

Article

HitSim: An Efficient Algorithm for Single-Source and Top-k SimRank Computation

Jing Bai ^{1,2,†}, Junfeng Zhou ^{2,*,†} , Shuotong Chen ^{2,†}, Ming Du ^{2,†}, Ziyang Chen ^{3,†} and Mengtao Min ^{2,†}

¹ College of Information Technology, Shanghai Jian Qiao University, 1111 Hucheng Ring Road, Pudong New Area, Shanghai 201306, China; 1199132@mail.dhu.edu.cn

² School of Computer Science and Technology, Donghua University, 2999 Renmin North Road, Songjiang District, Shanghai 201620, China; duming@dhu.edu.cn (M.D.)

³ School of Information Management, Shanghai Lixin University of Accounting and Finance, 2800 Wenxiang Road, Songjiang District, Shanghai 201209, China

* Correspondence: zhoujf@dhu.edu.cn

† These authors contributed equally to this work.

Abstract: SimRank is a widely used metric for evaluating vertex similarity based on graph topology, with diverse applications such as large-scale graph mining and natural language processing. The objective of the single-source and top-k SimRank query problem is to retrieve the k vertices with the largest SimRank to the source vertex. However, existing algorithms suffer from inefficiency as they require computing SimRank for all vertices to retrieve the top-k results. To address this issue, we propose an algorithm named *HitSim* that utilizes a branch and bound strategy for the single-source and top-k query. *HitSim* initially partitions vertices into distinct sets based on their shortest-meeting lengths to the source vertex. Subsequently, it computes an upper bound of SimRank for each set. If the upper bound of a set is no larger than the minimum value of the current top-k results, *HitSim* efficiently batch-prunes the unpromising vertices within the set. However, in scenarios where the graph becomes dense, certain sets with large upper bounds may contain numerous vertices with small SimRank, leading to redundant overhead when processing these vertices. To address this issue, we propose an optimized algorithm named *HitSim-OPT* that computes the upper bound of SimRank for each vertex instead of each set, resulting in a fine-grained and efficient pruning process. The experimental results conducted on six real-world datasets demonstrate the performance of our algorithms in efficiently addressing the single-source and top-k query problem.

Keywords: vertex similarity; SimRank; single-source and top-k query



Citation: Bai, J.; Zhou, J.; Chen, S.; Du, M.; Chen, Z.; Min, M. HitSim: An Efficient Algorithm for Single-Source and Top-k SimRank Computation. *Information* **2024**, *15*, 348. <https://doi.org/10.3390/info15060348>

Academic Editor: Yong Gao

Received: 19 March 2024

Revised: 1 May 2024

Accepted: 4 June 2024

Published: 12 June 2024



Copyright: © 2024 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

1. Introduction

In various real-world scenarios, the measurement of similarity between entities is often crucial. For instance, in recommendation systems, predicting potential friendships based on the similarity between individuals in a social network [1,2]; or recommending items to users based on their behavior and preferences [3]. In security systems, analyzing email similarity to detect spam messages [4]; or analyzing account similarity to identify fraudulent transactions [5]. Among existing similarity computation methods, those based on entity linkage relationships are most commonly employed. Among these, SimRank is a widely used model for computing vertex similarity based on the directed graph topology. SimRank was introduced by Jeh and Widom [6] in 2002, and it is formulated based on two intuitive statements: (1) If two entities (vertices) are referenced by similar entities (i.e., in a directed graph, if the in-neighbors of two different vertices are similar or identical), then those two entities are also considered similar, and (2) an entity is most similar to itself. The classical algorithm, named *power method* [6], along with its variations [7], serves as the foundation for computing SimRank between each two vertices. However, these algorithms suffer from the time and space complexity of $\Omega(n^2)$, where n represents

the total number of vertices in the graph G , as there exist $\Omega(n^2)$ vertex pairs in G . To tackle this challenge, Ref. [8] proposed the single-source and top- k query problem. This problem focuses on efficiently retrieving the k vertices with the largest SimRank to a specific source vertex. Existing approaches [8–16] for the single-source and top- k query problem primarily utilize the random walk method, resulting in notable improvements in both time and space efficiency. Ref. [10] proposed two heuristic algorithms employing truncated random walk and prioritized propagation strategies. Ref. [11] devised an index for SimRank computation based on $\sqrt{c}$ -walk, despite the notable overhead in terms of space and preprocessing. Ref. [12] proposed an index-free algorithm named *ProbeSim* that outperforms the index-based approaches. The state-of-the-art algorithm *CrashSim* [13] improves the computational efficiency of SimRank in *ProbeSim* by truncating walk lengths.

Challenges. In the single-source and top- k query problem, existing approaches require computing the SimRank for all the vertices and subsequently sorting the top- k results in descending order based on their SimRank. However, computing SimRank for vertices requires sampling a large number of random walks, which is a time-consuming operation. Moreover, in real-world applications, the desired result scale k specified by users is often much smaller than the vertex scale n of the network. Therefore, computing SimRank for vertices with negligible SimRank value is redundant.

Our approach. Motivated by the above observations, we propose a novel algorithm called *HitSim* for the single-source and top- k query problem, which employs a branch and bound strategy. Specifically, by leveraging the inherent property that the SimRank of a vertex will decrease as its meeting length increases, *HitSim* partitions vertices into distinct sets based on their shortest-meeting lengths (Definition 5) and computes the upper bound of SimRank for each set. To reduce the redundant computation for the vertices with negligible SimRank, *HitSim* preferentially processes the vertices within the set with a larger upper bound. If the upper bound of a set is less than the minimum SimRank of the current results, the vertices within the set can be efficiently pruned in batch.

However, in scenarios where the graph becomes dense, the number of vertices within the same set can grow rapidly. This may result in a scenario where vertices with the same shortest-meeting lengths but significantly different SimRank are partitioned into the same set. Consequently, the efficiency will decrease since the algorithm may preferentially process the vertices with the shortest-meeting lengths but with a small SimRank. To address this issue of inefficient pruning in *HitSim*, we propose an optimized algorithm called *HitSim-OPT*. *HitSim-OPT* computes the upper bound of SimRank for each vertex, allowing for fine-grained pruning of vertices. Similar to *HitSim*, *HitSim-OPT* maintains the minimum SimRank of the current top- k results and prunes unpromising vertices by comparing their upper bounds with the minimum value. Our contributions are as follows:

- We propose an efficient algorithm, named *HitSim*, based on a branch and bound strategy to answer the single-source and top- k query. *HitSim* can efficiently prune the unpromising vertices within a set in batch, reducing redundant computation and improving overall efficiency.
- We further propose an optimized algorithm, named *HitSim-OPT*. By computing the upper bound of SimRank for each vertex, *HitSim-OPT* performs a fine-grained pruning strategy, resulting in further improvements in efficiency.
- We conduct experiments on six real-world datasets. The experimental results show that our algorithms can efficiently answer the single-source and top- k query.

Organization. The rest of this paper is summarized as follows. Section 2 provides some preliminaries. In Section 3, we give a review of existing works. In Section 4, we propose an efficient algorithm based on a branch and bound strategy for the single-source and top- k query. In Section 5, we further propose an optimized algorithm. Section 6 shows the experimental results. Section 7 concludes this paper.

2. Preliminaries

In this section, we formally introduce the notation and definitions. Mathematical notations used throughout this paper are summarized in Table 1.

Table 1. Summary of notation.

Notation	Description
G	Directed graph
V, E	The vertex and edge sets of G
n, m	The number of vertices and edges in G , $n = V $, $m = E $
$I(u), Out(u)$	Sets of in-neighbors and out-neighbors of vertex u
l_{max}	Limited length of $\sqrt{c}$ -walk
c	Decay factor in the definition of SimRank
$s(u, v)$	SimRank of two vertices u and v
$\bar{s}(u, v)$	A SimRank estimation of $s(u, v)$
$\bar{s}_{min}$	The minimum value of SimRank of current results
B_v, B_M	The upper bound of SimRank of vertex v , set M
$W(u)$	A reverse $\sqrt{c}$ -walk starting from u
ε	The difference between $\bar{s}(u, v)$ and $s(u, v)$
δ	The failure probability

Definition 1 (SimRank). Given two vertices u and v in directed graph $G = (V, E)$, the SimRank of u and v , denoted as $s(u, v)$, is defined as

$$s(u, v) = \begin{cases} 1, & u = v \\ \sum_{p \in I(u)} \sum_{q \in I(v)} \frac{c \times s(p, q)}{|I(u)| \times |I(v)|}, & u \neq v \end{cases} \quad (1)$$

where $I(u)$ denotes the set of in-neighbors of u , and $c \in (0, 1)$ is a decay factor [12,13].

For example, in G_1 of Figure 1, if the decay factor $c = 0.64$, according to Equation (1), we have $s(v_0, v_0) = 1$. The SimRank between v_1 and v_2 is $s(v_1, v_2) = \sum_{p \in I(v_1)} \sum_{q \in I(v_2)} \frac{c \times s(p, q)}{|I(v_1)| \times |I(v_2)|} = 0.64$. Similarly, $s(v_1, v_3) = s(v_2, v_3) = 0.64$, $s(v_4, v_5) = s(v_5, v_6) = 0.486$, $s(v_4, v_6) = 0.467$, and the SimRank between other vertex pairs is 0.

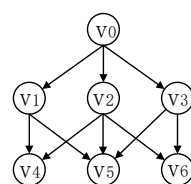


Figure 1. Directed graph G_1 .

Definition 2 (Reverse random walk). Given vertex u in directed graph G , a reverse random walk from u is a sequence of vertices $W(u) = (u_0, u_1, u_2, \dots)$, such that $u_{i+1} (i \geq 0)$ is selected uniformly at random from the in-neighbors of u_i .

Definition 3 ($\sqrt{c}$ -walk [11]). Let c denote the decay factor, a $\sqrt{c}$ -walk in G is defined such that (1) in each step of the reverse random walk, we have $1 - \sqrt{c}$ probability of stopping; (2) for the remaining $\sqrt{c}$ probability, one of the in-neighbors of the current vertex is selected uniformly at random as the next step. We denote a $\sqrt{c}$ -walk starting from u as $W(u) = (u_0, u_1, \dots, u_i, \dots)$, where $u_0 = u$.

According to Definition 3, Ref. [11] also defined the SimRank estimation $\bar{s}(u, v)$ as the total probability that $\sqrt{c}$ -walk $W(u)$ starting from u meets $\sqrt{c}$ -walk $W(v)$ starting from v , i.e., $\bar{s}(u, v) = \Pr[W(u) \text{ and } W(v) \text{ meet}]$.

Definition 4 (First-meeting probability [12]). Given a reverse path $\sqrt{c}$ -walk $W(u) = (u_0, \dots, u_i)$ starting from u ($u_0 = u$) and $v \in V$ ($v \neq u_0$), the first-meeting probability of v with respect to $W(u)$ is defined as

$$P(v, W(u)) = \Pr_{W(v)}[v_i = u_i, v_{i-1} \neq u_{i-1}, \dots, v_0 \neq u_0]. \quad (2)$$

where $W(v) = (v_0, \dots, v_i, \dots)$ is a random $\sqrt{c}$ -walk starting from $v_0 = v$.

According to Definition 3 and 4, Ref. [12] defined $\bar{s}(u, v)$ as the total probability that $\sqrt{c}$ -walk $W(u)$ starting from u and $\sqrt{c}$ -walk $W(v)$ starting from v first meet at each vertex u_i , i.e.,

$$\bar{s}(u, v) = \Pr[W(u) \text{ and } W(v) \text{ meet}] = \sum_i \Pr[W(u) \text{ and } W(v) \text{ first meet at } u_i]. \quad (3)$$

Definition 5 (Shortest-meeting length). Given any two $\sqrt{c}$ -walks $W(v) = (v_0, \dots, v_t, \dots)$ starting from v ($v_0 = v$) and $W(u) = (u_0, \dots, u_t, \dots)$ starting from u ($u_0 = u$), $u \in V$ and $v \in V$ ($v \neq u$), the shortest-meeting length t between u and v is defined as the least t such that the probability of meeting after t steps is nonzero.

For example, in Figure 1, any two $\sqrt{c}$ -walks $W(v_4)$ starting from v_4 and $W(v_5)$ starting from v_5 may first meet at v_1 or v_2 with walking one step, at v_0 with walking two steps. Then, the shortest-meeting length between them is 1.

Definition 6 (Approximate single-source SimRank query). In the graph $G = (V, E)$, given source u , the average absolute error ϵ allowed in SimRank computation, and the failure probability δ , for any vertex v ($v \neq u$), of an approximate single-source SimRank query returns an estimated SimRank $\bar{s}(u, v)$ to the ground-truth SimRank $s(u, v)$, which satisfies

$$\Pr\{|s(u, v) - \bar{s}(u, v)| \leq \epsilon\} \geq 1 - \delta. \quad (4)$$

Problem Statement. (Approximate single-source and top-k query) In graph $G = (V, E)$, given source vertex u , decay factor c , and integer k , return the top-k vertices $V_k(u) = \{v_i \neq u, 1 \leq i \leq k\}$ with the largest SimRank.

3. Related Works

In this section, we review the state-of-the-art algorithms for the single-source and top-k query problem, which are based on the widely used random walk method.

Ref. [11] first proposed the *SLING* algorithm, which utilizes the $\sqrt{c}$ -walk to compute SimRank. Subsequently, several studies [9,11,14] adopted the *Monte Carlo (MC)* method to sample $\sqrt{c}$ -walks for each vertex v and source u a certain number of times. By counting the number of times they meet (*cnt*) out of the total number of walks sampled (*num*), the SimRank is obtained as $\bar{s}(u, v) = \text{cnt} / \text{num}$. After computing the SimRanks with source u for all vertices and sorting them in descending order, the algorithms return the top-k results. Ref. [15] proposed the *TSF* algorithm based on the *MC method*. *TSF* constructs an index comprising *one-way graphs* that contain the coupling of random walks of length T from each vertex. This approach helps reduce storage space requirements. To address the computational cost of simulations, Ref. [16] proposed an index-based algorithm called *READS*. The index consists of compressed $\sqrt{c}$ -walks, which significantly improves the efficiency of queries. In the case of *MC*-based algorithms, there is a problem that when the SimRank of many vertices is negligible, resulting in the $\sqrt{c}$ -walks within a limited length not meeting the path of the source u , then the processing of these vertices is redundant. To tackle this issue, an index-free algorithm called *ProbeSim* was proposed by Ref. [12]. Instead of sampling $\sqrt{c}$ -walks from each vertex v to determine whether v and source u can meet at any u_i within the $\sqrt{c}$ -walk $W(u)$ from u , *ProbeSim* conducts a graph traversal from each u_i to identify vertices that have a non-negligible probability of walking to u_i .

This process is repeated for n_r iterations to obtain results with approximate guarantees. By avoiding the processing of unpromising vertices, this approach significantly reduces computational overhead. However, it requires generating a large number of probing trees to determine whether $W(u)$ can meet every v at each step of the $\sqrt{c}$ -walk $W(u)$ starting from u .

To overcome the mentioned issue, Ref. [13] proposed an improved algorithm called *CrashSim* that builds upon the principles of *ProbeSim*. The main idea behind *CrashSim* is to consider the SimRank as the average probability of two $\sqrt{c}$ -walks meeting within a limited length. *CrashSim* first computes a reverse reachable tree of source vertex u with a limited length of $\sqrt{c}$ -walk, denoted as l_{max} . It then iteratively generates a $\sqrt{c}$ -walk for each vertex v and determines whether it can meet the limited $\sqrt{c}$ -walk path from u with a non-negligible probability. This process is repeated for n_r iterations to obtain approximate results with certain guarantees. By traversing the reverse reachable tree for only the source vertex u instead of exploring the entire graph for each vertex $u_i \in V \setminus u$, *CrashSim* significantly reduces redundant computations.

However, the above mentioned algorithms require computing the SimRank for each vertex $u_i \in V \setminus u$, sorting them in descending order, and returning the top- k results. Such a method results in significant redundant computational overhead when querying for a small value of k , considering that k is typically much smaller than the total number of vertices n in real-world applications.

4. HitSim

Although the state-of-the-art algorithm *CrashSim* reduces redundant computation by traversing the reverse reachable tree for only source vertex u , it still generates $\sqrt{c}$ -walks for each vertex v to determine whether it can meet the $\sqrt{c}$ -walks from u within a limited length. However, since the parameter k is typically much smaller than the total number of vertices n , processing unpromising vertices with negligible SimRank values is redundant. To overcome this issue, we propose an efficient algorithm called *HitSim*. It utilizes a branch and bound strategy to batch-prune unpromising vertices to avoid the redundant processing. The algorithm is implemented in three steps, as follows.

Step 1: Computing the reverse reachable tree U . In the first step of *HitSim*, we perform the same approach as *CrashSim* [13]. This involves computing the reverse reachable tree from the source vertex u and generating a matrix U . Each element $U_{step}(v)$ in the matrix represents the probability of the $\sqrt{c}$ -walk $W(u)$ stopping at vertex v by walking $step$ steps.

Example 1. We illustrate the computation of the reverse reachable tree U using the graph G_2 shown in Figure 2. Given the source v_0 , we set the limited length l_{max} of $\sqrt{c}$ -walk $W(v_0)$ to 4 and decay factor c to 0.25 ($\sqrt{c} = 0.5$). The algorithm computes the reverse reachable tree of v_0 , which is shown in Figure 3. Note that the in-neighbor of vertex v that is equal to the parent of v is ignored to avoid recomputing the probability due to the cycle in the graph [13]. Simultaneously, it computes the probability of the $\sqrt{c}$ -walk stopping at different vertices with different lengths. For level 0, it sets the probability $U_0(v_0) = 1$. Next, for level 1, $U_1(v_1) = U_0(v_0) \cdot \frac{\sqrt{c}}{|I(v_1)|} = 1 \cdot \frac{0.5}{2} = 0.25$, $U_1(v_2) = U_0(v_0) \cdot \frac{\sqrt{c}}{|I(v_2)|} = 1 \cdot \frac{0.5}{3} = 0.167$. Similarly, for level 2, $U_2(v_4) = 0.0625$, $U_2(v_1) = 0.0417$, $U_2(v_3) = 0.0417$, and for level 3, $U_3(v_7) = 0.0156$, $U_3(v_0) = 0.0104$, $U_3(v_4) = 0.0104$, $U_3(v_1) = 0.0104$.

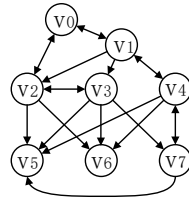


Figure 2. Directed graph G_2 .

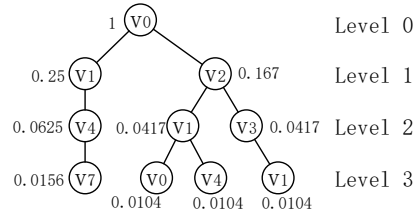


Figure 3. Reverse reachable tree of v_0 .

Step 2: Partitioning vertices. Based on *CrashSim*, assuming the x th sampling, if a $\sqrt{c}$ -walk starting from v , i.e., $W(v) = (v_0, \dots, v_t, \dots, v_i)$ ($v = v_0, t \in [1, i], 1 \leq i \leq l_{max}$), first meets the $\sqrt{c}$ -walk $W(u)$ starting from source u at v_t , then

$$\bar{s}_x(u, v) = U_0(v_0) + \dots + U_t(v_t) + \dots + U_i(v_i). \quad (5)$$

Lemma 1. Given a graph $G = (V, E)$ and a source vertex u , if there exists a vertex v ($v \neq u$), and its shortest-meeting length is t ($1 \leq t \leq l_{max}$), then the upper bound of SimRank for v , denoted as B_v , is $\sum_{l=t}^{l_{max}} p_l$, where p_l represents the maximum probability in level l of the reverse reachable tree of the source u .

Proof of Lemma 1. Since the shortest-meeting length of v is t ($1 \leq t \leq l_{max}$), any $\sqrt{c}$ -walk $W_x(v) = (v_0, \dots, v_t, \dots, v_i)$ ($v_0 = v, t \in [1, i], 1 \leq i \leq l_{max}$) starting from v will not meet the $\sqrt{c}$ -walk $W_x(u)$ starting from source u before v_t . Then, we have

$$U_0(v_0) = \dots = U_{t-1}(v_{t-1}) = 0. \quad (6)$$

Furthermore, p_l is the maximum probability in level l of the reverse reachable tree of source u . Then,

$$U_t(v_t) \leq p_t, \dots, U_i(v_i) \leq p_i, (t \in [1, i], 1 \leq i \leq l_{max}). \quad (7)$$

Based on Equations (5)–(7), for any $\sqrt{c}$ -walk $W_x(v) = (v_0, \dots, v_t, \dots, v_i)$ ($v_0 = v, t \in [1, i], 1 \leq i \leq l_{max}$) starting from v , we have

$$\bar{s}_x(u, v) \leq \sum_{l=t}^i p_l, (t \in [1, i], 1 \leq i \leq l_{max}). \quad (8)$$

Since $i \leq l_{max}$, then $\sum_{l=t}^i p_l \leq \sum_{l=t}^{l_{max}} p_l$. The average value after n_r trials is

$$\bar{s}(u, v) \leq \sum_{l=t}^{l_{max}} p_l. \quad (9)$$

Thus, for any vertex v , the upper bound of SimRank is $B_v = \sum_{l=t}^{l_{max}} p_l$, where t is the shortest-meeting length, i.e., Lemma 1 holds. $\square$

According to Lemma 1, for any two different vertices v and w ($v \neq u, w \neq u$), if the shortest-meeting length of v and w are t_v and t_w , respectively, then $B_v = \sum_{l=t_v}^{l_{max}} p_l$ and

$B_w = \sum_{l=t_w}^{l_{max}} p_l$. If $t_v < t_w$, then $B_v = \sum_{l=t_v}^{l_{max}} p_l > B_w = \sum_{l=t_w}^{l_{max}} p_l$. From the above analysis, we have Observation 1 as follows:

Observation 1. For any vertex v , the smaller the shortest-meeting length, the larger the upper bound B_v .

According to Lemma 1, vertices with the same shortest-meeting length share the same upper bound. By partitioning the vertices into distinct sets based on their shortest-meeting lengths, we ensure that vertices within the same set share the same upper bound, i.e., for any vertex v in the same set M , $B_v = B_M$, where B_M is the upper bound of SimRank for set M . If we identify a set M satisfying $B_M \leq \bar{s}_{min}$, where $\bar{s}_{min}$ is the minimum SimRank value of the current results, we can safely prune these unpromising vertices within M batches. According to Observation 1, we should preferentially process sets with smaller shortest-meeting lengths. Such preprocessing allows the algorithm to preferentially compute SimRank for vertices with a larger SimRank upper bound, leading to the effective pruning of unpromising vertices.

Step 3: Computing $\bar{s}(u, v)$ and maintaining the top-k results. Following step 2, *HitSim* generates $\sqrt{c}$ -walks and computes the SimRank only for the vertices within the sets whose upper bounds exceed $\bar{s}_{min}$. Simultaneously, *HitSim* maintains the top-k results and the current minimum SimRank $\bar{s}_{min}$ by sorting the results based on their SimRank in descending order.

Since step 1 is identical to *CrashSim*, we will omit its detailed description. The detailed descriptions of steps 2 (Section 4.1) and 3 (Section 4.2) are as follows.

4.1. Partitioning Vertices

We now describe the *ParVer* algorithm, which is used to partition vertices into distinct sets based on their shortest-meeting lengths with the source vertex. Given a graph $G = (V, E)$, a source vertex $u \in V$, a limited length l_{max} of $\sqrt{c}$ -walk, and a probability matrix U , *ParVer* returns a vertex set M_t and the upper bound of SimRank B_{M_t} for each $t = 1, \dots, l_{max}$, where t is the shortest-meeting length of the vertices within M_t . Notably, M_t does not include any vertices from the set M_{t-1} .

The pseudo-code of *ParVer* is depicted in Algorithm 1. It initializes a hashset M_t for each $t = 1, \dots, l_{max}$ to store the vertices with shortest-meeting lengths equal to t (line 1). To avoid revisiting vertices of the same level, it requires a hashset Du_t for each $t = 1, \dots, l_{max}$ to store the visited vertices (line 2). The probabilities p_t and B_{M_t} are initialized to store the maximum value of U_t and the upper bound of SimRank of M_t for each $t = 1, \dots, l_{max}$ (line 3). To identify all the vertices within M_t for each $t = 1, \dots, l_{max}$, it iterates over each vertex tpr in level t of the reverse reachable tree of source u (lines 4 to 20). For each vertex tpr in level t of the reverse reachable tree of source u , it records the current maximum probability p_t in level t (line 6). Then, it performs forward walks of t steps from tpr to obtain each vertex v whose shortest-meeting length is t (lines 7 to 20). Specifically, it first initializes a queue Q to store the vertices that tpr visits in t -step forward walks. Then, it puts tpr into Q and Du_i , where i represents the current level. Next, it forward walks from the vertex tpr and records the size of Q . With each iteration, the step i is reduced by 1. Then, it visits Q and pops the top element q of Q . For each out-neighbor v of q , if the step of the current forward walk does not exceed the maximum walk length i and v has not been visited before, then v is inserted directly into Du_i and Q . If v does not exist in any set that stores vertices with shortest-meeting lengths from 0 to $t - 1$ and $v \neq u$, it is added to M_t . Then, all the vertices within M_t for each $t = 1, \dots, l_{max}$ are obtained. Afterward, it computes the upper bound B_{M_t} based on Lemma 1 (lines 21 to 22). Finally, *ParVer* returns the vertex set M_t and its upper bound B_{M_t} for each $t = 1, \dots, l_{max}$ (line 23).

Algorithm 1: *ParVer*

Input: Graph $G = (V, E)$, source $u \in V$, limited length l_{max} of $\sqrt{c}$ -walk, and probability matrix U

Output: Vertex set M_t and upper bound B_{M_t} for each $t = 1, \dots, l_{max}$

```

1 Initialize hashset  $M_t$  for each  $t = 1, \dots, l_{max}$ ;
2 Initialize hashset  $Du_t$  for each  $t = 1, \dots, l_{max}$ ;
3 Initialize  $p_t = 0$  and  $B_{M_t} = 0$  for each  $t = 1, \dots, l_{max}$ ;
4 for  $t = 1, \dots, l_{max}$  do
5   for each vertex  $tpr$  in Level  $t$  of reverse reachable tree of  $u$  do
6      $p_t = \max(p_t, U_t(tpr))$ ;
7      $i \leftarrow t$ ;
8     Initialize a queue  $Q$ , and push  $tpr$  into  $Q$ ;
9     Insert  $tpr$  to  $Du_i$ ;
10    while  $Q$  is not empty and  $t > 0$  do
11       $len \leftarrow Q.size$ ;
12       $i \leftarrow i - 1$ ;
13      for  $j = 1, \dots, len$  do
14         $q$  is top element of  $Q$ , and pop it;
15        for each  $v \in Out(u)$  do
16          if  $i > 0$  and  $v \notin Du_i$  then
17            Insert  $v$  to  $Du_i$ ;
18            Push  $v$  into  $Q$ ;
19          else if  $i = 0$  and  $v \neq u$  and  $v \notin M_d (d = 1, \dots, t - 1)$  then
20            Insert  $v$  to  $M_t$ ;
21 for  $t = 1, \dots, l_{max}$  do
22    $B_{M_t} = \sum_{l=t}^{l_{max}} p_l$ ;
23 return  $M = \{M_1, \dots, M_{l_{max}}\}$  and  $B = \{B_{M_1}, \dots, B_{M_{l_{max}}}\}$ ;

```

Example 2. Considering the graph G_2 shown in Figure 2, we illustrate step 2, which involves the partitioning of vertices. Assume that the source vertex is v_0 . For simplicity, we set $l_{max} = 3$ and the decay factor $c = 0.25$ ($\sqrt{c} = 0.5$). In step 1, it computes the reverse reachable tree U , as shown in Example 1. Continuing from Example 1, in step 2, it partitions vertices of G_2 into distinct sets based on their shortest-meeting lengths with source v_0 . Starting with level 1 of the reverse reachable tree, it forward walks 1 step from the first vertex v_1 . Consequently, it inserts v_1 's out-neighbors v_2, v_3 , and v_4 into set M_1 . It omits v_0 since v_0 is the source vertex. Next, it forward walks one step from the second vertex v_2 in level 1. It inserts v_2 's out-neighbors v_5 and v_6 into set M_1 and omits v_3 since v_3 is already in M_1 . Thus, we have $M_1 = \{v_2, v_3, v_4, v_5, v_6\}$, and the maximum probability is $p_1 = \max\{U_1(v_1), U_1(v_2)\} = 0.25$. Moving to level 2, it forward walks 2 steps from v_4, v_1 , and v_3 , respectively. This leads to $M_2 = \{v_1, v_7\}$. Note that, v_2, v_3, v_4, v_5 , and v_6 are not inserted into M_2 since they are already in M_1 . The maximum probability is computed as $p_2 = \max\{U_2(v_4), U_2(v_1), U_2(v_3)\} = 0.0625$. Lastly, in level 3, it forward walks three steps from v_7, v_0, v_4 , and v_1 , respectively. As a result, $M_3 = \emptyset$, indicating that no new vertices are added to M_3 . The maximum probability is computed as $p_3 = \{U_3(v_7), U_3(v_0), U_3(v_4), U_3(v_1)\} = 0.0156$. Finally, according to Lemma 1, the SimRank upper bounds of set M_1 and M_2 are computed as $B_{M_1} = \sum_{l=1}^3 p_l = 0.25 + 0.0625 + 0.0156 = 0.3281$ and $B_{M_2} = \sum_{l=2}^3 p_l = 0.0625 + 0.0156 = 0.0781$, respectively.

4.2. Computing $\bar{s}(u, v)$ and Maintaining the Top-k Results

The complete pseudo-code of *HitSim* is illustrated in Algorithm 2. Given a graph $G = (V, E)$, a source $u \in V$, an integer k , a decay factor c , a limited length l_{max} of $\sqrt{c}$ -walk, an average absolute error ε , and a failure probability δ , *HitSim* returns the queue Q of the

top-k results. It first invokes the *revReach* algorithm (referenced as *CrashSim*) to construct the reverse reachable tree of source u and return a matrix U (line 1). Then, it invokes the *ParVer* algorithm (referenced as Algorithm 1) to partition vertices into distinct sets (line 2). Step 3, which involves computing $\bar{s}(u, v)$ and maintaining the top-k results, starts from line 3 of Algorithm 2.

In step 3, the algorithm initializes a priority queue Q to store the results (line 4). Based on Observation 1, for any vertex v , a smaller shortest-meeting length corresponds to a larger B_v . Thus, *HitSim* processes the sets in ascending order of their shortest-meeting lengths t in order to obtain the top-k results as early as possible (lines 5 to 20). Specifically, for each vertex v in M_t , it runs $n_r = \frac{3c}{\varepsilon^2} \log \frac{n}{\delta}$ independent trials (lines 6 to 16). The computation of n_r , which represents the minimum number of iterations that guarantees an error less than ε with at least $1 - \delta$, is defined by [13]. During each iteration, it generates a $\sqrt{c}$ -walk starting from v and limits the length of the walk to l_{max} (lines 8 to 9). Then, for the $\sqrt{c}$ -walk with length i , where $i \in [1, l_{max}]$, it accumulates the total first-meeting probability of this walk meeting the $\sqrt{c}$ -walk starting from u at the i th element of $W(v)$ (lines 10 to 12). After completing the n_r trials, it computes the average of the results to obtain the final SimRank $\bar{s}(u, v) = \frac{1}{n_r} \bar{s}_x(u, v)$ and inserts $(v, \bar{s}(u, v))$ into Q (lines 14 to 15). When the size of the current results is k , it checks whether the upper bound $B_{M_{t+1}}$ of the next set to be processed is larger than $\bar{s}_{min}$. If it is, *HitSim* continues generating $\sqrt{c}$ -walks for the vertices within M_{t+1} . Otherwise, it terminates and returns the current top-k results (lines 17 to 19).

Algorithm 2: *HitSim*

Input: Graph $G = (V, E)$, source $u \in V$, integer k , decay factor c , limited length l_{max} of $\sqrt{c}$ -walk, the average absolute error ε , and the failure probability δ

Output: $Q = \{v, \bar{s}(u, v)\}$

```

1  $U \leftarrow \text{revReach}(G, u, l_{max}, c);$  /*Step 1*/
2  $M, B \leftarrow \text{ParVer}(G, u, l_{max}, U);$  /*Step 2*/
3  $n_r \leftarrow \frac{3c}{\varepsilon^2} \log \frac{n}{\delta};$ 
4 Initialize priority queue  $Q$ ;
5 for  $t = 1, \dots, l_{max}$  do
6   for each  $v \in M_t$  do
7     for  $x = 1, \dots, n_r$  do
8       Generate a  $\sqrt{c}$ -walk  $W(v) = \{v_0, v_1, \dots, v_l\}$  starting from  $v$ , where
9          $v_0 = v$ ;
10        limit  $l \leq l_{max}$ ;
11        for  $i = 1, \dots, l$  do
12           $\bar{s}_x(u, v) + = U_i(W_i(v));$ 
13         $\bar{s}(u, v) = \frac{1}{n_r} \bar{s}_x(u, v);$ 
14        Maintain priority queue  $Q$  of size  $k$  and insert  $(v, \bar{s}(u, v))$  into  $Q$ ;
15      if  $Q.size = k$  and  $Q.top > B_{M_{t+1}}$  then
16        break;
17 return  $Q$ ;
```

Example 3. Continuing from Example 2, we set k to 1; the objective of *HitSim* is to return the top-1 result. The algorithm begins by processing the first vertex v_2 within the set $M_1 = \{v_2, v_3, v_4, v_5, v_6\}$ and $B_{M_1} = 0.3281$. Suppose that at the x th trial, it generates a $\sqrt{c}$ -walk $W(v_2) = (v_2, v_3, v_1, v_0)$ starting from v_2 . It computes the SimRank as $\bar{s}_x(v_0, v_2) = U_0(v_2) + U_1(v_3) + U_2(v_1) + U_3(v_0) = 0 + 0 + 0.0417 + 0.0104 = 0.0521$. After conducting a total of n_r iterations, it computes the average value of $\bar{s}(v_0, v_2) = \frac{1}{n_r} \bar{s}_x(v_0, v_2)$. Once all the vertices in M_1 have been processed, it compares the minimum value of SimRank $\bar{s}_{min}$ with the upper bound $B_{M_2} = 0.0781$ of the set M_2 . If $B_{M_2} \leq \bar{s}_{min}$, *HitSim* batch-prunes all the vertices in M_2 .

The SimRanks between v_0 and any other vertices are listed in Table 2, and they have been computed by [6] within a 10^{-5} error. From Table 2 we can see that, due to the branch and bound strategy, *HitSim* is able to efficiently batch-prune vertices such as v_1 and v_7 within M_2 , whose SimRanks are notably less compared to the rest.

Table 2. SimRank with respect to v_0 .

Vertex	v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7
$s(v_0, *)$	1.0	0.0064	0.048	0.132	0.074	0.04	0.048	0.0064

4.3. Analysis

The time and space complexity of *HitSim* can be analyzed in steps 1, 2, and 3.

- Step 1 (computing the reverse reachable tree U): In the worst case, it requires traversing each edge once and storing each vertex, resulting in a time complexity of $O(m)$ and the space complexity is $O(n)$.
- Step 2 (partitioning vertices): This step involves traversing each vertex of the reverse reachable tree and storing all the vertex sets. The worst case is that we need to traverse every edge in the graph by visiting the out-neighbors of each vertex, and the worst-case time complexity is $O(m)$. The space complexity is $O(n)$.
- Step 3 (computing $\bar{s}(u, v)$ and maintaining the top-k results): In the worst case, it requires running n_r trials of generating a $\sqrt{c}$ -walk for each vertex, with a limited length of l_{max} . Additionally, it requires maintaining a priority queue Q of size k . Thus, the time complexity is $O(n \cdot l_{max} \cdot \frac{3c}{\epsilon^2} \log \frac{n}{\delta} + n \log k)$, and space complexity is $O(n + k)$.

In summary, the total time complexity of *HitSim* is $O(2m + n \cdot l_{max} \cdot \frac{3c}{\epsilon^2} \log \frac{n}{\delta} + n \log k)$, and the total space complexity is $O(n + k)$.

5. Optimization

The batch-pruning method employed in *HitSim* aims to prune unpromising vertices within the sets whose upper bounds of SimRank are less than the minimum SimRank of the current results. However, in certain cases, such as Example 3, where k is set to 2, the batch-pruning strategy may fail. In the example, after computing the SimRank for all vertices in M_1 , the current top-2 results are v_3 and v_4 , with a minimum SimRank $\bar{s}_{min}$ of approximately 0.074, as shown in Table 2. Comparing $\bar{s}_{min}$ with the upper bound $B_{M_2} = 0.0781$, *HitSim* continues to compute the SimRank for vertices v_1 and v_7 within M_2 , indicating a failure of the batch-pruning method.

The reason for this failure is that in dense graphs, vertices may have a large number of in-neighbors. Vertices with the same shortest-meeting length but significantly different SimRanks can be partitioned into the same set, leading to imprecise pruning. For instance, in G_2 , the reverse reachable tree of v_7 (shown in Figure 4) reveals that v_7 cannot meet source v_0 at v_4 in level 2, where $U_2(v_4)$ is the maximum value in level 2. Similarly, v_7 cannot meet source v_0 at v_7 in level 3, where $U_3(v_7)$ is the maximum value in level 3. Consequently, although v_7 is partitioned into M_2 , its upper bound B_{M_2} , computed by accumulating the maximum values from level 2 and level 3, significantly exceeds its actual upper bound B_{v_7} . When processing such sets (e.g., M_2), it becomes inevitable to compute the SimRank for the vertices with a small SimRank (e.g., v_7), leading to the batch pruning being ineffective.

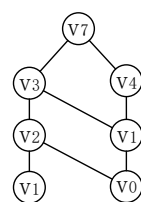


Figure 4. Reverse reachable tree of v_7 .

To address the issue, we propose an optimized algorithm called *HitSim-OPT*. *HitSim-OPT* aims to perform fine-grained and effective vertex pruning by computing a tighter upper bound of SimRank for each individual vertex, instead of that for each set. The optimized algorithm consists of the following three steps.

Step 1: Computing the reverse reachable tree U . The details of this step are omitted as they are the same as step 1 of *HitSim*.

Step 2: Computing the upper bound of SimRank for each vertex. According to Lemma 1, the upper bound of SimRank for any vertex v is computed by $B_v = \sum_{l=t}^{l_{max}} p_l$, where t represents the shortest-meeting length and p_l denotes the maximum probability in level l of the reverse reachable tree of u . To refine the upper bound B_v , we introduce a modification by replacing p_l with $p_l(v)$, which represents the maximum probability in level l of the reverse tree of v . Since each vertex tpv in the reverse reachable tree of v is the potential vertex within the $\sqrt{c}$ -walks $W(v)$ starting from v , $\sum_{l=t}^{l_{max}} p_l(v)$ provides a more precise bounding than $\sum_{l=t}^{l_{max}} p_l$.

By utilizing the inequality $\sum_{l=t}^{l_{max}} p_l(v) \leq \sum_{l=0}^{l_{max}} p_l(v)$, we can ensure $B_v \leq \sum_{l=0}^{l_{max}} p_l(v)$. Consequently, we can carefully modify the upper bound for each vertex v as

$$B_v = \sum_{l=0}^{l_{max}} p_l(v). \quad (10)$$

Step 3: Maintaining the top-k results. Similar to step 3 of *HitSim*, *HitSim-OPT* maintains the top-k results and the current minimum SimRank $\bar{s}_{min}$ by sorting the results based on their SimRank in descending order. For each vertex v , *HitSim-OPT* determines whether to further compute its actual SimRank by comparing B_v with $\bar{s}_{min}$. If B_v is less than $\bar{s}_{min}$, *HitSim-OPT* prunes v .

Since step 1 is identical to that of *HitSim*, we will omit its detailed description. The detailed descriptions of the algorithms in steps 2 (Section 5.1) and 3 (Section 5.2) are as follows.

5.1. Computing Upper Bound of SimRank for Each Vertex

A straightforward approach is to construct a reverse reachable tree for each vertex v and compute the maximum probability of v meeting the source vertex u at each level of u 's reverse reachable tree, as computed by Equation (10). Taking v_7 from Figure 2 as an example, we construct the reverse reachable tree of v_7 , as shown in Figure 4. At level 1, the maximum probability is 0 because v_7 cannot meet the source vertex v_0 with one step (see the reverse reachable tree of source v_0 in Figure 3). In level 2, the maximum probability is 0.0417 as v_7 can meet v_0 at v_1 with two steps. Similarly, in level 3, the maximum probability is 0.0104 as v_7 can meet v_0 at either v_1 or v_0 with three steps. By accumulating the maximum probabilities of the three levels, we obtain $B_{v_7} = 0 + 0.0417 + 0.0104 = 0.0521$.

However, constructing a reverse reachable tree for each vertex in the straightforward approach suffers from significant challenges in terms of time and space complexity. To address this challenge, we further propose a more efficient method called *UBV* (*Upper Bound of Vertex*) based on dynamic programming to compute the upper bound of SimRank for each vertex v . It utilizes an array dp_w of length l_{max} for each potential meeting point w to store the maximum probabilities at levels 0 to l_{max} . Such processing allows the algorithm to only traverse source u 's reverse reachable tree once instead of constructing a reverse reachable tree for each vertex. For each potential meeting point w in level t ($0 \leq t \leq l_{max}$), $dp_w(i)$ ($0 \leq i \leq l_{max}$) represents the maximum probability that v can meet u at level i . *UBV* dynamically computes $dp_w(i)$ for each w starting from the bottom of u 's reverse reachable tree. By the end of the traversal of level 0, a set of dp_w values can be obtained, where the vertices w in level 0 are those that can meet u within l_{max} steps. The upper bounds of SimRank for these vertices are computed as $\sum_{i=0}^{l_{max}} dp_w(i)$.

Lemma 2. For each potential meeting point w , the state array dp_w in level t is computed by the following equation.

$$dp_w(i) = \begin{cases} 0, & i < t \\ U_t(w), & i = t \\ \max\{dp_{tpr}(i)\}, & i > t \end{cases} \quad (11)$$

where $U_t(w)$ is the probability of source u traversing and stopping at vertex w , and tpr is the in-neighbor of vertex w .

Proof of Lemma 2. (1) When $i < t$, the $\sqrt{c}$ -walk starting from w at level t cannot meet any vertex at level i in the reverse reachable tree of source vertex u , thus $dp_w(i) = 0$; (2) When $i = t$, the $\sqrt{c}$ -walk starting from w at level t happens to meet w itself, thus $dp_w(i) = U_t(w)$; (3) When $i > t$, the $\sqrt{c}$ -walk starting from w at level t must pass through one of its in-neighbors, denoted as tpr . In this case, $dp_w(i)$ is equal to the maximum probability among its in-neighbors meeting at level i , i.e., $dp_w(i) = \max dp_{tpr}(i)$. $\square$

The pseudo-code of *UBV* is illustrated in Algorithm 3. *UBV* initializes an array dp_{tpr} for each tpr in the reverse reachable tree of source vertex u from the bottom to the top (lines 4 to 9). Subsequently, it updates dp_v for each out-neighbor of tpr based on Equation (11) (lines 10 to 20). Note that the vertices v at level 0 represent the vertices that can meet u within l_{max} steps, and $\sum_{i=0}^{l_{max}} dp_v(i)$ denotes the upper bounds of SimRank (lines 22 to 26).

Algorithm 3: *UBV*

Input: Graph $G = (V, E)$, limited length l_{max} , probability matrix U
Output: SimRank upper bounds B of all vertices

```

1 Initialize  $B_v$  for each  $v \in V$  ( $v \neq u$ );
2 Initialize a hash map  $H^t$  for each  $t = 0, \dots, l_{max}$ ;
3 for  $t = l_{max}$  to 0 do
4   if  $t > 0$  then
5     for each vertex  $tpr \in U_t$  do
6       Initialize vector  $dp$ , and set  $dp(t) = U_t(tpr)$ ;
7       Insert  $(tpr, dp)$  to  $H^t$ ;
8   if  $t < l_{max}$  then
9     for each vertex  $tpr \in H^{t+1}$  do
10      for  $v \in Out(tpr)$  do
11        Initialize vector  $dp$ , and set  $dp = H^t(v)$ ;
12        for  $l = 1, \dots, l_{max}$  do
13           $dp(l) = \max(H_l^{t+1}(tpr), dp(l))$ ;
14          Insert  $(v, dp)$  to  $H^t$ ;
15 for each  $v \in V$  do
16   for  $l = 1, \dots, l_{max}$  do
17      $B_v += H_l^0(v)$ ;
18 return  $B$ ;
```

Example 4. We use G_2 in Figure 2 to illustrate step 2 of *HitSim-OPT*, i.e., the computation of the SimRank upper bound for each vertex. Given source v_0 , for simplicity, we set $l_{max} = 4$ and the decay factor $c = 0.25$ ($\sqrt{c} = 0.5$). Continuing with Example 2, *HitSim-OPT* computes the upper bound of SimRank for each vertex. Specifically, *UBV* initializes a hash map H^t for each level t to store the array's dp . Starting from the bottom of u 's reverse reachable tree, it sets $dp_{v_7}(3) = U_3(v_7)$, $dp_{v_0}(3) = U_3(v_0)$, $dp_{v_4}(3) = U_3(v_4)$, and $dp_{v_1}(3) = U_3(v_1)$ based on Equation (11). After the first iteration ($t = l_{max}$), the result H^3 is shown in Figure 5. When $t = 2$, *UBV* first sets $dp_{v_4} = U_2(v_4)$, $dp_{v_1} = U_2(v_1)$, and $dp_{v_3} = U_2(v_3)$. Then, *UBV* computes

dp for all the out-neighbors of each $tpr \in H^3$. After this iteration, H^2 is obtained. Similarly, when $t = 1$ and 0 , H^1 and H^0 can be obtained. Finally, UBV computes $\sum_{i=0}^{l_{max}} dp_v(i)$ for each $v \in H^0$. The upper bounds of all vertices are $B_{v_2} = 0.3281$, $B_{v_3} = 0.3281$, $B_{v_4} = 0.3281$, $B_{v_5} = 0.2451$, $B_{v_6} = 0.2191$, $B_{v_1} = 0.0521$, $B_{v_7} = 0.0521$.

	v2	v3	v4	v5	v6	v1	v7		v
H^0 :	0	0	0	0	0	0	0	$dp_v(0)$	
	0.25	0.25	0.25	0.167	0.167	0	0	$dp_v(1)$	
	0.0625	0.0625	0.0625	0.0625	0.0417	0.0417	0.0417	$dp_v(2)$	
	0.0156	0.0156	0.0156	0.0156	0.0104	0.0104	0.0104	$dp_v(3)$	

	v1	v2	v5	v6	v7	v0	v3	v4	
H^1 :	0.25	0.167	0	0	0	0	0	0	
	0.0625	0.0417	0.0625	0.0625	0.0625	0.0417	0.0417	0.0417	
	0.0156	0.0104	0.0104	0.0156	0.0156	0.0104	0.0104	0.0104	

	v4	v1	v3	v5	v2	v6	v7	v0	
H^2 :	0.0625	0.0417	0.0417	0	0	0	0	0	
	0.0156	0.0104	0.0104	0.0156	0.0104	0.0104	0.0104	0.0104	

	v7	v0	v4	v1					
H^3 :	0.0156	0.0104	0.0104	0.0104					

Figure 5. Array dp of each vertex.

5.2. Maintaining the Top-k Results

To obtain the top-k results, the algorithm requires computing SimRank for vertices whose upper bounds of SimRank are larger than the current minimum SimRank $\bar{s}_{min}$. However, the computation of SimRank involves a large number of iterations for generating $\sqrt{c}$ -walks to obtain an average value. Ref. [13] defines the minimum number of iterations, denoted as n_r , that guarantees an error less than ε with at least $1 - \delta$. However, when n_r increases, the number of iterations for generating $\sqrt{c}$ -walks also increases. This can lead to significant time consumption when computing SimRank for the vertices with negligible SimRank. Here, we introduce an effective method to avoid the redundant generation for such vertices. Based on Lemma 3, at the x th iteration, we assume that the SimRanks of the remaining $n_r - x$ iterations are all computed as the upper bound B_v , if the average value of total n_r iterations is not larger than the current minimum SimRank $\bar{s}_{min}$; then, we can safely skip the computation of the remaining $n_r - x$ iterations and prune this vertex. This enables us to reduce computational overhead and improve efficiency.

Lemma 3. During the computation of SimRank for vertex v , at the x th iteration for generating a $\sqrt{c}$ -walk, if

$$\frac{\sum_{i=1}^x \bar{s}_i(u, v) + (n_r - x) * B_v}{n_r} \leq \bar{s}_{min} \quad (12)$$

then the remaining $n_r - x$ iterations can be safely skipped.

Proof of Lemma 3. Since B_v is the upper bound of the SimRank of v , for any y th trial, we have

$$\bar{s}_y(u, v) \leq B_v \quad (13)$$

Furthermore,

$$\bar{s}(u, v) = \frac{\sum_{i=1}^{n_r} \bar{s}_i(u, v)}{n_r} \quad (14)$$

Then, we have

$$\bar{s}(u, v) = \frac{\sum_{i=1}^x \bar{s}_i(u, v) + \sum_{j=n_r-x}^{n_r} \bar{s}_j(u, v)}{n_r} \leq \frac{\sum_{i=1}^x \bar{s}_i(u, v) + (n_r - x) * B_v}{n_r} \quad (15)$$

Based on Equation (15), if Equation 12 is satisfied, it implies that $\bar{s}(u, v)$ is less than $\bar{s}_{min}$. Therefore, the remaining $n_r - x$ trials can be safely skipped. $\square$

The complete pseudo-code of *HitSim-OPT* is illustrated in Algorithm 4. It first invokes the *revReach* algorithm (referenced as *CrashSim* [13]) to construct the reverse reachable tree of source u and return a matrix U (line 1). Subsequently, it invokes the *UBV* algorithm to compute the upper bound B_v for each vertex v (line 2).

Algorithm 4: *HitSim-OPT*

Input: Graph $G = (V, E)$, source $u \in V$, integer k , decay factor c , limited length l_{max} , the average absolute error ϵ , and the failure probability δ

Output: $Q = \{v, \bar{s}(u, v)\}$

```

1  $U \leftarrow \text{revReach}(G, u, l_{max}, c);$ 
2  $B \leftarrow \text{UBV}(G, u, l_{max}, U);$ 
3  $n_r \leftarrow \frac{3c}{\epsilon^2} \log \frac{n}{\delta};$ 
4 Initialize priority queue  $Q$ ;
5 for each  $v \in V$  do
6   if  $Q.size = k$  and  $B_v < Q.top$  then
7     continue;
8   for  $x = 1, \dots, n_r$  do
9     Generate a  $\sqrt{c}$ -walk from  $v$ ,  $W(v) = \{v_1, v_2, \dots, v_l\}$ , where  $v_l = v$ ;
10    limit  $l \leq l_{max}$ ;
11    for  $i = 1, \dots, l$  do
12       $\sum_{i=1}^x \bar{s}(u, v) + = U_i(W_i(v));$ 
13    if  $Q.size = k$  and  $\frac{\sum_{i=1}^x \bar{s}(u, v) + B_v * (n_r - x)}{n_r} < Q.top$  then
14      break;
15     $\bar{s}(u, v) = \frac{\bar{s}(u, v)}{n_r};$ 
16    Maintain priority queue  $Q$  of size  $k$  and push  $(v, \bar{s}(u, v))$  into  $Q$ ;
17 return  $Q$ ;
```

In step 3, it maintains a priority queue Q of size k to store the top- k results (lines 4 to 22). Specifically, it determines whether to prune v to avoid generating $\sqrt{c}$ -walks for v by comparing B_v with the top element $\bar{s}_{min}$ in Q . If $B_v \leq \bar{s}_{min}$, v can be pruned (lines 6 to 8). If v is not pruned, the algorithm generates a $\sqrt{c}$ -walk for n_r trials to compute $\bar{s}(u, v)$ for v (lines 9 to 20). During the n_r trials, if the current $\sum_{i=1}^x \bar{s}(u, v)$ satisfies Lemma 3, the generation process breaks (lines 15 to 17).

5.3. Analysis

The time and space complexity of *HitSim-OPT* can be analyzed in steps 1, 2, and 3.

- Step 1 (computing the reverse reachable tree U). It is the same as step 1 of *HitSim*.
- Step 2 (computing the upper bound of SimRank for each vertex). Computing an array of l_{max} for all vertices is essential, resulting in a time complexity of $O(m \cdot l_{max})$ and a space complexity of $O(n \cdot l_{max})$.
- Step 3 (maintaining the top- k results). It is the same as step 3 of *HitSim*.

Based on the above analysis, the total time complexity of *HitSim-OPT* is $O(m + m \cdot l_{max} + n \cdot l_{max} \cdot \frac{3c}{\epsilon^2} \log \frac{n}{\delta} + n \log k)$, and space complexity is $O(2n + k + n \cdot l_{max})$.

6. Experiments

6.1. Experimental Setup

We conduct extensive experiments to evaluate the performance of our algorithms. The algorithms evaluated in our experiments are summarized as follows:

- MC [14];
- CrashSim [13];
- HitSim: Algorithm 2;
- HitSim-OPT: Algorithm 4.

We conduct all the experiments on an Ubuntu machine with Intel(R) Core(TM) i7-12700 CPU 2.10 GHz and 64 G memory. All of the algorithms are implemented in C++.

Datasets. We used six datasets to evaluate the performance of all the algorithms. The dataset soc-Epinions (<http://konect.cc/networks/> (accessed on 20 September 2022)) represents the trust network from the online social network Epinions; emai-EuAll (<http://snap.stanford.edu/data/> (accessed on 1 December 2021)) is the email communication network of a large European institution; amazon (<http://konect.cc/networks/> (accessed on 20 September 2022)) is the network of items on Amazon; wiki-topcats (<http://snap.stanford.edu/data/> (accessed on 1 December 2021)) is a web graph of Wikipedia hyperlinks; soc-LiveJournal (<http://snap.stanford.edu/data/> (accessed on 1 December 2021)) is a free online community with almost 10 million members; wikipedia-link-en (<http://konect.cc/networks/> (accessed on 20 September 2022)) consists of the wikilinks of Wikipedia in the English language. These datasets represent relevant knowledge in various fields. Detailed statistics of these datasets are summarized in Table 3, where $|V|$, $|E|$, and $\bar{d}$ denote the number of vertices, the number of edges, and the average of degrees.

Table 3. Statistics of datasets.

Dataset	$ V $	$ E $	$\bar{d}$
soc-Epinions(SE)	75,879	508,837	6.71
emai-EuAll(EF)	265,214	420,045	1.58
amazon(AZ)	403,394	3,387,388	8.40
wiki-topcats(WC)	1,791,489	28,511,087	15.92
soc-LiveJournal(SL)	4,847,571	68,993,773	14.23
wikipedia-link-en(WE)	13,593,032	437,217,424	32.16

Parameters. Consistent with previous studies [3,7,13,17], we set the decay factor c to 0.6 and l_{max} to 5. To evaluate the performance under different error guarantees, we vary the parameter ε to achieve overall absolute error guarantees of 0.1, 0.05, 0.025, and 0.0125, while maintaining a failure probability of $\delta = 0.01$.

Metrics. For evaluating the quality of results for each single-source and top-k query from the source vertex u , we employ two metrics: $AE@k$ and $Precision@k$. The $AE@k$ metric, representing the average absolute error, is computed as $AE@k = \frac{1}{k} \sum_{i=1}^k |s(u, v) - \bar{s}(u, v)|$, where $s(u, v)$ is the true SimRank score between vertices u and v , and $\bar{s}(u, v)$ is the SimRank estimation. The $Precision@k$ metric measures the proportion of correctly identified results among the top-k results. It is computed as $Precision@k = |V_k \cap V'_k| / k$, where V_k represents the list of top-k vertices returned by the algorithm being evaluated, and V'_k represents the ground-truth top-k results.

6.2. Performance of Algorithms

Absolute error in querying (AE@50). In this experiment, we set k to 50 and vary ε from 0.1 to 0.05, 0.025, and 0.0125. Figure 6 illustrates the trade-offs between $AE@50$ and the query time of each algorithm. We can observe that as ε decreases the value of $AE@50$ decreases. However, the query time increases for each algorithm. The curves in the graph show a near-linear relationship, suggesting that the algorithms achieve a faster query time when ε is large.

Additionally, when comparing algorithms with the same parameters, both *HitSim* and *HitSim-OPT* demonstrate similar average *AE* values compared to *CrashSim*, while requiring less query time. This can be attributed to the fact that *CrashSim* computes SimRank for all vertices to obtain the top- k results, whereas *HitSim* only computes SimRank for vertices within sets whose upper bounds of SimRank exceed $\bar{s}_{min}$, and *HitSim-OPT* computes SimRank only for vertices with larger upper bounds.

It is worth noting that the results of *MC* may be unstable when n_r (the number of iterations) is not sufficiently large, as *MC* requires generating a large number of $\sqrt{c}$ -walks for each vertex v to determine if it meets source u .

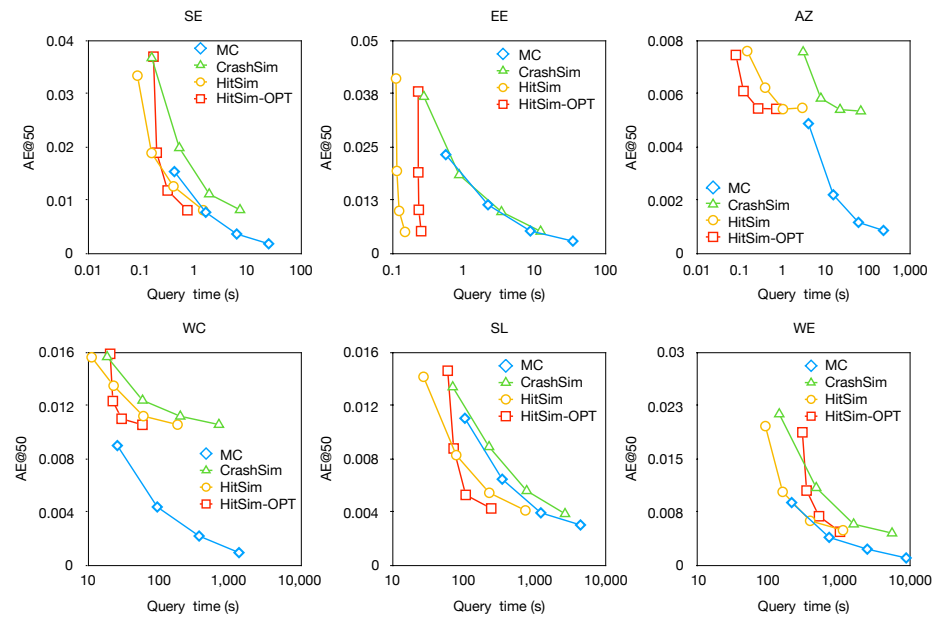


Figure 6. The performance of *AE@50*.

Precision in querying (*Precision@50*). In this experiment, we fix the value of k to 50 and vary the parameter ε from 0.1 to 0.05, 0.025, and 0.0125. The trade-off between *Precision@50* and query time is illustrated in Figure 7. We observe that, as ε decreases, both *Precision@50* and the query time increase. When $\varepsilon = 0.0125$, all algorithms achieve a precision of nearly 1. The curves in the graph show a near-linear relationship, indicating that algorithms achieve a faster query time when ε is large.

Moreover, when comparing algorithms with the same parameters, both *HitSim* and *HitSim-OPT* demonstrate similar precision values compared to *CrashSim*, while requiring less query time. The reason is the same as the last experiment. The results of *MC* may be unstable when n_r (the number of iterations) is not sufficiently large.

Running time. In this experiment, we compare the running time of querying the top-250 results using the same parameters for each algorithm. Figure 8 shows the running time of each algorithm. The results demonstrate that on all datasets *HitSim* and *HitSim-OPT* exhibit faster performance compared to *MC* and *CrashSim*. Specifically, *HitSim* is approximately 7 times faster than *MC* on average and 3 times faster than *CrashSim*. Similarly, *HitSim-OPT* is approximately 30 times faster than *MC* on average and 11 times faster than *CrashSim*.

The main reason is that *CrashSim* processes all vertices, whereas *HitSim* only processes the vertices within sets whose upper bounds of SimRank exceed $\bar{s}_{min}$, and *HitSim-OPT* processes only vertices with larger upper bounds. As mentioned in the previous analysis, the running time of generating $\sqrt{c}$ -walks for vertices dominates the overall performance. Therefore, processing fewer vertices results in less time cost for *HitSim* and *HitSim-OPT* compared to *MC* and *CrashSim*.

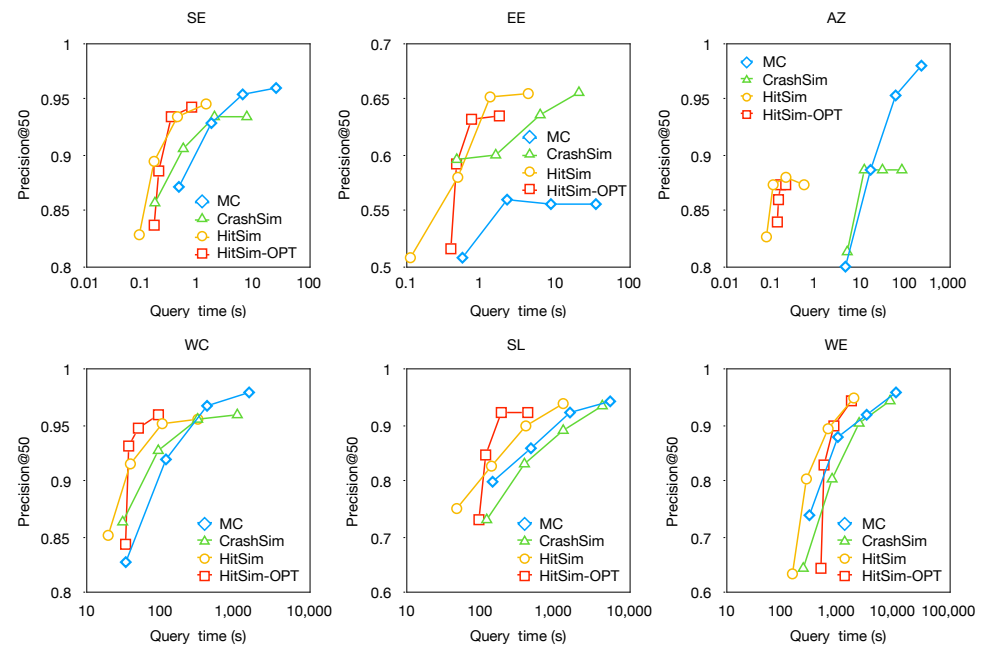


Figure 7. The performance of *Precision@50*.

To validate the correctness, we also test the number of vertices processed by each algorithm, as shown in Table 4. The results confirm that *MC* and *CrashSim* process all the vertices, while *HitSim* and *HitSim-OPT* process significantly fewer vertices due to their batch-pruning methods. Additionally, the number of vertices processed by *MC* or *CrashSim* is approximately 4 times more than *HitSim* on average and 16 times more than *HitSim-OPT*. The results demonstrate that *HitSim* and *HitSim-OPT* can efficiently return the single-source and top-k query.

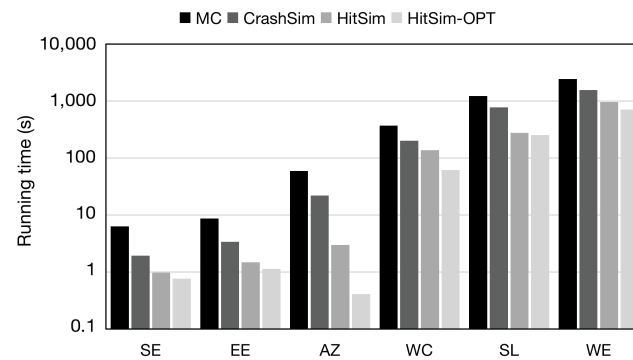


Figure 8. Running time of top-250 results query under the same parameters.

Table 4. The number of vertices processed. The column *total* represents the total number of vertices processed by *HitSim-OPT*, while the column *n_r iteration* represents the number of vertices that completed all n_r iterations of $\sqrt{c}$ -walks.

Dataset	MC	CrashSim	HitSim	HitSim-OPT	
				Total	n_r Iteration
soc-Epinions (SE)	75,879	75,879	41,659	21,847	833
emai-EuAll (EE)	265,214	265,214	32,504	18,919	695
amazon (AZ)	403,394	403,394	59,484	5434	423
wiki-topcats (WC)	1,791,489	1,791,489	1,290,877	1,057,051	872
soc-LiveJournal (SL)	4,847,571	4,847,571	1,863,063	3,841,961	597
wikipedia-link-en (WE)	13,593,032	13,593,032	2,683,698	5,147,351	849

Performance of *HitSim* and *HitSim-OPT*. In this experiment, we compare the performance of the preprocessing and query steps of *HitSim* and *HitSim-OPT* under the parameters $k = 250$ and $\varepsilon = 0.025$. As described in Section 4, *HitSim* and *HitSim-OPT* return the top-k results in three steps. We consider the first two steps as preprocessing and the third step as the query phase.

Figure 9 illustrates the performance comparison between the preprocessing and query steps of *HitSim* and *HitSim-OPT* on all datasets. The results indicate that *HitSim-OPT* outperforms *HitSim*. Notably, the preprocessing step of *HitSim-OPT* is slower than that of *HitSim*. This is attributed to the fact that *HitSim-OPT* requires constructing dp arrays of length l_{max} for all vertices on the reverse reachable tree to obtain their SimRank upper bounds, whereas *HitSim* only requires visiting all vertices on the reverse reachable tree. The query performance of *HitSim-OPT* demonstrates a significant improvement over *HitSim*. This enhancement can be attributed to the finer-grained pruning rules of *HitSim-OPT*, allowing it to achieve efficiency by processing fewer vertices. However, in large-scale dense graphs such as SL and WE, many vertices' upper bounds may be quite similar, resulting in a large number of vertices meeting the processing condition (upper bound $B > \bar{s}_{min}$). Nonetheless, during the processing phase, *HitSim-OPT* leverages its efficient pruning rule (Lemma 3) to early-terminate $\sqrt{c}$ -walks, which is the most time-consuming operation. As a result, many vertices do not need to complete all the n_r iterations of $\sqrt{c}$ -walks, thereby reducing the computational overhead. For instance, on SL, only 597 vertices complete all n_r iterations of $\sqrt{c}$ -walks, while $3,841,961 - 597 = 3,841,364$ vertices execute 1 to n_r iterations of $\sqrt{c}$ -walks. Similarly, on WE, only 849 vertices complete all n_r iterations of $\sqrt{c}$ -walks, while $5,147,351 - 849 = 5,146,502$ vertices execute 1 to $n_r - 1$ iterations of $\sqrt{c}$ -walks.

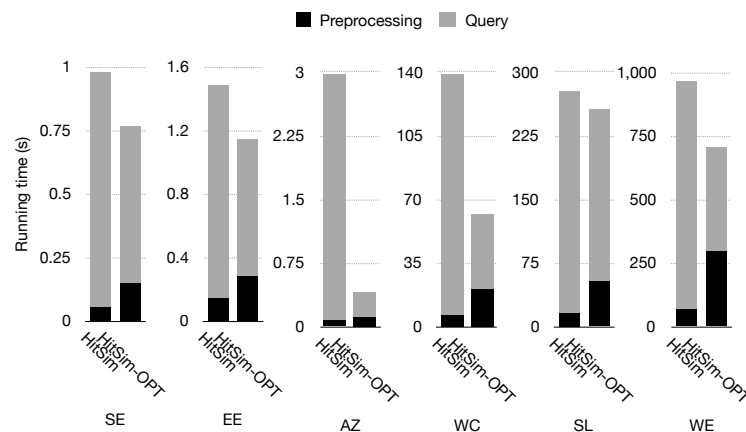


Figure 9. Preprocessing and query of *HitSim* and *HitSim-OPT*.

Scalability. In this experiment, we test the scalability of *HitSim* and *HitSim-OPT*. We generate four subgraphs by randomly sampling 20–80% of the edges from the WE dataset. We test the running time of *HitSim* and *HitSim-OPT* on WE with fixed parameters $c = 0.6$, $\varepsilon = 0.025$, and $\delta = 0.1$. Figure 10 shows the results of the scalability test. We can see that both *HitSim* and *HitSim-OPT* show near-scalability as the number of edges $|E|$ increases from 20% to 100%, and as the value of k ranges from 20 to 2000. Additionally, we note that as k increases, the running time of both *HitSim* and *HitSim-OPT* also increases. This is because the batch-pruning strategy is triggered later as k increases.

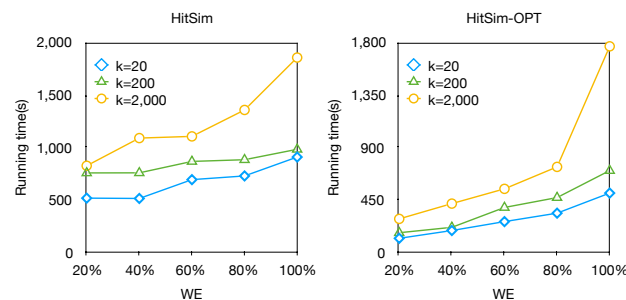


Figure 10. Scalability tests of *HitSim* and *HitSim-OPT* by varying $|E|$ from 20% to 100% and k from 20 to 2000 on WE.

7. Conclusions

In this paper, we study the single-source and top- k SimRank query problem. We first propose an efficient algorithm called *HitSim* that utilizes a branch-and-bound strategy. *HitSim* partitions vertices into distinct sets based on their shortest-meeting lengths to the source vertex. Subsequently, it computes the upper bound of SimRank for each set. By batch-pruning vertices within the same set whose upper bound is less than the minimum SimRank of the current results, *HitSim* significantly enhances computational efficiency. Furthermore, we propose an optimized algorithm called *HitSim-OPT*, which employs a fine-grained pruning strategy. *HitSim-OPT* computes the upper bound of SimRank for each vertex, thereby improving pruning efficiency. Our experimental results on six real-world datasets demonstrate that, while maintaining comparable precision and absolute error to *CrashSim*, *HitSim* achieves an average speedup of 3 times compared to *CrashSim*, and *HitSim-OPT* achieves an average speedup of 11 times.

Author Contributions: Conceptualization, J.B. and J.Z.; methodology, J.B. and M.M.; software, M.M. and S.C.; validation, J.B. and J.Z.; formal analysis, M.D. and M.M.; investigation, J.B., M.M. and S.C.; data curation, J.B.; writing—original draft preparation, M.M. and J.B.; writing—review and editing, J.B. and J.Z.; supervision, J.Z., M.D. and Z.C.; funding acquisition, J.Z. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by grants from the Natural Science Foundation of China (No.: 62372101, 61873337, 62272097).

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The data that support the findings of this study are available on request from the first author upon reasonable request.

Conflicts of Interest: The authors declare no conflicts of interest.

References

- Jin, R.; Lee, V.E.; Hong, H. Axiomatic ranking of network role similarity. In Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Diego, CA, USA, 21–24 August 2011; pp. 922–930. [\[CrossRef\]](#)
- Liben-Nowell, D.; Kleinberg, J.M. The link-prediction problem for social networks. *J. Assoc. Inf. Sci. Technol.* **2007**, *58*, 1019–1031. [\[CrossRef\]](#)
- Antonellis, I.; Garcia-Molina, H.; Chang, C. Simrank++: Query rewriting through link analysis of the click graph. *Proc. VLDB Endow.* **2008**, *1*, 408–421. [\[CrossRef\]](#)
- Spirin, N.; Han, J. Survey on web spam detection: Principles and algorithms. *SIGKDD Explor.* **2011**, *13*, 50–64. [\[CrossRef\]](#)
- Rothe, S.; Schütze, H. CoSimRank: A Flexible & Efficient Graph-Theoretic Similarity Measure. In Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics, ACL 2014, Baltimore, MD, USA, 22–27 June 2014; Long Papers; Volume 1, pp. 1392–1402. [\[CrossRef\]](#)
- Jeh, G.; Widom, J. SimRank: A measure of structural-context similarity. In Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Edmonton, AB, Canada, 23–26 July 2002; pp. 538–543. [\[CrossRef\]](#)
- Lizorkin, D.; Velikhov, P.E.; Grinev, M.N.; Turdakov, D. Accuracy estimate and optimization techniques for SimRank computation. *VLDB J.* **2010**, *19*, 45–66. [\[CrossRef\]](#)

8. Tao, W.; Yu, M.; Li, G. Efficient Top-K SimRank-based Similarity Join. *Proc. VLDB Endow.* **2014**, *8*, 317–328. [[CrossRef](#)]
9. Fogaras, D.; Rácz, B. Scaling link-based similarity search. In Proceedings of the 14th International Conference on World Wide Web, WWW 2005, Chiba, Japan, 10–14 May 2005; pp. 641–650. [[CrossRef](#)]
10. Lee, P.; Lakshmanan, L.V.S.; Yu, J.X. On Top-k Structural Similarity Search. In Proceedings of the IEEE 28th International Conference on Data Engineering (ICDE 2012), Washington, DC, USA, 1–5 April 2012; pp. 774–785. [[CrossRef](#)]
11. Tian, B.; Xiao, X. SLING: A Near-Optimal Index Structure for SimRank. In Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, 26 June–1 July 2016; pp. 1859–1874. [[CrossRef](#)]
12. Liu, Y.; Zheng, B.; He, X.; Wei, Z.; Xiao, X.; Zheng, K.; Lu, J. ProbeSim: Scalable Single-Source and Top-k SimRank Computations on Dynamic Graphs. *Proc. VLDB Endow.* **2017**, *11*, 14–26. [[CrossRef](#)]
13. Li, M.; Choudhury, F.M.; Borovica-Gajic, R.; Wang, Z.; Xin, J.; Li, J. CrashSim: An Efficient Algorithm for Computing SimRank over Static and Temporal Graphs. In Proceedings of the 36th IEEE International Conference on Data Engineering, ICDE 2020, Dallas, TX, USA, 20–24 April 2020; pp. 1141–1152. [[CrossRef](#)]
14. Liu, Y.; Zou, L.; Ge, Q.; Wei, Z. SimTab: Accuracy-Guaranteed SimRank Queries through Tighter Confidence Bounds and Multi-Armed Bandits. *Proc. VLDB Endow.* **2020**, *13*, 2202–2214. [[CrossRef](#)]
15. Shao, Y.; Cui, B.; Chen, L.; Liu, M.; Xie, X. An Efficient Similarity Search Framework for SimRank over Large Dynamic Graphs. *Proc. VLDB Endow.* **2015**, *8*, 838–849. [[CrossRef](#)]
16. Jiang, M.; Fu, A.W.; Wong, R.C.; Wang, K. READS: A Random Walk Approach for Efficient and Accurate Dynamic SimRank. *Proc. VLDB Endow.* **2017**, *10*, 937–948. [[CrossRef](#)]
17. Wei, Z.; He, X.; Xiao, X.; Wang, S.; Liu, Y.; Du, X.; Wen, J. PRSim: Sublinear Time SimRank Computation on Large Power-Law Graphs. In Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, 30 June–5 July 2019; pp. 1042–1059. [[CrossRef](#)]

Disclaimer/Publisher’s Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.