Efficient Top-K SimRank-based Similarity Join

Wenbo Tao Minghe Yu Guoliang Li
Department of Computer Science, Tsinghua University, Beijing, China
twb11@mails.tsinghua.edu.cn; yumh12@mails.tsinghua.edu.cn; liguoliang@tsinghua.edu.cn

ABSTRACT

SimRank is a popular and widely-adopted similarity measure to evaluate the similarity between nodes in a graph. It is time and space consuming to compute the SimRank similarities for all pairs of nodes, especially for large graphs. In real-world applications, users are only interested in the most similar pairs. To address this problem, in this paper we study the *top-k SimRank-based similarity join* problem, which finds k most similar pairs of nodes with the largest SimRank similarities among all possible pairs. We encode each node as a vector by summarizing its neighbors and transform the calculation of the SimRank similarity between two nodes to computing the dot product between the corresponding vectors. We devise an efficient two-step framework to compute top-k similar pairs using the vectors. For large graphs, exact algorithms cannot meet the high-performance requirement, and we also devise an approximate algorithm which can efficiently identify top-k similar pairs under user-specified accuracy requirement. Experiments on both real and synthetic datasets show our method achieves high performance and good scalability.

1. INTRODUCTION

With the proliferation of web search, clustering and collaborative filtering, identifying analogous nodes in a large graph has attracted unprecedented research attention. The similarity measures between pairs of nodes in a graph play an important role in many real-world applications such as friend recommendation and link prediction in social networks. SimRank is a popular and widely-adopted measure [7], which recursively computes the similarity between two nodes based on the similarities of their neighbors.

Nevertheless, it is rather challenging to devise efficient SimRank computation algorithms. First, the iterative method proposed by Jeh and Widom in [7] to solve the all pair SimRank problem that computes the similarities of all pairs of nodes has the time complexity of $O(\xi n^2 D^2)$, where n is the number of nodes in a graph, D is the average in-degree and ξ is the number of iterations, and will regress to $O(n^4)$ in the worst case. That is uncomputable for nowadays increasingly large networks. Second, although several recent works [17,19] have been proposed to optimize the inefficient all pair SimRank calculation, the state-of-the-art algorithm proposed in [11] still has the time complexity of $O(\xi n^3)$, which is still not scalable, especially for large graphs.

In real-world applications, users are only interested in *highly similar* node pairs. For example, social network systems want to identify similar users and recommend potential friends for users. In knowledge bases, we aim to find the similar concepts to facilitate concept/entity linking. To identify the *highly similar* node pairs, existing methods usually require users to input a similarity threshold and two nodes are similar if their SimRank value exceeds the threshold [10,20]. However, this threshold is not known and different applications have different thresholds. Moreover, it is rather hard to select an appropriate threshold because a large threshold leads to few results and a small threshold generates large numbers of irrelevant results. An appealing alternative is to find k most similar pairs of nodes with the largest SimRank similarities. We call this *top-k SimRank-based similarity join* (`SRK-Join`). The advantage of `SRK-Join` is obvious – it does not require a specified threshold and avoids a tedious step to tune the threshold.

To identify the most similar node pairs, we encode each node as a vector by summarizing its neighbors and transform the calculation of the SimRank similarity between two nodes to computing the dot product between the corresponding two vectors. Therefore, the `SRK-Join` problem is equivalent to identifying top-k pairs of vectors with the largest dot product values. The advantage of using the vector is that we can directly compute the SimRank value of two nodes using the two corresponding vectors and avoid the expensive iterative method which computes the SimRank values relying on similarities of other node pairs. In addition, we devise an efficient two-step framework to compute top-k similar pairs using the vectors. In the first phase, we identify a set of candidate nodes which the top-k similar pairs must be composed by. We propose effective techniques to reduce the number of candidate nodes to $2k$. In the second phase, we develop a tree-based `WAND` algorithm to efficiently identify the top-k similar pairs from the candidate nodes. For large graphs, exact algorithms cannot meet the high-performance requirement, and we also devise an approximate algorithm which can efficiently calculate the top-k similar pairs under user-specified accuracy requirement.

To summarize, we make the following contributions.

Proceedings of the VLDB Endowment, Vol. 8, No. 3

- We study the top-k SimRank-based similarity join problem. We novelly convert the SimRank similarity computation to the dot product calculation.
- We devise a two-step framework to identify top-k similar pairs of vectors with the largest dot product values. The first step identifies a set of candidate nodes and reduces the size of the candidate nodes to $2k$. The second step utilizes a tree-based `WAND` algorithm to efficiently identify the answers based on candidate nodes.
- We devise an approximate algorithm which can efficiently calculate the top-k similar pairs under user-specified accuracy requirement.
- We conduct extensive experiments on both real and synthetic datasets and the results show the good scalability and high performance of our method.

The rest of this paper is organized as follows. We formalize the problem in Section 2. We introduce the transformation of SimRank computation to dot product calculation in Section 3. Section 4 describes the two-step framework for answering the `SRK-Join` queries. An approximation algorithm is presented in Section 5 and the experimental results are reported in Section 6. Section 7 reviews the related work and Section 8 concludes the paper.

2. PRELIMINARIES

In this section, we first introduce the iterative model to compute SimRank in Section 2.1 and then discuss its equivalent random surfer model in Section 2.2. Finally we present the formal definition of our problem in Section 2.3.

2.1 Iterative Model

Consider a directed graph $G(V, E)$ with a set of nodes V and a set of directed edges E. We denote $|V| = n$ as the number of nodes in G and $|E| = m$ as the number of edges. Let D denote the average in-degree, i.e, $D = \frac{m}{n}$. For a node a, let $I(a)$ denote the set of its in-neighbors and $I_i(a)$ denote the i-th in-neighbor of a[1]. Given two nodes a and b, the SimRank similarity $S(a, b) \in [0, 1]$ is defined by Eq(1) :

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

where $C \in [0, 1]$ is a decay factor. The basic intuition behind Eq(1) is that two objects are similar if they are linked by similar objects. The initial case of the recursion is $S(a, a) = 1.0$ which indicates each node is completely similar to itself.

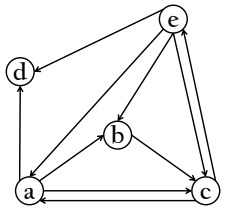
For directed acyclic graphs (`DAG`), we can solve the system of linear equations formulated by Eq(1) through dynamic programming. However, many real-world graphs have cycles, and we have to use the following iterative form to compute the SimRank similarity:

$$S(a,b) \approx R_t(a,b) = \frac{C}{|I(a)||I(b)|} \sum_{i=1}^{|I(a)|} \sum_{j=1}^{|I(b)|} R_{t-1}(I_i(a), I_j(b))) \quad (2)$$

where $R_t(*, *)$ is the SimRank value on the t-th iteration.

Example 1. *Consider a tiny Twitter network G in Figure 1(a) with 5 nodes and 10 directed edges. We label the nodes from a to e for simplicity. In this graph, each node represents a user and the directed edges represent the relationship between users. For instance, the edge $\langle b, c\rangle$ denotes user b follows user c on Twitter. We set the decay factor $C = 0.36$ and the number of iterations $\xi = 3$. The SimRank similarities on the final iteration, i.e, $R_3(*, *)$, are shown in Figure 1(b). From the table we can see the SimRank matrix is symmetric and has value 1.0 on all its diagonal grids.*

[1]We randomly assign a number for each of its in-neighbors.



	a	b	c	d	e
a	1.000	0.115	0.086	0.115	**0.184**
b	0.115	1.000	0.149	**0.212**	0.019
c	0.086	0.149	1.000	0.149	0.030
d	0.115	**0.212**	0.149	1.000	0.019
e	**0.184**	0.019	0.030	0.019	1.000

(a) An example graph (b) Result of the 3rd iteration

Figure 1: A graph and its 3-iteration results

Convergence of SimRank. The iterative form of SimRank has fast convergence rate. A few iteration steps will be enough to yield desirable accuracy and fixing the number of iterations was applied in most of state-of-the-art works [5, 6,10,17,19] to quickly calculate approximate SimRank with little accuracy loss. Existing work [7] shows the ranking of SimRank stabilizes within 5 iterations. Therefore, performing limited number of iterations is acceptable for ranking queries such as top-k search [10] and `SRK-Join`.

2.2 Random Surfer Model

While Eq(1) serves as a fundamental tool for SimRank calculation, Jeh and Widom [7] offered another figurative model for computing SimRank values equivalently. Intuitively, SimRank measures how soon two random surfers are expected to meet at the same node if they start at node a and b respectively and randomly travel the backward edges. To formulate this model, we introduce a concept:

Definition 1 (Two-Way Path). *Given a graph $G(V, E)$, a node-pair sequence $\mathcal{TP} = \{(a_1, b_1) \to (a_2, b_2) \to \cdots \to (a_{\ell+1}, b_{\ell+1})\}$ is called a two-way path if it satisfies :*

$$\forall \quad i \in [1, \ell+1], a_i, b_i \in V,$$
$$\text{and} \quad \forall \quad i \in [1, \ell+1), \langle a_{i+1}, a_i\rangle, \langle b_{i+1}, b_i\rangle \in E.$$

We use $\ell(\mathcal{TP})$ to denote the length of a two-way path and $st(\mathcal{TP})$ to refer to the starting pair of it. The probability of this two-way path is defined as :

$$P(\mathcal{TP}) = \prod_{i=1}^{\ell} \frac{C}{|I(a_i)||I(b_i)|} \quad (3)$$

We call $\mathcal{TP}$ a **first-meeting** two-way path if $a_i \neq b_i$ for $1 \leq i \leq \ell$ and $a_{\ell+1} = b_{\ell+1}$. For example, consider a first-meeting two-way path in Figure 1(a). $\mathcal{TP} = \{(a, b) \to (c, e) \to (b, c) \to (e, e)\}$. It is clear that $st(\mathcal{TP}) = (a, b)$ and $\ell(\mathcal{TP}) = 3$. The probability of two random surfers starting from nodes a and b respectively and surfing on this two-way path can be calculated according to Eq(3) as:

$$\begin{aligned} P(\mathcal{TP}) &= \frac{C}{|I(a)||I(b)|} \cdot \frac{C}{|I(c)||I(e)|} \cdot \frac{C}{|I(b)||I(c)|} \\ &= \frac{0.36^3}{2 \times 2 \times 3 \times 1 \times 2 \times 3} = 0.000648. \end{aligned}$$

We can compute the SimRank based on the probability of two-way paths as formalized in Theorem 1.

THEOREM 1 (RANDOM SURFER MODEL).

$$R_t(a,b) = \sum_{\mathcal{TP}} P(\mathcal{TP}) \tag{4}$$

where $\mathcal{TP}$ is an arbitrary first-meeting two-way path with $st(\mathcal{TP}) = (a,b)$ and $\ell(\mathcal{TP}) \leq t$.

It requires complex statements to prove Theorem 1 and interested readers can refer to [7] for detailed proofs. The rationale of this model is that, the larger probabilities two random surfers meet with, the more similar their starting nodes are. Theorem 1 suggests by summarizing all first-meeting two-way paths which start at (a,b) and end within t steps, we can compute SimRank values on the t-th iteration.

2.3 Problem Definition

DEFINITION 2 (TOP-k SIMRANK-BASED SIMILARITY JOIN). *Given a graph $G(V,E)$ with n nodes and m edges and an integer k, the top-k SimRank-based similarity join problem (`SRK-Join`) aims to find a set of k node pairs $\mathcal{K}$ such that for $(a,b) \in \mathcal{K}$ and $(a',b') \in V * V - \mathcal{K}$, $S(a,b) \geq S(a',b')$.*

For example, consider the graph in Figure 1(a) and assume $k = 2$. `SRK-Join` returns two pairs - (b,d) and (a,e) - as the query answer. Their SimRank similarities on the 3-rd iteration are highlighted in Figure 1(b).

3. FROM SIMRANK TO DOT PRODUCT

A straightforward approach to calculate SimRank $S(a,b)$ enumerates all possible two-way paths within length ξ. However, this approach is rather expensive for `SRK-Join` queries, because the complexity of computing $S(a,b)$ is determined by the number of enumerated two-way paths which is $O(D^{2\xi})$ and the overall time complexity is $O(n^2 D^{2\xi})$. To address this issue, we introduce a partition-and-combine method which splits a two-way path into independent one-way paths and then merges them together. This method reduces the number of enumerated paths to $O(nD^{\xi})$. Next we transform the SimRank computation to the dot product calculation.

3.1 Partitioning the Two-Way Paths

Consider a first-meeting two-way path $\mathcal{TP} = \{(a_1,b_1) \to (a_2,b_2) \to \cdots \to (a_{\ell+1},b_{\ell+1})\}$. Two surfers a_1 and b_1 walk simultaneously and *first* meet at the same location $a_{\ell+1} = b_{\ell+1}$. We observe that the probability of this two-way path $P(\mathcal{TP})$ can be computed based on the probabilities of $a_1 \to a_2 \to \cdots \to a_{\ell+1}$ and $b_1 \to b_2 \to \cdots \to b_{\ell+1}$. Next we present the details of this method. For ease of presentation, we first introduce some concepts.

DEFINITION 3 (ONE-WAY PATH). *Given a graph $G(V,E)$, a node sequence $\mathcal{OP} = \{a_1 \to a_2 \to \cdots \to a_{\ell+1}\}$ is called a one-way path if it satisfies:*

$$\forall \quad i \in [1, \ell+1], a_i \in V,$$
$$\textit{and} \quad \forall \quad i \in [1, \ell+1), \langle a_{i+1}, a_i \rangle \in E.$$

Let $\ell(\mathcal{OP})$ denote the length of one-way path $\mathcal{OP}$. The probability of $\mathcal{OP}$ is defined as:

$$P(\mathcal{OP}) = \prod_{i=1}^{\ell} \frac{\sqrt{C}}{|I(a_i)|} \tag{5}$$

For example, $\{a \to e \to c\}$ is a one-way path on the graph in Figure 1(a). $P(\{a \to e \to c\}) = \frac{\sqrt{C} \cdot \sqrt{C}}{|I(a)||I(e)|} = \frac{0.36}{2 \times 1} = 0.18$.

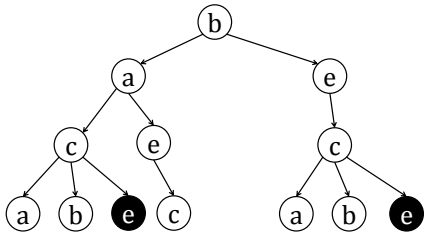


Figure 2: A path tree rooted at user b.

We can prove that the probability of a two-way path $P(\mathcal{TP})$ equals to the multiplication of the probabilities of two corresponding one-way paths.

LEMMA 1. *For any two-way path $\mathcal{TP} = \{(a_1,b_1) \to (a_2,b_2) \to \cdots \to (a_{\ell+1},b_{\ell+1})\}$, let $\mathcal{OP}_1 = \{a_1 \to a_2 \to \cdots \to a_{\ell+1}\}$ and $\mathcal{OP}_2 = \{b_1 \to b_2 \to \cdots \to b_{\ell+1}\}$. Then, we have*

$$P(\mathcal{TP}) = P(\mathcal{OP}_1) \cdot P(\mathcal{OP}_2) \tag{6}$$

If we only require to enumerate all one-way paths, the overall time complexity is significantly reduced to $O(nD^{\xi})$ from $O(n^2 D^{2\xi})$ (which enumerates all two-way paths). This fact motivates us to partition the random two-way paths into autonomous one-way paths and utilize the information of one-way paths to calculate the probabilities in Eq(4). To this end, we propose a new notation $Sim(*,*,*)$ to help gather the information of one-way paths.

DEFINITION 4 (SUMMATION OF ONE WAY PATHS). *Given a graph $G(V,E)$, $Sim(a \in V, x \in V, l \in [0,\xi])$ is the sum of the probabilities of all one-way paths with starting node a, destination x and length l.*

$Sim(*,*,*)$ captures the most essential information about the probabilities of one-way paths. On one hand, it eliminates redundant nodes in the path. For instance, for the graph in Figure 1(a), consider the two different one-way paths : $\{b \to a \to c \to e\}$ and $\{b \to e \to c \to e\}$. When we compute $Sim(*,*,*)$, the probabilities of them are both added into $Sim(b,e,3)$. On the other hand, as we will see in following discussions, the summation of one-way paths is a powerful tool to obtain the summation of two-way paths. For simplicity, we use "*summation of paths*" to refer to "*summation of the probabilities of paths*" if the context is clear.

We propose an efficient algorithm for calculating the summation of one-way paths based on the fact that the summation of one-way paths with longer length can be calculated through the combination of shorter ones. Thus, the Sim array can be calculated by dynamic programming and the transition equation is formulated in Eq(7):

$$Sim(a,x,l) = \sum_{y|x \in I(y)} \frac{\sqrt{C}}{|I(y)|} \cdot Sim(a,y,l-1) \tag{7}$$

We can utilize this property to compute the summation of all possible one-way paths. Given a node, we generate *a path tree* rooted at the node. The children of the root include all of its in-neighbors. We recursively add the in-neighbors of these children as their child nodes. The recursion terminates if the path tree reaches ξ levels. Obviously for each node in the path tree, the path which travels from the root to the node along the tree edges corresponds to a valid one-way path. For example, Figure 2 shows the path tree rooted at b with depth $\xi = 3$ for the example graph in Figure 1(a).

Then we utilize the path tree to compute the summation of one-way paths. Algorithm 1 illustrates the pseudo-code

Algorithm 1: Generate-Sim$(G(V,E), C, \xi)$

Input: $G(V,E)$: A directed graph;
C : A decay factor between 0 and 1
ξ : The maximum length of one/two-way paths
Output: Sim: Summation of one-way paths

1 $Sim \leftarrow$ an empty hash map;
2 **for** *each node* $a \in V$ **do**
  // $\mathcal{R}[i]$ is the set of the nodes with depth i in the path tree rooted at a
3 $\mathcal{R}[1 \dots \xi] \leftarrow \emptyset$; $\mathcal{R}[0] = \{a\}$;
4 $Sim(a,a,0) \leftarrow 1.0$;
5 **for** *each* $l \in [0, \xi - 1]$ **do**
6 **for** *each* $y \in \mathcal{R}[l]$ **do**
7 **for** *each* $x \in I(y)$ **do**
8 $Sim(a,x,l+1) += \frac{\sqrt{C}}{|I(y)|} \cdot Sim(a,y,l)$;
9 $\mathcal{R}[l+1] \leftarrow \mathcal{R}[l+1] \cup \{x\}$;
10 return Sim;

for calculating Sim values. For each node a (line 2), we process the nodes in the path tree from top to bottom. We denote the set of nodes with depth i as $\mathcal{R}[i]$. Initially, for each depth i greater than 0, $\mathcal{R}[i]=\emptyset$; for depth 0, $\mathcal{R}[0]=\{a\}$ (line 3). Then, the algorithm uses Eq(7) to calculate the Sim array with increasing order of l (lines 5-8). Line 9 updates the node set $\mathcal{R}[i]$ of current depth i.

Example 2. *Consider calculating $Sim(b,*,*)$ on the graph in Figure 1(a) by using the path tree in Figure 2. For $l=0$, $Sim(b,b,0)=1.0$ and $\mathcal{R}[0]=\{b\}$. When $l=1$, $Sim(b,a,1) = Sim(b,e,1) = \frac{\sqrt{0.36}}{2} = 0.3$ and $\mathcal{R}[1]=\{a,e\}$. Then,*

$$\begin{aligned} Sim(b,c,2) &= \frac{\sqrt{C}}{|I(a)|} \cdot Sim(b,a,1) + \frac{\sqrt{C}}{|I(e)|} \cdot Sim(b,e,1) \\ &= 0.3 \times 0.3 + 0.6 \times 0.3 = 0.27 \end{aligned}$$

The calculation of the nodes in depth 3 is similar. Note that $\mathcal{R}[3]$ only has 4 elements while there are 7 one-way paths with length 3. This is the effect of combining redundant paths by summarizing one-way paths.

3.2 Combining the One-Way Paths

Algorithm 1 can efficiently calculate the Sim values. Now, we propose Theorem 2 to integrate the one-way paths to compute the summation of two-way paths.

Theorem 2.

$$\sum_{\mathcal{TP}} P(\mathcal{TP}) = Sim(a,x,l) \cdot Sim(b,x,l) \tag{8}$$

where $\mathcal{TP}$ is an arbitrary two-way path with $st(\mathcal{TP}) = (a,b)$ and meets at node x in exactly l steps.

Proof. Suppose $Sim(a,x,l)$ consists of the probabilities of N_a one-way paths, $\mathcal{OP}_{a,1}, \mathcal{OP}_{a,2}, \cdots, \mathcal{OP}_{a,N_a}$, $Sim(b,x,l)$ consists of the probabilities of N_b one-way paths, $\mathcal{OP}_{b,1}, \mathcal{OP}_{b,2}, \cdots, \mathcal{OP}_{b,N_b}$. Then, we have

$$\begin{aligned} & Sim(a,x,l) \cdot Sim(b,x,l) \qquad (9) \\ = & \sum_{i=1}^{N_a} P(\mathcal{OP}_{a,i}) \sum_{j=1}^{N_b} P(\mathcal{OP}_{b,j}) = \sum_{i=1}^{N_a} \sum_{j=1}^{N_b} P(\mathcal{OP}_{a,i}) P(\mathcal{OP}_{b,j}) \end{aligned}$$

Note that $\mathcal{OP}_{a,i}$ and $\mathcal{OP}_{b,j}$ are both ending at x in exactly l steps. So if we denote $\mathcal{TP}_{i,j}$ as the two-way path with length l constituted by $\mathcal{OP}_{a,i}$ and $\mathcal{OP}_{b,j}$, according to Lemma 1, Eq(10) can be transformed to

$$Sim(a,x,l) \cdot Sim(b,x,l) = \sum_{i=1}^{N_a} \sum_{j=1}^{N_b} P(\mathcal{TP}_{i,j}) \tag{10}$$

Note that $\forall i,j, st(\mathcal{TP}_{i,j}) = (a,b)$. So Eq(10) tells us that the multiplication of $Sim(a,x,l)$ and $Sim(b,x,l)$ is the summation of $N_a \cdot N_b$ two-way paths which start at (a,b) and meet at node x in precisely l steps.

Next, consider an arbitrary two-way path $\mathcal{TP}$ whose starting pair is (a,b), ending pair is (x,x) and length is l. Obviously $\mathcal{TP}$ is contained in the $N_a \cdot N_b$ two-paths in Eq(10). □

Theorem 2 suggests for two-way paths sharing the same starting node, meeting location and length, we calculate the summation of probabilities of them by first summing up the one-way paths and then multiplying the summation.

Nevertheless, recall Theorem 1, the random surfer model requires the summation of all **first-meeting** two-way paths. In Eq(8), it is obvious that there are multi-meeting paths. For example, let us consider the graph in Figure 1(a) and the multiplication, $Sim(a,e,3) \cdot Sim(b,e,3)$, which summarizes all two-way paths that start from (a,b) and meet at e in exactly 3 steps. We can see that the two-way path $\{(a,b) \to (e,a) \to (c,c) \to (e,e)\}$, which is a multi-meeting path because the two surfers already met at node c before they met at e, is contained in this summation. Accordingly, we must subtract these extra paths from Eq(8) in order to compute the real SimRank similarity. To this end, we introduce the notation of the second meeting probability.

Definition 5 (Second Meeting Probability). *The second meeting probability $\Delta(x,l)$ is the summation of probabilities of two-way paths that satisfy (1) starting from node x; (2) ending at the same node; (3) no other meeting node; and (4) length no longer than $\xi - l$.*

The Δ array actually describes the probability of two random surfers both start at node x and meet only once again within $\xi - l$ steps. To efficiently compute the probability, as these two-way paths all have a common starting node, namely, x, we can utilize the path tree rooted at node x to calculate $\Delta(x,l)$ for $l \in [0,\xi]$. For each one-way path $\mathcal{OP}$ in the path tree, it is simple to use the inclusion-exclusion principle to find the set of one-way paths which have the same length as $\mathcal{OP}$ and only share a common ending node with $\mathcal{OP}$(we do not take into account the starting nodes, because they are always the same). Then according to Lemma 1, by summarizing the multiplication of these one-way paths we can easily compute $\Delta(x,l)$.

For example, consider the path tree in Figure 2. When we calculate $\Delta(b,0)$, the probability of two random surfers starting from b and meeting again at b within $3-0=3$ steps, for the one-way path $\{b \to a \to c\}$, we find another one-way path $\{b \to e \to c\}$ with the same length 2 which only shares a common ending node with it, so we add to $\Delta(b,0)$ the multiplication of these two one-way paths, i.e, $P(\{b \to a \to c\}) \cdot P(\{b \to e \to c\})$. We will also add to $\Delta(b,0)$ the multiplication of $\{b \to a \to c\}$ and itself, i.e, $P^2(\{b \to a \to c\})$ in that according to the inclusion-exclusion principle, we must add to $\Delta(b,0)$ the multiplications one-way paths which share **at least** one common node and obviously the

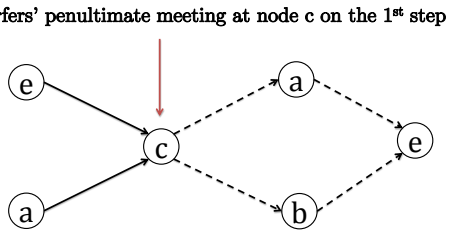


Figure 3: An illustration of Lemma 2

two identical paths meet this requirement. Next, when we compute the multiplication of one-way paths which share at least two common nodes, we will subtract $P^2(\{b \to a \to c\})$ from $\Delta(b, 0)$. Thus, by using the inclusion-exclusion principle, for $\{b \to a \to c\}$, we can correctly find the set of one-way paths that have the same length with it and only share a common ending node with it. Similarly, we can summarize every one-way path in the path tree.

LEMMA 2.

$$\sum_{x \in V} \sum_{l=0}^{\xi} Sim(a, x, l) \cdot Sim(b, x, l) \cdot \Delta(x, l) \tag{11}$$

is the summation of all ***multi-meeting*** *two-way paths that start at (a, b) and end at the same node within ξ steps.*

PROOF. According to Theorem 2, $Sim(a, x, l) \cdot Sim(b, x, l)$ is the summation of a set of two-way paths that share the same starting nodes (a, b), the same meeting node x and the same length l. A thoughtful observation reveals that, when it is multiplied by $\Delta(x, l)$, the summation of another set of two-way paths which have starting nodes (x, x) and end up meeting at another node within $\xi - l$ steps, it becomes the summation of all multi-meeting two-way paths shorter than or equal to ξ steps which have their **penultimate** meeting on the l-th step at node x. And by summarizing different penultimate meeting nodes and steps, we can prove that Eq(11) equals to the summation of all multi-meeting two-way paths that start at (a, b) and meet within ξ steps. □

Figure 3 shows a multi-meeting two-way path on the graph in Figure 1(a) which illustrates this process. The solid lines represent a two-way path which was added into $Sim(e, c, 1) \cdot Sim(a, c, 1)$. The slashed lines represent a two-way path which was added into $\Delta(c, 1)$. When the two two-way paths are combined, it becomes a 2-meeting two-way path, the probability of which is contained in $Sim(e, c, 1) \cdot Sim(a, c, 1) \cdot \Delta(c, 1)$. Lemma 2 exploits a crucial property of a multi-meeting two-way path – there is always a penultimate meeting situation, i.e, the meeting node and meeting step.

Next, we propose the transformation from SimRank to dot product in Theorem 3.

THEOREM 3.

$$S(a, b) \approx R_\xi(a, b) = \sum_{x \in V} \sum_{l=0}^{\xi} Sim'(a, x, l) \cdot Sim'(b, x, l) \tag{12}$$

where $Sim'(a, x, l) = Sim(a, x, l) \cdot \sqrt{1 - \Delta(x, l)}$.

PROOF HINT. This Theorem can be easily proved based on Theorem 1 and Lemma 2. □

To facilitate computation, we model $Sim'(a, *, *)$ as a vector with $n(\xi+1)$ dimensions. We first number the nodes from 0 to $n-1$ and let $|x|$ denote the order of node x. The dimension with respect to $Sim'(*, x, l)$ is $(|x| \cdot (\xi + 1) + l)$. Then

Node	Non-zero dimensions and values										
a	0	2	3	6	7	9	10	11	17	18	19
	0.90	0.05	0.05	0.05	0.04	0.28	0.17	0.05	0.24	0.05	0.07
b	1	3	4	7	10	11	17	18	19		
	0.27	0.05	0.89	0.05	0.25	0.05	0.24	0.07	0.05		
c	1	2	3	5	7	8	10	11	17	18	19
	0.18	0.05	0.04	0.18	0.04	0.93	0.17	0.09	0.16	0.10	0.05
d	1	3	7	**10**	11	12	17	18	19		
	0.27	0.05	0.05	**0.25**	0.05	0.89	0.24	0.07	0.05		
e	2	3	6	9	11	16	18	19			
	0.11	0.04	0.11	0.56	0.11	0.80	0.10	0.07			

Figure 4: The vector matrix of Figure 1(a)

the calculation of SimRank similarity is simply equivalent to the calculation of dot product between two vectors. For node x, let $\vec{x}$ denote its corresponding vector. In the following, we will **use node x and vector $\vec{x}$ interchangeably** when the context is clear.

EXAMPLE 3. *Figure 4 is the vector matrix of the graph in Figure 1(a) with $C = 0.36$ and $\xi = 3$. A gray grid represents a dimension and the white grid below is the corresponding value of this dimension rounded to two decimals. In the example graph, there are $5 \times (3+1) = 20$ dimensions in total, numbered from 0 to 19. For instance, if the numeric orders of a, b, c, d and e are $0, 1, 2, 3$ and 4 respectively, then the dimension with respect to $Sim'(*, c, 2)$ is the $2 \times (3+1)+2 = 10$-th dimension. From the highlighted grids in the vector matrix, we can see $Sim'(d, c, 2) = 0.25$.*

Consider calculating the dot product between vector d and vector e. $\vec{d} \cdot \vec{e} = 0.04 \times 0.05 + 0.05 \times 0.11 + 0.07 \times 0.10 + 0.05 \times 0.07 = 0.019$ which is the same as the 3-rd iteration results shown in Figure 1(b).

Complexity Analysis. The time complexity of calculating $Sim(*, *, *)$ is $O(nD^\xi)$ because the scale of one-way paths is $O(nD^\xi)$. When we compute Δ array, for each one-way path in all n path trees, we perform a hash-based inclusion-exclusion operation, so there are approximately $2^\xi nD^\xi$ hash operations. Existing works [7] have shown $\xi = 5$ is enough for stabilizing the relative ranking of SimRank, so in practice 2^ξ can be regarded as a constant with $O(1)$ time complexity. Thus the time complexity of calculating Δ array is $O(nD^\xi)$ and the overall time complexity of the transformation from SimRank to dot product is $O(nD^\xi)$.

The space complexity is $O(nD^\xi)$ which is determined by the number of non-zero elements in Sim' array. Note that the vector matrix is theoretically very sparse and we can use adjacent lists like Figure 4 to store the whole matrix.

4. TWO-STEP JOIN FRAMEWORK

In this section, we first propose a two-step join framework to address the dot product problem in Section 4.1 and then devise a novel technique in Section 4.2 to generate a set of candidate nodes such that the top-k similar pairs can be composed by the candidate nodes. We propose an efficient algorithm to compute the top-k similar pairs using the candidate nodes in Section 4.3.

4.1 Join Framework

Basic Idea. Consider a common situation where a user requires the system to return the 1000 most similar pairs on a graph with 1M nodes. The answer pairs will contain at most 2000 distinct nodes. That is to say, there are almost $\frac{1M-2000}{1M} = 99.8\%$ nodes that are not included in any answer

pairs. So we are motivated to devise a method which can efficiently eliminate unpromising nodes.

Formally, given a vector $\vec{x}$(which corresponds to one of the n nodes in a graph), we use $\vec{S_x}$ to denote the vector that has the largest dot product value with $\vec{x}$ among other $n-1$ vectors. We extract $2k$ pairs with the largest dot product values from the pair set $\{(\vec{x}, \vec{S_x}) \mid x \in V\}$ and denote them as $(\vec{x_1}, \vec{y_1}), (\vec{x_2}, \vec{y_2}), \ldots, (\vec{x_{2k}}, \vec{y_{2k}})$. We prove that nodes from $\mathcal{S} = \{\vec{x_1}, \vec{y_1}, \vec{x_2}, \vec{y_2}, \ldots, \vec{x_{2k}}, \vec{y_{2k}}\}$ are enough to constitute an answer set $\mathcal{K}$ of the `SRK-Join` query.

THEOREM 4. *Nodes from the set $\mathcal{S}$ are enough to constitute an answer pair set $\mathcal{K}$ of the* ***`SRK-Join`*** *query.*

PROOF. Consider an arbitrary answer pair set $\mathcal{K}'$. If $\mathcal{K}'$ contains a node pair (x, y) such that $x \notin \mathcal{S}$, then we have:

$$\vec{x} \cdot \vec{y} \leq \vec{x} \cdot \vec{S_x} \leq \vec{x_1} \cdot \vec{y_1}, \vec{x_2} \cdot \vec{y_2}, \cdots, \vec{x_{2k}} \cdot \vec{y_{2k}} \tag{13}$$

Assume (x_1, y_1) is not in $\mathcal{K}'$, then if we replace (x, y) with (x_1, y_1), $\mathcal{K}'$ will still be a qualified answer pair set according to Definition 2. If there still exists a pair (x', y') such that $x' \notin \mathcal{S}$, we replace it with another pair that is not in $\mathcal{K}'$ and is from the set $\{(\vec{x}, \vec{S_x}) \mid x \in V\}$. Note that there are at least k distinct pairs of vectors in $(\vec{x_1}, \vec{y_1}), (\vec{x_2}, \vec{y_2}), \cdots, (\vec{x_{2k}}, \vec{y_{2k}})$ and there are exactly k pairs in $\mathcal{K}'$, so after less than or equal to k replacements, $\mathcal{K}'$ will consist nodes only from $\mathcal{S}$. □

Framework. We can improve the idea and only find the top pairs with exactly $2k$ distinct nodes. Thus we can reduce the candidate node size to $2k$ which is not large because k in practice is usually small. Based on this idea, we propose a two-step framework. In the first step, we generate a candidate node set with $2k$ nodes. In the second step, we identify the top-k similar pairs based on the candidate nodes.

4.2 Phase 1: Candidate Generation

4.2.1 Algorithm Overview

Algorithm 2 illustrates how to generate the candidate nodes. We maintain a heap $\mathcal{H}$ which contains at most $2k$ elements (line 1). Each element has the form $(pair, value)$. The elements in the heap represent the pairs with the largest dot product values. For each vector $\vec{x}$ (line 2), we use function $find(\vec{x}, \theta)$ to check whether $\vec{x} \cdot \vec{S_x}$ is larger than or equal to the smallest value θ in $\mathcal{H}$. If so (line 5), we replace the smallest value in $\mathcal{H}$ with $\vec{x} \cdot \vec{S_x}$(Line 8). At the end of the algorithm, nodes in $\mathcal{H}$ are added to the candidate node set. In the algorithm, the $find(\vec{x}, \theta)$ function takes an important role to achieve high performance. A naive algorithm to implement the function enumerates all vectors, finds the most similar one $\vec{S_x}$, and checks whether $\vec{x} \cdot \vec{S_x} \geq \theta$. But this method is rather expensive. To address this issue, we propose an inverted-list-based method, which can significantly improve the performance.

4.2.2 Inverted-List-Based Early Termination

We build inverted lists on top of the vectors. In the dot product model, the entry of the inverted index is a dimension and the inverted list of each dimension is a list of elements with the form $\langle vector\_id, value \rangle$ which represent vectors that have non-zero values in this dimension, associating with the corresponding values. For example, consider the 9-th dimension in Figure 4, its inverted list consists of 2 elements : $\langle a, 0.28 \rangle$ and $\langle e, 0.56 \rangle$.

Using the inverted lists, we can quickly find vectors which have a non-zero dot product value with a given vector $\vec{x}$. A naive idea to compute $\vec{S_x}$ is to union the inverted lists of all its non-zero dimensions for each $\vec{x}$, and aggregate the dot product values at the same time. However, this approach becomes increasingly inefficient with increasing number of vectors. To this end, we propose an early-termination strategy which skips unnecessary inverted lists and only uses a small portion of lists to find $\vec{S_x}$.

Algorithm 2: Generate-Candidate($\mathcal{X}$, k)

Input: $\mathcal{X}$: A set of n vectors
k : The `SRK-Join` query integer
Output: $\mathcal{S}$: The candidate node set

```
1  H ← an empty small heap;
2  for each x ∈ X do
3      θ←H has 2k distinct nodes?0 : smallest value in H;
4      (exist, S_x, val) ← find(x, θ);
5      if exist then
6          if H has 2k distinct nodes then
7              pop the element with the smallest val;
8          insert ((x, S_x), val) into H;
9  S ← all nodes in H;
10 return S;
```

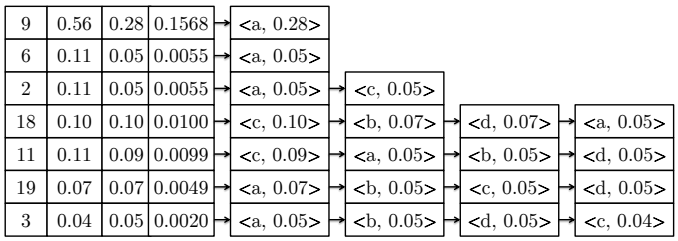


Figure 5: An example of inverted lists

The basic intuition is to maintain a variable `currentMax`, which stores the maximum dot product value currently, and an upper bound for each dimension ub, which marks the maximum possible value a dot product can get from this dimension. We can use the bounds to do early termination. The pseudo-code is shown in Algorithm 3. We first pick out the non-zero dimensions into a list (line 1), then initialize the upper bounds (line 3) and the references of the inverted lists (line 4). The notation $\mathcal{V}_{\vec{x},\mathcal{D}_i}$ represents the value of the $\mathcal{D}_i$-th dimension of $\vec{x}$. Then Algorithm 3 uses a hash map `aggr` (line 7) to store the dot product values of the current stage. We aggregate the inverted lists one by one (line 8). For each element in the inverted list (line 9), we aggregate it onto corresponding `aggr` entry (line 10) and update `currentMax` (line 11). Line 13 embodies an effective early termination idea. If the summation of upper bounds(ub) of all dimensions which have not been aggregated is already smaller than `currentMax` or θ, we can conclude that vectors that we have not seen yet will not be the expected answer. To terminate as early as possible, we sort inverted lists in non-increasing order by $\frac{ub_i}{|\mathcal{I}_i|}$ (line 5). The reason is that, (1) the fewer elements an inverted list contains, the earlier this list should be aggregated. Then early termination will avoid aggregating long lists; (2) the larger the upper bound of an inverted list is, the earlier this list should be aggregated as this would make `currentMax` become large in early stages and at the same time make the remaining summation of upper bounds become small. Thus it will make the pruning more powerful.

Next we find all vectors with positive `aggr` values (line 15)

Algorithm 3: $find(\vec{x}, \theta)$

Input: $\vec{x}$: One of the n vectors; θ : A threshold
Output: Whether $\vec{x} \cdot \vec{S_x} \geq \theta$. If yes, return $\vec{S_x}$, $\vec{x} \cdot \vec{S_x}$

1 $\mathcal{D} \leftarrow$ the list of non-zero dimensions of $\vec{x}$;
2 **for** *each* $i \in [1, |\mathcal{D}|]$ **do**
3 $\quad ub_i \leftarrow \mathcal{V}_{\vec{x},\mathcal{D}_i} \cdot \max_{\vec{y} \in \mathcal{X}} \mathcal{V}_{\vec{y},\mathcal{D}_i}$;
4 $\quad \mathcal{I}_i \leftarrow$ the inverted list of the $\mathcal{D}_i$-th dimension;
5 sort the dimensions in $\mathcal{D}$ in non-increasing order of $\frac{ub_i}{|\mathcal{I}_i|}$;
6 `currentMax` $\leftarrow 0$;
7 `aggr` $\leftarrow$ an empty hash map from vector to real value;
8 **for** *each* $i \in [1, |\mathcal{D}|]$ **do**
9 $\quad$ **for** *each element* $\langle \vec{y}, \texttt{val} \rangle \in \mathcal{I}_i$ **do**
10 $\quad\quad$ `aggr`$(\vec{y}) \leftarrow$ `aggr`$(\vec{y}) +$ `val` $\cdot \mathcal{V}_{\vec{x},\mathcal{D}_i}$;
11 $\quad\quad$ **if** `aggr`$(\vec{y}) >$ `currentMax` **then**
12 $\quad\quad\quad$ `currentMax` $\leftarrow$ `aggr`$(\vec{y})$; $\vec{S_x} \leftarrow \vec{y}$;
13 $\quad$ **if** $\sum_{i+1}^{|\mathcal{D}|} ub_i <$ max(`currentMax`, θ) **then**
14 $\quad\quad$ `stopDimension` $\leftarrow i$; break;
15 $\mathcal{F} \leftarrow$ the list of vectors which have `aggr` > 0;
16 sort the vectors in $\mathcal{F}$ in non-increasing order of `aggr`;
17 **for** *each* $\vec{y} \in \mathcal{F}$ **do**
18 $\quad$ **for** each $i \in$ [`stopDimension` $+ 1, |\mathcal{D}|$] **do**
19 $\quad\quad$ `aggr`$(\vec{y}) \leftarrow$ `aggr`$(\vec{y}) + \mathcal{V}_{\vec{y},\mathcal{D}_i} \cdot \mathcal{V}_{\vec{x},\mathcal{D}_i}$;
20 $\quad\quad$ **if** $\sum_{i+1}^{|\mathcal{D}|} ub_i +$`aggr`$(\vec{y}) <$max(`currentMax`, θ) **then**
21 $\quad\quad\quad$ break;
22 $\quad$ **if** `aggr`$(\vec{y}) >$ `currentMax` **then**
23 $\quad\quad$ `currentMax` $\leftarrow$ `aggr`$(\vec{y})$; $\vec{S_x} \leftarrow \vec{y}$;
24 **if** `currentMax` $\geq \theta$ **then** return (Yes, $\vec{S_x}$, `currentMax`); **else** return (No, `null`, `null`);

which may become S_x and complete the aggregating process of these vectors (line 19). Line 20 is also a pruning skill: if the current `aggr` value plus the maximum possible dot product value of the remaining dimensions is still less than `currentMax` or θ, we do not need to aggregate this vector any more. To make this pruning more effective, before aggregating, we first sort the vectors by their prior `aggr` values (line 16). Similar to previous techniques, the reason is that this will make `currentMax` become large quickly and cut off a lot of unnecessary aggregate operations.

Example 4. *Consider the vector matrix in Figure 4 and* $find(\vec{e}, 0.1)$ *to check whether* $\vec{e} \cdot \vec{S_e} \geq 0.1$. *Figure 5 shows the inverted lists of non-zero dimensions of vector e. The lists are sorted according to Line 5. After we aggregate the inverted list of the 9-th dimension, we have* $\texttt{aggr}(\vec{a}) = 0.56 \times 0.28 = 0.1568$. *The summation of the upper bounds of the remaining 6 dimensions is* 0.0378 *which is already smaller than* $\texttt{aggr}(\vec{a})$. *We conclude that* $\vec{S_e} = \vec{a}$. *In this example, we get the answer by only aggregating one short list.*

4.3 Phase 2: Tree-Based Wand Algorithm

After generating the candidate nodes, we can enumerate all candidate node pairs to compute top-k similar pairs with $O(k^2 D^\xi)$ time complexity which is already affordable for small k. Nonetheless, for social network analysis where k can be up to 1,000 or 10,000, this method is not efficient. Here, we propose a fast method to efficiently retrieve the answer pairs from the candidate nodes. Our algorithm is inspired from the `WAND` algorithm [2] which was originally proposed for efficient top-k document retrieval where queries are short, e.g, less than 30. However, its overhead becomes evident for long queries, i.e, vectors with many non-zero elements. We present a tree-based `WAND` algorithm which can handle long queries to find top-k pairs based on the candidate nodes. The tree-based `WAND` algorithm is characterized by a function $treewand(\theta, \vec{x})$ which returns a vector whose dot product value with the vector $\vec{x}$ is probably larger than θ. To identify the join answer, we maintain a heap similar to Algorithm 2 which stores the most similar pairs and uses the $treewand$ function incrementally for each of the $2k$ vectors.

Algorithm 4: TreeWand-locatePivot(`root`, θ)

Input: `root`: The root node of the `Bst`
θ : The threshold of $treewand$ function
Output: The pivot dimension

1 `curNode` $\leftarrow$ `root`; `pivotDimension` $\leftarrow$ `null`;
2 **while** `curNode` is not empty **do**
3 $\quad$ **if** `curNode`.$leftSum$ + `curNode`.$ub > \theta$ **then**
4 $\quad\quad$ `pivotDimension` $\leftarrow$ `curNode`.$dimension$;;
5 $\quad\quad$ `curNode` $\leftarrow$ `curNode`.$leftChild$;
6 $\quad$ **else**
7 $\quad\quad$ $\theta \leftarrow \theta -$ `curNode`.$leftSum -$ `curNode`.ub;
8 $\quad\quad$ `curNode` $\leftarrow$ `curNode`.$rightChild$;
9 return `pivotDimension`;

To implement the $treewand$ function, all the elements in each list are first sorted in alphabetical order and each list maintains a top element which is initially pointed to the first element in the sorted list. Then we sort all lists in alphabetical order of their top elements. Next, we find a pivot dimension, whose prefix summation of the upper bounds, ub, is larger than θ. After the pivot dimension is identified, the elements before this dimension (in sorted order) whose alphabetical values are smaller than the top element of the pivot dimension will not have dot product larger than θ. Thus we skip these elements and repeat the above process until the top elements of the dimensions before the pivot dimension are the same as the pivot element. We combine the process of sorting the lists and locating the pivot dimension together by a balanced binary search tree (`Bst`) with alphabetical value as the key. Each node in the `Bst` represents a top element of a dimension. The operation of skipping inverted lists and getting a new top element can be done by first deleting an element and then inserting a new element. Thus the inefficient sort process can be omitted.

This `Bst` can also be used to locate the pivot dimension. The pseudo-code of locating the pivot is presented in Algorithm 4. The variable `curNode` has various fields. $leftChild$ and $rightChild$ denote the pointer of the left child and right child of this node respectively. $leftSum$ is maintained throughout the whole query procedure and represents the summation of all upper bounds in the left subtree. Algorithm 4 traverses the `Bst` from top to bottom. For each node, we first check whether the $leftSum$ plus ub of this top element is larger than θ. If so, we are sure that the pivot dimension is in the left subtree, then we record this dimension as a temporary answer and go left. Otherwise, we subtract $leftSum + ub$ from θ and go to the right subtree recursively. Once we reach a leaf node, we terminate the process and return the answer. Example 5 shows the procedure of the first repetition of $treewand(0.15, \vec{a})$ where

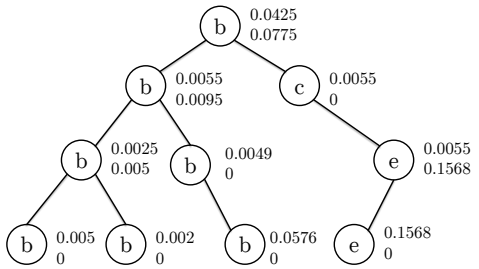


Figure 6: An example of `Bst`

$\vec{a}$ is the query vector in Figure 4.

EXAMPLE 5. *Figure 6 shows a `Bst` with 10 nodes. Each node represents a top element of a dimension. The upper number beside each node is ub and the lower number is $leftSum$. We start from the root. As $leftSum + ub = 0.0775 + 0.0425 < 0.15$, we set $\theta = 0.15 - 0.0775 - 0.0425 = 0.03$ and go right. At the node labeled with c, $0.0055 + 0 < 0.03$, we set $\theta = 0.03 - 0.0055 = 0.0245$ and go right again. At next node we find the summation of $leftSum$ and ub is larger than θ, so we record the this dimension as a temporary answer and go left. At the leaf node labeled with e, the summation of $leftSum$ and ub is still larger than θ, hence we set the final pivot dimension to the dimension of this node and terminate the process. Here we get the answer by accessing only 4 nodes. The linear search requires 9 accesses.*

Given one of the $2k$ vectors, the difference between tree-based `WAND` and the original `WAND` is that `WAND` sorts the dimensions in $O(d \log d)$ time and finds the pivot dimension using a linear search in $O(d)$ time where d is the number of non-zero dimensions of the vector. Thus `WAND` fails to handle long queries. Instead, by the use of a balanced `Bst`, we reduce the time complexity of both sorting the lists and locating the pivot dimension to $O(\log d)$. The top-k search `WAND` algorithm is significantly improved to support long queries.

5. APPROXIMATION ALGORITHM FOR SCALE-FREE GRAPHS

Recall that the scale of one-way paths with regard to a starting point is $O(D^{\xi})$ which is theoretically affordable for most graphs. However, many real-world graphs are scale-free [3] which means there is a small portion of high-degree nodes. These nodes pose a significant challenge to the SimRank-based similarity join problem, because if the path tree generated by Algorithm 1 contains such nodes, it will spread a lot of branches and reduce the efficiency. On the other hand, the high-degree nodes also bring opportunities. We find that high-degree nodes will cause the probability of a one-way path to be negligible, and thus the probabilities of many one-way paths have tiny difference. If users can tolerate the tiny difference between different pairs, we can devise more efficient approximation algorithms.

For example, consider a one-way path of length 2: $\{a_1 \rightarrow a_2 \rightarrow a_3\}$ where $|I(a_1)| = |I(a_2)| = |I(a_3)| = 100$. If $C = 0.5$, then the probability of this path is 5×10^{-5}. If we still extend the path tree from a_3 as Algorithm 1 does, it will generate another 100 one-way paths, each with the probability less than 5×10^{-7}. Note that each of them is only a one-way path, and if we use them to compute the SimRank values with other paths, they should be multiplied by another one-way path. Thus the maximum contribution of these one-way paths to the final SimRank value is $5 \times 10^{-7} \times (\sqrt{C})^3$, which is rather small and can be ignored.

Algorithm 5: Generate-Sim($G(V,E), C, \xi$)

```
// Insert the following statements before
   Line 7 of Algorithm 1.
```

1 **if** $\sum_{d=1}^{\xi-l} \mathcal{N}(y,d) \cdot \frac{Sim(a,y,l)}{|I(y)|} \cdot C^{\frac{2d+l}{2}} \leq \delta$ **then**
2 $\quad \delta \leftarrow \delta - \sum_{d=1}^{\xi-l} \mathcal{N}(y,d) \cdot Sim(a,y,l) \cdot C^{\frac{2d+l}{2}}$;
3 $\quad$ continue;

In this section, we present an approximation algorithm which can efficiently identify the top-k similar pairs under user-specified accuracy requirement. We modify Algorithm 1 to prune unnecessary one-way paths while still meeting a user-specified accuracy requirement δ.

User-Specified Accuracy Requirement. As we mentioned before, the ranking of SimRank values stabilizes within 5 iterations [7]. So we set $\xi = 5$ and use $R_5(*,*)$ as the ground truth for `SRK-Join` query. If we denote the SimRank values of the top-k results generated from $R_5(*,*)$ as $\psi(1), \ldots, \psi(k)$, sorted by SimRank value in non-increasing order, and those of the approximate results as $\Psi(1), \ldots, \Psi(k)$, sorted by SimRank value in non-increasing order, then the accuracy loss is formulated as $\max_{i=1}^{k} |\psi(i) - \Psi(i)|$. The accuracy loss describes the maximum absolute difference between the similarity values of answer pairs returned from the approximation algorithm and the real result $R_5(*,*)$.

The user requirement has a parameter δ which requires the accuracy loss is no more than δ:

$$\max_{i=1}^{k} |\psi(i) - \Psi(i)| \leq \delta \tag{14}$$

Upper Bound of Accuracy Loss. In Algorithm 1, before we aggregate the Sim value of the current node in the path tree to its children, we first check whether the accuracy loss of pruning the one-way path represented by this node would be greater than δ. To this end, we introduce a new notation $\mathcal{N}(x,d)$ which refers to the number of nodes with depth d in the path tree rooted at x. This array can be calculated recursively by the following equation:

$$\begin{aligned} \mathcal{N}(x,1) &= |I(x)|, \forall x \in V \\ \mathcal{N}(x,d) &= \sum_{y \in I(x)} \mathcal{N}(y,d-1), \forall x \in V \text{ and } d > 1 \end{aligned}$$

Then we propose Theorem 5 to predict the upper bound of accuracy loss of pruning a one-way path.

THEOREM 5. *If we prune the one-way path represented by $Sim(a,y,l)$, i.e, we do not aggregate its value any more, the accuracy loss is bounded by :*

$$\sum_{d=1}^{\xi-l} \mathcal{N}(y,d) \cdot \frac{Sim(a,y,l)}{|I(y)|} \cdot C^{\frac{2d+l}{2}} \tag{15}$$

PROOF. For each $d \in [1, \xi - l]$, there are $\mathcal{N}(y,d)$ one-way paths which were pruned. $\frac{Sim(a,y,l)}{|I(y)|} \cdot C^{\frac{d}{2}}$ is the upper bound of the probabilities of these one-way paths. Then according to Eq(12), to get the maximum contribution of these one-way paths to the real SimRank value, each of them should be multiplied by another one-way path, the maximum probability of which is $C^{\frac{d+l}{2}}$. Thus, the upper bound of accuracy loss of pruning one-way paths on depth d is $\mathcal{N}(y,d) \cdot Sim(a,y,l) \cdot C^{\frac{2d+l}{2}}$. By summarizing d from 1 to $\xi - l$, we can prove Eq(15). $\square$

Pruning Algorithm. Algorithm 5 illustrates the basic idea of our approximation algorithm by modifying Algorithm 1. In Line 1, we use Theorem 5 to judge whether the accuracy loss caused by pruning current one-way path represented by node y exceeds δ. If it does not exceed δ, we prune this one-way path(Line 3) and subtract the upper bound of the accuracy loss from δ(Line 2).

Accuracy Analysis. Algorithm 5 prunes a one-way path as long as the upper bound of the accuracy loss is within δ. This will decrease the SimRank values no more than δ. Similarly, we can design an algorithm for pruning one-way path for Δ array which will increase the SimRank values no more than δ. So the absolute difference between the SimRank values after pruning and $R_5(*,*)$ is no more than δ. Then it is evident that Eq(14) is satisfied.

Efficiency Analysis. The time complexity of calculating $\mathcal{N}$ array is obviously $O(n\xi)$. In Algorithm 5, the time complexity of the pruning operation(Line 1) is $O(1)$ as we can pre-calculate the summation $\sum_{d=1}^{\xi-l}\mathcal{N}(y,d)$ and the exponentiation $C^{\frac{2d+l}{2}}$, thus the time complexity of Algorithm 5 is $O(nD^{\xi})$ but it avoids calculating a significant number of unnecessary one-way paths caused by high-degree nodes.

6. EXPERIMENTAL STUDY

We have implemented our method and conducted extensive experiments, using both real and synthetic datasets, to evaluate our method. We have also compared our method with two baselines extended from state-of-the-art works.

Real Datasets. *Epinion*[2] is who-trust-whom online social network of a general consumer review site `Epinions.com`. It has 76K nodes and 509K directed edges. *Berkstan*[3] is a dense Berkeley-Stanford web page graph with 685K pages and 7.6M edges. The edges represent hyperlinks between web pages. *Youtube*[4] is a social network of 1.14M Youtube users and 4.9M connections. All these three real datasets are scale-free graphs. Table 1 shows the details of the datasets.

Synthetic Datasets. We generate two different synthetic datasets. One is ED (Evenly Distributed) which is controlled by three parameters, the number of nodes in a graph, the minimum in-degree I_{min} and the maximum in-degree I_{max}. The default values for I_{min} and I_{max} are 2 and 5 respectively. ED generates a graph with evenly distributed in-degrees. Another is SCALE-FREE [5] which generates scale-free graphs. The in-degrees in SCALE-FREE satisfy the power-law distribution. There are two parameters in SCALE-FREE, the number of nodes and α which controls the distribution of in-degrees. The default value of α is 3.

Parameter Choosing. In our experiments, we set the decay factor $C = 0.3$ which is an effective value used in [20]. The maximum path step ξ is set to 5 which enables the stabilization of rankings of SimRank values [7]. The user-specified accuracy tolerance δ is 10^{-3} if it is not specified.

Experiment Setup. All algorithms were implemented in C++ and compiled using GCC 4.8.1 with -O3 flag. The experiments were conducted on a Ubuntu server with two Intel Xeon X5670 CPUs (2.93GHz) and 64GB RAM. Each experiment was run 5 times and average time are reported.

[2] http://snap.stanford.edu/data/soc-Epinions1.html
[3] http://snap.stanford.edu/data/web-BerkStan.html
[4] http://konect.uni-koblenz.de/networks/youtube-links
[5] http://fabien.viger.free.fr/liafa/generation/

Table 1: Real Datasets.

Dataset	$\|V\|$	$\|E\|$	$D = \frac{\|E\|}{\|V\|}$	$\max_{x\in V}\|I(x)\|$
Epinion	75,879	508,837	6.71	3,622
Berkstan	685,230	7,600,595	11.10	84,290
Youtube	1,138,499	4,942,297	4.34	54,051

6.1 Evaluating Tree-Based Wand Algorithm

In this section, we will first evaluate the second phase of the join framework which identifies the `SRK-Join` answers among all the candidate nodes generated in the first phase. We compared our tree-based wand algorithm (denoted as treewand) with the original `WAND` algorithm and the naive algorithm (naivek2). We used Algorithm 5 to generate the vector matrix and used Algorithm 3 and Algorithm 2 to generate the candidate node set. Figure 7 depicts the results.

We can see for small k, e.g., $k = 100$, naivek2 had high performance and even was better than `WAND`. This is because its time complexity was $O(k^2D^{\xi})$ which was very efficient for small k values. However for large k, e.g., $k = 3000$, naivek2 had much worse performance than the other two methods. Our treewand method achieved the best performance over all three datasets, because treewand combined the sorting dimension and searching pivot process to $O(\log d)$ while the original `WAND` algorithm needed to iterate the pivot finding process a lot of times and each iteration took $O(d \log d)$ to sort the dimensions and $O(d)$ to search the pivot. For example, on the *Youtube* dataset, treewand took 5.48s which was 8 times faster than `WAND` (45.41s) and 30 times faster than naivek2 (152.45s) when $k = 3000$.

6.2 Evaluating the Approximation Algorithm

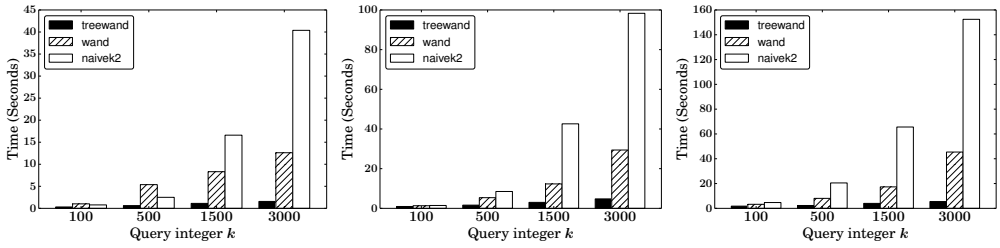
In this section, we evaluate the accuracy and efficiency of our approximation algorithm, denoted as approx, which cuts off unnecessary one-way paths while has theoretical accuracy guarantee. We also compare the efficiency of the approx algorithm with the exact algorithm, denoted as exact.

6.2.1 Accuracy

We first evaluate the influence of the user-specified accuracy requirement δ in Eq(14) on the "accuracy" of approx. Given a query integer k, if we denote the answer pair set returned by exact and approx as $\mathcal{P}_E$ and $\mathcal{P}_A$ respectively, then the accuracy is defined as $\frac{|\mathcal{P}_E \cap \mathcal{P}_A|}{k}$. Intuitively, the "accuracy" describes the proportion of the "right positioned pairs". We evaluated the accuracy of three different δ values, $10^{-2}, 10^{-3}$ and 10^{-4}. We denote the approximation algorithm which uses the vector matrix generated by $\delta = 10^{-2}, \delta = 10^{-3}, \delta = 10^{-4}$ as respectively 1e-2, 1e-3 and 1e-4. Figure 8 shows the results. We can see from Figure 8 that for $\delta = 10^{-2}$, the accuracy was always less than 90% for each k over the three real datasets. But for $\delta = 10^{-3}$ and 10^{-4}, approx achieved a higher accuracy which was above 95% and 98% respectively, because larger δ values were prone to cut off more one-way paths and thus lead to the loss in accuracy. We can also observe that for the same δ, the accuracy values on three datasets had no large differences. The reason of "graph independence" is that Algorithm 5 gives **each node** an accuracy "budget" δ, subtracts from δ the upper bounds of the accuracy losses and prunes the branches. This makes accuracy only relate to the vicinity of a single node rather than the entire graph.

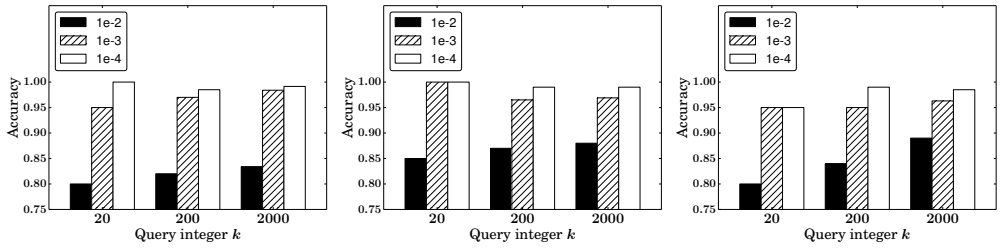
6.2.2 Running Time

We evaluated the running time of the approx algorithm with three different δ values. The overall running time of



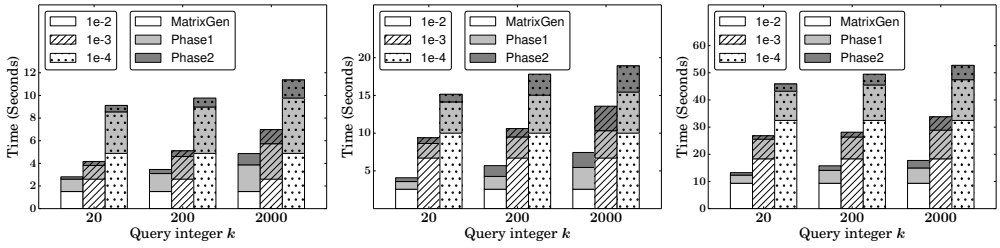
(a) *Epinion* (b) *Berkstan* (c) *Youtube*

Figure 7: Evaluation on Phase 2



(a) *Epinion* (b) *Berkstan* (c) *Youtube*

Figure 8: Accuracy of the approximation algorithms with different δ



(a) *Epinion* (b) *Berkstan* (c) *Youtube*

Figure 9: Running time of approximation algorithms

Table 2: Percentage of lists skipped by Algorithm 3

	$k = 20$	$k = 200$	$k = 2000$
Epinion	98.83%	98.47%	98.10%
Berkstan	99.85%	99.68%	99.52%
Youtube	99.28%	99.10%	98.93%

Table 3: The ratio of #non-zero to n^2

Dataset	exact	1e-2	1e-3	1e-4
Epinion	0.8×10^{-1}	7.7×10^{-5}	4.2×10^{-4}	1.4×10^{-4}
Berkstan	1.5×10^{-1}	7.9×10^{-6}	6.2×10^{-5}	4.5×10^{-4}
Youtube	0.3×10^{-1}	3.6×10^{-6}	3.5×10^{-5}	5.5×10^{-4}

approx consists of three parts - the vector matrix generation (denoted as MatrixGen), the Phase1 and the Phase2 of our join framework. Figure 9 depicts the detailed running time of different parts of the approx algorithm.

We made the following observations. First, the total running time increased when δ decreased as smaller δ values limited the pruning power of one-way paths in Algorithm 5. Second, Phase1 which generated $O(k)$ number of candidate nodes from all n nodes had high efficiency. For example, on *Youtube*, a graph with more than 1M nodes and 4M edges, 1e-3 returned the candidate in 7.2 seconds for $k = 20$ and in 8 seconds for $k = 200$. This efficiency should be attributed to the early-termination strategy adopted in Algorithm 3.

Table 2 shows the percentage of skipped inverted lists in Line 13 of the 1e-3. Although the number of skipped lists decreased as k increased because larger k values will have slower growth of the threshold θ in Algorithm 3, the results on three real datasets show that our method still skipped more than 98% of the lists and thus our early-termination strategy was very powerful.

Third, the running time of our approximation algorithms were not sensitive to the query integer k. For example, on *Berkstan*, the overall running time for $k = 20$, 200 and $2,000$ of 1e-4 were 15.2s, 17.8s and 18.9s. We analyzed the reason for all the three parts of approx. (1) For the MatrixGen part, the running time were identical for any k. (2) The early termination in Algorithm 3 did not rely too much on k as shown in Table 2. (3) The running time of Phase2 was very small as compared to the overall running time. Thus, although it was subjected to k, the influence of k on it was ignorable when we evaluated the overall time.

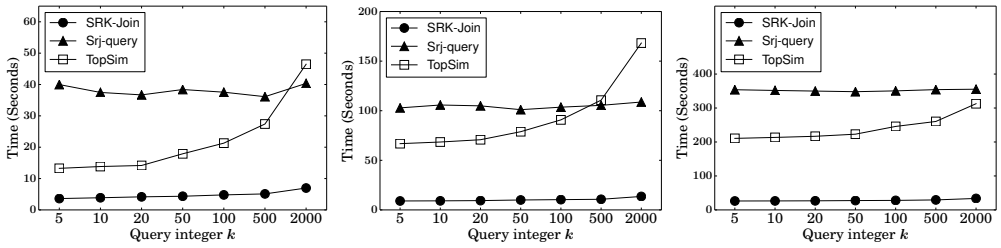
Fourth, as shown in Figures 8 and 9, $\delta = 10^{-3}$ was an optimal value for pragmatic uses among all three values to trade off accuracy for time. It maintained an accuracy higher than 95% and ran less than 35 seconds on all three datasets.

Fifth, on all the three real datasets, the overall time of exact were 476s, 3,594s and 5,278s. It indicated that (1) approx was much faster than exact; (2) for real graphs which are always scale-free, exact suffered a lot from the existence of high-degree nodes. Table 3 shows the ratio of non-zero elements in the vector matrix. We can see that the approximation algorithms significantly reduced the number of non-zero elements in the vector matrix.

To deeply compare the exact algorithm and the approx algorithm, we used the Ed generator to generate graphs with evenly distributed in-degrees. The results are shown in Table 4. We can see that on *SYN1* and *SYN2* which are relatively sparse and have evenly distributed in-degrees, the running time of exact was tolerable and approx was faster.

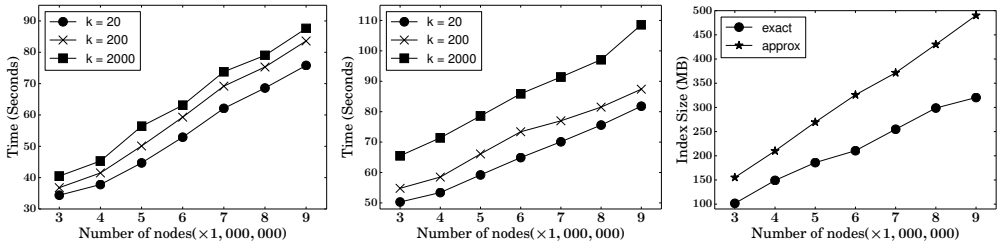
6.3 Comparison with State-of-the-art Methods

We extended two state-of-the-art methods Srj-query and TopSim which solve the SimRank-based similarity join problem and the top-k SimRank search problem respectively to



(a) *Epinion* (b) *Berkstan* (c) *Youtube*

Figure 10: Comparison with state-of-the-art algorithms Srj-query and TopSim



(a) Running time of exact (b) Running time of approx (c) Index size

Figure 11: Scalability

support our problem and compared the performance.

Srj-query [20] aims to solve the SimRank-based similarity join problem which takes a threshold θ as the input and returns all pairs of nodes whose SimRank values exceed θ. We extended Srj-query to solve the `SRK-Join` problem by dynamically tuning θ and adjusting the searching bounds according to the number of returned pairs. Srj-query had a very time-consuming off-line phase for creating indices, and we discarded the time of this phase and compared our overall running time with its online query time. Moreover, Srj-query used a partition-based approach for large graphs which was also an approximation approach. We set the approximation ratio of Srj-query to 10^{-3}.

TopSim [10] finds the most similar nodes with regard to a query node. We extended it to our join framework by first selecting candidate node set using TopSim and then extracting the most similar pairs among the candidate nodes. TopSim adopts the random surfer model and prunes two-way paths to offer an approximation algorithm for large graphs. We also set the approximation ratio to 10^{-3}.

LS-Join [14] studies link-based similarity joins, which identify top-k similar pairs from two (usually small) sets of nodes and can support SimRank, personalized PageRank, and common neighbors. However LS-Join did not fully utilize the intrinsic feature of SimRank and was expensive for two large sets. For example, on *Epinion*, when $k = 100$, it took more than one hour while our method only took less than 5 seconds. So we did not include LS-Join in the comparison.

To enable comparison, we set the accuracy requirement $\delta = 10^{-3}$ and used the approximation algorithm to compare with Srj-query and TopSim. We denoted our method as SRK-Join. Figure 10 shows the results. On *Epinion* and *Berkstan* where D is relatively large, TopSim was better than Srj-query for small k values because it used k to do early termination. But for large k values, e.g., $k = 2000$, TopSim became very inefficient. Our method outperformed both of the two approaches significantly for any query integers, even by an order of magnitude. The main reason is that, (1) Srj-query used an offline strategy to solve the SimRank equation system (Eq(2)). As the graph became increasingly large, the number of equations grew quadratically. (2) TopSim aggregated two-way paths and had a time complexity of $O(nD^{2\xi})$, thus when D was large, it had to enumerate a huge number of paths. Instead, our method aggregated one-way paths and utilized the information of one-way paths to calculate the summation of two-way paths. For instance, on *Berkstan*, when $k = 500$, Srj-query took 105s and TopSim took 110s. Our method improved it to 13s.

Table 4: Running time: exact vs approx on synthetic datasets ($k = 200$)

	$\|V\|$	I_{min}	I_{max}	exact	1e-2	1e-3	1e-4
SYN1	10,000	2	5	5.6s	2.1s	4.1s	5.3s
SYN2	100,000	1	4	12.4s	7.3s	9.9s	11.8s

Table 5: Comparison of the running time with state-of-the-art methods over synthetic datasets($k = 200$)

	$\|V\|$	I_{min}	I_{max}	exact	Srj-query	TopSim
SYN3	300,000	2	5	15.9s	78.9s	110.6s
SYN4	500,000	1	5	22.6s	107.6s	130.3s
SYN5	800,000	1	3	31.8s	306.1s	254.2s

On *Youtube*, a relatively sparse and huge social network, TopSim outperformed Srj-query for all tested query integers and our method was $7 \sim 9$ times faster than TopSim. This is because Srj-query had to tackle a significant number of equations on a huge graph, although it could utilize off-line indices. We can also see that the running time of TopSim varied significantly with the query integers. On the other hand, our method had very stable performance for different k values. For example, when $k = 20$, the running time of our method and TopSim were 27s and 217s, when $k = 2000$, they became 33s and 312s respectively.

We also compared the performance of the exact algorithms. The exact form of the two-state-of-the-art works cannot support scale-free graphs, thus we use graphs generated by ED. Table 5 shows the results. We can see that our exact algorithm outperformed TopSim and Srj-query greatly.

6.4 Evaluating Scalability

In this section, we used synthetic datasets to test the scalability of our methods. We used ED to generate sparse and in-degree evenly distributed graphs to test the scalability of exact. The scalability of approx was tested by scale-free

graphs which were generated by SCALE-FREE.

Figure 11 shows the results. We can see that the running time of both exact and approx achieved linear scalability. For example, for exact, the overall running time on graphs with 4M, 5M and 6M nodes when $k = 200$ were respectively 42s, 50s and 59s. For approx, the overall running time on graphs with 5M, 6M, 7M nodes when $k = 2000$ were respectively 79s, 86s and 92s. We also evaluated the scalability of the index size, i.e, the memory usage of the entire vector matrix, and the results are shown in Figure 11(c). We can see that both two algorithms had very good scalability. This is attributed to our transformation from the iterative SimRank computation to the dot product calculation and our pruning and early-termination techniques.

7. RELATED WORK

Recently significant efforts have been devoted to optimize the calculation of all-pair SimRank [15,17,19]. Lizorkin et. al. [11] proposed an accuracy-guaranteed method to efficiently compute all-pair SimRank in $O(n^3)$ time. There have been many works which focused on new queries that accessed only a small portion of nodes. Lee et. al. [10] studied the top-k search problem which returns the most similar nodes with regard to a query node. Kusumoto et. al. [8] also addressed the top-k search problem using a linear recursive framework and obtained similar results to [10]. Fujiwara et. al. [5] focused on the range search problem which returns the nodes whose similarities with a query node exceed a specified threshold. He et. al. [6] concentrated on solving the single pair query problem. Sun et. al. [14] studied the link-based similarity join problem, which can support personalized PageRank, SimRank, and common neighbors. However it is expensive for two large node sets because it cannot utilize the intrinsic feature of SimRank to optimize the join operation. Zheng et. al. [20] studied a more fundamental database operation, the traditional similarity join problem, where a user inputs a threshold t and requires the system to return all pairs of nodes whose SimRank values exceed t. Yu et. al. [18] addressed a "zero-SimRank" issue and proposed methods to improve the quality of the SimRank metric. Antonellis et. al. [1] proposed a refined SimRank similarity called SimRank++.

Different from existing works, we study how to efficiently identify the top-k similar node pairs from a graph based on SimRank. We extended state-of-the-art threshold-based similarity join algorithm Srj-query [20], top-k search method TopSim [10], and LS-Join [14] to support our problem and compared them with our algorithm. Experiment results in Section 6.3 show that our method significantly outperformed them. Nonetheless, it is not feasible to revise all-pair or single-pair SimRank computation methods to solve the SRK-Join problem because even if we compute the SimRank value for each pair in $O(1)$ time, we still have to make a quadratic number of computations. On the other hand, our method first builds the vector matrix by visiting only a small portion of neighbors of each node and uses a fast inverted-list-based method to select only $2k$ candidate nodes. Thus our method is efficient for the top-k join problem.

The other related studies focused on dot product similarity. Broder et. al. [2] proposed WAND algorithm for top-k information retrieval. Although there have been many works which focused on optimizing the WAND algorithm such as [4,13,16], none of them made WAND efficient for long queries. Other works include top-k cosine similarity join [9,12].

8. CONCLUSION

We have studied the top-k SimRank-based similarity join problem. We encoded a node to a vector by summarizing all one-way paths between this node to its neighbors. We converted the calculation of SimRank similarities to calculating the dot product between vectors. We designed a two-step framework to find top-k similar pairs. In the first phase, we proposed effective techniques to reduce the candidate node size to $O(k)$. In the second step, we developed a tree-based WAND algorithm to efficiently identify answers based on the candidate nodes. We also devised an approximate algorithm under user-specified accuracy requirement. Experiments on both real and synthetic datasets showed our method achieved high performance and good scalability.

Acknowledgement. This work was partly supported by the 973 Program of China (2015CB358700 and 2011CB302206), and the NSFC project (61373024 and 61422205), YETP0105, Tencent, Huawei, SAP, the "NExT Research Center" (WBS:R-252-300-001-490), and the FDCT/106/2012/A3.

9. REFERENCES

[1] I. Antonellis, H. Garcia-Molina, and C.-C. Chang. Simrank++: query rewriting through link analysis of the clickgraph (poster). In *WWW*, pages 1177–1178, 2008.

[2] A. Z. Broder, D. Carmel, M. Herscovici, A. Soffer, and J. Y. Zien. Efficient query evaluation using a two-level retrieval process. In *CIKM*, pages 426–434, 2003.

[3] C. Cooper and A. M. Frieze. Random walks with look-ahead in scale-free random graphs. *SIAM J. Discrete Math.*, 24(3):1162–1176, 2010.

[4] M. Fontoura, V. Josifovski, J. Liu, S. Venkatesan, X. Zhu, and J. Y. Zien. Evaluation strategies for top-k queries over memory-resident inverted indexes. *PVLDB*, 4(12):1213–1224, 2011.

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

[6] J. He, H. Liu, J. X. Yu, P. Li, W. He, and X. Du. Assessing single-pair similarity over graphs by aggregating first-meeting probabilities. *Inf. Syst.*, 42:107–122, 2014.

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

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

[9] D. Lee, J. Park, J. Shim, and S. goo Lee. An efficient similarity join algorithm with cosine similarity predicate. In *DEXA (2)*, pages 422–436, 2010.

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

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

[12] Y. Low and A. X. Zheng. Fast top-k similarity queries via matrix compression. In *CIKM*, pages 2070–2074, 2012.

[13] O. Rojas, V. G. Costa, and M. Marín. Efficient parallel block-max wand algorithm. In *Euro-Par*, pages 394–405, 2013.

[14] L. Sun, R. Cheng, X. Li, D. W. Cheung, and J. Han. On link-based similarity join. *PVLDB*, 4(11):714–725, 2011.

[15] Y. Sun, J. Han, X. Yan, P. S. Yu, and T. Wu. Pathsim: Meta path-based top-k similarity search in heterogeneous information networks. *PVLDB*, 4(11):992–1003, 2011.

[16] H. Yan, S. Ding, and T. Suel. Inverted index compression and query processing with optimized document ordering. In *WWW*, pages 401–410, 2009.

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

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

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

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