Scalable Single-source SimRank Computation for Large Graphs

Xingkun Gao, Nianyuan Bao, Jie Liu, Jie Tang, and Gangshan Wu
State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, P.R. China
{MG1533012, DG1533001, MG1533026}@smail.nju.edu.cn, {tangjie, gswu}@nju.edu.cn

Abstract—**SimRank is an effective similarity measure between vertices in a graph, which has become a fundamental technique in graph analytics. Despite its popularity, computation of SimRank is often costly in both space and time, especially with the ever growing scale of graph data nowadays. In this paper, we focus on the computation of Single-Source SimRank: given a query vertex, return the similarities between this vertex and any other vertices in the graph. The traditional centralized SimRank algorithms are not efficient for this problem. To fully utilize the computing power of modern distributed systems, we propose sssSimRank, an efficient distributed algorithm based on the random walk model. Our algorithm achieves scalability via minimizing the total number, the space cost, and the matching time of random walks. We implement our approach on the popular distributed processing platform Spark. Experimental results demonstrate the effectiveness, efficiency and scalability of our method.**

Index Terms—**graph analytics; big data; SimRank; random walk; distributed algorithm; Spark;**

I. Introduction

Graph can be used to model complicated relations between entities from various domains such as the Internet, social networks and the Internet of Things. With the advent of the Big Data, the scales of graphs are growing in rapid speed. While more and more advanced techniques are being developed to fully exploit the hidden patterns of graph data, the huge scale poses great challenges in graph mining.

One of the fundamental tasks in graph mining is to evaluate similarities between vertices. It plays an important role in many applications, including recommender systems [1], entity resolution [2], information retrieval [3], and so on. Many similarity measures have been proposed, e.g., Jaccard similarity [4], cosine similarity [5] and Dice similarity [6], all of which are motivated by the intuition that *two vertices are more similar if they share more neighbors*. However, these measures cannot capture the topology of the whole graph, for instance, they fail to assess similarities between vertices with no common neighbors. To address this problem, SimRank [7], a new similarity measure based on the intuition that *two vertices are similar if their in-neighbors are similar too* was proposed. Based on random surfer model, SimRank owns a theoretical foundation stemming from PageRank [8], where a webpage is important if the webpages pointing to it are important too. Studies show that for link-based similarity measures, SimRank outperforms other related measures.

However, the recursive nature in the definition of SimRank makes it very difficult to be computed both effectively and efficiently. The time complexity of naive iterative algorithm is $O(kn^2d^2)$, where n is the number of vertices, d is the average in-degree of vertices, and k is the number of iterations. Some studies give some optimization techniques to speed up the computation process [9][10]. These techniques are specifically designed to optimize the running time of computing similarity scores of all-pair vertices, i.e., all n^2 pair-wise similarities. However, in the situation where only the similarity between a given vertex u and any other vertex in the graph is required, all-pair similarities turn out to be cumbersome for the following reasons: 1) to compute SimRank similarity of a single pair (u, v), all-pair algorithms need to compute all the similarity scores between in-neighbors of vertex u and in-neighbors of vertex v, and this dependency goes on and on due to the recursive nature of SimRank; 2) all-pair SimRank is inadaptable when dealing with incremental graphs. In many real-world applications the topology of the graph changes over time, any addition or removal of vertices or edges may force a costly recomputation of similarities of all vertex pairs.

Another problem is that due to the ever increasing scale of real-world graphs, solutions designed for a single machine are limited by its restricted computing power and main memory capacity. For a typical web graph with millions of vertices and billions of edges, it is difficult to even load the whole graph into the main memory, thus studying scalable algorithms that fully exploit the computing power of modern distributed systems has practical significance.

In this paper, we address these problems and propose effective and efficient method to compute the single-source SimRank in a distributed manner. Our main contributions are summarized as follows.

- We analyze the inefficiency of directly decomposing the computation of single-source SimRank into single-pair SimRank problem [11]. From the analysis we educe optimization techniques to speed up the computing process, including reducing the total number of walks generated, compressing the data representation of walks and speeding up matching of walks by dynamic programming.
- Based on these optimizations we propose our scalable single-source SimRank computation algorithm, or sssSimRank. We also give a thorough analysis of it.
- We manage to implement our algorithm on the popular data processing system, Spark. We evaluate our method on real datasets. Experimental results show that our implementation is effective, efficient and parallelizable.

The rest of paper is organized as follows. We review related work in Section II and preliminaries in Section III. We show the details of our algorithm as well as some optimization techniques in Section IV. We describe our implementation on top of Spark in Section V, and present experimental results in Section VI. Section VII concludes this paper.

II. Related Work

The efficiency of computing SimRank is an obstacle to prevent its applicability on large datasets. Therefore, many approaches have been proposed to speed up SimRank computation. These algorithms can be classified into the following categories. To ease presentation, the number of vertices and edges of the input directed graph are denoted by n and m. We also denote by d the average in-degree of the graph, and by k the iterations required by the algorithm.

Matrix-multiplication-based Algorithms. Jeh and Widom [7] proposed the first iterative algorithm to compute SimRank similarities by matrix computations. It computes the SimRank similarities between all pairs of vertices in $O(kn^2d^2)$ time. [9] improved the computational complexity of the iterative algorithm to $O(kn^2d)$ by pruning, partial sum memorization, and local access reduction techniques, and [10] speeded up the algorithm by fast matrix multiplication. [12] further improved the time complexity to $O(kd'n^2)$ time, where $d' < d$. In [13], a non-iterative SimRank matrix formula has been established based on the Kronecker product and singular vector decomposition (SVD). It first pre-computes some auxiliary matrices offline in $O(r^4n^2)$ time and then retrieves the SimRank between a given vertex and all other vertices in $O(r^4n)$ time, where r is the rank of the adjacency matrix of the graph. [14] employed GPU to speed up SimRank computation. All the above algorithms require $O(n^2)$ space and come at great cost — they need to perform expensive matrix operations and maintain all n^2 similarities simultaneously, which makes it impossible to query individual single-pair or single-source SimRank without querying the rest. For single-pair SimRank computation problem, [11] gave an algorithm with time complexity $O(kd^2 \cdot \min\{n^2, d^k\})$. [15] further improved the running time to $O(km^2-m)$ by utilizing position probabilities. Although these algorithms both use random walks model to help formulate their final solutions, their actual computation are still based on matrix operations. As a result, they still require $O(n^2)$ space.

Random-walk-based Algorithms. The SimRank similarity between two vertices u and v can be represented in the form of the expectation of probabilities that two random walks starting form u and v, respectively, meet at the same vertex for the first time. [16] gave the first random walk based algorithm by first building as an index of size $O(nN)$ the fingerprints of N random walks and then querying the single-pair SimRank similarities based on this index. [17] reinterpreted the SimRank computation via linearization and then developed sampling techniques based on random walks to compute single pair SimRank similarity. [18] studied top-k most similar vertices measured by SimRank of single vertex, where k is typically very small. It transformed the single-source problem on graph G to finding the authorities on the product graph $G \times G$. The above mentioned algorithms are designed for a single machine, and are therefore far from acceptable for large problems due to restricted computing power and limited storage capacity.

Distributed SimRank Algorithms. Cal et al. [19] proposed a MapReduce method to compute all-pair SimRank. The amount of data transferred from mappers to reducers in each iteration is $O(d^2n^2)$, so the overall communication cost is $O(kd^2n^2)$, which is inefficient for single-source SimRank.

III. Preliminaries

In this section we review some preliminary knowledge, including the SimRank similarity measure, random walks on graph, and the Spark distributed data processing platform.

We first list some notations used throughout this paper. A *graph* is a pair (V, E), where V is the set of vertices and $E \subseteq V \times V$ is the set of edges. We denote by n and m the number of vertices and number of edges in the graph respectively. In this paper, we consider *directed graphs*. A vertex u is said to be an *in-neighbor* (or an *out-neighbor*) of a vertex v if (u, v)(or (v, u)) is an edge in G. The sets of in-neighbors and out-neighbors of a vertex u are denoted by $I(u) = \{v : (v, u) \in E\}$ and $O(u) = \{v : (u, v) \in E\}$ respectively. The average *in-degree* (also *out-degree*) of the vertices in the graph is denoted by d. The SimRank similarity between vertices u and v is denoted by $s(u, v)$, and the $n \times n$ SimRank similarity matrix of the whole graph is denoted by S with $S_{uv} = s(u, v)$. The single-source SimRank of vertex u is denoted by $s(u, *)$.

A. SimRank

SimRank [7] is a structural-context similarity measure for vertices in a directed graph which is designed based on the intuition that *two vertices are similar if their in-neighbors are similar too*. Mathematically, the similarity of vertices u and v is defined as:

$$s(u,v) = \begin{cases} 1, & u = v; \\ \dfrac{c}{|I(u)||I(v)|} \displaystyle\sum_{u' \in I(u), v' \in I(v)} s(u', v'), & u \neq v. \end{cases} \tag{1}$$

where $0 < c < 1$ is called the decay factor. [7] proved that a unique solution to Eq. (1) always exists and can be computed iteratively. Suppose S^k is the computed SimRank matrix after kth iteration, given that S^0 is initialized with $S_{uv} = 1$ for $u = v$ and $S_{uv} = 0$ otherwise. Then to compute S^{k+1}, we use the following recursion:

$$S^{k+1}_{uv} = \begin{cases} 1, & u = v; \\ \dfrac{c}{|I(u)||I(v)|} \displaystyle\sum_{u' \in I(u), v' \in I(v)} S^k_{u'v'}, & u \neq v. \end{cases} \tag{2}$$

It has been proven that $\lim_{k \to \infty} S^k_{uv} = s(u, v)$ in [7]. The naive iterative matrix-multiplication-based SimRank algorithm computes the similarities by iterating over all pairs of vertices, thus each iteration requires $O(n^2)$ space and $O(n^2d^2)$ time.

B. Random Walk Model

Another generalization of SimRank is based on the *Random Walk Model*. A *walk* on G is defined as a sequence of vertices $W = v_0 v_1 v_2 \dots v_l$ such that (v_i, v_{i+1}) is an edge in G for $0 \le i \le l-1$. A walk on graph is called *random walk* if it satisfies *Markov's property*:

$$\begin{aligned} &Pr(X_i = v_i | X_0 = v_0, \dots, X_{i-1} = v_{i-1}) \\ &\quad = Pr(X_i = v_i | X_{i-1} = v_{i-1}) \end{aligned} \tag{3}$$

for all $i \ge 1$ and all $v_0, v_1, \dots, v_i \in V$, where X_i is the random variable of the vertex the walk will be on at time i. For any $u, v \in V$, $Pr(X_i = v | X_{i-1} = u)$ is the *transition probability* that the random walk will make a transition onto vertex v in the next step if it is on vertex u at time $i-1$.

In the random walk model interpretation of SimRank, a random surfer surfs by following the edges backwards, i.e., moves to one of the in-neighbors of the vertex it is currently on in each step. The transition probability is define as:

$$Pr(X_i = v_i | X_{i-1} = v_{i-1}) = \begin{cases} \frac{1}{|I(v_{i-1})|}, & (v_i, v_{i-1}) \in E; \\ 0, & otherwise. \end{cases} \tag{4}$$

Accordingly, the *walk probability* of W is formulated by:

$$Pr(W) = \prod_{i=1}^{l} Pr(X_i = v_i | X_{i-1} = v_{i-1}) \tag{5}$$

If two random surfers start from vertex u and v respectively at the same time, walk stepwise, meet for the first time at an arbitrary vertex x and then stop at x, we call the corresponding two walks W_u and W_v they produced a pair of *meeting walks* or *matching walks*. The length of a pair of meeting walks is the length when they stopped. We also define their *meeting probability* as:

$$Pr\big((W_u, W_v)\big) = Pr(W_u) Pr(W_v) \tag{6}$$

[7] revealed that $s(u, v)$ can be interpreted as the expectation of meeting probabilities of random walks as:

$$s(u, v) = \sum_{W_u, W_v} c^l Pr\big((W_u, W_v)\big) \tag{7}$$

where (W_u, W_v) is an arbitrary pair of meeting walks starting from u and v respectively, l is their length, and c is the decay factor. Note that here l could be arbitrarily large.

C. Spark

In principle, our algorithm can be implemented on any general-purpose distributed data processing platforms. We choose Spark [20] simply for its generality, efficiency and user friendliness. One of the core concepts in Spark is the in-memory storage abstraction known as Resilient Distributed Datasets (RDDs) [20], which fully utilize the memory of each computing node in the cluster. An RDD can be seen as a collection of records, where two types of operations over RDDs are available: *transformations*, which create new RDD by applying some transformations on old RDDs; and *actions*, which return some global statistics or computed results of RDDs to the driver program. All transformations are lazy, in the sense that the transformations to be applied would not take actual effect until an action is triggered. RDDs are fault-tolerant since Spark automatically keeps the lineage information, i.e., the transformation history of the data for fast recovery from data losses. A job submitted to Spark is divided into several stages according to the dependency between the series of transformations applied to the RDD, and a stage is further divided into multiple computing units which are executed in parallel.

Transformations used in this paper include `map`, `flatMap`, `filter`, `reduceByKey`, `leftOuterJoin` and `join`. In detail, `map` applies a one-to-one mapping of the records of the input RDD to form a new one (similar to the `map` operation in MapReduce), and `flatMap` applies a one-to-many mapping in a similar way; `filter` as the name suggests, produces a new RDD composing the records satisfying the predicate specified by user; `reduceByKey` aggregates the records with the same key using the user-provided reduce function (similar to the `reduce` operation in MapReduce); `join` performs a join over two RDDs, and so does `leftOuterJoin`. Among the above mentioned transformations, `reduceBykey`, `leftOuterJoin` and `join` will shuffle data between different machines. Other transformations perform their computations locally. The only action we use in this paper is `collect`, which returns all the records distributed over the cluster to the driver program in master node. Spark also provides `broadcast` interface to allow programmers to keep a read-only variable cached on each machine rather than shipping a copy of it with tasks.

IV. Scalable Single-source SimRank

Given the definition of SimRank and its random walk model interpretation, the way to compute single-source SimRank $s(u, *)$ is straightforward. Intuitively the computation of $s(u, *)$ can be decomposed into subproblems of computing $s(u, v)$ for all $v \in G$. To compute $s(u, v)$, we first find all pairs of meeting walks starting from u and v respectively, then we aggregate their meeting probabilities according to Eq. (7). Note that enumeration of meeting walks arbitrarily long (till infinity) is impossible, therefore in practice only meeting walks of length up to a limit are considered.

With this line of reasoning, we now describe the details of our single-source SimRank algorithm, or sssSimRank. To ease presentation, we let u be the query vertex, i.e., the 'source' in 'single-source'. We also assume that the maximum length of random walks is set as k. Random walks starting from u are called *master walks*, and other walks are called *slave walks*.

A. A Naive Method

Our work is motivated by the single-pair SimRank algorithm spSimRank proposed in [11]. The core idea of spSimRank is that to compute the SimRank similarity between vertex pair (u, v), two random surfers starting from vertex u and vertex v respectively move backwards by following their in-edges. A walk will split into $|I(t)|$ different walks after passing a vertex

Algorithm 1 Naive Single-source SimRank

```
1:  procedure SINGLESOURCESIMRANK($G, u, k$)
2:      for $l = 1$ to $k$ do
3:          $W_u[l] \leftarrow$ all walks of length $l$ starting from $u$;
4:      for $v \in V(G)$ do
5:          $s(u, v) \leftarrow$ SPSIMRANK($G, W_u[], v, k$);
6:      return $s(u, *)$.
7:  procedure SPSIMRANK($G, W_u[], v, k$)
8:      $s(u, v) \leftarrow 0$;
9:      for $l = 1$ to $k$ do
10:         $W_v[l] \leftarrow$ all walks of length $l$ starting from $v$;
11:         $s_l(u, v) \leftarrow 0$;
12:         for $w_u$ in $W_u[l]$ do
13:             for $w_v$ in $W_v[l]$ do
14:                 if $w_v$ and $w_u$ first meet at index $l$ then
15:                     add $score(w_u, w_v)$ to $s_l(u, v)$;
16:                                                 ▷ According to Eq. (7)
17:         add $s_l(u, v)$ to $s(u, v)$;
18:     return $s(u, v)$.
```

t. Therefore after k moves, a total of $O(d^k)$ various walks of length up to k existing in the graph topology are generated in a brute force manner. A data structure called Path-Tree is used to compress all the random walks to save space. Then all the master walks and slave walks in the resulting two path trees are matched to select the meeting walks. Finally $s(u, v)$ is computed based on their meeting probabilities. Compared to all-pair SimRank, in spSimRank similarity scores of other unrelated vertex pairs do not need to be computed. As a result, the computational cost of this algorithm does not increase if the underneath graph becomes large. Intuitively, we can invoke spSimRank for all vertex pairs $(u, *)$ to get the single-source SimRank. We call it Naive Single-source SimRank algorithm as listed in Algorithm 1.

The Naive Single-source SimRank algorithm is inefficient for the following reasons: 1) A total of $O(d^k)$ walks are generated for each v, but the majority of them cannot match a master walk at all; 2) Although the data structure Path-Tree is used to save memory, the space cost is still high. This will incur high network communication overhead in distributed environments since we need to exchange lots of data between different computing nodes; and 3) In the matching process of the algorithm, walks of the same length are compared in a brute force way, which will degrade the overall performance. We address these problems and try to improve the efficiency in the following aspects: fewer total number of walks, more compact representation of walks, and faster matching process.

B. Fewer Walks

In Naive Single-source SimRank, a total of $O(nd^k)$ walks are generated. But only a fraction of them will ever meet with a master walk within k steps. If we could reduce the factor n to a smaller value C such that $C \ll n$, the quantity of walks will drastically decline. We now discuss how to achieve this.

$$\begin{array}{ll} u, w_1, w_2, w_3, \ldots, x, \ldots & x, \ldots, w_3, w_2, w_1, u, \ldots \\ v, w_1', w_2', w_3', \ldots, x, \ldots & x, \ldots, w_3', w_2', w_1', v, \ldots \end{array}$$

Fig. 1: The left are two walks starting from u and v respectively, first meeting at x; the right are reversed walks starting from x, passing u and v respectively.

We first give the definition of *reversed walk* as the reversed vertex sequence of the random walk defined in Section III. It is easy to see that there is a one-to-one correspondence between the original random walk and its reverse walk. Recall in the random walk model interpretation of SimRank, random walks are generated by following the in-edges of the G. Accordingly, a reverse walk can be generated in the similar way by following the out-edges. As a result, the transition probability in a reversed walk should be rewritten as:

$$Pr(X_i = v | X_{i-1} = v_{i-1}) = \begin{cases} \frac{1}{|I(v_i)|}, & (v_{i-1}, v_i) \in E; \\ 0, & otherwise. \end{cases} \tag{8}$$

From Eq. (5) we see that the probability of a random walk and its reversed walk are exactly the same. Given the definition of reverse walk, we have the following observation:

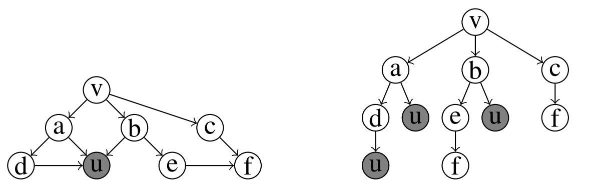
Observation 1. Suppose two random walks starting from u and v respectively, say W_u and W_v, met at vertex x after following in-edges for l steps. Then if we start from x and follow the out-edges in the graph to generate the set of all possible reversed walks of length l, reversed walks of W_u and W_v must be in it.

For example, in Fig. 1 the left two walks are W_u and W_v, and the right are the corresponding two reversed walks starting from x, passing u and v respectively. Observation 1 says that if the left two walks do exist in the graph topology, then their reversed walks on the right always exist.

Motivated by this, and because of the one-to-one correspondence between reversed walk and random walk, SimRank can be computed based on reversed walks. The benefit here is that although the time costs of generating random walks and reversed walks are almost equal, the number of vertices starting from which we need to generate reversed walks could be reduced greatly based on the following observation:

Observation 2. Let $W_u = uw_1w_2 \ldots w_l$ be a master walk. Then slave walks with length l could only meet with W_u at one of $\{w_1, w_2, \ldots, w_l\}$. In other words, If we denote by Nei the set of vertices reachable from u in k steps by following in-edges, it suffices to generate reversed walks starting from vertices in Nei rather than V to compute $s(u, *)$.

The correctness of this observation is evident. Based on the two observations, it is easy to see that now the total number of reversed walks needed is $O(|Nei|d^k)$ rather than $O(nd^k)$.



(a) $N_G(v, 4)$. (b) Path-tree of $N_G(v, 4)$.

Fig. 2: (a) A neighborhood of v. (b) The corresponding Path-Tree representation of the neighborhood.

In fact, $|Nei|$ is roughly $O(d^k)$, which is much smaller than n and is independent of the scale of the underneath graph. This guarantees efficiency when handling huge web graphs.

So now we give our new algorithm flow. We first generate master walks of length up to k and update the reachable vertex set Nei along the way. These master walks are carefully recorded and will be shared by other vertices in the matching process later. Then for each vertex in Nei, we take it as starting point and generate all slave reversed walks. Note that master walks are generated by following in-edges of the graph, while slave walks move along out-edges. In the following subsections, we discuss for each master walk, how we can find all its matching slave walks and finally compute $s(u, *)$ based on their meeting probabilities.

C. More Compact Representation of Walks

Although the number of reversed walks needed has been reduced, it still grows exponentially with the average in-degree d. Storing so many reversed walks is costly. Besides, there also exist redundancies as lots of walks share common subsequences. For instance, any prefix or suffix of the vertex sequence of a walk could also form another walk.

Our method addresses this problem and do as follows. For an arbitrary vertex v in the graph, we do not store reversed walks in any specially designed data structure at all. Instead, starting from v we let a surfer walk k steps by following the out-edges of the graph and collect a portion of the input graph — a *neighborhood* of v denoted by $N_G(v, k)$. Formally, $N_G(v, k)$ is an induced subgraph formed by all the vertices that are up to k steps away from v. For example, Fig. 2(a) shows a neighborhood of vertex v. Its corresponding Path-Tree representation is shown in Fig. 2(b). The vertex in gray is the query vertex u. Note that $N_G(v, k)$ is itself a compressed representation of all the reversed walks, and its graph structure is much more compact than the Path-Tree structure.

We pay special attention to low space complexity because in distributed environments, the process of generating (reversed) walks will inevitably incur network communication overhead since a single computing node has no random access to the whole graph. But in most cases, data communication via network is one of the leading factors that degrade performance of distributed applications. In the next subsection, we will show that although the information of all walks is hidden in a compact neighborhood, we can still achieve fast matching of walks.

D. Faster Matching Process

After each vertex in Nei collects all reversed walks starting from itself, we need to match them with the master walks to compute the final SimRank score. Assume that all master walks can be accessed by all vertices. Note that each vertex is responsible for the reversed walk collections of his own, so the whole computation can be distributed across the cluster easily. To ease presentation, we take vertex v as an example. In the matching process, only the reversed walks that share the first common vertex v, and differ in all remaining vertex sequences with the master walk are of our interest. For example, in Fig. 2(b) paths vau and vbe are a pair of meeting walks which contributes to $s(u, e)$. Meeting walks of vau also include vbu and vcf. But vad and vau are not meeting pair because they first meet at vertex a rather than v (actually, in a distributed environment their contribution will be recognized by a somewhere else). These observations immediately tell us that, if the query vertex u resides at some level of the Path-Tree of the neighborhood, then all other vertices w at that level satisfying $LCA(u, w) = v$ will contribute to $s(u, w)$ by $c^l Pr(W_u)Pr(W_w)$, where LCA means the Lowest Common Ancestor, W_u and W_v are the reversed walks corresponding to the root-to-node path formed by w and v in the tree.

We search for all meeting pairs by Depth First Searching (DFS) the neighborhood starting from v. During DFS, the probability of the root-to-node path so far is maintained. When the recursion depth approaches l, the length of the matched master walk, DFS procedure stops and the probability of the corresponding reversed walk is recorded in a global hash map. When we go on to match reversed walks of length $l + 1$, we no longer need to start DFS from v anymore. Because probabilities of reversed walks shorter than $l + 1$ could have been recorded before. For example, in Fig. 2(b) the only matching walks of $vadu$ is $vbef$. To get its probability, we only need to choose the third vertex e as the staring point of DFS, because the probability of vbe has been recorded when searching for the meeting walks of vau.

It is not hard to see that our way of matching is a Dynamic Programming approach. We use memorization to avoid recomputing overlapping subproblems. The details are shown in Algorithm 2. Procedure LevelMatch shows how a vertex v computes meeting probabilities for master walks of length l. Among the parameters, W_l is a collection of master walks of length l. N is the neighborhood of v. $M_{l'}$ is the previous memorized meeting probabilities for master walks of length l'. In detail, $M_{l'}$ is a hash map composed of $(key, value)$ pairs, where key could be one of the neighbors of v in N, and $value$ is a list of pairs each representing the ending vertex and probability of a matching reversed walk belonging to the subtree with root key. Similarly M_l is an empty hash map to be filled for W_l. The last parameter δ is the probability threshold we will describe in the following subsections. We first loop the master walk p in W_l, and get to know in which subtree p resides (line 2-3). Then we start to DFS all other different subtrees if M_l contains no information about that

Algorithm 2 Dynamic Programming Path Matching

```
1: procedure LevelMatch($W_l, N, v, M_{l'}, M_l, \delta$)
2:     for $p \in W_l$ do
3:         $br \leftarrow$ second last vertex of $p$;
4:         for $t \in (v_N.neighbors - br)$ & $t \notin M_l$ do
5:             if $!M_{l'}$.contains($t$) then
6:                 DFS($N, t, l, M_l, \delta, t, 1, 1$);
7:             else
8:                 for $w \in M_{l'}$ do
9:                     for $nei \in w_N.neighbors$ do
10:                        DFS($N, nei, l, M_l, \delta, br, w.mul, l'$);
11:    return $M_l$.
12: procedure DFS($N, v, l, M, \delta, br, mul, depth$)
13:    $mul \leftarrow mul * v_N.indegree$;
14:    if $mul > \delta$ then
15:        return;                          ▷ Early termination.
16:    if $depth = l$ then
17:        add $(v, mul)$ to $M(br)$;       ▷ Record probability.
18:    else
19:        for $nei \in v_N.neighbors$ do
20:            DFS($N, nei, l, M, \delta, br, mul, depth + 1$);
21:    return $M$.
```

subtree (line 4). If $M_{l'}$ does not contain information of that subtree, we start DFS right from its root (line 5-6). Otherwise, we choose to start DFS from those vertices recorded in $M_{l'}$ (line 8-10). Procedure DFS show the detailed search process. We first check if we can terminate the search process early, which will be discussed in detail in next subsection (line 13-15). Then we check if the depth limit of DFS is reached. If so we stop there and record the probability (line 16-17). Otherwise, we go on to DFS the next level (line 19-20). By invoking LevelMatch for all W_l ($l \leq k$), we can compute meeting probabilities efficiently.

E. Early Termination of Walks

Many real-world graphs are scale-free [21], which means that a small portion of the vertices in the graph have very large degrees. Our algorithm could suffer a lot from the presence of these high degree vertices, since higher degree means more splits of walks. Thus, we adopt probability sieve [9] to filter out the walks whose probability is already small enough caused by containing many high degree vertices. The filtered walks will no longer advance any further because the contribution of subsequent walks are insignificant. The value of the probability threshold δ can be set manually to achieve a balance between tolerance of accuracy loss and improvement of computing efficiency. Note that this optimization generally can be applied to every walk generation process, e.g., in line (13-15) of the DFS procedure of Algorithm 2.

F. sssSimRank: Putting it All Together

Given all the techniques that we described in the previous subsections, we now list the framework of our sssSimRank algorithm as the following phases:

1) Find the set of vertices Nei that are reachable from u by following in-edges, and at the same time maintain corresponding master walks. All master walks are then broadcast to all machines in the cluster;
2) Each vertex in Nei collects its neighborhood by following out-edges;
3) Each above vertex then computes meeting probabilities based on its own copy of master walks and the reversed walks extracted from its own neighborhood;
4) All meeting probabilities scores are aggregated to get the final $s(u, *)$.

In a typical distributed environment, most of the network communication happens in phase 1 and especially phase 2, as neighbor information needs to be exchanged between all machines. Phase 4 will also incur some communication cost. Most of the computations, including searching for meeting walks and calculating their probabilities, are conducted locally by each machine in the cluster.

We analyze the complexity of our algorithm in all respects. The total amount of vertices reachable from u is $O(d^k)$. For each of the reachable vertices, it has a neighborhood of size $O(d^k)$ containing structure information of all $O(d^k)$ reversed walks. So the total space complexity is $O(d^{2k})$. The communication cost involved in Phase 1 and Phase 2 is also $O(d^{2k})$ because at most all walks are transferred. Phase 3 uses memorization technique to compute meeting probabilities, thus for a single neighborhood at most all $O(d^k)$ reversed walks are matched. This leads to a total computation cost of $O(d^{2k})$. A tight bound of communication cost of phase 4 is hard to give, but we can assert that it only depends on the topology rather than the scale of the graph. In summary, our approach of computing single-source SimRank is highly efficient and allows for high parallelism.

V. Implementation on Spark

In this section, we describe the implementation of sssSimRank on Spark. Note that phase 1 and phase 2 of our algorithm are quite similar. In both cases we start from some vertices and find their reachable neighbors. The only differences lie in the number of starting vertices (1 vs. $|Nei|$) and the direction of movement (following in-edge vs. out-edges). So we omit phase 1 and explain the other three in detail.

A. Collect Neighborhood

The process of collecting neighborhood for each vertex $v \in Nei$ is shown in Algorithm 3. The algorithm takes 3 input parameter. $edgeRDD$ is the RDD of the graph, u is the query vertex, and k is the maximum walk length. First we convert the graph from plain edge format to adjacent list format (line 2-3). We then initialize $nbrhdRDD$ from $graphRDD$ by filtering out the vertices not in Nei, which contains the one-step neighborhood of each vertex. And `cache`

Algorithm 3 Collect Neighborhood

```
1: procedure COLLECT($edgeRDD, u, k$)
2:     $graphRDD \leftarrow edgeRDD$
3:         .reduceByKey().cache();
4:     $nbrhdRDD \leftarrow graphRDD$.map().cache();
5:     $nbrRDD \leftarrow graphRDD$.filter().cache();
6:     for $l = 2$ to $k$ do
7:         $nbrRDD \leftarrow nbrDD$
8:             .join($graphRDD$).map();
9:         $nbrhdRDD \leftarrow nbrhdRDD$
10:            .leftOuterJoin($nbrRDD$).map();
11:    return $nbrhdRDD$.
```

Algorithm 4 Compute SimRank

```
1: procedure COMPUTE($nbrhdRDD, MW, u, k$)
2:     $simRankRDD \leftarrow hbrhdRDD$
3:         .flatMap(($v, nbrhd$) ⇒ {
4:             create $MP$;
5:             for $W_l$ in $MW$ do
6:                 $M \leftarrow$ LEVELMATCH($W_l, \dots$);
7:                 extend $MP$ with all elemetns in $M$;
8:             for ($v, score(u, v)$) in $MP$ do
9:                 yield ($v, score(u, v)$);
10:        })
11:    $s(u, *) \leftarrow simRankRDD$.reduceByKey().collect();
12:    return $s(u, *)$.
```

is called to persist $nbrhdRDD$ into memory for later iterations (line 4). At the very beginning, the neighborhood only covers the out-neighbors of the vertex. In the following iterations, the neighborhood will expand larger by advancing a step further from the out-most vertices. $nbrRDD$ is the RDD that contains the out-most vertices of neighborhood of each vertex. It is initialized as out-neighbors of u from $graphRDD$ (line 5). We now begin to expand all the neighborhoods iteratively. In each iteration, first the out-most neighbors are updated by joining the underneath graph (line 7-8). Then the new out-most neighbors are transferred to neighborhoods who can reach them (line 9-10). Finally, we get the neighborhood for each starting vertex after k iterations.

B. Compute and Aggregate Meeting Probabilities

After all neighborhoods of each starting vertex have been collected, we begin to match all master walks and slave walks. The details are listed in Algorithm 4. The algorithm takes 4 parameters. The first parameter $nbrhdRDD$ is the output of Algorithm 3, the RDD containing pairs of vertex and its neighborhood. The second parameter MW is the output of phase 1, i.e., all the master walks starting from u. MW is represented as a hash map of $(key, value)$ pairs, where key is the ending vertex of each master walk (note that $key \in Nei$), and $value$ is a list of the random walks ending with key. The remaining two parameters are the given query vertex and maximum walk length, respectively. For each record $(v, nbrhd)$ in $hbrhdRDD$, we invoke LevelMatch in Algorithm 2 to compute meeting probabilities of all meeting walks in $nbrhd$ for v. The results are appended into the list MP (line 5-7). Then all meeting probabilities are emitted by `flatMap` (line 8-9). In the end, `reduceByKey` aggregates all the probabilities to form the final SimRank score, which are subsequently collected to the driver program (line 11-12).

VI. Experimental Results

In this section, we evaluate our implementation of sssSimRank experimentally. We first describe our running environments, datasets used, and parameter settings. Then we analyze our results in terms of effectiveness, efficiency and scalability.

TABLE I: Description of Datasets

Dataset	#Nodes	#Edges	Avg Deg	Size
p2p-gnutella08 [1]	6,301	20,777	3.29	215.2KB
wiki-vote [2]	7,115	103.689	14.57	1.1MB
eu-2005 [3]	862,664	19,235,140	22.29	256.4MB
ljournal-2008 [4]	5,363,260	79,023,142	14.73	1.2GB
arabic-2005 [5]	22,744,080	639,999,458	28.14	10.9GB

A. Setup

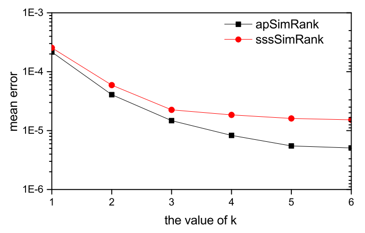
1) Running Environment: We do all experiments on a cluster of 6 machines, each has two 12-core Intel Xeon E5-2650 2.10 GHz processors, 64 GB RAM, and 1TB hard disk. Each of them are connected via a Gigabit network. All machines are running Ubuntu 14.04. The version of Spark and underlying HDFS deployed are 1.6.2 and 2.6.0 respectively. All machines are configured as slave node, and one of them also plays the role of master node. Each executor in Spark is allocated with 10GB memory.

2) Datasets: We use 5 real-word datasets of various scales. The details of each dataset are listed in Table I. Each graph is stored as plain text format, one edge per line. All datasets are uploaded into the HDFS beforehand.

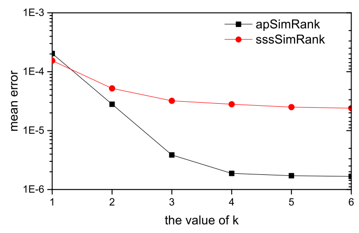
3) Parameters: As discussed by [9], the maximum steps k is determined by c and the accuracy we want. if we want make the error loss $s^*(u, v) - s^k(u, v)$ smaller than ϵ, we need to set $k = \lfloor \log_c \epsilon \rfloor$. In our experiments, we choose $\epsilon = 0.01$, which is accurate enough for most real-world applications. c is set to be 0.5 and as a result, $k = 6$.

For the optimization technique using threshold to ignore walks with small probabilities, we set $\delta = 0.002$. Notice that here δ is for a single walk. For a pair of meeting walks, the equivalent threshold for the meeting probability would roughly be δ^2 according to Eq. (6). After multiplying a factor of c^l,

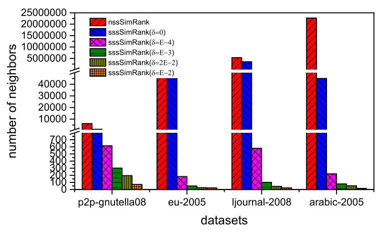
[1]https://snap.stanford.edu/data/p2p-Gnutella08.html
[2]https://snap.stanford.edu/data/wiki-Vote.html
[3]http://law.di.unimi.it/webdata/eu-2005/
[4]http://law.di.unimi.it/webdata/ljournal-2008/
[5]http://law.di.unimi.it/webdata/arabic-2005/



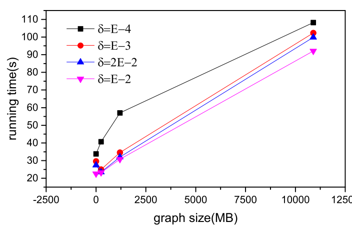
(a) Comparison of accuracy between sssSimRank and apSimRank on p2p-gnutella08.



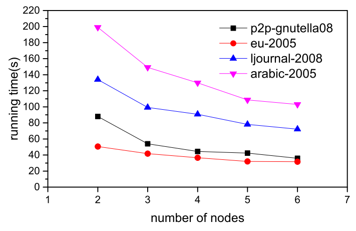
(b) Comparison of accuracy between sssSimRank and apSimRank on wiki-vote.



(c) Comparison of $|Nei|$ between nssSimRank and sssSimRank with varying probability threshold.



(d) Running time with varying graph sizes.



(e) Running time with varying number of nodes.

Fig. 3: Results of accuracy loss and convergence rate, efficiency and scalability.

it will be extremely small and therefore could be reasonably ignored.

To avoid particularity, all experiments are conducted multiple times with the query vertex chosen randomly from the graph. If not otherwise specified, for small graphs (< 10MB) we repeat 100 times and present the average results while for large graphs the repetition is 1000.

B. Effectiveness

We compare the accuracy and convergence rate between all-pair SimRank (apSimRank) and sssSimRank algorithm. We evaluate the effectiveness by computing the *mean error* $ME = \frac{1}{n}\sum_{v\in V}|s(u,v) - s^k(u,v)|$, where $s(u,v)$ is the ground truth given by Eq. (2) until convergence, and $s^k(u,v)$ is the output of algorithms running k iterations (or within k steps). The comparison is conducted on two small graphs, p2p-gnutella08 and wiki-vote. P2p-gnutella08 is a sparse graph ($d = 3.29$) while wiki-vote is much denser ($d = 14.57$). The results are shown in Fig. 3(a) and 3(b). We can see that both apSimRank and sssSimRank converge within 6 iterations. In both Fig. 3(a) and Fig. 3(b), sssSimRank achieves a faster convergence rate, with accuracy loss less than 10^{-4} after 3 iterations. Another thing to notice is that apSimRank shows greater comparative advantages in terms of accuracy in Fig. 3(b) than in Fig. 3(a). This is because sssSimRank uses a threshold δ to filter out walks with very small probabilities, which is particularly effective for scale-free graphs like wiki-vote. By doing so we improved efficiency at the cost of some accuracy. But such a small accuracy loss ($< 10^{-4}$) is ignorable for most real-world applications.

C. Efficiency

We implement all-pair SimRank (apSimRank) based on Eq. (2), and Naive Single-source SimRank (nssSimRank) based on single-source similarity discussed in [11]. Both of them are implemented as distributed algorithms on Spark. We find that a straight comparison between them and our algorithm on all graph datasets is impractical because computing apSimRank and nssSimRank is extremely time-consuming even for small graphs. For apSimRank, the smallest graph p2p-gnutella08 takes about 2.1 hour to finish. We think this fact is enough to prove that our algorithm outperforms apSimRank greatly. For nssSimRank, it takes 1.1 hour. Note that the running time of random-walk-based algorithms mainly depends on the total number of random walks generated, which further depends on the number of neighborhoods. Thus, we compare the size of Nei for both nssSimRank and sssSimRank with different δ. The results are shown in Fig. 3(c). In nssSimRank, $|Nei|$ is equal to n, because all vertices in the graph need to collect its neighborhood. In contract, our algorithm reduce $|Nei|$ drastically. When δ is 0, which means no probability sieve is used, sssSimRank generates several order of magnitude (up to 1500x) fewer neighborhoods. As δ becomes larger, which means our probability sieve is more fine-meshed, number of survivors decreases accordingly. Fig. 3(c) also indicates that the reduction ratio for eu-2005 and arabic-2005 are higher than that of p2p-gnutella08 and ljournal-2008. This shows that probability sieve works better for denser graphs, which is in line with our expectations.

D. Scalability

In this section, we investigate the scalability of our proposed algorithm. The input graphs used are p2p-gnutella08, eu-2005, ljournal-2008 and arabic-2005. We first evaluate the performance when the data size increases. The running time of sssSimRank with different δ on the four graphs are shown in Fig. 3(d). We can see that our algorithm scales well as the size of graph increases thousands of times. Fig. 3(d) also suggests that for a fixed graph, a larger δ will bring about larger amount of walks, causing increased running time.

We also evaluate the performance when the number of computing machines increases. The number of computing nodes increases from 2 to 6. The configurations for all datasets are the same, with $k = 6$ and $\delta = 1E\text{-}4$. We adopt a small δ to increase the overall computing load, because with small workload Spark initialization will dominate the running time. The results are shown in Fig. 3(e). The nature of the curve indicates that, as the number of nodes increases the running time decreases as expected for strong scalability.

VII. Conclusion

We have proposed and implemented a highly parallelizable algorithm, sssSimRank, for the computation of single-source SimRank. Our algorithm is based on the random walk model. It achieves good performance by minimizing the number of walks, compressing representation of intermediate data, and fast matching via dynamic programming. Experimental results verify the effectiveness, efficiency of our approach.

Acknowledgment

This paper is partly funded by National Science and Technology Major Project of the Ministry of Science and Technology of China under grant 2011ZX05035-004-004HZ. The corresponding author of this paper is Jie Tang. We thank the reviewers for helping us refine this paper.

References

[1] F. Fouss, A. Pirotte, J.-M. Renders, and M. Saerens, “Random-walk computation of similarities between nodes of a graph with application to collaborative recommendation,” *IEEE Transactions on knowledge and data engineering*, vol. 19, no. 3, pp. 355–369, 2007.

[2] I. Bhattacharya and L. Getoor, “Entity resolution in graphs,” *Mining graph data*, p. 311, 2006.

[3] J. Dean and M. R. Henzinger, “Finding related pages in the world wide web,” *Computer networks*, vol. 31, no. 11, pp. 1467–1479, 1999.

[4] P. Jaccard, *Etude comparative de la distribution florale dans une portion des Alpes et du Jura*. Impr. Corbaz, 1901.

[5] R. Baeza-Yates, B. Ribeiro-Neto *et al.*, *Modern information retrieval*. ACM press New York, 1999, vol. 463.

[6] L. R. Dice, “Measures of the amount of ecologic association between species,” *Ecology*, vol. 26, no. 3, pp. 297–302, 1945.

[7] G. Jeh and J. Widom, “Simrank: a measure of structural-context similarity,” in *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2002, pp. 538–543.

[8] L. Page, S. Brin, R. Motwani, and T. Winograd, “The pagerank citation ranking: bringing order to the web.” 1999.

[9] D. Lizorkin, P. Velikhov, M. Grinev, and D. Turdakov, “Accuracy estimate and optimization techniques for simrank computation,” *Proceedings of the VLDB Endowment*, vol. 1, no. 1, pp. 422–433, 2008.

[10] W. Yu, W. Zhang, X. Lin, Q. Zhang, and J. Le, “A space and time efficient algorithm for simrank computation,” *World Wide Web*, vol. 15, no. 3, pp. 327–353, 2012.

[11] P. Li, H. Liu, J. X. Yu, J. He, and X. Du, “Fast single-pair simrank computation.” in *SDM*. SIAM, 2010, pp. 571–582.

[12] W. Yu, X. Lin, and W. Zhang, “Towards efficient simrank computation on large networks,” in *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*. IEEE, 2013, pp. 601–612.

[13] C. Li, J. Han, G. He, X. Jin, Y. Sun, Y. Yu, and T. Wu, “Fast computation of simrank for static and dynamic information networks,” in *Proceedings of the 13th International Conference on Extending Database Technology*. ACM, 2010, pp. 465–476.

[14] G. He, H. Feng, C. Li, and H. Chen, “Parallel simrank computation on large graphs with iterative aggregation,” in *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2010, pp. 543–552.

[15] 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,” *Information Systems*, vol. 42, pp. 107–122, 2014.

[16] D. Fogaras and B. Rácz, “Scaling link-based similarity search,” in *Proceedings of the 14th international conference on World Wide Web*. ACM, 2005, pp. 641–650.

[17] M. Kusumoto, T. Maehara, and K.-i. Kawarabayashi, “Scalable similarity search for simrank,” in *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. ACM, 2014, pp. 325–336.

[18] P. Lee, L. V. Lakshmanan, and J. X. Yu, “On top-k structural similarity search,” in *2012 IEEE 28th International Conference on Data Engineering*. IEEE, 2012, pp. 774–785.

[19] L. Cao, B. Cho, H. D. Kim, Z. Li, M.-H. Tsai, and I. Gupta, “Delta-simrank computing on mapreduce,” in *Proceedings of the 1st International Workshop on Big Data, Streams and Heterogeneous Source Mining: Algorithms, Systems, Programming Models and Applications*. ACM, 2012, pp. 28–35.

[20] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, “Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing,” in *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association, 2012, pp. 2–2.

[21] L. Li, D. Alderson, J. C. Doyle, and W. Willinger, “Towards a theory of scale-free graphs: Definition, properties, and implications,” *Internet Mathematics*, vol. 2, no. 4, pp. 431–523, 2005.