Finding Information Nebula over Large Networks

Lijun Chang, Jeffrey Xu Yu, Lu Qin, Yuanyuan Zhu
The Chinese University of Hong Kong
Hong Kong, China
{ljchang,yu,lqin,yyzhu}@se.cuhk.edu.hk

Haixun Wang
Microsoft Research Asia
Beijing, China
haixunw@microsoft.com

ABSTRACT

Social and information networks have been extensively studied over years. In this paper, we concentrate ourselves on a large information network that is composed of entities and relationships, where entities are associated with sets of keyword terms (kterms) to specify what they are, and relationships describe the link structure among entities which can be very complex. Our work is motivated but is different from the existing works that find a best subgraph to describe how user-specified entities are connected. We compute information nebula (cloud) which is a set of top-K kterms P that are most correlated to a set of user-specified kterms Q, over a large information network. Our goal is to find how kterms are correlated given the complex information network among entities. The information nebula computing requests us to take all possible kterms into consideration for the top-K kterms selection, and needs to measure the similarity between kterms by considering all possible subgraphs that connect them instead of the best single one. In this work, we compute information nebula using a global structural-context similarity, and our similarity measure is independent of connection subgraphs. To the best of our knowledge, among the link-based similarity methods, none of the existing work considers similarity between two sets of nodes or two kterms. We propose new algorithms to find top-K kterms P for a given set of kterms Q based on the global structural-context similarity, without computing all the similarity scores of kterms in the large information network. We performed extensive performance studies using large real datasets, and confirmed the effectiveness and efficiency of our approach.

Categories and Subject Descriptors

H.3 [**Information Storage and Retrieval**]: Miscellaneous

General Terms

Design, Management, Performance

1. INTRODUCTION

Social and information systems analysis plays an important role in modern knowledge discovery [12, 16, 14] to extract information and/or discover knowledge from large information networks [10]. In an information network, the two primary concepts are entities and relationships, where an entity is associated with information to describe the entity in a form of label, attributes, or a short description with a set of terms, and a relationship (or link) shows how two entities are related and the relationships among the entities can be very complex. Examples of information networks include the publication network [17], knowledge base [24], social networks [9], etc. In the literature, the existing work over information networks includes clustering, ranking, classification, data quality and search, as discussed in [10].

In this paper, we focus on an information network, $\mathcal{G}$, where entities are associated with a short description in a form of a set of terms, which we call keyword terms or simply kterms in short, that best describe the entities. We study a new problem on identifying a set of top-K kterms, P, that are most correlated to a set of user-specified kterms, Q, for $P \cap Q = \emptyset$, based on a goodness function such that every kterm in P is similar to every kterm in Q and the similarity is measured by a structural similarity over the complicated link structures among the entities in the large information network $\mathcal{G}$.

Our study is motivated by two reported studies [25, 14]. Tong and Faloutsos [25] and Kasneci et al. [14] find an informative connection subgraph (*CSG*) which best explains the relationships among a set of user-specified entities in an information network $\mathcal{G}$. Both the approaches effectively identify such informative connection subgraph following a goodness function. The identified subgraph is the best of many possible subgraphs to explain the relationships among user-specified entities, and the information (e.g. the kterms in our context) associated with entities is simply ignored. In our work, we study the relationships between kterms at the kterm level rather than at the entity level. Instead of finding a single best subgraph to explain the connectivities among entities, we find a set of top-K kterms P by exploring all possible subgraphs among entities that can possibly connect every kterm in P to every kterm in Q. It is worth noting that in [25, 14], the entities are explicitly given to identify the subgraph to explain the set of entities. In our problem, the kterms are given, there exist a large number of entities that may use some of kterms to describe themselves, and such entities may appear anywhere in the large information network $\mathcal{G}$. Our approach can be a complement of [25, 14].

Consider the semantic knowledge base *YAGO* (`http://www.mpi-inf.mpg.de/yago-naga/yago/`), which is a heterogeneous information network [10], an excerpt is shown in Fig. 1. Each node represents an entity, such as, a person, an organization, a country, etc. Based on the link structure among entities, suppose a user wants to know what the best is to connect {"Albert_Einstein", "Walther_Bothe"}. The *CSG* approach returns a subgraph induced

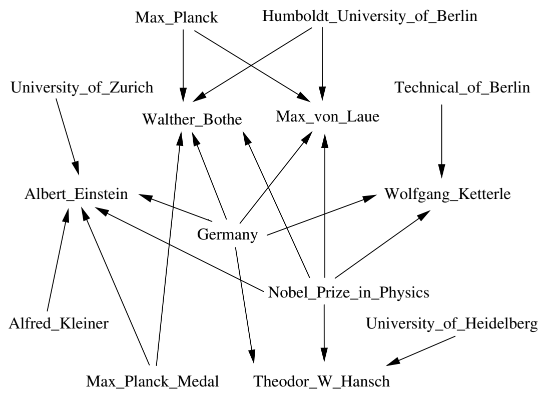


Figure 1: An Excerpt from *YAGO*

by the five nodes, "Albert_Einstein", "Walther_Bothe", "Germany", "Max_Planck_Medal", and "Nobel_Prize_in_Physics", which indicates that both "Albert_Einstein" and "Walther_Bothe" are from "Germany", and they are both Max Planck Medal and Nobel Prize winners. However, suppose a user wants to find similar scientists like $Q = \{$"Albert_Einstein", "Walther_Bothe"$\}$, the *CSG* approach cannot find "Wolfgang_Ketterle" and "Max_von_Laue", since the *CSG* is designed to identify the best subgraph to explain given entities. We consider $\{$"Albert_Einstein", "Walther_Bothe"$\}$ as kterms to describe two entities, and attempt to find the top-K kterms that are similar to $\{$"Albert_Einstein", "Walther_Bothe"$\}$ which include "Wolfgang_Ketterle" and "Max_von_Laue". Note that in this example, since all entities have unique labels (kterms), there are no obvious differences between entities and kterms. In other words, in this example, our goal can be seen as to find most related entities connected to $\{$"Albert_Einstein", "Walther_Bothe"$\}$.

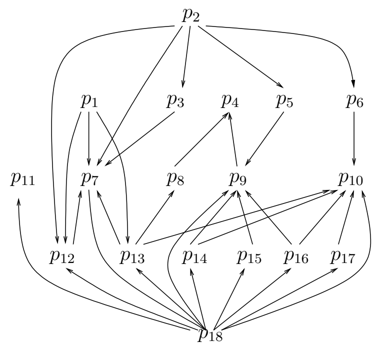
Consider an information network, *DBLP-CP*, which is an induced subgraph of the *DBLP* dataset (`http://www.informatik.uni-trier.de/~ley/db/`). In *DBLP-CP*, a node represents a paper, and each edge reflects a citation relationship between two papers. A subgraph of *DBLP-CP* is shown in Fig. 2, e.g., $p_1, \cdots, p_{18}$ represents 18 papers, and the edge between p_2 and p_3 indicates that paper p_3 is cited by paper p_2. Each node is described using a set of kterms, e.g., "histogram" and "wavelet" as shown in Fig. 2(a), and the node p_{11} contains kterms "selectivity" and "histogram". Suppose a new graduate student wants to study a research topic related to data compression by submitting a query $Q = \{$"histogram", "wavelet"$\}$, which are known as two of the basic methods in data compression, and hopes to identify highly correlated kterms to Q. Under this scenario, each kterm in Q appears in several nodes. The top-3 kterm to be identified are $\{$"selectivity","practicality", "optimality"$\}$. It is worth noting that there are a large number of occurrence of kterms including "selectivity", "practicality", "optimality", as well as others. All such occurrences may be related to $Q = \{$"histogram", "wavelet"$\}$ via possible connections in the underneath information network. It is almost impossible to enumerate all such possible subgraphs with all possible combination of kterms, in order to compute the top-K kterms. For this example, the two approaches [25, 14] cannot be effectively used since the entities are not explicitly specified.

The main contributions of this paper are summarized below. First, we study a new problem of identifying top-K correlated kterms, P, in an information network $\mathcal{G}$ for a set of user-specified kterms Q. All possible kterms exhibited in $\mathcal{G}$ are taken into consideration for the top-K selection. We compute the top-K correlated kterms P regarding Q based on a global structural-context similarity between nodes in $\mathcal{G}$ without enumerating connection subgraphs. It is important to note that our approach is to measure correlated kterms using a global structural-context similarity but is independent of connection subgraphs. Among the link-based similarity measures [11, 13, 23], none of them consider similarity between two sets of nodes like we do in this work. Second, we propose two new top-K algorithms using the framework of the threshold algorithm [4, 5]. We study how to obtain sorted lists in different ways and how to optimize the threshold algorithms. Our algorithms avoid computing all pairs similarities, which is hard in large networks. Third, we propose an algorithm to build index for computing correlation scores, which optimizes the existing algorithm in both index construction time and index size. And we give a block-based index which reduces the I/O cost and memory requirement to hold the index in memory. Finally, we conducted extensive performance studies using large real datasets, and confirmed the effectiveness and efficiency of our approach.

The remainder of the paper is organized as follows. We discuss our problem definition in Section 2. In Section 3, we give a solution overview. We discuss an extended network representation, and a naive algorithm to find correlated kterms for a query Q over extended network. We also give a 2-step solution with two algorithms, called CL-TOPK and IL-TOPK. The CL-TOPK algorithm is given in Section 4. The IL-TOPK is discussed in Section 5. We conducted experimental studies and discuss our findings in Section 6. The related works are discussed in Section 7. We conclude our paper in Section 8.

ID	Labels
p_4	balancing *histogram optimality practicality* $\cdots$
p_7	join *synopses* approximate query answering $\cdots$
p_9	random sampling *histogram* construction $\cdots$
p_{10}	*wavelet histograms selectivity* estimation $\cdots$
p_{11}	*selectivity* estimation compressed *histogram* $\cdots$
p_{12}	*histogram* approximation set-valued $\cdots$
p_{16}	comparison *selectivity* estimators $\cdots$

(a) Labels on Nodes

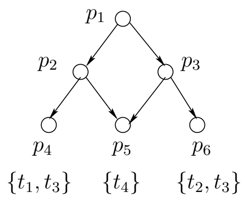


(b) Link Structure

Figure 2: A Subgraph from *DBLP-CP*

2. PROBLEM DEFINITION

We consider an information network, $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{K})$. $\mathcal{K}$ is the corpus of keyword terms (or simply kterms) contained in $\mathcal{G}$, and entities are described by kterms which are a small number yet important terms to describe entities. For a node $v \in \mathcal{V}$, let $\mathcal{K}(v)$ denote the set of kterms associated with v. Then $\mathcal{K} = \cup_{v \in \mathcal{V}} \mathcal{K}(v)$. For a kterm in the corpus, $t \in \mathcal{K}$, let $\mathcal{K}^{-1}(t)$ denote the set of nodes in $\mathcal{G}$ that contain t, i.e., $\mathcal{K}^{-1}()$ is the inverse of $\mathcal{K}()$. Note that, $|\mathcal{K}^{-1}(t)| \geq 1$ for any $t \in \mathcal{K}$, and $|\mathcal{K}^{-1}(t)| = 1$ if t is an entity. For a pair of nodes $u, v \in \mathcal{V}$, there may exist a directed edge from u to v, i.e., $\langle u, v \rangle \in \mathcal{E}$. Let $\mathcal{I}(v)$ denote the set of in-



(a) Sample Network

node pair u, v	$sim(u, v)$
p_2, p_3	0.8
p_4, p_5	0.72
p_4, p_6	0.64
p_5, p_6	0.72

(b) SimRank Values

Figure 3: Information Network

neighbors of v, i.e., $\mathcal{I}(v) = \cup_{\langle u,v\rangle \in \mathcal{E}}\{u\}$, and $\mathcal{O}(v)$ denote the set of out-neighbors of v, i.e., $\mathcal{O}(v) = \cup_{\langle v,u\rangle \in \mathcal{E}}\{u\}$.

Example 2.1: Fig. 3(a) shows an information network $\mathcal{G}(\mathcal{V}, \mathcal{E}, \mathcal{K})$, where $\mathcal{V} = \{p_1, p_2, \cdots, p_6\}$. There are kterms contained in nodes p_4, p_5 and p_6, e.g., $\mathcal{K}(p_4) = \{t_1, t_3\}$, and $\mathcal{K}(p_5) = \{t_4\}$. The kterm corpus is $\mathcal{K} = \{t_1, t_2, t_3, t_4\}$. The set of nodes that contain t_3 is $\mathcal{K}^{-1}(t_3) = \{p_4, p_6\}$. The in-neighbors of p_5 is $\mathcal{I}(p_5) = \{p_2, p_3\}$. The out-neighbors of p_2 is $\mathcal{O}(p_2) = \{p_4, p_5\}$. □

We also consider entity names as one kind of kterms if they are explicitly given. For example, in Fig. 1, the entity names are listed, they are also treated as kterms, i.e., $\mathcal{K} = \{$"Albert_Einstein", "Germany", $\cdots\}$. For *DBLP-CP*, the kterms are explicitly given instead of entity names, i.e., $\mathcal{K} = \{$"histogram", "estimators", $\cdots\}$. In the following, we consider our problem in a kterm level instead of entity level.

Problem Statement: Given an information network $\mathcal{G}(\mathcal{V}, \mathcal{E}, \mathcal{K})$, a query Q consists of l kterms from $\mathcal{K}$, $\{k_1, \cdots, k_l\}$, and l weights corresponding to the kterms, $\{w_1, \cdots, w_l\}$, i.e., $Q = \{(k_1, w_1), \cdots, (k_l, w_l)\}$. Here, a weight is a positive value ≤ 1 and is 1 by default. The problem is to find an information nebula (cloud) P related to Q, which is a list of top-K kterms from $\mathcal{K}$ with $P \cap Q = \emptyset$.

In order to identify the top kterms similar to Q, we define a correlation score, $kcorr(Q, t)$, for each kterm $t \in \mathcal{K}$, which quantifies the similarity of t to Q in the information network $\mathcal{G}$.

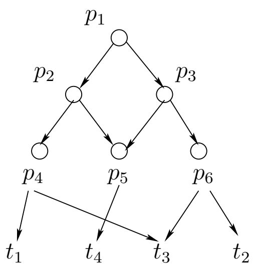
$$kcorr(Q, t) = \sum_{(k_i, w_i) \in Q} w_i \cdot kcorr(k_i, t) \tag{1}$$

where $kcorr(t_1, t_2)$ is a correlation score between two kterms t_1 and t_2 computed in $\mathcal{G}$.

In an information network, each kterm appears at nodes $\mathcal{K}^{-1}(t)$. Thus, the similarity between two kterms t_1 and t_2, $kcorr(t_1, t_2)$, should consider the similarity between the set of nodes containing t_1 and the set of nodes containing t_2. This is because that each similarity between one node from $\mathcal{K}^{-1}(t_1)$ and one node from $\mathcal{K}^{-1}(t_2)$ is a witness of the correlation between t_1 and t_2. Among the link-based similarity measures [11, 13, 23], none of them considers similarity between two sets of nodes. In SimRank [13], it defines the similarity between two nodes as an average of the similarities between their in-neighbors, which are two sets of nodes. Similarly, we define,

$$kcorr(t_1, t_2) = \frac{\sum_{i=1}^{|\mathcal{K}^{-1}(t_1)|} \sum_{j=1}^{|\mathcal{K}^{-1}(t_2)|} sim(\mathcal{K}_i^{-1}(t_1), \mathcal{K}_j^{-1}(t_2))}{|\mathcal{K}^{-1}(t_1)||\mathcal{K}^{-1}(t_2)|} \tag{2}$$

Here, $\mathcal{K}^{-1}(t)$ denotes all nodes that contain the kterm t and $\mathcal{K}_i^{-1}(t)$ denotes the i-th element in $\mathcal{K}^{-1}(t)$. Also, $|\mathcal{K}^{-1}(t)|$ denotes the size of $\mathcal{K}^{-1}(t)$. Let $sim(u, v)$ denote a structural correlation measure of two nodes in a network, which will be discussed shortly. Eq. (2) computes the correlation between two kterms t_1 and t_2 by computing the average structural correlation ($sim(,)$) between every node that contains t_1 and every node that contains t_2 in the network $\mathcal{G}$.



(a) Extended Network

node pair u, v	$sim(u, v)$
t_1, t_2	0.512
t_1, t_3	0.656
t_1, t_4	0.576
t_2, t_3	0.656
t_2, t_4	0.576
t_3, t_4	0.576

(b) SimRank Values

Figure 4: Extended Network

In this paper, for the similarity between two nodes u, v in $\mathcal{G}$, $sim(u, v)$ in Eq. (2), we adopt a structural-context similarity measure in the literature, SimRank [13]. SimRank is a popularly used measurement, which quantifies the similarity of a pair of nodes based on the link structures around these two nodes in the network. Let $sim(u, v)$ denote the similarity between two nodes u and v. u is always similar to itself, i.e., $sim(u, u) = 1$. Otherwise, it is defined in a recursive fashion as follows,

$$sim(u, v) = \frac{C}{|\mathcal{I}(u)||\mathcal{I}(v)|} \sum_{i=1}^{|\mathcal{I}(u)|} \sum_{j=1}^{|\mathcal{I}(v)|} sim(\mathcal{I}_i(u), \mathcal{I}_j(v)) \tag{3}$$

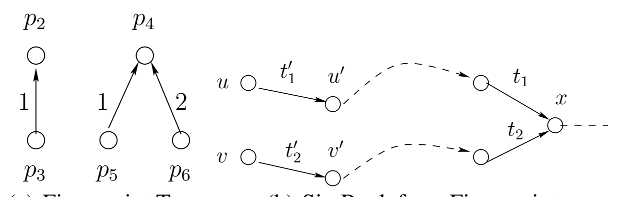
where $\mathcal{I}_i(u)$ means the i-th element in $\mathcal{I}(u)$, $|\mathcal{I}(u)|$ is the number of elements in $\mathcal{I}(u)$, and C is a constant between 0 and 1. When the in-neighbor of u or v is empty, i.e., $\mathcal{I}(u) = \emptyset$ or $\mathcal{I}(v) = \emptyset$, Eq. (3) is not defined, and it is assigned with value 0. The SimRank value will always be between 0 and 1, i.e., $0 \leq sim(u, v) \leq 1, \forall u, v \in \mathcal{V}$. The SimRank values for node pairs in the network in Fig. 3(a) are shown in Fig. 3(b), where C is chosen as 0.8.

The similarity $sim(u, v)$ (see Eq. (3)) between two nodes in a network is defined as an average of the similarities of combinations of in-neighbors of u and in-neighbors of v. Meanwhile, our correlation score between two kterms, $kcorr(t_1, t_2)$ (see Eq. (2)), is also defined as an average of the similarities of combinations of nodes containing t_1 and nodes containing t_2. Thus, we can consider kterms in $\mathcal{K}$ as *kterm nodes*, and treat the nodes containing $t \in \mathcal{K}$, $\mathcal{K}^{-1}$, as in-neighbors of the kterm node t. Based on this, we can use the techniques of computing $sim(,)$ to compute our correlation score $kcorr(,)$. We define the *extended network*.

Extended Network: An extended network $\mathcal{G}^e = (\mathcal{V}^e, \mathcal{E}^e, \mathcal{K})$ is the information network $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{K})$ extended by kterm nodes and edges between kterm nodes and other nodes, i.e., $\mathcal{V}^e = \mathcal{V} \cup \mathcal{K}$ and $\mathcal{E}^e = \mathcal{E} \cup \{\langle v, t\rangle \mid t \in \mathcal{K} \wedge v \in \mathcal{K}^{-1}(t)\}$. For example, Fig. 4(a) shows the extended network of Fig. 3(a).

Although, in the extended network, each kterm in the query Q corresponds to exactly one node, the method used in [25] to compute a connection subgraph for a set of user-given nodes, cannot be applied to our extended network. The reasons are twofold. First, the methods in [25] work only on undirected graphs. However, if we treat the extended network as undirected, the similarity score between node-pairs may change dramatically. For example, in Fig. 4, the similarity of t_1 and t_2 is the lowest among all the kterm node-pairs. Based on the random walk with restart measure [25], $sim'(t_1, t_2)$ will be higher than $sim'(t_1, t_4)$ and $sim'(t_2, t_4)$, since p_4 and p_6 are now very similar to each other by the connection through t_3. Second, in order to find the global top-K correlated kterms, all connection subgraphs are needed, while [25] only finds one best connection subgraph.

In the extended network, we can compute $sim(,)$ for pairs of kterm nodes as shown in Fig. 4(b), using the techniques of computing SimRank. From Eq. (3) and Eq. (2), we know that $sim(t_1, t_2) = C \cdot kcorr(t_1, t_2)$ for any pair of kterm nodes. Then the correlation



(a) Fingerprint Tree (b) SimRank from Fingerprint

Figure 5: Fingerprint and Compute SimRank

score of a kterm t to query Q can be rewritten as

$$kcorr(Q,t) = \frac{1}{C}\sum_{(k_i,w_i)\in Q} w_i \cdot sim(k_i,t) \qquad (4)$$

With the extended network and Eq. (4), our approach finds top-K kterms based on $sim(,)$.

A Naive Algorithm: A naive algorithm, NAIVE, to compute top-K correlated kterms for a user-given query Q, can be designed to work as follows. It first gets the kterm corpus, $\mathcal{K}$, of $\mathcal{G}$, which consists of all the kterms appeared at nodes in $\mathcal{G}$. Then, it builds the extended network $\mathcal{G}^e$ as defined above. For each kterm t in $\mathcal{K}$ but not in the query Q, we compute a correlation score $kcorr(Q,t)$. Finally, the top-K correlated kterms can be identified.

In NAIVE, the most expensive computational cost comes from the $O(l\cdot|\mathcal{K}|)$ $sim(k_i,t)$ values computation, where l is the number of kterms in query Q. However, $|\mathcal{K}|$ can be of the same magnitude of $|\mathcal{V}|$, and it is impossible to compute the $O(l\cdot|\mathcal{K}|)$ $sim(k_i,t)$ values online, even for an information network with modest size, i.e., $n \geq 10,000$.

3. A 2-STEP APPROACH

The problem of computing SimRank for node-pairs in large networks is hard in general [21]. It is impractical to compute SimRank value for so many node-pairs on-line as illustrated in the NAIVE algorithm. We propose a two-step approach which makes use of the properties of fingerprint index [7]. In the first step, we process $\mathcal{G}$ (or more precisely fingerprint index of $\mathcal{G}$) to generate a set of sorted lists, $\mathcal{L} = \{L_1, L_2, \cdots\}$, where the definition of a list and the number of lists are determined by the specific algorithm. In the second step, we adopt the idea of *threshold algorithm* [4, 5] to compute top-K kterms over the set of sorted lists $\mathcal{L}$. The main issues are, how to define and obtain the set of lists $\mathcal{L}$, and how to optimize the threshold algorithm in our problem situation. We discuss the properties of fingerprint index in the following, and propose our algorithms in the next two sections. Note that the extended network is only used to illustrate the ideas, and does not need to be materialized.

3.1 Properties of Fingerprint

In the random surfer-pairs model, $sim(u,v)$ is equal to the expected first meeting distance of two independent random walks traveling backwardly starting from u and v respectively [13]. Let $\tau_{u,v}$ denote the random variable that is equal to the first meeting time of the random walks starting from u and v. $\tau_{u,v} = 0$ if $u = v$, and $\tau_{u,v} = \infty$ if the random walks do not meet. It shows in [13, 7] that, the equation $\mathbb{E}(C^{\tau_{u,v}}) = sim(u,v)$ holds.

Fogaras and Racz [7] generate a set of coalescing walks where each pair of walks will follow the same path after their first meeting time. With these coalescing walks, the equation $\mathbb{E}(C^{\tau_{u,v}}) = sim(u,v)$ still holds, since any pair of walks are independent until they first meet. A fingerprint index is designed to encode and store the set of random walks efficiently. Initially, a path of length zero is generated for each node in $\mathcal{G}$. Then it goes for L iterations to generate random walks up to length L. At each iteration i, all the paths travel one step backward by randomly choosing one of its in-neighbors. For all the paths that end at the same node, e.g., the paths starting from $u_1,\cdots,u_k$, a directed edge $\langle u_j,u_1\rangle$ with weight i is added to the fingerprint index for $2\leq j\leq k$, and the paths starting from $u_2,\cdots,u_k$ are discarded. As an example, one fingerprint index for the network shown in Fig. 3(a) is shown in Fig. 5(a), the labels of the edges are their weights.

Let FPG denote the final graph obtained, it is a forest of rooted trees with edges towards the roots. FPG is of size $2n$, where $n = |\mathcal{V}|$. The FPG compactly encodes all-pairs first meeting time $\tau_{*,*}$, and $\tau_{u,v}$ can be computed using FPG. If u and v are in different trees in FPG, then $\tau_{u,v} = \infty$. Otherwise, u and v has one lowest common ancestor in FPG, let it be x, as shown in Fig. 5(b). Let $p(u,x)$ be the directed path from u to x, then $\tau_{u,v} = \max_{\langle a,b\rangle\in p(u,x)\cup p(v,x)} weight(\langle a,b\rangle)$, i.e., it is equal to the maximum weight among all the edges in the path from u or v to x. For example, in Fig. 5(a), $\tau_{p_2,p_5} = 0$, $\tau_{p_2,p_3} = \max\{0,1\} = 1$, and $\tau_{p_5,p_6} = \max\{1,2\} = 2$.

Note that, PSimRank [7], as a variant of SimRank, modifies SimRank by allowing random walks to meet with higher probability when they are close to each other, i.e., a pair of reverse random walks at u and v will advance to the same node in the next step with probability of Jaccard coefficient $\frac{\mathcal{I}(u)\cap\mathcal{I}(v)}{\mathcal{I}(u)\cup\mathcal{I}(v)}$ of their in-neighbors. PSimRank also has a recursive formula similar to Eq. (3) [7]. In our approach, we can use either SimRank or PSimRank as $sim(,)$.

4. A KTERM-LIST TOP-K ALGORITHM

From Eq. (4), we can see that the correlation score of a kterm t to query Q, $kcorr(Q,t)$, is a linear combination of $sim(k_1,t),\cdots,$ $sim(k_l,t)$, and it is a monotonic function. Therefore, the general idea of threshold algorithm [4, 5] can be adopted here to compute top-K correlated kterms for a user-given query. We define a list for each query kterm $k_i \in Q$, $L_{k_i} = \{(t, sim(k_i,t)) \mid t \in \mathcal{K}\}$, which consists of kterms $t \in \mathcal{K}$ and their correlation scores to k_i. We call this *kterm-based lists*, as one list is defined for each query kterm. There are total l lists, $\mathcal{L} = \{L_{k_1},\cdots,L_{k_l}\}$. The l lists should be computed in advance or on-demand. In order to adopt the idea of threshold algorithm, two interfaces are needed for each list. One is NEXT(L_{k_i}) which returns the next entry $(t, sim(k_i,t))$ in L_{k_i} with the highest $sim(k_i,t)$ value, and it corresponds to the sorted access of a list in the threshold algorithm. The other one is GET(L_{k_i},t) which retrieves the $sim(k_i,t)$ value for kterm t, and it corresponds to the random access of a list in the threshold algorithm. Given l lists, and a sorted access interface (NEXT) and a random access interface (GET) for each list, the threshold algorithm [4, 5] can be used to find top-K correlated kterms for a user-given query.

CL-TOPK is shown in Alg. 1. Here, a priority queue $\mathcal{Q}$ is used to store potential results, each element in it is a kterm in $\mathcal{K}$ with its correlation score to the query Q as the key, i.e., $(t, sim(Q,t))$, and $\mathcal{Q}$ is a minimum queue so that the top element has the minimum key value. For simplicity, we access the l lists in a round-robin fashion, and each list has both sorted access interface (NEXT) and random access interface (GET). An upper bound of the correlation score of unseen kterm is maintained to terminate the algorithm earlier, denoted as *threshold*. We access the l lists in decreasing order, and an upper bound value of $sim(k_i,t)$ is maintained to be the last retrieved entry from L_{k_i} through interface NEXT, denoted as $L^u_{k_i}$. Then, the upper bound correlation score for any unseen kterms in $\mathcal{K}$ is bounded by the sum of $L^u_{k_i}$, i.e., $threshold = \sum_1^l L^u_{k_i}$. For each seen kterm $t \in \mathcal{K}$, its correlation score is computed based on

Algorithm 1 CL-TOPK $(\mathcal{G}, Q)$

```
Input:  an information network $\mathcal{G}$, and a query $Q$.
Output: top-$K$ kterms in $\mathcal{K}$ ranked with respect to $kcorr(Q,t)$.

1: Process $\mathcal{G}$ to generate $l$ lists, $\mathcal{L} = \{L_{k_1}, \cdots, L_{k_l}\}$, and prepare
   NEXT($L_{k_i}$) and GET($L_{k_i}, t$) for each list $L_{k_i} \in \mathcal{L}$;
2: Initialize a priority queue $\mathcal{Q}$ to be empty;
3: Initialize $threshold = \infty$;
4: while $\mathcal{Q}$.SIZE() $< K$ or $\mathcal{Q}$.TOP() $< threshold$ do
5:     Let $L_{k_i}$ be the next list to be accessed, in a round robin fashion;
6:     $(t, sim(k_i, t)) \leftarrow$ NEXT($L_{k_i}$);
7:     for $j \leftarrow 1$ to $l$ and $j \neq i$ do
8:         $sim(k_j, t) \leftarrow$ GET($L_{k_j}, t$);
9:     Compute $kcorr(Q, t)$ based on Eq. (4);
10:    Update $\mathcal{Q}$ with $(t, kcorr(Q, t))$;
11:    Update $threshold$ with $sim(k_i, t)$;
12: Output the kterms in $\mathcal{Q}$;
```

Eq. (4), where $sim(k_i, t)$ for all l lists are retrieved through interface GET. The priority queue $\mathcal{Q}$ maintains the current top-K kterms among all the seen kterms. If the lowest correlation score in $\mathcal{Q}$ is no less than $threshold$, which means that none of the unseen kterms can have a correlation score larger than the lowest correlation score in $\mathcal{Q}$, then the K kterms stored in $\mathcal{Q}$ are guaranteed to be the top-K kterms in $\mathcal{K}$.

Note that, in CL-TOPK, it first generates l lists $\mathcal{L}$, then it works on $\mathcal{L}$ and discards the network $\mathcal{G}$. For each list $L_{k_i} \in \mathcal{L}$, the entries are accessed through two interfaces: NEXT and GET. To implement NEXT(L_{k_i}) and GET(L_{k_i}, t), the entries $(t, sim(k_i, t)) \in L_{k_i}$ are ordered in non-increasing order according to $sim(k_i, t)$. For an information network with millions of nodes, no scalable algorithm in the literature can compute $sim(k_i, t)$ exactly for millions of pairs of nodes in practical time. Because the query kterms are contained in the kterm corpus, i.e., $Q \subset \mathcal{K}$, one may think of generating a list L_t for each $t \in \mathcal{K}$ off-line. However, it is impractical to generate and store the lists $\{L_t \mid t \in \mathcal{K}\}$, because it will take space $O(|\mathcal{K}|^2)$, where $\mathcal{K}$ contains millions of kterms.

To generate the l lists, $\mathcal{L} = \{L_{k_1}, \cdots, L_{k_l}\}$, efficiently on-line, we use an approximate algorithm to compute $sim(,)$, which is based on the fingerprint index as discussed in Section 3.1. It is worth noting that $sim(,)$ values computed using the fingerprint index are approximate values. Let N be the number of fingerprint indices (based on different random walks) used to improve the accuracy, denoted as FPG $= \{\text{FPG}_1, \cdots, \text{FPG}_N\}$, and $sim(,)$ can be computed as,

$$sim(u, v) = \frac{1}{N} \sum_{j=1}^{N} sim_j(u, v) \tag{5}$$

where $sim_j(u, v)$ is the $sim(,)$ value computed on FPG$_j$.

5. AN INDEX-LIST TOP-K ALGORITHM

In order to generate the lists, for each kterm $t \in \mathcal{K}$ and $k_i \in Q$, CL-TOPK needs to compute $sim(k_i, t) = \sum_{j=1}^{N} sim_j(k_i, t)$, which needs to access all the N copies of FPGS, thus it needs a lot of I/Os. In this section, we propose another formulation of lists $\mathcal{L}$, and an optimized top-k algorithm to efficiently find top-K correlated kterms directly from FPG.

In order to develop an efficient algorithm, we rewrite Eq. (4) by combining Eq. (5) as follows,

$$kcorr(Q, t) = \sum_{j=1}^{N} \frac{1}{C} \frac{1}{N} \sum_{(k_i, w_i) \in Q} w_i \cdot sim_j(k_i, t) \tag{6}$$

Algorithm 2 IL-TOPK (FPG, Q)

```
Input:  fingerprint index FPG $= \{\text{FPG}_1, \cdots, \text{FPG}_N\}$ which is
        stored on disk, and a query $Q$.
Output: top-$K$ kterms in $\mathcal{K}$ ranked with respect to $kcorr(Q,t)$.

1: PROCESS(FPG, $Q$);
2: Initialize a minimum priority queue $\mathcal{Q}$ to be empty;
3: Initialize $threshold = \infty$;
4: Initialize a sorted list $ub$ to be empty;
5: for $i \leftarrow 1$ to $N$ do
6:     $ub$.INSERT($(i, \infty)$);
7: while $\mathcal{Q}$.SIZE() $< K$ or $\mathcal{Q}$.TOP() $< threshold$ do
8:     $i \leftarrow ub$.FIRST().$id$;
9:     $(t, corr_i(Q, t)) \leftarrow$ NEXT($L_{F_i}$);
10:    for $j \leftarrow 1$ to $N$ and $j \neq i$ do
11:        $kcorr_j(Q, t) \leftarrow$ GET($L_{F_j}, t$);
12:    $kcorr(Q, t) \leftarrow \sum_{j=1}^{N} kcorr_j(Q, t)$;
13:    Update $\mathcal{Q}$ with $(t, kcorr(Q, t))$;
14:    Update $threshold$ with $kcorr_i(Q, t)$;
15:    Update the key of $i$ in $ub$ to be $kcorr_i(Q, t)$;
16: Output the kterms in $\mathcal{Q}$;
```

If we define a correlation score based on each FPG$_j$, such as,

$$kcorr_j(Q, t) = \frac{1}{C} \frac{1}{N} \sum_{(k_i, w_i) \in Q} w_i \cdot sim_j(k_i, t)$$

then, $kcorr(Q, t) = \sum_{j=1}^{N} kcorr_j(Q, t)$. Here, $kcorr(Q, t)$ is regarded as the global correlation score, and $kcorr_j(Q, t)$ (computed on FPG$_j$) is regarded as the local correlation score. We define a list for each FPG$_j$, denote as $L_{F_j} = \{(t, kcorr_j(Q, t)) \mid t \in \mathcal{K}\}$, which consists of kterms t and their local correlation scores $kcorr_j(Q, t)$ computed on FPG$_j$. We call this *index-based lists*, as one list is generated for each fingerprint index FPG$_j$. There are total N lists, i.e., $\mathcal{L} = \{L_{F_1}, \cdots, L_{F_N}\}$. In each list L_{F_j}, the entries $(t, kcorr_j(Q, t))$ are ordered according to $kcorr_j(Q, t)$. It is worth noting that, for each list L_{F_j}, only those kterms t with non-zero $kcorr_j(Q, t)$ values are of interest, the size of which usually is not large (e.g., in the order of hundreds to thousands). The interfaces for sorted access and random access of list L_{F_j} are NEXT(L_{F_j}) and GET(L_{F_j}, t), respectively, where NEXT(L_{F_j}) gets the next entry $(t, kcorr_j(Q, t))$ from L_{F_j}, and GET(L_{F_j}, t) retrieves the local correlation score of t computed on FPG$_j$.

The algorithm to compute top-K correlated kterms for a user-given query is shown in Alg. 2 (IL-TOPK). In IL-TOPK, Line 1, PROCESS(FPG, Q) generates N sorted lists, $\mathcal{L} = \{L_{F_1}, \cdots, L_{F_N}\}$ which will be discussed shortly. A sorted list ub is used to store the upper bound $kcorr_i(Q, t)$ for each list L_{F_i}. Each entry of ub consists of two fields, id and key, where id denote the list id and key denote the upper bound of the corresponding list. Entries in ub are sorted in deceasing order of key. Here, ub is used to determine the list on which the next sorted access should be. Different from CL-TOPK which accesses the l kterm-based lists in a round-robin fashion, we access the index-based list with highest upper bound in IL-TOPK, i.e., ub.FIRST().id. This is based on the fact that there are N lists in IL-TOPK, which is much larger than that of l lists in CL-TOPK, i.e., $N \gg l$. Thus, we can access the list that is more likely to have answers.

Theorem 5.1: *Let m denote the number of distinct kterms processed, i.e., it is the number of calls of* NEXT*, the time complexity of* IL-TOPK *is $O(m \cdot (N + \log K))$; the space complexity of* IL-TOPK *is $O(K + N)$.* □

Proof Sketch: For each kterm processed (Lines 9-12), it takes $O(N)$ time to compute $kcorr(Q, t)$. Here, we assume a hash index

Algorithm 3 OPTFINGERPRINT $(\mathcal{G}, N, L)$

```
Input:  an extended network $\mathcal{G}$, $N$ indicates the number of
        FPGs, and $L$ indicates the length of random walks.
Output: $N$ fingerprint graphs, $\text{FPG}_1, \cdots, \text{FPG}_N$.

 1: for $i \leftarrow 1$ to $N$ do
 2:     Initialize $\text{FPG}_i$ as $|\mathcal{K}|$ isolated nodes $t_1, \cdots, t_{|\mathcal{K}|}$;
 3:     for each $t \in \mathcal{K}$ do
 4:         $PathEnd[t] \leftarrow t$;
 5:     for $j \leftarrow 1$ to $L$ do
 6:         Generate a random permutation $\delta$ with the $n$ nodes in $\mathcal{V}$;
 7:         for each $t \in \mathcal{K}$ with $PathEnd[t] \neq$ "stopped" do
 8:             $PathEnd[t] \leftarrow \text{argmin}_{u \in \mathcal{I}(PathEnd[t])} \delta(u)$;
 9:         for each set of nodes with the same $PathEnd$, $u_1, \cdots, u_k$ do
10:             Add edges $\langle u_2, u_1 \rangle, \cdots, \langle u_k, u_1 \rangle$ with weights $j$ to $\text{FPG}_i$;
11:             Set $PathEnd[u_2], \cdots, PathEnd[u_k]$ to be "stopped";
```

is used to implement GET. Updating $\mathcal{Q}$ (Line 13) takes $O(\log K)$ time, because the size of $\mathcal{Q}$ is at most K. Updating ub (Line 15) takes $O(N)$ time, because ub is of size N. So the total time complexity is $O(m \cdot (N + \log K))$.

For the space complexity, IL-TOPK maintains a priority queue $\mathcal{Q}$ of size at most K, and a priority queue ub of size N. So it totally takes space $O(K + N)$. □

5.1 The IL-TOPK Implementation Details

In the following, we discuss some IL-TOPK implementation details. First, we show an optimization of the fingerprint index for our problem, which reduces both the index construction time and index size. We propose a general index structure to effectively store the N copies of the fingerprint index on disk. We give an algorithm to generate the N lists L_{F_i} efficiently based on our optimized fingerprint index.

Optimized Fingerprint: In PSimRank, to extend the random walks one step backwardly, instead of choosing one of its in-neighbors independently for all the paths, it first generates an independent random permutation δ on the nodes of $\mathcal{G}$. Then, the random in-neighbor for v is chosen as $\text{argmin}_{u \in \mathcal{I}(v)} \delta(u)$. We show an algorithm, called OPTFINGERPRINT, that directly generates the FPG on $\mathcal{G}$ efficiently in Alg. 3. Initially, FPG_i is initialized as $|\mathcal{K}|$ isolated nodes (Line 2). To expand the paths one step backwardly (Lines 7-8), a random permutation is generated first (Line 6). OPTFINGERPRINT generates only $|\mathcal{K}|$ random walks, while a naive generation of FPG on an extended network needs to generate $|\mathcal{K}| + |\mathcal{V}|$ random walks. With Alg. 3, the FPG generated is of size $2|\mathcal{K}|$. It is smaller than the size of FPG on $\mathcal{G}^e$, which is $2 \cdot (|\mathcal{K}| + |\mathcal{V}|)$. FPG generated by OPTFINGERPRINT contains all the information needed for our algorithm.

A New Block-Based Index Structure: With N copies of FPGs, the total index size is $O(N|\mathcal{K}|)$, where $|\mathcal{K}|$ is of same magnitude of $|\mathcal{V}|$. So it can not completely reside in main memory, we show data structures to store them on disk so that it can be efficiently retrieved and processed by the following query answering algorithms. Recall that, each FPG is a forest of rooted trees. We call a connected tree in a FPG as a *block*, denote as B, since the nodes in it should be stored in consecutive blocks on disk as they will be retrieved as a unit by any processing algorithm. Let the set of FPGs be $\text{FPG}_1, \cdots, \text{FPG}_N$. The index structure consists of *Global index*, where there exists only one for the whole index, and *In-block index*, where there exists one for each block, see Table 1.

For the *Global index*, it has two parts: a mapping between kterms and kterm ids (*kid*); and an inverted list of block ids (*bid*) for each kterm. Because kterms are strings, and they are more costly to

Algorithm 4 PROCESS(FPG, Q)

```
Input:  optimized fingerprint index $\text{FPG} = \{\text{FPG}_1, \cdots, \text{FPG}_N\}$
        which is stored on disk, and a query $Q$.
Output: $N$ sorted list $L_{F_1}, \cdots, L_{F_N}$.

 1: Find and load $l$ inverted lists kid $\rightarrow$ bids into main memory;
 2: for $i \leftarrow 1$ to $N$ do
 3:     Let $\mathcal{B}$ be the set of bids belonging to $\text{FPG}_i$, in the $l$ inverted lists;
 4:     $L_{F_i} \leftarrow \emptyset$;
 5:     for each $bid \in \mathcal{B}$ do
 6:         Load the block of $\text{FPG}_i$ that has block id $bid$ into main memory,
            denote it as $B$;
 7:         for each $t \in B$ do
 8:             Compute $kcorr_i(Q, t)$ and insert $(t, kcorr_i(Q, t))$ into $L_{F_i}$;
 9:     Sort $L_{F_i}$ to build a sorted list with $kcorr_i(Q, t)$ as the key;
10: Initialize a pointer for each list to point to the first entry;
```

Global index (one for the whole index)	
kterm ↔ *kid*	*kid* → *bid*s
···	···
"algebra" ↔ 47	$47 \rightarrow \{(1,10), (2,15), \cdots, (N,6)\}$
"algorithm" ↔ 48	$48 \rightarrow \{(1,11), (2,12), \cdots, (N,17)\}$
···	···
"keyword" ↔ 1030	$1030 \rightarrow \{(1,10), (2,12), \cdots, (N,8)\}$
···	···
In-block index (one for each block)	
kid ↔ *iid*	$\langle u, v \rangle, weight(\langle u, v \rangle)$
···	···
47 ↔ 10	$\langle 8, 10 \rangle, 3$
49 ↔ 11	$\langle 13, 11 \rangle, 2$
···	···
1280 ↔ 138	$\langle 7, 138 \rangle, 5$
···	···

Table 1: Example of Index Structure for Disk Store

store and process than integers, a unique *kid* is used to identify each kterm. In the beginning of query processing, all the input kterms are transformed to their *kid*s. After the processing, the resulting *kid*s are transformed to the corresponding kterms and output to users. Usually, the *kid*s are from the corpus $\{1, 2, \cdots, |\mathcal{K}|\}$, where $|\mathcal{K}|$ is the size of the kterm corpus. For example, in Table 1, the kterm "algebra" has *kid* 47, and the kterm "algorithm" has *kid* 48.

In order to fast retrieve the corresponding block in each FPG_i that contains a specific kterm, an inverted list is built for each *kid*. Note that, a *kid* is contained in exactly one block in each FPG_i, and only the kterms in these blocks have non-zero correlation scores with it. For example, in Table 1, the kterm "algebra" with *kid* 47 has an inverted list, $\{(1,10), (2,15), \cdots, (N,6)\}$, which means that the block id (*bid*) in each FPG_i that contains kterm "algebra" are, *bid* 10 in FPG_1, *bid* 15 in FPG_2, and *bid* 6 in FPG_N.

For each block $B \in \bigcup_{i=1}^{N} \text{FPG}_i$, we build an in-block index so that it can be processed efficiently. Recall that, each block is a reversed tree that all edges direct to the root. Furthermore, in a block B, the kterms contained are not with consecutive *kid*s. In order to store and process a tree in memory effectively and efficiently, the nodes should be mapped to consecutive integers starting from 1. So we store a mapping between kterm ids (*kid*s) and in-block ids (*iid*s). Then, each edge is stored as *iid* of the start node , *iid* of the end node, and its weight. For example, in Table 1, *kid* 47 has *iid* 10, and *kid* 1280 has *iid* 138; the edge $\langle 8, 10 \rangle$ has weight 3, which means that the parent of *kid* 47 (with *iid* 10) is the node with *iid* 8.

List Sorting: PROCESS used in IL-TOPK is shown in Alg. 4. Given FPG and Q, it generates N lists, $\mathcal{L} = \{L_{F_1}, \cdots, L_{F_N}\}$. First, it loads the l inverted lists *kid* → *bid*s (see Table 1), one for each user-specified query kterm, into main memory (Line 1). For each kterm, the *bid*s in the inverted lists are those blocks in the corresponding FPG$_i$ that contain it. Then, a sorted list L_{F_i} is built for each FPG$_i$ (Lines 3-9). Let $\mathcal{B}$ be the set of *bid*s belonging to FPG$_i$, which is found from the l inverted lists (Line 3). Note that, the size of $\mathcal{B}$ can be smaller than l, i.e., there can be more than one user-specified kterms in one block. Then L_{F_i} can be constructed from those blocks indicated by $\mathcal{B}$. For each block in $\mathcal{B}$ (Line 5), we first load the whole block into main memory (Line 6). For each kterm t in the block B (Line 7), $kcorr_i(Q,t)$ can be computed based on B, and the pair $(t, kcorr_i(Q,t))$ is inserted into list L_{F_i} (Line 8). As the kterms are inserted into L_{F_i} in random order, we have to sort it so that the kterms are ranked in non-increasing order with respect to the local correlation scores $kcorr_i(Q,t)$ (Line 9). Note that, after processing a block B (Lines 5-8), block B is discarded from memory.

After processing FPG, there are N lists $\mathcal{L}$, and they can reside in main memory since each list usually is not large. Also, only those N lists should be in memory. In order to call NEXT correctly, a pointer is constructed for each list, which points to the entry that should be returned when NEXT is called next time, and it is initialized to point to the first entry of the corresponding list. Each time when NEXT(L_{F_i}) is called, the entry pointed by the pointer is returned, and the pointer moves to the next entry in the list. GET(L_{F_i}, t) gets the local correlation score $kcorr_i(Q,t)$ for a kterm t, an in-memory index should be built to retrieve it efficiently, because the *kid*s in a list L_{F_i} usually are not with consecutive integer ids.

Theorem 5.2: *Let n_i denote the size of list L_{F_i}, the time complexity of* PROCESS *is $O(\sum_{i=1}^{N}(n_i \cdot l \cdot L + n_i \log n_i))$, where l is the number of query kterms in Q and L is the maximum length of random walks in* FPG*; the space complexity of* PROCESS *is $O(\sum_{i=1}^{N} n_i + l \cdot N)$.* □

Due to space limit, we omit the proof.

Kterm-based lists vs Index-based lists: Recall that, kterm-based lists, $\{L_{k_1}, \cdots, L_{k_l}\}$, is used in CL-TOPK, which generates a list for each $k_i \in Q$, and index-based lists, $\{L_{F_1}, \cdots, L_{F_N}\}$, is used in IL-TOPK, which generates a list for each FPG$_i$ $\in$ FPG. The number of lists in index-based lists is N, which is much larger than that of l kterm-based lists. However, there are advantages of the index-based lists. (1) The generation of index-based lists is more efficient. To generate a list L_{F_i}, only the related blocks, which is less than l, from FPG$_i$ are needed. The related blocks from FPG$_i$ usually are stored closely on disk, which is called the localization of index-based lists. (2) In generation of index-based lists, a block is loaded from disk at most once to compute correlation scores. While in kterm-based lists, a block will be loaded from disk multiple times, once for each query kterm $k_i \in Q$. (3) In order to enable sorted access on lists, entries in a list should be sorted. The size of a list in index-based lists is much smaller than that in kterm-based lists. Therefore, it is much more efficient for the index-based lists to be sorted than that of kterm-based lists.

6. EXPERIMENTS

We conducted extensive performance studies to test the algorithms proposed in this paper. We implemented our algorithms: CL-RR, IL-RR, and IL-BF. Here, CL-RR is for Alg. 1 which is kterm-list based, IL-RR is Alg. 2 with round robin access of the sorted lists, and IL-BF is for Alg. 2 which is index-list based. All algorithms were implemented in Visual C++ 2003, and all tests were conducted on a 2.8GHz CPU and 2GB memory PC running Windows XP.

We used two large real datasets, *DBLP*[1] and *YAGO*[2] [24] for testing. For *DBLP*, the network consists of two types of nodes, `Author` and `Paper`. Each node of type `Author` has a text attribute which stores the author name, and each node of type `Paper` has a text attribute which stores the title of the paper and the conference name of the paper. The kterms are author names and terms contained in paper titles. There are two types of relationships (edges) in *DBLP*, namely, author-write-paper relationship and paper-cite-paper relationship. After processing, the *DBLP* network consists of 2,120,264 nodes and 3,515,292 edges. The total number of distinct kterms and the average number of kterms per node are 445,314 and 5.05, respectively. We also extracted a subgraph from the *DBLP* network, denoted as *DBLP-CP*. It considers only paper-cite-paper relationship, and the papers from some top-ranked conferences are considered. The resulting *DBLP-CP* network consists of 6,705 nodes and 30,559 edges. The total number of distinct kterms and the average number of kterms per node are 4,723 and 6.0, respectively.

YAGO is a huge semantic knowledge base. Each node represents an entity, such as, a person, an organization, a city, etc. The text attribute of a node stores its name, and kterms are entity names. Each edge corresponds to a fact in *YAGO* knowledge base, such as, a person works at which university, a city belongs to which country. After processing, the *YAGO* network we used consists of 314,025 nodes and 398,257 edges. The total number of distinct kterms and the average number of kterms per node are 314,066 and 1.0, respectively.

6.1 Effectiveness Testing

To test the effectiveness of our algorithm, we used the implementation of IL-BF algorithm. We also tested the Center-Piece algorithm proposed in [25], denote as CePS[3], which returns a connection subgraph for a query Q which consists of entities. Since CePS does not scale to large graphs, in order to run CePS, we first extract a subgraph with 3k-7k nodes around the query entities. As the input graphs of CePS are undirected, we treat each edge as undirected in order to run CePS.

(Q1) {"Albert_Einstein", "Walther_Bothe"} on *YAGO*. The connection subgraph obtained by CePS is shown in Fig. 6(a), which indicates that both "Albert_Einstein" and "Walther_Bothe" are "Germany", and they both have won "Nobel_Prize_in_Physics" and "Max_Planck_Medal". However, CePS cannot find the persons in *YAGO* with similar relations as the two querying persons. The top-8 correlated persons returned by our approach are shown in Table 2. We also show in Fig. 6(b) the connection subgraph returned by CePS by adding the first two persons into Q. We know that both "Wolfgang_Ketterle" and "Max_von_Laue" are from "Germany" and have won "Nobel_Prize_in_Physics", and "Max_von_Laue" and "Walther_Bothe" have the same academic advisor "Max_Planck".

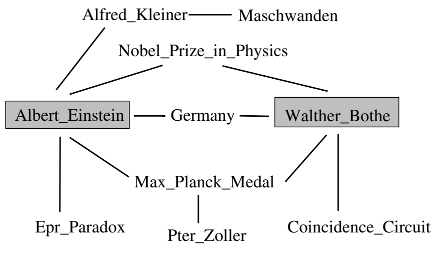
(Q2) {"Isabel_Sanford", "Nathan_Lane"} on *YAGO*. The top-8 correlated actors to the two actors in query are shown in Table 2. These returned actors have won similar awards with actors in Q, such as they have won "Emmy_Award", "Hollywood_Boulevard", and/or "Hollywood_Walk_of_Fame", as shown in Fig. 7. However, the CePS approach can only find the awards that "Isabel_Sanford" and "Nathan_Lane" have won, not actors with similar awards.

(Q3) {"histogram", "wavelet"} on *DBLP-CP*. The top-ranked kterms "synopses" denotes that both "histogram" and "wavelet" are meth-

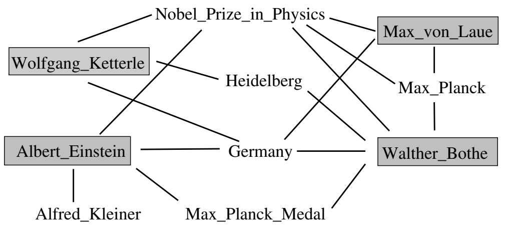
[1]`http://www.informatik.uni-trier.de/~ley/db/`
[2]`http://www.mpi-inf.mpg.de/yago-naga/yago/`
[3]`http://www.cs.cmu.edu/~htong/work/CePS.zip`

$Q1$	$Q2$	$Q3$	$Q4$	$Q5$
Wolfgang_Ketterle	Vivian_Vance	synopses	discovering	Craig_Silverstein
Max_von_Laue	Nanette_Fabray	practicality	maximum	Michelle_Peterson
Theodor_W._hänsch	Tyne_Daly	rectangle	itemsets	David_Hecherman
J._Hans_D._Jensen	Imogene_Coca	fitting	association	Pini_Mogilevski
Leopold_Gmelin	Ron_Holgate	curve	rules	Bay-Wei_Chang
Max_Wolf	Loretta_Swit	approximation	matching	Takuhiro_Nagafuji
Johannes_Georg_Bednorz	Dick_Smothers	medians	episodes	Chung_T._Kwok
Harry_Rosenbusch	Henderson_Forsythe	selectivity	condensed	Brian_Milch

Table 2: Top correlated kterms returned by our approach



(a) CePS for $Q = \{AE, WB\}$



(b) CePS for $Q = \{AE, WB, WK, ML\}$

Figure 6: Subgraphs Obtained by CePS for Q1

ods to build synopses. "rectangle" and "medians" are approximation methods used in generating histograms and wavelets. "curve" "fitting" sometimes is used together with "histogram" and "wavelet" to get better results.

(Q4) {"frequent", "pattern", "mining"} on *DBLP-CP*. The top correlated kterms for this query are shown in Table 2. "discovering" has the same meaning as "mining", which is to discover frequent "itemsets" or frequent patterns from a database, or even to derive "association" "rules" from frequent itemsets. There are also some research papers about finding frequent "episode" patterns from sequential data.

(Q5) {"Sergey_Brin", "Larry_Page"} on *DBLP*, which is to discover the authors who have written papers with those two Google co-founders, or whose papers have similar citation patterns. The top authors are shown in Table 2. Among these eight authors, only "Craig_Silverstein", "Bay-Wei_Chang", and "Brian_Milch" are coauthors of "Sergey_Brin", while the other authors have written papers with similar citation patterns with the papers written by "Sergey_Brin" and "Larry_Page".

6.2 Efficiency Testing

For the efficiency testings, we report the query processing time and peak memory consumption for each test case. The query processing time of CL-RR, IL-RR, and IL-BF includes the time of processing the FPG index to build sorted lists and the time to find top-K kterms, and the time to build FPG index is the index construction time.

Parameter	Values	Default
Kterm Freq (p)	1, 2, 3, 4, 5	3
Kterm Num (l)	1, 2, 3, 4, 5	3
Return Num (K)	10, 20, 50, 100, 1000	50

Table 3: Parameters for Efficiency Testings

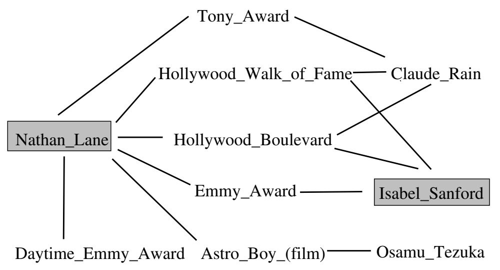
For each dataset, we select representative queries with different kterm frequencies. After removing all the stop words, let τ denote the maximum kterm frequency among all the kterms, the kterms are divided into five categories depends on their frequencies which are evenly divided between 0 and τ. For simplicity, we say a kterm is in category p ($p \in \{1, 2, 3, 4, 5\}$), if and only if its frequency is between $(p-1) \cdot \tau/5$ and $p \cdot \tau/5$.

#	*DBLP*	*YAGO*
1	robust segmentation context	school village home
2	object oriented environment	ohio carolina minnesota
3	dynamic programming method study optimization	robert george michael paul wales
4	distributed web network	wisconsin illinois north
5	system design information	battle park south

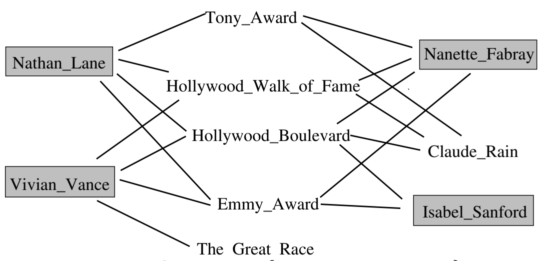
Table 4: Query kterms used in Efficiency Testing (# is the kterms frequency)

For all the testings, we vary three parameters, namely, the kterm frequency (p), the query kterm number (l), and the number of kterms returned (K). The corresponding values for the three parameters are shown in Table 3 together with their default values. When varying the value of one parameter, the other two parameters take their default values. The query kterms selected for the datasets are shown in Table 4. When varying query kterm number l, the first l kterms from the default frequency category are selected.

Exp-1 (Testing on *DBLP* dataset): The time to construct indices for CL-RR and IL-BF algorithms are 32 minutes and 21 minutes, respectively. For *DBLP*, we test three algorithms: CL-RR, IL-RR, and IL-BF, where IL-RR has the same FPG index as IL-BF. The testing results are shown in Fig. 8. From Fig. 8(a) and Fig. 8(b), we know that the query processing time and peak memory consumption of all the three algorithms are not affected too much by the kterm frequency. Because the factors that affect the running time of IL-BF and CL-RR are the number of FPG indices (i.e., N) and the number of nodes in the index blocks that contain user-given kterms. Consistently, IL-BF takes less CPU time than IL-RR, and IL-RR takes less CPU time than CL-RR. IL-BF and IL-RR have the same index, and the same preprocessing algorithm. But, IL-BF accesses the N sorted lists in a best first manner, so it can stop earlier with less access of the lists. IL-RR outperforms CL-RR based on the fact that, the preprocessing algorithm for IL-RR is more efficient because it retrieves each block from disk only once. The memory consumption of CL-RR is less than that of IL-RR and IL-BF, which have almost the same memory consumption. This is because that the main memory consumption for CL-RR and IL-BF

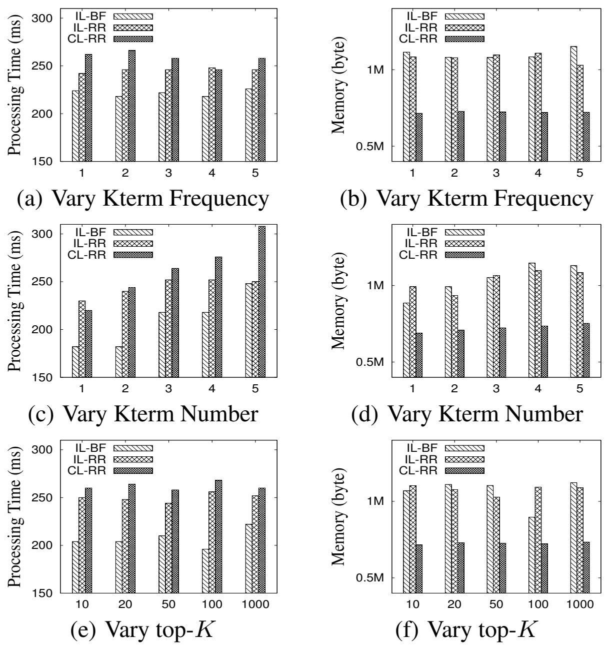


(a) CePS for $Q = \{IS, NL\}$



(b) CePS for $Q = \{IS, NL, VV, NF\}$

Figure 7: Subgraphs Obtained by CePS for Q2

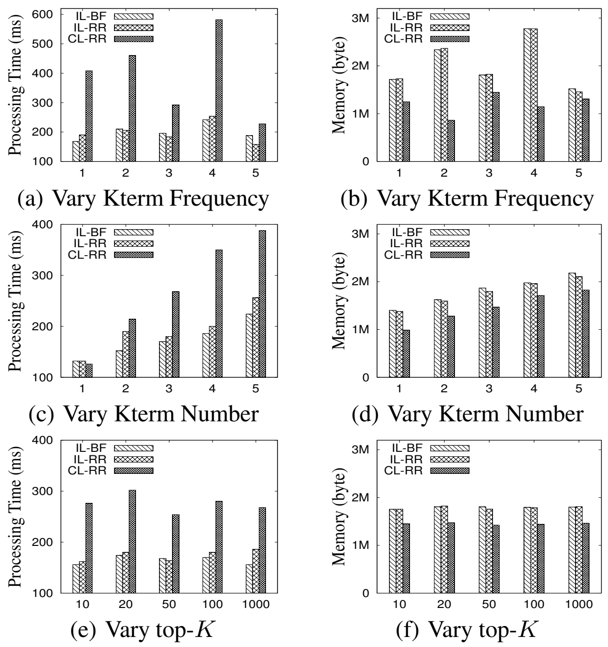


(a) Vary Kterm Frequency (b) Vary Kterm Frequency

(c) Vary Kterm Number (d) Vary Kterm Number

(e) Vary top-K (f) Vary top-K

Figure 8: Testing Results for the *DBLP* Dataset



(a) Vary Kterm Frequency (b) Vary Kterm Frequency

(c) Vary Kterm Number (d) Vary Kterm Number

(e) Vary top-K (f) Vary top-K

Figure 9: Testing Results for the *YAGO* Dataset

algorithms are the l sorted lists and the $N(> l)$ sorted lists, respectively. When increasing the query kterm number, the CPU time and peak memory consumption of all the three algorithms increase, because the size of the sorted lists increases, as shown in Fig. 8(c) and Fig. 8(d). Fig. 8(e) shows that, when the top-K value increases, the CPU time of IL-BF increases while that of CL-RR and IL-RR almost remain the same. This is because that, the best first access makes IL-BF stop earlier when fewer results are desired. It also shows that round robin is inefficient when only a few results are desired.

Exp-2 (Testing on *YAGO* dataset): For this dataset, the time to construct indices for CL-RR and IL-BF algorithms are 4.2 minutes and 2.4 minutes, respectively. The testing results for IL-BF, IL-RR, and CL-RR are shown in Fig. 9, which have similar trends as that for *DBLP* dataset.

7. RELATED WORK

Our work is the first to find correlated kterms for a set of query kterms. It differs from other works in two aspects. First, one kterm can appear at many nodes in a graph. Second, we consider a global structural-context similarity between two sets of nodes in our correlation measure of two kterms. In the literatures, there are lots of work trying to find the best connection subgraph(s) to describe how the query terms are connected. Among them, query terms in [6, 25, 14] are entity names which correspond to unique nodes in a graph, while query terms in [3, 8, 2, 15] are kterms as our work. Instead of finding connection subgraphs, complementally, we find the correlated kterms surrounding the query kterms which is measured by a structural correlation score.

Finding Connection Subgraphs: Given an Entity-Relationship graph and a set of entities, the works in the literature return one (or a set of) connection subgraph (*CSG*), which is a subgraph describes the connection between query entities. Faloutsos et al. [6] proposed a delivered current based method to find a connection subgraph between two query nodes. To handle more than two query nodes, Tong and Faloutsos [25] proposed to find center-piece subgraph (CePS), which defines the proximity of a node t to node s as the probability that a random walk with restart from s will end at t. The center-piece subgraph consists of nodes whose proximity measures to all query nodes are high, and paths connecting to those query nodes. Kasneci et al. [14] first find a steiner tree between the query nodes, and then add nodes based on the probability that random walks from a node in the steiner tree to another node in the steiner tree will pass through it. In the above mentioned works, each query entity corresponds to a unique node in the graph.

Keyword Search: Keyword search in graphs finds small subgraphs which describe the connection between user-given keywords, where a keyword can appear at many nodes. The existing approaches define the cost of a subgraph based on the weights of edges. In [3, 8, 2, 15], they find a set of steiner trees with minimum costs. Exact [3] and approximate algorithms [8, 15] were studied. Dalvi et al. [2] considered external memory graphs. Qin et al. [22] return subgraphs (communities) for a keyword query. All these works aim at finding connection subgraph for a set of terms, based on the weights of edges. Our work is to find similar kterms for a set of user-given kterms based on the similarity between sets of nodes containing the corresponding kterms.

Link-Based Similarity Measures: One fundamental problem in graphs is to determine the similarity between node-pairs, which has a wide range of applications, such as link predication, graph clustering, etc. Liben-Nowell and Kleinberg [20] used similarity scores to predicate links between un-linked node-pairs, and reported that ensemble of paths performs the best. Examples of similarity measures of using the ensembles of paths are, personalized PageRank [11], SimRank [13], etc. In the personalized PageRank, the basic idea is to start a random walk from a node v, and at any setp the walk moves to a random neighbor with probability $1 - \alpha$ and is reset to the start node v with probability α. One characteristic of the personalized PageRank is that nodes close to the starting node will have higher stationary probability [23], and this is used to find connection subgraph for query nodes [25] and find relevant nodes for query keywords [12].

SimRank: SimRank was first proposed by Jeh et al. [13] to measure the similarity of two objects in a relationship graph based on the link structure. Iterative algorithms was proposed in [13]. Lizorkin et al. [21] gave an accuracy estimation for the number of iterations. Optimization techniques were also studied in [21] to reduce the computational cost for each iteration. A non-iterative method was proposed by Li et al. in [18]. Exact SimRank computation based on these techniques can only scale to graphs with thousands of nodes. Fogaras and Racz [7] proposed an index to scale the SimRank computation to graphs with millions of nodes, while the SimRank values computed are approximated values. Also, PSimRank was proposed in [7] to improve the quality of measuring node-pair similarity. As the data graph in our situation can have a size up to millions of nodes, we adopt the index proposed in [7]. However, the query in our work is different from that of [7], and the query answering techniques used in [7] can not be applied in our situation. A technique to efficiently compute SimRank value for a single-pair of nodes was proposed by Li et al. [19]. However, the technique can not be applied to our situation, because it has to compute SimRank values for millions of pairs of nodes. Variants of SimRank were proposed in the literature [26, 1]. P-RANK was proposed by Zhao et al. [26], which takes both in- and out-neighbors into consideration while SimRank only considers in-neighbors. Iterative algorithms were proposed to compute P-RANK values, so it does not scale to graphs with millions of nodes. Antonellis et al. [1] proposed SimRank++ for a weighted bipartite graph, which is an adaptation of SimRank to the special case of weighted bipartite graph.

8. CONCLUSION

In this paper, we studied knowledge discover from information networks which is composed of entities and relationships. Here, we consider the kterms associated with entities, and study the problem in the kterm level rather than entity level. We compute information nebula (cloud) which is a set of top-K kterms P that are most correlated to a set of user-specified kterms Q over a large information network. A global structural-context similarity between two sets of nodes is considered in our correlation measure of two kterms. We present efficient algorithms to find top-K kterms without computing correlation scores for all kterms in the information network. We conducted extensive performance studies using large real datasets, and confirmed the effectiveness and efficiency of our approach.

Acknowledgment: The work was supported by grants of the Research Grants Council of the Hong Kong SAR, China No. 419008 and 419109, and Microsoft Research Asia FY12-RES-OPP-002.

9. REFERENCES

[1] I. Antonellis, H. Garcia-Molina, and C.-C. Chang. Simrank++: query rewriting through link analysis of the click graph. *PVLDB*, 1(1), 2008.

[2] B. B. Dalvi, M. Kshirsagar, and S. Sudarshan. Keyword search on external memory data graphs. *PVLDB*, 1(1), 2008.

[3] B. Ding, J. X. Yu, S. Wang, L. Qin, X. Zhang, and X. Lin. Finding top-k min-cost connected trees in databases. In *Proc. of ICDE'07*, 2007.

[4] R. Fagin. Combining fuzzy information from multiple systems. *J. Comput. Syst. Sci.*, 58(1):83–99, 1999.

[5] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. *J. Comput. Syst. Sci.*, 66(4), 2003.

[6] C. Faloutsos, K. S. McCurley, and A. Tomkins. Fast discovery of connection subgraphs. In *Proc. of KDD'04*, 2004.

[7] D. Fogaras and B. Rácz. Scaling link-based similarity search. In *Proc. of WWW'05*, 2005.

[8] K. Golenberg, B. Kimelfeld, and Y. Sagiv. Keyword proximity search in complex data graphs. In *Proc. of SIGMOD'08*, 2008.

[9] M. Gomez-Rodriguez, J. Leskovec, and A. Krause. Inferring networks of diffusion and influence. In *Proc. of KDD'10*, 2010.

[10] J. Han, Y. Sun, X. Yan, and P. S. Yu. Mining knowledge from databases: an information network analysis approach (tutorial). In *Proc. of SIGMOD'10*, 2010.

[11] T. H. Haveliwala. Topic-sensitive pagerank. In *Proc. of WWW'02*, 2002.

[12] V. Hristidis, H. Hwang, and Y. Papakonstantinou. Authority-based keyword search in databases. *ACM Trans. Database Syst.*, 33(1), 2008.

[13] G. Jeh and J. Widom. Simrank: a measure of structural-context similarity. In *Proc.of KDD'02*, 2002.

[14] G. Kasneci, S. Elbassuoni, and G. Weikum. MING: mining informative entity relationship subgraphs. In *Proc. of CIKM'09*, 2009.

[15] G. Kasneci, M. Ramanath, M. Sozio, F. M. Suchanek, and G. Weikum. STAR: Steiner-tree approximation in relationship graphs. In *Proc. of ICDE'09*, 2009.

[16] A. Khan, X. Yan, and K.-L. Wu. Towards proximity pattern mining in large graphs. In *Proc. of SIGMOD'10*, 2010.

[17] M. Ley. DBLP - some lessons learned. *PVLDB*, 2(2), 2009.

[18] 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 *Proc. of EDBT'10*, 2010.

[19] P. Li, H. Liu, J. X. Yu, J. He, and X. Du. Fast single-pair simrank computation. In *Proc. of SDM'10*, 2010.

[20] D. Liben-Nowell and J. M. Kleinberg. The link prediction problem for social networks. In *Proc. of CIKM'03*, 2003.

[21] D. Lizorkin, P. Velikhov, M. N. Grinev, and D. Turdakov. Accuracy estimate and optimization techniques for simrank computation. *PVLDB*, 1(1), 2008.

[22] L. Qin, J. X. Yu, L. Chang, and Y. Tao. Query communities in relational databases. In *Proc. of ICDE'09*, 2009.

[23] P. Sarkar and A. W. Moore. Fast nearest-neighbor search in disk-resident graphs. In *Proc. of KDD'10*, 2010.

[24] F. M. Suchanek, G. Kasneci, and G. Weikum. Yago: a core of semantic knowledge. In *Proc. of WWW'07*, 2007.

[25] H. Tong and C. Faloutsos. Center-piece subgraphs: problem definition and fast solutions. In *Proc. of KDD'06*, 2006.

[26] P. Zhao, J. Han, and Y. Sun. P-Rank: a comprehensive structural similarity measure over information networks. In *Proc. of CIKM'09*, 2009.