Keyword Search over Graph-structured Data for Finding Effective and Non-redundant Answers

Chang-Sup Park
Department of Computer Science
Dongduk Women's University
Seoul, Korea
cspark@dongduk.ac.kr

***Abstract*—In this paper, we propose a new method for keyword search over large graph-structured data to find a set of answers which are not only relevant to the query but also reduced and duplication-free. We define an effective answer structure and a relevance measure for the candidate answers to a keyword query on graph data. We suggest an efficient indexing scheme on relevant and useful paths from nodes to keywords in the graph. We present a top-*k* query processing algorithm to find relevant and non-redundant answers in an efficient way by exploiting pre-constructed indexes. We show by experiments using real datasets that the proposed approach can produce effective and non-redundant answers efficiently compared to the previous methods.**

Keywords-graph data; keyword search; top-k query processing

I. INTRODUCTION

Recently, graph-structured data is widely used in various fields such as social networks, semantic web, linked open data, and knowledge bases. A relational database also can be considered a directed graph based on the foreign-key relationships among tuples. A graph data consists of nodes and edges, which can represent relationships among entities effectively. As the amount of data increases rapidly, an efficient and effective query system is much needed. Keyword search has been attracting a lot of attention since it allows users to express their information need using simple keywords [1-10].

Keyword search on graph data usually returns a set of connected sub-structures, showing that which nodes include query keywords and how they are inter-connected. Many approaches find minimal connected sub-trees as succinct answers to a given query [1-6, 8, 10]. Since there can be a significant number of answer sub-trees in a large graph data, a relevance scoring function is often used to rank candidate answers and select top-*k* ones having the highest relevance. There have been proposed several approaches based on the *distinct root semantics*, where the relevance of a sub-tree is computed as a function of the shortest paths from the root to the nodes containing query keywords. For each node in the graph, they choose at most one sub-tree rooted at the node as a candidate answer to the query [2, 3, 5, 10]. By reducing the number of candidates significantly, they can process top-*k* query over a large volume of data more efficiently than other approaches. It also facilitates exploitation of indexes on graph data to improve query performance [3].

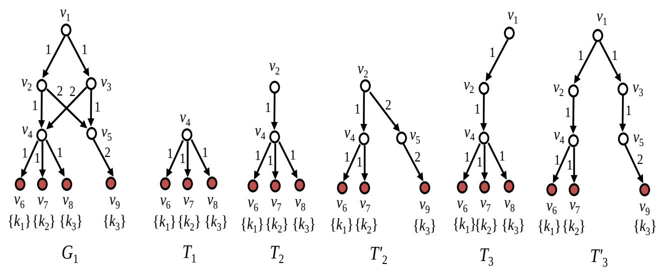


Figure 1. Reduced answers vs. non-reduced answers

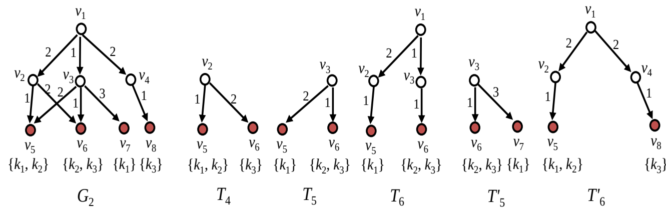


Figure 2. Duplicate answers vs. duplication-free answers

However, the previous methods have a common limitation; they can produce ineffective answers called a *non-reduced* tree and *duplicate* tree. The former is a sub-tree where the root node contains no query keyword and has only a single child node. For example, consider a directed weighted graph G_1 in Fig. 1 where nodes $v_6 \sim v_9$ contain keywords and edges are labeled with a weight representing distance between nodes. Given a keyword query $q = \{k_1, k_2, k_3\}$ over G_1, five sub-trees shown in the figure can be answers to q since they have all the keywords in q in their nodes. Note that T_2 and T'_2 are rooted at the same node v_2 while having different nodes containing keyword k_3, i.e. v_8 and v_9, respectively. Since the distance from v_2 to v_8 is shorter than that from v_2 to v_9, search methods usually select T_2 as the answer rooted at v_2. However, it should be noted that T_2 is a non-reduced answer tree which has a smaller reduced answer T_1 as its sub-tree while T'_2 is a reduced answer. If T_1 is included in top-*k* results, selecting T'_2 instead of T_2 makes the search results more diverse even though the relevance score of T'_2 is lower than that of T_2. It is also desirable to choose a reduced sub-tree T'_3 instead of a non-reduced tree T_3 as an answer rooted at node v_1.

DOI reference number: 10.18293/SEKE2016-140

The other ineffectiveness in the previous approaches is that search results can include similar answer trees containing same set of content nodes for the query keywords. In Fig. 2, for instance, when a query $q = \{k_1, k_2, k_3\}$ is given on G_2, the set of top-3 answers to the query based on the distinct root semantics is $\{T_4, T_5, T_6\}$. Note that these sub-trees have different root nodes but share the same set of content nodes $\{v_5, v_6\}$. If we select T'_5 and T'_6 instead of T_5 and T_6, respectively, into top-3 answers, we can obtain results which are duplication-free and diverse in terms of the content nodes.

Top-*k* search results including many non-reduced and duplicate answer trees have drawbacks. First, similar and redundant answer trees decrease diversity of the search results and do not satisfy users who want to get various answers. Second, if an answer tree turns out to be irrelevant to the query, non-reduced or duplicate answer trees related with the answer would be also irrelevant to the query.

In this paper, we propose a new approach to keyword search over graph data which can produce not only relevant but also diverse results by searching top-*k* answers consisting of only reduced and duplicate-free answer trees. We suggest an extended indexing scheme on the selected paths in the graph and propose an efficient search algorithm exploiting the indices to find relevant and non-redundant answer trees.

II. Related Work

Most previous approaches to keyword search on graph data find minimal sub-trees containing query keywords based on either Steiner-tree semantics [1, 4, 6] or distinct-root semantics [2, 3, 5, 10]. Under distinct-root semantics, sub-trees returned as query answers should be rooted at a distinct node. Thus, for each potential root node in the graph, only a single sub-tree having a minimal weight is considered a candidate answer, where the weight is defined by the sum of the lengths of the shortest paths from its root to keyword nodes. This semantics can deal with queries over a large graph data more efficiently than Steiner-tree semantics [11].

The Bi-directional Search proposed in [2] performs backward explorations of the graph starting from the nodes containing query keywords and also executes forward search from a potential root of an answer tree toward keyword nodes. However, it does not take advantage of any prior knowledge on the graph and depends on a heuristic activation strategy hence it shows poor performance on large graphs. BLINKS approach [3] proposes an efficient indexing scheme on the graph to speed bi-directional exploration with a good performance guarantee. It pre-computes the shortest paths and their distances from nodes to keywords in the graph and stores them in sorted inverted lists and a hash map. By exploiting indexes, it can avoid a lot of explorations in the graph and find top-*k* answers efficiently. For efficient search of a large graph data, [5] suggests creating and utilizing a multi-granular representation of graph data, and presents search algorithms on a multi-granular graph extended from BANKS [1] and Bi-directional Search. A recent study in [10] has proposed an extended answer structure with a new relevance measure and proposed an indexing and query processing scheme similar to BLINKS to produce effective and various top-*k* answers.

These approaches, however, have a common drawback of producing sub-trees that are non-reduced or duplicate in content nodes. Although graph exploration approaches such as BANKS and Bi-directional Search can detect and exclude such answer trees, an exponential number of sub-trees should be probed, resulting in severe performance overhead. BLINKS does not consider redundancies in answers, and even if a redundant sub-tree is detected, no other sub-tree rooted at the same node can be found since the method stores in its index only one optimal path from a node to keyword in the graph. That is, all alternative sub-trees sharing a root node are excluded from consideration. For example, BLINKS cannot produce answer trees T'_2 in Fig. 1 and T'_5 in Fig. 2 as alternatives to the redundant answers T_2 and T_5. This can result in a set of limited and less relevant top-k answers to the query than the other approaches.

III. Problem Definition

A data graph $G(V, E)$ is a directed weighted graph where nodes in V contain keywords and edges in E have a weight representing distance between two incident nodes. The nodes containing a keyword k are called *keyword nodes* or *content nodes* regarding k and the set of those nodes is denoted by $V(k)$. The length of a directed path between two nodes in G is defined as the sum of the weights on edges in the path. Based on [11], we define an answer to a keyword query as follows.

Definition 1. Given a graph $G(V, E)$ and a query $q = \{k_1, k_2, \ldots, k_l\}$ over G, an answer to q is a sub-tree T of G which contains a multiset $C = \{v_1, v_2, \ldots, v_l\}$ of keyword nodes where $v_i \in V(k_i)$ $(1 \le i \le l)$ and satisfies the following conditions: (a) T contains the shortest path from its root to each node in C, (b) all the leaf nodes of T belong to C, and (c) if the root of T has only one child, the root also belongs to C. □

We denote an answer tree having a root node n and a multiset C of keyword nodes by $T(n, C)$. The shortest path from the root n to a keyword node v_i in C is called a *root-to-keyword path* for k_i and denoted by $n \rightarrow k_i$ or $n \rightarrow v_i$. The conditions in Definition 1 specify that answer trees should only have the nodes which are necessary and sufficient to connect their content nodes. In particular, condition (c) requires that answer trees should be *reduced*, i.e., the root of answer trees should have at least two child nodes or be a keyword node in itself. Assume that the root of an answer tree T has only one child and is not a keyword node. Then there exists a sub-tree T' in T which is reduced and has the same set of keyword nodes as T. Since T' is usually given the higher relevance score and preferred by a search method, T becomes a redundant answer to the query.

To find the most relevant answers to a given query, we propose a measure to the relevance of answer trees considering both content nodes and root-to-keyword paths in them. Given a node v having a keyword k, the relevance of v to k can be computed based on the TF-IDF weighting scheme which is popularly used in Information Retrieval [13]. For instance, adopting the weighting scheme used in Apache Lucene text search engine, relevance of v to k is defined by

$$rel(k,v) = \sqrt{tf(k,v)} \cdot \left(1 + \log\left(\frac{|V|}{|V(k)|+1}\right)\right)^2$$

where $|V|$ and $|V(k)|$ are the numbers of nodes in V and $V(k)$ and $tf(k, v)$ is the number of occurrences of k in v.

We also consider the length of the path from the root to each keyword node to measure structural relevance of answer trees. We consider that given an answer tree T, the shorter the distance $dist(n, v_i)$ from its root n to a keyword node v_i, the more relevant to the query the tree T is. Now, the relevance scoring function for answer trees is defined as follows.

Definition 2. Given an answer tree $T(n, \{v_1, v_2, \ldots, v_l\})$ for a query $q = \{k_1, k_2, \ldots, k_l\}$, the relevance of T to the query q is

$$rel(T,q) = \sum\nolimits_{1 \le i \le l} rel(n, k_i, v_i)$$

where

$$rel(n, k_i, v_i) = \frac{rel(k_i, v_i)}{r_{max}} \cdot \left(1 + \log\left(\frac{1}{dist(n, v_i) + 1}\right)\right)$$

and r_{max} is the maximal value of $rel(k, v)$ for all keyword terms k and nodes v in G. □

Note that, in the above definition, relevance $rel(T, q)$ is computed as the sum of relevance $rel(n, k_i, v_i)$ defined on each keyword node and root-to-keyword path in T.

In this paper, we search for the answer trees which are not only reduced but also duplication-free in regard of their content nodes. It means that the answer trees should have different sets of content nodes, as well as should be rooted at distinct nodes. Based on this semantics, we aim to finding k most relevant answers to the query using the relevance function defined above.

IV. Proposed Method

A. Indexing Scheme

To enable efficient search of top-k answers in a large graph data, we propose an indexing scheme for selected node-to-keyword paths in the graph, based on that of BLINKS [3]. It pre-computes the most relevant and useful paths using the relevance function proposed in Definition 2 and stores them in the index consisting of the following components.

- *KNList*(*Keyword-Node Lists*) is a set of inverted lists *KNList*(k) defined for each keyword k in the graph. It stores the most relevant node-to-keyword path from each node to a keyword node for k. Specifically, let $P(n, k) = \{n \to v_i \mid v_i \in V(k)\}$ for a node n and keyword k, and $p_m(n, k)$ be the optimal path in $P(n, k)$ which has the highest value of $rel(n, k, v_i)$. *KNList*(k) stores entries representing $p_m(n, k)$ for all nodes n in the graph. Entries are quadruples (n, v_m, f_m, r_m), where v_m denotes the keyword node containing k, f_m is the *first node* except the start node n (i.e., the next node of n), and r_m is the relevance of the path, $rel(n, k, v_m)$. The entries in the list are sorted in a decreasing order of relevance, which enables efficient search of most relevant paths for keyword k.

- *NKMap*(*Node-Keyword Map*) is a hash table to store information about the most relevant paths for each pair (n, k) of node and keyword in the graph. It stores entries of a pre-defined number of n-to-k paths with the highest relevance, including the optimal path $p_m(n, k)$. The entries are triples (v_i, f_i, r_i) where the fields denote the same as those in *KNList*(k).

- *NKMap$_s$* is a secondary hash table to store information about an alternative node-to-keyword path for all pairs of node n and keyword k in the graph. It is the most relevant path which has the first node different from that of the optimal path $p_m(n, k)$. This index is used to find the most relevant reduced answer trees.

B. Query Processing Algorithm

Our query processing model is based on the Threshold Algorithm [12] which is used to evaluate top-k queries on multi-dimensional data such as multimedia objects. Algorithm 1 shows the sketch of our search algorithm, which uses a priority queue Q_t called top-k queue to maintain top-k candidate answers.

```
Algorithm 1. Keyword Search
Input: a keyword query $q = \{k_1, k_2, \ldots, k_l\}$, $k \in Z^+$
Output: a set of top-$k$ answer trees for $q$
1:  a priority queue $Q_t$ and a set $C$ of nodeID's by $\phi$.
2:  $curRel[i] \leftarrow 0.0$ and $V[i] \leftarrow null$ for all $i \in [1,l]$
3:  Let $L(q) = \{KNList(k_i) \mid k_i \in q\ (1 \le i \le l)\}$.
4:  while an entry exists in a list in $L(q)$ do
5:      Select a list $L_i$ in $L(q)$ in a round-robin manner.
6:      Read an entry $(n, v, f, r)$ at the current position in $L_i$.
7:      $curRel[i] \leftarrow r$
8:      if $n \notin C$ then
9:          $V[i] \leftarrow (v, f, r)$
10:         for-each $k_j \in q$ such that $j \ne i$ do
11:             Look up the first entry $(v_j, f_j, r_j)$ with key $(n, k_j)$ in NKMap.
12:             if the entry was found then $V[j] \leftarrow (v_j, f_j, r_j)$
13:             else goto line #21
14:         if $T(n, V)$ is a non-reduced answer then
15:             $V \leftarrow findReducedAnswer(n, V, q)$
16:         if $V \ne \phi$ and $T(n, V)$ is a duplicate answer then
17:             $V \leftarrow findUniqueAnswer(n, V, q)$
18:         if $V \ne \phi$ then $Q_t \leftarrow Q_t \cup \{(n, V)\}$
19:         $C \leftarrow C \cup \{n\}$
20:     if $|Q_t| = k$ and $rel_k \ge \sum_{1 \le i \le l} curRel[i]$ then break
21: Derive top-$k$ answer trees from the top-$k$ entries in $Q_t$.
```

Given a query $q = \{k_1, k_2, \ldots, k_l\}$, let $L(q)$ be the set of keyword-node lists *KNList*(k_i) for all keywords k_i in q. The algorithm performs sequential scan on the lists in $L(q)$ in parallel (line 5~6). Whenever a new entry is read from a list, its relevance value is stored in an array *curRel* (line 7). If an entry (n, v, f, r) regarding an optimal path from a node n is first retrieved from $L(q)$, entries of the optimal paths $p_m(n, k_j)$ for all the other keywords k_j in q are looked up in *NKMap* and aggregated into an array V (line 8~13). If all the optimal paths are found, an optimal answer tree rooted at n can be derived. We examine whether it has a reduced form and has a unique set of content nodes compared to the other candidates in top-k queue. If it does not, we seek an alternative reduced and unique answer tree using algorithms which will be detailed later (line

14~17). The result tree is stored in Q_t if it is one of the k most relevant found yet (line 18). Since the entries in each list are sorted in a decreasing order of relevance, the sum of the values in *curRel* can serve as an upper bound of relevance of the answer trees which have not been found yet. Thus, the algorithm can terminate safely with the correct top-k answers in Q_t if the condition in line 20 is met, where rel_k is the relevance of the k-th answer tree in Q_t.

C. *Finding Reduced and Unique Answer Trees*

Given a potential root node n, Algorithm 1 searches for an optimal answer tree consisting of the best root-to-keyword paths for each query keyword. If the optimal answer is a non-reduced tree where its root has only one child but is not a keyword node, the first nodes of l root-to-keyword paths are the same to the child of the root. Thus, we can find if the optimal tree is reduced or not by examining the first nodes of all the root-to-keyword paths retrieved from *NKMap*. However, it should be considered that if the root contains all the query keywords and is selected as keyword nodes for them, the root itself can be a reduced answer tree.

Assuming that $T(n)$ is a non-reduced answer tree rooted at node n, if there are multiple reduced answer trees rooted at the same node n, we should select one with the highest relevance score as an alternative to $T(n)$. For keyword k_i in q, let $p_a(n, k_i)$ be the path from n to a node v in $V(k_i)$ which has the first node different from that of $p_m(n, k_i)$ and has the highest score of $rel(n, k_i, v)$. Also suppose that $T_i(n)$ be the sub-tree obtained by replacing the optimal path $p_m(n, k_i)$ with $p_a(n, k_i)$ in $T(n)$. Note that $T_i(n)$ is a reduced answer to q since the first node of $p_a(n, k_i)$ is not equal to those of the other root-to-keyword paths $p_m(n, k_j)$ for keywords k_j in q ($j \neq i$). Now, among l alternative sub-trees $T_i(n)$, the one with the highest relevance is the best reduced answer tree rooted at n.

Algorithm 2 exploits $NKMap_s$ index proposed in Section 4.1, which pre-computes and stores the optimal alternative paths $p_a(n, k)$ for all pairs of node n and keyword k in the graph. Given a non-reduced answer $T(n)$, it first looks up information on alternative paths from n to all the query keywords in $NKMap_s$ (line 2~4). An optimal reduced answer tree can be easily obtained by selecting such a keyword k_i that difference between $p_m(n, k_i)$ and $p_a(n, k_i)$ is the smallest and replacing $p_m(n, k_i)$ with $p_a(n, k_i)$ in $T(n)$ (line 7~8).

Algorithm 2. *findReducedAnswer*

Input: a *nodeID n*, an array V of (*nodeID*, *nodeID*, *rel*)'s, a query q
Output: an array $V[1..l]$ of (*nodeID*, *nodeID*, *rel*)'s

```
1: Let $A[1..l]$ be an array and $A[i] \leftarrow null$ for all $i \in [1,l]$.
2: for-each $k_i \in q$ do
3:     Look up entry $(v_i, f_i, r_i)$ with key $(n, k_i)$ in $NKMap_s$.
4:     if the entry was found then $A[i] \leftarrow (v_i, f_i, r_i)$
5: if $A[i] = null$ for all $i \in [1,l]$ then return $\phi$
6: else
7:     Find $i \in [1,l]$ such that $(V[i].rel - A[i].rel)$ is minimal.
8:     $V[i] \leftarrow A[i]$
9: return $V$
```

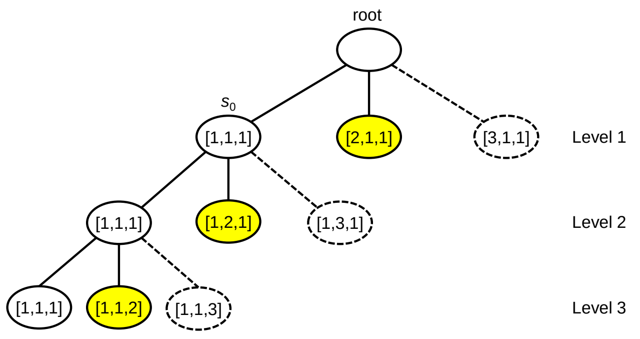


Figure 3. State space search performed by Algorithm 3

Now, we consider finding top-k answer trees which are duplication-free in regard of content nodes. An answer tree $T(n, C)$ is duplicate if and only if (a) its relevance score is greater than that of the k-th candidate answer tree in Q_t and (b) there exists an answer tree $T(m, C)$ in Q_t which contains the same set of content nodes as $T(n, C)$ and has no smaller relevance score than $T(n, C)$. Given a duplicate tree T, we should find another answer tree $T(n, C')$ which is rooted at the same node n as T and has a set of content node different from all the other answers in Q_t. Assuming that the graph has p root-to-keyword paths from n to each keyword in the query of size l, p^l answer trees rooted at n can be derived from the combinations of the paths. To find the optimal one which is not duplicate and has the highest relevance score efficiently without generating all the possible answers, we suggest a state space search algorithm based on branch-and-bound strategy, shown in Algorithm 3. It exploits *NKMap* index which has information on p most relevant paths from a node to keyword in a decreasing order of relevance. As shown in an example in Fig. 3, the search space forms a tree of states each of which represents a combination of root-to-keyword paths for query keywords and derives an answer tree. A state at level i selects one of p paths for k_i ($1 \le i \le l$) and has the optimal paths for all keywords k_j where $i < j \le l$. For the other keywords, it inherits the selection of paths from its parent state. In Fig. 3, the indexes of the selected paths are presented in the states. We can see that an answer tree represented by a state s has no smaller relevance score than those derived from the descendent states of s in the states tree. Also note that sibling states choose a different path to the same keyword in a decreasing order of relevance from left to right hence the answer tree from s has no smaller relevance than those derived from the right sibling states of s. Considering these features, Algorithm 3 explores the search space in an efficient way by pruning a large portion of unnecessary states.

In Algorithm 3, a priority queue Q_s is used to store states to be explored, starting from the initial state which has the optimal paths for all the query keywords (line 2). At each stage, a state e with the highest relevance score is selected from the queue (line 4~5). If the score of e is greater than that of the best state found yet, denoted by LB, the next sibling state s of e is generated and investigated (line 8~16). If the score of s is no greater than LB, all of its descendent states can be safely excluded from further exploration. If s derives a reduced and unique candidate answer tree and its score is greater than LB, it is considered a new best solution state (line 12~14). Otherwise, it is stored in Q_s. If e is not a leaf state, the first child of e is generated and the above process is repeated on it (line 17~19). Fig. 3 shows the states generated in the first round of the outermost loop in Algorithm 3. During the best-first search, if a state selected from Q_s derives no better answer trees than the best solution state found yet, the search can terminate and return the best answer tree (line 6).

Algorithm 3. *findUniqueAnswer*

```
Input: a nodeID $n$, an array $V$ of ($nodeID$, $nodeID$, $rel$)'s, a query $q$
Output: an array $V'[1..l]$ of ($nodeID$, $nodeID$, $rel$)'s
1:  $UB \leftarrow score(T(n, V))$, $LB \leftarrow rel_k$, $bestSolution \leftarrow \phi$
2:  a priority queue $Q_s \leftarrow \{s_0\}$, where $s_0$ is an initial state.
3:  while there exists a state in $Q_s$ do
4:      $e \leftarrow$ a state in $Q_s$ whose $score$ is maximal
5:      $Q_s \leftarrow Q_s / \{e\}$
6:      if $score(e) \le LB$ then break
7:      loop
8:          if $e$ has the next sibling state then
9:              Generate the next sibling state $s$ of $e$.
10:             if $score(s) > LB$ then
11:                 if $score(s) \le UB$ then
12:                     if $s$ is a solution state then
13:                         $bestSolution \leftarrow s$
14:                         $LB \leftarrow score(s)$
15:                     else $Q_s \leftarrow Q_s \cup \{s\}$
16:                 else $Q_s \leftarrow Q_s \cup \{s\}$
17:         if $e$ is a non-leaf state then
18:             Generate the first child state $c$ of $e$.
19:             $e \leftarrow c$
20:         else break
21: if $bestSolution \neq \phi$ then
22:     for-each $k_i \in q$ do
23:         $V'[i] \leftarrow$ an entry ($v_i$, $f_i$, $r_i$) which is looked up in $NKMap$
     with the key ($n$, $k_i$) and selected by $bestSolution$
24:     return $V'$
25: else return $\phi$
```

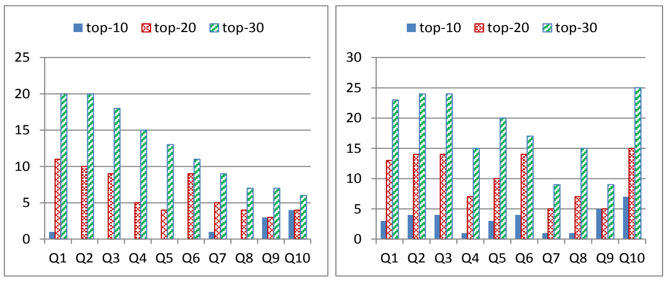
V. PERFORMANCE EVALUATION

We evaluate effectiveness and efficiency of the proposed m by experiments using real graph data. We compare the performance of our method with BLINKS [3], which uses path indexes similar to our method, and a modified version of it, called *BLINKS-N*, which detects and excludes non-reduced or duplicate sub-trees from the candidates. We implemented two versions of the proposed method; *Reduced* method only finds top-*k* reduced answers while *Reduced&Unique* method produces both reduced and duplication-free answers. All the algorithms are implemented in Java. We used JGraphT[1] library to construct in-memory graph structures and compute the shortest paths between pairs of nodes. We exploited Apache Lucene[2] library to extract keywords from nodes in the graph and compute the relevance of the nodes to keyword terms.

As for the test graph datasets, we use a geographic data Mondial[3] and a movie database IMDB[4]. From Mondial, we selected a subset of entities and relationships to build a graph including 6,431 nodes, 19,951 edges, and 15,815 keyword terms. In IMDB database, we derived a large graph consisting of 831K nodes, 2.82M edges, and 303K keyword terms. For the sake of simplicity and efficiency of experiments, we assume that all edges have the same weight of 1 and only use the node-to-keyword paths the length of which are no longer than 5. Our experiments have been conducted on a LINUX server having 10 1.7GHz hexa-core CPUs and 32GB RAM.

TABLE 1. TEST QUERIES

Mondial		IMDB	
Query	Keyword list	Query	Keyword list
Q1	caldera, lake, america	Q11	drama, sports, competition
Q2	cape, gulf, africa	Q12	friendship, love, marriage
Q3	vienna, donau, alps	Q13	emperor, war, battle
Q4	lake, quebec, canada	Q14	hitchcock, mystery, thriller
Q5	himalaya, india, pakistan	Q15	police, crime, violence
Q6	river, minnesota,	Q16	human, vampire, fight
Q7	city, desert, california	Q17	thriller, murder, crime
Q8	lake, michigan, ontario	Q18	natural, disaster, war
Q9	island, vancouver, seattle	Q19	president, politics, drama
Q10	alaska, arctic, sea	Q20	accident, explosion, crash



(a) non-reduced answers (b) duplicate answers

Figure 4. Number of non-reduced and duplicate answers by BLINKS

Table 1 shows a subset of the test queries used in experiments. We executed the search methods to find top-*k* answers to the queries. Fig. 4 shows top-10, top-20, and top-30 answers to the queries on Mondial obtained by BLINKS. It shows that the results include respectively 1, 6, and 13 non-reduced answers and 3, 10, and 18 duplicate answers on average while the proposed method has returned no non-reduced or duplicate answer at all.

Fig. 5 presents relevance scores of top-30 answers to the queries on Mondial obtained by each method. It shows that the results by our method have lower relevance scores than those by BLINKS, but *Reduced* method achieves higher relevance than *BLINKS-N* for all test queries. Fig. 6 shows that the average relevance score of the results by *Reduced* is about 6.6% lower than the results by BLINKS, but about 3.1% higher than the answers obtained by *BLINKS-N*. This indicates that even though our method replaces non-reduced answers with reduced ones which may have lower relevance scores, it can produce more effective results than *BLINKS-N*.

Fig. 6 also shows that average execution times of two versions of our method increase by about 37.8% and 52.7% respectively compared to BLINKS, due to the overhead

[1] http://www.jgrapht.org/
[2] http://lucene.apache.org/java/docs/index.html
[3] http://www.dbis.informatik.uni-goettingen.de/Mondial/
[4] http://www.imdb.com/

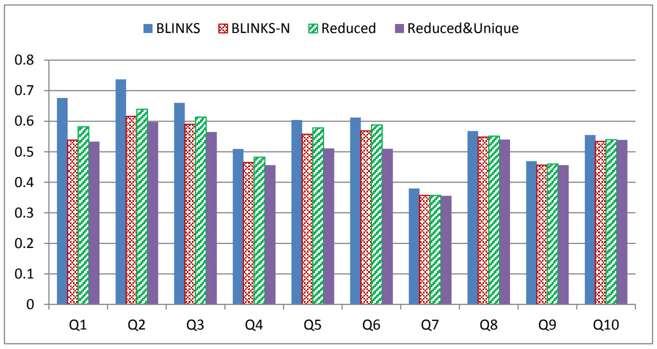


Figure 5. Relevance of top-30 answers

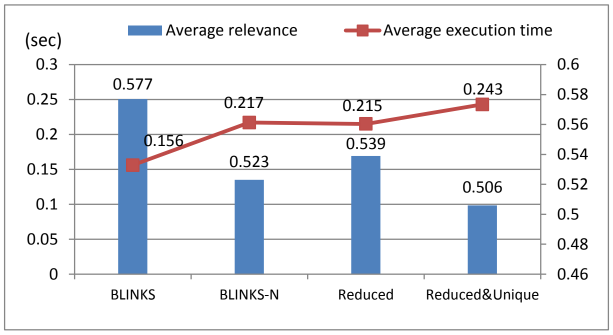
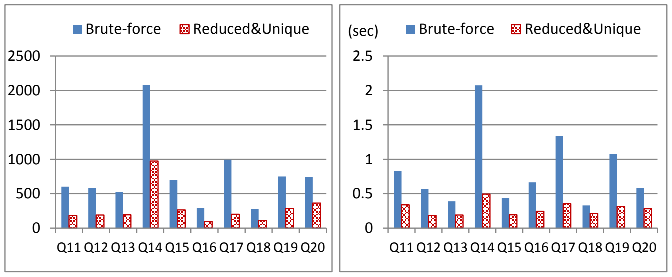


Figure 6. Average relevance and execution time



(a) Number of states generated (b) Execution time

Figure 7. Performance of the state search algorithm to find duplication-free answers

occurred by additional search for the optimal reduced and duplication-free answers. However, we can observe that *BLINKS-N* is also degraded by 39.1% and our *Reduced* method slightly outperforms it.

In Fig. 7, we compare execution performance of the best-first state search strategy employed in *Reduced&Unique* method with that of a naïve approach to conducting brute-force search to find optimal duplication-free answers, based on the search result for the test queries on IMDB dataset. Fig. 7-(a) shows the number of states, i.e., answer trees explored in each method, and Fig. 7-(b) presents their execution time. We observe that our method achieves performance improvement about 62.0% in the number of states generated and 65.9% in execution time.

VI. Conclusion

In this paper, we proposed a new approach to keyword search over graph data to find answers which are not only relevant to the query but also reduced and duplication-free. We suggested an efficient indexing scheme to index relevant paths between nodes and keywords in the graph, and proposed a top-*k* query processing algorithm to find the most relevant non-redundant answers in an efficient way by exploiting the pre-constructed indexes. By producing non-redundant and relevant answer trees, it can provide users with diverse and effective query results and thus satisfy the users' information need. We showed by experiments using a real graph dataset that our approach is effective and efficient for a large graph data.

Acknowledgment

This work was conducted in part while the author was visiting the Department of Computer Science and Engineering, University of California, San Diego, USA. The author would like to thank Prof. Yuanyuan Zhou for inviting him to the UC San Diego and for providing research environment.

References

[1] G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, and S. Sudarshan, "Keyword searching and browsing in databases using BANKS," In Proc. of IEEE 18th Int. Conf. on Data Engineering, pp.431-440, 2002.

[2] V. Kacholia, S. Pandit, S. Chakrabarti, S. Sudarshan, R. Desai, and H. Karambelkar, "Bidirectional expansion for keyword search on graph databases," In Proc. of the 31st Int. Conf. on Very Large Data Bases, pp.505-516, 2005.

[3] H. He, H. Wang, J. Yang, and P. S. Yu, "BLINKS: ranked keyword searches on graphs," In Proc. of 2007 ACM SIGMOD Int. Conf. on Management of Data, pp.305-316, 2007.

[4] 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 IEEE 23rd Int. Conf. on Data Engineering, pp.836-845, 2007.

[5] B. B. Dalvi, M. Kshirsagar, and S. Sudarshan, "Keyword search on external memory data graphs," Proceedings of the VLDB Endowment, Vol.1, No.1, pp.1189-1204, 2008.

[6] K. Golenberg, B. Kimelfeld, and Y. Sagiv, "Keyword proximity search in complex data graphs," In Proc. of 2008 ACM SIGMOD Int. Conf. on Management of Data, pp.927-940, 2008.

[7] L. Qin, J. X. Yu, L. Chang, and Y. Tao, "Querying communities in relational databases," In Proc. of IEEE 25th Int. Conf. on Data Engineering, pp.724-735, 2009.

[8] T. Tran, S. Rudolph, P. Cimiano, and H. Wang, "Top-k exploration of query candidates for efficient keyword search on graph-shaped data," In Proc. of IEEE 25th Int. Conf. on Data Engineering, pp.405-416, 2009.

[9] M. Kargar and A. An, "Keyword search in graphs: finding r-cliques," Proceedings of VLDB Endowment, Vol.4, No.10, pp.681-692, 2011.

[10] C. Park and S. Lim, "Efficient processing of keyword queries over graph databases for finding effective answers," Information Proc. and Mgmt, Vol.51, No.1, pp.42-57, 2015.

[11] J. X. Yu, L. Qin, and L. Chang, "Keyword search in relational databases: a survey," Bulletin of the IEEE CS Technical Committee on Data Engineering, Vol.33, No.1, pp.67-78, 2010.

[12] R. Fagin, A. Lotem, and M. Naor, "Optimal aggregation algorithms for middleware," Journal of Computer and System Sciences, Vol.66, No.4, pp.614-656, 2003.

[13] S. Buttcher, C. Clarke, and G. Cormack, Information retrieval: implementing and evaluating search engine. MIT Press, 2010.