# A Framework for Structural Refinement of XML Keyword Search

Qingsong Guo, Yongluan Zhou
*University of Southern Denmark*
{qguo, zhou}@imada.sdu.dk

## ABSTRACT

In this paper, we develop a framework for querying XML data that can take advantage of the strengths of both structured queries and keyword queries. Basically, we refine a keyword query intelligently into a number of tree patterns, from which the user can interactively select the queries that match his query intention. This is a challenging problem since most of possible structured queries constructed from user-issued keywords are incompatible with the structural constraints in the data and hence useless. Alternatively, we extract a *relation graph* from the given XML document $D$, which captures the corresponding structural constraints of user query. We then transform the query refinement into a classical problem in graph theory and hence take advantage of the existing research results on this problem.

## Categories and Subject Descriptors

H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval

## General Terms

Theory

## Keywords

XML, Structural Refinement, Keyword Search, Tree Pattern

## 1. INTRODUCTION

XML has become a standard for representation and exchange of data in a variety of applications. However, the complexity and heterogeneity make the retrieving information from XML documents to be a challenging task. Thus, it attracts a lot of attentions from both the DB and IR research communities [2].

The DB community mainly focused on designing structured query languages, such as XPath/XQuery, and developing the corresponding query processing techniques. Taking an XML bibliographic record as an example, as shown in Fig. 1, the following XPath query interests in the title of books published by Pearson Education, $//book[publisher = `Pearson\ Education']/title$. In general, a structured query can be represented as a *tree pattern* [3], as illustrated in Fig. 2, which is a *rooted tree* with edges denoting the parent-child or ancestor-descendant relationship among nodes. Hereafter, we will use tree patterns to represent structured queries and use these two terms interchangeably. With structured query languages, users can precisely specify their queries. However, it requires users have good knowledge of the documents' structure and hence gives extra burdens.

On the contrary, the IR community interested in keyword search. One can simply express the aforementioned query with a set of keywords: "*title of book published by Pearson Education*". Obviously, it is less demanding on the users' knowledge and skills. The major downside of keyword query is that it fails to capture the structures of XML documents and hence cannot precisely express one's interests. It typically return a mass of imprecise XML fragments that match the issued keywords, from which users have to sort out useful information that they interested.

To address the dilemma of choosing between keyword queries and structured queries, we explore a new refinement framework that integrates both approaches and get the best out of both worlds. Our framework provides a keyword-based query and a mechanics underneath that automatically refines the keywords query into a number of tree patterns. These tree patterns are ranked according to their relevance with the keyword query and could be executed on demand to retrieve the results. In this way, users can precisely retrieve data by selecting the structured queries to be executed.

To realize the above idea, many obstacles need to be overcome. Firstly, an enormous amount of tree patterns could be constructed from a set of keywords but only a minor part of them is *compatible* with the structural constraints of the data. For instance, given a set of keywords $Q = (k_1, k_2, \ldots, k_n)$, we can construct $n^{n-1}$ tree patterns according to *Cayley's formula*[1]. While only the *compatible patterns* can return results and are useful, pruning the massive number of incompatible patterns needs the structural information of the data and hence may involve frequent and expensive data access. Moreover, the tree pattern with the top score may not match with the user's query intention. It is preferable to present an interactive refining framework that can return tree patterns on user's demand. This requires the algorithm to be able to progressively produce structured queries in the order of their ranking scores.

Secondly, it is quite often that very few tree patterns can be generated if we only use the keywords in the query to construct the tree patterns. It is highly possible for users to issue an ill-formed keyword queries because of the lack of knowledge about the data. Instead of repeatedly asking the user to try a different query, it is more friendly to expand the initial query so as to generate more

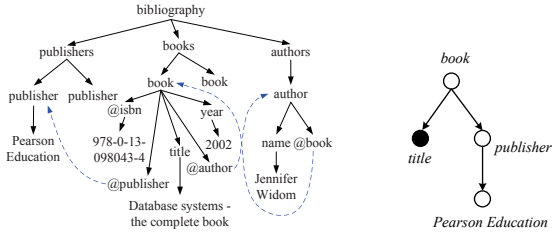


Figure 1: Part of Bibliographic Records




Figure 2: Tree Pattern

compatible tree patterns to enrich user's choices. We call this process as *structural query expansion*.

Our framework addresses the above challenges and achieve those desirable properties or features. In general, we extract the structural constraints of the user-specified keywords from the data, which forms a relation graph G. G is a weighted directed graph, and its spanning rooted trees are taken as the desired tree patterns. Hence, we transform the query refinement into a classical combinatorial problem: computing the top-k minimum spanning rooted trees of a directed graph.

**Related Work.** Many efforts were made to improve the preciseness of XML keyword query, such as the structural semantics like *lowest common ancestor* (LCA) and *smallest LCAs* (SLCA) [11]. Some approaches, such as structural relevance feedback [5] and natural language processing techniques [10], put the refinement a step further to produce structured queries directly. But these methods are either inefficient or too complicated in practice. Petkova et al. [7] proposed a more practical approach that can automatically create structured queries for a given keyword query. It first converts terms into target sets, such as "//$a$" and "//$a$[$\sim$ '$x$']", and then combines them into singletons. There are enormous combinations but only a small portion are *compatible patterns* that consist with the structural constraints of source data. Three operators are introduced to prune the useless combinations, but this approach would still suffer from serious performance problem.

# 2. XML QUERY REFINEMENT

In our context, we aim at refining the XML keyword query into tree patterns. A tree pattern is a *rooted tree* [3], in which each node can have various labels such as element tags and predicates on attribute values and each edge is either a *parent-child*("/") or *ancestor-descendant*("//") relation.

A huge number of tree patterns can be constructed from a given set of keywords. However, most of them do not consist with the structure of a specific document $\mathcal{D}$. To avoid flooding the users with a number of meaningless tree patterns, we should only return those that definitely have query results.

DEFINITION 1 (COMPATIBLE QUERY). *Given an XML document $\mathcal{D}$ and a tree pattern $t$, $t$ is compatible with $\mathcal{D}$ if any edge $(a, b)$ of $t$ satisfies the following conditions:*

- *$a$ and $b$ have matching nodes $n_a$ and $n_b$ in $\mathcal{D}$; and*
- *the relation between $n_a$ and $n_b$ is either $n_a/n_b$ or $n_a//n_b$.*

Accordingly, our target is refining a given keyword query $Q$ into a set of tree patterns, $\mathcal{T} = \{t_1, t_2, \ldots, t_k\}$, that are compatible with structural constraints of data. The overall procedure of our refining framework is as follows.

(1) **Keyword query preprocessing:** Traditional IR techniques, such as word splitting, stop words deletion, word stemming, acronym expansion, word merging, phrase segmentation, etc., are applied in advance to transform the initial query $Q$ into a set of refined terms $K$. For instance, the query in our previous example will be refined into {*title, book, publish, Pearson Education*}.

(2) **Relation graph extraction:** In this step, we extract the structural constraints among the terms of $K$ which form the relation graph $G$. For a structural term in $K$, the matching node is put into $G$ directly. With regard to a content term, its parent, which is a structural node, is taken into $G$ instead. Each edge of $G$ associates with a weight, which is calculated according to the weighting scheme.

(3) **Relation graph reformation:** It is possible for a relation graph $G$ that has no compatible tree patterns. Therefore, this step is to reform $G$ into a new graph that has at least one compatible tree pattern.

(4) **Query generation:** *Top-k* compatible tree patterns are generated by incrementally computing the maximum spanning rooted trees of the relation graph. Besides, other processes, such as transitive error elimination, are carried out simultaneously.

Note that the *structural query expansion* can be accomplished by repeating Step (3) and (4) on user's demand.

# 3. RELATION GRAPH

Assuming we already have a set of refined terms $K$. We next present the details of relation graph, such as extraction, weighting, and the reformation of relation graph.

## 3.1 Data Model and Structural Summary

An XML document can be represented as a directed graph, as illustrated in Fig. 1. Three types of nodes are of our concern, such as element nodes, attribute nodes, and text nodes. These nodes can be classified into two categories, where text nodes are referred to as *content nodes*, while element nodes and attribute nodes are referred to as *structural nodes*. Accordingly, terms in $K$ have matching nodes from the document can be classified into two categories: *structural terms* referring to those that match structural nodes and *content terms* referring to those that match content nodes.

All structural nodes and their relations determines the structure of the source data. The structure can be captured by a structural summary, which is a directed graph $S = (V_S, E_S)$, where $V_S$ and $E_S$ consist of structural nodes and their relations respectively. A relation graph is derived from $S$ and it is defined as follow.

DEFINITION 2 (RELATION GRAPH). *Given an XML document $\mathcal{D}$, with structural summary $S = (V_S, E_S)$, and a set of refined terms $K = \{k_1, k_2, \ldots\}$, the relation graph $G = (V, E)$ of $K$ corresponding to $\mathcal{D}$ is a directed graph such that: (1) $V \subseteq V_S$; (2) for each $(u, v) \in E$, either $(u, v) \in E_S$ or there is a path $p_{uv}$ in $S$.*

**Extracting Relation Graph.** An edge of $G$ is either "/" or "//" relation. These relations can be extracted via auxiliary indexes or by directly scanning the source data. A synonym thesaurus is useful in this step, as it can be used to expand the query while a term of $K$ has no matching nodes. Due to the flexibility of XML definition, a tag may appear in two distinct elements with "//" relation, and the same pair of tags may also appear in different places of the document. The consequence is that loops and parallel edges would involve in $G$. Thus, $G$ should be modified to a simple directed graph by removing the loops and combing the parallel edges. If two parallel edges have different relation labels, the combined edge will be marked as "//".

## 3.2 Weighting the Relation Graph

In order to rank the output tree patterns, we introduce a novel weighting mechanism that integrates the term frequencies (TF) and

the structural importances of XML elements. The structural importances of nodes depend on theirs relative locations to the root of the XML data. Apparently, nodes closer to the root are more important in the sense that a query matching it can return more information than those matching nodes closer to the leaves. As we described earlier, a node in $G$ stands for the elements with the same tag in data. Each edge in $G$ represents a set of paths in the XML data, and all parallel edges are combined into a single edge, thus its weight can be measured by the cumulative probability of all involved paths.

To capture the importances of different nodes, we introduce a probabilistic XML model [6], in which elements within $\mathcal{D}$ are not deterministic. Instead, each element $y$ associates with a probability $Prob(y)$, which is a conditional probability with regard to its parent element. It can be calculated as follows.

**(1)** The root $r$ of $\mathcal{D}$ is given a probability of 1, i.e. $Prob(r) = 1$.

**(2) Element $y$ has a unique path from the root**. Suppose $p_{ry}$ is the only path from $r$ to node $y$ and $x$ is the preceding node of $y$. Then, $Prob(y)$ can be calculated by applying Bayes' formula

$$Prob(y|x) = \frac{Prob(x|y) * Prob(y)}{Prob(x)}. \quad (1)$$

$Prob(y|x) = \frac{1}{k}$, where $k$ is the number of children of $x$. As $y$ has a unique parent node $x$, $x$ must exist when $y$ exists, i.e. $Prob(x|y) = 1$. So we have $Prob(y) = Prob(y|x) * Prob(x)$. $Prob(y)$ can be calculated by applying the formula recursively.

**(3) Element $y$ has more than one paths**. Suppose there are $h$ paths from $r$ to $y$, $\{p_i|1 \leq i \leq p_h, h \geq 2\}$. Then the probability of $y$ is

$$Prob(y) = \sum_{i=1}^{h} Prob(y_i), \quad (2)$$

where $Prob(y_i)$ is the probability of $y$ appears on the path $p_i$ and can be computed according (1).

By applying formula (1) and (2), we can calculate the probability for each element in $\mathcal{D}$ and then the weight of each edge in the relation graph will be calculated as the sum of the probabilities for all its corresponding arcs (or paths) in the document. In a graph modeled XML data, there may exist more than one arcs (or paths) between a pair of elements. For instance, assuming $x/y$ and $c_1//c_h$ are two edges in $G$, where there are $m$ arcs $\{e_i|e_i = x_i \rightarrow y_i, 1 \leq i \leq m\}$ and $n$ paths $\{p_i|p_i = c_0 \rightarrow \cdots \rightarrow c_h, 1 \leq i \leq n\}$ in the document corresponding to them respectively. Then, the weight of $x/y$ is $\omega(x/y) = \sum_{i=1}^{m} Prob(e_i) = \sum_{i=1}^{m} Prob(y_i|x_i)$, where $Prob(y_i|x_i)$ is the probability of arc $x_i \rightarrow y_i$ as explained in formula 1. Similarly, $\omega(c_h//c_1) = \sum_{i=1}^{n} Prob(p_i)$, where $Prob(p_i)$ is the product of the probabilities of all the arcs on path $p_i$. Suppose $p_i = \langle c_1, c_2, \ldots, c_{h-1}, c_h \rangle$. Then, $Prob(p_i) = Prob(c_h|c_{h-1}, \ldots, c_1) = \prod_{i=1}^{h-1} Prob(c_{i+1}|c_i)$.

# 4. SPANNING ROOTED TREE

A directed graph $G = (V, E)$ consists of a set of vertices $V = \{v_1, v_2, \ldots, v_n\}$ and a set of directed edges $E$. Each edge in $E$ is an ordered vertices pair $(v_i, v_j)$, $v_i, v_j \in V$. A *path* $p_{uv}$, denoted as $\langle u, \ldots, v \rangle$, is a sequence of edges from $u$ to $v$ in $G$. A path is *simple* if all vertices are distinct and it is a *cycle* if $u$=$v$. $G$ is *strongly connected* if there is a path between any pair of vertices in $V$. $G$ is *weakly connected* while its underlying graph $U(G)$ is a connected graph.

## 4.1 Sufficient and Necessary Condition for the Existence of SRT

A **spanning rooted tree** (SRT) is a spanning subgraph of $G(V, E)$ which has a root such that there is a unique path directed from the root to each node. The exact number of SRTs of $G$ can be calculated with *Laplacian Matrix* and *Matrix-tree theorem* [9].

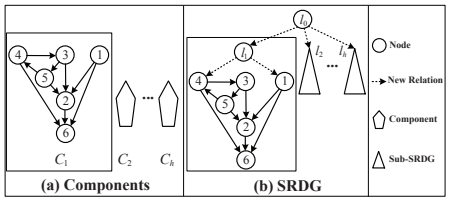

Figure 3: Relation Graph Reformation

To guarantee that each query has at least one matched tree pattern, we give a sufficient condition for the existence of SRT. We will perform a reformation on the relation graph when it does not satisfied the sufficient condition. Furthermore, we also proposed a necessary condition, which indicates a reformation with the minimum cost. Both conditions are promised by *quasi-strong connectivity*

DEFINITION 3 (QUASI-STRONG CONNECTIVITY). *A directed graph $G$ is* quasi-strongly connected *if, for any pair of distinct vertices $u$ and $v$ in $G$, there are two paths direct to $u$ and $v$, respectively, which start at the same vertex $w$, where $w$ could be $u$ or $v$.*

Suppose that $G$ has a SRT $T$, then there should be a path within $G$ that is directed from the root of $T$ to any other vertices. It shows that quasi-strong connectivity is a necessary condition for the existence of SRT in $G$. Quasi strong-connectivity is also a sufficient condition for the existence of SRT, which is gave by following theorem. The detailed proof is omitted here.

THEOREM 1. *A directed graph $G$ has spanning rooted tree if and only if it is quasi-strongly connected.*

## 4.2 Reforming the Relation Graph

The relation graph reformation is conducted according to the *quasi-strong connectivity*. We focus on a specific type of quasi-strongly connected graph, called as single root directed graph (SRDG). A directed graph is called SRDG if it has at most one vertex $v$ that $d^+(v) = 0$ and its underlying graph is connected.

Apparently, a SRDG can be transformed into a directed acyclic graph $G'$ with a single root by collapsing each cycle into a pseudo vertex. There is a path directs away from the root to any other vertices in $G'$, thus it satisfies the *quasi-strong connectivity* property.

The reformation of relation graph $G$ can be completed in following two steps: (1) transforming each weakly connected component to a SRDG; (2) connecting all the sub-SRDGs to an integral SRDG by adding the lowest common ancestor of the roots of these sub-SRDG components and corresponding edges.

Let $\mathcal{C} = \{C_1, \ldots, C_h\}$ be the connected components of $U(G)$, and $v_{ij}$, $1 \leq i \leq h$ and $1 \leq j \leq k$, be a vertex of $c_i$ that has no incident edge. Let $l_i = lca(v_{i1}, \ldots, v_{ik})$ be the lowest common ancestor of all these vertices. The reforming on a single component proceeds simply by adding $l_i$ to $C_i$. Then, $l_i$ is the root of the newly formed SRDG component.

Suppose $l_0$ is the lowest common ancestor of those roots, i.e. $l_0 = lca(l_1, l_2, \ldots, l_k)$. We then repeat the above process and combine all components to a single SRDG by adding $l_0$ as the new root. The reforming process is illustrated in Fig. 3. Obviously, there is at least one path to any other node from $l_0$, and hence the new created graph is definitely a SRDG. Furthermore, the XML data has a unique root, which is an ancestor of all other nodes, thus all these nodes, $l_0$ to $l_k$, can be found. It guarantees that any relation graph can be transformed to a SRDG.

# 5. INCREMENTAL QUERY GENERATION

## 5.1 Tree Patterns Generation

**Best queries generation:** Tree patterns generation is based on the algorithm for computing the *k-best* SRTs. Tarjan described an efficient implementation for generating maximum SRT (MSRT) in time complexity of $\mathcal{O}(m \cdot \log n)$ [8], where $n$ and $m$ are the number of vertices and edges of the graph respectively. The essence of MSRT is to build a maximal *critical graph* $M$. $M$ initially contains all vertices of $G$ but without any edges. The task is accomplished by alternately expanding and collapsing $M$. The expanding routine continually chooses and adds the incident edge with maximum weight for an unvisited vertex into $M$. If a cycle $S_i$ emerges, then it will be collapsed to a pseudo vertex $u_{n+i}$. Then, edges directs to a vertex in $S_i$ from outside are redirected to $u_{n+i}$ and with their weights modified accordingly. After finite repetitions of the above procedure, a *critical graph* $M$ is constructed. Choosing an incident edge with the minimum weights to break each circle, and hence the MSRT obtained.

Suppose that the MSRT subjects to two constraints $Y$ and $Z$ ($Y \cap Z = \emptyset$), where $Y$ and $Z$ are the set of edges that should be included in and should be excluded from the solutions respectively. Furthermore, let $BEST(G, Y, Z)$ denote the MSRT. Assume $H$ is a subset of $E - Y$ and each of its edges shares endpoint with an edge of $Y$, i.e. $H = \{e_i | \tau(e_i) = \tau(y_i), e_i \in E - Y \wedge y_i \in Y\}$. The constraints can be easily fulfilled by removing all edges of $Z$ and $H$ from $G$ while running the MSRT algorithm. The remaining graph is called *restricted graph*, which is denoted as $G_{Y,Z}$.

**Top-k queries generation:** The algorithm to generate *k-best* MSRT is based on a routine of finding an SRT whose score is next to that of MSRT and such an SRT is denoted as NSRT. Obviously, MSRT has at least one edge that does not exist in NSRT. Therefore, if MSRT has $n-1$ edges, we can derive the NSRT by $n-1$ iterations. Within each iteration, we pick one edge $e$ from the MSRT and run the MSRT algorithm on the graph without considering $e$. This will produce $n-1$ SRTs and the one with the best score among them is the NSRT. A previous study [4] shows that this procedure can be optimized into a single iteration.

Suppose $A$ is the MSRT of $G_{Y,Z}$ and $A_i$ is the MSRT of $G_{Y,Z \cup f_i}$, where $f_i$ is the *i-th* edge of $A$. Suppose that $A_h$ is the solution with the maximum total weights among $\{A_1, \ldots, A_{n-1}\}$, and we say that $f_h$ is the edge that leads to NSRT. Clearly, $A_h$ is the spanning rooted tree next to $A$. A straightforward way to obtain $A_h$ is by exhaustedly enumerating each $A_i$ ( $1 \leq i \leq n-1$), which requires $\mathcal{O}(n)$ MSRT computations. In [4], Camerini et al. devised an efficient way to get NSRT with complexity of $\mathcal{O}(m \cdot \log n)$. Essentially, it tries to find $f_h$ within a single MSRT computation. The algorithm to find $f_h$ is denoted as $NEXT(G, Y, Z, A)$.

Let $A^j$ ($1 \leq j \leq k$) denote the *j-th MSRT* of $G_{Y,Z}$, where $A^1 = A$. Suppose that we have already obtained $A^1, A^2, \ldots, A^{j-1}, 1 < j < k$. The remaining MSRTs are partitioned into $j-1$ disjoint sets, $P_i^{j-1} = \{A^h : j \leq h \leq k; Y_i^{j-1} \subseteq A^h \subseteq E - Z_i^{j-1}\}$,$1 \leq i < j$. Let $b$ be the edge that leads to $A^j$. Then, $A^j$ is either the MSRT of $G_{Y_i^{j-1}, Z_i^{j-1} \cup f}$ or $G_{Y_i^{j-1} \cup f, Z_i^{j-1}}$. The *k-best* MSRTs can be generated by alternatively using $BEST(G, Y, Z)$ and $NEXT(G, Y, Z, A)$, which is based on the fact that the next solution either will include $f$ or will not include $f$. All compatible SRTs can be obtained by setting $k$ to $\infty$.

## 5.2 Eliminating Incompatible SRTs

Although each edge in $G$ is compatible with the structural constraints of data, a SRT generated from $G$ may still be incompatible. For example, as shown in Fig. 4, $(a, b)$ and $(b, c)$ are two edges of

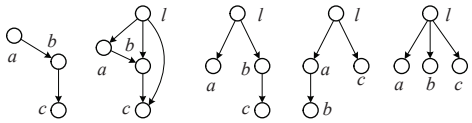

Figure 4: Transitive Errors Elimination

$G$, but $b$ in two edges refer to different nodes in XML document. A relation $a//c$ can be derived from $G$, but it is incompatible with the data. This error, called *transitive error*, is caused by the ambiguity of $b$ and the *transitivity property* of relations.

Any tree pattern containing both $(a, b)$ and $(b, c)$ is incompatible with the data. Transitive error detection can be incorporated into the SRTs generation. Before generating the queries, we find out all adjacent edge pairs that may incur transitive errors and each of them can be represented as a triplet $\tau = \{\tau_i | \tau_i = \langle a_i, b_i, c_i \rangle\}$. When an SRT is generated, we just need to check whether it contains a pair of edges involved in a transitive error triplet $\tau_i$. To be noted that there could be no SRTs left after the error elimination, for instance, the relation graph shown in Fig. 4(a) does not contain any SRT without transitive error. Therefore, we will add some additional relations into $G$ to guarantee there is at least a SRTs.

**Example.** *Fig. 4 illustrates an example on transitive error elimination, which just contains one transitive error component. Edges between the lowest common ancestor $l$ and the other nodes, i.e. $(l, a)$, $(l, b)$, and $(l, c)$, are added into the original relation graph (Fig. 4(a)), which is transformed to a new graph (Fig. 4(b)). There are three spanning trees that have no transitive error as shown in Fig. 4(c), 4(d), 4(e) respectively. Therefore, no SRT of the graph shown in Fig. 4(d) contains $(a, b)$ and $(b, c)$ simultaneously. By performing* top-k *query generation, SRTs like Fig. 4(e) and 4(f) would be generated from the graph in Fig. 4(d).*

# 6. CONCLUSION

In this paper, we proposed a novel framework for structural refinement of XML keyword search. Our framework aims at providing an automatic mechanism for refining XML keyword queries into compatible tree patterns. We have transformed the structured query generation problem into a classical combinatorial optimization problem, i.e. computing the *k-best* MSRT from the relation graph composed by the querying terms.

# 7. REFERENCES


[1] M. Aigner and G. M. Ziegler. *Proofs from THE BOOK (4th Edition)*. Springer, 2009.

[2] S. Amer-Yahia and et al. Report on the DB/IR Panel at SIGMOD 2005. *SIGMOD Record*, 34(4):71–74, 2005.

[3] N. Bruno, N. Koudas, and D. Srivastava. Holistic Twig Joins: Optimal XML Pattern Matching. *SIGMOD*, 2002.

[4] P. M. Camerini, L. Fratta, and F. Maffioli. The k best spanning arborescences of a network. *Networks*, 10(2):91–110, 1980.

[5] L. Hlaoua and M. Boughanem. Structural Relevance Feedback in XML Retrieval. *FQAS*, pages 168–178, 2009.

[6] A. Nierman and H. V. Jagadish. ProTDB: Probabilistic Data in XML. *VLDB*, pages 646–657, 2002.

[7] D. Petkova, W. B. Croft, and Y. Diao. Refining Keyword Queries for XML Retrieval by Combining Content and Structure. *ECIR*, 2009.

[8] R. E. Tarjan. Finding Optimum Branchings. *Networks*, 7(1):25–35, 1977.

[9] W. T. Tutte and Nash-Williams. *Graph Theory*. Cambridge University Press, Cambridge, revised. edition, 2001.

[10] A. Woodley and S. Geva. NLPX - An XML-IR System with a Natural Language Interface. *ADCS*, pages 71–74, 2004.

[11] Y. Xu and Y. Papakonstantinou. Efficient keyword search for smallest LCAs in XML databases. *SIGMOD*, 2005.