Retrieving Top Relaxed Tightest Fragments for XML Keyword Search

Lingbo Kong∗, Rémi Gilleron, Aurélien Lemay
Mostrare, INRIA Futurs, Villeneuve d’Ascq, Lille, 59650 FRANCE
∗mLinking@gmail.com

Abstract. In this paper, we revisit the problem of retrieving meaningful result for XML keyword search. We begin by introducing the concept of Relaxed Tightest Fragment (RTF) as the basic result type, which preserves all the related keyword nodes implied in corresponding fragments in the XML data. This is different from the prior popular proposals like LCA (Lowest Common Ancestor), SLCA (Smallest LCA) variants. We then propose a ranking method, XKSMetaRank, to compute the meaningfulness of the RTFs, which takes into account the information contained both in the meta data and the XML data. Finally, we propose a flexible algorithm to cope with this problem, named as LISA/RTF (Layered Intersection Scan Algorithm). It takes a bottom-up layer-by-layer way to collect the RTFs and computes its meaningfulness. We implement our algorithm, and compare it with the method in [1]. Experiments on real and synthetic data verify the efficiency and effectiveness of our method.

1 Introduction

XML is rapidly emerging as the de facto standard for data representation and exchange of Web applications, such as Digital Library, Web service, and Electronic business. XML’s semi-structured and self-expressive properties have led to its widespread use in real applications. More and more XML data are generated. An XML data is usually modeled as a rooted labeled tree, \( T = (r, V, E, \Sigma, \lambda) \). \( V \) is the node set, \( r \) is a special node in \( V \) as the root, and \( E \) is the edge set. For each node \( v \in V \), \( \lambda \) is the function to assign a “label” in the string set \( \Sigma \) to \( v \), that is \( \lambda : v \in V \rightarrow l \in \Sigma \). There are two kinds of nodes in \( V \) according to whether the node contains “value” or not. Fig. 1 is an XML instance, in which every node has a node name (label), and every leaf node also contains the value (text). The integer sequence beside each node is the Dewey code [2], such as “0.2.0.1”. In this paper, \( T \) is simplified as \( (r, V, E) \).

Along with the widespread use of XML, adapting keyword search to XML data has become attractive, generalized as XML keyword search (XKS). Different from the keyword search on the text documents in tradition IR (Information Retrieval) [3], the basic idea of XKS is to locate the fragments (not the whole XML data), in each of which all the keywords of the keyword query occur. But it still remains an open problem, that is, what the answer should be like? There are many proposals based on LCA (Lowest Common Ancestor) concept [4–7], including Interconnected pairs [8], MLCA (Meaningful LCA) [9], SLCA [10–13], ELCA (Exclusive LCA) [14, 12, 15], GDMCT
Fig. 1. An XML instance

(Grouped Distance MCT – Minimum Connecting Tree) [16], MIU (Minimum Information Unit) [17], and meaningful fragment [18, 19]. Among them, the SLCA concept is popular. Regrettably it has two problems concluded in [1]: the false positive problem and false negative problem. The first problem means that the returned result may not be meaningful; the second problem means that not all the interesting fragments were included in the result. In view of this, [1] proposed CVLCA-based answer (Compact Valuable LCA); however, it had its own problems, concluded as the false negative problem and the incompleteness problem. Here, the incompleteness problem means that the answer may not contain all the related keyword nodes implied in the corresponding fragments in the XML data. In order to understand the two problems of CVLCA, we first clarify several concepts in the following three definitions: the content of a node, a node set or a tree, the concepts related to keyword node, and LCA.

**Definition 1 (Content of a node/node set/tree)** Let \( T = (r, V, E) \) be a rooted labeled tree, the content of a node \( v \in V \) is the string set that includes all the words in its label and its value, marked as \( C_v \).

The content \( C_S \) of a node set \( S \) in \( T \) is the content union of all the nodes in \( S \), defined as \( C_S = \bigcup \{ C_v \mid v \in S \} \). The content \( C_T \) of \( T \) is determined by \( C_V \).

**Definition 2 (Keyword node/keyword node set of a tree)** Given a rooted labeled tree \( T = (r, V, E) \), and a keyword query \( K = \{ w_1, \ldots, w_k \} \), a node \( v \in V \) is a keyword node if \( C_v \cap K \neq \emptyset \), where \( \emptyset \) is the empty set. \( T \) is said to cover all the given keywords in \( K \) if \( K \subseteq C_T \). \( K_i \) is used to record all the keyword nodes in \( T \) w.r.t (with respect to) the keyword \( w_i \) (1 ≤ \( i \) ≤ \( k \)).

**Definition 3 (Internal node set & LCA)** Given a rooted labeled tree \( T \), \( u \) and \( v \) are two nodes of \( T \). If there is a path from node \( u \) to node \( v \) (denoted as \( u \rightarrow v \)), node \( u \) is called an ancestor of \( v \), denoted as \( u \prec v \); meanwhile \( v \) is called a descendant of \( u \), denoted as \( v \succ u \). The internal node set of the path from \( u \) to \( v \) is the node set which is composed of all the nodes on the path, denoted as \( I(u, v) \), \( (u, v) \in I(u, v) \). If \( u \prec v \) and \( \left| I(u, v) \right| = 2 \), \( u \) is the parent of \( v \), and \( v \) is the child of \( u \).
Given a node set $S$ of $T$, $I(u,S)$ stands for the node set which contains all the internal nodes from $u$ to every node in $S$, that is $I(u,S) = \bigcup I(u,v_i)$ where $1 \leq i \leq |S|$, and $v_i \in S$. The node $u$ in $T$ is called the LCA node of all those nodes in $S$ iff the following two conditions hold: (1) $u$ is the ancestor of all the nodes in $S$; (2) except for $u$, there is no other node in $I(u,S)$ that is also the ancestor of all the nodes in $S$.

Without loss of generality, the LCA node $LCA(S_1, \ldots, S_n)$ of $n$ node sets $S_1, \ldots, S_n$ is defined as $LCA(S_1 \cup \cdots \cup S_n)$.

When using those concepts on XML keyword search, we have several notations here, which are used later. Given an XML document $T$ and the keyword query $K = \{w_1, \ldots, w_k\}, \{(\{v_1, \ldots, v_k\}|v_i \in K_i, where 1 \leq i \leq k\}$ contains all the distinct keyword node sets, each of which covers the keyword query $K$. We denote it as $C_K^T$, and without loss of generality $C_K^T$ (1 \leq j \leq \prod_{i=1}^k |K_i| if \bigcap_{i=1}^k K_i = \emptyset)$ represents an instance of $C_K^T$. And $C_K^T$ corresponds to the keyword node subset in $C_K^T$ that contains the key-word node $w_i$, $LCA(C_K^T)$ stands for the LCA node of the keyword node set $C_K^T$, $LCA(C_K^T)$ is defined as $\{LCA(C_K^T)|C_K^T \in C_K^T\}$. The node set $\{C_K^T \cup \{LCA(C_K^T)\}|C_K^T \in C_K^T\}$ comprises all the basic result sketches of the prior XKS proposals, either in the way of returning only the LCA nodes (such as XRank, SLCA, MLCA, ELCA), or returning the fragments (including MIU, GDMCT, Interconnected pairs, CVLCA). Among them, SLCA is a popular concept to filter those LCA nodes. It is described as $SLCA(C_K^T)$ = $\{v|v \in LCA(C_K^T)\}$ and $\exists u, v \prec u, v \in LCA(C_K^T)$.

Now we continue the discussion before. Li et al [1] investigated the prior proposals and concluded two problems of SLCA: the false positive problem and the false negative problem. Here are the examples for them. When doing keyword query “Yi Chen XML View” on the XML instance in Fig. 1, the methods with false positive problem will re-view the fragments (including MIU, GDMCT, Interconnected pairs, CVLCA). Among them, SLCA is a popular concept to filter those LCA nodes. It is described as $SLCA(C_K^T)$ = $\{v|v \in LCA(C_K^T)\}$ and $\exists u, v \prec u, v \in LCA(C_K^T)$.

In view of those problems, Li et al [1] proposed VLCA concept (Definition 5) to overcome the false positive problem, which was used to guarantee the meaningfulness of the answer. The core of the VLCA concept is the Homogenous concept, described in Definition 4. Its essence is to filter out those candidates in $C_K^T$, in which there are distinct internal nodes in $I(LCA(C_K^T), C_K^T) \prec C_K^T$ who have same label. And they further proposed CVLCA (Definition 6) to overcome the false negative problem, and suggested to return the CVLCA node and the related keyword nodes with their labels as the result.

\footnote{[15] also noticed this problem, and proposed its own method to retrieve those nodes untouched by SLCA. However, its result was only the set of those filtered LCA nodes, and did not consider the meaningfulness.}
Definition 4 (Homogenous/Heterogenous) Given two nodes \( u, v \), and \( n = LCA(u, v) \). Let \( nSet = uSet \cup vSet - \{u, v\} \), \( u \) and \( v \) are heterogenous (denoted as \( u \sim v \)), iff \( \exists u', v' \in nSet \), the labels of \( u' \) and \( v' \) are same. On the contrary, \( u \) and \( v \) are homogenous (denoted as \( u \sim v \)), iff \( \forall u', v' \in nSet \) their labels are not same.

Definition 5 (Valuable LCA) Given \( k \) nodes \( v_1, \ldots, v_k \), \( u = LCA(v_1, \ldots, v_k) \). \( VLCA(v_1, \ldots, v_k) = u \) holds iff these \( k \) nodes are homogenous.

Definition 6 (Compact VLCA) Given a rooted labeled tree \( T \) and the keyword query \( K = \{w_1, \ldots, w_k\} \), \( K_i \) records all the keyword nodes w.r.t the keyword \( w_i \) \( (1 \leq i \leq k) \). For each keyword node set instance \( \mathcal{C}_{K_j}^T \), \( u = LCA(\mathcal{C}_{K_j}^T) \) is called to dominate \( \mathcal{C}_{K_j}^T \), if \( u \) is the descendant of or equals to \( LCA(v_1, \ldots, v_{i-1}, \mathcal{C}_{K_j}^T, v_{i+1}, \ldots, v_k) \), where \( \forall v_1 \in K_1, \ldots, \forall v_{i-1} \in K_{i-1}, \forall v_{i+1} \in K_{i+1}, \ldots, \forall v_k \in K_k \). \( u \) is a Compact LCA w.r.t \( \mathcal{C}_{K_j}^T \), if \( u \) dominates each \( \mathcal{C}_{K_j}^T \), \( (1 \leq i \leq k) \). \( u \) is a Compact VLCA (CVLCA), if \( u \) is a compact LCA and it is also a VLCA.

Regrettably, the VLCA concept based on the “homogenous” idea is not universal. In some situations, it has the false negative problem. We use Example 1 to illustrate this problem.

![Fig. 2.](image)

(a): False negative of VLCA; (b) Incompleteness of CVLCA; (c) Two RTF instances

Example 1 (False negative problem of VLCA) We first introduce the positive example for VLCA concept. Back to the XML instance in Fig. 1, the nodes “name (0.2.0.0.0.0)” \((v_1)\) and “ref (0.2.0.3.0)” \((v_2)\) are homogenous. Their LCA node is “article (0.2.0)” \((n_1)\), and the node sets in the paths of \( n_1 \rightarrow u_1 \) and \( n_1 \rightarrow v_1 \) are \( \{\text{“author (0.2.0.0.0)”, “authors (0.2.0.0)”}\} \) \((u_1\text{Set})\) and \( \{\text{“reference (0.2.0.3)”}\} \) \((v_1\text{Set})\). We can see that any distinct node in \( n_1\text{Set} = u_1\text{Set} \cup v_1\text{Set} = \{\text{“author (0.2.0.0)”}, “authors (0.2.0.0)”\} \), “reference (0.2.0.3)”\} has its unique label.

However, in some situations, this homogenous concept is not appropriate to filter the LCA nodes so as to guarantee the meaningfulness. Here is a keyword query – “Liu Chen keyword” \((KQ_1)\). When we run it on the XML data in Fig. 1, it is clear that the most interesting fragment is that in Fig 2(a). But we never get this according to the VLCA concept. This is because the two nodes “name (0.2.1.0.0.0)” \((v_1)\) and “name
(0.2.1.0.1.0)" (v2) are heterogenous according to Definition 4. We can see that their LCA node is “authors (0.2.1.0)" (u), and the node sets on the paths of \( u \rightarrow v_1 \) and \( u \rightarrow v_2 \) are \{“author (0.2.1.0.0)”\} (v1 Set) and \{“author (0.2.0.0)”\} (v2 Set). It is clear that the two nodes – “author (0.2.1.0.0)” \( \in v_1 \text{Set} \) and “author (0.2.1.0.1)” \( \in v_2 \text{Set} \) – have the same label “author”. But, the fragment in Fig 2(a) is definitely a meaningful answer.

As for the CVLCA-based answer, its motivation is to collect those neglected but interesting fragments. However, there is a critical thing it did not pay much attention to. That is, it did not incorporate all the related keyword nodes w.r.t the CVLCA node in the result. Example 2 demonstrates the incompleteness problem.

**Example 2 (Incompleteness problems of CVLCA)** When running keyword query “Guo keyword” (KQ_2) on the XML data in Fig. 1, there are two CVLCA nodes – “article (0.2.0)” and “ref (0.2.0.3.0)”. [1] directly returns the node CVLCA node “ref (0.2.0.3.0)” with its label and value as one of the final result (Fig. 2(c)(2)). As for the other CVLCA node, [1] just returns the CVLCA node and its related keyword nodes – “name (0.2.0.0.0.0)” and “title (0.2.0.1)" with the texts. Its sketch is shown in Fig. 2(b). We should point out that, in fact, the two nodes – “authors (0.2.0.0)” and “author (0.2.0.0.0)” in Fig. 2(b), are not contained in the final CVLCA-based answer. This is the first incompleteness problem. Intuitively, it is easy to cope with this by extending the method to collect those internal nodes. But the second is not so easy to overcome for CVLCA.

When there are more than one nodes containing same keyword in the subtree rooted at the CVLCA node, the second incompleteness problem happens. Fig. 2(c)(1) shows the whole subtree rooted at the CVLCA node “article (0.2.0)” following the keyword query \( KQ_2 \). Compared with Fig. 2(b), Fig. 2(c)(1) preserves one more node – “abstract (0.2.0.2)”, which also contains the keyword “keyword”. We can see that it will return more interesting information when taking Fig. 2(c)(1) as the result rather than the CVLCA-based answer. In fact, the fragment in Fig. 2(c)(1) is a RTF (see Section 2).

Besides, the CVLCA concept is not adequate to confine the interesting fragment. Intuitively once the interesting target fragment is determined, the answer from this fragment should be unique. However, if there are many keyword nodes with same label in the fragment, the answer following CVLCA is not unique. Back to the keyword query \( KQ_2 \), the LCA node of the keyword nodes “name (0.2.0.0.0.0)”, “abstract (0.2.0.2)” is also “article (0.2.0)”. And it clearly satisfies both the VLCA concept and the Compact LCA concept.

In this paper, we first propose the RTF concept to represent the result for the XML keyword search so as to cope with the incompleteness problem in CVLCA. A RTF contains all the related keyword nodes, and the corresponding internal nodes on the paths from the RTF’s root node to the keyword nodes. As for the false negative problem, we propose a ranking method, XKSMetaRank, to filter the retrieved RTFs so that the top RTFs are more meaningful. Finally, we propose a flexible algorithm to retrieve and rank the RTFs together, named as LISA/RTF. The contributions of this paper can be concluded as follows:
We propose the RTF concept to represent the result for the XML keyword search. It follows the motivation of the CVLCA concept, and collects those untouched keyword nodes and the internal nodes. By this idea, it is easy to reconstruct the interesting content implied in the raw fragments. Consequently, it is also convenient for users to navigate/browse or ranking/filtering function later since it preserves enough hints.

We propose XKSMetaRank to filter the RTFs so as to guarantee the meaningfulness of the top RTFs. Inspired by XRank [14], XKSMetaRank takes into account the organization information in DTD and XML data together, while XRank considered only the organization information in XML data. Here we conceive that there is a DTD (Document Type Definition) for any XML data, since it is easy to extract one if there is no DTD.

We propose an algorithm to achieve and filter the RTFs for the given keyword query, named as LISA/RTF. It takes a bottom-up way to collect the RTFs layer by layer. During the retrieving of the RTF, LISA/RTF records the connections between the candidate RTF’s root and all the keyword nodes in that RTF. Consequently, it could directly figure out the XKSMetaRank value once the RTF is produced.

We implement our algorithm, and do extensive experiments on both real and synthetic datasets. The results verify that LISA/RTF could enumerate all the RTFs, and has competent performance.

The rest of this paper is organized as follows. Section 2 presents the formal description of the RTF. Section 3 illustrates the idea of XKSMetaRank. The implementation of our proposals, LISA/RTF, is discussed in Section 4. Section 5 illustrates the experimental result. Section 6 briefly reviews the related work. Finally, Section 7 concludes this paper, and sheds light on future work.

2 Relaxed Tightest Fragment

In this section, we introduce our proposal – Relaxed Tightest Fragment (RTF), which is the basic result for the given keyword query. The motivation of the RTF is to overcome the incompleteness problems in prior proposals, such as SLCA, CVLCA, GDMCT. A RFT contains all the related interesting nodes, including the keyword nodes and the internal nodes. The XKSMetaRank in Section 3 is used to filter out the meaningful RTFs from the retrieved RTFs.

According to the discussions in Section 1, there are mainly two kinds of incompleteness problems: missing of internal nodes and missing of other related keyword nodes. Since a Dewey code, which is popular in XKS, spontaneously preserves the Dewey codes of all its ancestors, the problem of missing internal nodes could be coped with after locating the keyword nodes and their LCA node. So the major task here is to propose a description of the result, which not only overcomes the false negative problem of SLCA mentioned in [1], but also preserves all the related keyword nodes in the raw fragment.

From Section 1, we can see that the keyword node combination set $c^T_{K}$ determines all the candidates of the prior proposals when running a keyword query $K$ on the XML data $T$. The difference among the prior methods is only the strategy of how to filter
those candidates in \( C_{TK} \). So the composition strategy of \( C_{TK} \) is critical for the final result. When we reconsider the description of \( C_{TK} \), we conceive that the concept of \( C_{TK} \) could not be used for the situation here. This is because every element in \( C_{TK} \) takes into account only one keyword node from the corresponding keyword node set. Consequently, it is impossible that two keyword nodes with same keyword occur in any \( C_{TC,K},j \), even though they should be. For example, it is clear that Fig. 2(c)(1) should be an interesting answer when running \( KQ_2 \) on the XML instance in Fig. 1. But the two keyword nodes “title (0.2.0.1)” and “abstract (0.2.0.2)” will never coexist in the same candidate.

In order to cope with this, we should extend the node picking strategy, and the picking now should be based on the keyword node subset (not just picking one node) from each keyword node set \( K_i \). We mark this new combination set as \( EC_{TK} \), defined in Definition 7.

**Definition 7 (Extended Keyword Node Combination Set)** Given a rooted labeled tree \( T \), and the keyword query \( \mathcal{K} = \{w_1, w_2, \ldots, w_k\} \), \( K_i \) records all the keyword nodes w.r.t the keyword \( w_i \). The set \( V_i \) contains all the valid keyword node subsets of \( K_i \), which means \( V_i = 2^{K_i} - \{\emptyset\} \), where \( 2^{K_i} \) is the power set of \( K_i \), and \( \emptyset \) is the empty set. The extended keyword node combination set on all \( K_i \) \((1 \leq i \leq k)\) is defined as \( \{V \mid V = \bigcup_{i=1}^{k} v_i \text{ where } v_i \in V_i\} \), denoted as \( EC_{TK} \).

Without loss generality, \( EC_{TK,j} \) \((1 \leq j \leq \prod_{i=1}^{k} (2^{|K_i|} - 1), \text{ if } \bigcap_{i=1}^{k} K_i = \emptyset \) represents an element of \( EC_{TK} \), and \( EC_{TK,j,i} \) is the keyword node set w.r.t keyword \( w_i \) in \( EC_{TK,j} \).

The basic idea of \( EC_{TK} \) is to enumerate all the combinations of the corresponding keyword node sets. It contains the prior \( C_{TK} \), and it also covers the situation that many keyword nodes with same keyword occur in some \( EC_{TK,j} \). Besides, it is clear that the LCA concept shown in Definition 3 could apply to \( C_{TK,j} \). Example 3 expresses these properties.

**Example 3 (Illustration of \( EC_{TK} \))** We run the keyword query \( \mathcal{K} = KQ_2 \) on the XML data in Fig. 1 again. We take “Guo” as \( w_1 \), and “keyword” as \( w_2 \). When running \( KQ_2 \), the keyword node set corresponding to \( w_1 \) is \( K_1 = \{\text{name (0.2.0.0.0.0)}, \text{ref \ldots}\} \), and
Given a rooted labeled tree $T$ and the keyword query $K = \{w_1, w_2, \ldots, w_k\}$, $K_i$ records all the keyword nodes w.r.t the keyword $w_i$, $V_i$ is the set of the valid keyword node subsets of $K_i$, and $\mathcal{E}^T_{K_i}$ is the extended keyword node combination set on all $K_i$. 

For each element $\mathcal{E}^T_{K_i,j}$, $\text{LCA}(\mathcal{E}^T_{K_i,j})$ is the LCA node based on the nodes in $\mathcal{E}^T_{K_i,j}$, and $I(\mathcal{E}^T_{K_i,j}) = \{\text{LCA}(\mathcal{E}^T_{K_i,j}), \mathcal{E}^T_{K_i,j}\}$ records all the internal nodes determined by the keyword node set $\mathcal{E}^T_{K_i,j}$ and its LCA node. $V_j\{i\}$ stands for the valid subsets of $\mathcal{E}^T_{K_i,j}\{i\}$, namely $V_j\{i\} = 2^{\mathcal{E}^T_{K_i,j}\{i\}} \setminus \{\emptyset\}$.

The relaxed tightest fragment problem of XKS is to filter out all the RTFs from $\mathcal{E}^T_{K_i}$. Definition 8 proposes the rules to filter out RTFs from $\mathcal{E}^T_{K_i}$, which not only copes with the false negative problem of SLCA, but also overcomes the incompleteness problems of CVLCA. The first rule is used to make sure there is no other keyword node subset in $\mathcal{E}^T_{K_i,j}$, which covers the given keyword query, but whose LCA node is different with $\text{LCA}(\mathcal{E}^T_{K_i,j})$. The second rule is make sure that the keyword node set $\mathcal{E}^T_{K_i,j}$
is the maximum set whose LCA node is \( LCA(\mathcal{EC}_K^{T,j}) \). As for the third rule, it means any keyword node in \( \mathcal{EC}_K^{T,j} \) can not be included in other keyword node sets whose LCA node is the descendant of \( LCA(\mathcal{EC}_K^{T,j}) \). Besides, any RTF satisfying Definition 8 clearly covers the given keyword query \( \mathcal{K} \). We use Example 4 to illustrate the concept of RTF.

**Example 4 (Illustration of RTF concept)** Following Example 3, there are only two keyword combination sets in \( \mathcal{EC}_K^{T} \) satisfying the RTF concept, \( \{r\} \) and \( \{n, t_1, a\} \). All the other combination sets do not satisfy all those three conditions.

For instance, \( \{n, t_2\} \) does not satisfy the third condition, because the LCA node of \( \{n, t_1, a\} \) is node “article (0.2.0)”, while node “article (0.2.0)" is a descendant node of the LCA node “Articles (0.2)" of \( \{n, t_2\} \).

As for the set \( E\mathcal{C}_{K,j}^{T} = \{n, t_1\} \), it is not a RTF either because it does not satisfy the second condition. This is because we could find a node set \( V' = \{t_1, a\} \in V_2 - \{t_1\} \), and \( \{t_1\} \subset V' \). We can learn that the LCA node of \( V' \cup \{n\} \) is “article (0.2.0)”, which is also the LCA node of \( \{n, t_1\} \).

As for the set \( E\mathcal{C}_{K,j}^{T} = \{r, t_1\} \), it is not a RTF because of the first rule. Here, \( E\mathcal{C}_{K,j}^{T}|_1 \) is \( \{r\} \) and \( E\mathcal{C}_{K,j}^{T}|_2 \) is \( \{r, t_1\} \). We have \( LCA(\mathcal{EC}_K^{T,j}) = “article (0.2.0)” \). The set \( \{r\} \) is a subset of \( E\mathcal{C}_{K,j}^{T}|_2 \) and \( LCA(\mathcal{EC}_K^{T,j}|_1, \{r\}) \) is node r (“ref (0.2.0.3.0)”). Clearly, node “article (0.2.0)” is different with node “ref (0.2.0.3.0)”.

Like the LCA-based methods, there is also a brute-forth method to retrieve all the RTFs for the given keyword query. The idea is to construct the \( \mathcal{EC}_K^{T} \), and then filters out the RTFs based on the rules in Definition 8. However, it is clear that this brute-forth method is inefficient, because the number of candidate RTFs is \( \prod_{i=1}^{k} (2^{|K_i|} - 1) \) when \( \bigcap_{i=1}^{k} K_i = \emptyset \). In view of this, LISA/RTF is proposed in Section 4.

### 3 XKSMetaRank

In this section, we introduce our ranking method for the retrieved RTFs, XKSMetaRank, which not only overcomes the false negative problem of VLCA, but also guarantees the top RTFs are more meaningful. It has three steps: the collection of label’s weight of an element in DTD (Section 3.1), the weight of a keyword node (Section 3.2) and the computation of a RTF’s meaningfulness (Section 3.3). We first present the related definitions in corresponding sections, and use a concrete example (Example 5) to illustrate the whole computation of XKSMetaRank for the RTF.

Here we conceive that there is a DTD (Document Type Definition) for any XML data, since it is easy to extract one if there is no DTD. Fig. 3(a) shows the DTD segment derived from the XML instance in Fig. 1.
3.1 Collection of label’s weight

According to the DTD specification\textsuperscript{2,3}, a DTD defines the XML document structure with a list of legal elements and attributes. The relationship between two DTD elements determines the semantic/meaningfulness of their instances in XML data. So the first step of XKSMetaRank is to collect relationships among the elements in DTD, especially the relationship between an element and its children elements. The attributes of an element have similar role as its child elements. This is because the element together with all its child elements and attributes determines a semantic block in XML data. Since the name of an element or an attribute is the label of corresponding node of the XML tree, we directly use label to stand for the name. Consequently the first step of XKSMetaRank is to collect the number of child elements and attributes of an element.

Since it is the different label of the element that represents the semantic of the XML data, we only pay attention to the child elements and attributes with distinct names/labels in the collection. We use $E(\lambda)$ to denote that number, where $\lambda$ is a distinct label. So the corresponding number $E(\lambda)$ of an element with label $\lambda$ is the number of distinct labels in the child elements and attributes. So the element with notations like “*” (zero or more occurrence) and “?” (zero or one occurrence) only contributes 1 in its $E(\lambda)$. It is easy to determine the number of attributes – just counting their number (denoted as $A(\lambda)$). While for the collecting of the number of child elements, there are three cases, corresponding to the three element organization types in DTD. “elemname” stands for the name of an element.

- **Sequence**: $E(\text{elemname}) = k$ if the element is defined as following:

  ```xml
  <!ELEMENT elemname (child\textsubscript{1},\textellipsis, child\textsubscript{k})>
  ```

  If we use seq to stand for the sequence “(child\textsubscript{1},child\textsubscript{2},\textellipsis, child\textsubscript{k})”, the notation $E(\text{seq})$ is equal to $E(\text{elemname})$.

- **Or**: $E(\text{elemname}) = 1$ if the element is defined as following:

  ```xml
  <!ELEMENT elemname (elem\textsubscript{1} | elem\textsubscript{2} | \textellipsis )>
  ```

\textsuperscript{2} http://www.w3.org/TR/REC-xml/
\textsuperscript{3} http://www.w3schools.com/dtd/default.asp
• Mix: $\mathcal{E}(\text{elemname}) = \max \{ \mathcal{E}(\text{seq}_1), \ldots, \mathcal{E}(\text{seq}_k) \}$ where $\text{seq}_1, \ldots, \text{seq}_k$ are the sequences defined in the mixture element following:

```
"<!ELEMENT elemname (seq1 | \ldots | seq_k)>"
```

After the collection of the label number $\mathcal{E}(\lambda)$ for an element in DTD, and the attribute number $\mathcal{A}(\lambda)$, the weight of the label (the name of the element) $\mathcal{L}(\lambda)$ is defined as $\frac{1}{\mathcal{E}(\lambda) + \mathcal{A}(\lambda)}$, which is used later in Section 3.3.

### 3.2 Weight of a keyword node

Besides the weight of a distinct label from DTD, another important parameter is the weight of a keyword node in the XML data following that DTD. It is used to represent the weight of the content of the keyword node w.r.t the given keyword query. Here we introduce a simple weight: the naïve weight in Definition 9.

**Definition 9 (Naïve weight of a keyword node)** Given a rooted labeled tree $T$ and a keyword query $K$, $v$ is a keyword node, and its content is $C_v$. The naïve weight $K_N(v)$ of $v$ is defined as $K_N(v) = |C_v \cap K|$.

### 3.3 Meaningfulness computation of a RTF

After collecting the weights of the labels and the keyword nodes, XKSMetaRank is ready to compute the meaningfulness of a RTF which satisfies the given keyword query $K$. Like the rank computation in XRank [14], the meaningfulness of the RTF corresponds to the weight of the RTF’s root by transferring the weights of the keyword nodes. During the transferring, XKSMetaRank takes into account the label weights of the internal nodes in the RTF.

**Definition 10 (Total weight of a node in a RTF)** Given a RTF $T$, $u$ is a node of $T$. There are three cases to compute the total weight $T(u)$ of the node $u$ based on the conditions: whether $u$ has children in the RTF or not, and whether $u$ is also a keyword node or not. It is defined in a recursive way.

1. If $u$ has no child, this means $u$ is a pure keyword node in the RTF; then $T(u) = K_N(u)$. And, the label weight $\mathcal{E}(u)$ of this node is assigned as 1 no matter what value it is collected from the DTD.
2. If $u$ has $n$ children in the RTF – $v_1, \ldots, v_n$, but it is not a keyword node, $\lambda(v_i)$ is the label of the node $v_i$ $(1 \leq i \leq n)$, and $\mathcal{L}(\lambda(v_i))$ is the label weight of node $v_i$. We use $T(v_i)$ to stand for the total weight of the node $v_i$. Then $T(u)$ is defined as $\frac{1}{2} \left( \sum_{i=1}^{n} \mathcal{L}(\lambda(v_i)) \ast T(v_i) \right)$.
3. If $u$ has $n$ children in the RTF – $v_1, \ldots, v_n$, and it is also a keyword node, then $T(u)$ is defined as $K_N(u) + \frac{1}{2} \left( \sum_{i=1}^{n} \mathcal{L}(\lambda(v_i)) \ast T(v_i) \right)$. 
Definition 10 presents the fundamental computation of XKSMetaRank. It focuses on the weight computation of a node – the elementary unit of a RTF. There are two things that need more explanation. The first one is that the label weight of a pure keyword node is assigned with 1. This rule is implicitly consistent with the intuition that the nodes other than the nodes in the RTF do not have contribution to the meaningfulness once the RTF is determined. The other is that \( \frac{1}{2} \) in the latter two cases is used as a decay parameter. It implies the contribution of a node to its parent is the half of its total weight. Of course the decay parameter could be others, such as an exponential decay function \( e^{-\lambda t} \) whose value changes with the variable \( t \) of length from the RTF’s root to a node, where \( \lambda \) is a parameter determined by users.

Based on the definition of the total weight of a node, it is straightforward to define the meaningfulness/weight of a RTF. We have Definition 11.

**Definition 11 (Meaningfulness of a RTF)** Let \( T \) be a XML data, and \( \mathcal{K} = \{ w_1, w_2, \ldots, w_k \} \) is a keyword query, \( \mathcal{F} \) is one of the RTFs that satisfies Definition 8. \( \mathcal{F}_K \) is the keyword node set in \( \mathcal{F} \). The notation \( r_\mathcal{F} \) represents the root node of \( \mathcal{F} \), and \( T(r_\mathcal{F}) \) is the total weight of the RTF’s root \( r_\mathcal{F} \). The meaningfulness of \( \mathcal{F} \), \( XKSMetaRank(\mathcal{F}) \), is defined by Equation (1).

\[
XKSMetaRank(\mathcal{F}) = \frac{T(r_\mathcal{F})}{|\mathcal{F}_K|}
\]  

From the above definitions, we can see that the meaningfulness \( XKSMetaRank(\mathcal{F}) \) of a RTF not only considers the information in meta data by the label weight, but also takes into account the organization information in the XML data by the weight transferring. Example 5 illustrates the whole computation of XKSMetaRank.

**Example 5 (Illustration of XKSMetaRank)** When given the XML data of Fig. 1 and its DTD segment of Fig. 3(a), the number for every distinct label (namely the name of the element or attribute) could be collected according to the Section 3.1. Fig. 3(b) presents the numbers. And the numbers for other labels that are not listed in Fig. 3(b) are 1. Based on these numbers, we could get the label weight \( \mathcal{L}(\lambda) \) for every label \( \lambda \) according to the definition \( \mathcal{L}(\lambda) = \frac{1}{E(\lambda) + A(\lambda)} \). For example, the label weight for label “author” is \( \mathcal{L}(author) = \frac{1}{2} \), and \( \mathcal{L}(authors) = 1 \). While \( \mathcal{L}(name) = \mathcal{L}(title) = \mathcal{L}(abstract) = 1 \).

We use two keyword queries \( \mathcal{K}_Q_1 \) and \( \mathcal{K}_Q_2 \) here: “Liu Chen keyword” and “Guo keyword” respectively. When running \( \mathcal{K}_Q_1 \) and \( \mathcal{K}_Q_2 \) on the XML instance in Fig. 1, the RTF for \( \mathcal{K}_Q_1 \) is shown in Fig. 4(a), and the two RTFs for \( \mathcal{K}_Q_2 \) are shown in Fig. 4(b). We first present the meaningfulness computation of the RTF for \( \mathcal{K}_Q_1 \), which illustrates the advantage of our XKSMetaRank of overcoming the false negative problem of VLCA. We then discuss the meaningfulness computations of the two RTFs for \( \mathcal{K}_Q_2 \), which demonstrate the ability of our XKSMetaRank of guaranteeing the top RTFs more meaningful.

For the RTF in Fig. 4(a), both keyword nodes with keyword “Liu” and “Chen” have the same total weight 1 according to the first case in Definition 10. Since their parents (the two nodes with label “author”) only have one child in the RTF, the total weight values of...
their parents are same, as $\frac{1}{2} \times 1$, following the second case of Definition 10. As for the node with label “authors”, it has two children, which have same total weight $\frac{1}{2}$ and label weight $L(\text{author}) = \frac{1}{3}$. Consequently, its total weight is $\frac{1}{2} \times (\frac{1}{3} \times \frac{1}{2} + \frac{1}{3} \times \frac{1}{2}) = \frac{1}{6}$, where $\frac{1}{3}$ corresponds to the label weight of its children’ label $- L(\text{author}) = \frac{1}{3}$. It contributes to the RTF’s root with $\frac{1}{2} \times (1 \times \frac{1}{6}) = \frac{1}{12}$ according to the second case of Definition 10. Similarly we can get the contribution of the other keyword node with label “title” to the RTF’s root as $\frac{1}{2} \times 1 = \frac{1}{2}$. Finally, the meaningfulness of this RTF for $KQ_1$ is $\frac{1}{12} + \frac{1}{2} = \frac{7}{12} \approx 0.583$.

As for the RTFs in Fig. 4(b) for $KQ_2$, we can get their meaningfulness following the similar computation of the RTF in Fig. 4(a) for $KQ_1$. We have $\frac{25}{36} \approx 0.694$ for the RTF$_1$ of Fig. 4(b)(1), , and 2 for the RTF$_2$ in Fig. 4(b)(2). These two values clearly express that RTF$_2$ is more meaningful than RTF$_1$.

Finally, even though we just illustrate our XKSMetaRank based on DTD, it could be directly applied to XML schema, which is another specification of rules used for organization XML data. Besides the naïve weight of a keyword node here, it is also convenient to use other weight proposals into our XKSMetaRank, such as the popular $TF \times IDF$ concept [3].

4 LISA/RTF

In this section, we introduce the algorithm of LISA/RTF (Layered Intersection Scan Algorithm) to retrieve and rank RTFs for the given keyword query $K$.

From the above discussions, we can see that one of the basic task of RTF problem is to filter out all the RTFs from the candidates, each of which contains all the related keyword nodes. This requires the corresponding method to compute the RTF differently compared with the prior methods like SLCA, CVLCA, ELCA. That is, the keyword node set of a candidate is the combination of keyword node subsets of the corresponding keyword node sets. While, the prior methods only consider the combination of keyword nodes, in which there is only one node from the corresponding keyword node set. Even though the brute-forth method could enumerate all the candidates and filter out the RTFs, the large number of the candidates leads to its inefficiency. In order to overcome this problem, we propose LISA/RTF, which collects the RTFs layer by layer beginning from the bottom. We have following observations shown in Example 6.

Example 6 (Observations for LISA/RTF) Given a keyword query, it is easy to collect the Dewey code set for each keyword. Following Example 5, the Dewey code sets for $KQ_2 = \text{“Guo keyword”}$ is shown in Fig. 4(c), where we take “Guo” as $w_1$, and “keyword” as $w_2$. The Dewey code set $D_1$ corresponds to $w_1$ and it has two Dewey codes $D_{11}$ and $D_{12}$; while the Dewey code set $D_2$ corresponds to $w_2$ and it has four Dewey codes $D_{21}, \ldots, D_{24}$ according to Fig. 1. The integers at the top of Fig. 4(c) demonstrate the level information, which is used to reflect the length information of the corresponding Dewey codes and the prefixes.

\footnote{http://www.w3.org/TR/xmlschema11-2/}
After grouping the Dewey codes of the keyword nodes like Fig. 4(c), we have following observations.

1) When given an level integer \( j \), it is easy to collect all the corresponding prefixes (with length \( j \)) of the Dewey codes in each Dewey code set (including the raw Dewey codes whose length is \( j \)). According to the properties of Dewey encoding\(^3\), every prefix is also a Dewey code. We denote the prefix set at level \( j \) from Dewey code set \( D_i \) as \( D_i^j \).

For example, when given 3, the prefix set of \( D_1 \) is the Dewey code set \( D_1^3 = \{ \text{"0.2.0"} \} \); while the prefix set of \( D_2 \) is the Dewey code set \( D_2^3 = \{ \text{"0.2.0", "0.2.1"} \} \). There are two Dewey codes in \( D_1 \) connected to \"0.2.0\". They are \( D_{11} \) and \( D_{12} \). As for \( D_2^3 \), \( D_{21} \), \( D_{22} \) and \( D_{23} \) are the descendant nodes of \"0.2.0\"; and \"0.2.1\" only has \( D_{24} \).

2) If the intersection of all the prefix sets of level \( j \) is not empty, every element of the intersection and the keyword nodes connected to it correspond to a RTF candidate, which covers the given keyword query.

Following (1), the intersection of \( D_1^3 \) and \( D_2^3 \) is \{\"0.2.0.3.0\"\}. From (1), we can collect all the related Dewey codes of the keyword nodes for \"0.2.0.3.0\". They are \( D_{11} \), \( D_{12} \) from \( D_1 \), and \( D_{21} \), \( D_{22} \), \( D_{23} \) from \( D_2 \). We can see that the node w.r.t \"0.2.0\" is the LCA node of all those keyword nodes. And the content of those nodes together covers all the keywords in \( KQ_2 \).

Those nodes could compose a RTF candidate, but it is not a real RTF according to the third rule in Definition 8. This is because there is a RTF based on node \( D_{12} \) from \( D_1 \) and node \( D_{23} \) from \( D_2 \), whose LCA node corresponds to the Dewey code \"0.2.0.3\". Clearly, \"0.2.0.3.0\" is a descendant of \"0.2.0\".

3) Given an integer \( j \), if the RTFs, whose roots are located at levels lower than \( j \), have been collected, and the Dewey codes related to those RTFs have been deleted from \( D_i \) (\( 1 \leq i \leq k \)), the intersection of the prefix sets at level \( j \) based on the renewed \( D_i \) (\( 1 \leq i \leq k \)) contains all the roots of the RTFs at this level \( j \), if it is not empty.

We use Fig. 5 to demonstrate this observation. In order to compute the RTFs whose roots are at level 3, we have to enumerate all the RTFs with roots at levels lower than 3. There are two valid levels lower than 3 – level 4 and level 5. As for 6, it is not valid because the prefix set of \( D_2 \) is empty. We do the computation at the lowest level first.

Fig. 5(a) illustrates the situation for level 5. The vertical rectangle w.r.t level 5 crosses three Dewey codes – \( D_{11} \), \( D_{12} \) in \( D_1 \), and \( D_{23} \) in \( D_2 \). The corresponding prefix sets are \{\"0.2.0.3.0\", \("0.2.0.0.0\"\} for \( D_1 \), and \{\"0.2.0.3.0\"\} for \( D_2 \). Following (2), the intersection is \{\"0.2.0.3.0\"\}. It is obvious that node \"0.2.0.3.0\" itself is a RTF following Definition 8.

\(^3\) The Dewey code has a useful property. It stores the Dewey codes of all its ancestors. The operation \texttt{prefix}(v, j)\ gets the Dewey code of \( v \)’s ancestor at level \( j \) (\( 1 \leq j \leq \text{length}(v) \)). The operation \texttt{length}(v)\ computes the length of \( v \), which corresponds to the path length from root node to \( v \). Based on \texttt{prefix}(v, j), it is easy to determine whether two nodes have some structural relationship or not. Relationships include parent-child, ancestor-descendant, and even sibling.
As for level 4, we have similar figure at level 5 – Fig. 5(b), in which the related Dewey codes of the prior RTFs have been deleted. We can see that the intersection of the prefix sets is empty. There is no RTFs at level 4.

Continuing to level 3, we have Fig. 5(c). We can see that the intersection of the prefix sets has the same Dewey code as that in (2) – “0.2.0”. However, the keyword nodes w.r.t it are \( D_{12} \) from \( D_1 \), and \( D_{21} \) from \( D_2 \) now, without including \( D_{11} \) of \( D_1 \), and \( D_{23} \) of \( D_2 \). According to Definition 8, “0.2.0” and those related nodes – \( D_{12} \), \( D_{21} \) and \( D_{22} \) – corresponds to a RTF.

So, there are two RTFs whose roots are lower than or equal to level 3. And they are also the final result, because there is no possible RTFs whose roots are located at higher level than 3. After deleting the related Dewey codes, we get Fig. 5(d) for level 2. Since there is no Dewey code in \( D_1 \) now, there is no RTF any more.

![Diagram](image)

(a) Level 5  
(b) Level 4  
(c) Level 3  
(d) Level 2  
(e) Label sequences

Fig. 5. A running example of LISA/RTF for the keyword query “Guo keyword” on the XML instance of Fig. 1

Our LISA/RTF is developed directly based on those observations except for the meaningfulness computation of the RTFs. It first collects the Dewey codes w.r.t the given keywords – \( D_i \) for keyword \( w_i \) (1 ≤ \( i \) ≤ \( k \)), and groups them according to their lengths – recording the Dewey codes with length as \( h \) of \( D_i \) in \( D^h_i \). Then LISA/RTF collects the RTFs layer by layer in a bottom-up way. In each level, the intersection of the corresponding prefix sets records the Dewey codes of the roots of the RTFs at that level, just like the computations shown in (3) and (2) of Example 6.
As for the meaningfulness computation of the RTFs, we need an additional encoding to record all the labels of the ancestors of a node so as to retrieve the label weights for the computation of Section 3. We propose a straightforward scheme for this. We assign a unique number for each distinct label in the XML data. Like Dewey encoding, we use a sequence to record the numbers following the path from the root to that node. Fig 5(e) illustrates this idea; in which the left rectangle records the labels together with their unique number and label weights, and the right part illustrates the number sequences of corresponding Dewey codes. For example, “author 6, 1/3” means the the label “author” is assigned the number “6”, and its label weight is “1/3”. “l_{11} [1 2 4 8 9]” in L_1 corresponds to the label path “Proceedings/Articles/article/references/ref”. It is easy to retrieve the label at the given level from this sequence. We mark this sequence as label sequence.

The whole pseudo code of LISA/RTF is illustrated in Algorithm 1. Except for the meaningfulness computation, it is easy to understand Algorithm 1 following the observation (3). Here we pay our attention to explain the meaningfulness computation based on the example in Fig. 5. The related codes are lines 9, 10 and 14 in Algorithm 1. We have following Example 7.

**Example 7 (LISA/RTF illustration for meaningfulness)** The illustration of meaningfulness computation here follows the example in Fig. 5.

At first, it is easy to collect the corresponding contributions of the keywords to the total weights of the nodes in D_1 and D_2. For example, the nodes in D_1 cover the keyword “Guo”, this means the keyword “Guo” contributes 1 to the total weight of every node in D_1. Similarly, the keyword “keyword” contributes 1 to the total weight of every node in D_2.

After the RTF collecting step at level 5 shown in Fig. 5(a), there is only one RTF, whose root corresponds to the Dewey code “0.2.0.3.0”. According to Definition 10, the total weight of the node w.r.t node “0.2.0.3.0” is just the sum of the keyword contributions, that is 1 + 1 = 2 (line 18). And this is also the meaningfulness value for the corresponding RTF.

Besides the meaningfulness computation of the retrieved RTF rooted at node “0.2.0.3.0”, there is another task in this step. We need transfer the keyword contribution of a node to its parent. For example, the keyword contribution of “Guo” for keyword node d_{12} = “0.2.0.0.0.0” is 1. When collecting the prefix of d_{12} at level 5, its parent is the node with Dewey code “0.2.0.0.0.0”. We have to transfer the contribution of d_{12} to this prefix (line 10). Following Definition 10, the contribution of node “0.2.0.0.0.0” is \( \frac{1}{2} \) now.

At level 4, there is no RTF. But we still need transfer the contribution of node “0.2.0.0.0” to its parent “0.2.0.0.0”. This time, however, we need incorporate the label weight. From Fig. 5(e), we know the label w.r.t node “0.2.0.0.0” is “author”, and its label weight is \( \frac{1}{3} \). So the transferred contribution is \( \frac{1}{2} \times \frac{1}{3} \times \frac{1}{2} = \frac{1}{12} \).

When continuing to level 3, the prefix set \( \text{pref}_1 \) only contains “0.2.0” with weight \( \frac{1}{2} \times 1 \times \frac{1}{2} = \frac{1}{4} \) from node “0.2.0.0”. While the prefix set \( \text{pref}_2 \) contains “0.2.0” and “0.2.1”, with weights \( \frac{1}{2} \times 1 \times 1 + \frac{1}{2} \times 1 \times 1 = 1 \), and \( \frac{1}{2} \times 1 \times 1 = 0.5 \) respectively. This time, there is another RTF’s root in inter – “0.2.0”, which corresponds to the Dewey
code of the second RTF’s. Its total weight is \( \frac{1}{24} + 1 = \frac{25}{24} \). So the meaningfulness of the corresponding RTF shown in Fig. 2(c) is \( \frac{25}{24} \approx 0.347 \).

The computation for the RTF in Fig. 4(a) is similar following LISA/RTF.

**Algorithm 1 LISA/RTF**

**Input:** Dewey sets \( D_i \) (each Dewey code contains the information of the label sequence and the total weight) for each keyword \( w_i \) (1 \( \leq \) i \( \leq \) k), and the label tuples like \((\lambda, n, L(\lambda))\) where \( \lambda \) is the distinct label, \( n \) is its unique number and \( L(\lambda) \) records its label weight.

The Dewey codes in \( D_i \) are grouped according to their lengths. \( D_h^i \) is the Dewey code set of \( D_i \) whose length is \( h \).

**Output:** RTFs with meaningfulness value

**Begin:**
1. \( \text{res} \leftarrow \{\} \);
2. \( \text{keynodes} \leftarrow \{\} \);
3. Collect the largest length \( L_i \) of the Dewey codes in \( D_i \);
4. \( m \leftarrow \min\{L_i|1 \leq i \leq k\} \);
5. \( \text{pref}_i \leftarrow \{\} \) where \( 1 \leq i \leq k \);
   /* prepare a prefix set for each keyword */
6. \( \text{For} \ (\text{loop} \leftarrow m \ \text{To} \ 2) \)
   /* loop = 1 means the root of the XML document */
7. \( \text{inter} \leftarrow \{\}; \) /* temporary intersection set */
8. \( \text{For} \ (i \leftarrow 1 \ \text{To} \ k) \)
9. If \( \text{pref}_i \) is not empty
10. Collect the prefixes at level \( \text{loop} \) of the Dewey codes in \( \text{pref}_i \), transfer the weight of a node to its parent following the discussions in Section 3, then renew \( \text{pref}_i \);
11. Collect the Dewey codes in \( D_h^i \) (where \( h \geq \text{loop} \)) into \( \text{pref}_i \), and delete the used \( D_h^i \);
12. If \( \text{pref}_i \) is empty
13. Exit For loop;
14. Do intersection between \( \text{pref}_i \) and \( \text{inter} \), then renew \( \text{inter} \) with the result;
15. If \( \text{inter} \) is empty
16. Continue to the next loop;
17. Else
18. Compute the total weight of the nodes in the renewed \( \text{inter} \);
19. End of For i;
20. Delete all Dewey codes from \( \text{pref}_i \) (1 \( \leq \) i \( \leq \) k) which are contained in \( \text{inter} \);
21. Add \( \text{inter} \) into \( \text{res} \);
22. End of For loop;
23. Return \( \text{res} \);
24. End

Now we analyze the complexity of our LISA/RTF. We have Theorem 1.
**Theorem 1** The complexity of LISA/RTF is \( O(\frac{m}{2} \sum_{i=1}^{k} |D_i| + \frac{m}{2} |D_1| + (2k - 1)|R|) \).

\( k \geq 2 \) is the number of input keyword query \( \mathcal{K} = \{w_1, \ldots, w_k\} \). \( D_1, \ldots, D_k \) are the Dewey code sets w.r.t the keywords in the keyword query, that is every node w.r.t the Dewey code in \( D_i \) covers the keyword \( w_i \). \( D_1 \) is the set with smallest cardinality. For each \( D_i \), \( L_i \) is the largest length among the Dewey codes in \( D_i \), and \( m \) is the minimum length of \( L_1, \ldots, L_k \), namely \( m = \min\{L_i|1 \leq i \leq k\} \). \( R \) is the RTF set.

**Proof 1** From Algorithm 1 we can learn that its computation is a dynamic process. The cost heavily depends on the properties of the data, such as the distribution of the roots of the RTFs. However, the number of RTFs is no more than the smallest cardinality \( |D_1| \) according to the Definition 8.

In order to conclude the complexity, we assume that the keyword nodes and the roots of RTFs are evenly distributed for all levels. And we only need pay attention to the computations from level \( m \) to 2, because there is only one node at level 1 – the root of the XML data. So we have \( \frac{|D_i|}{m - 1} \) (1 \( \leq i \leq k \)) keyword nodes for each keyword \( w_i \), and \( \frac{|R|}{m - 1} \) root nodes of the RTFs at every level.

**Table 1.** The computation sequence of LISA/RTF

<table>
<thead>
<tr>
<th>( m )</th>
<th>Prefix Collecting</th>
<th>Intersecting</th>
<th>Deleting</th>
</tr>
</thead>
<tbody>
<tr>
<td>( m )</td>
<td>( \sum_{i=1}^{k} \frac{</td>
<td>D_i</td>
<td>}{m-1} )</td>
</tr>
<tr>
<td>( m-1 )</td>
<td>( \sum_{i=1}^{k} \frac{</td>
<td>D_i</td>
<td>}{m-2} - \frac{</td>
</tr>
<tr>
<td>( m-2 )</td>
<td>( \sum_{i=1}^{k} \frac{</td>
<td>D_i</td>
<td>}{m-2} - 2 \frac{</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>2</td>
<td>( \sum_{i=1}^{k} (m - 1) \frac{</td>
<td>D_i</td>
<td>}{m-1} - (m - 2) \frac{</td>
</tr>
</tbody>
</table>

According to the idea of LISA/RTF, at each level there are mainly three basic tasks: collecting the prefix Dewey codes, computing the intersection of all retrieved prefix sets, and deleting used Dewey codes. Following the computation of LISA/RTF, we get a sequence of cost from \( m \) level to 2 level, shown in Table 1. For example, at level \( m \), LISA/RTF collects \( \frac{|D_i|}{m-1} \) prefixes from the grouped \( D_i \) (Prefix Collecting), so there are...
\[
\sum_{i=1}^{k} \left[ \frac{|D_i|}{m-1} \right] \text{ operations. When doing intersection, it begins from intersecting between the two smallest prefix sets after the prefix collecting (we still assume the } \frac{|D_1|}{m-1} \text{ is the smallest), we have } \frac{|R|}{m-1} \text{ elements in “inter”. And the later } k-1 \text{ intersections happen between the “inter” and other prefix sets. We have } \frac{|D_1|}{m-1} + (k-1) \frac{|R|}{m-1}. \text{ As for the deleting, LISA/RTF delete } \frac{|R|}{m-1} \text{ from each prefix sets, so the cost is } k \frac{|R|}{m-1}. \]

The computations at higher levels are similar, except that the older prefixes at lower level should be added into the renewed prefix sets. For example, the number of the operations for prefix collecting at level } m-1 \text{ is } \sum_{i=1}^{k} \left[ \frac{|D_i|}{m-1} \right] + \sum_{i=1}^{k} \left[ \frac{|D_i|}{m-1} - \frac{|R|}{m-1} \right] = \sum_{i=1}^{k} \left[ 2 \frac{|D_i|}{m-1} - \frac{|R|}{m-1} \right], \text{ in which } \sum_{i=1}^{k} \left[ \frac{|D_i|}{m-1} - \frac{|R|}{m-1} \right] \text{ is the number of prefixes from the level } m. \]

The total cost for LISA/RTF could be computed by adding all the operations in Table 1 together. We have the following equations. Equation 2 records the cost w.r.t “Prefix Collecting”, denoted as \(\text{Cost}_P\). Equation 3 records the cost w.r.t “Intersecting”, denoted as \(\text{Cost}_I\). Equation 4 records the cost w.r.t “Deleting”, denoted as \(\text{Cost}_D\).

\[
\text{Cost}_P = \sum_{i=1}^{k} \left[ 1 + 2 + \cdots + (m-1) \right] \frac{|D_i|}{m-1} - \sum_{i=1}^{k} \left[ 0 + 1 + \cdots + (m-2) \right] \frac{|R|}{m-1} = \sum_{i=1}^{k} \frac{m * (m-1)}{2} \frac{|D_i|}{m-1} - \sum_{i=1}^{k} \frac{(m-1) * (m-2)}{2} \frac{|R|}{m-1} = \sum_{i=1}^{k} \frac{m}{2} |D_i| - \frac{k(m-2)}{2} |R| \]  \tag{2}
\]

\[
\text{Cost}_I = \left[ 1 + 2 + \cdots + (m-1) \right] \frac{|D_1|}{m-1} - \left[ 0 + 1 + \cdots + (m-2) \right] \frac{|R|}{m-1} + \frac{(m-1) \times (k-1)}{1} \frac{|R|}{m-1} = \frac{m}{2} |D_1| - \frac{m-2}{2} |R| + (k-1) |R| \]  \tag{3}
\]

\[
\text{Cost}_D = k |R| \]  \tag{4}
\]

The whole cost is the sum of those three parts, listed as Equation (5).
\[\text{Cost} = \text{Cost}_P + \text{Cost}_I + \text{Cost}_D \]

\[= \frac{m}{2} \sum_{i=1}^{k} |D_i| + \frac{m}{2} |D_1| + (2k-1) |R| - (k+1) \frac{m-1}{2} |R| \]

We get \[\frac{m}{2} \sum_{i=1}^{k} |D_i| + \frac{m}{2} |D_1| + (2k-1) |R| - \frac{m-2}{2} |R|(k+1). \] Finally the complexity is \(O\left(\frac{m}{2} \sum_{i=1}^{k} |D_i| + \frac{m}{2} |D_1| + (2k-1) |R|\right)\) after omitting \(-\frac{m-2}{2} |R|(k+1).\)

\[\text{Table 2. The statistic information of the experimental datasets}\]

<table>
<thead>
<tr>
<th>XML</th>
<th>Size (MB)</th>
<th>Number of elements</th>
<th>Number of attributes</th>
<th>Number of keywords</th>
</tr>
</thead>
<tbody>
<tr>
<td>dblp20040213</td>
<td>197.6</td>
<td>4884836</td>
<td>1035747</td>
<td>11865622</td>
</tr>
<tr>
<td>xmark standard</td>
<td>111.1</td>
<td>1666315</td>
<td>381878</td>
<td>10628632</td>
</tr>
<tr>
<td>xmark data1</td>
<td>334.9</td>
<td>5010250</td>
<td>1148624</td>
<td>31991727</td>
</tr>
<tr>
<td>xmark data2</td>
<td>669.6</td>
<td>10023502</td>
<td>2299001</td>
<td>63885317</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>The frequency of interesting keywords</th>
</tr>
</thead>
<tbody>
<tr>
<td>dblp20040213</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>xmark standard</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>xmark data1</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>xmark data2</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

5 Experiments

In this section, we present the experimental result to illustrate the performance of our LISA/RTF. We compare it with CVLCA in [1] either on real (DBLP\(^6\)) or synthetic datasets (generated by XMARK\(^7\)).

5.1 Configurations

The statistical information about the datasets is shown in Table 1. The top subtable collects the numbers of related elements, attributes, and keywords in each dataset; while

---

\(^6\) http://www.cs.washington.edu/research/xmldatasets/
\(^7\) http://monetdb.cwi.nl/xml/
the bottom records the frequency of those interesting keywords. Based on the statistical information, we choose many keywords to compose our different keyword queries as follows.

Fig. 6. Performance of the algorithms on the datasets with different keyword queries. The complexity line in each subfigure records the computed complexity values (in $\log_{10}$) for LISA/RTF according to Theorem 1.

- **Keywords for DBLP**: keyword (90), similarity (1242), recognition (6447), algorithm (14181), data (25840), probabilistic (2284), xml (2121), dynamic (7281), sigmoid (3983),
tree (3549), query (3560), automata (3337), pattern (6513), retrieval (5111), efficient (8279), understanding (1450), searching (4618), yldb (2313), henry (1322), semantics (3694)

- **Keywords for XMARK series:** particle (12, 33, 69), dominator (56, 150, 285), threshold (123, 405, 804), chronicle (426, 1286, 2568), method (552, 1667, 3356), strings (615, 1847, 3620), unjust (1000, 3044, 6150), invention (1546, 4715, 9404), egypt (2064, 5255, 12466), leon (2519, 7647, 15210), previous (66216, 199365, 397672), description (11681, 35168, 70230), order (12705, 38141, 76271)

The integer(s) behind each keyword is the number(s) of corresponding keyword in the XML data, for example, “**keyword** (90)” means that there are 90 “keyword” words in DBLP; while “**particle** (12, 33 and 69)” means that there are 12, 33 and 69 “particle” words in xmark standard, data1 and data2 respectively.

The underlined letter in each keyword is used as the abbreviation of that keyword in the keyword queries. For instance, the “vdo” for XMARK series in Fig. 6(b)–6(d) means that the keyword query is “previous description order”. By random combinations of these keywords, we could construct different keyword queries, which cover the different frequency requirements.

We run those keyword queries composed by the above selected keywords in CVLCA and LISA/RTF. The efficiency of a method is illustrated by recording the elapsed time of a keyword query on a dataset. As for the effectiveness, we take CVLCA as the baseline, and compare its result on a keyword query with that of LISA/RTF. By recording the ratio of the retrieved results, we learn the effectiveness of LISA/RTF, that is it has same result as CVLCA when there is no nested label in DTD, and it can overcome the false negative problem of CVLCA when there is nested label in DTD.

---

**Fig. 7.** Ratio figures of the retrieved results by CVLCA and LISA/RTF
5.2 Platform

Our experiments are run on a Dell OPTIPLEX 745 using Ubuntu 7.0.4, with two Intel Core CPUs (2.4GHz), 2GB memory and 160GB hard disk. The algorithms are implemented in Java using Eclipse 3.2. We use Xerces 2.9.0 to parse the XML documents, and it is the stop-word filtering function in Lucence \(^8\) to filter the stop-words \(^9\). The shredded records are stored in PostgreSQL 8.2.4 with three simple tables – “label (label, number)”, “element (node’s label, Dewey, level, label number sequence)” and “value (node’s label, Dewey, attribute, keyword)”.

5.3 Experiment Results

- **Performance**: Fig. 6 shows the performance of CVLCA and LISA/RTF when running the keyword queries on different datasets. We only record the elapsed time after retrieving the Dewey codes of the corresponding keyword nodes so as to concentrate on the performance of the algorithms themselves. There is a complexity line in each subfigure. The value at each point w.r.t the keyword query is \(\log_{10} x\) in which \(x\) is the computed complexity value based on the formula in Theorem 1. We can see that the complexity lines reflect the major changes of LISA/RTF well.

From Fig. 6(a)∼6(d), we can see LISA/RTF has competent performance when running a keyword query on the datasets. We also can see from Fig. 6(b)∼6(d) that the top three keyword queries with largest elapsed time are “do”, “vd” and “vdo”, even there are other queries that also contain the keywords “v”, “d” or “o”. This can be explained by the complexity analysis, because the computation of LISA/RTF is also influenced by additional \(|D_1|\) and \(|R|\) in \(\left(\frac{m}{2}|D_1| + (2k - 1)|R|\right)\). Even though the other queries contain the keywords “v”, “d” or “o”, their smallest Dewey code set \(D_1\) is none of those w.r.t the three keywords. And \(|R|\) also contributes to the complexity even it is usually smaller than \(|D_1|\).

Besides, LISA/RTF is also flexible. We could construct an alert index to inform LISA/RTF to output the result increasingly during the computation. For example LISA/RTF could output the RTFs in the levels lower than 3 first; and if necessary, it could continue to compute the RTFs in higher levels. If we know the number of RTFs in advance, the complexity formula will be a good index since it reflects the elapsed time well. Regrettably, we can not. So we use the complexity formula without \(|R|\), namely \(\sum_{i=1}^{k} |D_i| + |D_1|\), as the index. Fig. 8, 9 and 10 illustrate this flexibility of LISA/RTF. There are three parameters which are used to determine the levels scanned by LISA/RTF – 0.3, 0.5 and 0.7. When taking a parameter \(\alpha\), the integer of the stop level is determined by \(m \times (1 - \alpha)\). For example, if \(\alpha = 0.3\) and \(m = 10\), this means that LISA/RTF collects RTFs from level 10 to level 10 \(\times (1 - 0.7) = 7\). Fig. 8(a), 9(a) and 10(a) demonstrate the performance of LISA/RTF on XMark standard, data1 and data2 respectively. We can see that by different parameters from 0.3 to 0.7, the performance of LISA/RTF changes from low to high. Fig. 8(b), 9(b) and 10(b) illustrate the percentages of the result under

\(^8\) [http://lucene.apache.org/](http://lucene.apache.org/)

\(^9\) [www.syger.com/jsc/docs/stopwords/english.htm](http://www.syger.com/jsc/docs/stopwords/english.htm)
different parameters with the total RTFs. It is clear that the higher the parameter is, the more LISA/RTF returns RTFs.

![Graph](image)

**Fig. 8.** (a): Performance of LISA/RTF with different parameters on XMark Standard; (b): Percentage of the result under different parameters with the total RTFs.

- **Effectiveness:** Fig. 7 illustrates the comparison of the results by CVLCA and LISA/RTF on different keyword queries. The left Y-axis records the number of the retrieved result, and the right Y-axis illustrates the ratio of the two retrieved results, defined as 

\[
\frac{|V \cap L|}{|L|}
\]

where \(V\) and \(L\) correspond to the results by CVLCA and LISA/RTF respectively. When collecting \(V\), we take the CVLCAs with same LCA node as one.

Fig. 7(a) illustrates the comparison of the results of CVLCA and LISA/RTF on dataset “dblp20040213”. It has some empty nodes, because the result there corresponds to the root of the XML document either in CVLCA or LISA/RTF, which we do not consider. From the ratio line, we can see that CVLCA and LISA/RTF have same result for those keyword queries on “dblp20040213”.

Fig. 7(b) shows the comparison of the results of CVLCA and LISA/RTF on dataset “xmark standard”. This time, the results are different. There is no keyword query with same result on dataset “xmark standard”. The largest ratio value in the Ratio line happens at keyword query “do”, which is 0.98. And there are many nodes whose ratio values are 0. By investigating the results of LISA/RTF on those nodes, we learned the reason, that is all the RTFs in those results do not satisfy the concept of VLCA in Definition 5. But those RTFs, especially those top RTFs according to their meaningfulness values, are still meaningful, since the DTD of “xmark standard” is “xmark-auction.dtd”, and “xmark-auction.dtd” itself has some elements whose nesting definitions contain duplicate name/label. For example, the element “parlist” is defined as
Fig. 9. (a): Performance of LISA/RTF with different parameters on XMark data1; (b): Percentage of the result under different parameters with the total RTFs.

Fig. 10. (a): Performance of LISA/RTF with different parameters on XMark data2; (b): Percentage of the result under different parameters with the total RTFs.

“<!ELEMENT parlist (listitem)*>; <!ELEMENT listitem (text|parlist)*>”. On the contrary, CVLCA delete all of them based on Definition 5. When comparing the results on other keyword queries, LISA/RTF returns more fragments than CVLCA, and the difference usually corresponds to the fragments filtered out by VLCA concept from the
result of LISA/RTF. All these reflect that VLCA concept is not universal, and it commits the false negative problem when the DTD w.r.t the XML data contains elements with duplicate label in their nesting definitions.

In summary, in terms of efficiency, the experimental result shows that LISA/RTF has competent performance like CVLCA. Further, it is also flexible to output the RTFs increasingly based on its property of returning RTFs layer by layer. As for the effectiveness, LISA/RTF overcomes the false negative problem of CVLCA when the DTD of the XML data contains elements with duplicate label in their nesting definitions. And the XKSMetaRank further provides a filtering mechanism, which could compute the meaningfulness of the RTFs during the retrieving, and output the top RTFs with higher meaningfulness.

6 Related work

In this section, we briefly review the related work. We cut it into two parts: the related work of adapting keyword search on XML data (Section 6.1), and the related work of adapting keyword search on other type of data (Section 6.2), like the graph data.

6.1 XML keyword search

In terms of keyword search over XML data, there are mainly two parts of related work. One is the research on what the result should be retrieved for the given keyword query, described in Section 6.1. The other is about the ranking/filtering of the retrieved result so as to guarantee the meaningfulness, described in Section 6.1.

Result types of XKS There are mainly two kinds of result types for XKS. The first class only returns the nodes as the result, such as LCA variants [14, 9–11, 18]. SLCA [10] avoided the false positives of LCA (formalized in [1]), but it did so at the expense of the false negative problem. Multiway-SLCA [11] offered a search paradigm in support of keyword search beyond the traditional AND semantics, including both AND and OR boolean operators. XSeek [18, 19] generated nodes which could be explicitly inferred from keywords or dynamically constructed according to entities in the data that are relevant to the search. [20] investigated the performance of returning SLCA nodes based on materialized views. [15] further proposed an efficient algorithm for ELCA. However, since only returning the filtered LCA nodes, it had the incompleteness problem, and it did not discuss the meaningfulness either. Finally, for the proposals here, even though the whole subtree, rooted at the retrieved node based on the related keyword nodes, could be achieved later, intuitively, it is more convenient for the users to show them the sketch content implied in the raw subtree in the first place.

In view of this, the second class tried to return the sketched fragments as the result, including Interconnected pairs [8], GDMCT [16], MIU [17] and CVLCA-based answer [1]. [8] focused on the semantics and the ranking of the results, and it used an all-pairs interconnection index to check the connectivity between the nodes. [16] proposed GDMCT, which generated the grouped subtrees to answer keyword queries. It needed to traverse the whole graph to identify answers. [17] first partitioned the XML data into
meaningful units based on the analysis of the DTD and the XML data. Then it returned the units as the result, which were connected to the interesting LCA nodes. In order to overcome the false positive and negative problems, [1] proposed VLCA concept to guarantee the meaningfulness of the returned LCA nodes. Its final answer was based on the VLCA, and returned the VLCA node together with some related keyword nodes. Regrettably, it had two limits, which are discussed in Section 1. They are false negative problem of VLCA concept, and the incompleteness problem of CVLCA.

Guaranteeing the meaningfulness There are many methods proposed to guarantee the meaningfulness of the retrieved result. The first step is to constrain the LCA nodes. [8] proposed the interconnection relationship to filter the subtrees composed by the LCA nodes and the keyword nodes. Given two nodes \( n \) and \( n' \) in a tree \( T \), \( LCA(n, n') \) is their LCA node, and we can get a node set composed by the nodes in the paths of \( LCA(n, n') \rightarrow n \) and \( LCA(n, n') \rightarrow n' \), denoted as \( \nabla_{n,n'}T \). The two nodes are interconnected iff \( \nabla_{n,n'}T - \{n, n'\} \) does not contain two distinct nodes with same label. [8] then concluded that only those interconnected fragments were meaningful. From the false negative problem of VLCA in Section 1, we can see that the interconnection relationship concept had similar problem. The other step of filtering LCA nodes is the SLCA concept, summarized in [10]. [14] also proposed a ranking method to sort the retrieved nodes, which inherited the idea of PageRank, and considered only the organization information implied in XML data.

After [14, 10], the SLCA concept became the core of the related research. [17] first proposed a partitioning mechanism for the schema graph into meaningful units, and subsequently partitioned the XML data based on the meaningful units into MIUs. Finally [17] returned the MIUs composed along with computing of the XLCA (eXclusive LCA) nodes as the result, where XLCA concept is similar with CLCA [1] and ELCA [15]. [18, 19] took a different idea. It analyzed both XML data structure and the keyword patterns to understand the roles of the nodes in XML data, and the requirements for the result implied in the given keyword patterns. It then composed the final result based on those analysis after retrieving the keyword nodes and the LCA nodes. The idea of [18, 19] could be easily combined in other XKS methods including ours. In view of the false positive and negative problems of the SLCA, [1] proposed VLCA concept to confine the LCA nodes. The core of the VLCA was the homogenous concept of two nodes, which was similar to the interconnection relationship in [8]. It had the false negative problem as discussed in Section 1.

6.2 Keyword search on other kind of data

Besides the above research concentrating mainly on XML data (semi-structured data), the research of using keyword search on other data, such as the structured data (relational) [21–28], and the graph data, [29–37], is also becoming attractive. In fact, the related work on the structured data could be included by the research on the graph data, because the research on the structured data also used to take the data as a connected graph. So here we review them together.
DBXplorer [21], DISCOVER-I [23], DISCOVER-II [24], BANKS-I [22] and BANKS-II [29] are systems built on top of relational databases. DISCOVER and DBXplorer generate trees of tuples connected through primary-foreign key relationships that contain all of the input keywords. BANKS identifies connected trees in a labeled graph by using an approximation of the Steiner tree problem. DISCOVER-II considers the problem of keyword proximity search in terms of disjunctive semantics, as opposed to DISCOVER-I which only considers conjunctive semantics. Kacholia et al. [29] presented the bidirectional strategy (BANKS-II) to improve the efficiency of keyword search over graph data. However, their method still works by identifying Steiner trees from the whole graph, which is inefficient as it is rather difficult to identify structural relationships through inverted indices. Liu et al. [25] proposed a novel ranking strategy to solve the effectiveness problem for relational databases. It employs phrase-based and concept-based models to improve search effectiveness by introducing IR techniques. Guo et al. [31] proposed data topology search to retrieve meaningful structures from richer structural data, such as complex biological databases. He et al. [30] proposed a partition-based method to improve search efficiency with a novel BLINKS index. Luo et al. [26] proposed a new ranking method that adapts state-of-the-art IR ranking function and principles into the ranking trees of joined database tuples.

As for the research on graph data, there is a long history on methods for querying graphs. Most previous works focused on exact graph or subgraph matching, i.e. graph or subgraph isomorphism. Subgraph isomorphism was proved to be NP-complete. Ullmann [38] proposed a subgraph matching algorithm based on a state space search method with backtracking. However, this algorithm is prohibitively expensive for querying against database with a large number of graphs. To reduce the search space, Graph-Grep [39], GIndex [40] and TreePi [41] index substructures of the database (paths, frequent subgraphs and trees respectively) to filter out graphs that do not match the query. There are also several index-based methods for approximate subgraph matching. Most of these techniques only apply to small graphs and allow limited approximation. Grafil [42] and PIS (Partition-based Graph Index and Search) [43] are both built on top of the exact subgraph matching method GIndex. However, neither method allows node insertion or deletion in their match models. CDIndex [44] only applies to graphs with limited sizes, as it exhaustively enumerates and indexes all the subgraphs in the database. GString [45] utilizes sequence matching to answer graph queries, but it only applies to applications in which the graphs contain a small number of basic substructures. C-Tree [46], which employs an R-tree like index structure, is a more general tool than the above methods. In view of this problem, [34] proposes NH-Index (Neighborhood Index) to capture the information implied in graph data which is based on the neighborhood information for each node. Meanwhile, [37] proposes GCoding (graph coding), a spectral encoding method, to improve the performance of processing graph pattern on large graph database. While, [33] proposes a join-based (R-join and R_semi-join) method to cope with the graph pattern matching problem.

[36, 32] are two latest papers on adapting keyword search over graph data. In [36], Golenberg et al proposes an engine for the keyword proximity search that is based on an incremental algorithm for enumerating subtrees in a 2-approximate order [47].
by increasing height (i.e., the approximation ratio is 2). By modeling all the data as a graph, [32] proposes a framework for keyword search on that graph.

7 Conclusion

This paper revisits XML keyword search, and proposes the RTF concept to represent the result. A RTF contains all the related keyword nodes. We further propose XKSMetaRank to filter the RTFs so as to guarantee the meaningfulness of the top RTFs. It overcomes the false negative problem in CVLCA [1]. Finally, we propose LISA/RTF algorithm to combine RTF concept and XKSMetaRank mechanism together. During the retrieving of a RTF, LISA/RTF could give out its meaningfulness once the RTF is determined. We implement our LISA/RTF and compare it with CVLCA. The experimental result shows that LISA/RTF has competent performance like CVLCA. Further, it is also flexible to output the RTFs increasingly based on its property of returning RTFs layer by layer. As for the effectiveness, LISA/RTF overcomes the false negative problem in CVLCA, and preserves all the related keyword nodes implied in the subtree in the XML data whose root is also the root of the RTF. To the best of our knowledge, the RTF is the first work aiming at this.

Even though the XKSMetaRank here is helpful to judge the meaningfulness of the RTFs, we still need other similarity measures when doing keyword search on heterogeneous XML data with different DTDs. We need take into account not only the semantic difference determined by the labels but also the structural difference.

References


19. ——, “Reasoning and identifying relevant matches for xml keyword search,” in *34th International Conference on Very Large Data Bases (VLDB)*, 2008.


43. X. Yan, F. Zhu, J. Han, and P. S. Yu, “Searching substructures with superimposed distance,” in *Proceedings of the 22nd International Conference on Data Engineering (ICDE)*, 2006.

graph databases,” in 23rd International Conference on Data Engineering (ICDE), 2007, pp.
566–575.
of the 22nd International Conference on Data Engineering (ICDE), 2006.
47. R. Fagin, A. Lotem, and M. Naor, “Optimal aggregation algorithms for middleware,” in
Proceedings of the Twentieth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of