Efficient Partial-Pairs SimRank Search on Large Graphs

Weiren Yu†, Julie A. McCann†
†Imperial College London, United Kingdom
{weiren.yu, j.mccann}@imperial.ac.uk

ABSTRACT

The assessment of node-to-node similarities based on graph topology arises in a myriad of applications, e.g., web search. SimRank is a notable measure of this type, with the intuition that “two nodes are similar if their in-neighbors are similar”. While most existing work on SimRank only considers all-pairs SimRank $s(\ast, \ast)$ and single-source SimRank $s(\ast, j)$ (scores between every node and query $j$), there are appealing applications for partial-pairs SimRank, e.g., similarity join. Given two node subsets $A$ and $B$, partial-pairs SimRank assessment aims to retrieve only $\{s(a, b)\}_{a \in A, b \in B}$. However, the best-known solution [17] is not self-contained since it hinges on the premise that the SimRank scores with node-pairs in an $h$-go cover set must be given beforehand.

This paper focuses on efficient assessment of partial-pairs SimRank in a self-contained manner. (1) We devise a novel “seed germination” model that computes partial-pairs SimRank in $O(k|E| \min\{|A|, |B|\})$ time and $O(|E| + k|V|)$ memory for $k$ iterations on a graph of $|V|$ nodes and $|E|$ edges. (2) We further eliminate unnecessary edge access to improve the time of partial-pairs SimRank to $O(m \min\{|A|, |B|\})$, where $m \leq \min\{k|E|, \Delta^{2k}\}$, and $\Delta$ is the maximum degree. (3) We show that our partial-pairs SimRank model also can handle the computations of all-pairs and single-source SimRanks, as well as partial-pairs SimRank (a related notion of SimRank). (4) We empirically verify that our algorithms are (a) 38x faster than the best-known competitors, and (b) memory-efficient, allowing scores to be assessed accurately on graphs with tens of millions of links.

1. INTRODUCTION

An overarching problem in link analysis is the measure of relevance between nodes based on graph topology. This type of relevance, termed link-based similarity, has long pervaded fertile fields of data management, e.g., link prediction [7], web page ranking [2], and automatic image captioning [11]. Link-based similarity, unlike its text-based counterpart, is structure dependent and does not rely on domain knowledge.

With the recurring emergence of many similarity measures, SimRank [7] has become attractive, due to its concise and recursive idea that “two nodes are assessed as similar if their in-neighbors are similar”, along with the base case that “every node is most similar to itself”. Compared with other link-based measures, SimRank bears two impressive merits: (1) It exploits global graph topology for both node-pair and node ranking, unlike PageRank [2] that only ranks nodes. (2) It also captures multifaceted relations (multiple paths of different lengths) between nodes, as opposed to Shortest Path that only tallies a single path of the shortest length. Hence, SimRank has been widely-accepted as an important tool for link analysis over the last decade [3-5, 7, 10, 15, 17].

However, prior iterative methods (e.g., [13, 15]) for assessing SimRank have two limitations:

(1) They mainly consider all-pairs SimRank $s(\ast, \ast)$ computation, and are not efficient if one needs to assess only partial-pairs similarities. This is because, even for retrieving a single-pair similarity $s_k(a, b)$ between nodes $a$ and $b$ at iteration $k$, their iterative models require all-pairs similarities $s_{k-1}(\ast, \ast)$ at iteration $(k − 1)$ to be determined beforehand. We refer to this phenomenon as “high iteration coupling”. To the best of our knowledge, there are several pioneering works trying to break the “high iteration coupling” barrier, such as single-pair SimRank [3, 6, 12], and top-K single-source SimRank [4, 9, 10], but they still require excessive time and memory, or deliver probabilistic results. (see Related Work section for a detailed comparison).

(2) Existing iterative methods for assessing all-pairs SimRank are not memory-efficient, which is the main obstacle to handling large graphs. Indeed, [13, 15] require $O(|V|^2)$ memory\(^1\) at each iteration to store all-pairs scores from the previous iteration, which is due to “high iteration coupling”. Thus, prior methods assessing all-pairs SimRank on a graph with $|V| \geq 500K$, if without loss of exactness, would require $\geq (500K)^2 = 250G$ memory, which is rather impractical. To address this issue, Lizorkin et al. [13] devised an appealing heuristic to iteratively eliminate small similarities below a threshold. This method has the advantage of controlling the number of node-pairs of non-zero similarities fewer enough to fit into memory at the expense of exactness. However, it is difficult for the heuristic to enhance the quality of real applications. Fortunately, we observe that, if partial-pairs SimRank can break the “high iteration coupling” nature, all-pairs SimRank can be divided into several independent tasks of assessing partial-pairs SimRank with small memory.

\(^1\)Though [13, 15] claimed their methods require $O(|V|)$ intermediate memory, they exclude $O(|V|^2)$ memory to write all-pairs similarity outputs from the previous iteration.
is hard to know in advance the SimRank scores of the

### 1.1 Contributions

In this paper, we make the following key contributions:

- We first design a fast "seed germination" iterative model to retrieve partial-pairs SimRank in a self-contained fashion, in $O(|E| \min(|A|, |B|))$ time and $O(|E| + k|V|)$ memory for $k$ iterations on a graph of $|V|$ nodes and $|E|$ edges, where $|A|$ and $|B|$ are the sizes of two query node subsets $A$ and $B$, respectively. (Section 3.1)

- We then propose a pruning strategy, coupled with a hashing structure, to eliminate unnecessary computation with no loss of accuracy, which can further speed up the time of partial-pairs SimRank to $O(m \min(|A|, |B|))$, where $m \leq \min\{k|E|, \Delta^2\}$ and $d$ is the maximum degree. (Section 3.2)

- By setting $A = V$ and $B = \{j\}$, our partial-pairs SimRank, as a special case, also improves the time of single-source SimRank to $O(\min\{k|E|, \Delta^2\})$. In comparison, the fastest known top-K version [10] needs $O(\Delta^2)$ time. (Section 4.1)

- As a by-product, we derive an $O(|E| + k|V|)$-memory efficient all-pairs SimRank algorithm, by dividing it into independent tasks of partial-pairs SimRank, whose total time, $O(\min\{k|E|, \Delta^2\}|V|)$, is comparable to the best-known all-pairs SimRank [13,15], yet without using $O(|V|^2)$ memory to store scores from previous iterations. (Section 4.2)

- We extend our methods to partial-pairs SimRank*, a semantically-enhanced version of SimRank. (Section 4.3)

The empirical evaluations on real and synthetic data verify that (a) our partial-pairs SimRank is self-contained, and runs drastically faster than the best-known competitors, and (b) our methods are memory-efficient, allowing SimRank being assessed accurately on large graphs with over 69M links.

### 1.2 Related Work

The problem of efficient SimRank computation has been well-studied, starting from the work of Jeh and Widom [7], and continuing with a long line of research, [3,4,6,10–13,15,17]. A comparison of our results (in terms of computational time, memory, and accuracy) to those in previous works is summarized in Table 1. We categorize these works below.

#### 1.2.1 Partial-Pairs SimRank

The most relevant work to our research is proposed by Zheng et al. [17]. They gave (1) an estimated shortest-path bound for SimRank scores to prune unlikely node-pairs, and (2) heuristics to find an h-go cover set, and stored the SimRank scores of the h-go cover set, based on which, other scores can be computed easily. However, by using only the algorithms of [17], it is hard to determine in advance the scores of node-pairs in the h-go cover set. Our work differs from [17] in that our methods are more efficient and self-contained, needing no determination of extra SimRank scores in advance. We also analyze the complexity bound.

Sun et al. [14] provided a top-K link-based similarity join, with an $e$-function that generalizes PageRank and SimRank. They also obtained $e$-function bounds to prune unlikely nodes, requiring $O(k^2|E|)$ time to estimate the upper bound of a single-pair score. However, our methods can accurately compute $|A| \times |B|$ pairs of scores in $O(k|E| \min\{|A|, |B|\})$ time.

#### 1.2.2 Single-Source SimRank

Lee et al. [10] proposed TopSim-SM via a random surfing process for top-K $s(x, j)$ search in $O(\Delta^2)$ worst-case time. They also used heuristics to merge some repeated nodes, truncate low scores, and prioritize propagation. Their methods have the merits of fast speed when the top-K size is very large.

---

**Figure 1**: A Labeled Citation Network $G$
small, e.g., K = 50 (< |V|), but are not efficient for a larger top-K that will lead to more iterations k, since O(Δ^2k) grows exponentially w.r.t. k. In contrast, our “seed germination” method, covering single-source SimRank as a special case, can avoid this exponential growth.

Kusumoto et al. [9] has proposed an algorithm via Monte Carlo method, called Monte-Carlo-SR, that retrieves top-K single-source SimRank in O(k^2E + k(P + N)|V|) time, where P and N are the sampling parameters. Their method can avoid the exponential growth of the O(Δ^2k) time in [10]. However, [9] delivers probabilistic results, and is slower than our “seed germination” model since it does not consider the computation sharing for single-source SimRank retrieval.

Recently, Fujiwara et al. [4] has devised an appealing matrix-based SimMat in O(r|V|^2) time (exclude preprocessing time) to retrieve top-K s(*, j), where r is the low rank of approximation. Their approach is deterministic, but requires O(r|V|^2) precomputation time to determine input matrices L and R for Algorithm 2 of [4], which is costly.

1.2.3 All-Pairs SimRank

Our focus is devoted to [11, 13, 15] as they are the state of the art. Izzokin et al. [13] proposed an excellent method of partial sum memoization, reducing the time of all-pairs SimRank to O(k|V||E|). They also devised a threshold-pruning heuristic and essential node-pairs selection for further improvement. Yu et al. [15] used a fine-grained clustering strategy for sub-summation sharing, yielding O(kd’|V|^2) time with d’ ≤ |E|/|V|. Li et al. [11] used singular value decomposition for assessing SimRank in O(r|V|^2) time, but it is not always efficient when the target rank r is not small. None of these methods are memory-efficient; [13, 15] require O(|V|^2) to store all-pairs scores from the previous iteration.

1.2.4 Single-Pair SimRank

The earliest mention of single-pair SimRank can be traced back to Fogaras et al. [3] who estimated s(a, b) from the first hitting time of two random surfers starting from nodes a and b. However, [3] delivers probabilistic results. To enhance search quality, Li et al. [12] devised an iterative model in O(kd’ min(Δ^2, |V|^2)) worst-case time and O(|V|^2) memory, where d’ is the graph average degree. Recently, [6] has also employed position probability to reduce the time of [12] to O(k(|E|^2 − |E|)). All the bounds of these deterministic single-pair SimRank are still expensive.

There has also been work [1, 8, 16] on SimRank variants. Antonellis et al. [1] extended SimRank for query rewriting. Jin et al. [8] integrated automorphism (role similarity) into SimRank. Yu et al. [16] devised SimRank*, by tallying more paths to enrich SimRank semantics. In Section 4.3, we also extend our methods to partial-pairs SimRank*.

2. PRELIMINARIES

Based on the idea that “two nodes are assessed as similar if their in-neighbors are similar”, SimRank has two forms: (1) Basic Form [7, 13]. Given a digraph G = (V, E), for any node a ∈ V, we denote by I(a) := {x ∈ V | (x, a) ∈ E} the in-neighbor set of a, and |I(a)| the in-degree of a.

The SimRank similarity between nodes a and b, s(a, b), is defined as (i) s(a, b) = 0, if I(a) or I(b) = ∅; (ii) otherwise,

\[ s(a, b) = \sum_{x, y \in \text{I}(a) \cap \text{I}(b)} \frac{1}{|I(x)|} \sum_{(i, j) \in \text{E}(x, y)} s(i, j), \]

where C ∈ (0, 1) is a decay factor.

(2) Matrix Form [5, 11]. Let S ∈ R^{|V|×|V|} be a SimRank matrix whose entry S_{i,j} is the score s(i, j). W the column normalized adjacency matrix whose entry W_{i,j} = 1/|I(j)| if ∃ an edge i → j, and 0 otherwise, and I the |V| × |V| identity matrix. Then, SimRank can be also defined as

\[ S = C \cdot W^T \cdot S \cdot W + (1 - C) \cdot I, \]

where (·)^T is the matrix transpose. The term (1 − C) · I in Eq.(2) ensures all the diagonal entries of S are maximal, which guarantees that every node is most similar to itself.

3. OUR SOLUTIONS

Our solutions to partial-pairs SimRank involve two ideas: (1) We devise a novel “seed germination” iterative model that breaks “high iteration coupling” of the existing models. (2) In light of this, we design a pruning strategy to further skip unnecessary computation, without loss of accuracy.

The following notations are used throughout this paper. (1) \( e_i \) is a |V| × 1 vector of all 1s except for a 1 in j-th entry. (2) For matrix X, (a) let \( X_{i,j} \) be the j-th column of X, (b) \( X_{i,*} \) the i-th row of X, (c) \( X_{i,j} \) the (i, j)-entry of X.
3.1 A “Seed Germination” Iterative Model

We first formulate the existing “high iteration coupling” barriers, and then propose our “seed germination” methods.

3.1.1 “High Iteration Coupling” Barrier

Let \( S_k \) denote the \( k \)-th iterative SimRank matrix whose entry \( [S_k]_{ij} = s_k(i, j) \). Then, \( S \) in Eq.(2) can be iterated as

\[
S_k = C \cdot W^T \cdot S_{k-1} \cdot W + (1 - C) \cdot I. \tag{3}
\]

By post-multiplying by \( e_j \) on both sides of Eq.(3), and using the fact that \( X \cdot e_j = [X]_{\cdot j} \), it follows that

\[
[S_k]_{\cdot j} = C \cdot W^T \cdot S_{k-1} \cdot [W]_{\cdot j} + (1 - C) \cdot e_j, \quad \forall j. \tag{4}
\]

Intuitively, Eq.(4) indicates that, even for finding only a single column \([S_k]_{\cdot j}\), the entire matrix \( S_{k-1} \) (not \([S_{k-1}]_{\cdot j}\)) at iteration \((k - 1)\) must be known in advance. Thus, it is hard for the conventional Eq.(3) to find only partial-pairs SimRank, without iteratively computing scores of others. We call this the SimRank “high iteration coupling” barrier.

3.1.2 Partial Series Form of \( S_k \)

To break the barrier of “high iteration coupling”, we first express \( S_k \) as a partial series form. This is because, from Eq.(4), we notice that the computational bottleneck of \([S_k]_{\cdot j}\) is the appearing of the entire matrix \( S_{k-1} \) on the right-hand side. To prevent its appearing, we have the following lemma.

**Lemma 1.** For every \( k = 0, 1, \ldots, \), the \( k \)-th iterative SimRank matrix \( S_k \) derived from Eq.(3) can be expressed as

\[
S_k = (1 - C) \cdot \sum_{l=0}^{k} C^l \cdot (W^T)^l \cdot W^l. \tag{5}
\]

(The proof can be readily completed by induction on \( k \), and is omitted here, due to space limitations.)

Lemma 1 tells us that \( S_k \) in Eq.(3) is actually the \( k \)-th partial sum of the infinite SimRank power series.

3.1.3 Duplicate Computation in \( S_k \)

Lemma 1 is introduced for efficiently computing \([S_k]_{\cdot j}\). Unfortunately, Eq.(5), if carried out naively, contains many duplicate computations, as illustrated below.

First, we post-multiply by \( e_j \) on both sides of Eq.(5):

\[
[S_k]_{\cdot j} = (1 - C) \sum_{l=0}^{k} C^l \cdot (W^T)^l \cdot W^l \cdot e_j = (1 - C) \cdot e_j + \sum_{l=1}^{k} C^l \cdot (W^T)^l \cdot W^{l-1} \cdot ([W]_{\cdot j}) \tag{6}
\]

Since \([W]_{\cdot j}\) is a vector in Eq.(6), if the multiplications in \((W^T)^l \cdot W^{l-1} \cdot [W]_{\cdot j}\) are grouped from “right-to-left”:

\[
W^T \cdots (W^T \cdot (\cdots (W^T \cdot W^l \cdot [W]_{\cdot j}))),
\]

then we can safely avoid using matrix-matrix multiplications while preventing the appearing of the entire matrix \( S_{k-1} \) throughout the summation of computing \([S_k]_{\cdot j}\) in Eq.(6). Specifically, to compute \((W^T)^l \cdot W^{l-1} \cdot [W]_{\cdot j}\), our “right-to-left association” needs \((2l-1)\) matrix-vector multiplications, whereas the traditional way entails \((2l-2)\) matrix-matrix multiplications \((T \leftarrow (W^T)^l \cdot W^{l-1})\) and \((b)\) one matrix-vector multiplication \((T \cdot [W]_{\cdot j})\), being rather expensive.

However, only the “right-to-left association” is not enough since there is duplicate computation across many summands. For example in Eq.(6), by writing out the terms for \( l = 2, 3, \)

\[
l = 2 : W^T \cdot W^2 \cdot [W]_{\cdot j} \quad l = 3 : W^T \cdot W^2 \cdot W \cdot [W]_{\cdot j}
\]

we notice that there are many common parts shared by the summands of \( l = 2 \) and \( l = 3 \), such as \( W \cdot [W]_{\cdot j} \) (wavy underlined) and \( W^T \cdot W^2 \cdot W \) (underlined). Unfortunately, it is hard to maximally share these common parts among the summands of different \( l \) while employ “right-to-left association” to prevent matrix-matrix multiplications at the same time, since some common parts sharing may destroy the “right-to-left association”. As an example for \( l = 2, 3 \), our only opportunity seems to memoize the common parts \( W \cdot [W]_{\cdot j} \) in order to secure the “right-to-left association”. Otherwise, if we memoized more common parts for sharing (e.g., \( W^T \cdot W^2 \cdot W \)), matrix-matrix multiplications would be unnecessarily involved in Eq.(6) during \([S_k]_{\cdot j}\) computation.

3.1.4 A “Seed Germination” Iterative Model for Single-Source SimRank

To eliminate duplicate computation across the summands in Eq.(6) and also guarantee the “right-to-left association”, we now propose our “seed germination” model.

**Theorem 1.** Given a query \( j \), the single-source SimRank \([S_k]_{\cdot j}\) at iteration \( k \) can be computed as

\[
[S_k]_{\cdot j} = (1 - C) \cdot \sum_{l=0}^{k} C^l \cdot (W^T)^l \cdot W^l \cdot e_j \tag{7}
\]

where vector \( v_k \) is iterated as: \( \forall l = 1, \ldots, k \)

\[
v_l = C \cdot W^T \cdot v_{l-1} + u_{l-1} \quad \text{with} \quad v_0 = u_k
\]

after vectors \( u_0, u_1, \ldots, u_k \) are iterated as: \( \forall l = 1, \ldots, k \)

\[
u_l = W \cdot u_{l-1} \quad \text{with} \quad u_0 = e_j
\]

**Proof.** We show that \([S_k]_{\cdot j}\) obtained from Eqs.(7)–(9) is exactly the \( k \)-th partial sum in Eq.(6).

First, successive substitution applied to Eq.(9) yields \( u_l = W^l \cdot e_j \). Thereby, \([S_k]_{\cdot j}\) in Eq.(6) can be rewritten as

\[
[S_k]_{\cdot j} = (1 - C) \cdot \sum_{l=0}^{k} C^l \cdot (W^T)^l \cdot W^l \cdot e_j \tag{10}
\]

Next, we show that successive substitution applied to Eq.(8) yields \( v_k = \sum_{l=0}^{k} C^l \cdot (W^T)^l \cdot u_l \). Fixing \( l \) and pre-multiplying by \( C^{k-l} (W^T)^l \) on both sides of Eq.(8) produce

\[
C^{k-l} (W^T)^l \cdot v_l = C^{k-l} (W^T)^l \cdot u_{l-1} + C^{k-l} (W^T)^l \cdot u_{l-1}
\]

Then, taking sums \( \sum_{l=1}^{k} (*)& on both sides yields

\[
\sum_{l=1}^{k} C^{k-l} (W^T)^l \cdot v_l = \sum_{l=1}^{k} C^{k-l} (W^T)^l \cdot u_{l-1} + \sum_{l=1}^{k} C^{k-l} (W^T)^l \cdot u_{l-1}
\]

Eliminating \( \sum_{l=1}^{k} C^{k-l} (W^T)^l \cdot v_l \) on both sides gets

\[
v_k = C^{k} (W^T)^k \cdot u_0 + \sum_{l=0}^{k} C^{l} (W^T)^l \cdot u_l \]

Combining this with Eq.(10) yields Eq.(6).

As an illustrative example, the last column of Figure 3 depicts how Theorem 1 computes \([S_k]_{\cdot j}\) step by step.

Theorem 1 provides a novel iterative algorithm for computing single-source SimRank \([S_k]_{\cdot j}\). It works as follows. Given query \( j \) and iteration number \( k \), it first utilizes Eq.(9) to iteratively compute \( k \) auxiliary vectors \( u_1, \ldots, u_k \). Using \( u_1, \ldots, u_k \), it then iteratively obtains \( v_k \) from Eq.(8).
Figure 2: Merge paths \( p_1, p_2, p_3 \) to a compact tree \( p' \)

For each iteration \( l \) in Eq. (8), once \( v_i \) is computed, \( v_{l-1} \) and \( u_{k-l} \) can be freed. This iterative process continues until \( l \) reaches \( k \). Finally, applying \( v_k \) to Eq. (7) returns \([S_k]_{i,j}\).

Theorem 1 can effectively skip duplicate multiplications across the summands in Eq. (6). Precisely, our method only requires \( 2k \) matrix-vector multiplications (\( k \) multiplications for \( W \cdot u_{l-1} \) in Eq. (9) and \( k \) for \( W^T \cdot v_{l-1} \) in Eq. (8)). In contrast, even though “right-to-left association” is used, Eq. (6) requires \((2l - 1)\) matrix-vector multiplications for the \( l \)-th summand; thus, for \( k \) summands, Eq. (6) requires \( \sum_{l=1}^{k} (2l - 1) = k^2 \) matrix-vector multiplications in total.

### 3.1.5 Main Idea

Intuitively, the idea of our method in Theorem 1 to eliminate computational redundancies is an efficient “seed germination” way of tallying “specific” paths for assessing \([S_k]_{i,j}\). To clarify this, we begin with the following lemma.

**Lemma 2.** The single-source \([S_k]_{i,j}\) in Eq. (6) tallies the weighted sum of \( k \) length-2l “specific” paths (\( l = 1, \cdots, k \)):

\[
\begin{align*}
\text{path } p_1 & \\
(\text{W}^T & )^{l-1} & \circ \cdots \circ \cdots \circ \cdots \circ \text{path } p_2 & \\
\text{path } p_3 & \\
(\text{W}^T & )^{l} & \circ \cdots \circ \cdots \circ \cdots \circ \text{path } p_4 & \\
\end{align*}
\]

where \( \circ \) and * denote any node in graph \( G \).

(The proof can be readily completed by the adjacency matrix power property. We omit it here, due to space limits.)

Lemma 2 suggests that assessing \([S_k]_{i,j}\) in Eq. (6) is actually the process of tallying the weighted sum of the “specific” paths in (11). Hence, the problem of eliminating duplicate computation in \([S_k]_{i,j}\) boils down to the merging of repetitive node access when tallying paths (11) of different length. Furthermore, the “right-to-left association” in Eq. (6), in the context of paths tallying, is associated with the order of node access in paths (11) from right end \( j \) to left end *. Based on Lemma 2, we describe the essence of our method in Theorem 1, with an example of assessing \([S_k]_{i,j}\).

The conventional Eq. (6) using only the “right-to-left association” would carry out the following calculation:

\[
[S_k]_{i,j} = (1 - C)(e_j + \sum_{l=1}^{k} \text{path } p_1 \cdot \text{path } p_2) + \text{path } p_3 + \text{path } p_4
\]

where each summand corresponds to tallying a kind of “specific” path \( p_i \), as shown in Figure 2'.' When tallying these paths from right end \( j \) to left end *, we observe that there are many repetitive node access (enclosed with dotted lines).

For instance, node \( j \) and its in-neighbors are repeatedly accessed when paths \( p_1, p_2, p_3 \) are tallied; the 2-hop in-neighbors of node \( j \) are repeatedly accessed when \( p_2 \) and \( p_3 \) are tallied. \( p_1, p_2, p_3 \) in Figure 2 are right-aligned at node \( j \) for merging, corresponding to the “right-to-left association” computing.

To eliminate redundancies, we first merge these paths into a compact tree \( p' \) (in Figure 2'), and then efficiently access all nodes in tree \( p' \) via “seed germination” iteration.

Figure 3 visualizes our “seed germination” iteration step-by-step for tallying paths in \( p' \) for \([S_k]_{i,j}\), starting from \( j \). Each step corresponds to one iteration, at which we need to select “seed” nodes to produce new “bud” nodes, aiming to minimize repetitive node access for tallying \( p_1, p_2, p_3 \) in \( p' \).

For example, in “Step 1” row of Figure 3, we select node \( j \) as the “seed” to produce its “bud” nodes (\( v \)). This process is associated with the iteration \( u_{l} := W \cdot e_j \). Moreover, in each step, we can choose as “seed” nodes either the new “bud” nodes from the last step (e.,g., “seeds” in Steps 1–4), or the mixture of the new “bud” nodes and the old “germinated” nodes from several prior steps (e.,g., “seeds” in Steps 5–6).

For example, in “Step 5” row, the selected “seed” nodes (encircled with red dotted line) consist of two parts: one is the new “bud” nodes from Step 4 (boxed with blue dotted line, associated with the term \((C \cdot W^T \cdot v_0) \)) in \( v_j \); the other is the old “germinated” nodes from Step 2 (boxed with green dashed line, associated with the term \( v_2 \cdot u_2 \)) of which are integrated into \( v_1 \) in Step 4 and used as “seeds” in Step 5. In contrast, the conventional Eq. (6) tallies each of the paths \( p_1, p_2, p_3 \) in Figure 2 independently, by choosing only one “seed” to produce only one “bud” in one path \( p_i \) at each step, incurring excessive computational cost.

(See Appendix B.1 for a detailed numerical example of our “seed germination” iteration.)

### 3.1.6 Complexity of Single-Source SimRank

We next analyze the complexity of single-source SimRank.

**Theorem 2.** Given a graph \( G = (V, E) \) and query \( j \in V \), it takes \( O(k|E|) \) time and \( O(|E| + k|V|) \) memory to assess the \( k\)-th iterative single-source SimRank \([S_k]_{i,j}\) via Theorem 1.

**Proof.** Assessing \([S_k]_{i,j}\) via Theorem 1 has three phases:

1. **(1) Iteratively obtaining** \( u_0, u_1, \ldots, u_k \) via Eq. (9). For any fixed \( l \), it requires \( O(|E|) \) time and \( O(|V|) \) memory to compute \( u_l \), which is dominated by the matrix-vector multiplication \((W \cdot u_{l-1})\). Once computed, \( u_0, \ldots, u_{k-1} \) are memoized, which will be reused in subsequent iterations Eq. (8). Thus, this phase requires \( O(k|E|) \) time and \( O(|V|) \) memory for \( k \) iterations, plus \( O(|E|) \) memory to store the graph.

2. **(2) Iteratively computing** \( v_k \) via Eq. (8). It takes \( O(|E|) \) time and \( O(|V|) \) memory to compute \( v_k \) per iteration, which is dominated by the matrix-vector multiplication \((W^T \cdot v_{l-1})\). Once \( v_k \) is derived, \( v_{l-1} \) and \( u_{l-1} \) are freed. Thus, this phase takes \( O(k|E|) \) time and \( O(|V|) \) memory for \( k \) iterations.
(3) Computing $\|x_k\|_2$ from $v_k$ via Eq.(7). This phase requires $O(V)$ time and $O(V)$ memory for vector scaling.

Taking (1)-(3) together, the total complexity is bounded by $O(k[E])$ time and $O(k[V]+[E])$ memory.

In Section 3.2, this complexity will be further improved.

### 3.1.7 Applying to Partial-Pairs SimRank

We now apply the “seed germination” model of Theorem 1 to partial-pairs SimRank.

Let us first introduce the following notations. Given two subsets $A$ and $B$ of nodes in $V$, for any $|V| \times |V|$ matrix $X$, we denote by $[X]_{A,B}$ the $|A| \times |B|$ submatrix of $X$ that lies on the intersection of each row in $A$ with each column in $B$; $[X]_{A,*}$, the $|A| \times |V|$ submatrix of $X$ selecting all rows in $A$; $[X]_{*,B}$ the $|V| \times |B|$ submatrix of $X$ selecting all columns in $B$. For instance, given subsets $A = \{1, 3\}$ and $B = \{2, 4\}$ of node set $V = \{1, 2, 3, 4\}$, and SimRank matrix $S \in \mathbb{R}^{4 \times 4}$, below is the partial-pairs SimRank matrix $[S]_{A,B}$:

$$S = \begin{bmatrix} s_{11} & s_{12} & s_{13} & s_{14} \\ s_{21} & s_{22} & s_{23} & s_{24} \\ s_{31} & s_{32} & s_{33} & s_{34} \\ s_{41} & s_{42} & s_{43} & s_{44} \end{bmatrix} \Rightarrow [S]_{A,B} = \begin{bmatrix} s_{11} & s_{12} & s_{13} & s_{14} \\ s_{21} & s_{22} & s_{23} & s_{24} \end{bmatrix}$$

We now consider the “seed germination” iteration for partial-pairs SimRank. Recall that its counterpart for single-source SimRank is based on two main ideas: one is the “right-to-left association” trick that avoids matrix-matrix multiplications; the other is to skip duplicate computation across the summations of Eq.(6) by tallying compacted paths in a “seed germination” manner. Indeed, the latter idea can be readily ported to partial-pairs SimRank, whereas the former is not a natural extension since it is unclear whether the “right-to-left association” is still well suited for partial-pairs SimRank. To shed light on this, we first provide the following lemma.

**Lemma 3.** Given two subsets $A$ and $B$ of a node set $V$, the partial-pairs SimRank matrix $[S]_{A,B}$ can be obtained from the entire $|V| \times |V|$ matrix $S$ via the transformation below:

$$[S]_{A,B} = [I]_{A,*} \cdot S \cdot [I]_{*,B}^T$$

(13)

(13) (Please refer to Appendix A.1 for a detailed proof.)

Intuitively, to transform $S \in \mathbb{R}^{||V|| \times |V|}$ to $[S]_{A,B} \in \mathbb{R}^{||A|| \times |B|}$, Lemma 3 constructs $[I]_{A,*} \in \mathbb{R}^{||A|| \times |V|}$ (“row-selector”) and $[I]_{*,B} \in \mathbb{R}^{||V|| \times |B|}$ (“column-selector”) such that Eq.(13) holds.

For example in Eq.(12), we find $[I]_{A,*} = [e_1 \ | \ e_3]^T \in \mathbb{R}^{2 \times 4}$ and $[I]_{*,B} = [e_1 \ | \ e_2 \ | \ e_4]^T \in \mathbb{R}^{4 \times 3}$ s.t. $[S]_{A,B} = [I]_{A,*}^T S [I]_{*,B}$.

Lemma 3 is introduced for efficiently computing $[S]_{A,B}$. To obtain an Eq.(6)-like formula for partial-pairs SimRank, we first pre-multiply by $[I]_{A,*}$ and post-multiply by $[I]_{*,B}$ on both sides of Eq.(5), and then apply Lemma 3, which yields

$$[S_k]_{A,B} = (1 - C)([I]_{A,*} + \sum_{l=1}^k C^l [W]_{A,*} \cdot ([W]^{-1} \cdot [W]_{A,*}))$$

However, unlike single-source SimRank, the “right-to-left association” trick to Eq.(6) may or may not apply to Eq.(14), depending on the sizes of node collections $A$ and $B$, and the graph structure. This is because both $[W]_{A,*}$ and $[W]_{A,*}$ in Eq.(14) are rectangular matrices. As the matrix multiplication satisfies the associative law, for each fixed $l$ in Eq.(14), the multiplication orders in $[W]_{A,*} \cdot ([W]^{-1} \cdot [W]_{A,*})$ can be tactically adjusted to reduce its computational cost.

**Lemma 4.** For every fixed $l$ in the summation of Eq.(14), $[W]_{A,*} \cdot ([W]^{-1} \cdot [W]_{A,*})$ can be efficiently computed in $O(2(l-1) \cdot \min(|A|, |B|) \cdot |E| + |A||B|)$ worst-case time, by grouping all the matrix multiplications from “left-to-right” (resp. “right-to-left”) if $|A| < |B|$ (resp. $|A| \geq |B|$).

**Proof.** We shall consider the following three cases:

1. If multiplications are grouped from “left-to-right” as

$$(((...([W_{A,*} \cdot ([W]^{-1} \cdot [W]_{A,*})] \cdot ...)) \cdot [W]_{A,*}))$$

then the total time is bounded by $O(2(l-1) \cdot |A||B|)$.

2. If multiplications are grouped from “right-to-left” as

$$([W]_{A,*} \cdot ([W]^{-1} \cdot ([W]_{A,*} \cdot ...)))$$

then the total time is bounded by $O(2(l-1)|A||B|)$.

3. If multiplications are grouped from “both ends” to meet at a contain position $p$ ($p = 2, 3, \cdots, 2(l-1)$) as

$$(((...([W_{A,*} \cdot ([W]^{-1} \cdot [W]_{A,*})] \cdot ...) \cdot [W]_{A,*}))$$

then it needs $O(p-1)\cdot |A||B| + (2l - p - 1)\cdot |B||E| + |V||A||B|$ time in total. Note that the last part requires $O(|V||A||B|)$ time (in lieu of $O(|A||B|)$) since either of the two remaining rectangular matrices may not be sparse.

Apart from the above cases, other grouping orders of matrix multiplications beyond our consideration (e.g., grouping from a position $p \in [2, 2(l-1)]$ to two ends) would incur excessive costs, since at least $O(|V||E|)$ time is needed for every multiplication of two $|V| \times |V|$ matrices, which will dominate the total time.

Based on the above analysis, it suffices to show that the optimum time is achieved by Case 1 or Case 2. Indeed, the time of Case 3 can be written as $O(f(p))$, with

$$f(p) = (p - 1)|A||B| + (2l - p - 1)|B||E| + |V||A||B|$$

and $z := (2l - 1)|B| - |A| + |V||A||B|$. Thus, the minimum value of $f(p)$ occurs at $p = \{\frac{2l}{2}, \text{ if } |A| \leq |B|; \frac{2(l-1)}{2}, \text{ if } |A| > |B|\}$.

We can verify the time of Cases (1) and (2) is better than $O(f(2l-1))$ and $O(f(2))$, respectively. Thus, the optimum time cannot be achieved by Case (3), $\forall p \in [2, 2(l-1)]$.

**Lemma 4** implies that a unidirectional order of grouping all matrix multiplications in $[W]_{A,*} \cdot ([W]^{-1} \cdot [W]_{A,*})$ can attain the optimum computational time. It comprises our aforementioned “right-to-left association” trick of single-source SimRank as a special case when $(A, B) := (V, \emptyset)$.

In what follows, we can tacitly assume that $|A| \geq |B|$; otherwise, the given $A$ and $B$ can be swapped without affecting the results, due to SimRank symmetry ($s(a,b) = s(b,a)$).

With this assumption, the unidirectional grouping order in Lemma 4 refers particularly to “right-to-left association”, the same as the grouping order for single-source SimRank. As such, our method of tallying paths in a compact tree for single-source SimRank can be readily ported to partial-pairs SimRank, as illustrated in Theorem 3.

**Theorem 3.** Given two subsets $A$ and $B$ of nodes in $V$ (we assume $|A| \geq |B|$ without loss of generality), the partial-pairs SimRank $[S_k]_{A,B}$ at iteration $k$ can be computed as

$$[S_k]_{A,B} = (1 - C) \cdot (C \cdot [W]_{A,*} \cdot [V]_{k-1} + [I]_{A,B})$$

(15)
where $V_{k-1} \in \mathbb{R}^{[V \times \{1\}]^k}$ is iterated as: $\forall l = 1, \ldots, k - 1$

\[
V_l = C \cdot W^T \cdot V_{l-1} + U_{k-1} \quad \text{with } V_0 = U_k \quad (16)
\]

after $U_0, U_1, \ldots, U_k \in \mathbb{R}^{[V \times \{1\}]^k}$ are iterated as: $\forall l = 1, \ldots, k$

\[
U_l = W \cdot U_{l-1} \quad \text{with } U_0 = I, \quad (17)
\]

(The proof is similar to that of Theorem 1. We omit it here.)

Theorem 3 provides an algorithm, referred to as Par-SR, to iteratively assess partial-pairs SimRank $[S_{k}]_{A,B}$ in a self-contained manner. One can easily verify by direct manipulation that Par-SR correctly computes partial-pairs SimRank.

### 3.1.8 Complexity of Partial-Pairs SimRank

We now analyze the complexity of Par-SR in Algorithm 1.

**Theorem 4.** Given a graph $G = (V, E)$ and two subsets of nodes $A$ and $B$, Par-SR needs $O(k|E| \min(|A|, |B|))$ time and $O(|V| + |E|)$ memory to compute $[S_{k}]_{A,B}$.

**Proof.** Without loss of generality, we shall only consider the case when $|A| \geq |B|$. For each $j \in B$, Par-SR consists of three phases: (a) compute $u_0, \ldots, u_k$ (Lines 5–6), (b) obtain $v_0$ (Lines 8–10), and (c) assess $[S_{k}]_{A,j}$ (Lines 11–12). In each phase, it can be verified that $O(k|E|)$, $O(k|E|)$ and $O(|d(A)|)$ time (resp. $O(k|V|)$, $O(k|V|)$ and $O(|V|)$ memory) are required. Thus, the total complexity is bounded by $O(k|B||E|)$ (resp. $O(k|A||E|)$) time and $O(|V| + |E|)$ memory for $|A| \geq |B|$ (resp. $|A| < |B|$).

It is worth mentioning that, for achieving high memory efficiency, for every column index $j \in B$, we first compute $[U_0]_{i,j}, \ldots, [U_k]_{i,j} \in \mathbb{R}^{[V \times \{1\}]^k}$ in Eq.(16) as: $\forall l = 1, \ldots, k$

\[
[U_{l-1}]_{i,j} = W \cdot [U_{l-1}]_{i,j} + [U_0]_{i,j} = e_j \quad (18)
\]

Using columns $[U_0]_{i,j}, \ldots, [U_k]_{i,j}$ (not matrices $U_0, \ldots, U_k$), we then solve $[V_{l}]_{i,j}$ in Eq.(16) as: $\forall l = 1, \ldots, k - 1$

\[
[V_{l-1}]_{i,j} = C \cdot W^T \cdot [V_{l-1}]_{i,j} + [U_{l-1}]_{i,j} \quad \text{with } [V_0]_{i,j} = [U_0]_{i,j} \quad (19)
\]

Once $[V_{l}]_{i,j}$ is computed, $[U_0]_{i,j}, \ldots, [U_k]_{i,j}$ are all freed. This process continues until each $j \in B$ is walked through.

Compared with Eqs.(16) and (17), our trick in Eqs.(18) and (19) that converts matrix-matrix multiplications (e.g., $W \cdot U_{l-1}$) into matrix-vector multiplications (e.g., $W \cdot U_{l-1} \cdot e_j$) has high memory efficiency.

**3.1.9 Iteration Accuracy**

The following theorem shows that our “seed germination” iterative model can achieve exactly the same accuracy as the conventional iterative counterparts (e.g., [6,7,12,13,15,17]).

**Theorem 5.** For any iteration number $k = 0, 1, \ldots$, the gap between the partial-pairs SimRank $[S_{k}]_{A,B}$ in Algorithm 1 and the exact one $[S]_{A,B}$ is bounded by

\[
\| [S_{k}]_{A,B} - [S]_{A,B} \| \leq C^{k+1},
\]

where $\|X\|_\infty := \max_{i,j} \{ |X_{i,j}| \}$ is the entry-wise max norm.

**Proof.** It follows from Eq.(14) that $\| [S_{k}]_{A,B} - [S]_{A,B} \| \leq (1-C) \sum_{i=1}^{k} C^i \| [W^T]_{A,i} \cdot [W^T]_{A,i} \cdot \cdots \cdot [W^T]_{A,i} \| \leq C^{k+1}. \quad \Box$

Theorem 5 implies that, for attaining a given accuracy $\epsilon$, the total number of iterations for Par-SR is $k = \lceil \log_\epsilon(1) \rceil - 1$. In practice, since the decay factor $C = 0.6$ is a preferable setting [13], a typical value of $k$ is 9 suffices to guarantee $\epsilon = 0.9^{k+1} = 0.006 < 0.01$ accurate to two decimal places.

Indeed, the effect of $k$ is a speed-accuracy trade-off. The higher $k$ indicates more complexity to compute summands of Eq.(14), but makes $[S_{k}]_{A,B}$ even closer to the exact $[S]_{A,B}$. This is consistent with our interpretation of the “seed germination” model in Figure 2, where we have discerned that the increasing of $k$ implies more paths $p_{ij}$ to be merged into a compact tree $p'$, leading to high accuracy yet more costs.

### 3.2 Eliminating Unnecessary Edge Access

After the “seed germination” iteration has been devised to assess partial-pairs SimRank in $O(|E| \min(|A|, |B|))$ time, our pruning strategy in this section will further reduce the time to $O(m \min(|A|, |B|))$, with $m \leq \min(|k|, |E|, \Delta^{2k})$.

**3.2.1 Traditional SpMxM Redundancy**

Recall that the computational time of Par-SR is dominated by the matrix multiplications ($W \cdot U_{l-1}$) in Eq.(17) and ($W^T \cdot V_{l-1}$) in Eq.(16). In Section 3.1, we simply adopt standard sparse matrix multiplications (SpMxM), by viewing $W \in \mathbb{R}^{[V \times \{1\}]^k}$ as a sparse matrix with $|E|$ nonzero.

However, there exists unnecessary edge access in SpMxM.

To clarify this, let us interpret SpMxM in the context of our “seed germination” paradigm.\(^5\) When SpMxM is used to compute $W \cdot [U_{l-1} \cdot e_j]$, it can be written entry-wise as:

\[
[U_{l-1}]_{i,j} = \sum_{i \in V(j)} W_{i,j} \cdot [U_{l-1}]_{i,j} \quad \text{with } [U_0]_{i,j} = \begin{cases} 1, & \text{if } j = i, \\ 0, & \text{otherwise} \end{cases}, \quad \forall i, \forall j (20)
\]

where $O(i)$ is the out-neighbor set of node $i$. Consequently, $\forall i \in V(j)$, $O(i) = |E|$ edges are accessed to compute $W \cdot [U_{l-1}]_{i,j}$ via SpMxM, which contains redundancy in general.

**Example 2.** Recall graph $G$ in Figure 1, and our iterative process of finding its $[S]_{A,B}$ with $A = \{3,4\}$ and $B = \{2,5\}$. According to Theorem 3, when SpMxM is directly applied to $[S]_{A,B}$ with $A = \{3,4\}$ and $B = \{2,5\}$.

\(^5\) Real graphs are often sparse with $|E| < |V|^2$ in practice.

\(^6\) For space interests, our focus will be devoted to $W \cdot U_{l-1}$ in Eq.(17), which also suits ($W^T \cdot V_{l-1}$) in Eq.(16).
zero, since successive substitution applied to Eq. (18) yields that the incoming edges of node $x$ are the "wild edges" accessed for computing $\langle W \cdot U, i \rangle$ from query node $j$. Thus, there are no length-($l-1$) paths from $x$ to $j$ whenever $[U_{l-1}]_{x,j} = 0$. This tells us that all in-links of node $x \in V \setminus D_l$ are not originally germinated from query $j$ with $l$ steps; they are "wild edges" that can be safely pruned when $(W \cdot U_{1:l-1}, j)$ is computed.

**Example 3.** Recall Example 2 and the iterative process in Figure 5: $[U_{2}]_{x,j} := W \cdot [U_{1}]_{x,j}$ with $[U_{1}]_{x,j} := [W]_{x,j}$. Using our method of Eq. (21) to compute $(W \cdot U_{1:l-1}, j)$ needs only 2 edges access (denoted as blue solid arrow in Figure 4), corresponding to 2 multiplications, i.e.,

$\alpha_\star \cdot [W]_{x,\star} + \alpha_\star \cdot [W]_{j,\star}$

as pictured in Figure 5, where symbols $\times$, $\ast$, $\bullet$, $\circ$ in a square denote nonzero entries. Precisely, given $[U_1]_{x,j} := [W]_{x,j}$, Eq. (21) first finds $D_2 := \{x \in V \mid [U_1]_{x,j} \neq 0\} = \{v_2, v_6\}$ (framed in red dotted line in Figure 5), implying that all the in-links of node $x \not\in D_2$ are "wild edges" (in dashed arrows) that are not originally germinated from query $v_5$ with 2 steps. Then, Eq. (21) prunes off such "wild edges", by converting $(W \cdot U_{1:l-1})$ to a combination of only two columns $[W]_{1:b, \ast}$ and $[W]_{1:b, \star}$, with respective coefficients $[U_{1:l-1}]_{x, \ast}$ and $[U_{1:l-1}]_{x, \star}$, as visualized in Figure 5. Thus, computing $(W \cdot U_{1:l})$ via Eq. (21) needs only 2 multiplications, as opposed to SpMM$^3$M scanning all nonzeros of $W$ with $[E] = 8$ multiplications.

### 3.2.3 Hashing Strategy

To efficiently compute the combinations $\sum_{x \in D_l, \ast} \alpha_x \cdot [W]_{x, \ast}$ in Eq. (21), besides using compressed column storage for $W$, we also leverage a hashing strategy to eliminate unnecessary zero entry-wise vector additions. Precisely, we first build an empty hash table in such a way that the hash maps the node index $i$ (hash key) to its value $[U_i]_{x,j}$ (the result of $\sum_{x \in D_l, \ast} \alpha_x \cdot [W]_{x, \ast}$). To obtain $[U_i]_{x,j}$, for each $x \in D_l$, we pick all nonzero entries of $[W]_{x, \ast}$, one by one, and search the hash table for the picked entry. If the entry exists in the table, we increase its value by $(\alpha_x \cdot [W]_{x, \ast})$; otherwise, we enter it in the hash table with the initial value $(\alpha_x \cdot [W]_{x, \ast})$. After all entries for $x \in D_l$ are entered in the hash table, the final result of $\sum_{x \in D_l} \alpha_x \cdot [W]_{x, \ast}$ is derived. For example in Figure 5, our hashing method of computing $(W \cdot U_{1:l})$ via Eq. (21) only needs 2 operations to enter $[U_{1:l}]_{x, \ast}$ and $[U_{1:l}]_{x, \star}$ in the hash table.

### 3.2.4 Pruning Algorithm

By integrating our pruning and hashing methods into the partial-pairs SimRank, we now provide an enhanced version of Par-SR, referred to as PrunPar-SR, in Algorithm 2.

One can readily verify, by direct manipulation and Eq. (21), that PrunPar-SR (1) correctly prunes redundant edge access with a-priori zero scores in $[S_x]_{A,B}$, and (2) returns exactly the same accuracy as Par-SR.

Regarding its complexity, we have the following theorem.

**Theorem 6.** Given a graph $G = (V, E)$ and two subsets of nodes $A$ and $B$, PrunPar-SR requires $O(m \min(|A|, |B|))$ worst-case time and $O(|E| + k|V|)$ memory for $k$ iterations, with $m \leq \min\{k|E|, \Delta^{2k}\}$ and $\Delta$ the maximum degree of $G$.

**Proof.** We can readily verify that the total complexity of PrunPar-SR is dominated by Lines 6-7 and Lines 10-11.

First, we show that, for every $l$ Lines 6-7 requires $O(m_l)$ time in the worst case, with $m_l \leq \min\{|E|, \Delta^l\}$.
Algorithm 2: PrunPar-SR (G, C, k, A, B)

Input/Output: the same as Lines 1–2 of Algorithm 1;

1-2 the same as Lines 1–2 of Algorithm 1;

3 foreach j ∈ B do
4 initialize u₀ := e_j;  
5 for l = 1, . . . , k do
6 set D := \{x ∈ V | u[l−1] = 0 and I(x) ≠ ∅\};  
7 u_l := \{v ∈ D | W_{l−1→l} via hashing;  
8 initialize v₀ := u_l;  
9 for l = 1, . . . , k − 1 do
10 set D := \{x ∈ V | v[l−1] = 0 and O(x) ≠ ∅\};  
11 v_l := (C − \sum_{e ∈ D} v[l−1] D W_{l, u−l−1} + u[l−1]) via hashing;  
12 free v[l−1] and u[l−1];  
13 set D := \{x ∈ V | v[l−1] = 0 and O(x) ≠ ∅\};  
14 S[l−1,j] := (1 − C) \cdot (C − \sum_{e ∈ D} [v[l−1] D W]_{l, A} + [I]_{A,j}) via hashing;  
15 free v[l−1];  
16 end  
17 the same as Lines 13–14 of Algorithm 1;

(i) On one hand, for each x ∈ D, it takes O(I(x)) time to compute (a_x \cdot |V|). Since |I(x)| is the indegree of x, the total time of Lines 6–7 is bounded by

O(\sum_{e ∈ D} |I(x)|) ≤ O(\sum_{e ∈ D} I(x)) = O(^{(E)}.)

(ii) On the other hand, let \{u[l−1] ≠ 0\}. Then, computing W \cdot u[l−1] via Line 7 implies that

|u[l−1]| ≤ \min\{\Delta \cdot |V|\} with |u[0]| = 1. \hspace{1cm} (22)

Hence, the cost of computing (W \cdot u[l−1]) via Line 7 requires linear combinations of k−1 columns of W in the worst case. Since there are at most \Delta nonzeros entries (i.e., multiplications via hashing) in each column, the total time is O(^{(E)}.).

Taking (i) and (ii) together, Lines 6–7 require O(m) total time in the worst case, with m ≤ O(\Delta i). Similarly, we can show that, for every i, Lines 10–11 yield O(\Delta i) time in the worst case, where \Delta i ≤ O(\Delta i).

This is because (W \cdot u[l−1]) can be viewed as a linear combination of \{u[l−1] \neq 0\} columns of W; each column has at most \Delta nonzero entries. Thus, in the worst case when the position of each nonzero in each column is different from one another, there are at most \min\{\Delta \cdot |V|\} nonzero entries in u[l−1]. Successive substitution applied to Eq.(22) yields

|u[l−1]| ≤ \min\{\Delta i, |V|\}.

Hence, the cost of computing (W \cdot u[l−1]) via Line 7 requires linear combinations of k−1 columns of W in the worst case. Since there are at most \Delta nonzeros entries (i.e., multiplications via hashing) in each column, the total time is O(^{(E)}.).

Theorem 7. Given two subsets A and B of nodes in V (we assume |A| ≥ |B| without loss of generality), the partial-par SimRank at iteration k can be computed as

\[ S_{k} = S_{k−1} \cdot (W^T \cdot S) + (1 − C) \cdot \mathbf{I}. \] (23)

where \( V_{k−1} \in \mathbb{R}^{|V|\times|B|} \) can be iterated column by column independently as follows: for j ∈ B, \( V_l = 1, . . . , k − 1, \)

\[ |V_{l−1,j} = W^T \cdot V_{l−1,j} + [U_{l−1}]_{A,B} \quad \text{with } |V_{0,j} = |U_{0,j} = 1 \] (24)

It is worth mentioning that the O(min{|A|, |B|}) time of PrunPar-SR with m = min(|k|E, \Delta^2 k) on dense graphs\(^7\), has the same complexity bound of Par-SR. However, in practice, PrunPar-SR runs much faster than Par-SR, especially in the first several iterations. In Section 5, we shall further experimentally evaluate the speedup of PrunPar-SR.

4. Extensions

We next provide three other consequences of PrunPar-SR.

4.1 Improving Single-Source SimRank

For any query j in a graph G = (V, E), by setting A := V and B := {j} in PrunPar-SR, we have the following corollary.

Corollary 1. For any query j in a graph G = (V, E), it is in \( O(\min\{|k|E|, \Delta^2 k\}) \) time and \( O(|E| + |V|) \) memory to compute single-source SimRank \( s_k(x, j) \) for k iterations.

In contrast to the best-known single-source SimRank [4, 9, 10], Corollary 1 has the following advantages: (1) Unlike the \( O(\Delta^2 k) \) time of [10] that increases exponentially w.r.t. k, our method scales well to \(|E|\). (2) The Monte Carlo method of [9] delivers probabilistic results, whereas our algorithm is deterministic. (3) [4] needs \( O(|E|^2) \) preprocessing time for a low-rank factorization, but our method has no such costs.

4.2 Reducing Memory for All-Pairs SimRank

By setting A := V and B := V, another variant of PrunPar-SR is a memory-efficient algorithm for all-pairs SimRank.

Corollary 2. Given a graph G = (V, E), all-pairs SimRank \( s_k(x, y) \) can be computed in \( O(\min\{|k|E|V|, \Delta^2 k|V|\}) \) time and \( O(|E| + |V|) \) memory for k iterations.

Compared with the best-known all-pairs SimRank [15] that takes \( O(kd^2|V|^2 \Delta^4) \) time with \( d' \leq |E|/|V| \), Corollary 2 achieves high computational efficiency. Moreover, it breaks the “high iteration coupling” barrier of the conventional SimRank that requires to store \(|O(|V|^2)| \) scores from previous iterations. In addition, our method is easy to be parallelized since each column \( S_k \) can be computed simultaneously.

4.3 Porting to Partial-Pairs SimRank

Recently, the previous work of [16] has found the “zero-similarity” problem of the existing SimRank model, namely, two nodes a and b are assessed as dissimilar (i.e., \( s(a, b) = 0 \)) if there are no nodes with equal distance to both a and b. To resolve this problem, SimRank* has been proposed:

\[ S = S \cdot (S \cdot W \cdot W^T) \cdot (1 − C) \cdot \mathbf{I}. \] (23)

However, the existing iterative method of SimRank* [16] also has a SimRank-like “high iteration coupling” barrier. To fix this problem, we now present a “seed-germination” iterative model for partial-pairs SimRank computation.

Theorem 7. Given two subsets A and B of nodes in V (we assume |A| ≥ |B| without loss of generality), the partial-pair SimRank* \( S_{k,A,B} \) at iteration k can be computed as

\[ S_{k,A,B} = (1 − C) \cdot (W^T \cdot S_{k−1}) + [I]_{A,B} \] (24)

where \( V_{k−1} \in \mathbb{R}^{|V|\times|B|} \) can be iterated column by column independently as follows: for j ∈ B, \( V_l = 1, . . . , k − 1, \)

\[ |V_{l−1,j} = W^T \cdot V_{l−1,j} + [U_{l−1}]_{A,B} \quad \text{with } |V_{0,j} = |U_{0,j} = 1 \] (25)

\(^7\)Here, a dense graph refers to a graph with \(|E| = O(|V|^2)\).
5. EXPERIMENTS

Our experiments on real and synthetic data will evaluate (1) the time, memory and accuracy of our partial-pairs SimRank, and (2) the high efficiency of its extension to single-source and all-pairs SimRank, and SimRank* model [16].

5.1 Experimental Setting

We use both real and synthetic data in our experiments. (1) Real Data. For efficiency evaluation, we use 6 graphs8:

(a) P2P, a Gnutella peer-to-peer file sharing network, where each node is a host labeled with the number of its connected hosts (categorized into e.g., leaf, ultraspeers), and an edge a connection from one host to another in the graph topology.

(b) DBLP, a co-authorship DBLP graph, where a node is an author labeled with his expertise (e.g., DB, IR) categorized by his conference publications. An edge is a co-authorship.

(c) WebS, a web graph from stanford.edu, where each node is a page, labeled with its “importance” ordered by the PageRank values that are split into 200 equal-sized buckets. Each directed edge is a hyperlink between them.

(d) AM, an Amazon product co-purchasing graph, where a node is a product labeled with product category, and rating. An edge links products a and b if a is co-purchased with b.

(e) CitP, a US patent network, in which a node is a patent labeled with grant date, country of first inventor, technological category. Edges are directed citations made by patents.

(f) SocL, a LiveJournal social network, in which nodes are members labeled with their communities, countries, and ages (crawled from TArtu site9), and edges are friendships.

The size of $|G(V, E)|$ of the graphs is as follows.

| Graph | $|V|$ | $|E|$ |
|-------|------|-----|
| P2P   | 2.1T | 6.3K |
| DBLP  | 1.2k | 20k  |
| AM    | 8.8M | 400k |
| DB    | 9.6M | 90M  |
| WebS  | 2.6M | 26k  |
| CitP  | 2.6M | 5.4M |

5.2 Synthetic Data. To produce synthetic network SYN, we use a generator GTgraph10, controlled by $|V|$ and $|E|$.

5.3 Query Generator. (a) For partial-pairs $\{s(a, b) | a, b \in A, a \neq b\}$ assessment, we generate query pairs $(A, B)$ as follows: For queries on real graphs, the labels are taken from the datasets; for synthetic graphs, we randomly sample $A$ and $B$ from $V$, controlled by their size $|A|$ and $|B|$. To ensure that the selected nodes in $A$ and $B$ can comprehensively cover a broad range of any possible queries, we first divide all nodes ordered by their density $(|E|/|V|)$ into 10 equal-sized buckets, and then randomly sample $|A|$ and $|B|$ nodes, respectively, from each bucket. (b) For single-source $s(\star, j)$ assessment on both real and synthetic graphs, we randomly select query $j$ from $V$ in a similar way. For every experiment, the average performance is reported over all test queries.

5.4 Algorithms. We implement the following, all in VC++:

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>PrunPar-SR</td>
<td>our algorithm in Sect. 3.2, with pruning</td>
<td>partial pairs</td>
</tr>
<tr>
<td>PrunPar-SR*</td>
<td>variation of PrunPar-SR ported to SimRank</td>
<td>single pairs</td>
</tr>
<tr>
<td>SJR</td>
<td>SimRank-based similarity join</td>
<td>all pairs</td>
</tr>
<tr>
<td>TopSim-M</td>
<td>top-K random walk based SimRank</td>
<td>all pairs</td>
</tr>
<tr>
<td>Memo-SR*</td>
<td>edge concentration SimRank*</td>
<td>all pairs</td>
</tr>
</tbody>
</table>

5.5 Parameters. We set the following default parameters:

(a) $C = 0.6$, the decay factor suggested in [13]. (b) $k = 10$, the number of iterations that ensures $s_i(\star, \star)$ accurate to 2 decimal places as $C^{k+1} = 0.6^{10+1} < 0.01$, according to [13].

(6) Accuracy Metric. For accuracy comparison, we adopt Normalized Discounted Cumulative Gain (NDCG) at position $p$ w.r.t. query $j$. NDCG$_p(j) = \frac{1}{\log(2+j)} \sum_{i=1}^{p} 2^{s_i(j)}$, where $s_i(j)$ is the similarity score between nodes $i$ and $j$, and IDC$_p(j)$ is a normalized factor ensuring the “true” NDCG ordering to be 1. Since all the compared algorithms come from SimRank and SimRank* families, we can choose all-pairs scores of [13] and [16] as their baselines.

We run algorithms PrunPar-SR, Par-SR, SJR, and SingPair on six real datasets. By randomly issuing different queries, we compare their total time of assessing partial-pairs SimRank. Figure 6a depicts the results. (1) On each dataset, PrunPar-SR and Par-SR are always faster than SJR and SingPair. This is because our “seed germination” iterative model can merge repeated node access into a compact tree, significantly reducing the cost in the summations. (2) On P2P and CitP, PrunPar-SR is almost 5x faster than Par-SR. This suggests that our pruning is powerful on small density graphs, which is consistent with our analysis in Section 3.2. (3) When the data size becomes larger, the time of each algorithm increases in general. However, SJR will crash on large AM, CitP, SocL, due to its expensive cost for finding an $h$-go cover on the tensor graph. SingPair only survives on small P2P due to large memory. However, PrunPar-SR and Par-SR scale well to large graphs, which is consistent with our complexity analysis in Section 3.

Figure 6b compares the time of PrunPar-SR, Par-SR, SJR with respect to the different query pairs $(A, B)$ on DBLP. The results show that (1) for various queries, PrunPar-SR consistently outperforms Par-SR, SJR, (e.g., it is 2.4x, 39.6x faster than Par-SR, SJR, respectively, for query $(DB, DM)$.) (2) In all cases, SJR exhibits the worst performance since it computes every node-pair score in $(A, B)$ one by one without computation sharing, unlike our “seed germination” method that merges repetitive node access. (3) When the query size of $(A, B)$ grows, the time for PrunPar-SR, Par-SR increases this is in good agreement with our complexity analysis.

Figure 6c indicates how elapsed time changes with $|A|$ on WebS, fixing $|B| = 338$. The trend shows that (1) when $|A|$ grows, the time for PrunPar-SR, Par-SR grows accordingly; however, the growing ratio begins to reduce when $|A| > 10^3$.

---

8http://snap.stanford.edu/data/index.html
9http://community.livejournal.com/tartu/profile
10http://www.cse.psu.edu/~madduri/software/GTgraph/index.html
which is due to the “right-to-left association” trick in our “seed germination” model that swaps $A$ and $B$ if $|A| > |B|$. (2) SJR crashes as $|A| \geq 100$, whereas PrunPar-SR, Par-SR scale well with $|A|$. (3) The speedup of PrunPar-SR on DBLP is more obvious than that on WebS, due to small density.

Figure 6d depicts the effect of iteration $k$ on the PrunPar-SR and Par-SR time on WebS and CitP. We see that (1) the PrunPar-SR and Par-SR time is directly proportional to $k$. (2) When $k = 6.9$, the time difference between Par-SR and PrunPar-SR on CitP is more pronounced than that on WebS. This is because CitP has small average degree, compared with WebS, which agrees with our complexity predictions that $d$ has a large impact on the PrunPar-SR.

Figure 6e compares the time efficiency of the single-source SimRank algorithms on six real datasets when $k = 4$. The results show that (1) The PrunPar-SR and Par-SR time exhibit a similar trend to partial-pairs assessment in Figure 6a. (2) The speedup of PrunPar-SR is more obvious on P2P, DBLP, CitP due to their small average degree. (3) PrunPar-SR consistently outperforms TopSim-SM by $+9.2x$; however, on DBLP and CitP, TopSim-SM is faster than Par-SR. This is due to the small degrees of DBLP and CitP, implying that TopSim-SM is more sensitive to $d$ than Par-SR, given $k$.

Figure 6f presents how the time for PrunPar-SR, Par-SR, TopSim-SM changes with $k$ on CitP for single-source queries. (1) When $k < 6$, TopSim-SM is faster than Par-SR, however, when $k > 6$, Par-SR outperforms TopSim-SM. (2) When $k$ grows, the linear increasing trend of TopSim-SM in the log-log scale axis indicates that the TopSim-SM time will exponentially increase with respect to $k$ with fixing degree $d$, which highlights a limitation of TopSim-SM for computing scores of all nodes w.r.t. a query. In contrast, the PrunPar-SR time increases with $k < 6$, but this increase slows greatly when $k > 6$, due to its pruning on “seed germination” model.

Figure 6g compares the average time to compute each column of all-pairs SimRank matrix for PrunPar-SR, OIP and Psum, and the all-pairs SimRank* for PrunPar-SR*, Memo-SR* and Psum-SR*, respectively. For SimRank, OIP and Psum crash on AM and SocL due to their large memory for storing all-pairs scores from the previous iteration. For SimRank*, this also occurs on Memo-SR* and Psum-SR* for the same reason. However, PrunPar-SR* can execute on large graphs, due to our partial-pairs method that computes the entire SimRank matrix column by column independently, where each column computation can fit into memory.

Figure 6h depicts the effect of $d$ on the time for PrunPar-SR, Par-SR, TopSim-SM, SimMat on a synthetic dataset for single-source SimRank assessment with $k = 10$ and $|V| = 100K$. We can see that (1) when $d$ increases, TopSim-SM crashes for $d \geq 6$ due to the TopSim-SM time being too sensitive to $d$ when $k$ is large; however, for the remaining algorithms, the increasing trend is relatively steep w.r.t. $d$. (2) when $d$ is smaller ($< 5$), PrunPar-SR is much better than Par-SR, illustrating the effectiveness of our pruning technique on graphs with low density. (3) SimMat is less sensitive to $d$, due to its SVD that may destroy graph sparsity.
Figure 6i reports the pruning power of PrunPar-SR on six real datasets for partial-pair and single-source SimRank assessment, respectively. The pruning ratio is defined as 

\[ \left(1 - \frac{\text{of edges accessed by PrunPar-SR}}{\text{of edges accessed by Par-SR}}\right) \times 100\%. \]

The results show that the pruning power is more significant on graphs with small degrees (P2P, DBLP, CitP), but is less efficient on graphs with larger degrees (WebS, AM). The larger the degree \( d \), the fewer the number of nonzeros in \( U_k \) or \( U_k \) during our “seed germination” iteration, as expected.

5.2.2 Memory Efficiency

Figure 6j shows the memory efficiency of PrunPar-SR, Par-SR, SJR, SingPair for partial-pairs assessment on real data. We observe that SJR requires more memory than PrunPar-SR and Par-SR on P2P, DBLP, and it crashes on the remaining four datasets. This is because SJR needs to store the \( h \)-go cover set that could be very large on the tensor graph, whereas PrunPar-SR and Par-SR require to memoize only auxiliary \( U_k \) for \( k \) iterations.

Figure 6k shows the memory of PrunPar-SR and Par-SR w.r.t. the growing \( k \) on synthetic data, when query size of \((A, B)\) is fixed. It can be discerned that (1) when \( k \) grows, the memory consumption increases steadily. This is because PrunPar-SR and Par-SR require to memoize \( k \) intermediate \( U_k \) after \( k \) iterations. (2) Given \( k \), PrunPar-SR needs slightly more memory than Par-SR, due to its hashing strategy.

Figure 6l compares the memory for single-source SimRank on real data. For \( k = 4 \), TopSim-SM that computes all nodes w.r.t. a query only survives on small P2P and DBLP, which requires large memory because SimMat requires to be stored for each node \( x \). SimMat only survives on small P2P and DBLP as well since it requires considerable memory to store the decomposed matrices vis VSD. SJR crashes on all the datasets except for P2P because it needs to find an \( h \)-go cover set on a large tensor graph. In contrast, PrunPar-SR and Par-SR for single source SimRank are highly memory efficient since they only need to store \( k \) vectors.

Figure 6m shows the memory of the all-pairs SimRank and SimRank* algorithms on DBLP, AM, SocL. On AM and SocL, OIP and Psum crashes for SimRank, whereas Memo-SR* and Psum-SR* crash for SimRank* because they require quadric memory space to store all-pairs similarities from the previous iterations. In contrast, PrunPar-SR and PrunPar-SR* compute the whole similarity matrix column by column, requiring considerably less memory.

5.2.3 Accuracy

Figure 6n shows the accuracy of single-source algorithms for both SimRank and SimRank* on real datasets. We randomly select 1000 queries, use the average NDCG@50 as accuracy measure. Using the all-pairs similarities as the baselines, our results on all the datasets show that for SimRank (resp. SimRank*), PrunPar-SR, Par-SR, SJR, SingPair (resp. PrunPar-SR*, Memo-SR*) do not sacrifice accuracy for high computational efficiency. However, the accuracy for TopSim-SM and SimMat is slightly lower as their computational paradigms are based on top-K search.

Finally, Figure 6o shows the top-K size affects the NDCG\(_K\) of PrunPar-SR, Par-SR, TopSim-SM. The results show that with increasing size of top-K, the accuracy of TopSim-SM is gradually reduced, compared with PrunPar-SR and Par-SR which do not compromise accuracy at all, as expected.

6. CONCLUSIONS

This paper focuses on efficient computation of partial-pairs SimRank. (1) A “seed germination” model is proposed to compute partial-pairs SimRank in \( O(k|E| \min\{|A|, |B|\}) \) time and \( O(|E| + k|V|) \) memory. (2) A pruning strategy is devised to skip redundant edges access for further speeding up partial-pairs computation to \( O(m \min\{|A|, |B|\}) \) time with \( m \leq \min\{|E|, \Delta^2\} \). As a by-product, our partial-pairs SimRank model not only as a special case improves the computation of the fastest known single-source SimRank, but also induces a memory-efficient algorithm for all-pairs SimRank that can break “high iteration coupling”. We also show that our techniques can be readily extended to partial-pairs SimRank* computation. Finally, our experimental results on real and synthetic data have verified the superiority of our algorithms on large graphs against the baselines.

Acknowledgment
This work forms part of the Big Data Technology for Smart Water Network research project funded by NEC Corporation, Japan.

7. REFERENCES

APPENDIX

A. DETAILED PROOFS

A.1 Proof of Lemma 3

Proof. Let \( A = \{a_1, \ldots, a_{|A|}\} \) and \( B = \{b_1, \ldots, b_{|B|}\} \) be two node subsets of \( V \). It can be directly verified that

\[
[I]_{A,B} \cdot [S]_{A,B} = [e_{a_1}]_{A,B} \cdot [e_{b_1}]_{A,B} = \sum_{i=1}^{k} \left( \sum_{j=1}^{k} \left( \sum_{l=0}^{k} f(i, l) \right) \right) \cdot \left( W^{\sigma} \right)_{A,B} \cdot [S]_{A,B}.
\]

First, for any \( j \in B \), Eq. (26) implies \([U]_{A,B} = \left( \sum_{i=1}^{k} \left( \sum_{j=1}^{k} \left( \sum_{l=0}^{k} f(i, l) \right) \right) \cdot \left( W^{\sigma} \right) \right)_{A,B} \cdot [S]_{A,B} \). Then, successive substitution yields to Eq. (27) produces \([U]_{A,B} = \left( \sum_{i=1}^{k} \left( \sum_{j=1}^{k} \left( \sum_{l=0}^{k} f(i, l) \right) \right) \cdot \left( W^{\sigma} \right) \right)_{A,B} \cdot [S]_{A,B} \).

Next, analogous to the proof of Theorem 3, one can verify that successive substitution used to Eqs. (24) and (25) yields \([S]_{A,B} = \left( \sum_{i=1}^{k} \left( \sum_{j=1}^{k} \left( \sum_{l=0}^{k} f(i, l) \right) \right) \cdot \left( W^{\sigma} \right) \right)_{A,B} \cdot [S]_{A,B} \).

Finally, interchanging the order of double summations, i.e., \( \sum_{j=1}^{k} \sum_{l=0}^{k} f(j, l) = \sum_{l=0}^{k} \sum_{j=1}^{k} f(j, l) \), gets our result.

\( \blacksquare \)

A.2 Proof of Theorem 7

Proof. We prove this by showing that successive substitution applied to Eq. (24)–(27) yields the power series form of SimRank* (see Eq. (7) of [16]), i.e.,

\[
[S]_{A,B} = \left( (1 - C) \cdot \sum_{l=0}^{k} \left( \sum_{j=0}^{k} \left( \sum_{i=0}^{k} f(i, j) \right) \right) \cdot (W^{\sigma}) \cdot W^{(-\sigma)} \right)_{A,B}.
\]

First, for any \( j \in B \), Eq. (26) implies \([U]_{A,B} = \left( \sum_{i=1}^{k} \left( \sum_{j=1}^{k} \left( \sum_{l=0}^{k} f(i, l) \right) \right) \cdot \left( W^{\sigma} \right) \right)_{A,B} \cdot [S]_{A,B} \). Then, successive substitution produces \([U]_{A,B} = \left( \sum_{i=1}^{k} \left( \sum_{j=1}^{k} \left( \sum_{l=0}^{k} f(i, l) \right) \right) \cdot \left( W^{\sigma} \right) \right)_{A,B} \cdot [S]_{A,B} \).

Next, analogous to the proof of Theorem 3, one can verify that successive substitution used to Eqs. (24) and (25) yields \([S]_{A,B} = \left( \sum_{i=1}^{k} \left( \sum_{j=1}^{k} \left( \sum_{l=0}^{k} f(i, l) \right) \right) \cdot \left( W^{\sigma} \right) \right)_{A,B} \cdot [S]_{A,B} \).

Finally, interchanging the order of double summations, i.e., \( \sum_{j=1}^{k} \sum_{l=0}^{k} f(j, l) = \sum_{l=0}^{k} \sum_{j=1}^{k} f(j, l) \), gets our result.

\( \blacksquare \)

B. ILLUSTRATIVE EXAMPLES

B.1 A Numerical Example of “Seed Germination” Model in Figure 3

We now detail the process to show how our “seed germination” iterative model computes the single-source SimRank scores \([S]_{a,v}\) with respect to query node \( v_5 \) at iteration \( k = 3 \) step by step in Figure 1. (given the damping factor \( C = 0.6 \))

There are six steps in total. They are depicted concisely in Figure 7, and explained in detail as follows:

1. We first select the query node \( v_5 \) as the “seed” node, and find all the in-neighbors of \( v_5 \) (i.e., \( \{v_3, v_4\} \)). Nodes \( \{v_3, v_4\} \) are the “bud” nodes germinated from \( v_5 \).

2. We next choose \( \{v_3, v_4\} \) as the “seed” nodes, and then find all their in-neighbours, which are \( \{v, v\} \). Nodes \( \{v, v\} \) are the “bud” nodes germinated from \( \{v_3, v_4\} \).

3. Similarly, we next choose \( \{v, v\} \) as the “seed” nodes, and then find all their in-neighbours, which is \( \{v_5\} \).

4. Since our goal is to compute \([S]_{a,v}\) for \( k = 3 \) iterations, after the above 3 steps we are now ready to select the new “seed” nodes and consider their out-neighbours.

5. We next choose \( \{v_3, v_4\} \) as the “seed” nodes, and then find all its “bud” nodes \( \{v_5\} \), and finally germinate all the in-neighbors of \( \{v_5\} \), which consists of two parts: (a) all the out-neighbors of \( \{v_3, v_4\} \), i.e., \( \{v, v\} \) and (b) the previously germinated nodes in Step 2, i.e., \( \{v_5\} \).

The resulting vector \( u_2 \) has only one nonzero entry in the 3rd element, associated with \( \{v_5\} \), i.e., all the in-neighbors of \( \{v_5\} \).

6. We next choose \( \{v_3, v_4\} \) as the new “seed” nodes, and then find all the in-neighbors of \( \{v_3, v_4\} \), which are \( \{v_3, v_4\} \). Both parts are combined into the nonzero entries of \( v_1 \).

Figure 7: A Numerical Example of “Seed Germination” Model for Computing \([S]_{a,v}\)
The resulting vector \( v_2 \) has 3 nonzero entries (the 2nd, 3rd, and 4th), associated with
\[
\{v_2, v_3, v_4\} = (\{v_2, v_4\} \cup \{v_2, v_3, v_4\}) \cup \{v_3, v_4\},
\]
where (1) the first part is \( \{v_2, v_4\} \cup \{v_2, v_3, v_4\} \), i.e., all the out-neighbors of “seed” nodes \( \{v_2, v_3, v_6\} \cup \{v_1, v_6\} \), which corresponds to all the nonzero elements of \( W^T \cdot v_1 \); (2) the second part is \( \{v_2, v_4\} \), i.e., the previously “germinated” nodes in Step 1, which corresponds to all nonzero entries of \( u_1 \). Both parts are combined into the nonzero entries of \( v_2 \).

Finally, we choose \( \{v_2, v_4\} \cup \{v_2, v_3, v_4\} \cup \{v_3, v_4\} \) (i.e., the old “bud” nodes in Step 5) as the new “seed” nodes, and then find all their new “bud” nodes
\[
\{v_2, v_3, v_6\} = (\{v_3 \cup \{v_2, v_5, v_6\} \cup \{v_2, v_5, v_6\}) \cup \{v_3\}
\]
which consists of two parts: (a) all the out-neighbors of \( \{v_2, v_4\} \cup \{v_2, v_3, v_4\} \cup \{v_3, v_4\} \), i.e.,
\[
\{v_3 \cup \{v_2, v_5, v_6\} \cup \{v_2, v_5, v_6\}
\]
and (b) the previously given query node \( v_5 \).

The resulting vector \( v_3 \) has 3 nonzero entries (the 2nd, 5th, and 6th), associated with
\[
\{v_2, v_3, v_6\} = (\{v_3 \cup \{v_2, v_5, v_6\} \cup \{v_2, v_5, v_6\}) \cup \{v_3\}
\]
where (1) the first part is \( \{v_3 \cup \{v_2, v_5, v_6\} \cup \{v_2, v_5, v_6\} \cup \{v_3\} \), i.e., all the out-neighbors of the “seed” nodes
\[
\{v_2, v_4\} \cup \{v_2, v_3, v_4\} \cup \{v_3, v_4\},
\]
corresponding to all nonzeros of \( W^T \cdot v_2 \); (2) the second part is \( \{v_3\} \), i.e., the previously given query node \( v_5 \), which corresponds to \( e_5 \). Both parts are combined into the nonzero entries of \( v_3 \).

Then, scaling the resulting vector \( v_3 \) produces the final result \( \hat{S}_{3, \cdot} \).

### B.2 An Example of Partial-Pairs SimRank*

**Example 4.** Given two subsets of nodes \( A = V \) and \( B = \{j\} \), and iteration number \( k = 3 \), Theorem 7 computes SimRank* scores \( \hat{S}_{3, \cdot} \) as follows. For notational convenience, we abuse \( u_i \) and \( v_i \) to denote \( [U]_{i, \cdot} \) and \( [V]_{i, \cdot} \), respectively.

First, Eq. (26) initializes \( u_0, \cdots, u_3 \) as
\[
u_0 := e_j, \quad u_1 := \left(\frac{C}{2}\right)^2 e_j, \quad u_2 := \left(\frac{C}{2}\right)^3 e_j.
\]

Then, by virtue of Eq. (27), \( u_0, \cdots, u_3 \) are updated as

<table>
<thead>
<tr>
<th>n</th>
<th>update ((u_{n-1})_{{0,\cdots,C}})</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>( u_0 = u_0 + W \cdot u_0 = \left(\frac{C}{2}\right) I + \left(\frac{C}{2}\right) W e_j )</td>
</tr>
<tr>
<td>1</td>
<td>( u_1 = u_1 + W \cdot u_0 = \left(\frac{C}{2}\right) I + \left(\frac{C}{2}\right) W + \left(\frac{C}{2}\right) W^2 e_j )</td>
</tr>
<tr>
<td>2</td>
<td>( u_2 = u_2 + W \cdot u_1 = \left(\frac{C}{2}\right) I + \left(\frac{C}{2}\right) W + \left(\frac{C}{2}\right) W^2 + \left(\frac{C}{2}\right) W^3 e_j )</td>
</tr>
<tr>
<td>3</td>
<td>( u_3 = u_3 + W \cdot u_2 = \left(\frac{C}{2}\right) I + \left(\frac{C}{2}\right) W + \left(\frac{C}{2}\right) W^2 + \left(\frac{C}{2}\right) W^3 + \left(\frac{C}{2}\right) W^4 e_j )</td>
</tr>
</tbody>
</table>

Next, using \( u_0, \cdots, u_3 \), Eq. (25) computes \( v_0, v_1, v_2 \) as

<table>
<thead>
<tr>
<th>n</th>
<th>update ((v_{n-1})_{{0,\cdots,C}})</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>( v_0 := v_0 = \left(\frac{C}{2}\right) e_j )</td>
</tr>
<tr>
<td>1</td>
<td>( v_1 := W^T \cdot v_0 + u_1 = \left(\frac{C}{2}\right) W^T + \left(\frac{C}{2}\right) W + \left(\frac{C}{2}\right) W^2 e_j )</td>
</tr>
<tr>
<td>2</td>
<td>( v_2 := W^T \cdot v_1 + u_2 = \left(\frac{C}{2}\right) W^2 + \left(\frac{C}{2}\right) W + \left(\frac{C}{2}\right) W^2 + \left(\frac{C}{2}\right) W^3 + \left(\frac{C}{2}\right) W^4 e_j )</td>
</tr>
</tbody>
</table>

Finally, \( \hat{S}_{3, \cdot} \) can be computed from \( v_2 \) by Eq. (24) as
\[
\hat{S}_{3, \cdot} = \left(1 - C\right) \cdot \left(W^T \cdot v_2 + e_j\right)
\]

\[
= \left(1 - C\right) \cdot \sum_{l=0}^{2} \left(\frac{C}{2}\right)^l \cdot \left(\begin{array}{c}2 \\
l \end{array}\right) \cdot \left(W^T\right)^{l-\alpha} \cdot W^\alpha \cdot e_j,
\]

which is the first 3rd sums of SimRank* power series. □