On Link-based Similarity Join

Liwen Sun[†] Reynold Cheng[†] Xiang Li[†] David W. Cheung[†] Jiawei Han[§]

[†]Department of Computer Science, University of Hong Kong, Hong Kong, HKSAR, China
{lwsun, ckcheng, xli, dcheung}@cs.hku.hk

[§]Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL 61801, USA
hanj@cs.uiuc.edu

ABSTRACT

Graphs can be found in applications like social networks, bibliographic networks, and biological databases. Understanding the relationship, or *links*, among graph nodes enables applications such as link prediction, recommendation, and spam detection. In this paper, we propose *link-based similarity join* (LS-join), which extends the similarity join operator to link-based measures. Given two sets of nodes in a graph, the LS-join returns all pairs of nodes that are highly similar to each other, with respect to an *e*-function. The *e*-function generalizes common measures like Personalized PageRank (PPR) and SimRank (SR). We study an efficient LS-join algorithm on a large graph. We further improve our solutions for PPR and SR, which involve expensive random-walk operations. We validate our solutions by performing extensive experiments on three real graph datasets.

1. INTRODUCTION

Many emerging applications contain a large amount of inter-related information. For example, millions of Facebook users can establish friendship with each other. Bibliographic networks, such as DBLP and CiteSeer, contain author collaboration information and citation details. In e-commerce applications, complex business relationship exists among suppliers, retailers, and consumers. A *graph* naturally captures this kind of information. Figure 1 illustrates a graph that models the relationship among sales managers and customers in a company.

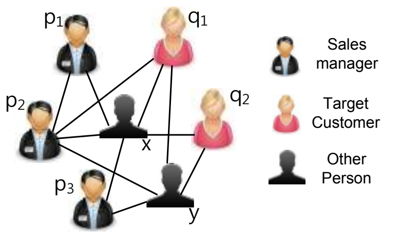


Figure 1: Promotion suggestion.

In recent years, a lot of effort has been devoted to extracting useful information from large graphs. Particularly, researchers have proposed link prediction [7], graph clustering [20, 21], spam detection [22], and recommender systems [1]. These techniques adopt a *link-based similarity* measure, which computes the proximity between two nodes based on the paths connecting them. Common examples of this measure include Personalized PageRank (PPR) [13], SimRank (SR) [9], and Common Neighbors [7]. In this paper, we study a new query called the *Link-based Similarity join* (or *LS-join* in short). Given two sets of nodes in a graph, the LS-join returns k pairs of nodes, which are ranked the highest according to some linked-based similarity score. This query can be used in a lot of applications, including:

- **Promotion suggestion.** To promote new products to customers, a social network that contains a company's salespersons and prospective customers can be a valuable tool. In particular, the LS-join suggests a set of (salesperson, customer)-pairs for promotion purposes. In Figure 1, we can see that p_2 (a sales manager) and q_1 (a target customer) not only know each other, but also have a lot of "indirect" connections, e.g., person x and y. If p_2 wants to recommend a product to q_1, he can seek the help from x or y. He can also give them coupons if they successfully invite q_1 to buy the product. A PPR measure that quantifies the closeness of two objects in terms of the paths between them can be used [13]. For example, the PPR of p_2 and q_1 is high, since they are closely related. A LS-join, which uses PPR to measure the proximity between salespersons and customers, may yield (p_2, q_1) as an answer. This result suggests that the company should let p_2 approach q_1.
- **Citation analysis**. In a bibliographic network, a node represents a publication, and a directed edge from node a to b indicates that paper a cites paper b. The similarity of two papers can be captured by the SR [9] – two papers have a high SR score if the sets of nodes that point to them are similar. Given the papers published in two areas (e.g., Artificial Intelligence (AI) and Database (DB)), a LS-join, using the SR, returns k pairs of papers that are cited by similar papers. The pairs of papers returned, which reflect that they investigate similar problems, can be useful to scientists interested in cross-disciplinary work in AI and DB.
- **Link prediction**. As a graph evolves over time, new edges can appear among nodes. For example, two users in a social network, who do not know each other, can become friends later. An edge then appears between the nodes that represent these users. The problem of *link prediction* is to estimate what edges will be generated in the future. One

way to predict links is to identify node pairs that are highly similar. In a social network, we can consider two users to be similar if they have many friends in common. They may also become friends later. Given two interest groups in a social network, we can use a LS-join retrieve k pairs of users, who have the highest tendency to become friends.

Challenges. Evaluating a LS-join query is not trivial. This is because many common link-based measures (e.g., PPR and SR) involve *random-walk* computation. For example, calculating a PPR score of two nodes a and b involves finding the probability that a random walker at a can reach b, for every path between a and b. This operation, which considers all paths between a and b, incurs significant overhead, especially on a large graph. A simple implementation of the LS-join by computing the scores for all node pairs, can be extremely expensive. We thus study efficient LS-join query solutions, as detailed below.

1. Developing the IDJ algorithm for the e-function. We observe that common similarity measures, such as PPR, SR, and Discounted Hitting times [19], can be generalized to a form called the ***e*-function**. Given nodes s and t, let $\mathcal{P}_i(s,t)$ be a probability function that involves i steps of random walks. The definition of $\mathcal{P}_i(s,t)$ depends on the similarity measure used. An e-function is a *weighted* sum of $\mathcal{P}_i(s,t)$'s, for all values of i, where the weight of $\mathcal{P}_i(s,t)$ decreases exponentially with i. When i is small (say, $i \leq 3$), $\mathcal{P}_i(s,t)$ can be used to estimate the e-function with a high accuracy. Based on this intuition, we propose the *Iterative Deepening Join* algorithm (or *IDJ* in short), which is inspired by the classical strategy of *iterative-deepening depth first search* in state-space search [18]. The term "iterative deepening" means that the length (or *depth*) of the paths between s and t considered increases with the number of iterations. This information, which can be easily obtained in the first few rounds, can prune nodes effectively. Another salient feature of the IDJ is that the depth increases exponentially with i. In our experiments, the IDJ can improve the performance of a basic solution by one order of magnitude.

2. Enhancing the IDJ for PPR and SR. We further customize the IDJ for two important e-functions, namely PPR and SR. For these measures, we develop fast methods (e.g., backward random walk) to compute their lower and upper bounds. Theoretically, these bounds attain a higher pruning efficiency than the IDJ. In our experiments, PPR and SR outperform IDJ by an order of magnitude.

Prior works. The evaluation of *similarity join* between two sets of objects has been well studied. [3, 11] addressed this in high dimensional databases, where the Euclidean distance was used as a similarity measure. In [2, 5], set- and string-similarity (e.g., Hamming and Edit distances) were used in data cleaning applications. Few works have studied the similarity join on graphs. [12] and [15] respectively investigated the similarity join for road network and graph-pattern matching, by using the *shortest-path* distance. It is not clear how these work can handle an e-function (e.g., PPR and SR), which we investigate in this paper.

Link-based measures has been widely used in network applications, such as link prediction [7] and k-nearest neighbor search [19]. The PPR [13] is extended from Google's PageRank model. To compute PPR scores, iterative methods [13], random-walk simulations [6], and approximation algorithms [19], have been developed. The SR [9] defines the node similarity recursively based on in-neighbors and is expensive to compute [16]. To evaluate SR efficiently, random walk simulation [8], iterative [16], and non-iterative methods [4], have been proposed. An algorithm for computing single-pair SR has been recently proposed [17]. Complementary to these works, we study how to prune expensive PPR and SR score computation during a join operation.

The rest of our paper is organized as follows. Section 2 formalizes the e-function and the LS-join. Section 3 describes the IDJ algorithm. We enhance the IDJ for PPR and SR in Sections 4 and 5 respectively. Section 6 presents our results on three real datasets. Section 7 concludes.

2. THE *E*-FUNCTION AND THE LS-JOIN

We now study the e-function (Section 2.1) and the LS-join (Section 2.2). We present a simple solution in Section 2.2.

2.1 The e_d-function and the e-function

Let G be a graph on which the e-function is defined. Also, let V and E be the sets of nodes and edges of G respectively. We assume that G is directed and weighted, with the weight of an edge (u,v) denoted by $w(u,v)$. Given a node u, we use $I(u)(O(u))$ to denote the set of in(out)-neighbor nodes of u. We suppose that G is represented by a data structure (e.g., an adjacency list), which allows efficient enumeration of neighbors for a given node.

In the *random walk* model, a *random surfer* traverses nodes in V according to the edges in E. At every step, the surfer moves from u where he is currently located, to an out-neighbor v of u with an *outgoing probability* $p_{out}(u,v)$. For an unweighted graph, $p_{out}(u,v) = \frac{1}{|O(u)|}$; for a weighted graph, $p_{out}(u,v) = \frac{w(u,v)}{\sum_{v' \in O(u)} w(u,v')}$. The random walk model is often used to define link-based similarity measures, for instance, PPR and SR. In these measures, a random-walk-related probability value, $\mathcal{P}_i(u,v)$, is defined, where i is the number of steps moved by the surfer. For example, in PPR, $\mathcal{P}_i(u,v)$ is the chance that a surfer at u can reach v at the i-th step. We now define the e_d-function and the e-function.

Definition 1 (e_d-function). *Given two nodes $u, v \in V$, the e_d-function, denoted by $\mathcal{S}_d(u,v)$, is:*

$$\mathcal{S}_d(u,v) = a \sum_{i=1}^{d} \lambda^i \mathcal{P}_i(u,v) + b \qquad (1)$$

- *a and b are real-valued constants, with $a > 0$;*
- *$\lambda \in (0,1)$ is called a constant decay factor; and*
- *$d \in \aleph$ is called the depth of the e_d-function.*

Here, d is also the maximum number of steps used in the random walk.

Definition 2 (e-function). *Given two nodes $u, v \in V$, the e-function, denoted by $\mathcal{S}(u,v)$, is:*

$$\mathcal{S}(u,v) = \lim_{d \to \infty} \mathcal{S}_d(u,v) \qquad (2)$$

The e_d-function, $\mathcal{S}_d(u,v)$, computes the similarity between u and v by using the random walk model up to d steps. Due to the presence of λ^i, $\mathcal{S}_d(u,v)$ converges as d approaches to infinity. Hence, the value of the e-function (Definition 2) exists. Moreover, in Equation 1, $\lambda^i \mathcal{P}_i(u,v)$ *decays exponentially* with i. Hence, a longer random-walk path contributes

Table 1: Summary of Notations

Notation	Meaning
$G(V, E)$	graph G of node set V and edge set E
P, Q	joining sets, where $P, Q \subseteq V$
u, v	graph nodes
(u, v)	a graph edge
$I(u), O(u)$	Set of in- and out-neighbors of u, $I(u), O(u) \subseteq V$
λ	decay factor
$\mathcal{S}_d(u, v)$	e_d-function score between u and v
$\mathcal{S}(u, v)$	e-function score between u and v
ε	tolerance
$S_z(u, v)$	the e_z-function that satisfies ε

less to the values of $\mathcal{S}_d(u, v)$ and $\mathcal{S}(u, v)$. We will exploit this observation to design an efficient LS-join algorithm. Now let us state without proof a simple but useful result:

LEMMA 1. *The e_d function (Equation 1) is a monotonically increasing function of d.*

The e-function can be specialized to a wide range of link-based similarity measures, for instance:

- Personalized PageRank (PPR) [13]: $a = 1 - \lambda$; $b = 0$; and $\mathcal{P}_i(u, v)$ is the probability that a random surfer from u *reaches* v at the i-th step.
- SimRank (SR) [13]: $a = 1$; $b = 0$; and $\mathcal{P}_i(u, v)$ is the probability that two surfers from u and v *first meet* each other at the i-th step.
- Discounted Hitting Times (DHT) [19]: $a = 1$; $b = 1$; and $\mathcal{P}_i(u, v)$ is the probability that a surfer from u *first reaches* v at the i-th step.

In this paper, we focus on PPR and SR. Nevertheless, we will also present an LS-join algorithm for any e-function.

2.2 The LS-Join Query

Observe from Equation 2 that $\mathcal{S}(u, v)$ requires an infinite number of $\mathcal{S}_d(u, v)$ terms. Hence, $\mathcal{S}(u, v)$ can be hard to find. A practical way is to compute $\mathcal{S}_z(u, v)$ instead:

$$|\mathcal{S}(u, v) - \mathcal{S}_z(u, v)| \leq \varepsilon \quad (3)$$

where $\varepsilon \in \Re$, called the *tolerance* of $\mathcal{S}(u, v)$, controls the accuracy of $\mathcal{S}_z(u, v)$ in estimating $\mathcal{S}(u, v)$. Let $P, Q \subseteq V$ be the *joining sets* of the LS-join, then we have:

DEFINITION 3 (LINK-BASED SIMILARITY JOIN (LS-JOIN)). *Given G, P, Q, ε, e-function $\mathcal{S}(u, v)$, and an integer k, the LS-Join returns a sorted-list of (p, q)-pairs with the k highest $\mathcal{S}_z(p, q)$ values, where $p \in P$ and $q \in Q$.*

Table 1 summarizes the notations used in this paper.

In all solutions studied in this paper, the value of z that satisfies Equation 3 is first obtained. Let us explain how this is done by showing the following lemma.

LEMMA 2. *$\mathcal{S}(u, v) \in [\mathcal{S}_d(u, v),\ \mathcal{S}_d(u, v) + X_d^+]$, where*

$$X_d^+ = \frac{a \cdot \lambda^{d+1}}{1 - \lambda} \quad (4)$$

Lemma 2 gives the lower and upper bounds of the e-function, by using the value of X_d^+. Its proof can be found in Appendix A. Observe from Equation 4 that as d increases, X_d^+ becomes smaller. Hence, $\mathcal{S}_d(u, v)$ approaches $\mathcal{S}(u, v)$ when d increases. We next obtain the following result.

LEMMA 3. *To satisfy Equation 3,*

$$z \geq \frac{\log \frac{(1-\lambda)\varepsilon}{a\lambda}}{\log \lambda} \quad (5)$$

This lemmas tells us how to find z for a given tolerance ϵ. Its proof can be found in Appendix A. In PPR, if $a = \lambda = 0.5$, $b = 0$, and $\varepsilon = 10^{-6}$, then $z \geq 19$. Notice that the smaller the tolerance, the larger is the value of z. In the sequel, we assume that z has been obtained by using this lemma.

A **basic solution** for performing LS-join is to compute $\mathcal{S}_z(p, q)$ for every node pair $(p, q) \in (P, Q)$, and then return the pairs with the k highest scores. Although efficient algorithms for computing similarity scores exist, they are tailored for specific measures (e.g., PPR [6]), and it is not clear how they can support other e-functions. Moreover, many node pairs that are not the answers still need to have their e_z-function values fully computed. Due to the prevalence of the small-world phenomenon [14], a smaller ε (or a big z) likely invokes expensive computation that spans a significant portion of a graph. As shown in our experiments, such a method does not perform well, especially on a large graph. We next study a better algorithm.

3. ITERATIVE DEEPENING JOIN (IDJ)

We now present the *Iterative Deepening Join* algorithm (or *IDJ*), which evaluates the LS-join for an e-function. We discuss the framework of the IDJ in Section 3.1. We then examine its details in Sections 3.2 and 3.3.

3.1 Algorithm Design

Recall that the LS-join returns node pairs $(p, q) \in (P, Q)$ that yield the k highest e_z-function scores ($\mathcal{S}_z(p, q)$). If z is large, $\mathcal{S}_z(p, q)$ can be expensive to compute. Thus, the IDJ chooses not to compute $\mathcal{S}_z(p, q)$ for every node pair in P and Q. Instead, it executes the LS-join in an *iterative* manner. At every round, an approximate value of $\mathcal{S}_z(p, q)$ is used to prune nodes. As the number of iterations grows, the approximation of $\mathcal{S}_z(p, q)$ is *deepened* – i.e., it becomes more precise at a higher cost. On the other hand, the number of node pairs that needs to be operated is reduced. In the final round, the IDJ evaluates the exact e_z-function values for node pairs that cannot be pruned. If most of the nodes can be pruned in early rounds, which are relatively cheap to process, then the IDJ can finish quickly. The idea of IDJ is inspired by the classical strategy of *iterative-deepening depth first search* (or *IDDFS*) for state-space search [18], on which we discuss in detail at Appendix B.

In the j-th iteration of the IDJ, we collect the following information about $\mathcal{S}_z(p, q)$:

- For every pair $(p, q) \in (P, Q)$, we derive $\mathcal{S}_z^-(p, q)$, a lower bound of $\mathcal{S}_z(p, q)$; and
- For each node $p \in P$, we compute $\mathcal{S}_z^+(p, Q)$, an upper-bound of the e_z-function between p and any node $q \in Q$, i.e., $\mathcal{S}_z^+(p, Q) \geq \mathcal{S}_z(p, q)$, for any $q \in Q$.

We then find a set R of k pairs with the highest $\mathcal{S}_z^-$ scores. Let T_k be the k-th largest $\mathcal{S}_z^-$ score of the pairs stored in R. We use the following *pruning rule*:

- *For any $p \in P$, if $\mathcal{S}_z^+(p, Q) < T_k$, then p is pruned.*

The tightness of the above bounds is controlled by a *depth function*, or $dep(j)$, where j is the number of iterations. This function is defined in a way that when j is small, the bounds are loose but are cheap to estimate. As j increases, tighter bounds can be obtained at a higher cost. We will revisit $dep(j)$ in Section 3.3.

Algorithm 1 details the IDJ. Step 1 initializes j to one[1]. In the j-th iteration, Step 3 executes `join-bound`. Given a depth function $dep(j)$, `join-bound` derives the set R of node pairs with the k highest $\mathcal{S}_z^-$ scores, as well as an array up, where $up[p] = \mathcal{S}_z^+(p, Q)$, for every $p \in P$. Step 4 finds T_k from R. We then use the *pruning rule* to remove nodes from P (Steps 5-7). This process (Steps 2-8) is repeated until $dep(j) \geq z$. Step 9 executes `join-refine`, which computes the exact e_z-function for nodes that remain in P and Q by using Equation 1, and put the node pairs with the k highest $\mathcal{S}_z(p, q)$ values to R. Step 10 returns R.

The IDJ is designed to alleviate the problems of the basic methods mentioned in Section 2.2. The IDJ only computes the exact e_z-function scores for node pairs in P and Q that cannot be pruned. This can be better than the basic solution that evaluates the e_z-function for all node pairs in P and Q. The main concern about the IDJ is whether it can carry out pruning (Steps 2-8) efficiently and effectively. This depends on: (1) the implementation of `join-bound`; and (2) the setting of $dep(j)$. Let us study these two important issues in more detail.

Algorithm 1: `Iterative-Deepening-Join (IDJ)`

```
   Input: Graph $G(V, E)$; $k$ and $z$; joining sets $P$,$Q$;
   Output: top-$k$ pairs $R$ with the highest $\mathcal{S}_z(p, q)$ values
1  init($j$);
2  while $dep(j) < z$ do
3      $(R, up) \leftarrow$join-bound($G, P, Q, k, dep(j)$);
4      $T_k \leftarrow R.getMin()$;
5      for each $p \in P$ do
6          if $up[p] < T_k$ then
7              Remove $p$ from $P$;
8      $j \leftarrow j + 1$;
9  $R \leftarrow$join-refine($G, P, Q, z$);
10 return $R$;
```

3.2 Bounding the e_z-function

We now present the `join-bound` and the `join-refine` routines used in Algorithm 1. First, we explain how to obtain the lower and upper bounds of the e_z-function (i.e., $\mathcal{S}_z^-$ and $\mathcal{S}_z^+$), which are used by these routines.

Deriving $\mathcal{S}_z^-$. Recall from Lemma 1 that Equation 1 monotonically increases with d . Thus, for any $d \in [0, z]$, $\mathcal{S}_d(p, q) \leq \mathcal{S}_z(p, q)$. Hence, we can set $\mathcal{S}_z^-$ as:

$$\mathcal{S}_z^-(p, q) = \mathcal{S}_d(p, q) \tag{6}$$

Deriving $\mathcal{S}_z^+$. We obtain $\mathcal{S}_z^+$ by:

$$\mathcal{S}_z^+(p, Q) = \text{MAX}\{\mathcal{S}_z^-(p, q) \mid q \in Q\} + X_d^+ \tag{7}$$

which can be deduced by: (1) $\mathcal{S}_z(p, q) \leq \mathcal{S}(p, q)$ (Lemma 1); (2) $\mathcal{S}(p, q) \leq \mathcal{S}_d(p, q) + X_d^+$ (Lemma 2); and (3) Equation 6.

Notice that as d increases, $\mathcal{S}_d(p, q)$ is more expensive to evaluate, and so are Equations 6 and 7.

Algorithm 2 presents the pseudocode of `join-bound`. For every $p \in P$, we invoke `CompLowerBound` to return a vector c (Step 3), which stores $\mathcal{S}_z^-(p, q)$ for every $q \in Q$, using Equation 6. For every $q \in Q$, we push $c[q]$ into R, using `updateResult` (Steps 4-5). Here, R is a *min-heap* of size k, and the details of `updateResult` can be found in Appendix B. In Step 6, `CompUpperBound` uses Equation 7 to computes $up[p]$. Step 7 returns R and up. The cost of Step 3, denoted by C_L, depends on the e-function used. Step 5 incurs a cost of $O(\log k)$, while Step 6 takes $O(|Q|)$ times. Hence, Algorithm 2 costs $O(p(C_L + q \log k + Q))$.

The `join-refine` function is implemented by invoking `compLowerBound` with depth z, in order to compute $\mathcal{S}_z(p, q)$. Its details can be found in Appendix B.

Algorithm 2: `join-bound`

```
  Input: graph $G(V, E)$; joining sets $P, Q$; size $k$; depth $d$
  Output: top-$k$ pairs $R$; upper-bound vector $up$
1 $R \leftarrow \emptyset$ //$R$ is a min-heap of size $k$;
2 for each $p \in P$ do
3     $c \leftarrow$compLowerBound($G, P, Q, k, d, p$);
4     for each $q \in Q$ do
5         $R \leftarrow$ updateResult($R, c[q], (p, q), k$);
6     $up[p] \leftarrow$ compUpperBound($c, d, a, \lambda$);
7 return $(R, up)$;
```

3.3 The Depth Function

When `join-bound` is invoked in Step 3 of Algorithm 1, its parameter d is the value of the *depth function*, $dep(j)$, for the j-th iteration. As explained in Section 3.2, the larger the value of d, the more costly is the bound computation. It is thus important to set $dep(j)$ appropriately, in order to attain *efficient* and *effective* pruning. Here we design $dep(j)$ based on two intuitions:

1. *$dep(j)$ should be an increasing function of j.* Lemma 1 implies that $\mathcal{S}_d(p, q)$ approaches $\mathcal{S}_z(p, q)$, as d tends to z. Also, the IDJ is designed in such a way that as j increases, `join-bound` should have a better estimate of $\mathcal{S}_z(p, q)$, in order to prune nodes that are not removed in previous iterations. Since `join-bound` uses $\mathcal{S}_d(p, q)$ (Section 3.2), and $d = dep(j)$, $dep(j)$ should increase with j.

2. *$dep(j)$ should be an exponential function of j.* Equation 7 tells us that $\mathcal{S}_z^+$ is affected by X_d^+. As shown in Equation 4, X_d^+ shrinks *exponentially* with d. Figure 2 shows how X_d^+ changes with d, at $a = \lambda = 0.5$ and $z = 19$. Hence, S_z^+ also shrinks exponentially with d. Since S_z^+ is used for removing nodes, the pruning effect drops drastically for large d. As S_z^+ is more costly to compute for large d, we should avoid executing `join-bound` at these d values.

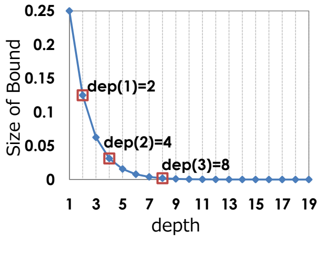


Figure 2: X_d^+ **vs.** d**.**

Thus, $dep(j)$ should be an exponential function of j (e.g., 2^j). Figure 2 shows the first three iterations of the IDJ, where `join-bound` is evaluated at $d = 2$, 4, and 8. We can see that these points capture the trend of X_d^+ well. We thus

[1]The `init` function for the PPR, which is more complex, will be discussed in Section 4.

use $dep(j) = 2^j$. Next, we study how to enhance the IDJ for PPR and SR measures.

4. CUSTOMIZING IDJ FOR PPR

We now discuss how the IDJ can be optimized for the PPR. We first present an implementation of `compLowerBound` in Section 4.1. In Section 4.2 we discuss a better design of `compUpperBound`, which efficiently returns a tighter upper bound of S_z than the method we studied in Section 3.2.

4.1 Design of `compLowerBound`

As discussed in Section 2.1, the PPR is an e-function, with $a = 1 - \lambda$, $b = 0$, and $\mathcal{P}_i(p,q)$=$V_i(p,q)$, where $V_i(p,q)$ is the probability that a random surfer at p visits q at the i-th step. Given a node $p \in P$, `compLowerBound` returns an array c, where for every $q \in Q$, $c[q]$ stores $\mathcal{S}_z^-(p,q)$, which is equal to $\mathcal{S}_d(p,q)$ (Equation 6), or:

$$\mathcal{S}_d(p,q) = (1-\lambda)\sum_{i=1}^{d} \lambda^i V_i(p,q) \tag{8}$$

The `compLowerBound` contains d iterations. At iteration i (where $i \in [1,d]$), let r be a vector, where $r[u]$ stores $V_{i-1}(p,u)$ for every $u \in V$. Initially, r stores $V_0(p,u)$ (i.e., $r[p] = 1$ and $r[u] = 0$ for $u \neq p$). We obtain $V_i(p,v)$ for every $v \in V$, by performing a *one-step* random walk:

$$V_i(p,v) = \sum_{(u,v)\in E} r[u]\cdot p_{out}(u,v) \tag{9}$$

The values of $V_i(p,v)$ are then written to r. We also accumulate the values of r in c, using Equation 8. After d iterations, each entry $c[q]$ contains the value $\mathcal{S}_z^-(p,q)$. This `compLowerBound` needs $O(d\cdot|E|)$ time and $O(|V|)$ space. In practice, it runs very fast in the first few iterations. The details (Algorithm 5) and a complexity analysis can be found in Appendix C.1.

4.2 Design of `compUpperBound`

To understand how we redesign `compUpperBound` for the PPR, we first present an important result about $\mathcal{S}_z^+(p,Q)$.

THEOREM 1. *Let $V_i(p,Q)$ be the probability that a surfer at p visits some node in Q at the i-th step. For any $p \in P$, and $0 \le d \le z$,*

$$\mathcal{S}_z^+(p,Q) = \text{MAX}\{\mathcal{S}_d(p,q) \mid q \in Q\} + Y_d^+(p,Q) \tag{10}$$

where

$$Y_d^+(p,Q) = \lambda^d \sum_{v\in V} V_d(p,v)\cdot Y_0(v,Q) \tag{11}$$

and

$$Y_d(p,Q) = (1-\lambda)\sum_{i=d+1}^{z} \lambda^i V_i(p,Q) \tag{12}$$

PROOF. (Detail in Appendix C.2) We first claim that:

LEMMA 4. *For any $p \in P$, $q \in Q$, and $0 \le d \le z$,*

$$\mathcal{S}_z(p,q) \le \mathcal{S}_d + Y_d(p,Q) \tag{13}$$

Next, we show that:

LEMMA 5. *For any $p \in P$, $0 \le d \le z$,*

$$Y_d(p,Q) \le Y_d^+(p,Q) \le Y_d(p,Q) + \varepsilon \tag{14}$$

That is, $Y_d^+(p,Q)$ differs from $Y_d(p,Q)$ by at most ε. From Lemma 5, we have:

$$\mathcal{S}_z(p,q) \le \mathcal{S}_d + Y_d^+(p,Q) \tag{15}$$

We then obtain Equation 10, and the theorem holds. □

New `compUpperBound`. Given a node $p \in P$, a set of $\mathcal{S}_z^-(p,q)$ values stored in c for every $q \in Q$, the routine `compUpperBound` developed in Section 3.2 computes $\mathcal{S}_z^+(p,Q)$. Now, if $Y_0(p,Q)$ is known (which we will explain how to get), Equation 11 can be used to get $Y_d^+(p,Q)$. We can then use c, together with Theorem 1, to obtain $\mathcal{S}_z^+(p,Q)$. The details are shown in Algorithm 8 in Appendix C.3. Note that the cost of deriving $Y_d^+(p,Q)$, which is $O(|V|)$, has the same cost of deriving X_d^+. Thus, Algorithm 8 has the same complexity as that of deriving $\mathcal{S}_z^+(p,Q)$ by using Equation 7.[2]

Algorithm 8 can yield a tighter upper bound than that of Equation 7. We first note the following:

LEMMA 6. *For any $p \in P$, $Y_d(p,Q) \le X_d^+$.*

The proof of the above lemma can be found in Appendix C.2. Also, Lemma 5, shows that the difference between $Y_d(p,Q)$ and $Y_d^+(p,Q)$ is ε or less. If ε is small, $Y_d^+(p,Q)$ is likely to be less than X_d^+. Thus, the upper bound obtained by Algorithm 8, using Equation 15, can be smaller than the one obtained from Equation 7.

Finding $Y_0(p,Q)$ by backward random walk. Recall that our approach needs to know $Y_0(p,Q)$. However, evaluating Y_0 (Equation 12) can be expensive, since it involves finding many $V_i(p,Q)$ values, each of which involves i steps of random walk. We now present an efficient algorithm of obtaining $Y_0(v,Q)$ for all $v \in V$. The new `compUpperBound` contains z iterations. At iteration i (where $i \in [1,z]$), let rb be a vector, where $rb[v]$ stores $V_{i-1}(v,Q)$ for every $v \in V$. Initially, rb stores $V_0(v,Q)$ (i.e., $rb[v] = 1$ for $v \in Q$ and $rb[v] = 0$ otherwise). We obtain $V_i(u,Q)$, for every $u \in V$, by performing a one-step *backward* random walk:

$$V_i(u,Q) = \sum_{(u,v)\in E} rb[v]\cdot p_{out}(u,v),\ \forall u \in V \tag{16}$$

The values of $V_i(u,Q)$ are then written to rb. We also accumulate the values of rb in another size $|V|$ vector $y0$, using Equation 12. After z iterations, each entry $y0[v]$ contains the value of $Y_0(v,Q)$. This method, detailed in Algorithm 9 of Appendix C.3, is used in Step 1 of the IDJ (Algorithm 1). It only needs $O(z\cdot|E|)$, the same cost of performing z steps of random walks on G.

5. CUSTOMIZING IDJ FOR SR

We now discuss how to optimize the IDJ for the SR. We present a simple implementation of `compLowerBound` for the SR in Section 5.1. Section 5.2 describes a faster version of `compLowerBound`.

5.1 Design of `compLowerBound`

As discussed in Section 2.1, the SR is an e-function, with $a = 1$, $b = 0$, and $\mathcal{P}_i(p,q)$=$F_i(p,q)$, where $F_i(p,q)$ is the

[2]We do not use $Y_d(p,Q)$ (Lemma 4) to obtain $\mathcal{S}_z^+(p,Q)$, because of its high cost: it involves evaluating many $V_i(p,Q)$ values, each of which incurs i steps of random walk. Instead, we use $Y_d^+(p,Q)$, which can be obtained more efficiently.

probability that two random surfers at p and q first meet each other at the i-th step. Given a node $p \in P$, `compLowerBound` returns an array c, where for every $q \in Q$, $c[q]$ stores $\mathcal{S}_z^-(p,q)$, which is equal to $\mathcal{S}_d(p,q)$ (Equation 6), or:

$$\mathcal{S}_d(p,q) = \sum_{i=1}^{d} \lambda^i F_i(p,q) \tag{17}$$

Recall that for the PPR, c can be obtained by performing random walk for p. Computing c for the SR is more complex; we also need to consider the random walk for each $q \in Q$, in order to evaluate the probability that p and q first meet. In our implementation, for each $q \in Q$, we invoke `compSR` on (p,q), which returns the SR score of p and q (i.e., $\mathcal{S}_d(p,q)$), and store the result in $c[q]$ (Algorithm 11 in Appendix D.1).

A sampling algorithm. A simple implementation of `compSR` can be very costly, since computing $F_i(p,q)$ involves remembering the previous locations of the two surfers, to check if they have met before. In fact, evaluating the exact value of $\mathcal{S}_d(p,q)$ needs $O(d|V||E|)$ time and $O(|V|^2)$ space [17]. We thus develop a sampling algorithm to evaluate $\mathcal{S}_d(p,q)$. This algorithm executes n rounds, where n is the number of samples decided by the Hoeffding's Inequality [10]. In the i-th round, we obtain a score by simulating the motion of two surfers at node p and q. Particularly, at each step, the two surfers randomly choose their out-neighbor to move forward. When the two surfers arrive at the same node, we compute a score of λ^j, where j is the number of steps they have taken. Then the i-th round stops, because we only consider the event that they *first* meet. The final score is obtained by dividing the sum of the scores from all rounds by n. Algorithm 12 of Appendix D.1 describes the details of `compSR`.

Since `compLowerBound` has to invoke `compSR` for $|Q|$ times, it is quite expensive. Our next algorithm can reduce the number of times that `compSR` is called.

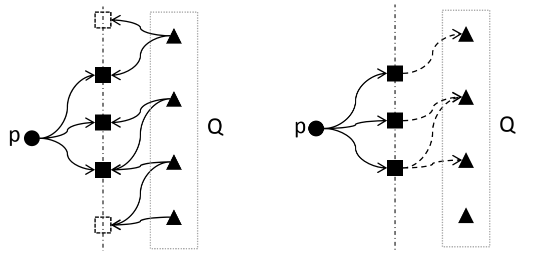
5.2 Design of `compLB-Advanced`

The `compLB-Advanced` is an alternative to `compLowerBound`. It enables pruning during the score computation process. Besides the parameters of `compLowerBound`, it requires the min-heap R, defined in Section 3.2, for keeping the current top-k pairs. It first computes an upper-bound of $\mathcal{S}_z(p,q)$, denoted by $\mathcal{S}_z^+(p,q)$ for every $q \in Q$. Then the pairs $(\mathcal{S}_z^+(p,q), q)$ are pushed to a max-heap H. These entries are popped in descending order of their $\mathcal{S}_z^+(p,q)$ values. Let T_k be the k-th largest score in R. When an entry is popped from H, if $\mathcal{S}_z^+(p,q) > T_k$, we compute $S_d(p,q)$, record it in $c[q]$, and use it to update R; otherwise, the algorithm stops. At this point, the nodes in H are pruned, and c is returned. Although the values of the pruned nodes in c are not computed (they are initialized as zero), they are not included in the top-k result, and so the algorithm is still correct. Algorithm 13 of Appendix D.2 describes the detail of `compLB-Advanced`.

We now present the formula for $\mathcal{S}_z^+(p,q)$, and discuss how to derive it efficiently.

THEOREM 2. *Let $M_i(p,q)$ be the probability that two random surfers at p and q meet each other at the i-th step. Then, $\mathcal{S}_z(p,q) \le \mathcal{S}_z^+(p,q)$, where*

$$\mathcal{S}_z^+(p,q) = \sum_{i=1}^{d} M_i(p,q) + X_d^+ \tag{18}$$



(a) Basic; (b) Backward random walk

Figure 3: Deriving $M_i(p,q)$ for every $q \in Q$.

PROOF. (Sketch) We first prove the following:

LEMMA 7. *$F_i(p,q) \le M_i(p,q)$ for any $p \in P$ and $q \in Q$*

By using Lemma 7, we can then prove that $\mathcal{S}_z(p,q) \le \mathcal{S}_z^+(p,q)$. The detailed proof can be found in Appendix D.3. □

Deriving $\mathcal{S}_z^+(p,q)$ by backward random walk. To evaluate $\mathcal{S}_z^+(p,q)$ (Equation 18), we need to first compute the $M_i(p,q)$ values. To obtain $M_i(p,q)$, we can perform an i-step random walk from q. Since `compLB-Advanced` requires finding $\mathcal{S}_z^+(p,q)$ for every $q \in Q$, $|Q|$ random walk operations need to be performed, incurring a high cost of $O(d|Q||E|)$. This process is illustrated in Figure 3(a), where we enumerate the nodes in Q for random walks, in order to meet the surfer from p (at the black squares). This solution is detailed in Appendix D.2.

We now propose a faster technique of deriving $\mathcal{S}_z^+(p,q)$ for every $q \in Q$. At the i-th step, we use a vector r to store the distribution of the surfer from p. Instead of performing random walk from the nodes in Q, we use another vector ur to perform i steps of random walks backwards from r, in order to touch the nodes in Q. Then, $ur[q]$ stores the value $M_i(p,q)$ for every $q \in Q$. This idea is shown in Figure 3(b), where we avoid enumerating the nodes in Q for performing random walk in Figure 3(a). The routine used in PPR (i.e., Algorithm 10, Appendix C.3) can be used here. The details are described in Steps 4-9 in Algorithm 13 of Appendix D.2. The algorithm finishes in $O(d^2|E|)$ time.

6. EXPERIMENTAL EVALUATION

We test our results on three datasets: *Yeast*, *Coauthor* and *Cora*, which have 2.36k, 188k, and 37k nodes respectively. These graphs have different properties; for instance, *Yeast* is undirected and unweighted, whereas *Coauthor* is directed and unweighted. The default values of our parameters are: $k = 50$, $\lambda = 0.2$, and $\varepsilon = 10^{-6}$. The default value of z, equal to 8, is derived from Equation 5. The details of the datasets and the setup can be found in Appendix E. Our source codes are also available[3]. We next present the results for PPR and SR in Sections 6.1 and 6.2 respectively.

6.1 Results on the PPR

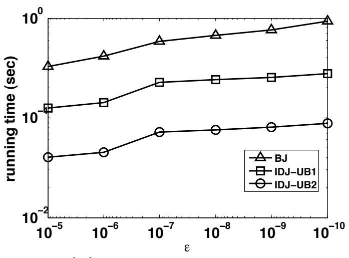
We have tested 3 methods on *Yeast* and *Coauthor*:

BJ: Use the basic solution (Section 2.2), where Algorithm 5, with $d = z$, is used to compute the PPR score.

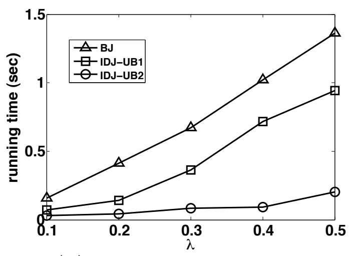
IDJ-UB1: Use IDJ with `compUpperBound` in Section 3.2.

IDJ-UB2: Use IDJ with `compUpperBound` in Section 4.2.

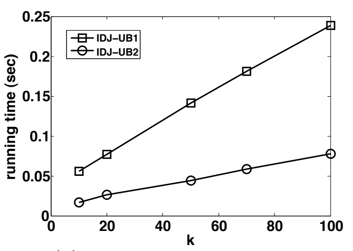
[3]`http://www.cs.hku.hk/~lwsun/codes/vldb11/`



(a) Running time vs. ε

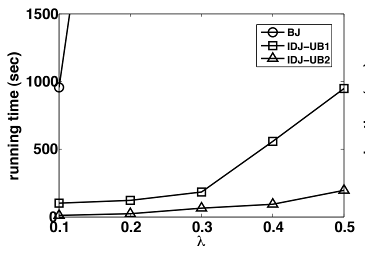


(b) Running time vs. λ

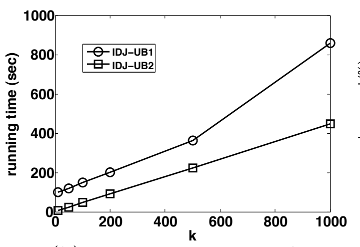


(c) Running time vs. k

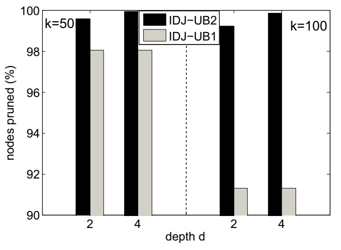
Figure 4: Results on Yeast for PPR.



(a) Running time vs. λ



(b) Running time vs. k



(c) Performance Analysis

Figure 5: Results on Coauthor for PPR.

6.1.1 The Yeast Dataset

Effect of ε (Figure 4(a)). As ε, the tolerance, decreases, the score needs to be more accurate. A higher depth z is then required. Hence, the running times of all three methods increase with ε. Note that IDJ-UB2 is better IDJ-UB1, which is better than BJ. In particular, IDJ-UB2 consistently outperforms BJ by more than an order of magnitude.

Effect of λ (Figure 4(b)). The decay factor, λ, determines the convergent rate of the e_d-function. A larger λ makes the e_d-function converges slower, and also needs a larger z. Moreover, when λ is large, the upper bound value X_d^+ (Equation 10) is larger. Thus, the upper bound is less tight, and the pruning becomes less effective. Thus, the running times of BJ and IDJ-UB1 increase with λ. Compared to IDJ-UB1, IDJ-UB2 is less sensitive to λ , since it uses $Y_d^+(p, Q)$ to compute upper bounds.

Effect of k (Figure 4(c)). The running time of BJ is almost the same for different k values (0.41 seconds), since it has to compute the scores for all (p, q) pairs. The evaluation times of IDJ-UB1 and IDJ-UB2 increase with k, because (1) we have to compute $\mathcal{S}_z(p, q)$ for all the top-k pairs; and (2) when k is large, the value of T_k becomes smaller, rendering a lower pruning effectiveness (Section 3.1). Observe that IDJ-UB2 outperforms IDJ-UB1; at $k = 50$, IDJ-UB2 is 70% faster than IDJ-UB1. This is because IDJ-UB2 uses tighter bounds, and is less sensitive to the increase in T_k.

6.1.2 The Coauthor Dataset

Since *Coauthor* is much larger than *Yeast*, computing the PPR scores is more expensive. We first study the effect of λ on running times, in Figure 5(a). The trends are similar to Figure 4(b). IDJ-UB1 and IDJ-UB2 are much faster than BJ, because they can prune many expensive computations in early rounds. For instance, at $\lambda = 0.1$, they outperform BJ by more than an order of magnitude. Figure 5(b) compares the methods for different k values. BJ, which takes more than an hour to complete, is not shown here. Similar to Figure 4(c), IDJ-UB2 grows slower than IDJ-UB1 with respect to k. It also outperforms IDJ-UB1. At $k = 1000$, it is 50% quicker than IDJ-UB1.

Further analysis. To understand why IDJ-UB2 is better than IDJ-UB1, we examine their pruning effectiveness in each round of IDJ. Since $z = 8$, two iterations of `join-bound` are run, using depth values 2 and 4. Figure 5(c) shows that the fraction of nodes pruned in these two iterations, with $k = 50$ (left) and $k = 100$ (right). Notice that IDJ-UB2 prunes more than 99% nodes from P in the first round. Since this iteration is the cheapest in the IDJ, IDJ-UB2 is highly efficient. The effectiveness of IDJ-UB1 is more sensitive to k, which decreases from 98% to 91% as k increases from 50 to 100. To conclude that the IDJ significantly outperforms BJ. The customized IDJ for PPR (IDJ-UB2) is also faster than the generic IDJ (IDJ-UB1).

6.2 Results on the SR

We have tested 3 methods on *Yeast* and *Cora*:

BJ: Use the basic solution (Section 2.2), where `compSR`, with $d = z$, is used to compute the SR score.

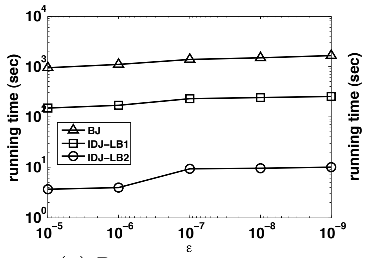
IDJ-LB1: Use IDJ with `compLowerBound` in Section 5.1.

IDJ-LB2: Use IDJ with `compLB-Advanced` in Section 5.2.

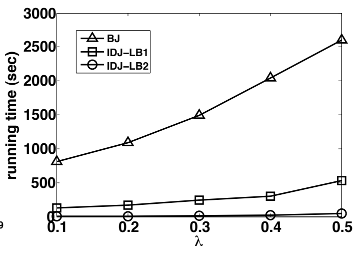
In all these methods, we use 10^5 samples to execute `compSR`.

The Yeast. Figure 6(a) shows that the running times of all methods increase with ε. Also, IDJ-LB2 consistently outperforms BJ by two orders of magnitudes. The execution times of these methods are longer than those in Figure 4(a), as in general, SR is more costly than PPR. In Figure 6(b), as λ increases, z also increases, and so all the methods are slower. Both IDJ-LB1 and IDJ-LB2 outperform BJ significantly. We also tested the effect of k in Figure 6. The running time of BJ, which is around 1100 seconds for different k values, is not shown. Notice that IDJ-LB2 improves IDJ-LB1 by an order of magnitude.

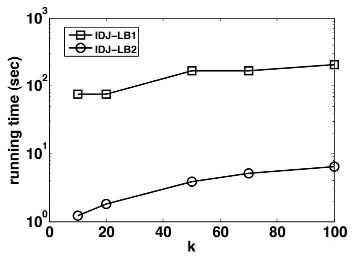
The Cora is a much larger dataset than the *Yeast*. The running time of BJ, which is more than 10 hours, is not shown here. Figures 7(a) and (b), which test the effect of λ and k, show similar trends as Figures 6(b) and (c). Figure 7(c) compares the effectiveness of IDJ-LB1 and IDJ-LB2,



(a) Running time vs. ε

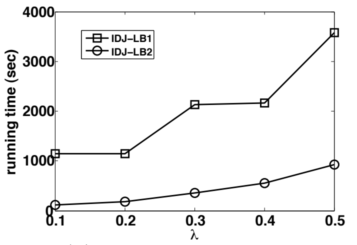


(b) Running time vs. λ

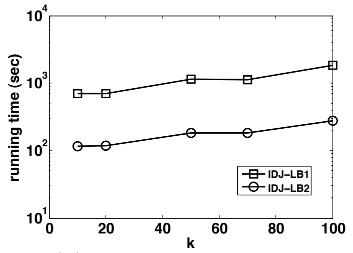


(c) Running time vs. k

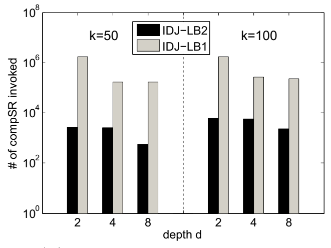
Figure 6: Results on Yeast for SR.



(a) Running time vs. λ



(b) Running time vs. k



(c) Performance Analysis

Figure 7: Results on Cora for SR.

in terms of the logarithm of the number of times `compSR` is run at each iteration. Recall that IDJ-LB2 is designed to avoid calling `compSR`. For both $k = 50$ (left) and $k = 100$ (right), IDJ-LB2 can prune many (over 10^6) `compSR` computations, by paying a relatively small cost of upper-bound computation. This explains why the effectiveness of IDJ-LB2 is an order of magnitude better than IDJ-LB1. We conclude that the IDJ is better than BJ. The customized IDJ for SR (IDJ-LB2) also outperforms the generic IDJ (IDJ-LB1).

7. CONCLUSIONS

In this paper, we propose the LS-join query, which supports many graph applications, including customer suggestion, citation analysis, and link prediction. To evaluate the LS-join, we propose the IDJ algorithm, which can be used on a class of link-based measures. We also enhance the performance of the IDJ for the PPR and the SR, which are common similarity measures. Our algorithms, which can be used on weighted and directed graphs, perform much better than basic solutions on large graphs. In the future, we will extend the LS-join to consider other features in a graph, for example, the contents of a social network user's blog. We will also study how to customize the IDJ for other common e-functions, for instance, the DHT.

8. REFERENCES

[1] Z. Abbassi and V. S. Mirrokni. A recommender system based on local random walks and spectral methods. In *WebKDD/SNA-KDD*, pages 102–108, 2007.
[2] A. Arasu, V. Ganti, and R. Kaushik. Efficient exact set-similarity joins. In *VLDB*, pages 918–929, 2006.
[3] C. Boehm *et al.* Epsilon grid order: An algorithm for the similarity join on massive high-dimensional data. In *SIGMOD*, pages 379–388, 2001.
[4] C. Li *et al.* Fast computation of SimRank for static and dynamic information networks. In *EDBT*, pages 465–476, 2010.
[5] C. Xiao *et al.* Efficient similarity joins for near duplicate detection. In *WWW*, pages 131–140, 2008.
[6] D. Fogaras *et al.* Towards scaling fully personalized pagerank: Algorithms, lower bounds, and experiments. *Internet Math.*, 2(3), 2005.
[7] D. Liben-Nowell *et al.* The link prediction problem for social networks. In *CIKM*, pages 556–559, 2003.
[8] D. Fogaras and B. Rácz. Scaling link-based similarity search. In *WWW*, pages 641–650, 2005.
[9] G. Jeh *et al.* SimRank: a measure of structural-context similarity. In *KDD*, pages 538–543, 2002.
[10] W. Hoeffding. Probability inequalities for sums of bounded random variables. In *Journal of the American Statistical Association*, volume 58, pages 13–30, 1963.
[11] J. Dittrich *et al.* GESS: a scalable similarity-join algorithm for mining large data sets in high dimensional spaces. In *KDD*, pages 47–56, 2001.
[12] J. Sankaranarayanan *et al.* Distance join queries on spatial networks. In *GIS*, pages 211–218, 2006.
[13] G. Jeh and J. Widom. Scaling personalized web search. In *WWW*, pages 271–279, 2003.
[14] J. M. Kleinberg. The small-world phenomenon: an algorithm perspective. In *STOC*, pages 163–170, 2000.
[15] L. Zou *et al.* Distance-join: pattern match query in a large graph database. *PVLDB*, 2(1):886–897, 2009.
[16] D. Lizorkin, P. Velikhov, M. Grinev, and D. Turdakov. Accuracy estimate and optimization techniques for simrank computation. *VLDBJ*, 19:45–66, 2010.
[17] P. Li *et al.* Fast single-pair simrank computation. In *SDM*, pages 571–582, 2010.
[18] S. J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach.* 2003.
[19] P. Sarkar and A. Moore. Fast nearest neighbor search in disk-resident graphs. In *KDD*, pages 513–522, 2010.
[20] X. Yin *et al.* LinkClus: efficient clustering via heterogeneous semantic links. In *VLDB*, pages 427–438, 2006.
[21] Y. Zhou *et al.* Graph clustering based on structural/attribute similarities. *PVLDB*, 2(1):718–729, 2009.
[22] Z. Gyöngyi *et al.* Combating web spam with trustrank. In *VLDB*, pages 576–587, 2004.

APPENDIX

A. LEMMA PROOFS FOR SECTION 2.3

Lemma 2 Proof. We first rewrite $\mathcal{S}(u,v)$ as:

$$\mathcal{S}(u,v) = \mathcal{S}_d(u,v) + X_d(u,v) \tag{19}$$

where

$$X_d(u,v) = a \sum_{i=d+1}^{\infty} \lambda^i \mathcal{P}_i(u,v) \tag{20}$$

We now prove the 2 parts of the lemma:

(Part 1) $\mathcal{S}(u,v) \geq \mathcal{S}_d(u,v)$: Since $\mathcal{P}_i$ is a probability value, $0 \leq \mathcal{P}_i \leq 1$, and $X_d(u,v) \geq 0$. By using Equation 19, we can see that this part is correct.

(Part 2) $\mathcal{S}(u,v) \leq \mathcal{S}_d(u,v) + X_d^+$. We first show that $X_d(u,v) \leq X_d^+$:

$$\begin{aligned} X_d(u,v) &\leq a \sum_{i=d+1}^{\infty} \lambda^i && (\text{as } \mathcal{P}_i(u,v) \leq 1) \\ &= a \cdot \frac{\lambda^{d+1}}{(1-\lambda)} && (\text{as } \lambda < 1) \\ &= X_d^+ \end{aligned}$$

Using Equation 19, this part holds.

Since Parts (1) and (2) are correct, the lemma is true.

Lemma 3 Proof. From Lemma 2, we know that

$$\mathcal{S}(u,v) \in [\mathcal{S}_z(u,v),\ \mathcal{S}_z(u,v) + X_z^+]$$

For Inequality 3 to hold, we need $X_z^+ \leq \varepsilon$. We can then obtain Inequality 5, and so the lemma is correct.

B. THE IDJ ALGORITHM (SECTION 3)

Algorithm 3: `updateResult` (Section 3.2)

```
  Input: current heap $R$, score $sc$, pair $(p,q)$, size $k$
  Output: updated heap $R$
1 if $sc < R.minEntry().key$ then
2     return $R$;
3 $R.push(Entry(sc,(p,q)))$; //Entry(key, content)
4 if $R.size() > k$ then
5     $R.popMin()$;
6 return $R$;
```

Algorithm 4: `join-refine` (Section 3.2)

```
  Input: graph $G(V,E)$; node sets $P, Q$; size $k$; depth $z$
  Output: top-$k$ pairs $R$
1 //$R$ is a MinHeap of fixed-size $k$;
2 $R \leftarrow \emptyset$;
3 for each $p \in P$ do
4     $c \leftarrow$ compLowerBound($G,P,Q,k,z,p$);
5     for each $q \in Q$ do
6         $R \leftarrow$ updateResult($R, c[q], (p,q), k$);
7 return $R$;
```

The **design of IDJ** is inspired by the classical strategy of *iterative-deepening depth first search* (or *IDDFS*), which is used in state-space search [18]. Notice that other methods like *BFS* (breadth-first-search) and *DFS* (depth-first search) can also be used in state-space search. In *IDDFS*, we perform depth-first search within a constrained scope first. If the goal is not found, we start over the search with an expanded scope. Compared to *DFS*, *IDDFS* avoids the high cost of digging too deep in early rounds. Compared to *BFS*, *IDDFS* avoids the high cost of storing all the intermediate states. Using the same spirit of *IDDFS*, we do not fully compute the exact $S_z(p,q)$ scores in early rounds (as in *DFS*), since they are expensive to compute. Also, we do not store the intermediate scores for later iterations (as in *BFS*), as they incur a high space cost. However, the scores computed in early rounds can effectively prune nodes from the joining sets.

C. THE PPR ALGORITHM (SECTION 4)

We first present the pseudocodes of `compLowerBound` in Section C.1. We then present the proofs of lemmas used by `compUpperBound` in Section C.2. The algorithm pseudocodes of `compUpperBound` is shown in Section C.3.

C.1 Algorithms for `compLowerBound`

Algorithm 5: `compLowerBound` (for PPR)

```
  Input: graph $G(V,E)$; depth $d$; node $p$
  Output: score vector $c$ of $\mathcal{S}_z$ values
1 Set $r \leftarrow \{0,0,\ldots,0\}$, $c \leftarrow \{0,0,\ldots,0\}$;
2 $r[p] \leftarrow 1$;
3 for $i \leftarrow 1$ to $d$ do
4     $r \leftarrow$ walkOneStep($G$,$r$);
5     $c \leftarrow$ addScore-PPR($r$,$c$,$\lambda$,$i$);
6 return $c$;
```

Algorithm 6: `walkOneStep`

```
  Input: Graph $G(V,E)$; current vector $r$
  Output: next-step vector $r'$
1 Set $r' \leftarrow \{0,0,\ldots,0\}$;
2 for each $u \in V$ do
3     if $r[u] > 0$ then
4         for each $v \in O(u)$ do
5             $r'[v] \leftarrow r'[v] + r[u] \cdot p_{out}(u,v)$;
6 return $r'$;
```

Algorithm 7: `addScore-PPR`

```
  Input: current vector $r, c$; decay factor $\lambda$; step $i$
  Output: updated $c$
1 for each $v \in V$ do
2     $c[v] \leftarrow c[v] + (1-\lambda)\lambda^i \times r[v]$;
3 return $c$;
```

Algorithm 5 describes how to compute c. It initializes all entries of c to be zero. Note that the sum of all entries in r, where $r[v] = V_{i-1}(p,v)$ for $v \in V$, is always equal to 1. Also, $r[p] = 1$ and $r[v] = 0$ for $v \neq p$. For every step i, it invokes Algorithm 6, which refreshes r by performing one step of random walk, based on Equation 9. The vector c is used to accumulate the r values, using Algorithm 7. After d steps of random walks, c stores the value of $\mathcal{S}_d(p,q)$.

Algorithm 6 runs with a time complexity of $O(|E|)$. Subsequently, Algorithm 5 needs $O(d \cdot |E|)$ time and $O(|V|)$ space. We remark that the cost of Algorithm 6 is much smaller when i is small, say $i < 3$, since most entries in r are zero. This property makes Algorithm 5 run very fast in the first few iterations.

C.2 Lemma Proofs for `compUpperBound`

The following are the proofs of lemmas used to support Theorem 1.

Proof of Lemma 4. First, observe that for any $q \in Q$, $V_i(p,q) \leq V_i(p,Q)$. This is because at the i-th step of random walk from p, the event A that the surfer visits *some* node in Q subsumes the event B that it visits a specified node $q \in Q$. Since $V_i(p,Q)$ and $V_i(p,q)$ correspond to the probabilities that event A and event B happen respectively, we have $V_i(p,Q) \geq V_i(p,q)$, for any specified $q \in Q$. Then,

$$\begin{aligned}
\mathcal{S}_z(p,q) &= \mathcal{S}_d(p,q) + (1-\lambda)\sum_{i=d+1}^{z}\lambda^i V_i(p,q) \\
&\leq \mathcal{S}_d(p,q) + (1-\lambda)\sum_{i=d+1}^{z}\lambda^i V_i(p,Q) \\
&= \mathcal{S}_d(p,q) + Y_d(p,Q)
\end{aligned}$$

Hence, the lemma holds.

Proof of Lemma 5. First, we show that $V_{d+i}(p,Q)$ can be calculated as:

$$V_{d+i}(p,Q) = \sum_{v \in V} V_d(p,v) \cdot V_i(v,Q) \qquad (21)$$

To compute $V_{d+i}(p,Q)$, instead of using $d{+}i$ steps of random walks, in Equation 21, we only perform random walks for d steps and obtain $V_d(p,v)$. For each $v \in V$, we multiply $V_d(p,v)$ by $V_i(v,Q)$, which is the probability of visiting some node in Q if we keep walking i steps from v. Finally, we can add up the products at each v to get $V_{d+i}(p,Q)$, since these probabilities are disjoint from each other.

Given Equation 21, we now prove this lemma by expanding the RHS of the equation:

$$\begin{aligned}
Y_d^+(p,Q) &= \lambda^d \sum_{v\in V} V_d(p,v)\cdot Y_0^+(v,Q) \\
&= \lambda^d \sum_{v\in V} V_d(p,v)\sum_{i=1}^{z}(1-\lambda)\lambda^i V_i(v,Q) \\
&= \lambda^d(1-\lambda)\sum_{i=1}^{z}\lambda^i\sum_{v\in V} V_d(p,v)V_i(v,Q) \\
&= (1-\lambda)\sum_{i=1}^{z}\lambda^{d+i}\sum_{v\in V} V_d(p,v)V_i(v,Q) \\
&= (1-\lambda)\sum_{i=1}^{z}\lambda^{d+i}V_{d+i}(p,Q) \qquad \text{(Equation 21)} \\
&= (1-\lambda)\sum_{i=d+1}^{z+d}\lambda^{i}V_{i}(p,Q) \\
&= Y_d(p,Q) + (1-\lambda)\sum_{i=z+1}^{z+d}\lambda^{i}V_{i}(p,Q)
\end{aligned}$$

Since

$$0 \leq (1-\lambda)\sum_{i=z+1}^{z+d}\lambda^i V_i(p,Q) \leq \varepsilon$$

the two sides of the inequality hold.

Proof of Lemma 6. If we refer to the proof of Lemma 2 in Appendix A, X_d^+ is an upper bound since $V_i(p,q) \leq 1$. On the other hand, we obtain $Y_d(p,Q)$ as an upper-bound because $V_i(p,q) \leq V_i(p,Q)$. Since $V_i(p,Q)$ is probability value and thus always no larger than 1, we have $Y_d(p,Q) \leq X_d^+$ for any $p \in Q$. Thus, the lemma holds.

C.3 Algorithms for `compUpperBound`

The new `compUpperBound` algorithm for PPR is shown in Algorithm 8. The new `init` algorithm, which precomputes the values of $Y_0(v,Q)$ for every $v \in V$, is presented in Algorithm 9. It makes use of a backward random walk procedure, called `walkOneStepBackwards`, as illustrated in Algorithm 10.

Algorithm 8: `compUpperBound` (for PPR)

```
  Input: vector $r, c, y0$, depth $d$
  Output: upper bound of $\mathcal{S}$ ($\mathcal{S}_z^+$)
1 $s \leftarrow$ MAX$\{c[q] \mid q \in Q\}$; // Compute Equation 10
2 for each $v \in V$ do
3     $s \leftarrow s + \lambda^d \cdot r[v] \cdot y0[v]$; // Use Equation 11
4 return $s$;
```

Algorithm 9: `init` (for PPR)

```
  Input: graph $G$; joining set $Q$; depth $z$;
  Output: upper-bound vector $y0$
1 Set $rb \leftarrow \{0,0,\ldots,0\}$, $y0 \leftarrow \{0,0,\ldots,0\}$;
2 for each $q \in Q$ do
3     $rb[q] \leftarrow 1$;
4 for $i \leftarrow 1$ to $z$ do
5     $rb \leftarrow$ walkOneStepBackwards($G$,$rb$);
6     $y0 \leftarrow$ addScore-PPR($rb$,$y0$,$\lambda$,$i$)
7 return $y0$;
```

Algorithm 10: `walkOneStepBackwards`

```
  Input: graph $G(V,E)$; vector $rb$
  Output: next-step vector $rb'$
1 $rb' \leftarrow \{0,0,\ldots,0\}$;
2 for each $v \in V$ do
3     if $r[v] > 0$ then
4         for each $u \in I(v)$ do
5             $rb'[u] \leftarrow rb'[u] + rb[v] \cdot p_{out}(u,v)$;
6 return $rb'$;
```

D. THE SR ALGORITHM (SECTION 5)

We first present the algorithms used by `compLowerBound`, in Secton D.1. In Sections D.2 and D.3, we discuss the algorithms and proofs used by `compLB-Advanced`.

D.1 Algorithms for `compLowerBound`

Algorithm 11 presents the `compLowerBound` for SR. It invokes a sampling method of evaluating the SR score for a given pair of nodes, as shown in Algorithm 12.

Details of `compSR`. Algorithm 12 describes the sampling-based method for evaluating the SR score for a given pair of nodes. The algorithm executes n iterations, where n depends on the sampling accuracy and will be discussed later. Let us discuss how to obtain a score in the i-th round: We simulate the motion of two surfers at node p and q. Particularly, at each step, the two surfers randomly choose their

Algorithm 13: compLB-Advanced (for SR)

```
Input: graph $G(V, E)$; joining set $Q$; result size $k$; depth $d$;
       node $p$; top-$k$ pairs $R$
Output: score vector $c$
$c \leftarrow \{0, 0, \cdots, 0\}$;
$r \leftarrow \{0, 0, \ldots, 0\}$; $uc \leftarrow \{0, 0, \ldots, 0\}$;
$r[p] \leftarrow 1$;
for $i \leftarrow 1$ to $d$ do
    $r \leftarrow$ walkOneStep($r$,$G$);
    $ur \leftarrow r$;
    for $j \leftarrow 1$ to $i$ do
        $ur \leftarrow$ walkOneStepBackwards($ur$, $G$);
    $uc \leftarrow$ addScore-SR ($ur, uc, \lambda, i$);
MaxHeap $H$;
for each $q \in Q$ do
    $H.push(Entry(uc[q] + X_d^+, q))$; //$Entry(key, content)$
while $H.notEmpty()$ do
    $s \leftarrow H.maxEntry().key$;
    $q \leftarrow H.maxEntry().content$;
    $H.popMax()$;
    if $s < R.minEntry().key$ then
        break;
    $c[q] \leftarrow$ compSR($G, p, q, d$);
    $R \leftarrow$ updateResult($R, s, (p, q), k$);
return $c$;
```

Algorithm 11: compLowerBound (for SR)

```
Input: graph $G(V, E)$; joining set $Q$; result size $k$; depth $d$;
       node $p$
Output: score vector $c$
for each $q \in Q$ do
    $c[q] \leftarrow$ compSR($G, p, q, d, \lambda$);
return $c$;
```

Algorithm 12: compSR

```
Input: graph $G(V, E)$; node $p, q$; depth $d$; decay factor $\lambda$
Output: score $s$
$s \leftarrow 0$;
for $i \leftarrow 1$ to $n$ do
    $p' \leftarrow p$, $q' \leftarrow q$;
    for $j \leftarrow 1$ to $d$ do
        $p' \leftarrow$ randomly select a node from $O(p')$;
        $q' \leftarrow$ randomly select a node from $O(q')$;
        if $p' = q'$ then
            $s \leftarrow s + \lambda^j$;
            break;
return $\frac{s}{n}$;
```

out-neighbor to move forward. When the two surfers arrive at the same node, we obtain a score of λ^j, where j is the number of steps they have taken. Then, the i-th round stops, as we only consider the event that they *first* meet. If they did not meet each other after d steps, the round stops with a zero score. The final score is obtained by dividing the sum of the scores from all rounds by n.

Correctness of compSR. Let $\hat{p}$ be the approximate score and p be the exact score. According to the Hoeffding's Inequality [10], we have:

$$Pr(|\hat{p} - p| \geq \alpha) \leq 2e^{-2n\alpha^2}$$

where n is the number of samples, and α is the error between the approximate value and true value.

D.2 Algorithms for compLB-Advanced

Algorithm 13 describes compLB-Advanced, which is an enhanced version of compLowerBound. It incorporates novel pruning techniques that avoids handling some nodes that cannot be in the top-k result.

A Basic Solution for finding $\mathcal{S}_z^+(p, q)$. The algorithm of compLB-Advanced requires the computation of $\mathcal{S}_z^+(p, q)$. One simple way to evaluate $\mathcal{S}_z^+(p, q)$ (Equation 18) is to first compute the $M_i(p, q)$ values. They can be obtained by considering two random surfers from p and q. Particularly, let rp and rq be two vectors that store respectively the distribution of the surfers from p and q at the i-th step. Since the surfers are independent, $M_i(p, q) = \sum_{v \in V} rp[v] \times rq[v]$. However, this can incur a high cost, since compLB-Advanced requires the finding of $\mathcal{S}_z^+(p, q)$ for every $q \in Q$. Consequently, we need to consider the random walk from each $q \in Q$, with a cost of $O(d|Q||E|)$.

D.3 Proofs for compLB-Advanced

Lemma 7. We consider two random events: at the i-th step of two random surfers, event A is that the two surfer "meet for the first time", and event B is that they "meet". If event A happens, event B must also happen. Since $F_i(p, q)$ and $M_i(p, q)$ correspond to the probabilities that event A and event B happen respectively, we have $F_i(p, q) \leq M_i(p, q)$.

Theorem 2 can now be proved as follows:

$$\begin{aligned} \mathcal{S}_z(p, q) &= \mathcal{S}_d(p, q) + X_d(p, q) \\ &= \sum_{i=1}^{d} \lambda^i F_i(p, q) + X_d(p, q) \\ &\leq \sum_{i=1}^{d} \lambda^i M_i(p, q) + X_d(p, q) \\ &\leq \sum_{i=1}^{d} \lambda^i M_i(p, q) + X_d^+ \\ &= \mathcal{S}_z^+(p, q) \end{aligned}$$

E. EXPERIMENT SETUP

We test our results on three graphs of different sizes: *Yeast*, *Coauthor* and *Cora*. The *Yeast* dataset is relatively small, on which we can finish all the experiments with basic solutions. We use *Coauthor* and *Cora* as large graphs to mainly test the scalability of our solutions, on which the basic solutions often cannot be finished within a feasible time. The default values of our parameters are: $k = 50$, $\lambda = 0.2$, and $\varepsilon = 10^{-6}$. The value of z is derived from Equation 5. All the experiments were carried out on the Windows XP operating system, on a machine with a 2.66 GHz Intel Core Duo processor and 2GB memory. The programs were written in C++ and compiled on Microsoft Visual Studio 2005. Our source codes are also available at http://www.cs.hku.hk/~lwsun/codes/vldb11/.

Datasets. The properties of the datasets used can be found in Table 2. Let us now describe their details.

• **Yeast**[4] is a protein-protein interaction graph, where each node denotes a protein and an edge denotes the interaction between two proteins. The graph is *undirected* and *unweighted.* The dataset contains the PIN class information,

[4]http://vlado.fmf.uni-lj.si/pub/networks/data/

Table 2: Datasets (D=directed, UD=undirected; W=weighted, UW=unweighted)

Dataset	Type	$\|V\|$	$\|E\|$	$\|P\|$	$\|Q\|$
Yeast	UD, UW	$2.36k$	$7.18k$	283	586
Coauthor	UD, W	$188k$	$1140k$	$12.06k$	$28.16k$
Cora	D, UW	$37k$	$710k$	$1.35k$	$4.61k$

where the proteins are partitioned into 13 clusters. We pick two largest partitions (encoded as 3-U and 8-D respectively) as the joining sets.

• **Coauthor** is exacted from the DBLP record[5]. A node represents an author, and an edge (u, v) indicates that authors u and v co-authored a paper. We construct this graph from the DBLP record as follows. We first find all the conference papers, and for each paper, we add edges between its authors. For each edge (u, v), we assign the weight $w(u, v)$ as the number of papers that u and v co-authored. This graph is *undirected* and *weighted*. We identify the authors as the two joining sets from the *database* (DB) and *artificial intelligence* (AI) areas respectively. We consider an author as in the DB area if he/she has published papers in a DB conference (e.g., VLDB).

• **Cora**[6] is a citation graph. A node denotes a scientific paper and an edge from u to v indicates that paper u cites paper v. The graph is *directed* and *unweighted*. The dataset contains the labels of research areas for the papers. We identify two sets of paper labeled as *database* and *machine learning* as our joining sets.

F. ACKNOWLEDGEMENTS

Reynold Cheng, Liwen Sun, and Xiang Li were supported by the Research Grants Council of Hong Kong (RGC Project HKU 711309E). We would like to thank the reviewers for their insightful comments.

[5]`http://www.informatik.uni-trier.de/~ley/db/`

[6]`http://www.cs.umass.edu/~mccallum/data.html`