An Effective Keyword Search Method for Graph-Structured Data Using Extended Answer Structure

Chang-Sup Park

Dongduk Women's University, 23-1 Wolgok-dong, Seongbuk-gu, Seoul, Korea
csparkt@dongduk.ac.kr

Abstract. This paper proposes an effective approach to ranked keyword search over graph-structured data which is getting much attraction in various applications. To provide more effective search results than the previous approaches, we suggest an extended answer structure which has no constraint on the number of keyword nodes and is based on a new relevance measure. For efficient keyword search, we also use an inverted list index which pre-computes connectivity and relevance information on the nodes in the graph. We present a query processing algorithm based on the pre-constructed inverted lists, which aggregates entries relevant to each node and finds top-*k* answer trees relevant to the given query. We also enhance the basic search method by storing additional information on the relevance of the related entries in the lists, in order to estimate the relevance score of each node more closely and to find top-*k* answers more efficiently. We show by experiments that the proposed keyword search method can provide effective top-*k* search results over large amount of graph-structured data with good execution performance.

Keywords: Graph-structured Data, Keyword Search, Top-*k* Query Processing.

1 Introduction

Recently, graph-structured data are widely used in various applications such as XML, semantic web, ontologies, social network services, and bio-informatics. Keyword-based search over graph-structured databases has been attracting much attention since it allows users to represent their information need using only a set of keyword terms without understanding and using a query language and underlying database schema [1-7]. Keyword-based query processing has also been studied extensively in the literature of relational databases. Many approaches materialize relational data as a directed graph where tuples are treated as nodes and foreign-key relationships among tuples are represented as edges [8-18].

The previous keyword search methods for graph-structured data usually return a set of connected structures, either sub-trees or sub-graphs, from the database, which represent how the data containing query keywords are interconnected in the database. Given a query, since there can be a significant number of answer structures in a large volume of graph data, search methods usually adopt a scoring function to evaluate and rank the answer structures and return top-*k* ones most relevant to the query.

B. Murgante et al. (Eds.): ICCSA 2013, Part V, LNCS 7975, pp. 620–635, 2013.
© Springer-Verlag Berlin Heidelberg 2013

To satisfy users' information need by finding more effective and relevant answers to a given query than the previous approaches, we suggest an extended answer structure which has no constraint on the number of keyword nodes chosen for each query keyword and is based on a new relevance measure for nodes in the graph. Then we propose an inverted list index to represent connectivity and relevance information on the nodes, as well as a query processing algorithm exploiting the pre-constructed index to find top-k answer trees. Aiming at improving the efficiency of the proposed method, we also present an enhanced inverted list which stores additional information on the relevance of related entries and an improved search algorithm which estimates the relevance score of each node more closely and can find top-k answers more efficiently.

The rest of the paper is organized as follows. Section 2 presents related work and motivation of our study. Section 3 defines a new answer structure for keyword queries and a relevance measure for it. In Section 4, we propose an inverted list index and describe a top-k query processing algorithm using the index. In Section 5, we present an extension of the inverted list and an enhanced search algorithm to process keyword queries more efficiently. We provide experimental results on the effectiveness and efficiency of the proposed methods in Section 6 and draw a conclusion in Section 7.

2 Related Work and Motivation

In the previous approaches to keyword-based search on a graph-structured database, tree structure is popularly used to describe an answer to a given query [3, 4, 5, 8, 9, 13, 14]. As a sub-tree of the database graph, an answer tree should have nodes directly containing the keywords in the query and its leaves should come from those keyword nodes. To rank the sub-trees satisfying the above conditions, weight functions were proposed in the literature based on two different semantics [19]. The Steiner tree-based semantics defines the weight of an answer tree as the total weight of the edges in the tree. Under this semantics, finding an answer tree with the smallest weight is the well-known optimal group Steiner tree problem which is NP-complete [20]. The previous approaches based on this semantics have limitations on the search result and performance against the large amount of graph data [4, 8, 9, 14].

As an alternative to the Steiner tree semantics, some approaches adopted easier semantics, namely distinct root semantics, to find answer trees rooted at distinct nodes [3, 5, 13]. For each node in the graph, only a single sub-tree is considered a possible answer to the query, which is rooted at the node and has the minimal weight. The weight of a sub-tree is defined as the sum of the shortest distances from the root to the keyword nodes chosen for each query keyword. Under this semantics, given a graph having n nodes, there can be at most n answer trees and thus we can deal with very large graph databases more efficiently than using the Steiner tree semantics. A bidirectional search algorithm proposed in BANKS-II [13] performs backward explorations of the graph starting from nodes containing query keywords, as well as forward explorations from the potential roots of answer sub-trees toward keyword nodes. It uses a heuristic activation strategy to prioritize nodes to expand during the bidirectional search. However, it does not take advantage of connectivity information in the graph hence it may lead to poor performance on certain graphs. In BLINKS approach [5], indexing schemes and query processing algorithms were proposed to speed up the bidirectional exploration of the graph with a good performance

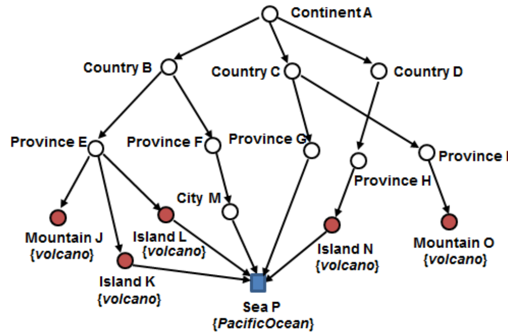


Fig. 1. An example of keyword search on graph data

guarantee. A single-level index, consisting of sorted keyword-node lists and a node-keyword map, pre-computes and indexes all the shortest paths and distances from nodes to keywords in the graph. By exploiting the index, the query processing algorithm performs graph search efficiently and finds top-k answers in a time and space efficient manner. To reduce the index space for a large graph, they also proposed graph partitioning strategies and a bi-level indexing scheme.

The previous approaches mentioned above have a common constraint in the answer structure employed: for each and every keyword in the query, one and only one node directly containing it should be included as a keyword node. For example, assume that a query $Q = \{volcano, PacificOcean\}$ is given on the graph-structured data shown in Fig. 1. The set of nodes directly containing the keywords in Q is $\{J, K, L, N, O, P\}$. Denoting an answer sub-tree by <root node, a set of selected keyword nodes>, possible answer sub-trees containing a keyword node for each keyword are $<E, \{J, P\}>$, $<E, \{K, P\}>$, $<E, \{L, P\}>$, and $<H, \{N, P\}>$. Note that in the previous approaches based on the distinct root semantics, at most one sub-tree is selected among those having the same root and returned as an answer to the query. Assuming that keyword nodes J, K, L, N, and O have the same relevance to keyword *volcano* and all the edges in the graph have the same weight, the answer trees rooted at E, $<E, \{J, P\}>$, $<E, \{K, P\}>$, and $<E, \{L, P\}>$, will have the same weight and rank as the answer tree rooted at H, $<H, \{N, P\}>$, under the distinct root semantics. Moreover, the answer trees rooted at B, $<B, \{J, P\}>$, $<B, \{K, P\}>$, and $<B, \{L, P\}>$, have larger weights (and thus lower ranks) than the answer tree rooted at C, $<C, \{O, P\}>$ since the shortest distance from B to P is longer than the shortest distance from C to P. However, we observe that nodes E and B have more paths to the nodes containing keyword *volcano* than nodes H and C, respectively, hence they should be considered more relevant to the given query.

To improve the quality of query results, we propose a new answer structure which has no constraint on the number of keyword nodes chosen for each query keyword. In an answer tree, multiple keyword nodes can be chosen for a certain keyword while no keyword node can be included for some keywords. Specifically, given a pre-defined constant p and a node n in the graph, we find top-p pairs of a query keyword k and a node s containing k, to which n is most relevant. The relevance of n to a (k, s) pair is computed by the relevance of s to k and the shortest distance from n to s. Then the

answer tree rooted at n is a sub-tree constructed by merging the shortest paths from n to each keyword node s contained in the p pairs of a keyword and node chosen for n. For example, assuming that $p = 4$, possible answers to the query {*volcano*, *PacificOcean*} over the graph in Fig. 1 include the sub-trees that are rooted at E or B and have nodes J, K, L, and P together, i.e. $<E, \{J, K, L, P\}>$ and $<B, \{J, K, L, P\}>$. According to the proposed relevance measure, they have higher scores and ranks than other answer trees such as $<H, \{N, P\}>$ and $<C, \{O, P\}>$. Thus our answer structure can produce more effective top-k search results compared to the previous approaches.

3 Problem Definition

Let $G = (V, E)$ be a directed graph representing a graph-structured database and K be a set of keyword terms extracted from the nodes in V. We define relevance of a node in V to a keyword term in K based on the *tf-idf* weighting scheme [21] which is popularly used in information retrieval. We consider that even if a node n does not contain a keyword k, it can be relevant to k if it has a path to a node s directly containing k, called a keyword node for k. We first define the relevance of n with respect to a pair of keyword k and a keyword node s as follows.

Definition 1. (Relevance of a node *n* to a keyword *k* contained in a node *s*) Given a keyword $k \in K$ and a node $s \in V(G)$, let $tf(k, s)$ be the number of occurrences of k in s and $df(k)$ be the number of nodes in $V(G)$ which contain k. The relevance of s to k is defined by

$$rel(s,k) = \sqrt{tf(k,s)} \cdot \left(1 + \log\left(\frac{N}{df(k)+1}\right)\right)$$

where N is the number of nodes in $V(G)$. For nodes n and s in $V(G)$, the relevance of n to s is defined by

$$rel(n,s) = \begin{cases} 1, & n = s \\ \dfrac{1}{|SP(n,s)+1|}, & n \neq s \text{ and } \exists (\text{a path from } n \text{ to } s) \\ 0, & otherwise \end{cases}$$

where $|SP(n,s)|$ is the length of the shortest path from n to s. The relevance of n with respect to k in s is defined by

$$rel(n,s,k) = rel(n,s) \cdot rel(s,k). \qquad \square$$

According to the above definition, when s does not contain k or there is no path from n to s, $rel(n, s, k)$ becomes 0. Also note that if n and s represent the same node, $rel(n, s, k)$ equals to the relevance of n to the keyword k, i.e., $rel(n, k)$. A node n is considered *relevant* to a keyword k if and only if there exists a keyword node s such that $rel(n, s, k) > 0$.

Given a keyword query $Q = \{k_1, k_2, \ldots, k_l\}$ and a positive integer p, an answer structure and its relevance to the query are defined based on Definition 1 as follows.

Definition 2. (Answer tree to a query *Q* and its relevance to *Q*)) Given a keyword query Q, a node $n \in V(G)$, and a constant p greater than or equal to $|Q|$, let $Top_p(n,Q)$ be the set of p pairs of a node $s \in V(G)$ and a keyword $k \in Q$ such that the relevance of n with respect to (s, k) is in the p highest among all the pairs of a node in $V(G)$ and a keyword in Q. That is,

$$Top_p(n,Q) = \{(s,k) | s \in V(G), k \in Q, rel(n,s,k) \text{ is in the } p \text{ highest of } rel(n,s_i,k_j)'\text{s for all } (s_i,k_j) \in V(G) \times Q\}$$

where ties in relevances are broken at random. Let $V(n,Q)$ be the set of keyword nodes selected for $Top_p(n,Q)$, i.e., $V(n,Q) = \{v \,|(v,k) \in Top_p(n,Q)\}$. An answer tree to the query Q rooted at a node n, denoted by $T(n,Q)$, is a sub-tree of G which contains all the nodes in $V(n,Q)$ and consists of the shortest paths from n to each node in $V(n,Q)$.

The relevance of $T(n,Q)$ to the query Q, denoted by $rel(n,Q)$, is the sum of the relevances of n with respect to the (node, keyword) pairs in $Top_p(n,Q)$, i.e.,

$$rel(n,Q) = \sum_{(s,k) \in Top_p(n,Q)} rel(n,s,k)$$

□

Note that our approach is based on the distinct root semantics and thus there is at most one answer tree $T(n, Q)$ rooted at a node n. It has multiple keyword nodes for some keywords in Q to which the root n is most relevant in terms of $rel(n, s, k)$.

4 Basic Search Method

In this section, we propose a keyword search method including an indexing scheme and query processing algorithm to find k best answers to a given query based on the relevance measure defined in the previous section.

4.1 Inverted List Index

To enable efficient exploration of the graph-structured data, we use an inverted list-style index on the nodes which pre-compute and store information on the relevant nodes for each keyword term. Based on the proposed relevance measure, we find all the relevant nodes, as well as keyword nodes, for each keyword in the graph and then build an inverted list per keyword which is formally defined as follows.

Definition 3. (Inverted list *L*(*k*) for a keyword *k*) For a keyword term k in the graph, let $S(k)$ be the set of nodes in $V(G)$ which contain k. The inverted list for k, denoted by $L(k)$, is a list of triples $(n, s, rel(n, s, k))$[1] obtained from all the pairs of nodes $n \in V(G)$

[1] Practically, the node IDs of n and s are stored in the entry.

and $s \in S(k)$ such that $rel(n, s, k) > 0$. The list entries are sorted in a non-increasing order of their relevance values. Formally,

$$L(k) = \langle (n_1, s_1, r_1), (n_2, s_2, r_2), \dots, (n_m, s_m, r_m) \rangle, \text{where } n_i \in V(G), s_i \in S(k),$$
$$r_i = rel(n_i, s_i, k)\ (1 \le i \le m), \text{and } r_i \ge r_{i+1} > 0\ (1 \le i \le m-1)$$

□

We call a list entry (n, s, r) an entry of node n. As defined above, $L(k)$ stores entries of the nodes that are directly or indirectly relevant to k in a decreasing order of relevance values. Therefore, we can find the nodes most relevant to k by reading the entries in $L(k)$ sequentially. Note that the proposed inverted list is different from the conventional ones used for ranked search over documents or multi-dimensional data [21, 22, 23] by the fact that it can have entries of the nodes that do not contain the keyword of the list in them, in addition to the entries of keyword nodes for k. The proposed list is also distinguished from the keyword-node list suggested in BLINKS [5] since it can have multiple entries of the same node n, one for each keyword node reachable from n, while the latter has only a single entry for each node n which refers to a keyword node in the shortest distance from n.

4.2 Query Processing

Our query processing model is based on the threshold algorithm [22, 23], which is popularly used for top-k query processing on multi-dimensional data, such as similarity search on multimedia objects [24, 25, 26]. Given a query $Q = \{k_1, k_2, \dots, k_l\}$, let $L(Q)$ be the set of inverted lists for the query keywords, i.e. $L(Q) = \{L(k_i) \mid k_i \in Q\}$. We perform sequential scans on the inverted lists in $L(Q)$ in parallel by reading their entries in a round-robin manner. During the scan, the query processor maintains the relevance value of an entry at the current scan position in each list $L(k_i)$, denoted by $curScore_i$. The largest one among those is called *maxCurScore*, i.e., $maxCurScore = max_{1 \le i \le l}\ curScore_i$. Note that since the entries in each list are stored in a non-increasing order of their relevance values, *maxCurScore* can serve as an upper bound of the relevance values of the entries that have not yet been read from the lists in $L(Q)$.

While reading the lists, the query processor also maintains a priority queue per a candidate root node n of an answer tree, called a relevance queue q_n. It stores at most p entries of n retrieved from the lists which have the highest relevances. From a triple (n, s, r), only the pair of s and r is stored in the queue. The list of relevance values in q_n that are greater than or equal to *maxCurScore* is denoted by R_n, i.e. $R_n = [r \mid (s, r) \in q_n, r \ge maxCurScore]$.

Since *maxCurScore* is an upper bound of the relevances of the entries currently unseen from the lists in $L(Q)$, we ensure that the relevance values in R_n belong to the p highest ones of all the entries of n in the lists. Thus, the sum of the values in R_n can be a lower bound of $rel(n, Q)$, the relevance of an answer tree rooted at n. Based on the observation, we define the worst score of n as follows:

$$worstScore(n) = \textstyle\sum_{r \in R_n} r\ . \qquad (1)$$

In addition, assuming that the unknown values in the final top-p relevances of n are the same as $maxCurScore$, an upper bound of $rel(n, Q)$, called the best score of n, can be defined as follows:

$$bestScore(n) = worstScore(n) + maxCurScore \cdot (p - |R_n|). \quad (2)$$

Note that since *maxCurScore* monotonically decreases as entries are retrieved from the lists, *worstScore*(n) monotonically increases whereas *bestScore*(n) monotonically decreases during the list scan. When the relevance queue of n has p entries and all the relevances in them are no less than *maxCurScore* (i.e., $|R_n| = p$), *bestScore*(n) and *worstScore*(n) equal *rel*(n, Q).

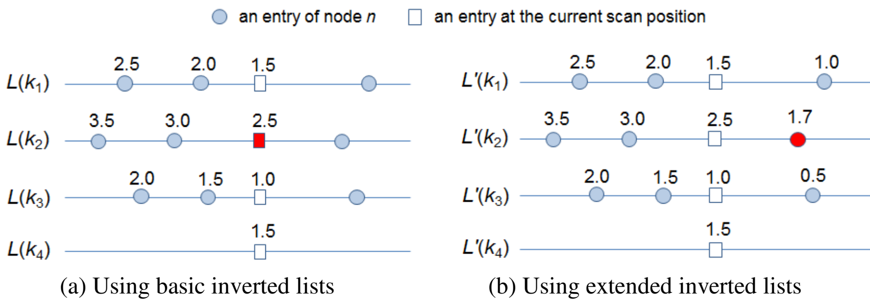


Fig. 2. An example of computing the worst and best scores of a node

Example 1. Fig. 2-(a) shows an example of computing the worst and best scores of a node n given a query $Q = \{k_1, k_2, k_3, k_4\}$ and $p = 6$. In the figure, lines represent inverted lists for the query keywords, scanned from left to right in a round-robin manner. In the lists, the entries of n are indicated by closed dots with their relevance values and the entries at the current scan positions are denoted by rectangles. $curScore_i$'s are 1.5, 2.5, 1.0, and 1.5, respectively, hence $maxCurScore = 2.5$. Currently, the relevance queue of n has 6 entries of n retrieved from the lists, i.e. $q_n = [(s_1, 3.5), (s_2, 3.0), (s_3, 2.5), (s_4, 2.0), (s_5, 2.0), (s_6, 1.5)]$, and the list of relevance values in q_n that are greater than or equal to *maxCurScore* is $R_n = [3.5, 3.0, 2.5]$. Consequently, based on Eq. (1) and Eq. (2), we have *worstScore*(n) = 3.5 + 3.0 + 2.5 = 9.0 and *bestScore*(n) = 9.0 + 2.5 · 3 = 16.5. □

As scanning the inverted lists, we find a set of nodes that can be roots of top-k answer trees using two priority queues.

- A top-k queue T stores at most k nodes having the highest worst scores among those that have been read from the lists. The nodes in T are sorted by their worst scores in a descending order. The minimum (i.e., rank-k) worst score value from the current top-k nodes is called *min-k*, i.e.,

$$\text{min-}k = \begin{cases} \min_{n \in T}\{worstScore(n)\}, & if\ |T| = k \\ 0, & otherwise \end{cases}$$

- A candidate queue C maintains candidate nodes which have a worst score smaller than *min-k* but could still make it into the top-k queue T. A node whose best score is smaller than *min-k* cannot belong to the final top-k nodes and thus

is removed from C. The nodes in C are sorted in a descending order of their best scores to facilitate looking up a node with the maximum best score.

Whenever the worst score and best score of a node change during the list scan, we check if the node can be entered into the top-k queue T or it should be maintained in the candidate queue C. Query processing can terminate safely with the correct top-k nodes in T when the maximum best score in C as well as the best score of any node n_u currently unseen from the lists is no higher than $min\text{-}k$, i.e., when

$$|T| = k \text{ and } max\left\{\max_{m\in C}\{bestScore(m)\}, bestScore(n_u)\right\} \leq min\text{-}k, \\ \text{where } bestScore(n_u) = maxCurScore \cdot p \quad (3)$$

Then, using each node in T and the set of keyword nodes stored in its relevance queue, we can derive top-k answer trees from the data graph as defined in Definition 2.

Algorithm 1. Basic Search

```
1  For a given query Q = {k1, k2, …, kl}, let L(Q)={L(ki) |ki∈ Q} and curScorei = 0 (1≤i≤l)
2  Initialize a top-k queue T and a candidate queue C empty.
3  repeat {
4      Select a list Li from L(Q) in a round-robin manner.
5      Read an entry e=(n, s, r) at the current scan position in Li.
6      curScorei := r and maxCurScore := max{curScorei} (1≤i≤l)
7      if (n had been evicted from C or top-p relevances of n had been found) continue
8      Insert (r, s) into the relevance queue of n, i.e., qn.
9      Compute worstScore(n) and bestScore(n) based on maxCurScore.
10     if (e is the first entry of n found in L(Q)) {
11         if (worstScore(n) > min-k)
12             Insert n into T (re-calculate min-k).
13         else if (bestScore(n) > min-k) Insert n into C.
14     }
15     else if (n is in T and worstScore(n) increases from the previous value) {
16         Remove and re-insert n (re-calculate min-k).
17     }
18     else if (n is in C) {
19         if (worstScore(n) > min-k)
20             Move n from C into T (re-calculate min-k).
21         else if (bestScore(n) ≤ min-k)
22             Remove n from C.
23         else if (bestScore(n) decreases from the previous value)
24             Remove and re-insert n.
25     }
26     if (a node m was ejected from T in Line 12 or 20 and bestScore(m) > min-k)
27         Insert m into C.
28     if ((C = ∅ or bestScore of the top node in C ≤ min-k) and maxCurScore · p ≤ min-k)
29         break
30     Update T and C periodically after every pre-defined number of entries is read.
31 } until (no entry remains in the lists in L(Q))
32 Build top-k answer trees using the nodes in T and the entries in their relevance queues.
33 return top-k answer trees.
```

Fig. 3. Query processing algorithm

Fig. 3 shows a sketch of the query processing algorithm described above. At each step of reading an entry of a node *n* from inverted lists in a round-robin manner, the following tasks are performed repeatedly. First, if either the node *n* had been evicted from the candidate queue *C* or *rel*(*n, Q*) had been already determined, the current entry is ignored (in Line 7). In Line 8~9, the relevance queue q_n of *n* is updated using the current entry, and *worstScore*(*n*) and *bestScore*(*n*) are computed based on q_n and *maxCurScore*. If the current entry is the first entry of *n* found from the lists, *n* can be inserted into the top-*k* queue *T* or candidate queue *C* depending on its worst and best scores and the current *min-k* value in *T* (in Line 10~14). When *n* is already in *T*, *T* should be reorganized based on the new *worstScore*(*n*) (in Line 15~17). Or, when *n* already exists in *C*, *n* is moved into *T*, remains in *C*, or is eliminated from *C* depending on its new worst and best scores and *min-k* value (in Line 18~25). As mentioned earlier, if Eq. (3) is satisfied by the result of the above tasks, query processing stops immediately and top-*k* answer trees can be derived from the graph using the nodes in *T* and the entries stored in their relevance queues (in Line 28~32).

In our method, the worst and best scores of the nodes stored in the top-*k* queue and candidate queue change as the list entries are read since they depend on *maxCurScore*. However, a naïve approach to re-calculating the worst and best scores of all the nodes in two queues and re-organizing the queues in every step of the list scan would incur very large overhead. Therefore, we perform periodic updates and cleaning of the queues after every pre-defined number of entries is read from the lists (in Line 30). We omit detailed algorithm of the queue updates due to the limit of space.

5 Enhanced Search Method

In the basic method described in Section 4, the worst and best scores of each node are estimated assuming that all the unknown relevance values in the entries unseen from the lists are equal to *maxCurScore*, i.e. the largest relevance value of the entries at the current scan positions. This strategy, however, is too conservative since the actual relevances of the entries unseen from a list $L(k_i)$ might be much smaller than the relevance of the entry at the current scan position in the list, i.e. $curScore_i$. We consider that when we read an entry of *n* from a list, if the relevance of the entry of *n* appearing *next* in the same list is available, we can predict *worstScore*(*n*) and *bestScore*(*n*) more closely to the correct relevance score of *n*, i.e. *rel*(*n, Q*), by exploiting it instead of $curScore_i$. Based on the consideration, we propose an extended structure of inverted list which has in each entry of a node, additional information on the relevance of the next entry of the same node, formally defined as follows.

Definition 4. (Extended inverted list $L'(k)$ for a keyword k) For a keyword term *k*, let *S*(*k*) be the set of nodes in *V*(*G*) which contain *k*. For a node *n* in *V*(*G*) and a keyword term *k*, let *L*(*n*, *k*) be the ordered list of triples (*n*, *s*, *rel*(*n*, *s*, *k*)) which are obtained from all the nodes *s* in *S*(*k*) such that $rel(n, s, k) > 0$ and are sorted in a non-increasing order of *rel*(*n*, *s*, *k*). Formally,

$$L(n,k) = \langle (n,s_1,r_1),(n,s_2,r_2),\ldots,(n,s_m,r_m)\rangle,$$
$$\text{where } s_i \in S(k), r_i = rel(n,s_i,k)\ (1 \le i \le m), \text{and } r_i \ge r_{i+1} > 0\ (1 \le i \le m-1).$$

Then we consider a list $L'(n, k)$ derived from $L(n, k)$ as follows:

$$L'(n,k) = \langle (n,s_1,r_1,r_1'),(n,s_2,r_2,r_2'),\ldots,(n,s_m,r_m,r_m')\rangle,$$
$$\text{where } r_i' = \begin{cases} r_{i+1}, & 1 \le i \le m-1 \\ 0, & i = m \end{cases}$$

The extended inverted list for k, denoted by $L'(k)$, is a list of quadruples (n, s, r, r') which are merged from the lists $L'(n_i, k)$ for all nodes $n_i \in V(G)$ and sorted in a non-increasing order of the relevance value r. □

Now, we suggest an enhanced query processing algorithm based on the extended inverted lists, which can compute a narrower range of (*worstScore*, *bestScore*) for each node in the lists and thus can find the top-k nodes relevant to a given query earlier. The overall query processing strategy is similar to the basic algorithm described in Section 4.2. Assuming that $L'(Q) = \{L'(k_i) \mid k_i \in Q\}$ for a given query Q, we scan the lists in $L'(Q)$ in parallel by reading entries in a round-robin manner. Like the basic search algorithm, the query processor maintains a relevance value $curScore_i$ at the current scan position of each list $L'(k_i)$, as well as the top-k queue and candidate queue of the nodes with the highest worst and best scores. For each node n in the queues, the enhanced method maintains a relevance queue as well as a *next relevance value* of n in each list $L'(k_i)$, denoted by $nextScore_{n,i}$, which is obtained from r' in an entry (n, s, r, r') of n read from the list $L'(k_i)$ most recently. It provides the relevance of the entry of n which will be found next when the scan on the list $L'(k_i)$ continues.

When no entry of n has been retrieved from a list $L'(k_i)$ yet and $nextScore_{n,i}$ is unknown, the maximum relevance of the entries of n in the list is estimated by the relevance of the entry at the current scan position, i.e., $curScore_i$. Therefore, an upper bound of the relevances in the entries of n unseen from the lists in $L'(Q)$ can be obtained from $nextScore_{n,i}$ and $curScore_i$ for all $i \in [1..l]$ as follows:

$$maxNextScore_n = max\begin{Bmatrix} \{nextScore_{n.i} \mid nextScore_{n,i} > 0, 1 \le i \le l\} \cup \\ \{curScore_i \mid nextScore_{n,i} = \infty, 1 \le i \le l\} \end{Bmatrix} \quad (4)$$

Now, a lower bound and upper bound for the relevances of a node n with respect to Q can be computed based on $maxNextScore_n$ instead of *maxCurScore*. We first identify from the relevance queue of n a list R'_n of relevance values that are no less than the current $maxNextScore_n$, i.e. $R'_n = [r \mid (s,r) \in q_n, r \ge maxNextScore_n]$. Since $maxNextScore_n$ is an upper bound for the relevances of n unseen from the lists, *worstScore*(n) and *bestScore*(n) are defined as follows:

$$worstScore(n) = \textstyle\sum_{r \in R'_n} r \quad (5)$$

$$bestScore(n) = worstScore(n) + maxNextScore_n \cdot (p - |R'_n|) \quad (6)$$

Since the entries in each list are sorted in a descending order of relevance, $curScore_i$ and $nextScore_{n,i}$ monotonically decrease during the list scan. Note that when an entry

of n is found from $L'(k_i)$ for the first time, its next relevance value is no greater than the previous $curScore_i$. Thus, $maxNextScore_n$ in Eq. (4) monotonically decreases as we proceed with the list scan. Therefore, for each node n, $worstScore(n)$ monotonically increases while $bestScore(n)$ monotonically decreases during the scan.

Example 2. Fig. 2-(b) shows an example of computing the worst and best scores of a node n when evaluating a query using extended inverted lists. Assuming that the graph data are the same as Example 1, at the current scan positions denoted by rectangles, the relevance queue q_n and $curScore_i$'s have the same entries and values as those in Example 1. From the extended inverted lists, however, the relevances of the entries of n which will appear next after the current scan positions in the lists are available, i.e., $nextScore_n = [1.0, 1.7, 0.5, \infty]$. Note that $nextScore_{n,4} = \infty$ since no entry of n has been found from $L'(k_4)$ yet. Therefore, according to Eq. (4), $maxNextScore_n$ becomes 1.7, which is the largest value among $nextScore_{n,i}$ for $i \in [1..3]$ and $curScore_4$, the current relevance value 1.5 in $L'(k_4)$. Based on it, we have $R'_n = [3.5, 3.0, 2.5, 2.0, 2.0]$, and according to Eq. (5) and (6), $worstScore(n)$ and $bestScore(n)$ are 13.0 and 14.7, respectively. Note that this range of the relevance score of n is much narrower than the result [9.0, 16.5] obtained from the basic inverted list in Example 1. □

Fig. 4 shows a sketch of the enhanced query processing algorithm we have described. The overall structure is the same as the basic search algorithm presented in Fig. 3. It should be noted that $nextScore_{n,i}$ is introduced for each node n to maintain the next relevance of n from list $L'(k_i)$ (in Line 9~10) and $maxNextScore_n$ is computed and used to estimate the worst and best scores of the current node n (in Line 11~12). Processing the current node n in the top-k queue or candidate queue, checking the termination condition, and updating the queues periodically, are the same as the basic algorithm. Note that periodic updates of the queues also exploit $maxNextScore_n$ for each node n in the queues instead of $maxCurScore$.

Algorithm 2. Enhanced Search
1 For a given query $Q=\{k_1, k_2, \ldots, k_l\}$, let $L'(Q)=\{L'(k_i) \mid k_i \in Q\}$ and $curScore_i = 0$ $(1 \le i \le l)$
2 Initialize a top-k queue T and a candidate queue C empty.
3 **repeat** {
4   Select a list L_i from $L'(Q)$ in a round-robin manner.
5   Read an entry $e=(n, s, r, r')$ at the current scan position in L_i.
6   $curScore_i := r$ and $maxCurScore := max\{curScore_i\}$ $(1 \le i \le l)$
7   **if** (n had been evicted from C **or** top-p relevances of n had been found) **continue**
8   Insert (r, s) into the relevance queue of n, *i.e.*, q_n.
9   **if** (e is the first entry of n found in the lists in $L'(Q)$) $nextScore_{n,i} := \infty$ $(1 \le i \le l)$
10   $nextScore_{n,i} := r'$
11   $maxNextScore_n := max\{\{nextScore_{n,i} \mid nextScore_{n,i} > 0, 1 \le i \le l\} \cup \{curScore_i \mid nextScore_{n,i} = \infty, 1 \le i \le l\}\}$
12   Compute $worstScore(n)$ and $bestScore(n)$ based on $maxNextScore_n$.
13~33: *Refer to Line 10~30 of **Algorithm 1** in Fig. 3.*
34 } **until** (no entry remains in the lists in $L'(Q)$)
35 Build top-k answer trees using the nodes in T and the entries in their relevance queues.
36 **return** top-k answer trees.

Fig. 4. Enhanced query processing algorithm

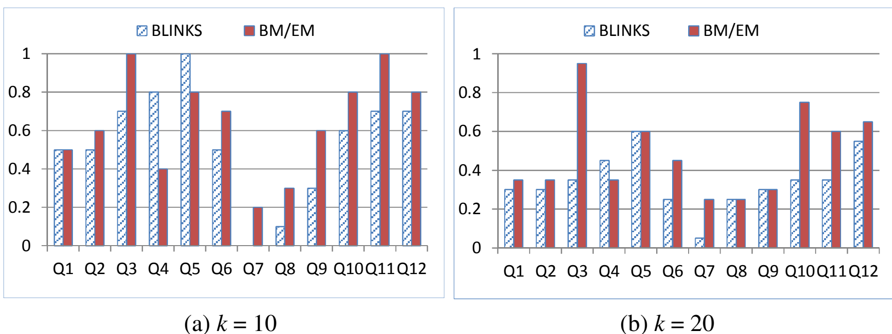
6 Performance Evaluation

In this section we evaluate effectiveness and efficiency of the proposed keyword search methods by experiments using real datasets. We implemented the proposed methods, Basic Method (BM) and Enhanced Method (EM), in Java. For the performance comparison, we also experimented with BLINKS [5] which adopts distinct root semantics and uses a kind of inverted list index.

For experimentation, we used Mondial [2] and IMDB [3] databases which store geographic data and movie-related data. In IMDB databases, we selected data on 147K movies released in 2006~2010 and derived a directed graph which contains about 831K nodes, 2.8M edges, and 303K keyword terms. We used JGraphT [4] library to construct and manipulate the graph. We also used Lucene [5] library to extract keywords from the nodes in the graph and to compute relevances of nodes to keywords. The top-*k* queue and candidate queue in the proposed algorithms were implemented using a Fibonacci heap. The top-*k* queue stores node IDs using the worst score value as a key to facilitate selecting of the min-*k* node, while the candidate queue uses the best score of a node as a key to find the node with the highest best score in the queue efficiently. We conducted experiments on a server machine with two 2.0GHz Quad Core CPUs and 8GB memory.

Table 1. Test queries on Mondial dataset

query ID	keyword set	query ID	keyword set
Q1	{Michigan, Wisconsin, Toronto}	Q7	{military, communist, Asia, Africa}
Q2	{Atlantic, cape, bay}	Q8	{volcano, island, Pacific, Ocean}
Q3	{monarchy, democracy, Europe}	Q9	{Spain, Morocco, Malta, Gibraltar}
Q4	{salt, lake, Asia}	Q10	{volcano, volcanic, mountain, island}
Q5	{republic, catholic, Europe}	Q11	{Himalaya, China, Nepal, India}
Q6	{APEC, Asia, America}	Q12	{Reykjavic, Ireland, Norwegian, sea}



(a) $k = 10$ (b) $k = 20$

Fig. 5. Precisions of the top-*k* answers obtained by the search methods

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

In the first experiment, we evaluated precision of the search results obtained by our approach and BLINKS. The precision of top-k answers to a query is measured by $P@k = \frac{|Res[1..k] \cap Rel|}{k}$, where $Res[1..k]$ is a set of top-k answer trees returned as a search result and *Rel* is a set of all the answer trees in the graph which are considered relevant to the given query [21]. In the experiment of our approach, parameter p is set to double the number of keywords in the query.

Fig. 5 shows the precisions of top-10 and top-20 search results for the test queries over Mondial dataset shown in Table 1. We can observe that the precision of the result of our method is higher than or equal to that of BLINKS for the most queries, specifically, for 10 queries in top-10 search and for 11 queries in top-20 search. The average precisions of our method and BLINKS for the given queries are respectively 0.53 and 0.64 in top-10 searches and 0.34 and 0.49 in top-20 searches. Note that for the queries with AND semantics such as *Q*4, *Q*5, and *Q*8, BLINKS shows higher precisions than our method while for the queries having OR semantics such as *Q*3, *Q*6, *Q*7, and *Q*11, our method achieves better performance than BLINKS. This is due to the fact that the previous methods including BLINKS only search for sub-trees containing keyword nodes for each and every keyword in the query while the proposed method finds sub-trees having a different number of keyword nodes for each query keyword which are most relevant to the root node. This enables the proposed method to find more relevant results than the previous methods for some queries such as *Q*10.

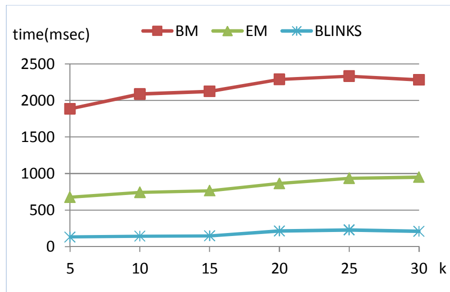
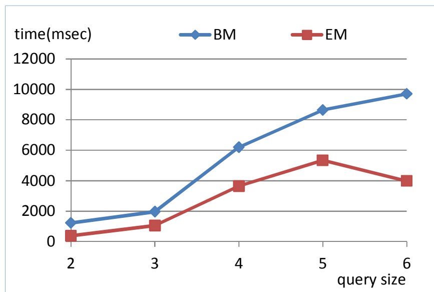
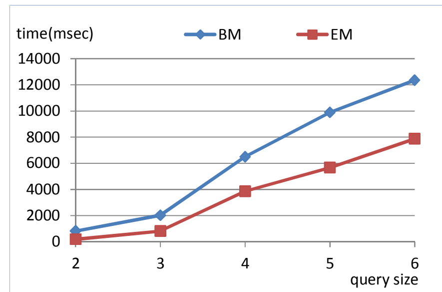


Fig. 6. Execution time of the search methods with varying k



(a) $p = 8$



(b) p = query size $\times$ 2

Fig. 7. Execution time of the proposed methods with varying p

In the second experiment, we evaluated performance of the proposed methods by measuring execution time in processing top-k queries. We generated and processed 20 test queries having 3 or 4 keywords. The parameter p is set to double the number of query keywords and periodic updates of the priority queues T and C are conducted after reading every 15,000 entries from inverted lists. We experimented with the same queries to find top-k answers varying k from 5 to 30. In Fig. 6, the results over IMDB dataset show that the average execution time of EM is about 4.7 times longer than that of BLINKS. This is mainly due to the overhead in finding answers in the generalized structure proposed in the paper. When comparing our basic method and enhanced method, the average execution time of EM based on the extended inverted lists reduces to about 38% of that of BM. Note that as k increases 6 times from 5 to 30, execution time of BM and EM increases only about 21% and 41%, respectively, and the performance gap between EM and BM gets wider about 10%.

Finally, we have evaluated performance of the proposed methods with respect to query size, i.e. the number of keywords in the query. We executed 20 test queries to find top-10 answers over IMDB dataset, varying the queries' size from 2 to 6. The parameter p is fixed to 8 in Fig. 7-(a) and set to double the query size in Fig. 7-(b), and update period of priority queues is 15,000. Fig. 7-(a) shows that as the query size grows from 3 to 6, the execution time of BM increases about 4.9 times while that of EM enlarges about 3.8 times. We can observe that the performance gap between EM and BM increases about 6.3 times and the execution time of EM is about 59% shorter than that of BM when the query size is 6. Thus EM is more efficient than BM for the queries of large size.

7 Conclusion

In this paper, we propose a new keyword search method for graph-structured databases. To find more effective top-k answers relevant to a given query, we suggest a generalized answer tree structure which has no constraint on the number of keyword nodes chosen for each keyword and selects a set of keyword nodes based on the relevance of the root to a query keyword contained in a specific node in the graph. For efficient top-k query processing based on the new answer structure, we propose an inverted list index which stores relevance and connectivity information on the nodes relevant to each keyword term. Then we provide a query processing algorithm exploiting the proposed inverted lists to find top-k answer trees most relevant to the given query. Furthermore, we also present Enhanced Method using an extended inverted list containing additional next relevance information in the list entry. It can estimate the relevance score of each node more closely to the correct value and find top-k answers more efficiently than Basic Method.

The experiments with real datasets and various test queries show that the precision of our approach is higher than that of BLINKS for most of the queries, especially for those with OR semantics. Thus, the proposed answer structure can satisfy users' information need better than the conventional ones. The performance of the proposed search algorithms is shown to be degraded compared to BLINKS mainly due to the

overhead incurred by adopting the extended answer structure. However, the execution performance of Enhanced Method based on the extended inverted list is much better than Basic Method, and it is scalable with respect to the number of answers to be found for top-*k* queries.

Acknowledgments. This work was supported by the Dongduk Women's University grant.

References

1. Amer-Yahia, S., Shanmugasundaram, J.: XML full-text search: Challenges and opportunities. In: 31st Int. Conf. on Very Large Data Bases, pp. 1368–1368 (2005)
2. Chen, Y., Wang, W., Liu, Z., Lin, X.: Keyword search on structured and semi-structured data. In: 2009 ACM SIGMOD Int. Conf. on Management of Data, pp. 1005–1010 (2009)
3. Dalvi, B.B., Kshirsagar, M., Sudarshan, S.: Keyword search on external memory data graphs. The Proceedings of the VLDB Endowment 1(1), 1189–1204 (2008)
4. Golenberg, K., Kimelfeld, B., Sagiv, Y.: Keyword proximity search in complex data graphs. In: 2008 ACM SIGMOD Int. Conf. on Management of Data, pp. 927–940 (2008)
5. He, H., Wang, H., Yang, J., Yu, P.S.: BLINKS: ranked keyword searches on graphs. In: 2007 ACM SIGMOD Int. Conf. on Management of Data, pp. 305–316 (2007)
6. Kim, H., Park, C.-S., Lee, Y.J.: Improving Keyword Match for Semantic Search. IEICE Trans. Inf. & Syst E94-D(2), 375–378 (2011)
7. Li, G., Ooi, B.C., Feng, J., Wang, J., Zhou, L.: EASE: an effective 3-in-1 keyword search method for unstructured, semi-structured and structured data. In: 2008 ACM SIGMOD Int. Conf. on Management of Data, pp. 903–914 (2008)
8. Bhalotia, G., Hulgeri, A., Nakhe, C., Chakrabarti, S., Sudarshan, S.: Keyword searching and browsing in databases using BANKS. In: 18th Int. Conf. on Data Engineering, pp. 431–440 (2002)
9. Ding, B., Yu, J.X., Wang, S., Qin, L., Zhang, X., Lin, X.: Finding top-k min-cost connected trees in databases. In: 23rd Int. Conf. on Data Engineering, pp. 836–845 (2007)
10. Hristidis, V., Gravano, L., Papakonstantinou, Y.: Effcient IR-Style keyword search over relational databases. In: 29th Int. Conf. on Very Large Data Bases, pp. 850–861 (2003)
11. Hristidis, V., Hwang, H., Papakonstantinou, Y.: Authority-based keyword search in databases. ACM Trans. Database Syst. 33(1), 1–40 (2008)
12. Hristidis, V., Papakonstantinou, Y.: DISCOVER: Keyword search in relational databases. In: 28th Int. Conf. on Very Large Data Bases, pp. 670–681 (2002)
13. Kacholia, V., Pandit, S., Chakrabarti, S., Sudarshan, S., Desai, R., Karambelkar, H.: Bidirectional expansion for keyword search on graph databases. In: 31st Int. Conf. on Very Large Data Bases, pp. 505–516 (2005)
14. Kimelfeld, B., Sagiv, Y.: Finding and approximating top-k answers in keyword proximity search. In: 25th ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems, pp. 173–182 (2006)
15. Liu, F., Yu, C.T., Meng, W., Chowdhury, A.: Effective keyword search in relational databases. In: 2006 ACM SIGMOD Int. Conf. on Management of Data, pp. 563–574 (2006)
16. Luo, Y., Lin, X., Wang, W., Zhou, X.: Spark: top-k keyword query in relational databases. In: 2007 ACM SIGMOD Int. Conf. on Management of Data, pp. 115–126 (2007)

17. Qin, L., Yu, J.X., Chang, L.: Keyword search in databases: The power of RDBMS. In: 2009 ACM SIGMOD Int. Conf. on Management of Data, pp. 681–694 (2009)
18. Yu, J.X., Chang, L., Tao, Y.: Querying communities in relational databases. In: 25th Int. Conf. on Data Engineering, pp. 724–735 (2009)
19. Yu, J.X., Qin, L., Chang, L.: Keyword search in relational databases: a survey. IEEE Data Engineering Bulletin 33(1), 67–78 (2010)
20. Hwang, F.K., Richards, D.S.: The Steiner tree problem. Networks 22(1), 55–89 (1992)
21. Buttcher, S., Clarke, C., Cormack, G.: Information Retrieval: Implementing and Evaluating Search Engine. MIT Press (2010)
22. Fagin, R., Lotem, A., Naor, M.: Optimal aggregation algorithms for middleware. Journal of Computer and System Sciences 66(4), 614–656 (2003)
23. Güntzer, U., Balke, W.-T., Kießling, W.: Towards efficient multi-feature queries in heterogeneous environments. In: 2001 Int. Symp. on Inf, pp. 622–628 (2001)
24. Bruno, N., Gravano, L., Marian, A.: Evaluating top-k queries over web-accessible databases. In: 18th Int. Conf. on Data Engineering, pp. 369–380 (2002)
25. Theobald, M., Weikum, G., Schenkel, R.: Top-k Query Evaluation with Probabilistic Guarantees. In: 30th Int. Conf. on Very Large Data Bases, pp. 648–659 (2004)
26. Best, H., Majumdar, D., Schenkel, R., Theobald, M., Weikum, G.: IO-Top-k: Index-access Optimized Top-k Query Processing. In: 32nd Int. Conf. on Very Large Data Bases, pp. 475–486 (2006)