Selection, Generation and Extraction of MCCTree using XMCCTree

Sazaly U.S., Selamat M.H., Baharom S., Ab. Jabar M.

***Abstract*—**Previous research proposed notions of CLCA and MCLCA to answer keyword query in XML document. The notions are implemented in its proposed algorithms and the result, namely MCCTree, is ranked with its proposed ranking method. The algorithms transform the XML tree into a compact global tree called CGTree, and select the MCCTree from the CGTree. The resulted MCCTree is in a compact structure; however the calculation in the ranking method requires the original structure as in the XML tree. Thus, this paper presents a new algorithm that implements the same notions with different approach. The MCCTree is returned in a structure as required by the ranking method. This algorithm, called *XMCCTree*, improve the efficiency of producing a set of MCCTree in answering keyword query in XML document.

***Keywords*—** XMCCTree, algorithm, MCLCA, CLCA, MCCTree, XML, keyword query.

I. Introduction

THE information retrieval research in XML intends to find the best approach on accessing data from XML document. This intention is strengthen by a yearly workshop organized by INEX [1] to analyze the approaches that have been introduced. The research in XML information retrieval (IR) has sets out some important requirements that must be in an XML IR system. The research in [2] stated that an XML IR system requires a query language to specify the component's nature in processing the query, a representation strategies that integrate the relationship between the content and the structure, and the ranking strategies to rank the results. All of these three requirements have been considered by the XML IR researchers. However, research to find the best approach is still ongoing, especially on the query language.

Manuscript received October 9, 2011. This research was supported by Fundamental Research Grant Scheme (FRGS), Research Fund Number: 03-04-10-867FR, under Ministry of Higher Education, MALAYSIA.

Sazaly U.S. Author is a researcher at Faculty of Computer Science and Information Technology, 43400 Serdang, Universiti Putra Malaysia. (Corresponding author to provide phone: 603-8647-1720; fax: 603-8946-6576; e-mail: ummusulaim2311@ gmail.com).

Selamat M.H. is with Universiti putra Malaysia. He is now with the Department of Computer Science as an Associate Professor, Faculty of Computer Science and Information Technology, Universiti Putra Malaysia, 43400 Serdang, Malaysia (e-mail: hasan@fsktm.upm.edu.my).

Baharom S. is with the Information System Department, Faculty of Computer Science and Information Technology, Universiti Putra Malaysia, 43400 Serdang, Malaysia (e-mail: salmi@fsktm.upm.edu.my).

Ab. Jabar M. is with the Information System Department, Faculty of Computer Science and Information Technology, Universiti Putra Malaysia, 43400 Serdang, Malaysia (e-mail: marzanah@fsktm.upm.edu.my).

Nowadays, almost all applications have accepted XML as a medium to store data. Thus, research in XML query has become a major focus in XML IR. Several notions and algorithms have been introduced to support query for XML. These approaches vary with the certain requirement restrict to some application of interest [3]. In processing a query, the XML document is first presented in a data model. Most researches modeled XML document in a tree structure. When the XML document is presented in a tree form, the notion of Lowest Common Ancestor (LCA) [4] is very suitable to be used in selecting possible answer for keyword query. As stated in [5], when a tree is defined in two views of subtree, an element in each view might be the same element in the tree. For example, a query with keywords *a*, *b* and *c* exists in a tree as shown in Figure 1.

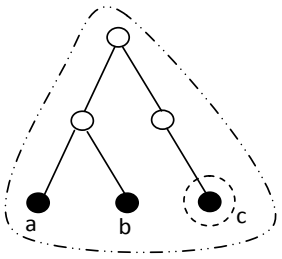

Fig. 1 An example of keyword tree with keyword *a*, *b* and *c*

When the query processing considers to return the result using the approaches of the semantic AND and single occurrence of the keyword, this query might return two answers. The AND semantic return a subtree with all the keywords in it. The single occurrence returned just a single keyword node *c* as the answer because the keyword node *c* is located far from the subtree of keyword nodes *a* and *b*. The same node *c* is returned twice when the query processing looks at the tree from the two different approaches. The research [5] informally says the two elements are strictly equal if they are the same element. It is inefficient if the result repeat the same element as the answer.

Thus, LCA overcomes this factor by allowing an element to be in only one structure of subtree so that the result is returned without recurrence element in other subtrees. Many researchers also provide algorithm to implement their LCA notions. The XRank [6] used stack in its RDIL Query

Processing algorithm. The enhance LCA by [7] and [8] used collection or list to implement their algorithms. Then, stack is reused in implementing notions of Compact LCA (CLCA) and Maximal CLCA (MCLCA) [9] in its proposed algorithm. At the end, the output of the algorithm must satisfy the notion that they introduced before.

In this paper, a new algorithm to generate a set of MCCTree is presented, called *XMCCTree* (eXtended MCCTree). Theoretically, the proposed algorithm is more efficient compared to the previous algorithms, the *CGTreeGenerator* and the *MCCTreeGenerator*. The *XMCCTree* is an enhancement of the previous algorithms which produce a set of MCCTree in a form that can be used directly as input for the ranking method. Section 2 discusses about several approaches proposed in previous research. Section 3 explains the concepts, steps and pseudocode of the *XMCCTree*. Section 4 presents comparisons between *XMCCTree* and *CGTreeGenerator+MCCTreeGenerator* with an example. Finally, conclusions were presented in Section 5.

II. The Previous Approach

In the beginning of the XML IR, the researcher adapts the concept of query from the previous relational database and Object-Oriented database. The concept of query is using a query language. Among the earliest XML query language is an XML-QL proposed in [10]. The XML cannot adapt the existing query language because it cannot directly capture the relationship between the tag and its content. The research led to the introduction of a path notation in making a query. A query needs to presents a 'URL' of the searched data in navigating through the structure of the XML document. The used of path regression are implemented in Xpath 1.0 [11], XQuery [12] and XPath 2.0 [13].

Besides the path regression, the researcher also used the traditional clause based from relational query language, SQL, which provide a pattern in restructuring the data, and notions of functional language adapted from Object-Oriented query language, OQL. These concepts are adapted in Quilt [14] which realize the potential on querying exchange between documents and databases. The initial proposed query languages will be easier if the user knows the exact location of the data. A query is based on the route path provided by the user. However, the problem will arise if the user does not have knowledge of the XML or the underlying structure.

The implementation of keyword query proposed in [15] is intended to support a novice user to make a query without the knowledge of the XML structure. Many researches try to enhance the quality of the XML keyword query processing. The XKeyword [16] performed 2 stages of query processing. The preprocessing stage stores the keyword's elements and creates a set of connection relations using inline fragments. This schema is then retrieved in the query processing stage, connects between them and passed to the query optimizer to display the results.

Several researches proposed a formatted query which allow user to input the keyword based on specification and features provide in the proposed system. The XSearch [17] format its keyword query by combining query semantics and path regression. The CTree [18] method received a keyword query after the XML tree is transformed into compact tree, Ctree, and indexed. The tree and the scanned data are then presented to the user. User can define the query processing features in making a query. Note that the formatted query still requires user to have knowledge on defining or selecting the query's features.

The XML tree presents the ancestor-descendant relationship or the parent-child relationship. The hierarchical level in the tree makes the notion of Lowest Common Ancestor (LCA) is the suitable method in processing the XML keyword query. This notion has been used in XRank [6]. Then, Yu and Yannis [7] introduced a Smallest LCA (SLCA) to avoid returning a tree with overlap keyword in it. The enhanced SLCA proposed by Chong et al. [8] introduced Multiway-SLCA (MSLCA). This MSLCA adds disjunctive semantic (OR) in processing the query. The MSLCA simplified the SLCA algorithm and optimized the intention to avoid redundant in the returning tree.

When the query processing has returned a set of query result, the needs to sort the answers is required. The XXL Search Engine [19] proposed the ranking method by matching the patterns similarities between paths. The XXL Search Engine still use path regression in its query which is not suitable to support keyword query. The ranking method in keyword query has been proposed in XRank [6]. The XRank used the LCA method in its query processing. Its ranking method calculates keyword specificity, keyword proximity and a new proposed method to cater the hyperlink awareness of the element called ElemRank. Unfortunately, the XRank only used a conjunctive semantic and a single occurrence of the keyword to process the query. Research in [9] stated that the use of both conjunctive and disjunctive semantics can produce a better number of results.

The keyword query processing finds the nodes that contain keyword(s) in XML tree. Then, it will be processed to find the LCA depending on the location of the nodes. The notion of LCA is an approach to identify the relevancy of the answer and the keyword. When it is used in a structured data like in XML, the relevancy is consistent with the research in [20]. The research stated that the conformity analysis of the contents in a structured data is in accordance to the nearest-neighbor similarity. Moreover, it can establish a scale for the data which can help in the ranking calculation.

Most algorithms in LCA research use Dewey indexing method in their approaches. It indexes the node with numbers based on its structure. This indexing method can be easily manipulated in processing the query using existing functions such as sorting and defining between the lowest and the highest level of the node.

Algorithms proposed in [9] implement notions of CLCA and MCLCA with Dewey indexing method. It generates a Compact

Global Tree (CGTree) and selects a set of compact Maximal Compact Connected Tree (MCCTree) to answer XML keyword query. These notions from the previous research have been proven that the accuracy and the relevancy of the produced results outperform the existing approaches. These approaches proposed in [9] have been compared with the existing XRank [6], XSearch [17] and Multiway-Smallest-LCA (MSLCA) [8]. The notion implemented in XRank only covers a conjunctive semantic. In this approach, the conjunctive semantic allows only LCA that has all of the keywords to be return as the answer which makes the number of the returned result is less. However, the notions in [9] and MSLCA still perform well in conjunctive semantic in compared with XRank. The difference between MSLCA and CLCA+MCLCA is apparent when the disjunctive semantic is applied. On the other hand, the performance for XRank in supporting the disjunctive semantic is not accurate since it only covers the conjunctive semantic.

Both XSearch and MSLCA cover the semantic relationship when answering keyword query in XML. Research in [9] stated that the performance of XSearch when executing more keywords in a huge document is not as efficient as the CLCA+MCLCA performance. The process of maintaining the index connectivity in XSearch makes it inefficient when executing more keywords as well as executing query in a large file. Furthermore, the XSearch use a related fragment as its processing method instead of the LCA which is the focus of this research. Meanwhile, the MSLCA is an extended research of Smallest-LCA [7] which support the disjunctive semantic and optimized the process of selecting the LCA. But the process of matching between the anchor nodes is too complex which increases the processing time when selecting the result.

The notions of CLCA and MCLCA with Dewey indexing method performs very well than the compared approaches. The resulted compact MCCTree is then ranked with its ranking method which focusing on the structural compactness and the text similarities. Unfortunately, the structural compactness in the ranking method calculates the distance of nodes and levels of MCCTree using the original structure from the XML tree. We assume that there is another process to retrieve the actual structure of the MCCTree before it can be used in the ranking method. Therefore, this paper proposes an algorithm to produce MCCTree in incompact structure. The output from this algorithm can be used directly as an input for the ranking method. Table 1 summarized all the approaches described in this chapter.

Table 1 Summary of research and part of approaches in XML query

XML Query	
Approach	**Proposed Method**
Path Regression	XPath 1.0[11], XQuery [12], XPath 2.0 [13]
Traditional Clause, SQL + OQL	Quilt [14]
XML keyword query	
Approach	**Proposed method**
Pattern, Predicate, Connection Relation	Entended XML-QL [15], XKeyword [16]
Formatted Query	XSEarch [17], CTree [18]
Lowest Common Ancestor (LCA)	XRank [6], SLCA [7], MSLCA [8], XSemantic [21], CLCA & MCLCA [9]
Ranking Method	
Proposed in	**Focus**
XXL Search Engine [19]	Path Similarities
XRank [6]	Keyword Specificities, Keyword Proximity, ElemRank (Hyperlink Awareness)
CLCA & MCLCA [9]	Structural Compactness, Textual Similarities

III. The XMCCTree Algorithm

A. The concept of XMCCTree

XMCCTree is developed with intended to generate the MCCTree in incompact structure. XMCCTree uses the concept of set or collection. The three steps that can describe the XMCCTree are select, generate and extract. XMCCTree starts with selecting a node that contains keywords in it. This node is determined whether it can be a possible root node of the resulted subtree. Once determined, XMCCTree will generate the subtree of the node until it reaches the deepest node that contains the keyword. When the subtree has been generated, the subtree will be extracted from the tree as MCCTree if it satisfies some rules in line 36 to line 39 as shown in the algorithm in section 2.3.

B. The steps of XMCCTree

XMCCTree needs three types of inputs to be processed. The inputs are an indexed XML tree, a set of indexes of the keyword nodes, K, sorted by index, and a variable named CMSet, which is declared as a node root. This root node then will be assigned to the variable T, a variable to hold the generated tree. The selecting steps define whether T has child by reading the set K and detect the child, ci, which has a starting index with root T. Then, ci, is added as child for T.

Before generating the subtree of each ci under T, XMCCTree will get a number of keyword(s) in each ci stored in CKti, and a number of keyword(s) in T stored in CKT. Then, ci will be sorted in descending order of the number of keyword(s) contain in its subtree. If the CKT is equal with the CKti, the ci will be eliminated from T and added into CMSet. Otherwise, the T is possibly a MCCTree. Now, the XMCCTree will generate the complete tree of T. The XMCCTree will generate a subtree of each ci under T one-by-one by searching the index of keyword K, which has a starting index at ci. The indexes are sorted in ascending order and stored in Kci. For each ci, a new node will be added as ci's child and indexed using the Dewey index path until it reaches the deepest node that contains the keyword in the branch. This keyword node has the same index value with the first value of Kci. A variable *node* follows the generated child. Then, the first value of Kci is pop out. If Kci still has an index, it means that the subtree rooted at ci still has another branch to be generated. The variable *node* travel back through the branch until it reaches node with index same with the starting index of the first value in Kci. The node is the ancestor of the first value in Kci. The XMCCTree continue generating another branch from this node to complete the subtree ci. The process will be repeated until each ci in T is completely generated.

The last step is to select the MCCTree. If the tree T has only a subtree, the subtree of T is selected as MCCTree. Otherwise, the tree T itself will be MCCTree. All the steps continued until the CMSet is empty. The XMCCTree will return a set of MCCTree to be used in ranking method proposed by [9] to answer XML keyword query.

C. The algorithm of XMCCTree

The XMCCTree is a part of process in keyword query processing. Before the XMCCTree can be executed, the XML document must be parsed to read the data. After parsing the document, the XML data must be transformed into an XML tree and indexed using the Dewey indexing method. The methods on performing all these procedures are independent, depends on the programmer's choice on how to implement it. The XMCCTree requires an input of indexed XML tree and a set of indexes of keyword nodes sorted in ascending order. The algorithm of the XMCCTree is as follows:

```
1 Begin
2
3 XMCCTreeSet ← null;
4 CMSet = 0;   //node root
5 while CMSet is not empty do
6 {
7   r = root (T Є CMSet );
8   CMSet = CMSet - { T };
9   ci =read_keyword();
10  foreach ci
11  { add ci as child for T
12    //get |CK (ci)| ();
13    foreach ci Є children(r) //(in descending
14                              order by |CK(ci )|)
15    {   ti = ci;
16     if | CK ( T )| == | CK ( ti )| then
17          T = T - ti ;
18       CMSet = CMSet ∪ { ti };
19    else
20       break;
21    }
22    if | CK (T) | > 0 then
23       foreach ci of T
24       { Kci = getKeyword( ci );   // keywords
                                  with ci as ancestor
25         node = ci ;
26         while(node == ancestor of Kci first()
                  && Kci first() != node)
27        { tci = tci ∪ child;  // add child
                              to subtree rooted at ci
28          node = child;
29          if (node == Kci.first())
30            Kci pop();
31            if Kci eol()
32              break;
33            else
34              while (node != ancestor of
                          Kci.first())
35              { node = node.parent; }
        } }
36  if T has only 1 subtree
37     XMCCTreeSet=XMCCTreeSet ∪ {ti};
38  else
39     XMCCTreeSet=XMCCTreeSet ∪ {T};
40  }
41  return XMCCTreeSet;
42 end
```

The following example illustrates the execution of the XMCCTree algorithm. The query has 3 keywords; k1, k2 and k3. Assume that we have an indexed XML tree and a set of keyword-node indexes, K = {0000, 0001, 001, 011, 020, 02100, 02101, 02111, 030, 031}, as shown in Fig. 2. Node index [0] is assigned as a value in CMSet. The algorithm starts when the value in CMset is assigned as a node *root*. Then, the assigned value is eliminated from the CMSet.

The XMCCTree detects a number of the children for node [0] from K = {0000, 0001, 001, 011, 020, 02100, 02101, 02111, 030, 031}. Then, the child-nodes [00], [01], [02], [03]

are added under [0]. For each child, XMCCTree detects number of unique keywords that they have and sort it in descending order. Based on the XML tree, the [00] have 3 keywords, [01] has only a keyword, [02] have 3 keywords and [03] have 2 keywords.

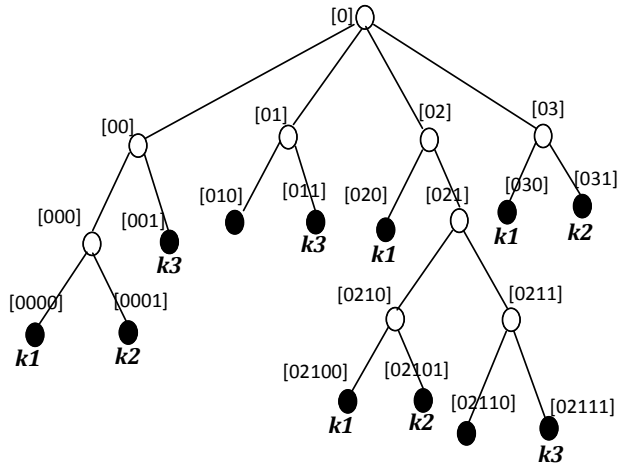

Fig. 2 An example of XML tree with keyword k1, k2 and k3

The sorted list will be [00], [02], [03] and [01]. Then, comparison between the child and the root will be done to select the possible MCCTree. When the number of unique keyword contains under root is equal with number of unique keyword contain under child *ci*, *ci* will be eliminated from the tree and added into CMSet. The graphical views of these steps are shown in Fig. 3. Children with index [00] and [02] are eliminated from the tree.

When the next child (in descending order), [03] has a number of keywords less than root (refer Fig. 3), the tree rooted at root which consists of its remaining children (node [03] and node [01]) is possibly a MCCTree. Next, XMCCTree will generate this tree.

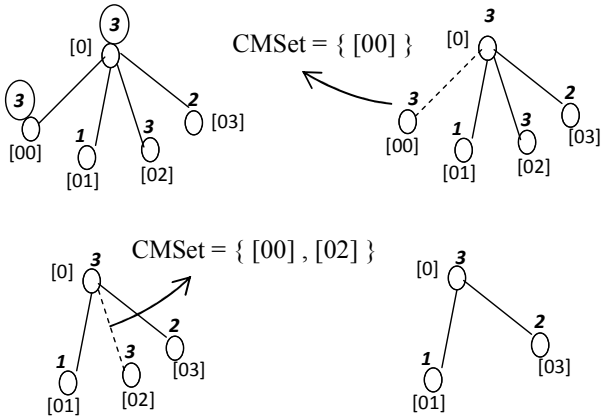

Fig. 3 Comparison between child and root

Node that contain keyword under *ci* is listed, Kci [01] = {011}. Index of *ci* is compared with the first value of Kci. XMCCTree generates node by node until the index of generated node is equal to the first value in Kci. Then, Kci eliminate that value and start the comparison and generating nodes again until the same index in Kci is reached. If the next index of Kci does not have the same ancestor with the current generated subtree (means that the nodes must not be in the same branch), the pointer will reverse through the ancestor until reach a node with the same ancestor with value in Kci. The steps can be visualized as in Fig. 4 as below:

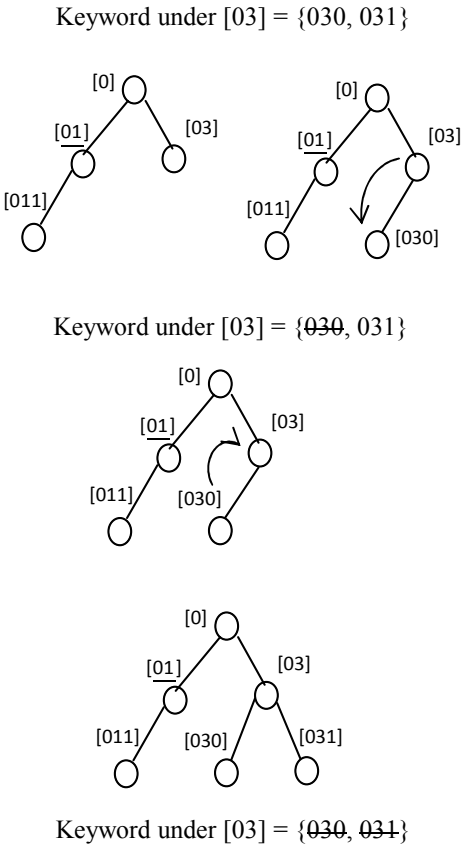

Fig. 4 illustrates part of steps in generating a tree

When all the nodes under all children have been completely generated, the tree is added into a list of MCCTree. If the tree has only a branch of child, then the subtree rooted at the child will be extracted as MCCTree. Otherwise, the tree rooted at root is returned as MCCTree.

IV. XMCCTree vs Previous Algorithm

As an enhancement of the previous algorithms, XMCCTree maintains the notions that have been used before. In this paper we used the term CGTreeMCCTreeGen to indicate a combination of the previous algorithms, the CGTreeGenerator and the MCCTreeGenerator. XMCCTree produce the same result of MCCTree as CGTreeMCCTreeGen but differ in terms of the structure of the MCCTree.

CGTreeGenerator transform XML tree into a compact global tree by eliminating linked nodes (nodes with only one subtree). Therefore, the structure of the compact global tree is in the compact structure. The process begins from the deepest nodes on the left in XML tree; which have the lowest Dewey index number. It is then inserted into a stack. The algorithm computes the LCA through travelling the nodes and the stack. At the end, a set of index node remaining in the stack construct the compact global tree.

The compact global tree is then used as an input for the MCCTreeGenerator. MCCTreeGenerator will select MCCTree in a top-down manner. Starting from root, CCTreeGenerator will define numbers of each keyword contains in each child. Child which has the same keywords and numbers of keyword with root will be eliminated from the tree to be another potential MCCTree. The remaining tree and its nodes will be a potential MCCTree. Then, *MCCTreeGenerator* will travel through the subtree of each child to identify the structure of the tree. After that, it will return the tree as MCCTree. Note that the structure of the returned MCCTree is in a compact structure because it is define from a compact global tree.

XMCCTree can be defined as an enhanced MCCTreeGenerator. XMCCTree starts generating tree from root with index [0], and identify its child from the keyword list. Then, the XMCCTree add the identified child and compute number of each keyword in each child. Similar to MCCTreeGenerator, a child that contains the same number of keyword is eliminated from the tree as a potential MCCTree. After that, XMCCTree will start generating the descendant of each child until it reaches the keyword node.

Research in [22] stated that the keyword query algorithm traverse all nodes begin from root, to find the target nodes. The traversing process makes the overhead of the algorithm is very high. However, both XMCCTree and CGTreeMCCTreeGen algorithms use the list of keyword node index as a limit in constructing the tree. It is used to avoid the algorithm travelling the nodes and paths that are not related to

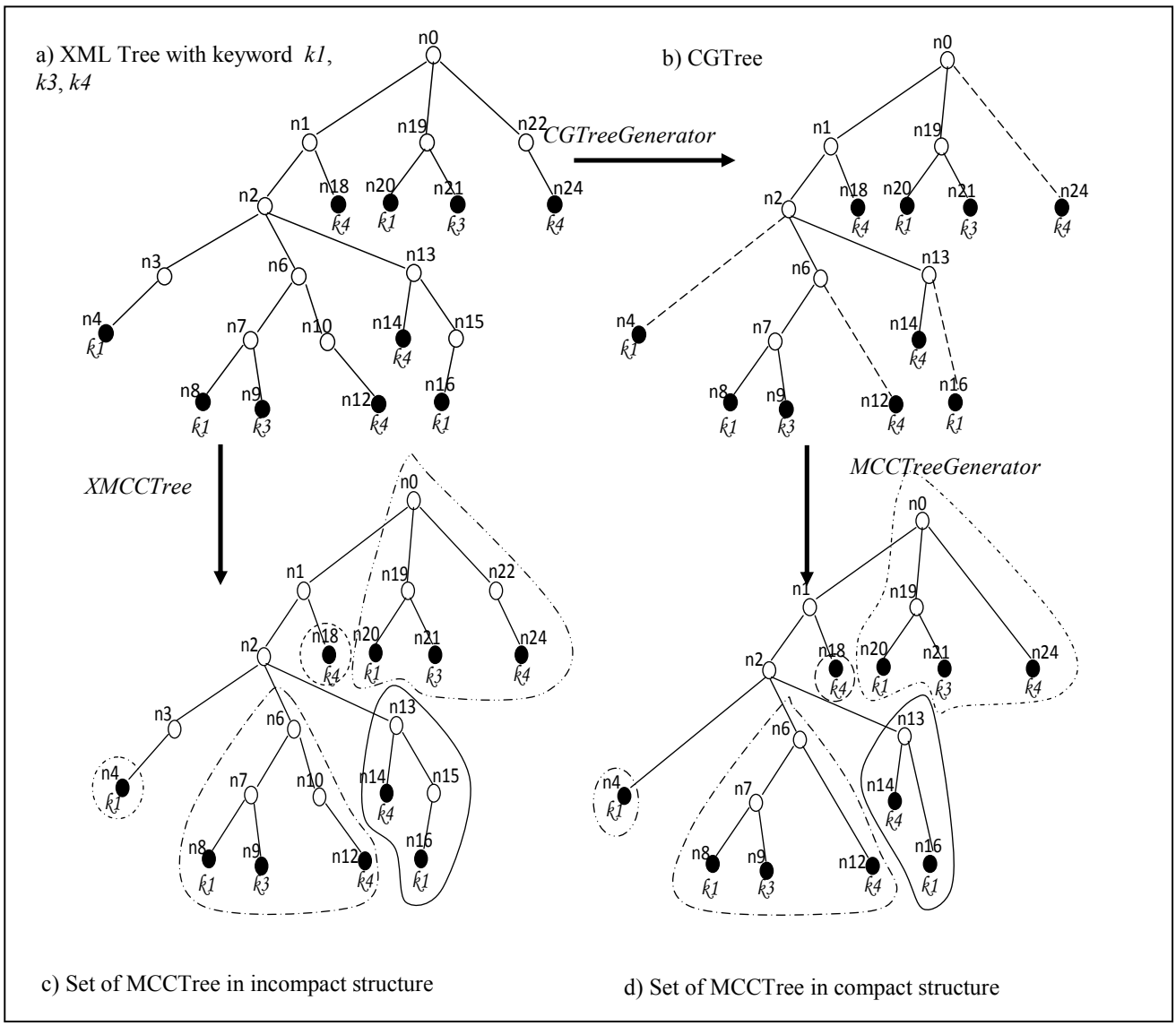

Fig. 5 Example of the XML keyword query with *XMCCTree* and *CGTreeMCCTreeGen*

the keyword.

The differences between both algorithms are:

- The *XMCCTree* is faster than *CGTreeMCCTreeGen* by avoiding the generating of compact global tree to find the lowest LCA.
- The *CGTreeMCCTreeGen* produce a set of MCCTree in a compact structure, while *XMCCTree* produce a set of MCCTree in an incompact structure.
- The *XMCCTree* is more efficient since the resulted MCCTree can be use directly in the ranking method proposed by Feng et al. [9].

Comparison between the output of the XML keyword query using *XMCCTree* and *CGTreeMCCTreeGen* is shown in Fig. 5.

In order to verify the efficiency of the XMCCTree, the mathematical theory is presented. As the focus of this paper, the algorithm is in the second level of the computational/communication process as stated in [23]. The computational time or the time complexity is the most related to the efficiency aspect. Therefore, we will use the time complexity as the mathematical theory to proof the efficiency of the algorithm and discuss the time complexity of the previous algorithms as stated in [9].

The selection of the minimal node from the XML tree requires O(*log m*) and the push and pop activities in the stack make the complexity is O(*d*). These processes are merge to construct the CGTree which make the time complexity for the CGTreeGenerator is O (*d**log *m**Σ(*i=1 to m*) | *Ti* |) where *m* is the number of keywords; *d* is a depth of the XML document; and | *Ti* | is the number of nodes that contain keyword *i*.

The counting and sorting process at the beginning of the MCCTreeGenerator needs O(*m**CN), where *m* is the number of keywords and CN is the number of the children of *n*. In generating a MCCTree, the complexity is O(*m***Ni*). *Ni* is a total number of nodes in the CGTree. Since *Ni* < 2*Σ(*i=1 to m*) |*Ti*|), the complexity of the MCCTreeGenerator is O(*m**Σ(*i=1 to m*) |*Ti*|). Since the CGTreeGenerator and the MCCTreeGenerator is a step by step algorithm, the total complexity is an addition of these two complexities. Thus, the CGTreeMCCTreeGen's time-complexity is O (*d**log *m**Σ(*i=1 to m*) |*Ti*|) + O(*m**Σ(*i=1 to m*) |*Ti*|).

In *XMCCTree*, each node *n* needs to sort its children with respect to the number of keywords contains in it, *m*, and for each child, algorithm needs to generate descendant into some level, *d*, until it reach the keyword nodes. Hence, the total complexity is O (*m**Σ(*i=1 to m*) |*Ti*|**d*).

V. Conclusion

In this paper, a new algorithm to select MCCTree from XML document is presented called *XMCCTree*. The *XMCCTree* is an enhancement of *CGTreeGenerator* and *MCCTreeGenerator* algorithms proposed in [9]. The algorithm enhanced the way to select and produce MCCTree in a way that can be use directly in the ranking method. Previous algorithms produce MCCTree in a compact structure but the *XMCCTree* produce MCCTree in incompact structure. The *XMCCTree* maintains the notions and ranking methods used to answer XML keyword query but modify its algorithm to enhance the query process.

In a worst case, previous algorithms use O (*d**log *m**Σ(i=1 to m) | T i |) + O (*m**Σ(i=1 to m) | T i |), whereas XMCCTree takes O (*m**Σ(i=1 to m) | T i |**d*).

We plan to develop a prototype of both algorithms and run an experiment to prove that the proposed algorithm, *XMCCTree,* is more efficient than the previous algorithm in producing MCCTree from XML document.

References

[1] N. Fuhr, N. Gövert, G. Kazai, and M. Lalmas, "INEX: INitiative for the Evaluation of XML Retrieval," *Proceedings of the SIGIR 2002 Workshop on XML and Information Retrieval*, vol. 2006, pp. 1-9, 2002.

[2] M. Lalmas, *XML Information Retrieval*: Taylor and Francis Group, 2009.

[3] K.-D. Schewe, B. Thalheim, S. Hartmann, H. Köhler, S. Link, T. Trinh, and J. Wang, "On the Notion of an XML Key," in *Semantics in Data and Knowledge Bases*, vol. 4925, *Lecture Notes in Computer Science*: Springer Berlin / Heidelberg, 2008, pp. 103-112.

[4] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, "On finding lowest common ancestors in trees," in *Proceedings of the fifth annual ACM symposium on Theory of computing*. Austin, Texas, United States: ACM, 1973.

[5] G. Buratti and D. Montesi, "Full-Text Capabilities for Querying XML Repositories: a Formal Model," presented at Proceedings of the 10th WSEAS International Conference on COMPUTERS, Vouliagmeni, Athens, Greece, 2006.

[6] G. Lin, S. Feng, B. Chavdar, and S. Jayavel, "XRANK: ranked keyword search over XML documents," in *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*. San Diego, California: ACM, 2003.

[7] X. Yu and P. Yannis, "Efficient keyword search for smallest LCAs in XML databases," in *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*. Baltimore, Maryland: ACM, 2005.

[8] S. Chong, C. Chee-Yong, and K. G. Amit, "Multiway SLCA-based keyword search in XML data," in *Proceedings of the 16th international conference on World Wide Web*. Banff, Alberta, Canada: ACM, 2007.

[9] J. Feng, G. Li, J. Wang, and L. Zhou, "Finding and ranking compact connected trees for effective keyword proximity search in XML documents," *Information Systems*, vol. 35, pp. 186-203, 2010.

[10] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu, "A query language for XML," *Computer Networks*, vol. 31, pp. 1155-1169, 1999.

[11] J. Clark and S. DeRose, "XML path language (XPath) recommendation.," vol. November 1999, 1999.

[12] D. Chamberlin, D. Florescu, J. Robie, J. Simeon, and M. Stefanascu., "XQuery: A query language for XML," in *Technical Report*, vol. February 2001: World Wide Web Consortium, 2001.

[13] A. Berglund, S. Boag, D. Chamberlin, M. FernÃ¡ndez, M. Kay, J. Robie, and J. r. m. SimÃ©on, "XML Path Language (XPath) 2.0 (W3C Recommendation)," 2007.

[14] G. Goos, J. Hartmanis, J. van Leeuwen, D. Suciu, G. Vossen, D. Chamberlin, J. Robie, and D. Florescu, "Quilt: An XML Query Language for Heterogeneous Data Sources," in *The World Wide Web and Databases*, vol. 1997, *Lecture Notes in Computer Science*: Springer Berlin / Heidelberg, 2001, pp. 1-25.

[15] D. Florescu, D. Kossmann, and I. Manolescu, "Integrating keyword search into XML query processing," *Computer Networks*, vol. 33, pp. 119-135, 2000.

[16] H. Vagelis, "Keyword Proximity Search on XML Graphs," presented at 19th International Conference on Data Engineering (ICDE'03), Bangalore, India, 2003.

[17] C. Sara, M. Jonathan, K. Yaron, and S. Yehoshua, "XSEarch: a semantic search engine for XML," in *Proceedings of the 29th international conference on Very large data bases - Volume 29*. Berlin, Germany: VLDB Endowment, 2003.

[18] Z. Qinghua, L. Shaorong, and W. C. Wesley, "Ctree: a compact tree for indexing XML data," in *Proceedings of the 6th annual ACM international workshop on Web information and data management*. Washington DC, USA: ACM, 2004.

[19] C. Jensen, S. Šaltenis, K. Jeffery, J. Pokorny, E. Bertino, K. Böhn, M. Jarke, A. Theobald, and G. Weikum, "The Index-Based XXL Search Engine for Querying XML Data with Relevance Ranking," in *Advances in Database Technology — EDBT 2002*, vol. 2287, *Lecture Notes in Computer Science*: Springer Berlin / Heidelberg, 2002, pp. 311-340.

[20] I. Liiv, R. Kuusik, and L. Vohandu, "Conformity Analysis with Strcutured Query Language," presented at Proceedings of the 6th WSEAS Int. Conf. on Artificial Intelligence, Knowledge Engineering and Data Bases, Corfu Island, Greece, 2007.

[21] U. Supasitthimethee, T. Shimizu, M. Yoshikawa, and K. Porkaew, "XSemantic: An Extension of LCA Based XML Semantic Search," *IEICE TRANSACTIONS on Information Systems*, vol. E92-D, pp. 1079-1092, 2009.

[22] X. Lin, D. Xu, and N. Wang, "NNQM: a novel non-navigating XML query model," in *Proceedings of the 7th Conference on 7th WSEAS International Conference on Multimedia, Internet \& Video Technologies - Volume 7*. Beijing, China: World Scientific and Engineering Academy and Society (WSEAS), 2007.

[23] M. Burgin, "Mathematical Theory of Information Technology," presented at Proceedings of the 8th WSEAS Int. Conf. on Data Networds, Communications, COmputers, Baltimore, USA, 2009.