Improving the Performance of Identifying Contributors for XML Keyword Search

Rung-Ren Lin[1], Ya-Hui Chang[2]*, and Kun-Mao Chao[1]
[1]Department of Computer Science and Information Engineering
National Taiwan University, Taipei, Taiwan.
{r91054, kmchao}@csie.ntu.edu.tw
[2]Department of Computer Science and Engineering
National Taiwan Ocean University, Keelung, Taiwan.
yahui@ntou.edu.tw

ABSTRACT

Keyword search is a friendly mechanism for users to identify desired information in XML databases, and *LCA* is a popular concept for locating the meaningful subtrees corresponding to query keywords. Among all the LCA-based approaches, MaxMatch [9] is the only one which could achieve the property of *monotonicity* and *consistency*, by outputting only *contributors* instead of the whole subtree. Although the MaxMatch algorithm performs efficiently in some cases, there is still room for improvement. In this paper, we first propose to improve its performance by avoiding unnecessary index accesses. We then speed up the process of *subset detection*, which is a core procedure for determining contributors. The resultant algorithm is called *MinMap* and *MinMap*$^+$, respectively. At last, we analytically and empirically demonstrate the efficiency of our methods. According to our experiments, our two algorithms work better than the existing one, and *MinMap*$^+$ is particularly helpful when the breadth of the tree is large and the number of keywords grows.

1. INTRODUCTION

Keyword search provides a convenient interface for users to obtain desired information from XML documents, but irrelevant data may be returned due to lacking exact query semantics. Therefore, there are a lot of researches on automatically reasoning the meaningful answers for users.

In general, an XML document could be viewed as a rooted tree, where each node represents an element or contents. The LCA-based approaches will identify *the LCA node* first, which contains every keyword under its subtree at least once [2, 4, 5, 6, 7, 12, 13, 14]. Since the LCA nodes sometimes are not very *specific* to users' query, Xu and Papakonstantinou [12] proposed the concept of *SLCA (smallest lowest common ancestor)*, where a node is said to be an SLCA if (i) it is an LCA, and (ii) it has no descendant nodes that also contain all the keywords. For example, consider the XML tree in Figure 1, where each node is associated with a unique Dewey number [15]. For the query Q_0 = (*pitcher, name*), the LCA list is [1, 1.1, 1.1.1, 1.1.3]. Since nodes 1 and 1.1 have descendant nodes 1.1.1 and 1.1.3 that are also LCA, only two nodes, that is, 1.1.1 and 1.1.3, are SLCA. We could see that the two corresponding *player* elements contain more specific information than the elements *players* (1.1) and *team* (1).

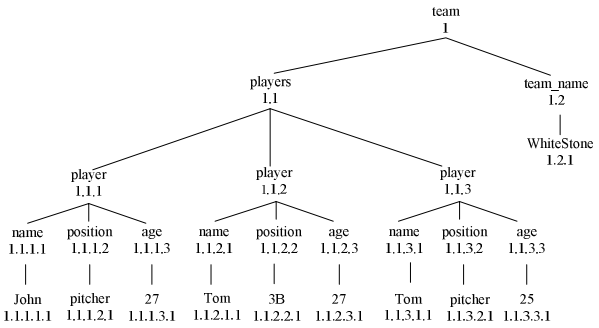


Figure 1: A sample XML tree.

The SLCA approach achieves *specificity* based on the ancestor/decendant relationship, but they do not distinguish the importance of sibling nodes. Therefore, Liu and Chen [9] further proposed the concept *contributor*, where a node is a contributor if it corresponds to more (or equal to) keywords compared with its sibling nodes, and only contributors will be returned. The basic concept of the contributor is to keep only those nodes which contain *richer* information under their subtrees than their siblings. Consider another query Q_1 = (*players, pitcher, Tom*). Since the subtree rooted at node 1.1.2 contains key-

*To whom all correspondence should be sent.

word *Tom* and the subtree rooted at node 1.1.3 contains keywords *Tom* and *pitcher*, node 1.1.3 will be a contributor, and will *prune* node 1.1.2.

One important characteristic of this work is that it satisfies the *monotonicity* and *consistency* properties, which capture a reasonable connection between the new query result and the original query result after an update to the query or to the data. Briefly, the monotonicity property indicates the change to the number of SLCA nodes, and the consistency property describes the change to the content of query result. These properties are sensible and worthwhile, yet none of the other approaches satisfy both properties.

The authors in [9] gave an efficient algorithm *MaxMatch* to locate all the contributors, but there is still room for improvement. First, we identify the places where index accesses are not necessary, and thus avoid unnecessary I/O accesses. Second, we improve the process of *subset detection*, which is a core operation in finding the contributors. We construct the corresponding algorithm MinMap and MinMap$^+$, and perform a series of experiments. Experimental results show that the MaxMatch algorithm is less efficient than our approach when the breadth of the tree is large and the number of keywords grows. Note that the two identified questions are generic and not limited in this framework. Although not major theoretic breakthrough, our findings indeed speedup query processing to a large extent, and can be applied to questions in different domains.

The rest of this paper is organized as follows. In Section 2, we briefly introduce the MaxMatch algorithm [9]. We then present the algorithm MinMap in Section 3, which improves the execution time by avoid unnecessary index accesses. The method for speeding up subset detection is described in Section 4. We further discuss the experimental studies in Section 5. Finally, Section 6 concludes this paper.

2. MAXMATCH

We explain the MaxMatch approach [9] in this section. The sample XML tree given in Figure 1 will be used in the running examples throughout this paper.

We first deliver the definitions given in MaxMatch. A node is a *match* if its tag name or the content corresponds to a given query keyword. The *descendant matches* of a node n, denoted as $dMatch(n)$, are a set of query keywords, each of which has at least one match in the subtree rooted at n. In addition, a node n is a *contributor* if (i) n is the descendant of a given SLCA or n itself is one of the SLCAs, and (ii) n does not have a sibling n' such that $dMatch(n') \supset dMatch(n)$. For each SLCA, the MaxMatch will return the root-to-a-match path, as long as the nodes in the path are all contributors.

Take query Q_1 as an example again. We have $dMatch(1.1.1) = \{pitcher\}$, $dMatch(1.1.2) = \{Tom\}$, and $dMatch(1.1.3) = \{pitcher, Tom\}$. According to the definitions mentioned above, nodes 1.1.1 and 1.1.2 are not contributors since both $dMatch(1.1.1)$ and $dMatch(1.1.2)$ are the proper subsets of $dMatch(1.1.3)$. The output will be the node list: [1.1 (*players*), 1.1.3 (*player*), 1.1.3.1 (*name*), 1.1.3.1.1 (*Tom*), 1.1.3.2 (*position*), 1.1.3.2.1 (*pitcher*)].

Observe that the second condition of the contributor involves *subset detection*. Given a node n_p with b children, the naive algorithm, which compares all possible pairs among those b children, takes $O(b^2)$ time and is time-consuming. To promote the efficiency, MaxMatch uses a boolean array $dMatchSet$ to record the information of each of n_p's child. Specifically, let $S = \{n_1, n_2, ..., n_b\}$ be the children set of n_p. The $dMatchSet$ array for n_p is of length 2^w, where w is the number of keywords. All the bits are initialized as $false$ at the beginning. The j^{th} bit, $0 \leq j \leq 2^w - 1$, is set to *true* if and only if n_p has at least one child $n_i \in S$ such that the decimal value of $dMatch(n_i)$ is j. Then, for each $n_i \in S$, MaxMatch can determine whether it is a contributor or not in $O(2^w)$ time by scanning the whole $dMatchSet$ array. Hence, it totally takes $O(b \cdot 2^w)$ time to deal with all the b children. Note that the number of query keywords w is generally small, and the branch factor b may be very large in the XML trees. Therefore, under the condition of $b \gg w$ (such as $b > 2^w$), MaxMatch works better than the naive algorithm.

Although MaxMatch is quite efficient, we will propose a more efficient way to perform subset detection which is described in Section 4. In addition, while setting the $dMatch$ values, MaxMatch retrieves the tag names of a node using the index. Since some of the nodes are pruned at last, it may cause redundant I/O accesses. We will propose an improved algorithm in Section 3.

3. AVOIDING INDEX ACCESSES

We first introduce the definitions of our approach. The *match tree* of a node t, denoted as $mTree(t)$, consists of the nodes along the path from each match up to t. The nodes in the match tree without matching any query keyword are called *non-keyword nodes*. Besides, node n is called a *hit node* if (i) n is contained in the match tree rooted at a given SLCA, (ii) n is a non-keyword node, and (iii) all the nodes

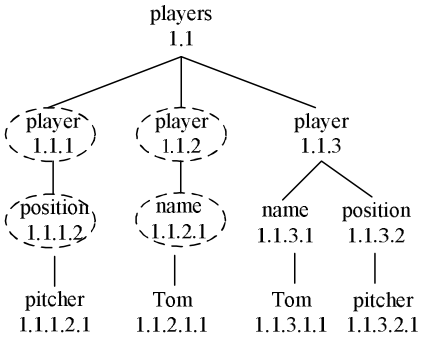


Figure 2: The match tree of Q_1.

on the path from n up to the SLCA are contributors. On the contrary, node n is called a *miss node* if n satisfies the first two conditions of hit node, but does not satisfy the third condition. Consider query Q_1 again. Recall that nodes 1.1.1 and node 1.1.2 are pruned by node 1.1.3. Therefore, the hit node list is [1.1.3, 1.1.3.1, 1.1.3.2], and the miss node list is [1.1.1, 1.1.1.2, 1.1.2, 1.1.2.1], where miss nodes are depicted by dashed circles as shown in Figure 2.

Recall that MaxMatch retrieves the tag names of nodes using indexes, but some of them are eventually pruned. In our approach, we let every node directly inherit the $dMatch$ information from its children without retrieving its own tag name, and use a variable to record if it is a non-keyword node. We retrieve its tag name only when it is confirmed to be a contributor, and thus save unnecessary index accesses. Interested readers can refer to [8] for the complete algorithm. We further define the *miss rate* as: Σ miss nodes /(Σ hit nodes + Σ miss nodes) for match trees of the SLCAs. As an example, the miss rate in Figure 2 is $4/(3+4) \approx 57\%$. Therefore, our approach will save up to 57% index accesses compared with the MaxMatch when constructing non-keyword part of the match tree. Later, the miss rate will be used for analyzing the processing time between MinMap and MaxMatch, and we will have more details in Section 5.

4. IMPROVING SUBSET DETECTION

Recall that MaxMatch has the time complexity $O(b \cdot 2^w)$ when performing subset detection. In this section, we propose an $O(b) + O(w \cdot 2^w)$-time method, to speed up the process when the breadth of the tree is large and the number of keywords grows.

4.1 The Algorithm

Consider a parent node in $mTree(t)$ with b children $\{n_1, n_2, ..., n_b\}$ and their descendant matches

Input: a set of b nodes $N = \{n_1, n_2, \ldots, n_b\}$ and their descendant matches $D = \{dMatch(n_1), dMatch(n_2), \ldots, dMatch(n_b)\}$. Suppose the number of keywords is w.

Output: all the nodes in N that are not pruned by any of their siblings.

FADC(*D*)
1: allocate an array *A* of length 2^w
2: set every element of *A* to *empty*
3: **for** each $dMatch(n_i) \in D$ $(1 \le i \le b)$ **do**
4: $k \leftarrow num(dMatch(n_i))$
5: **if** *A*[*k*] = *empty* **then**
6: *A*[*k*] ← *equal*
7: *SetSubset*($dMatch(n_i)$)
8: **for** each $dMatch(n_i) \in D$ $(1 \le i \le b)$ **do**
9: $k \leftarrow num(dMatch(n_i))$
10: **if** *A*[*k*] = *subset* **then**
11: prune n_i
12: output all of the unpruned nodes

SetSubset(*d*)
1: **for** each max-subset *d'* of *d* **do**
2: $k \leftarrow num(d')$
3: **if** *A*[*k*] = *empty* **then**
4: *SetSubset*(*d'*)
5: *A*[*k*] ← *subset*

Figure 3: The algorithm for improving subset detection.

$D = \{dMatch(n_1), ..., dMatch(n_b)\}$. The problem of determining contributors is to prune those nodes whose $dMatch$ sets are the proper sets of those of their siblings.

Suppose query Q has w keywords. There are totally 2^w distinct subsets of Q. We propose to classify each distinct subset d of Q into three *states* as follows:

- *empty*: There does not exist any $dMatch(n_i) \in D$ such that $d \subseteq dMatch(n_i)$.
- *subset*: There exists at least one $dMatch(n_i) \in D$ such that $d \subset dMatch(n_i)$.
- *equal*: There does not exist any $dMatch(n_i) \in D$ such that $d \subset dMatch(n_i)$. However, there exists at least one $dMatch(n_i) \in D$ such that $d = dMatch(n_i)$.

The main idea of our approach is to set the states of all the subsets of each $dMatch(n_i) \in D$, to facilitate later processing. To achieve this purpose, an array of length 2^w is used to record the state of every distinct subset of query Q, where w is the number of keywords of Q. We also define the concept of *max-subset*. Set d' is said to be a max-subset of set d if (i) $d' \subset d$, and (ii) $|d'| = |d| - 1$, where $|d|$ and $|d'|$ represent the numbers of keywords in sets $|d|$ and $|d'|$, respectively. It is obvious that d has exactly $|d|$ distinct max-subsets.

Figure 3 shows the complete algorithm. The input is a set of sets D, and every set in D is denoted as $dMatch(n)$ which represents the keywords that are contained in the subtree of node n.

0000	subset	1000	empty
0001	empty	1001	empty
0010	subset	1010	empty
0011	empty	1011	empty
0100	subset	1100	empty
0101	empty	1101	empty
0110	equal	1110	empty
0111	empty	1111	empty

(a) After $dMatch$(1.1.1) is processed

0000	subset	1000	subset
0001	empty	1001	empty
0010	subset	1010	subset
0011	empty	1011	empty
0100	subset	1100	subset
0101	empty	1101	empty
0110	subset	1110	equal
0111	empty	1111	empty

(b) After $dMatch$(1.1.3) is processed

Figure 4: The state-array of query Q_2.

At first, we allocate an array A of length 2^w to record the state of each distinct subset of Q. Each element of A is set to *empty* at the beginning in line 2. The *num* function (in line 4) transfers the $dMatch(n_i)$ into a decimal value. It keeps a boolean array of length w to record the keywords contained in $dMatch(n_i)$ set and then transfers the boolean array into a decimal value. Specifically, the j^{th} bit of this boolean array is set to *true* if $dMatch(n_i)$ contains the j^{th} keyword of Q. Let $k = num(dMatch(n_i))$. We check the state of $A[k]$ for each $dMatch(n_i) \in D$. If $A[k]$ is *empty* in line 5, it means that so far none of the sets from $dMatch(n_1)$ to $dMatch(n_{i-1})$ equal to $dMatch(n_i)$ or are the supersets of $dMatch(n_i)$. We then set $A[k]$ to *equal* and call procedure $SetSubset(dMatch(n_i))$ to set the states of $dMatch(n_i)$'s max-subsets. On the contrary, if $A[k]$ is not *empty* in line 5, we skip $dMatch(n_i)$ no matter the state of $A[k]$ is *equal* or *subset*. Because it implies that there may exist one or more sets from $dMatch(n_1)$ to $dMatch(n_{i-1})$ that equal to $dMatch(n_i)$ or are the supersets of $dMatch(n_i)$.

Procedure *SetSubset* takes a set d as the parameter, and recursively calls itself by sending every max-subset d' of d as the parameter if $A[k] =$ *empty*, where k is the decimal value of d' in Procedure *SetSubset*. After processing each $dMatch(n_i) \in D$ (lines 3 to 7), the state of $A[num(dMatch(n_i))]$ $(1 \leq i \leq b)$ is either *equal* or *subset*. Clearly, node n_i should be pruned if the state of $A[num(dMatch(n_i))]$ is eventually set to *subset*. Therefore, we can obtain all of the contributors (unpruned nodes) easily.

Example 1. Consider query Q_2 = (*25, pitcher, name, players*). Node 1.1 is the only SLCA of Q_2, and we are going to determine the contributors among nodes 1.1.1, 1.1.2, and 1.1.3. Suppose the first keyword (from left to right) corresponds to the leftmost bit. Therefore, the $num(dMatch)$ values of these three nodes are 0110_{bin}, 0010_{bin}, and 1110_{bin}, respectively. At first, an array A of length 16 (2^4) is allocated, and each $A[k]$ $(0 \leq k \leq 15)$ is assigned as *empty*. Consider the first input $dMatch$(1.1.1). Since $A[0110_{bin}]$ is *empty*, we set $A[0110_{bin}]$ to *equal* and then call procedure *SetSubset* by parameter $dMatch$(1.1.1). Accordingly, $A[0010_{bin}]$, $A[0100_{bin}]$, and $A[0000_{bin}]$ are recursively set to *subset* by Procedure *SetSubset* (as shown in Figure 4 (a)). The second input is $dMatch$(1.1.2). Clearly, nothing is changed since the state of $A[0010_{bin}]$ is not *empty*.

At last, consider the third input $dMatch$(1.1.3). We first set $A[1110_{bin}]$ to *equal* and then call procedure *SetSubset* to set the max-subsets of $dMatch$(1.1.3). Note that $A[0110_{bin}]$ is set to *equal* by $dMatch$(1.1.1), and now it would be changed to *subset*. We skip discussing the latter two max-subsets of $dMatch$(1.1.3), because they are quite similar to that of the first input $dMatch$(1.1.1). In summary, the final state-array of Q_2 is shown in Figure 4 (b). Since $A[0110_{bin}]$ and $A[0010_{bin}]$ are both set to *subset*, nodes 1.1.1 and 1.1.2 are pruned eventually. □

4.2 Time Complexity

Procedure *SetSubset* is called only when the state of a given $A[k]$ is *empty*, and then $A[k]$ will be set to *subset*. Obviously, procedure *SetSubset* is repeated at most 2^w times. In addition, for each call of procedure *SetSubset*, it takes $O(w)$ time to prepare all the max-subsets. Therefore, procedure *SetSubset* costs $O(w \cdot 2^w)$ during the whole process. At last, it takes $O(b)$ time to determine all of the pruned elements, so the time complexity is $O(b) + O(w \cdot 2^w)$ in total.

5. EXPERIMENTAL STUDIES

We have implemented the approached described in Section 3, and call the system MinMap. We have also included the approach described in Section 4 in MinMap and call the resultant algorithm MinMap$^+$. Therefore, the only difference between MinMap and MinMap$^+$ is in processing subset detection. All these algorithms are implemented in C++ with the environment of Windows XP and Visual Studio 6.0. In addition, the two proposed systems utilized two indices based on B-tree structure, similar to MaxMatch.[1] These indices are created and accessed based on the Oracle Berkeley DB [16].

5.1 Comparing MaxMatch with Our Approaches

In this section, we will compare the processing time of MinMap, MinMap$^+$, and MaxMatch. The

[1]In the first index, the key is the Dewey number of a node and returns the corresponding tag name. In the second index, the data associated with each keyword k is a sorted list of Dewey numbers of all the nodes which contain keyword k.

Table 1: Test queries for DBLP data set.

No.	query	FQ	MR
QD_1	1998, volume, 1	HHH	1%
QD_2	1998, LNCS, volume, 1	HLHH	98%
QD_3	JACM, cite, On the complexity of integer programming	LHL	35%
QD_4	2001, FOCS, article	HLH	84%
QD_5	2000, Springer, editor	HLH	0%
QD_6	1999, cdrom, Information Processing Letters	HHL	97%

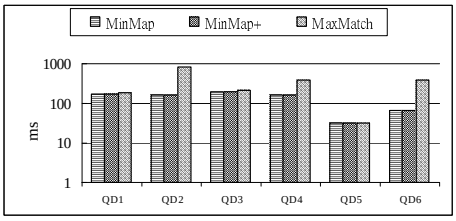
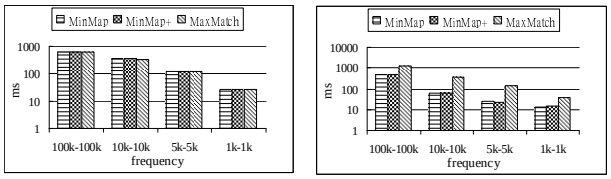


Figure 5: Processing time of the test queries in Table 1.

DBLP data set is used to perform the experiments. We start by designing queries based on different combination of keywords with high-frequency and/or low-frequency. Table 1 shows the testing queries. The keyword frequencies (denoted as FQ) are displayed in the third column, where "H" stands for *high* and "L" stands for *low*. The miss rate of the queries (denoted as MR) are also specified in the last column.

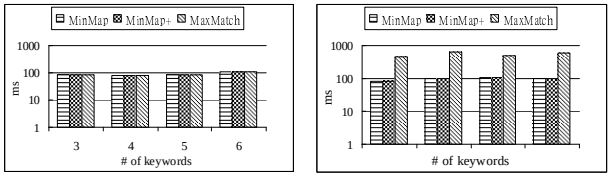
The results of processing time are shown in Figure 5. We can see that keyword frequencies have no direct impact on the performance of the three systems. For example, QD_4, QD_5, and QD_6 all consist of two high-frequency keywords and one low-frequency keyword. However, MaxMatch performs the same as our two proposed systems for QD_5, but performs worse for QD_4 and QD_6. The reason is that the elements matching the querying keywords of QD_5, that is, *year*, *publisher*, and *editor*, all reside under the same parent elements, which makes the miss rate low. In contrast, among the elements matching the querying keywords of QD_4, *article* is the parent of *booktitle* and *year*, which makes the miss rate high. However, we can observe that both MinMap and MinMap$^+$ work better than MaxMatch for all those high miss-rate queries.

In the next experiment, we specifically control the frequencies of keywords to identify their relationship with the miss rate and examine how they affect the performance. We keep the minimum frequency and the maximum frequency of the keywords to be close, and vary the frequencies simultaneously. Therefore, all the keywords in the same query have similar frequencies. As shown in Figure 6, observe again that the keyword frequency do not show direct impact on performance. However, our approaches perform a lot better than MaxMatch when the miss rate is high (Figure 6(b)), and the performance is even to an order of magnitude difference. It also shows that the miss rate has no obvious relationship with keyword frequencies alone.



(a) low-miss rate (DBLP data set) (b) high-miss rate (DBLP data set)

Figure 6: Varying the keyword frequencies.



(a) low-miss rate (DBLP data set) (b) high-miss rate (DBLP data set)

Figure 7: Varying the number of keywords.

Finally, we design different scenarios by changing the number of keywords. We fix the maximum frequency of the keyword and perform random testing. We then classify all the experimental results into low miss-rate (smaller than 10%) cases and high miss-rate (larger than 90%) cases. The experimental results depicted in Figure 7 show the similar result to the previous scenario.

We have also applied the other two data sets to perform the experiments: SwissProt.xml[2] and baseball.xml[3]. The testing results are similar to that of DBLP data set. Due to space limitation, we omit the experimental results.

5.2 Comparing MinMap and MinMap$^+$

We then compare MinMap with MinMap$^+$ in this subsection. Recall that MinMap applies the subset detection method in the original MaxMatch algorithm, while MinMap$^+$ applies the new method proposed in Section 4. Also recall that their time complexity is affected by the branch factor b and the number of keyword w. We use several datasets with different branch factor to perform the test. We then examine how the number of keywords affect

[2]http://www.cs.washington.edu/research/xmldatasets/.
[3]http://www.cafeconleche.org/books/biblegold/examples/.

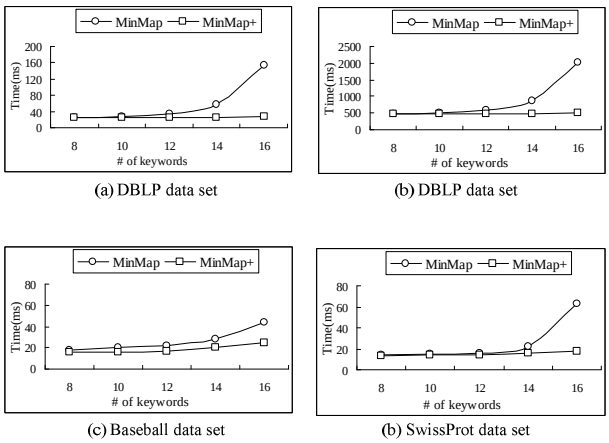


Figure 8: Comparing MinMap and MinMap$^+$.

the performance. We first apply the DBLP dataset. In Figure 8 (a)-(b), the maximum branch factors of the match trees are about 5,000 and 50,000, respectively. We also make the total frequencies of the keywords not change too much when the number of keywords grows. The result shows that the processing time of MinMap increases sharply while the processing time of MinMap$^+$ increases smoothly along with the number of keyword in both cases.

We then apply the Baseball and SwissProt data sets to perform the similar experiment. In Figure 8 (c)-(d), the maximum branch factors are about 50, and 7,000, respectively. Since the maximum branch factor of Figure 8 (c) is small compared with the other three, the improvement between MinMap and MinMap$^+$ is therefore not that large. However, MinMap$^+$ still outperforms MinMap.

6. CONCLUSIONS

In this paper, we propose two algorithms to improve the efficiency of MaxMatch. The MinMap algorithm is designed based on eliminating unnecessary index accesses during the construction of the match tree. The MinMap$^+$ algorithm is proposed to speed up the computation of subset detection. The experimental results show that MinMap outperforms MaxMatch when the miss rate is high. The experiments also show that MinMap$^+$ is particularly helpful when the breadth of the tree is large and the number of keywords grows. As part of our future work, we are interested in designing a novel ranking scheme to order the query results so that users may focus on the most desirable ones.

7. REFERENCES

[1] B. Cate and C. Lutz. The Complexity of Query Containment in Expressive Fragments of XPath 2.0. In *Journal of ACM*, pages 73-82, 2009.

[2] S. Cohen, J.Mamou, Y.Kanza, and Y.Sagiv. XSEarch: A Semantic Search Engine for XML. In *VLDB*, pages 45-56, 2003.

[3] S. Flesca, F. Furfaro, and S. Greco. A Query Language for XML Based on Graph Grammars. In *Journal of WWW*, pages 125-157, 2002.

[4] L. Guo, F. Shao, C. Botev, and J. Shanmugasundaram. XRANK: Ranked Keyword Search over XML Documents. In *SIGMOD*, pages 16-27, 2003.

[5] G. Koloniari and E. Pitoura. LCA-Based Selection for XML Document Collections. In *WWW*, pages 511-520, 2010.

[6] G. Li, J. Feng, J. Wang, and L. Zhou. Effective Keyword Search for Valuable LCAs over XML Documents. In *CIKM*, pages 31-40, 2007.

[7] Y. Li, C. Yu, and H. V. Jagadish. Schema-Free XQuery. In *VLDB*, pages 72-83, 2004.

[8] R. R. Lin, Y. H. Chang, K. M. Chao. Faster Algorithms for Searching Relevant Matches in XML Databases. In *DEXA* (LNCS 6261), pages 290-297, 2010.

[9] Z. Liu and Y. Chen. Reasoning and Identifying Relevant Matches for XML Keyword Search. In *PVLDB*, pages 921-932, 2008.

[10] I. Tatarinov, S. Viglas, K. Beyer, J. Shanmugasundaram, E. Shekita, and C. Zhang. Storing and Querying Ordered XML Using a Relational Database System. In *SIGMOD*, pages 204-215, 2002.

[11] X. Wu, D. Theodoratos, S. Souldatos, T. Dalamagas, and T. Sellis Evaluation Techniques for Generalized Path Pattern Queries on XML Data. In *Journal of WWW*, pages 441-474, 2010.

[12] Y. Xu and Y. Papakonstantinou. Efficient Keyword Search for Smallest LCAs in XML Databases. In *SIGMOD*, pages 527-538, 2005.

[13] Y. Xu and Y. Papakonstantinou. Efficient lca based keyword search in xml data. In *EDBT*, pages 535-546, 2008.

[14] R. Zhou, C. Liu, and J. Li. Fast ELCA Computation for Keyword Queries on XML Data. In *EDBT*, pages 549-560, 2010.

[15] Dewey Decimal Classification. http://www.oclc.org/dewey/.

[16] Oracle Berkeley DB. http://www.oracle.com/technetwork/database/.