A Fast Tree Pattern Matching Algorithm for XML Query

J. T. Yao M. Zhang
Department of Computer Science
University of Regina
Regina, Saskatchewan
Canada S4S 0A2
{jtyao,zhang2mi}@cs.uregina.ca

Abstract

Finding all distinct matchings of the query tree pattern is the core operation of XML query evaluation. The existing methods for tree pattern matching are decomposition-matching-merging processes, which may produce large useless intermediate result or require repeated matching of some sub-patterns. We propose a fast tree pattern matching algorithm called TreeMatch to directly find all distinct matchings of a query tree pattern. The only requirement for the data source is that the matching elements of the non-leaf pattern nodes do not contain sub-elements with the same tag. The TreeMatch does not produce any intermediate results and the final results are compactly encoded in stacks, from which the explicit representation can be produced efficiently.

1 Introduction

The XML (eXtensible Markup Language) is gaining popularity as a new standard for data representation and exchange on the internet. The problem of querying XML documents has been given much attention by researchers. The XML documents can be considered as semi-structured databases. There are generally two streams of query approaches to the XML data. The first one is to map the XML documents into relational databases in order to use a structured query languages such as SQL [11, 12]. It usually produces too many relations and loses some important information on relationships. For example, the explicit hierarchical relationship between XML elements may be lost. The second one is to design new query languages to extract information from XML documents [1, 4, 6, 13]. These query languages query not only the contents but also the structure.

The XML documents are usually modelled as trees and queries in XML query languages and are typically twig (or small tree) patterns with some nodes having value-based predicates. Therefore, finding all distinct matchings of a twig pattern becomes a core operation in an XML query evaluation. The existing methods for tree pattern matching in XML is typically a decomposition-matching-merging process [2, 5, 9, 16, 10].

The drawback of the decomposition-matching-merging methods is that the size of intermediate results may be much larger than the final answers. The main reason of having larger intermediate results and repeated matching of sub-patterns is due to the consideration of self-containment XML documents, i.e., an XML element that has the same tag with its sub-elements. However, in the real applications, we seldom find self-containment documents.

We propose a fast tree matching algorithm called TreeMatch that can directly find all matchings of a tree pattern in one step. The only requirement for the data source is that the matching elements of the non-leaf pattern nodes do not contain sub-elements with the same tag. There are at least two advantages of TreeMatch. First, the TreeMatch algorithm does not need to decompose the query tree pattern, as it matches the pattern against the data source directly. Therefore, it does not produce any intermediate results and does not need the merging process. Second, the final results are compactly encoded in stacks and explicit representation of the results, either as a tree or a relation with each tuple representing one matching, can be generated efficiently.

The paper is organized as follows: Section 2 introduces the fundamentals of XML query and related work for XML query evaluation. Section 3 discusses in detail the fast tree pattern match algorithm *TreeMatch*. Section 4 is the conclusion.

2 The XML Query Fundamentals

We present the background information of the XML query and notations used in this paper. An XML document consists of nested elements enclosed by user-defined tags, which indicate the meaning of the content contained. Figure 1 shows an example of an XML document named “pub.xml”, which contains some publication information. The hierarchical structure of an XML documents can be modelled as a tree. Figure 3 is the tree representation of the XML file in Figure 1. The XML documents on the Internet is an forest of XML trees and we call it an XML database.

```
<?xml version="1.0" ?>
<publication>
   <journal title="DBMS">
      <editor>Jack</editor>
      <article>
         <title>
            Index Construction
         </title>
         <author>Smith</author>
      </article>
  </journal>
  <journal title="Algorithm">
  </journal>
</publication>
```

Figure 1. An example of an XML document

The semi-structured format of XML documents brings the possibility of using database technology to query the XML data instead of information retrieval techniques applicable only to plain text documents. However, the mature SQL queries can not be applied directly since XML documents do not necessarily conform to a predefined, rigid schema required by the traditional database system [17].

Much research has been done on XML query languages. Although the query languages differ in detailed grammars, they share a common feature, that is: querying structure as well as the contents or values of elements. Queries in XML query languages make use of tree patterns to match portions of data in the XML database. For example, the following is a query expressed in Xquery [13] over the document in Figure 1 where “//” indicates ancestor-descendant relationship, and “/” indicates parent-child relationship.

```
FOR $a IN document(“http://.../pub.xml”)//journal/article
    $b IN $a/title
WHERE $a/author =“Smith” RETURN
    <article>$b </article>
```

This query retrieves the titles of articles authored by “Smith” and published in a journal. It contains both structure and content information. We can use a tree to depict the query as shown in Figure 2. In other words, this query will find all the matchings of the tree pattern in the XML database.

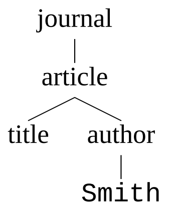

Figure 2. The tree representation of the example query

The existing methods for tree pattern matching in XML is typically a decomposition-matching-merging process: 1) decompose the tree pattern into linear patterns which might be binary (parent-child or ancestor-descendant) relationships between pairs of nodes or root-to-leaf paths; 2) find all matchings of each linear pattern; and 3) merge-join them to produce the result.

Most research in literature focuses on the second subproblem: find all matchings of a linear pattern. It can be classified according to the type of linear pattern they deal with, i.e., matching the binary structural relationships and matching path patterns. To match the binary structural relationships, Zhang *et al.* [16] proposed the MPMGJN (multi-predicate Merge Join) algorithm and Al-Khalifa *et al.* [9] gave the Stack-Tree algorithms. The algorithms accept two lists of sorted individual matching nodes and structurally join pairs of nodes from both lists to produce the matchings of the binary relationships.

The difference between the MPMGJN and Stack-Tree is that the MPMGJN is a variation of the traditional merge-join algorithm, requiring multiple scans of the input lists. The Stack-Tree algorithm is more efficient as it uses stacks to maintain the ancestor or parent nodes and it needs only one scan of the input lists. Li *et al.* [10] and Chien *et al.* [5] use an index to facilitate the structural join process and do not require sorted input lists.

Recently, Bruno *et al.* [2] proposed algorithms called PathStack and TwigStack. The former is for matching path patterns and the latter is claimed to solve the problem of twig pattern matching. Both of them use a chain of stacks to encode the partial result. However, TwigStack does not match the twig pattern directly. It still belongs to the decomposition-matching-merging category.

All the algorithms discussed above use the format

$$(\texttt{DocId}, \texttt{Start} : \texttt{End}, \texttt{Level})$$

to represent the nodes in the database. `DocId` is the identity of the document the node belongs to, $\texttt{Start} : \texttt{End}$ are the start and end positions of the corresponding element in the document and `Level` is the depth of the node in the tree hierarchy (e.g., for the root node, $\texttt{Level} = 0$).

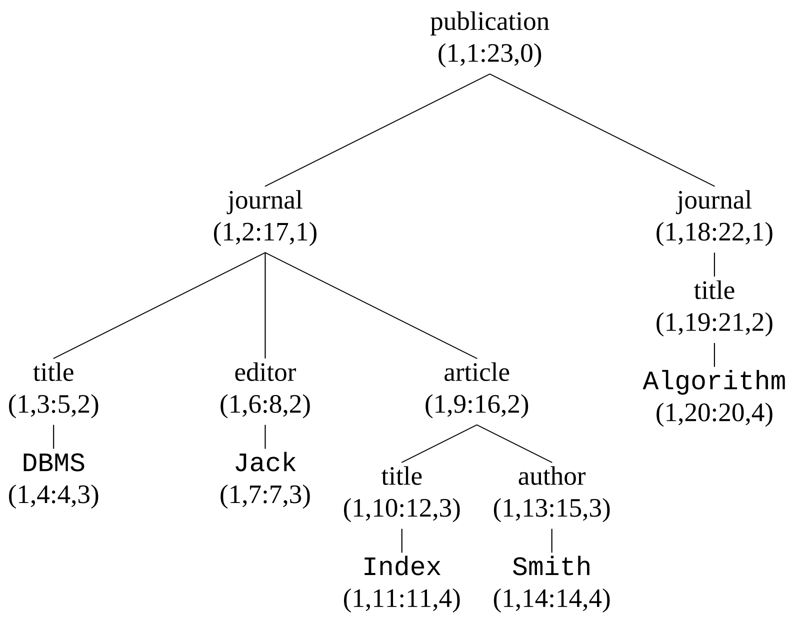

Figure 3. The tree representation of the XML document example

The advantage of using $(\texttt{DocId}, \texttt{Start} : \texttt{End}, \texttt{Level})$ to represent nodes is that we can determine the relationships between nodes in a constant time. A simple definition of the terms ancestor, descendant, parent and child is given as follows,

Definition 1 *Suppose that* x *and* y *are two nodes from an XML tree, we say that* y *is an ancestor of* x *and* x *is a descendant of* y *if* $\texttt{y.DocId} = \texttt{x.DocId}$ *and* $\texttt{y.Start} < \texttt{x.Start} < \texttt{y.End}$*.* y *is an parent of* x *and* x *is a child of* y *if 1)* $\texttt{y.DocId} = \texttt{x.DocId}$*, 2)* $\texttt{y.Start} < \texttt{x.Start} < \texttt{y.End}$*, and 3)* $\texttt{y.Level} = \texttt{x.Level} - 1$.

For example, in Figure 3, the author node $(1, 13 : 15, 3)$ is a descendant of the journal node $(1, 2 : 17, 1)$.

The TwigStack algorithm [2] partially solved the problem of larger intermediate results with decomposition-matching-merging methods. When the patterns have only ancestor-descendant edges, the intermediate result of each path matching is guaranteed to be part of the final result. However, TwigStack's requirement of matching all the root-to-leaf paths leads to repeated matching of the common nodes shared by multiple paths. If the query twig pattern has N leaf nodes, there will be N different root-to-leaf paths. The matching of common nodes would be computed up to N times. For example, in Figure 2, pattern Q will be decomposed to two root-to-leaf paths, "journal/article/title" and "journal/article/author/Smith". Sub-Path "journal/article" are shared by two paths and will be matched repeatedly.

The difficulty of directly matching tree patterns comes from the self-containment property of the XML elements, that is, elements have the same tag with their sub-elements. However, self-containment is seldom found in real XML documents. Moreover, the self-containment property is easily identifiable. For an XML document with DTD(Document Type Definition), this property is indicated by the DTD. For an XML document without DTD, it is easily identified during the index construction process.

Other works for XML queries focus on the preprocessing of query patterns before the matching against the XML data source is executed. Amer-Yahia *et al* [14] proposed a tree pattern minimization technique which aims at finding the smallest equivalent tree pattern of the original pattern by efficiently identifying and eliminating redundant nodes in the pattern. Flesca *et al* [7] took one step forward by considering the minimization for general case tree pattern with wildcard operators. Chen *et al* [3] proposed the concept of GTP(*generalized tree pattern*) and presented an algorithm to translate a general XQuery query expression, which consists of more than one tree pattern and possibly involves quantifiers, aggregation and nesting, into a GTP. Evaluating the query expression reduces to finding distinct matches of the GTP.

3. The TreeMatch Algorithm

3.1 Problem Definition

We first introduce some basic notations before giving the formal definition of the problem. An XML document is modelled as a tree $D = (N_d, E_d)$, where N_d is a set of

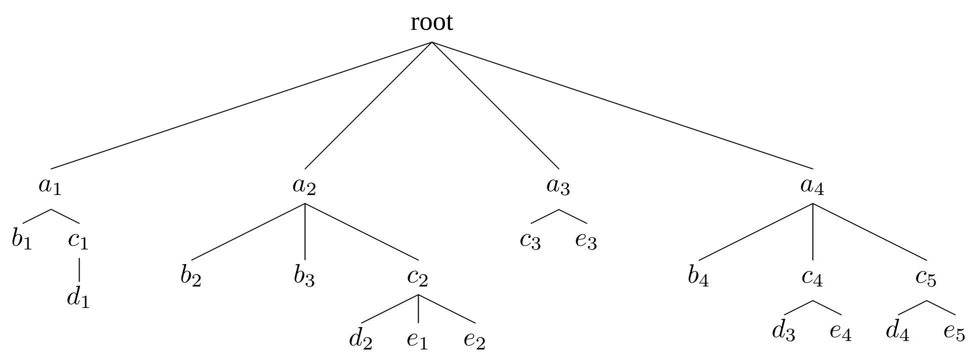

Figure 4. The data source

labelled nodes, E_d is a set of edges. We call D a data source. For a node $x \in N_d$, the label of x is denoted by $label_D(x)$.

A query tree pattern is a tree $Q = (N_q, E_q)$, where N_q is a set of labelled nodes and E_q is a set of edges. Each edge is represented by the pair of nodes it connects. There are two kind of edges in E_q, parent-child edges and ancestor-descendant edges. An ancestor-descendant edge is represented by $x//y$ and a parent-child edge is represented by x/y, where $x, y \in N_q$ are the nodes connected by the edge. For a node $x \in N_q$, $label_Q(x)$ denotes the label of x.

Definition 2 *Given two nodes $x \in N_q$ and $x' \in N_d$, we say that x' is an* occurrence *of x in D if $label_Q(x)$=$label_D(x')$.*

Definition 3 *Given a query tree pattern $Q = (N_q, E_q)$ and a data source $D = (N_d, E_d)$, for any node $q \in N_q$, if there exist two* occurrences *of q that have ancestor-descendant or parent-child relationship, q is said to have occurrence with self-containment.*

Definition 4 *Given a query tree pattern $Q = (N_q, E_q)$ and a data source $D = (N_d, E_d)$, we call $\psi(N_q)$ a matching of Q in D if there is a mapping $\psi : N_q \longrightarrow N_d$ that satisfies the following properties for every $x, y \in N_q$:*

1. *$x \neq y \Longleftrightarrow \psi(x) \neq \psi(y)$;*
2. *$label_Q(x)$=$label_D(\psi(x))$;*
3. *if $x//y \in E_q \Longrightarrow \psi(x)$ is an ancestor of $\psi(y)$; if $x/y \in E_q \Longrightarrow \psi(x)$is the parent of $\psi(y)$;*

The TreeMatch algorithm works in the condition of no occurrence of self-containment. In other words, the problem is defined as: given a tree pattern Q where each non-leaf node does not have occurrence with self-containment, and a XML data source D that has index structures to identify occurrences of Q's nodes, find all the distinct matchings of Q in D.

3.2 The TreeMatch Algorithm and Its Notions

The TreeMatch algorithm, presented in Section 3.3, can find all distinct matchings of a tree pattern in the data source directly. It is no longer a decomposition-matching-merging process. The basic idea is as follows: given a tree pattern, *TreeMatch* finds all the matchings of the pattern by recursively calling function *find(q)* to find the matchings of sub-pattern rooted by q and compactly encodes the matchings in the stacks associated with each pattern node. At any time the stacks associated with pattern node q and its descendants contain the matching nodes of sub-pattern rooted by q that are possibly extended to the final result.

We first introduce the notations used in the TreeMatch. In the algorithm, each pattern node q is associated with a stack S_q and a list T_q. S_q is used to store the potential solution. T_q is a list of sorted occurrences of q. Here we assume the data source is a single XML document so each node of T_q can be represented by a triplet $(\texttt{Start}, \texttt{End}, \texttt{Level})$. The nodes in T_q are sorted by `Start`. The algorithm can easily be extended to deal with multiple XML documents by adding an extra attribute `DocId` (document identity) to the node representation triplet and testing the equality of `DocId` before manipulating the nodes. Since the occurrence T_q lists can be obtained by accessing to an index structure [10], TreeMatch accepts them as input.

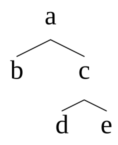

Figure 5. The pattern tree

In the algorithm, $T_q \rightarrow current$ points to the current node and *Advance(T_q)* moves to the next node in T_q. *Isleaf(q)* checks if pattern node q is a leaf node. *NumOfChildren(q)* returns the number of children of pattern node q and q_i ($i = 0, 1, \ldots NumOfChildren(q) - 1$) are children of

q. For example, when $q = a$ as shown in Figure 5, *NumOfChildren(q)* returns 2 and $q0 = b, q1 = c$. T_{q_0} and T_{q_1} are lists of matching nodes of b and c, respectively. In particular, given the data source shown in Figure 4, $T_b = \{b_1, b_2, b_3, b_4\}$ and $T_c = \{c_1, c_2, c_3, c_4, c_5\}$.

The No-Self-Containment property of the non-leaf node in the pattern brings about the possibility of compact encoding of the final matching results. Given a tree pattern with such a property, for any non-leaf pattern node q, any pair of nodes in its occurrence list T_q does not have common descendants. Therefore, we can have a partition of all matching nodes of the sub-tree pattern rooted by q with each partition containing the nodes of matchings rooted by one occurrence of q. For example, given the tree pattern of Figure 5 against data source of Figure 4, any node in $T_b = \{b_1, b_2, b_3, b_4\}$ has at most one ancestor or parent in $T_a = \{a_1, a_2, a_3, a_4\}$. Thus, any pair of nodes in T_a do not have common descendants in T_b. We can partition the nodes of all matchings, with each node belonging to exactly one partition. Each partition contains the nodes of the matchings rooted by one occurrence of a. In this example, matching nodes can be partitioned into two parts: $\{\{a_2, b_2, b_3, c_2, d_2, e_1, e_2\}, \{a_4, b_4, c_4, d_3, e_4, c_5, d_4, e_5\}\}$. Within each partition, the matching nodes of sub-patterns can be further partitioned. This operation can be recursively performed down to the leaf pattern node. Finally, the matching nodes looks as follows:$\{\{a_2, \{b_2, b_3\}, \{c_2, d_2, \{e_1, e_2\}\}\}$, $\{a_4, b_4, \{\{c_4, d_3, e_4\}, \{c_5, d_4, e_5\}\}\}\}$.

Since the nodes of the matchings can be partitioned, we can compactly encode the matching nodes and generate solutions partition by partition.

3.3 Algorithm

All the nodes of T_q are occurrences of the pattern node q in data source. Function *Find(q)* is called to determine whether the current occurrence T_q→current is a partial solution. T_q→current is a partial solution means matchings of sub-tree patterns rooted by q have been found and encoded in the stacks and these matchings are possibly extended to final results. If T_q→current is found not to be a partial solution, function *CleanStack()* is called to remove the recoded nodes that are descendants of T_q→current. Function GenerateSolution() and GenerateSolution2() produce two varieties of explicit representation of the final result.

```
TreeMatch($q$)
  While ($T_q$ is not empty)
      if(Find($q$)) push($S_q$, $T_q$→current, root);
        Advance($T_q$);
  GenerateSolution($q$);

Find($q$)
(1)  if (Isleaf($q$)) return true; /* $q$ is leaf node*/
(2)  $N$=NumofChildren($q$);
(3)  $i$=0; PartialSolution=false;
(4)  While(PartialSolution OR $T_{q_i}$ is not empty)
     /*$q_i (i = 0, 1, \ldots, N-1)$ are $q$'s children */
(5)      if ($T_{q_i}$ is empty OR
             $T_{q_i}$→current.start $>$ $T_q$→current.end)
(6)          if (PartialSolution)
(7)            $i = i + 1$;
(8)            PartialSolution=false;
             else
(9)            $j = 0$;
(10)           while ( $j$++ $< i$) CleanStack($q_j$);
(11)           return false;
(12)         if ($i = N$) return true;
         else
(13)     if ($T_{q_i}$→current.start $<$ $T_q$→current.start)
(14)         Advance($T_{q_i}$);
         else
(15)         if (Find($q_i$))
(16)           push($S_{q_i}$, $T_{q_i}$→current, $T_q$→current);
(17)           PartialSolution=true;
(18)         Advance($T_{q_i}$);
(19) return false;
```

Partial solutions can be recursively defined as follows:

Definition 5 *The occurrences of leaf pattern nodes are partial solutions. An occurrence x_i of a non-leaf pattern node x is a partial solution if 1) x_i has child nodes that are the occurrences of each child node of x; and 2) these child nodes of x_i are partial solutions.*

For example, given a tree pattern as shown in Figure 5 and data source in Figure 4, b_1, b_2, b_3, b_4 are all partial solutions since b is a leaf pattern node. c_2 is also a partial solution since its children d_2, e_1 match the children of node c, and d_2, e_1 are partial solution. In comparison, c_1 is not a partial solution and a_1 is not a partial solution.

According to above definition, a matching node is a partial solution if its matching child nodes are partial solutions. *Find(q)* finds T_q→current's child nodes in T_{q_i} (q_i is child node of q in the pattern) and recursively calls *Find(q_i)* to determine whether they are partial solutions. If a matching child node in T_{q_i} is found to be a partial solution, it is pushed into stack S_{q_i} (line 15-16). When T_q→current has child nodes matching each of q's child nodes, *Find(q)* returns true (line 12).

$(\texttt{Start}, \texttt{End}, \texttt{Level})$ representation of node is used to determine the relationship between nodes in T_q and T_{q_i}. The algorithm checks only the ancestor/descendant relationships. To deal with the parent/child relationship, we only need to test the Level of nodes by replacing line (15) with "if (T_{q_i}→current.Level= T_q→current.Level+1 AND Find(q_i))".

If $T_{q_i}\rightarrow$current.start is greater than $T_q\rightarrow$current.end (line 5), node $T_{q_i}\rightarrow$current and its followers in T_{q_i} are not descendants of $T_q\rightarrow$current. At this point, the algorithm has to check the present status before taking further action. If there is a node before $T_{q_i}\rightarrow$current that is a partial solution, the algorithm starts to check $T_{q_{i+1}}$ (line6-8). Otherwise, the algorithm cleans the stack S_{q_1}, S_{q_2}, …, $S_{q_{i-1}}$ by calling function *CleanStack* since the nodes in the stacks that are descendants of $T_q\rightarrow$current cannot be extended to produce a final result.

If $T_{q_i}\rightarrow$current.start is less than $T_q\rightarrow$current.start, which means node $T_{q_i}\rightarrow$current is not a descendant of $T_q\rightarrow$current but the nodes following $T_{q_i}\rightarrow$current might be, the algorithm simply moves to check the next node in T_{q_i} (line13-14).

If $T_{q_i}\rightarrow$current is a descendant of $T_q\rightarrow$current, *Find(q_i)* is called recursively to determine whether $T_{q_i}\rightarrow$current is a partial solution. If $T_{q_i}\rightarrow$current is a partial solution, it is pushed into the stack S_{q_i} together with $T_q\rightarrow$current (line15-17).

```
CleanStack($q$)
    $N$=NumOfChildren($q$);
    $i = 0$;
    While ($i$++ $< N$) CleanStack($q_i$);
    ParentNode=$S_q\rightarrow$top.parent;
    While($S_q$ is not empty AND $S_q\rightarrow$top.parent=ParentNode)
        Popup($S_q$);
```

```
GenerateSolution($q$)
(1)  $N$=NumOfChildren($q$);
(2)  $i = 0$;
(3)  while($i$++ $< N$)
(4)      GenerateSolution($q_i$);
(5)      $S_q$=Join($S_q$, $S_{q_i}$);
```

```
GenerateSolution2($q$, parent)
    while($S_q\rightarrow$bottom.Parent=parent)
        AppendtoSolution($S_q\rightarrow$bottom.Self);
        $N$=NumOfChildren($q$);
        $i = 0$;
        while ($i$++ $< N$)
            GenerateSolution2 ($q_i$, $S_q\rightarrow$bottom.Self);
        $S_q\rightarrow$bottom=$S_q\rightarrow$bottom$\rightarrow$up;
        If ($S_q\rightarrow$bottom.Parent=parent)
            AppendtoSolution(-1);
    AppendtoSolution(-1);
```

The final results are compactly encoded in the stacks associated with each pattern node. For any pattern node q, stack S_q stores the occurrences of q that are part of the matchings. Each element in stack S_q consists of a pair: (a node q' in T_q, the parent of node q' in $T_{parent(q)}$). When matching the pattern of Figure 5 in the data source of Figure 4, the encoded final result is shown in Figure 6.

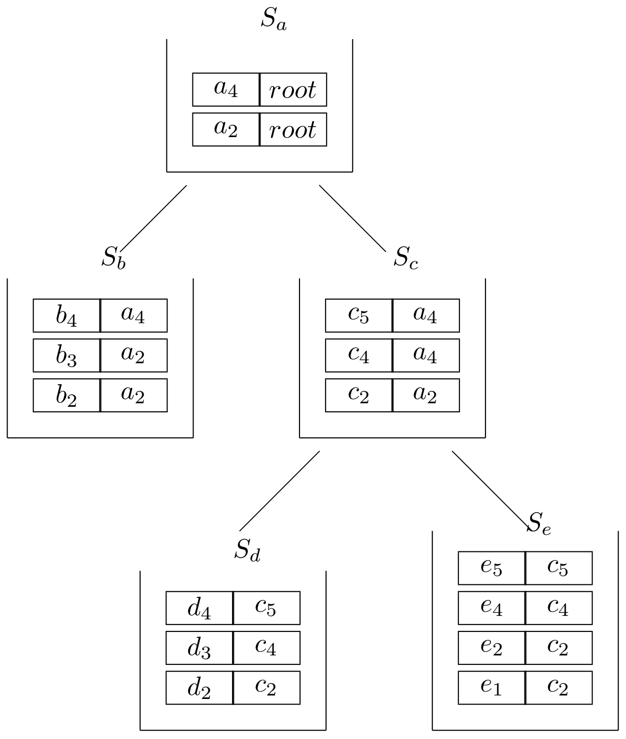

Figure 6. Encoded Result

The *GenerateSolution()* function is called to produce explicit representation of the final result from the encoded one. It generates the distinct matchings one by one. The `Join(`S_q`,` S_{q_i}`)`(line5) is an operation equivalent to the equal join in relational databases, the content of S_q and S_{q_i} being regarded as relations. The result of `Join(`S_q`,` S_{q_i}`)` is stored in S_q. This operation is performed from bottom to top of the pattern tree and the final result is stored in the stack associated with the root pattern node. Figure 7 shows the explicit representation of the encoded result in Figure 6.

The *GenerateSolution2()* function gives a concise representation of the final result. It organizes the matchings as a tree, and produces the string representation of the tree as illustrated in Figure 8. The string representation can uniquely determine the tree [15]. In other words, the tree can be uniquely recovered by scanning the string representation once. Specifically, it is recovered in a way of pre-order-traversal: at first we only have a virtual root node, while scanning the string, we add a child node to the current node when an element is read and we backtrack to the parent node when -1 is read.

4 Conclusion

The paper proposes a TreeMatch algorithm to directly find all distinct matchings of a query tree pattern in XML data sources. Unlike prior research for query tree pattern

S_a

a_4,b_4,c_5,d_4,e_5	*root*
a_4,b_4,c_4,d_3,e_4	*root*
a_2,b_3,c_2,d_2,e_2	*root*
a_2,b_3,c_2,d_2,e_1	*root*
a_2,b_2,c_2,d_2,e_2	*root*
a_2,b_2,c_2,d_2,e_1	*root*

Figure 7. Result of GenerateSolution()

$$a_2, b_2, -1, b_3, -1, c_2, d_2, -1, e_1, -1, e_2, -1, -1, -1, a_4,$$
$$b_4, -1, c_4, d_3, -1, e_4, -1, -1, c_5, d_4, -1, e_5, -1, -1, -1$$

Figure 8. Result of GenerateSolution2()

matching, the TreeMatch algorithm does not need to decompose the tree patten into linear patterns and do not produce any intermediate results that are not part of the final results. The TreeMatch algorithm is applicable when the non-leaf pattern nodes do not have occurrences with self-containment. The self-containment is seldom found in real XML documents and such a property can easily be identified. Therefore, the TreeMatch algorithm is more efficient than the existing methods under most cases.

Acknoledgements

Financial support through a grant of Natural Sciences and Engineering Research Council (NSERC) of Canada is gratefully acknowledged.

References

[1] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J.L. Wiener. The Lorel query language for semistructured data. International Journal on Digital Libraries, 1(1):68-88, April 1997.

[2] N. Bruno, D. Srivastava, and N. Koudas. Holistic twig joins: Optimal XML pattern matching. In SIGMOD, pp310-321, 2002.

[3] Z.M. Chen, H.V. Jagadish, L.V.S. Lakshmanan, and S. Paparizos, From Tree Patterns to Generalized Tree Patterns: On Efficient Evaluation of XQuery. Proceedings of VLDB Conference, Berlin, Germany, pp237-248, September 2003.

[4] D. Chamberlin, J. Robie, and D. Florescu. Quilt: An XML query language for heterogeneous data sources. In proceedings of International Workshop on the Web and Database, pp53-62, May 2000.

[5] S.Y. Chien, Z. Vagena, D. Zhang, V.J. Tsotras, C. Zaniolo. Efficient Structural Joins on Indexed XML Documents. Proceedings of the 28th VLDB Conference, Hong Kong, China, pp263-274, 2002.

[6] A. Deutsch, M. Fernandez, D. Florescu, A. Levy and D. Suciu. A Query Language for XML. In proceedings of 8th International World Wide Web Conference, pp77-91, May 1999.

[7] S. Flesca, F. Furfaro and E. Masciari. On the minimization of Xpath queries. Proceedings of the 29th VLDB Conference, Germany, pp153-164, 2003.

[8] H. Jiang, W. Wang, H.J. Lu and J.X. Yu. Holistic Twig Joins on Indexed XML documents. Proceedings of the 29th VLDB Conference, Berlin, Germany, pp273-284, 2003.

[9] S. Al-Khalifa, H.V. Jagadish, N. Koudas, J.M. Patel, D. Srivastava, Y. Wu. Structural Joins: A Primitive for Efficient XML Query Pattern Matching. ICDE 2002:141-152.

[10] Q. Li and B. Moon. Indexing and querying XML data for regular path expressions. Proceedings of the 27th VLDB Conference, pp361-370, 2001.

[11] J. Shanmugasundaram, K. Tufte, G. He, C. Zhang, D. DeWitt, and J. Naughton. Relational Databases for Querying XML Documents: Limitations and Opportunities. Proceedings of VLDB Conference, Edinburgh, Scotland, pp302-314, 1999.

[12] A. Schmidt, M. Kersten, M. Windhouwer, F. Waas. Efficient Relational Storage and Retrieval of XML Documents. In proceedings of International Workshop on the Web and Databases, pp47-52, 2000.

[13] W3C Recommendation. XQuery 1.0: An XML Query Language. Available at http://www.w3.org/TR/xquery.

[14] Sihem Amer-Yahia, SungRan Cho, Laks V. S. Lakshmanan, Divesh Srivastava: Tree pattern query minimization. VLDB Journal 11(4): 315-331, 2002.

[15] M.J. Zaki. Efficiently Mining Frequent Trees in a Forest. 8th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp71-80, 2002.

[16] C. Zhang, J.F. Naughton, D.J. DeWitt, Q. Luo and G.M. Lohman. On supporting containment queries in relational database management system. In proceedings of ACM SIGMOD, pp425-436, 2001.

[17] M. Zhang, J.T. Yao. The XML Algebra for Data Mining. In the proceedings of Data Mining and Knowledge Discovery: Theory, Tools, and Technology, Orlando, USA, pp209-217, April 2004.