Constraint Satisfaction in Semi-structured Data Graphs

Nikos Mamoulis[1] and Kostas Stergiou[2]

[1] Department of Computer Science and Information Systems
University of Hong Kong
nikos@csis.hku.hk,
[2] Department of Information and Communication Systems Engineering
University of the Aegean
konsterg@aegean.gr

Abstract. XML data can be modeled as node-labeled graphs and XML queries can be expressed by structural relationships between labeled elements. XML query evaluation has been addressed using mainly database, and in some cases graph search, techniques. We propose an alternative method that models and solves such queries as constraint satisfaction problems (CSPs). We describe common constraint types occurring in XML queries and show how query evaluation can benefit from methods for preprocessing and solving CSPs. We identify an important non-binary constraint that is a common module of XML queries and describe a generalized arc consistency algorithm with low cost that can ensure polynomial query evaluation. Finally, we demonstrate that maintaining the consistency of such non-binary constraints can greatly accelerate search in intractable queries that include referential relationships.

1 Introduction

XML is becoming a standard for information exchange over the Internet. This flexible markup language allows both data content and structure to be described in a single document. XML is very appropriate for describing semi-structured data, which do not comply to a well-defined schema. XML documents can be viewed as rooted, node-labeled graphs, where the intermediate nodes take values from the set of potential element labels and the leaves store textual information. The nodes of XML graphs can be viewed as object instances, whose labels identifies their class. Having stored semi-structured data in a large XML document, we are often interested in the retrieval of object instances which satisfy some *structural constraints* between them. Such requests can be modeled as XML queries, expressed in a language like XPath [19].

Although the evaluation of simple structural queries has received a lot of attention from database research, little has been done to address the evaluation of complex queries where there is a large structure of objects to be retrieved or/and the structural constraints between the objects are complicated. The rapid increase in the use of XML in a wide variety of applications makes the need to address such problems a pressing one.

In this paper we propose the use of the constraint satisfaction paradigm as a new way of handling XML navigational (i.e. path) queries. We demonstrate that the expressiveness of constraint satisfaction allows us to capture in a natural way a wide variety

of such queries, from simple to very complex ones. Also, advanced constraint programming algorithms and heuristics allow us to handle complex problems that are otherwise difficult to deal with.

We begin by providing a mapping of all the common structural relationships used in XML navigational queries to a set of unary and binary constraints with well-defined semantics. Evaluation of queries with primitive structural constraints only (e.g. parent, child, sibling, etc.) has been recently shown to be in PTIME [7, 16, 12]. For example, consider the query "find a faculty member who has an RA and a TA", issued in an XML document containing structural information about a university. In other words, we are looking for twigs in an XML graph (e.g., see Figure 1), where the parent element is tagged by `faculty` and has two children elements tagged by `RA` and `TA`. In general, XML navigational queries are instances of the graph containment problem which has been shown to be polynomial for some classes of graphs, like trees. Interestingly, we show that such results can be also derived by constraint programming theory.

In addition, we identify an interesting conjunctive non-binary constraint that is commonly found in XML queries. This *all-different + ancestor-descendant* constraint (ADAD for short) relates a parent (or ancestor in general) with a number of children (or descendants in general) and, in addition, children (descendants) of the same label are related with an all-different constraint. For instance, consider the query "find a faculty member who has at least 3 RAs", which does not allow the instances of the RA variables to take the same value. Such queries fall in a class defined in [8] that can be evaluated in PTIME. We present an alternative efficient way to process XML queries that contain only ADAD and primitive structural constraints. To achieve this, we provide a filtering algorithm of low complexity for ADAD constraints. Finally, we show how queries that also include precedence (i.e., ordering) constraints can be evaluated in PTIME using constraint programming techniques.

Although many classes of XML queries are in PTIME, the general containment problem in XML graphs, where arbitrary *referential* (i.e., IDREF) constraints are included (e.g., see [11]) is intractable. We show how such intractable containment queries can be modeled as CSPs, and make an empirical comparison of various CSP search algorithms. The results show that maintaining specialized non-binary constraints, such as ADAD, can significantly increase pruning and speed-up search.

The rest of the paper is organized as follows. Section 2 provides background about XML query processing in database systems, and constraint satisfaction problems. In Section 3 we describe how simple and complex XML queries can be formulated as CSPs. In Section 4 we discuss the application of constraint programming techniques to preprocess and solve this class of problems. We also elaborate on the complexity of query evaluation for different classes of queries and include an experimental evaluation of search algorithms on intractable XML queries when formulated as CSPs. Finally, Section 5 concludes and discusses future work.

2 Background

In this section, we review work carried out in the database community for XML query processing, focusing on the widely used query language XPath, and also give some basic background on constraint satisfaction.

2.1 XML Databases and Query Processing

XML *elements* in angled braces (i.e., '<' and '>') are used to denote object instances. Each element carries a *label* describing a class the object belongs to (e.g., university, department, faculty). All information related to an object is enclosed between the beginning and ending tag the object.

XML documents can be modeled as rooted node-labeled graphs (or simply trees, in the absence of reference links), where the intermediate nodes take values from the set of potential element labels (or else object classes) and the leaves store textual information. For example, Figure 1 shows the XML tree for a part of an XML document, containing information about a University. XML syntax also allows for reference links (i.e., IDREF) between elements. For instance, a faculty member could refer to the university where he graduated (e.g., 'BigSchool') by a reference link (e.g., IDREF='BigSchool').

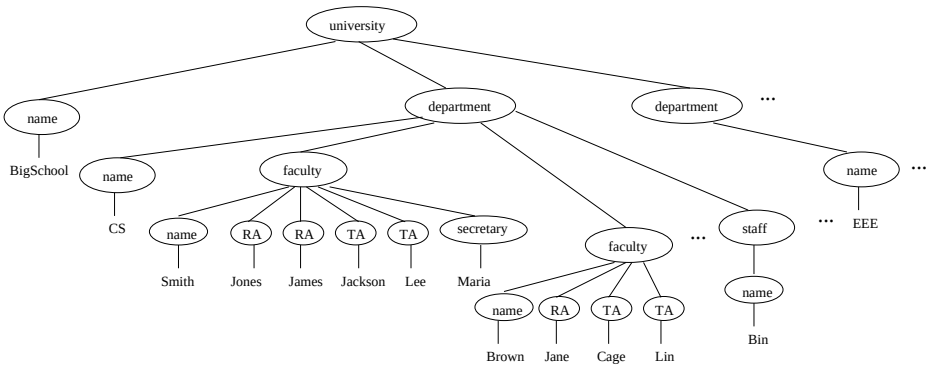


Fig. 1. An XML tree

The World Wide Web Consortium [19] has been continuously revising the definition of XPath; a query language for XML data. XPath allows for the definition of queries where the problem variables must have some structural relationships. For instance, the XPath expression document("university.xml")//`faculty`/`name` refers to all nodes labeled `name`, which have a parent labeled `faculty`, in the XML document file "university.xml".

The popularity of XML attracted database researchers to study the efficient management of XML data. As a result, a number of native XML management systems or

extensions of relational systems have been developed [15, 3]. In addition, new indexing methods and query processing algorithms have been proposed [16]. Some of these methods (e.g., [1, 9, 10]) consider the documents as labeled trees, ignoring the cycles due to IDREF links. Others (e.g., [11]), are also applicable for path queries in node labeled graphs, which may contain cycles. In general, from a CP perspective, only easy problems (i.e., high selective queries with few variables of large domains) have been considered and the methods aim at minimizing the I/O cost.

Recently is has been proved that all queries that can be expressed by XPath 1.0 can be evaluated in PTIME [7]. Different polynomial worst-case bounds have been provided for several classes of such queries. Interestingly, [7] demonstrates that several commercial XPath 1.0 evaluation engines still use exponential algorithms even for simple, polynomial queries such as the ones discussed in this paper. Our work is related to that of [7, 8] in the sense that we discuss polynomial algorithms for various classes of queries, albeit from a CSP perspective. However, we also discuss the evaluation of generic graph containment queries, beyond XPath 1.0. Such queries can be expressed by the new version (2.0) of XPath, as discussed later.

2.2 Constraint Satisfaction Problems

Constraint satisfaction is a paradigm that can capture a wide variety of problems from AI, engineering, databases, and other disciplines. A constraint satisfaction problem (CSP) consists of a set of variables $X = \{x_1, \ldots x_n\}$, a set of domains $D = \{D(x_1), \ldots, D(x_n)\}$, where $D(x_i)$ is the finite set of possible values for variable x_i, and a set C of constraints over subsets of the variables. A constraint c on variables $x_i, \ldots, x_j$ is a subset of the Cartesian product $D(x_i) \times \ldots \times D(x_j)$ that specifies the allowed combinations of values for variables $x_i, \ldots, x_j$. The operation performed to determine whether a constraint is satisfied is called a *consistency check*. An assignment of a value a to variable x_i is denoted by (x_i, a).

CSPs that contain constraints between at most two variables are called binary. CSPs with constraints between arbitrary numbers of variables are called n-ary (or non-binary). A CSP is usually represented by a constraint graph (or hyper-graph in the case of n-ary problems) where nodes correspond to variables and edges (hyper-edges) correspond to constraints. The basic goal in a CSP is to find one or all assignments of values to variables so that all the constraints are satisfied.

A constraint $c = (x_i, x_j)$ is *arc consistent* (AC) iff for each value a in $D(x_i)$ there exists a value b in $D(x_j)$ so that the assignments (x_i, a) and (x_j, b) satisfy c. In this case we say that b is a *support* for a on constraint c. A binary CSP is AC if all its constraints are arc consistent. These definitions extend to non-binary constraints in a straightforward way. A non-binary constraint is *generalized arc-consistent* (GAC) iff for any variable in the constraint and value that it is assigned, there exist compatible values for all the other variables in the constraint.

3 Formulating XML Queries as CSPs

While evaluating an XML query, we actually search for a set of elements in the XML graph with labels and structural relationships between them that match the labels of the

nodes in the query and the relationships between them. We can represent the entities in a query as variables, the elements (nodes) in the XML graph as the possible values of the variables, the labels of the query nodes as unary constraints, and the structural relationships between query nodes as directed binary constraints. Queries expressed in XPath can be transformed into CSPs in a straightforward way. Each variable in the XPath expression becomes a variable in the CSP, the domains are the nodes of the XML graph, and the constraints are the relationships between variables in the query.

Example 1. Consider the query "is there any department which has 3 faculty members with one RA and one TA?". Figure 2a shows how we can express it using XPath 2.0. Observe that the expression already uses similar terminology to CSPs. It asks for instances of nodes labeled `department` (variable x_1), which are ancestors of three *different* `faculty` nodes (variables x_2, x_3, x_4), which have children labeled RA and TA. The query can be modeled as a CSP; the corresponding constraint graph is shown in Figure 2b. There are 10 variables with unary constraints (some of the labels are omitted from the graph for the sake of readability). E.g., x_2, x_3, x_4 can take label `faculty`. There are also binary constraints, denoting ancestor/descendant or parent/child relationships. E.g., x_1 is an ancestor of each one of x_2, x_3, and x_4. Finally, there are inequality constraints between x_2, x_3, and x_4 in order to forbid these variables to take the same value. These constraints could be alternatively modeled as a non-binary *all-different* constraint.

We can represent each element e in the XML data graph with a quadruple $\langle label(e), pre(e), post(e), pre_{parent}(e)\rangle$, where $label(e)$ is the label of the node, and $pre(e)$ and $post(e)$ are the values given to element e by a preorder and postorder traversal of the rooted graph (ignoring IDREF links). We also keep the preorder value of e's parent node. This representation facilitates the fast implementation of various types of constraint checks. [9] uses a similar representation to build an indexing structure on XML data.[3]

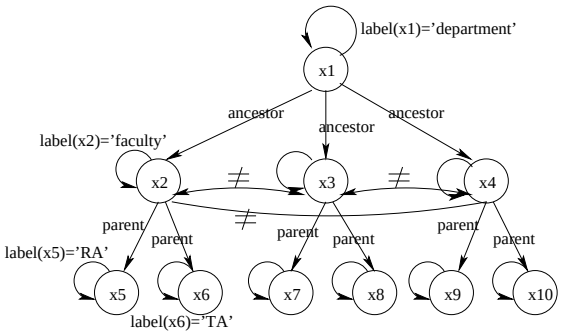
The *primitive* structural relationships between nodes in XML graphs are *child, parent, sibling, descendant, ancestor*. These relationships can be expressed as binary directed constraints in the CSP formulation. In Table 1, we define the semantics of the primitive constraints and also the precedence constraints *preceding, following, preceding_sibling, following_sibling*, which will be described shortly. For each constraint c on variables x_i and x_j, a constraint check amounts to checking whether the corresponding conditions of Table 1 hold. This can be done in constant time. Note that for every directed constraint $c(x_1, x_2)$ there is a equivalent inverse constraint $\overline{c}(x_2, x_1)$. For example, the inverse of $parent(x_1, x_2)$ is $child(x_2, x_1)$.

Another useful observation is that the child constraints are functional and the parent constraints are piecewise functional [18]. A binary constraint $c = (x_i, x_j)$ is *functional* if for every value $a \in D(x_i)$, there exists at most one support in $D(x_j)$. Consider a constraint $child(x_i, x_j)$. Each value $a \in D(x_i)$ can only have one parent in the XML graph. Therefore, the constraint is functional. A *piecewise functional* constraint $c = (x_i, x_j)$ is a constraint where the domains of x_j can be partitioned into groups such that

[3] Note that the choice of element representation is independent of the CSP formulation; other representations are possible with only slight changes in the definitions of constraints.

```
some $x1 in ("university.xml")//department
satisfies ((some $x2 in $x1//faculty
            satisfies $x2[RA and TA])
       and (some $x3 in $x1//faculty
            satisfies $x3[RA and TA])
       and (some $x4 in $x1//faculty
            satisfies $x4[RA and TA])
       and ($x2 isnot $x3)
       and ($x2 isnot $x4)
       and ($x3 isnot $x4))
```

(a) query expression in XPath 2.0



(b) a constraint graph representation

Fig. 2. Two representations of an XML query

each value of $D(x_i)$ is supported by at most one group of $D(x_j)$. Consider a constraint $parent(x_i, x_j)$. We can partition $D(x_j)$ into groups such that each group includes the children of a value $a \in D(x_i)$ in the XML graph. Now each value $a \in D(x_i)$ will be supported by at most one group, and therefore the constraint is piecewise functional.

$$
\begin{array}{l}
parent(x_1, x_2) \Leftrightarrow pre(x_1) = pre_{parent}(x_2) \\
child(x_1, x_2) \Leftrightarrow pre_{parent}(x_1) = pre(x_2) \\
sibling(x_1, x_2) \Leftrightarrow pre_{parent}(x_1) = pre_{parent}(x_2) \wedge pre(x_1) \neq pre(x_2) \\
descendant(x_1, x_2) \Leftrightarrow pre(x_1) > pre(x_2) \wedge post(x_1) < post(x_2) \\
ancestor(x_1, x_2) \Leftrightarrow pre(x_1) < pre(x_2) \wedge post(x_1) > post(x_2) \\
preceding(x_1, x_2) \Leftrightarrow pre(x_1) < pre(x_2) \\
following(x_1, x_2) \Leftrightarrow pre(x_1) > pre(x_2) \\
preceding\_sibling(x_1, x_2) \Leftrightarrow pre(x_1) < pre(x_2) \wedge sibling(x_1, x_2) \\
following\_sibling(x_1, x_2) \Leftrightarrow pre(x_1) > pre(x_2) \wedge sibling(x_1, x_2)
\end{array}
$$

Table 1. Semantics of primitive and precedence constraints

The precedence relationships *preceding* and *following* are used to express ordering associations between XML constructs, conventionally based on a preorder traversal of the graph. Note that the preorder traversal is possible if we ignore all relationships among elements except the hierarchical *child*, *parent* relationships. Other precedence relationships are *preceding_sibling*, *following_sibling*, with obvious meaning. The primitive and precedence relationships do not capture all structural information contained in any XML document. There could also be *referential relationships* that represent IDREF

links. Simply put an IDREF relationship allows one to specify a pointer from one element e to another element e'.

Precedence relationships can be captured in the CSP model using simple constraints on the preorder values of elements, as shown in Table 1. Since IDREF links do not carry any specific semantics they cannot be easily represented by a function (or predicate). We can represent them explicitly by their allowed tuples. That is, a referential constraint c between variables x_i and x_j is encoded as a table, where each row is a 2-tuple $\langle e, e' \rangle$, such that $e \in D(x_i)$, $e' \in D(x_j)$, and there exists a referential constraint between elements e and e' in the XML data graph.

The XML queries that have been investigated in the database literature typically have a few variables (of large domains) connected with primitive structural constraints. Such queries correspond to small and relatively easy CSPs and can be handled more efficiently using database techniques. However, these techniques are not suitable for queries involving large numbers of densely constrained variables, with relatively small domains. In such cases, advanced indexing methods are impractical and evaluation degenerates to the use of simple nested loops joins, corresponding to static chronological backtracking search in CSP terminology. In contrast, constraint programming has more advanced search algorithms combined with powerful heuristics to offer.

4 Query Evaluation as Constraint Satisfaction

In our framework, query evaluation can be simply viewed as constraint satisfaction search. We introduce a simple method for preprocessing CSPs with XML structural constraints that can reduce the search effort by adding implied constraints to the problem. We also discuss search algorithms for such CSPs.

4.1 Constraint Inference

Constraint inference has been shown to be very useful as a preprocessing step in certain classes of CSPs. By "constraint inference" we mean the addition of new constraints to the problem that are implied by existing ones. In temporal CSPs, for example, path consistency algorithms are used to preprocess the given problem, usually resulting in considerable reduction of the search effort [4]. Constraint inference can replace existing constraints with tighter ones and even detect inconsistency in a problem, avoiding unnecessary search. In general CSPs, algorithms like path consistency are not commonly used, but recent studies (e.g., [17, 6]) have demonstrated the benefits of adding implied (sometimes called redundant) constraints in various types of problems.

Constraint inference can also be used for preprocessing in the context of CSPs with structural constraints. Inference operations *inversion*, *intersection* and *composition* defined for (directed) temporal constraints in [4] can be naturally extended for (directed) structural constraints. Inversion of a constraint $c(x_i, x_j)$ (denoted by $\overline{c}(x_i, x_j)$) infers a constraint $c'(x_j, x_i)$. For instance, the inversion $\overline{parent}(x_i, x_j)$ of constraint $parent(x_i, x_j)$ is $child(x_j, x_i)$. Intersection (denoted by $\oplus$) computes the "tightest" constraint on variables x_i and x_j that can be derived from two constraints $c(x_i, x_j)$ and $c'(x_i, x_j)$. For instance, the intersection $ancestor(x_i, x_j) \oplus parent(x_i, x_j)$ is $parent(x_i, x_j)$. Note that not all intersections of primitive constraints give consistent

results. For example, the intersection $child(x_i, x_j) \oplus parent(x_i, x_j)$ is inconsistent. The composition of two constraints $c(x_i, x_j)$ and $c'(x_j, x_k)$ (denoted by $c(x_i, x_j) \otimes c'(x_j, x_k)$) derives a new constraint $c''(x_i, x_k)$ by transitivity. For instance, the composition $parent(x_i, x_j) \otimes parent(x_j, x_k)$ is $ancestor(x_i, x_k)$. Inversion, intersection, and composition tables for all combinations of primitive and precedence constraints can be easily derived. In addition, we can adapt path consistency algorithms used in temporal CSPs [4] to minimize the constraint graph of an XML query and/or detect inconsistency. Details are omitted due to space constraints.

4.2 Tractable queries

Primitive Structural Constraints An important and very common class of XML queries can be expressed using only the primitive structural relationships *child, parent, sibling, descendant, ancestor*. For example, consider the query "find faculty members with an RA and a TA". Note that their constraint graph corresponds to a tree. As a result, these queries can be processed in PTIME, since according to [5], AC ensures backtrack-free search in acyclic binary constraint graphs[4]. To be precise, AC for CSPs of such queries can be achieved in time O(ed), where e is the number of primitive constraints in the query and d is the domain size of the variables (i.e., the size of the data). This follows directly from [18] where it is proved that AC for functional and piecewise functional constraints is computable in O(ed) time. Then backtrack-free search can be achieved in linear time by visiting each variable x, starting from the root of the constraint graph, in depth-first order, assigning a value a to x and instantiating the children of x by the supports of (x, a) in them, recursively. Note that queries with primitive constraints only are part of a core fragment of XPath 1.0, which was shown to be in time linear to the size of the query and the size of the data in [7]. Here, we have shown how the same result can be derived using constraint programming theory.

All-different + Ancestor-Descendant Constraints We now study a special class of non-binary constraints that are commonly found in XML queries. Consider for example the constraint graph of Figure 2b. This graph is a tree of primitive structural constraints plus a set of binary "not-equals" constraints among sibling nodes with the same label. For example, there are three `faculty` nodes linked with binary "not-equals" constraints and connected to their parent in the graph by the *same* ancestor constraint. Alternatively, we could have a common parent/child constraint. We can consider this set of relationships as a special non-binary constraint, which we call *all-different + ancestor-descendant* constraint (ADAD for short).

Definition 1. An ADAD constraint on variables $x_1, \ldots, x_k$, where variables x_i, $i = 1, \ldots, k-1$ are siblings of the same label and variable x_k is their parent, is defined as $c_{AP}(x_k, x_1) \wedge \ldots \wedge c_{AP}(x_k, x_{k-1}) \wedge$ *all-different*$(x_1, \ldots x_{k-1})$, where c_{AP} is a single $parent$ or $ancestor$ constraint.

For example, the constraints between variables x_1, x_2, x_3, and x_4 in Figure 2b can be replaced by a 4-ary ADAD constraint. This constraint is the conjunction of the

[4] Note that a weaker form of AC, called directional AC, is enough to guarantee backtrack-free search.

all-different constraint between x_2, x_3, and x_4, and the ancestor-descendant constraints between x_1 and each of x_2, x_3, and x_4. We can now model an XML query graph, where the query involves only primitive relationships, as a set of ADAD constraints. Binary constraints like $parent(x_2, x_5)$ can trivially be considered ADAD constraints with a single child. Conjunctive non-binary constraints like ADAD with specialized filtering algorithms are useful in many CSPs (e.g., [14]).

The ADAD constraint can also be used to model aggregation queries with the XPath function *count(node-set)*. This function takes a set of nodes as argument and returns the number of nodes in the set, and can be used to select tree patterns with a specific number of elements or restrict the number of elements to be within a specific range. For example, the query //`faculty`[count(child::`lecturer`)$\geq$5] selects elements labeled `faculty` that have 5 or more children labeled `lecturer`. We can formulate such a query using an ADAD constraint on one variable labeled `faculty` and five variables labeled `lecturer`. The use of ADAD constraints is particularly suitable for queries where the count() function is applied on variables that are inner nodes (not leaves) of the query constraint graph. For example, the query "find a faculty member with 5 children labeled lecturer, such that each one of them has a child labeled TA".

In the discussion that follows, when we refer to an ADAD constraint, we use 'parent' to denote the common parent or ancestor in the constraint and 'children' to denote its children or descendants. Note that the term 'parent' is overloaded to denote the parent node of the *query graph*; e.g., in the graph of Figure 2b, the parent of x_2, x_3, x_4 is x_1, however, the relationships on the corresponding edges are *ancestor*.

An XML query with ADAD constraints could alternatively be modeled using only primitive relationships as a binary CSP, or as a CSP involving only binary and non-binary all-different constraints. Summarizing we can consider three models:

binary model - in this model the relationships are captured by (i) binary structural constraints (child, parent, ancestor, descendant) and (ii) binary 'not-equals' constraints between sibling nodes of the same label.

mixed model - in this model the relationships are captured by (i) binary structural constraints (child, parent, ancestor, descendant) and (ii) non-binary all-different constraints between sibling nodes.

non-binary model - in this model the relationships are captured by non-binary ADAD constraints only.

As we will show later, achieving GAC in the non-binary model guarantees that a solution can be found in backtrack-free manner. In Figure 3 we sketch an algorithm of low complexity that computes GAC for an ADAD constraint. The algorithm explicitly verifies the consistency of each value a of the parent variable x_k, by reducing the domains of the children variables according to a and applying GAC for the all-different constraint between them. If a value a of x_k is eliminated, the values of the children variables which are consistent with a are deleted from the corresponding domains.

To prove that the algorithm of Figure 3 achieves GAC for an ADAD constraint we need to show that if a value in the domain of some variable is not GAC then the algorithm will remove it. This divides in two cases. First, assume that value a of variable x_k (i.e. the parent node) is not GAC. This means that there is no supporting tuple

```
boolean GAC (ADAD constraint c(x_1, x_2, ..., x_k))
1:   for each x_i, i ∈ {1, ..., k − 1}
2:     for each value b ∈ D(x_i)
3:       if b has no support in D(x_k) remove b from D(x_i)
4:   for each value a ∈ D(x_k) //parent
5:     for each x_i, i ∈ {1, ..., k − 1}
6:       temporarily remove from D(x_i) all values b
                 such that parent(a, b)=false;
7:     compute GAC for constraint all-different(x_1, ... x_{k−1});
8:     if there is a domain wipeout
9:       remove a from D(x_k);
10:      for each x_i, i ∈ {1, ..., k − 1}
11:        permanently remove all values b from D(x_i)
                 such that parent(a, b)=true;
12:    restore all temporarily removed values;
13:  if D(x_k) is wiped out return false;
14:  return true;
```

Fig. 3. GAC filtering for an ADAD constraint

$\langle b_1, \ldots, b_{k-1} \rangle$, where $b_i \in D(x_i)$, such that $\forall\ b_i\ parent(a, b_i) = TRUE$ and *all-different*$(b_1, \ldots, b_{k-1}) = TRUE$. In this case, when value a is examined, GAC on the all-different sub-constraint will detect the inconsistency (lines 7, 8) and a will be removed (line 9). Second, assume that value b_j of variable x_j, with $j \neq k$, (i.e. a child node) is not GAC. This means that there is no supporting tuple $\langle b_1, \ldots, b_{j-1}, b_{j+1}, \ldots, b_{k-1}, a \rangle$, where $b_i \in D(x_i), i \neq j$ and $a \in D(x_k)$, such that the ADAD constraint is satisfied. This can happen in two cases; first if the parent of b_j in the XML graph is not in the domain of x_k (e.g., its parent is not labeled `department`). Such values are eliminated right in the beginning of the algorithm (lines 1–3). Now, assume that the parent of b_j in the XML graph is value $a \in D(x_k)$. When a is examined (line 4), the algorithm will compute the set of supporting values of a in each variable $x_1, \ldots x_{k-1}$, temporarily reducing the domain of variables to only values consistent with a. Since value b_j is not GAC, but it has support in x_k, it should be not GAC with the reduced domains of the other children variables and it will be eliminated during GAC of the sub-constraint *all-different*$(b_1, \ldots, b_{k-1})$.

We now discuss the complexity of the algorithm of Figure 3.

Proposition 1. The worst-case time complexity of applying GAC on one ADPC constraint is $O(d^2 k\sqrt{k})$.

Proof. The preprocessing step (lines 1–3) takes $O(kd)$ time to enforce AC on the $k-1$ ancestor-descendant constraints. At the same time we can build a data structure which for each value a of x_k holds two pointers; one to the first descendant of a (according to the preorder values of the elements in the XML graph) and another to its last descendant. The outer iteration of line 4 is executed at most d times; once for each value a of the ancestor variable x_k. In each iteration we reduce the domains of variables $x_1, \ldots x_{k-1}$ to include only values that are descendants of a in the XML graph. This is done using the data structure built before. Then, we apply a GAC algorithm on the all-different sub-constraint over variables $x_1, \ldots x_{k-1}$, with d domain size each in the worst case. Using

the algorithm of [13], this can done in O($dk\sqrt{k}$) time. For d values of the outer iteration, we get O($d^2k\sqrt{k}$). The complexity, including the preprocessing step, is O($kd+d^2k\sqrt{k}$) = O($d^2k\sqrt{k}$).

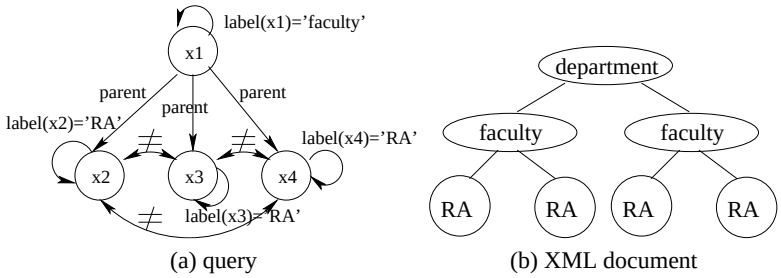
In the case where the ADPC constraint consists of parent-child constraints, the complexity of GAC reduces to O($dk\sqrt{k}$). This can be achieved by taking advantage of the piecewise functionality of parent-child constraints in the following way: The domains of the children are partitioned into g groups, one for each value in the domain of the parent (which has domain size g). Now, the iteration of line 4 in Figure 3 will be executed g times, and in each iteration the cost of GAC on the all-different constraint will be O($\frac{d}{g}k\sqrt{k}$). Thus, the complexity of GAC on the ADPC constraint will be O($g\frac{d}{g}k\sqrt{k}$)=O($dk\sqrt{k}$). The same holds for ADPC constraints consisting of ancestor-descendant constraints that are piecewise functional. This occurs when any label in the data graph does not appear more than once along any branch.

If some query contains more than one ADAD constraints then in order to achieve GAC in the constraint graph, we can adapt a standard AC algorithm to use the filtering function of Figure 3 and propagate the consistency of ADAD constraints using a stack. If some query contains e ADPC constraints then GAC can be applied in O($ed^2k\sqrt{k}$) time in the general case, and O($edk\sqrt{k}$) time for the special cases discussed above. Realistically, the ADPC constraints in a query do not share more than one variable, which means that $e = n/k$. In this case the complexity of GAC in the general case is O($nd^2\sqrt{k}$). In the special piecewise functional cases, GAC can be applied in O($nd\sqrt{k}$) time, which is particularly efficient.

According to [5], GAC ensures backtrack-free search in constraint graphs that are hyper-trees. Therefore achieving GAC in the non-binary model of an XML query, consisting of primitive relationships only, is enough to guarantee backtrack-free search, and thus polynomial query evaluation. In XPath, one expression binds exactly one variable and usually all solutions are required. That is we need to retrieve all the values of the variable that satisfy the query. In the CSP formulation, these values are the arc consistent values in the variable's domain. We can also easily retrieve the whole tree patterns that match the query pattern using the data structure described in the proof of Proposition 1. It is interesting to note that queries with ADAD constraints only belong to a special class called "extended Wadler fragment" in [8]. For queries in this class [8] provide an algorithm of O(n^2d^2) cost, whereas our GAC + backtrack free search approach can solve such problems at a lower complexity.

A natural question to ask is if it is necessary to introduce the non-binary model with the ADAD constraints in order to achieve backtrack-free search. As we discussed, there are two alternative models for the problem; the binary model and the mixed model. Can AC (GAC) in these two models guarantee backtrack free search? As the following example shows, the answer to this question is no.

Example 2. Consider the query depicted in Figure 4a and the XML data graph of Figure 4b. If we model this problem as a CSP we will have one variable x_1 for the academic and three variables x_2, x_3, x_4 for the RAs. The domain of x_1 comprises two values, corresponding to the two academics of the XML data graph. Similarly, the domains of x_2, x_3, and x_4 include the four RA nodes of the XML graph. Let us first assume that



(a) query (b) XML document

Fig. 4. necessity of non-binary model

AC is applied on the binary model (i.e., considering all constraints as binary). Observe that the two possible values for x_1 have supports in the domains of x_2, x_3, and x_4. In addition, the values of each child variable find support in the parent's domain and also in the domains of the sibling variables using the binary "non-equals" constraint. However, it is easy to see that the problem is inconsistent. To discover the inconsistency, we have to search. Similarly, AC on the mixed model leaves the variable domains unchanged; enforcing GAC using the non-binary all-different constraint does not prune any values. This example proves the following proposition.

Proposition 2. Let Q be an XML query, consisting of primitive relationships only, represented in either the binary or the mixed model. Achieving GAC in the constraint graph of Q does not guarantee backtrack-free search.

Precedence Constraints We now discuss how to deal with precedence constraints (i.e., *preceding, following*) that may be included in a query. Queries with primitive and precedence constraints only form the core fragment of XPath 1.0 [7]. Note that a precedence constraint in a query with only structural constraints may define a cycle, which, in theory, could make the problem intractable. However, we show how to obtain PTIME complexity (as proved in [7]) by changing the constraint model. The key idea is that a precedence constraint between any two nodes x_i, x_j of the query graph can be reduced to a precedence constraint between two of their ancestors $a(x_i), a(x_j)$ which are siblings. If there are no such ancestors, this means that the subgraphs where x_i and x_j belong are only connected by the precedence constraint and thus search complexity is not affected by it. Thus, we only need to consider how precedence constraints between siblings can affect search. In a way similar to the ADAD constraints, we can treat them as ternary constraints defined by $c_{AP}(p, a(x_i)) \wedge c_{AP}(p, a(x_j)) \wedge c_{PR}(x_i, x_j)$, where c_{AP} is a $parent$ or $ancestor$ constraint and c_{PR} is the original precedence constraint between x_i and x_j. Thus, each arbitrary precedence constraint is either a simple binary constraint connecting otherwise disconnected query subgraphs or can be reduced to a simple, local ternary constraint between a parent and its children. In any case, the constraint graph reduces to a hyper-tree. GAC can be applied on this hyper-tree in PTIME and ensures backtrack-free search. In summary, we have shown that any query with only simple structural, precedence, and ADAD constraints can be solved in PTIME using constraint programming techniques.

4.3 Intractable Queries

Queries that include referential relationships are intractable. Such relationships induce cycles in the constraint graph of the query. As proved in [5], CSPs whose constraint graph contains arbitrary cycles are NP-complete. The same result can be derived for arbitrary graph containment problems [12]. Note that such queries cannot be expressed by XPath 1.0, but only by XPath 2.0, which allows for the definition of variables and links between them via IDREF relationships.[5]

For intractable problems, we experimentally studied the effects of considering ADAD constraints in GAC preprocessing and search. First, we generated a large XML graph, which is a hierarchy of nested containments enriched by IDREF links. There are 7 labels (0–6, apart from an unlabeled root), each corresponding to a level of the tree; i.e., nodes with label 0 can only have children with label 1, nodes with label 1 can only have children with label 2, etc. Each generated node has a random number of children between 0 and 6. Finally, from the node pairs (e_i, e_j), which are have no parent/child relationship and do not have the same label, 10% are selected and an IDREF link is generated between e_i and e_j.

Next, we generated a number of XML queries (i.e., CSPs) as follows. Each query class is described by a quadruple $\langle s, e, k, p\rangle$. For each query in the class, there is one variable labeled s. Each variable labeled l, for $s \leq l < e$, has exactly k children labeled $l + 1$. Also, for $p\%$ of the variable pairs (x_i, x_j) with different labels, not linked by a parent/child constraint, an IDREF constraint is generated. For instance, in queries of the class $\langle 2, 4, 3, 10\rangle$ there is a variable labeled 2, with three children labeled 3, each of which has three children labeled 4. Also 10% of variable pairs of different labels not linked by a parent/child constraint are linked by an IDREF constraint.

We implemented two AC algorithms. The first is the AC2001 algorithm [2] that considers only binary constraints. The second is a GAC algorithm that considers non-binary ADAD constraints and employs the filtering technique of Figure 3. For binary IDREF constraints, GAC operates like AC2001. We also implemented four search algorithms. The first one, denoted by FC, is a version of *forward checking* that treats all constraints as binary. The second one, denoted by MAC, maintains AC on all binary constraints (using AC2001) after each instantiation. The third one, denoted by FC$^+$ is slightly stronger than FC, enforcing GAC on ADAD constraints when parent variables are instantiated. Finally, MGAC is a method that maintains full GAC.

In order to reduce the checks and computational effort of all filtering and search methods, the XML data elements are sorted on $(label(e), pre(e))$. We create a directory index that points for every label to the first and the last position in the elements array that contain this label. Note that this is done only once and used for all XML queries, later on. Before evaluating a query, we apply a preprocessing step that finds for each value e_x of a parent variable x the first consistent value in the domains of its children variables. For the children y labeled α of $x \leftarrow e_x$ we apply binary search on the sorted elements array to find the first value e_{y_f} labeled α with preorder larger than or equal to $pre(e_x)$.

[5] A polynomial fragment of queries, called XPattern, that considers IDs is shown to be in PTIME in [7]. However, this fragment only includes queries that explicitly select nodes based on their IDs and therefore such references can be dealt with at a preprocessing step. In a CSP context, we can consider such ID selections as unary (i.e., node) constraints.

During AC or search, we can immediately access all values $e_y \geq e_{y_f}$ of each child y of x variable are consistent with $x \leftarrow e_x$, while $post(e) < post(e_x)$, whenever we need to check the parent/child constraints for $x \leftarrow e_x$. The preprocessing step simulates a database technique (e.g., [9]) that could use an index for the same purpose. Moreover, we immediately eliminate e_x from the domain of x (and its children from the domains of the children variables), if the e_x has less children than k in the data graph (i.e., less than those in the ADAD constraint with x as parent). In this case, the all-different constraint cannot be satisfied among the children (Hall's theorem). Note that this is a cheap (but incomplete) filtering for the non-binary ADAD constraint and enforcing it before binary algorithms actually favors them.

Table 2 compares the efficiency of four filtering and search combinations for several query classes. For instance, AC-FC denotes that binary AC is used for filtering before search and if the problem is arc-consistent, FC is used for search. For each query class, we generated 30 instances and averaged the search time (in seconds) and the number of variable instantiations during search (enclosed in parentheses). Note that the parent-child constraints are the same for each problem in a query class; only the IDREF constraints change between problems. p is tuned so that query classes are in the hard region (i.e., roughly half problems are insoluble). Horizontal lines separate classes with different number of children in their ADAD constraints (i.e., different k).

query	AC-FC	AC-MAC	GAC-FC$^+$	GAC-MGAC
1. $\langle 2, 6, 2, 3.8\rangle$	32.9(29K)	3.7(86)	23.1(16.7K)	3.7(86)
2. $\langle 3, 6, 2, 14\rangle$	2.7(2843)	4.3(87)	1.9(2198)	4.3(87)
3. $\langle 3, 6, 2.5, 3\rangle$	21.5(6811)	8.9(130)	6.4(2225)	1.3(39)
4. $\langle 4, 6, 3, 25\rangle$	7.2(2113)	53.2(215)	3.6(937)	16.2(77)
5. $\langle 4, 6, 4, 8\rangle$	177.1(98K)	228.8(4102)	1.3(292)	3.2(32)
6. $\langle 4, 6, 5, 3\rangle$	1.6(616)	8.3(183)	0.1(44)	0.8(23)

Table 2. Search performance of algorithms

Algorithms that consider the non-binary constraint (i.e., GAC-FC$^+$ and GAC-MGAC) are up to two orders of magnitude faster than their binary counterparts (i.e., AC-FC and AC-MAC) and the difference, in general, increases with k. Note that when $k = 2$ (i.e., each parent variable has only two children) there is no difference between AC and GAC. For these two classes FC$^+$ performs better than FC; enforcing AC for the children of the instantiated variable pays-off. Full maintenance of arc consistency is not always beneficial; MAC algorithms were found better than their FC counterparts for problem classes 1 and 3 only.

5 Summary

In this paper we presented a new CSP-based way for the processing of XML queries. We identified the most common structural constraints between XML query variables and showed how constraint inference, filtering, and search can be adapted for networks of such constraints. We identified the particularly important *all-different + ancestor-descendant* (ADAD) non-binary constraint and theoretically showed that queries containing only such constraints can be processed efficiently using constraint programming. To achieve this, we described a polynomial filtering algorithm for this constraint.

Going one step further, we showed that maintaining this constraint brings significant benefits to search in intractable problems that contain ADAD and IDREF constraints. In the future, we plan to study the coverage of additional XPath constructs (e.g., quantifiers) with constraint programming techniques. In addition, intend to investigate the application of other (apart from *all-different*) specialized non-binary constraints (e.g., the global cardinality constraint) in hard search problems on XML data graphs. We will also perform an empirical comparison of constraint programming and database methods, such as the ones presented in [7].

References

1. S. Al-Khalifa, H. V. Jagadish, J. M. Patel, Y. Wu, N. Koudas, and D. Srivastava. Structural joins: A primitive for efficient XML query pattern matching. In *Proceedings of IEEE ICDE*, 2002.
2. C. Bessière and J. Régin. Refining the basic constraint propagation algorithm. In *Proceedings of IJCAI*, 2001.
3. A. B. Chaudhri, A. Rashid, and R. Zicari. *XML Data Management: Native XML and XML-Enabled Database Systems*. Addison-Wesley, 2003.
4. R. Dechter, I. Meiri, and J. Pearl. Temporal constraint networks. *Artificial Intelligence*, 49:61–95, 1991.
5. E. Freuder. A sufficient condition for backtrack-free search. *Journal of the ACM*, 29(1):24–32, 1982.
6. I. Gent and B. Smith. Symmetry Breaking in Constraint Programming. In *Proceedings of ECAI*, pages 599–603, 2000.
7. G. Gottlob, C. Koch, and R. Pichler. Efficient algorithms for processing XPath queries. In *Proceedings of VLDB*, 2002.
8. G. Gottlob, C. Koch, and R. Pichler. XPath query evaluation: Improving time and space efficiency. In *Proceedings of ICDE*, 2003.
9. T. Grust. Accelerating XPath location steps. In *Proceedings of ACM SIGMOD*, 2002.
10. H. Jiang, H. Lu, and W. Wang. Efficient processing of XML twig queries with or-predicates. In *Proceedings of ACM SIGMOD*, 2004.
11. R. Kaushik, P. Bohannon, J. F. Naughton, and H. F. Korth. Covering indexes for branching path queries. In *Proceedings of ACM SIGMOD*, 2002.
12. G. Miklau and D. Suciu. Containment and equivalence for a fragment of XPath. *J. ACM*, 51(1):2–45, 2004.
13. J. C. Régin. A filtering algorithm for constraints of difference in CSPs. In *Proceedings of AAAI*, 1994.
14. J. C. Régin and M. Rueher. A global constraint combining a sum constraint and difference constraints. In *Proceedings of CP*, 2000.
15. J. Shanmugasundaram, K. Tufte, C. Zhang, G. He, D. J. DeWitt, and J. F. Naughton. Relational databases for querying XML documents: Limitations and opportunities. In *Proceedings of the VLDB Conference*, 1999.
16. D. Shasha, J. T.-L. Wang, and R. Giugno. Algorithmics and applications of tree and graph searching. In *Proceedings of ACM PODS*, 2002.
17. B. Smith, K. Stergiou, and T. Walsh. Using auxiliary variables and implied constraints to model non-binary problems. In *Proceedings of AAAI*, 2000.
18. P. van Hentenryck, Y. Deville, and C.-M. Teng. A generic arc-consistency algorithm and its specializations. *Artificial Intelligence*, 57:291–321, 1992.
19. WWW Consortium. *XML Path Language (XPath) 2.0, W3C Working Draft, http://www.w3.org/TR/xpath20/*, November 2003.