Paper: jc10-5-2586: 2006/5/19

Minimization of XML Tree Pattern Queries in the Presence of Integrity Constraints

Yangjun Chen*, **and Dunren Che****

*Department of Applied Computer Science, University of Winnipeg
515 Portage Avenue, Winnipeg, Manitoba R3B 2E9, Canada
E-mail: ychen2@uwinnipeg.ca
**Department of Computer Science, Southern Illinois University
Carbondale, IL 62901, USA
E-mail: dche@cs.siu.edu

[Received October 28, 2005; accepted March 17, 2006]

In this paper, we provide a polynomial-time tree pattern query minimization algorithm whose efficiency stems from two key observations: (i) Inherent redundant "components" usually exist inside the rudimentary query provided by the user. (ii) Irredundant nodes may become redundant when constraints such as co-occurrence and required child/descendant are given. We show the result that the algorithm obtained by first augmenting the input tree pattern using the constraints, and then applying minimization, always finds the unique minimal equivalent to the original query. We complement our analytical results with an experimental study that shows the effectiveness of our tree pattern minimization techniques.

Keywords: XML documents, tree pattern queries, Xpath expressions, query evaluation, integrity constraints

1. Introduction

1.1. Processing of XML Query

XML (eXtensible Markup Language) has emerged as a standard for information exchange between Web applications. It offers a convenient syntax for representing data from heterogeneous sources. Various optimization strategies for XML queries are proposed, but most of them were following the same routine by transforming a query into a logical level plan, and then explore the (exponential) space of possible plans looking for the one with least estimated cost [1]. Path traversals (i.e., navigating subelement and reference links) always play a central role in query processing and a number of factors associated with XML data complicate the problem as well.

1.2. XML Constraints Under DTD

Constraints in XML are a particular type of containment constraints, which are important for semantic simplification of XML query evaluation. They are useful for query optimization, update anomaly prevention, and for information preservation in data integration [2]. XML constraints are traditionally part of the schema specification - DTDs (Document Type Definitions), which offer the so-called ID and IDREF attributes to identify and reference an element within an XML document. Just like any database, tree databases naturally come with application dependent constraints. For tree databases, constraints that require entries/elements to have child/descendant or subelements of specified types, as well as constraints that require type co-occurrences, are very natural [3]. For example, consider the query "*find the title and author of books that have a publisher*". If the constraint "every book has a publisher" is known to hold, then this query can be simplified to "*find the title and author of books*". Query minimization under constraints is traditionally achieved using semantic query optimization techniques. Existing techniques for semantic query optimization usually base on the notion of rewriting that transforms a query into an equivalent one [4]. Unfortunately, given a set of XML constraints, there are exponentially many ways in which a query can be rewritten. Therefore, an intuitive approach of cloning semantic query optimization is inappropriate for tree pattern minimization. Thus, we explore a different way in the paper to efficiently minimize tree pattern queries in the presence of XML constraints.

1.3. Related Work

To query tree databases such as XML style directories, tree pattern queries form a natural basis. Rudimentary query entered by the user can be considerably improved by reducing the pattern size. Doing so is closely related to conjunctive query minimization: a problem that is in general NP-complete for classical relational database. Much research has been conducted on the minimization of relational conjunctive queries. In [3], Amer-Yahia et al. point out that tree pattern query is essentially a special kind of conjunctive queries on a tree-structured domain. In [5], Florescu et al. showed that containment of conjunctive queries with regular path expressions, over semi-structured data, is decidable; for some special cases, they showed the problem is NP-Complete [7]. Techniques like predicate elimination and join minimization are used. However, such kind of optimization is based on algebraic rewritings, which often generate exponential search spaces and results in problems that cannot be solved in polynomial time.

Minimization of Xpath queries under tree structure database is studied in [20]. In that paper, Flesca et al. address the problem of minimizing XPath queries for a limited fragments of XPath, containing only the child, the descendent, the branch and the wildcard operators. In their work, they proved the *global minimality* property: a minimum tree pattern equivalent to a given tree pattern *p* can be found among the sub-patterns of *p*, and thus can be obtained by pruning "redundant" branches from *p*. Based on such an observation, they designed an algorithm for tree pattern minimization which works, in general case, in time exponential w.r.t. the size of the input tree pattern. They also characterized the complexity of the minimization problem, showing that given a tree pattern *p* in an XPath fragment and a positive integer *k*, the problems of testing if minimize$(p) > k$ is NP-complete.

Discovering XML semantic constraints plays an important role in tree pattern query minimization and has recently received increasing attention by the research community. In particular, Lee et al. [21] showed a variety of semantic constraints hidden implicitly or explicitly in the DTD of XML database, and proposed two algorithms on discovering and rewriting the semantic constraints in relational database notation. Yu et al. studied the problem of constraint-based XML query rewriting for the purpose of data integration in [22]. Two novel algorithms, *basic query rewrite* and *query resolution*, have been designed to implement the semantic constraints. More concretely, the basic query rewriting algorithm reformulates input queries in terms of the source DTD based on containment mapping (no constraint considered). The query resolution algorithm generates additional rewritings by incorporating XML semantic constraints.

Query minimization in the presence of XML constraints has also been studied by several authors. Calvanese et al. studied the problem of conjunctive query containment in the presence of a special class of inclusion dependencies in [6], and established some decidability/indecidability results. In addition, Wood [8–12] studied a special class of XPath queries that he called *simple XPath queries* [13]. A simple Xpath query is a tree pattern query without descendant child (nodes), but with the added flexibility that allows a special label "-", which stands for any type of nodes. Wood showed that, in the absence of constrains, the minimal query that equivalent to a simple XPath query can be found in polynomial time. Miklau and Suciu [14] showed that the problem of minimizing tree pattern queries that contain both child and descendant nodes as well as nodes labeled "-" is a co-NP complete problem.

Finally, Amer-Yahia et al. studied the minimization issue of general tree pattern queries in [3], in which constraint dependent minimization was addressed. They considered two types of constraints derived from XML schema or DTD, namely, required child/descendant and the co-occurrence of sibling sub-elements. The algorithm proposed by them needs $O(n^6)$ time for this kind of tree pattern minimization.

In this paper, we present a method to minimize queries with *child*, *descendant*, *subtype* and *co-occurrence* constraints based on the algorithm described in our another paper appearing in this issue. The time complexity of this method is bounded by $O(n^3)$.

1.4. Contributions and Overview

In this work, we address the following problem and focus on designing efficient algorithms to solve it.

Problem: Given a tree query Q and a set of constraints C, find another one Q' that is equivalent to Q under C and is of the smallest size.

Concretely, the following contributions are delivered.

- We develop an efficient algorithm, called *Coverage*, based on the concept of *containment mappings*, to obtain the minimal equivalent query. The algorithm takes worst-case time $O(n^2)$, where n refers to the size of the input query. This algorithm is given in [23].
- When constraints are specified on required children, required descendants and required co-occurrence, as well as subtypes, we first augment a query with redundant nodes and edges according to the given constraints, and then apply the minimization algorithm Coverage to it. This strategy always produces the minimal equivalent query to the input. The time complexity of the algorithm is bounded by $O(n^3)$.
- To investigate the effectiveness and correctness of the proposed techniques, we implemented our algorithms using a simulated tree structure in Java, and then performed a series of tests, which demonstrates both the practicality as well as the suitability of our methods.

The remainder of this paper is organized as follows. In Section 2, we discuss a method to minimize a TPQ in the presence of integrity constrains, based on the algorithm given in [23]. In Section 3, we show the implementation details and experiment results. Section 4 is a short conclusion.

2. Tree Reduction in the Presence of Constraints

2.1. Constraints

Suppose we are given a query Q and a set of constraints C. Naturally we will have a question whether there is an equivalent query of the least size? We will show in this section that when only required child, descendant and co-occurrence, as well as subtype constraints are considered, via our minimization scheme, we can always achieve a unique minimal equivalent query Q'.

2.2. Augmentation

In our study, the following constraints are considered for the minimization of tree pattern queries.

(i) Co-occurrance: Types A and B always occur together as children of another type, denoted by $A \downarrow B$.

(ii) Subtype: Every document node of type A is also of type B, denoted by $A \leq B$. For example, in a document, there may exist some nodes labeled with the type "technician" while some other nodes with the type "employee". Obviously, we have "technician" $\leq$ "employee".

(iii) Required child: Every document node of type A has a child of type B, denoted by $A \rightarrow B$.

(vi) Required descendant: Every document node of type A has a descendant of type B, denoted by $A \Rightarrow B$.

Among these constraints, the required child, required descendants and subtype constraints are used for both augmentation and reduction. The co-occurrence constraints are used only for reduction purpose.

To minimize a tree pattern query, not only explicit, but also implicit constraints should be applied. For an implied constraint, we mean a constraint derived from some existing constraints. For instance, from $A \leq B$ and $B \leq C$, $A \leq C$ can be derived. For this purpose, we organize all subtype constraints into a DAG (*directed acyclic graph*) G_s, and all required child and required descendant constraints into another DAG $G_{c\text{-}d}$. G_s and $G_{c\text{-}d}$ are defined as follows:

$G_s = < V_s, E_s >$, where each $v \in V_s$ represents a type, and each $e = (v_1, v_2) \in E_s$ represents a subtype constraint $v_2 \leq v_1$;

$G_{c\text{-}d} = < V_{c\text{-}d}, E_{c\text{-}d} >$, where each $v \in V_{c\text{-}d}$ represents a type, and each $e = (v_1, v_2) \in E_{c\text{-}d}$ represents a required child constraint (indicated as a single arrow) or a required descendant constraint (indicated as a double arrow).

In the following, we consider only the case that both $G_{c\text{-}d}$ and G_s are forests, i.e., every node of some type has only the same type parent, and each type has at most one parent supertype. We will show that for such a kind of subtype constraints (with the presence of required child and descendant, as well as co-occurence constraints), an $O(n^3)$ time algorithm can be devised for minimizing tree pattern queries. (In the case that a type has more than one parent supertype, we will use Amer's method to merge G_s and $G_{c\text{-}d}$ together to generate a general DAG for all the subtype, required child, and required descendant constraints [3]. The augmentation can then be done in the same way as Amer's, which will produce a new tree pattern query of size $O(n^2)$. But we will use Algorithm *query-minimization*( ) to do minimization. Therefore, the whole cost is bounded by $O(n^4)$ in that case.)

When both $G_{c\text{-}d}$ and G_s are forests, we use the following algorithm to do the augmentation, which generate a new tree pattern query whose size is still $O(n)$. In the algorithm, for each type t, we use $h(t)$ to represent another type that is an ancestor of t and does not have a parent. In addition, a node with more than one children is called a branching node.

```
Algorithm augmentation(Q)
input: Q – a tree pattern query
output: Q′ – an augmented version of Q
begin
1. for each type $t \in Q \cup E_{c\text{-}d}$, replace $t$ with $h(t)$;
2. for any type $p \in Q$, if there exists another type $a$ such
   that $a \downarrow p$, remove $p$ and all its descendants from $Q$;
3. let $Q'$ be the query after step (2); call repeating-leaf-
   removing($Q'$);
4. let $Q''$ be the query after step (3); for each descendant
   edge $e = (v_1, v_2) \in Q''$, replace it with a path $p$ in $E_{c\text{-}d}$,
   which starts from $v_1$ and ends at $v_2$; denote the new
   tree by $Q'''$;
5. for each type $t \in Q'''$ do {
6.     find all those nodes $u_1, \ldots, u_k$ in $Q'''$, which are asso-
       ciated with $t$;
7.     call path-reduction($\{u_1, \ldots, u_k\}$, $Q'''$);
8. for each $u_i$ $(1 \leq i \leq k)$ do {if it removed, reinsert it into
   the query tree;}
   }
end
```

In the above algorithm, we first replace each type in $Q \cup E_{c\text{-}d}$ with its "delegate", which is the root of the subtype tree containing it (see line 1). The goal of this step is to assimilate super- and subtypes. For instance, if we have the subtype constraints subtype $C_1 = \{b \geq b', c \geq c'\}$ and the required child constraints $C_2 = \{b \rightarrow d, d \rightarrow g, g \rightarrow c', e \rightarrow m, m \rightarrow s\}$, the tree pattern query shown **Fig.1(a)** will be changed as shown in **Fig.1(b)**, and C_2 will be changed to $C_2' = \{b \rightarrow d, d \rightarrow g, g \rightarrow c, e \rightarrow m, m \rightarrow s\}$. Such a change can be reversed in the final minimized query by setting back the original types.

In a next step, we remove any type $\beta \in Q$, if there exists another type α such that $\alpha \downarrow \beta$ holds (see line 2). For instance, if a co-ocurrance constraint $C_3 = \{q \downarrow p\}$ is present, the query tree will be further transformed into a tree as shown in **Fig.1(c)**. However, such a reduction is just to facilitate the detection of coverage and any node removed due to the co-occurrence constraints will be reinserted into the final minimized query (see section 2.3). In the third step, we remove the leaf nodes repeatedly according to $G_{c\text{-}d}$ (see line 3), by calling Algorithm *repeating-leaf-removing*( ), by which any leaf node v will be removed if there exists a constraint $\lambda(u) \rightarrow \lambda(v)$ and (u, v) is a child edge, or there exists a constraint $\lambda(u) \Rightarrow \lambda(v)$ and (u, v) is a descendant edge. This process is repeated until the query cannot be changed any more. For instance, by removing the leaf nodes repeatedly, the query tree shown in **Fig.1(c)** will be reduced to the tree shown in **Fig.1(d)**.

The following is a formal description of the algorithm.

```
Algorithm repeating-leaf-removing(Q)
begin
```

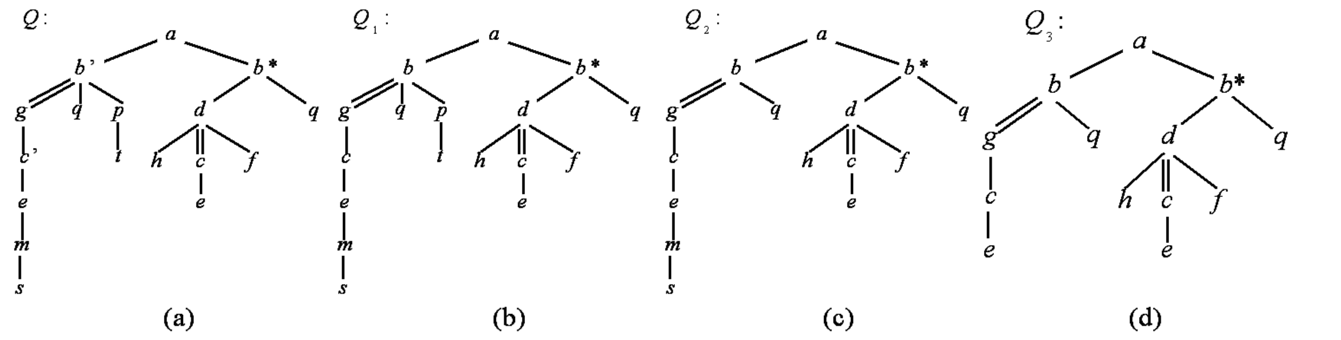


Fig. 1. Illustration for the execution of Algorithm *augmentation*( ).

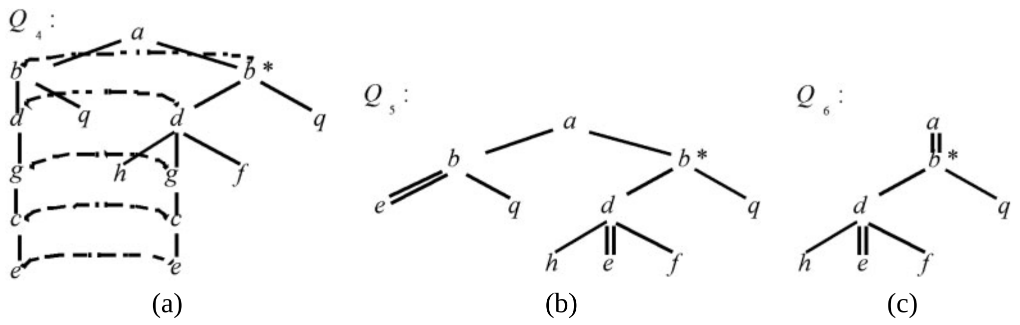


Fig. 2. Illustration for the execution of Algorithm.

```
    repeat until $Q$ cannot be changed
      let $v$ be a leaf node in $Q$; let $u$ be its parent;
      if $(u,v)$ is a child edge and $\lambda(u) \to \lambda(v)$ is in the
        constraint set, remove $v$;
      if $(u,v)$ is a descendant edge and $\lambda(u) \Rightarrow \lambda(v)$ is in
        the constraint set, remove $v$;
      let $Q'$ be the query tree obtained by executing the
        above operations;
      $Q := Q'$;
end
```

In the fourth step, the query will be augmented by replacing each descendant edge in Q with the corresponding path in $E_{c\text{-}d}$ (see lines 5-7 in Algorithm *augmentation*( )), which makes some paths that are originally different become identical. For instance, after this step, the query tree shown in **Fig.1(d)** will be augmented as shown in **Fig.2(a)**.

We notice that in the query tree shown in **Fig.1(b)**, the two paths: are different. After the path expansion, they become identical as illustrated by the dashed lines shown in **Fig.2(a)**. Then, the nodes on such paths can be removed to reduce to the paths as shown in **Fig.2(b)**. Such a strategy will not damage the subsequent detection of subtree coverage using *query-minimization*( ), nor does it change the semantics of the original query. By running *query-minimization*( ) against the query tree shown in **Fig.2(b)**, we will finally get a query tree as shown in **Fig.2(c)**.

The algorithm *path-reduction*( ) is described as follows.

```
Algorithm path-reduction$(U,Q')$
input: $U,Q'$ (*$U$ is a set of nodes in $Q'$.*)
output: a new query, which is obtained by removing some
nodes in $Q'$
begin
1. let $U = \{u_1,\dots,u_k\}$;
2. search all those paths starting from $u_i\,(1 \le i \le k)$
     bottom-up, in parallel, and at each step, do the following:
3.     let $a_1,\dots,a_k$ be the nodes encountered;
4.     if they are identical then {
         remove all of them except the branching nodes
         from the respective path;}
5.     else{divide $a_1,\dots,a_k$ into several groups $G_1,\dots,G_j$
6.          such that each of them has only identical nodes;
7.      for each $l(1 \le l \le j)$ do
8.          {for each $a \in G_l$, consider $u \in U$, which is on
             the same path as $a$;
9.              remove all the nodes between $a$ and $u$
                 (including $u$) on the corresponding path;
10.           remove all the branching nodes from $G_l$;
11.         if $G_l$ contains only one node $a$ then
12.         {let $v$ be the first branching node beyond $a$ on
              the corresponding path;
13.            remove all the nodes between $v$ and $a$ on
               the corresponding path;}
14.         else {if $G_l$ is not empty then path-reduction
                 $(G_l, Q')$;}}
          }
end
```

The goal of the above algorithm is to remove the nodes on the identical path segments. It works in a recursive

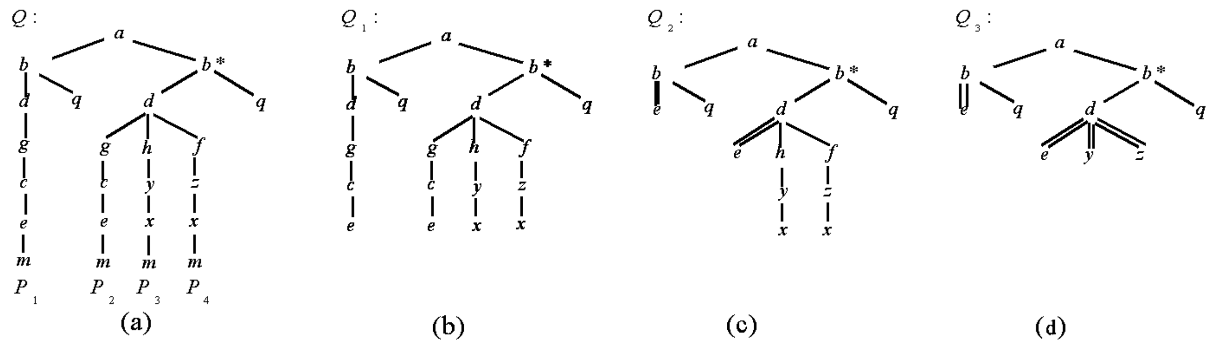


Fig. 4. Illustration for path reduction.

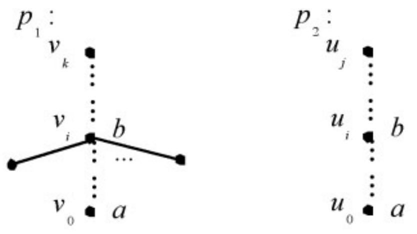


Fig. 3. Illustration for path reduction.

way. Let $u_1, \ldots, u_k$ be the nodes associated with the same type $t \in Q$. The algorithm will search the paths starting from all those nodes bottom-up in parallel. At each step, it will be checked whether all the nodes encountered along different paths are associated with the same type. If it the case, any of them will be removed from the corresponding path if it is not a branching node (See line 4). We illustrate this process in **Fig.3**, in which we show only two paths: p_1 and p_2 for a simple explanation.

On p_1, node v_i is a branching node while node u_i on p_2 is not. If we are searching only along these two paths, u_i will be removed while v_i remains. Such a strategy of removing nodes works based on the following observations.

1. The subtree rooted at u_i can not cover the subtree rooted at v_i since u_i's outdegree is exactly one while v_i's outdegree is larger than one.

2. If the subtree rooted at v_i covers the subtree rooted at u_i, we must have the subtree rooted at v_0 covers the subtree rooted at u_0. Even thought u_i is removed, the algorithm *query-minimization*( ) can eventually find this coverage.

During the searching of paths, if we meet a set of nodes whose types are not identical (called a *dividing set*), we will organize those nodes into a set of groups such that in each group, all the nodes have the same type (see line 5). For illustration, see the tree pattern query shown in **Fig.4(a)**. In this query, we have four paths starting from a node (bottom up) with the same type m: P_1, P_2, P_3, and P_4. The first dividing set met during the path searching is $\{P_1.e, P_2.e, P_3.x, P_4.x\}$, where $P_i.\alpha$ represents a node on path P_i with type α. It will then be divided into two groups: $G_1 = \{P_1.e, P_2.e\}$, and $G_2 = \{P_3.x, P_4.x\}$.

For each node a in a group G_l, we will remove all the nodes between a and $u \in U$ (including u), which is on the same path as a, for the reason that it is enough to keep only one different node on a path segment to differentiate it from any path in the other groups (see lines 8-9). After this step, the query tree is reduced to the tree shown in **Fig.4(b)**. Afterwards, any branching node will be removed from G_l since such a node will not take part in the path reduction any more (see line 10). Since no node in G_1 or G_2 is a branching node, no node is removed from each of them. Since each of them contains more than one nodes. We will have two recursive calls: *path-reduction*(G_1, Q_1) and *path-reduction*(G_2, Q_1) (see line 14). During the execution of *path-reduction*(G_1, Q_1), we will consecutively remove $P_1.c$ and $P_2.c$, $P_1.g$ and $P_2.g$, as well as $P_1.d$, getting a query tree shown in **Fig.4(c)**. We notice that $P_2.d$ is not eliminated because it is a branching node (see line 4). If a group contains only one node, the node must be different from any node on any other path (being considered). Thus, it can be used to differentiate the corresponding path from the others. For this reason, we will remove any node between it and the first branching node beyond it (see lines 11-13). For instance, during the execution of *path-reduction*(G_2, Q_1), we will meet the second dividing set: $\{P_3.y, P_4.z\}$. It will be divided into two groups: $G_3 = \{P_3.y\}$, and $G_4 = \{P_3.z\}$. In this case, we will remove $P_3.x$ along P_3, and remove $P_4.z$ along P_4 (see line 9). Furthermore, we will also remove $P_3.h$ and $P_4.f$. The resulting query tree is shown in **Fig.4(d)**. Finally, all the starting nodes of P_1, P_2, P_3, and P_4 will be reinserted to differentiate this set of paths from others. Therefore, we have the final query tree as shown in **Fig.5**.

Now we estimate the time complexity of the augmentation process. First, we see that O$(|Q'|)$ =O$(|Q|)$ since corresponding to each descendant edge, at most one new node is added. Thus, the time for running Algorithm *augmentation*( ) is bounded by O(n^3). It is because the size of the new query, created by replacing each descendant edge with a path in $G_{c\text{-}d}$, is on the order of O(n^2) and for each different type appearing in Q, it will be searched twice: one for finding all the nodes associated with that type, and one for path searching to remove useless nodes. So the total time is O(n^3) time.

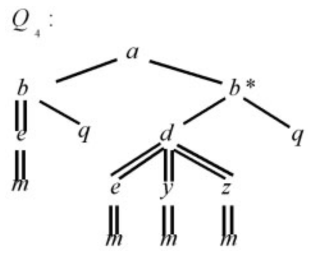


Fig. 5. Illustration for path reduction.

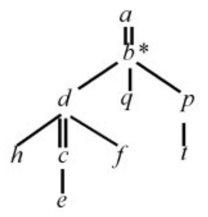


Fig. 6. A minimized query.

2.3. Minimization

In the minimization phase, we will run Algorithm *query-minimization*( ) against the query obtained in the augmentation phase.

Since the size of the query generated in the augmentation phase is equal to or smaller than the size of the original query, the time spent in the minimization phase is trivially bounded by $O(n^2)$.

Finally, any newly added nodes will be eliminated in the garbage phase and any node eliminated due to the co-occurrence constraints as well as path reduction can be reinserted. For instance, the query tree shown in **Fig.2(c)** can be rewritten as shown in **Fig.6**.

The garbage phase needs only $O(n)$ time.

3. Implementation and Experiment Results

We implemented the algorithms presented in this paper and experimentally compared their performance for minimization tree pattern queries. The experiments study in detail is presented by separating internal and external comparison.

- Internal Comparison: Compare the performance of Algorithm *query-minimization*( ) under variable query size and different number of constraints.
- External Comparison: Compare the performance of the proposed algorithms with Amer's *ACIM* and *CDM*, the algorithms designed for tree pattern minimization in [3].

3.1. Internal Comparison

As we discussed in both sections 3 and 4, the time spent on running *query-minimization*( ) depends on the input

Table 1. Average running time for *query-minimization*( ).

query size / constraints	10	20	30	40	50
0	0.102	0.180	0.502	0.822	1.182
10	0.120	0.282	0.560	0.882	1.262
20	0.140	0.340	0.640	1.024	1.442
30	0.150	0.386	0.732	1.148	1.582
40	0.160	0.460	0.826	1.282	1.764
50	0.180	0.524	0.934	1.422	1.924

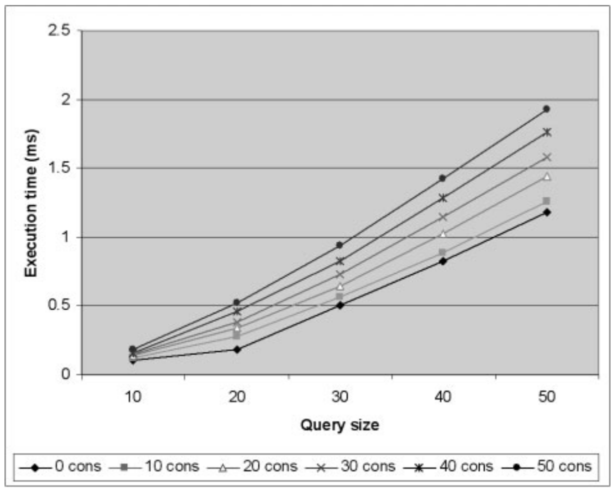
query size, the number of redundant nodes in the query and the number of constraints that might generate additional redundancy in the query. We report the total time on executing *query-minimization*( ) with a growing number of query size start from 10 and the growing number of constraints start from 0. The program runs under Windows XP Pro. system with JDK 1.5.0 Java compiler and the CPU is an Intel Pentium 4 processor of 1.4GHZ with 768MB RAM. In the experiment we find out that the running time of *query-minimization*( ) for queries with query size smaller than 40 can always be finished within 1 millisecond. By each execution, the input and constraint sets are the same. The result is then divided by 500 to reach an average run time for each algorithm execution, which is shown in **Table 1**.

The graph of **Fig.7(a)** and **(b)** shows the variation of running time of *query-minimization*( ) when query size and constraints are both growing.

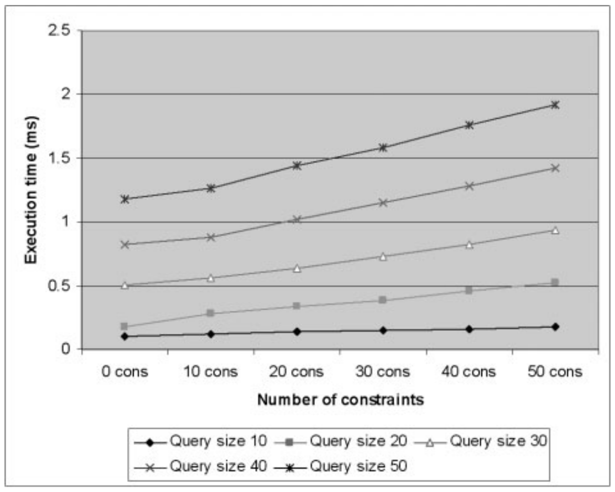
From the result shown in **Fig.7(a)**, we can see that when the size of a query is small (less than 20) the growing number of constraints does not affect the overall performance of *query-minimization*( ). This is because most of the constraints specified in the constraints set are irrelevant to the minimization. However, as the query size increases, the number of irrelevant constraints drops and more constraints have been adopted by the minimization. The more constraints we adopted, the more time we spend on constructing the augmented tree and removing temporary nodes in garbage collection phase. As we can see in section 4, when the query size is large, more time is needed on both the tree pattern augmentation and the tree minimization. **Fig.1** also confirms the above analysis, which shows that when the query size and constraints increase the time spent on minimization increases much faster than the increase of the query size.

3.2. Comparison with Related Work

The algorithms discussed in [3] by Amer et al. are the most similar work as what we presented in this paper. Naturally it becomes the best candidate to be compared with. ACIM and CDM are two algorithms they proposed for the purpose of tree pattern minimization. CDM is the improvement of ACIM under the presence of constraints. In general, CDM has a better performance than ACIM. However, without guarantee of reaching the minimal solution. According to their the analysis, the runtime for



(a)



(b)

Fig. 7. Performance vs. size and constraints.

Table 2. Time comparinson among ACIM, CDM and *query-minimizations*( ).

query size / Algo.	10	20	30	40	50
ACIM	5	9	15.6	20	24
CDM	3.5	5	7.5	9	10.5
query-mini.	0.84	1.974	3.92	6.174	8.834

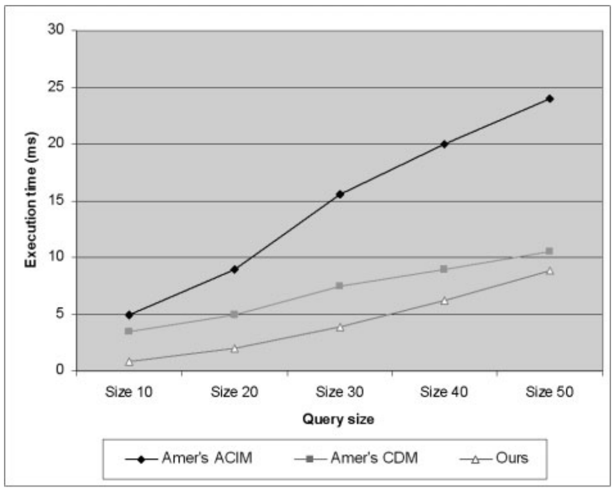


Fig. 8. Comparing ACIM, CMD and our algorithms.

both of the algorithms takes O(n^4) time in the absence of constraints, and O(n^6) in the presence of constraints. The experiment results showed that the time used by CDM is significantly smaller than ACIM with a growing query size and a fixed number of 40 constraints. But the experiments also showed that CDM, on average, removes only half of the redundant nodes that ACIM can in most of the cases. In our experiment, we perform a similar test as they did and record the total execution time for *query-minimization*( ). In addition, the different testing environment has been taken into account. Since the experiments carried out by Amer et al. is on a machine with a 300-500MHZ CPU and the RAM range from 64-128MB, 7 times slower than the machine we used, we test a data, which is 7 times larger than theirs, for each experiment, **Table 2** shows the overall performance of the three algorithms.

Figure 8 shows that *query-minimization*( ) uniformly outperforms ACIM when query size is smaller than 50, and the advantage increases with increasing query size. We do notice that as the query size increases, the time used by *query-minimization*( ) increases with a factor of 10, showing a linear behavior. This seems conflict with its O(n^3) time complexity. However, it is just an estimation in the worst-case, which happens very rarely, and does not reflect the overall performance of all the discussed algorithms. In fact, in our experiments, linear and quadratic time performance for all the three algorithms are very common. *query-minimization*( ) is also better than CDM when the query size is smaller than 40. But it has been caught up by CDM with the increasing of query sizes. Our explanation for this is that the data size used for testing *query-minimization*( ) is 7 times larger than that for testing CDM. When the data size is beyond a certain threshold, its impact on the computation shadows the difference between the two testing environments. On the other hand, since our algorithm is guaranteed of reaching the minimal state, which is not always the case for CDM, our algorithm is considered to be a better solution to the problem.

4. Conclusion

In this paper, we have presented an efficient algorithm for tree pattern query minimization in XML database systems. By adopting our algorithm, query discussed in the context of XML can be considerably improved by reducing the pattern size. We showed that for the tree pattern queries, the problem of reducing pattern size can be solved in polynomial time. More specifically, in the presence of constraints, the problem can be solved in O(n^3) time. We also show that the query achieved via our algorithm is the minimal equivalent to the given input tree pattern. In addition to providing the efficient algorithms for minimization with or without constraints, we also established their practicality by experiment studies.

Acknowledgements

We thank those graduate students at SIUC who contributed to this work by doing the reported experiments.

References:

[1] J. McHugh, and J. Widom, "Query Optimization for XML," Proceedings of the 25th VLDB Conference, Edinburgh, Scotland, 1999.

[2] W. Fan, and J. Simeon, "Integrity Constraints for XML," Proceedings of ACM PODS Conference, pp. 23-34, 2000.

[3] S. Amer-Yahia et al., "Minimization of Tree Pattern Queries," Proceedings of the ACM SIGMOD International Conference on Management of Data, pp. 497-508, 2001.

[4] S. Chakravathy, J. Grant, and J. Minker, "Foundations of Semantic Query Optimization for Deductive Databases," Foundations of DD and LP, 1988.

[5] D. Florescu, A. Levy, and D. Suciu, "Query Containment for Conjunctive Queries with Regular Expressions," Proceedings of the 17th ACM Symp., Principles of Database Systems, pp. 139-148, 1998.

[6] D. Calvanese, G. DeGiacomo, and M. Lenzerini, "Decidability of Query Containment under Constraints," Proceedings of 17th ACM Symp., Principles of Database Systems, pp. 149-158, 1998.

[7] M.R. Garey, and D.S. Johnson, "Computers and Intractability: A Guide to the Theory of NP-Completeness," W.H. Freeman & Co., NY, 1979.

[8] P.T. Wood, "Optimizing Web Queries Using Document Type Definitions," Proceedings of 2nd ACM CIKM International Workshop on We Information and Data Management, pp. 28-32, 1999.

[9] P.T. Wood, "On the Equivalence of XML Patterns," Proceedings of 1st International Conference on Computational Logic, Lecture Notes in Artificial Intelligence 1861, pp. 1152-1166, Springer-Verlag, NY, 2000.

[10] P.T. Wood, "Rewriting XQL Queries on XML Repositories," Proceedings of 17th British National Conf. on Databases, Lecture Notes in Computer Science 1832, pp. 209-226, Springer-Verlag, NY, 2000.

[11] P. T. Wood, "Minimizing Simple XPath Expressions," WebDB 2001.

[12] P.T. Wood, "Containment for XPath Fragments under DTD Constraints," Proceedings of 9th International Conference of Database Theory, pp. 300-314, 2003.

[13] World Wide Web Consortium. XML Path Language (XPath), W#C Recommendation, Version 1.0, November 1999. See http://www.w3.org/TR/xpath.

[14] G. Miklau, and D. Suciu, "Containment and Equivalence for an XPath Fragment," Proceedings of 21st ACM Symp., Principles of Database Systems, 2002.

[15] D. Chamberlin, J. Clark, D. Florescu, and M. Stefanescu, "XQuery1.0: An XML Query Language," http:// www.w3.org/TR/query-datamodel/.

[16] A. Deutch, M. Fernandex, D. Florescu, A. Levy, and D.Suciu, "A Query Language for XML," WWW'99.

[17] S. Al-Khalifa, H. V. Jagadish, N. Koudas, J. M. Patel, D. Srivastava, and Y. Wu, "Structural Joins: A Primitive for Efficient XML Query Pattern Matching," Proceedings of the 18th International Conference on Data Engineering, 2002.

[18] D. Che, and K. Aberer, "Query Processing and Optimization in XML Structured-Document Databases," In preparation for the VLDB Journal.

[19] Y.R.S. Yalamanchili, "Empirical Study of XML Query Optimization," SIUC thesis collection, Spring 2005.

[20] S. Flesca, F. Furfaro, and E. Masciari, "On the Minimization of Xpath Queries," Proceedings of the 29th VLDB Conference, Berlin, Germany, 2003

[21] D. Lee, and W.W. Chu, "Constraints-preserving Transformation from XML Document Type Definition to Relational Schema," Proceedings of the 19th International Conference on Conceptual Modeling, pp. 323-338, 2000.

[22] C. Yu, and L. Popa, "Constraint-Based XML Query Rewriting for Data Integration," Proceedings of the ACM SIGMOD International Conference, Paris, France, 2004.

[23] Y. Chen, and D. Che, "Efficient Processing of XML Tree Pattern Queries," Journal of Advanced Computational Intelligence and Intelligent Informatics, Vol.10, No.5, pp. -, 2006.

Name:
Yangjun Chen

Affiliation:
Associate Professor, Dept. Applied Computer Science, University of Winnipeg

Address:
515 Portage Ave. Winnipeg, Manitoba R3B 2E9, Canada

Brief Biographical History:

1982 Received B.S. degree in information system engineering from the Technical Institute of Changsha, China
1990 Received Diploma degree in computer science from the University of Kaiserslautern, Germany
1995 Received Ph.D. degree in computer science from the University of Kaiserslautern, Germany
1995-1997 Worked as a post-doctor at the Technical University of Chemnitz-Zwickau, Germany
1997-2000 Worked as a senior engineer at the German National Research Center of Information Technology (GMD), Germany
2000 Worked as a post-doctor at the University of Alberta, Canada
2000-present Professor in the Department of Applied Computer Science, the University of Winnipeg, Canada

Main Works:

• "On the Signature Tree Construction and Analysis," to appear in IEEE Transaction on Knowledge and Data Engineering.
• "Graph Traversal and Linear Binary-chain Programs," IEEE Transaction on Knowledge and Data Engineering, Vol.15, No.3, pp. 573-596, May/June, 2003.
• "Magic Sets and Stratified Databases," Int. Journal of Intelligent Systems, John Wiley & Sons, Ltd., Vol.12, No.3, pp. 203-231, March, 1997.

Name:
Dunren Che

Affiliation:
Assistant Professor, Department of Computer Science, Southern Illinois University Carbondale

Address:
Faner Hall 2125. SIU-Campus Carbondale, IL 62901, USA

Brief Biographical History:

1994 Received Ph.D. in Computer Science from the Beijing University of Aeronautics and Astronautics, Beijing, China
1994-2001 Gained postdoctoral research experience from various research Institutes, including, the Tsinghua University in China, the German National Research Center for Information Technology in Germany, and the Johns Hopkins University in the USA 2002-present Assistant Professor of Computer Science in the Southern Illinois University at Carbondale, USA

Main Works:

• "Query Optimization in XML Structured-Document Databases," The VLDB Journal, Vol.15, 2006.
• "Efficiently Processing XML Queries with Support for Negated Containments," International Journal of Computer & Information Science, Vol.6, No.2, pp. 109-120, June, 2005.