-DSR with only 1% loss of NDCG$_{30}$ and NDCG$_{50}$. For $p = 10$ (*i.e.,* top-10 query), OIP-DSR produces exactly the same result of OIP-SR, which is to be expected. Thus, we can gain a lot in speedup from OIP-DSR while suffering little loss in quality.

Figure 6h shows the top-30 co-authors of "Jeffrey Xu Yu", by using OIP-DSR on DBLP D11. Comparing this with the results of OIP-SR, we see that the results of OIP-DSR merely differ in one inversion at two adjacent positions (#23, #24), which is practically acceptable. This confirms our intuitions in Section IV, where we envisage that the slight modification of a damping factor in OIP-DSR never incurs high quality loss.

VI. CONCLUSIONS

In this study, we have proposed two efficient methods to speed up the computation of SimRank on large graphs. Firstly, we leveraged a novel clustering approach to optimize partial sums sharing. By eliminating the duplicates of computational efforts among the partial summations, an efficient algorithm was devised to greatly reduce the time complexity of SimRank. Secondly, we proposed a revised SimRank model based on the matrix differential representation, achieving an exponential speedup in the convergence rate of SimRank, as opposed to its conventional counterpart of a geometric speedup. Our empirical experiments on both real and synthetic datasets have shown that the integration of our proposed methods can significantly outperform the best known algorithm by about one order of magnitude, with very little sacrifice in quality.

REFERENCES

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

[2] U. M. Ascher and L. R. Petzold. *Computer Methods for Ordinary Differential Equations and Differential-Algebraic Equations*. Society for Industrial and Applied Mathematics, 1998.

[3] P. Berkhin. Survey: a survey on PageRank computing. *Internet Mathematics*, 2:73–120, 2005.

[4] J. Cho and S. Roy. Impact of search engines on page popularity. In *WWW*, 2004.

[5] F. R. K. Chung and A. Tsiatas. Finding and visualizing graph clusters using PageRank optimization. *Internet Mathematics*, 8:46–72, 2012.

[6] D. Fogaras and B. Rácz. Practical algorithms and lower bounds for similarity search in massive graphs. *IEEE Trans. Knowl. Data Eng.*, 19:585–598, 2007.

[7] H. N. Gabow, Z. Galil, T. H. Spencer, and R. E. Tarjan. Efficient algorithms for finding minimum spanning trees in undirected and directed graphs. *Combinatorica*, 6:109–122, 1986.

[8] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[9] M. Hassani. Approximation of the Lambert W function. *RGMIA Research Report Collection*, 8, 2005.

[10] G. He, H. Feng, C. Li, and H. Chen. Parallel SimRank computation on large graphs with iterative aggregation. In *KDD*, 2010.

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

[12] R. Kumar, J. Novak, and A. Tomkins. Structure and evolution of online social networks. In *KDD*, 2006.

[13] P. Lee, L. V. Lakshmanan, and J. X. Yu. On Top-k structural similarity search. In *ICDE*, 2012.

[14] 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*, 2010.

[15] P. Li, H. Liu, J. X. Yu, J. He, and X. Du. Fast single-pair SimRank computation. In *SDM*, 2010.

[16] D. Lizorkin, P. Velikhov, M. N. Grinev, and D. Turdakov. Accuracy estimate and optimization techniques for SimRank computation. *PVLDB*, 1:422–433, 2008.

[17] W. Yu, X. Lin, W. Zhang, Y. Zhang, and J. Le. SimFusion+: Extending SimFusion towards efficient estimation on large and dynamic networks. In *SIGIR*, 2012.

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

[19] P. Zhao, J. Han, and Y. Sun. P-Rank: a comprehensive structural similarity measure over information networks. In *CIKM*, 2009.