TESTING THE SATISFIABILITY OF TREE PATTERN QUERIES WITH NODE IDENTITY CONSTRAINTS

A Thesis by

Barbara Jane Gobbert

B. Science, University Of Queensland, 1979

B. Commerce, University Of Queensland, 1983

Submitted to the Department of Computer Science
and the faculty of the Graduate School of
Wichita State University
in partial fulfillment of the
requirements for the degree of
Master of Science

May 2007

TESTING THE SATISFIABILITY IN TREE PATTERN QUERIES
WITH NODE IDENTITY CONSTRAINTS

I have examined the final copy of this Thesis for form and content and recommend that it be accepted in partial fulfillment of the requirement for the degree of Master of Science, with a major in Computer Science.

_____
Prakash Ramanan, Committee Chair

We have read this Thesis
and recommend its acceptance:

_____
Prakash Ramanan, Committee Member

_____
Sattiraju Prabhakar, Committee Member

_____
Thomas DeLillo, Committee Member

DEDICATION

To my family

ACKNOWLEDGEMENT

I would like to thank my advisor, Dr. Ramanan, for his help and insight on issues raised during the course of this research. This work was motivated by my previous work with the use of XML in the management, storage and retrieval of long-term electronic records as part of the Victorian Electronic Records Strategy in Australia.

ABSTRACT

This research deals with testing the satisfiability of a subclass of XQuery and XPath expressions that contain node identity constraints. This subclass of expressions is called Conjunctive XPath. A query is satisfiable if there exists a database of XML documents that will result in a non-empty answer to the query, whereas a query that is not satisfiable will result in an empty answer when run against any database. Determining that a query is unsatisfiable prior to execution will result in savings in computer run-time by not executing unsatisfiable queries. Although the general problem is undecidable, we examine a subclass of queries called Conjunctive XPath where satisfiability is decidable. Previous researchers have presented algorithms for determining satisfiability based on predicate logic and also using non-deterministic finite automata. We present an algorithm for XPath queries with a single node identity constraint, based on topological sorting. This algorithm has faster run-time compared to previously known algorithms.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF ABBREVIATIONS / NOMENCLATURE

Symbol	Meaning
NIC	Node Identity Constraint
TPQ	Tree Pattern Query
W3C	World Wide Web Consortium
XML	eXtensible Mark-up Language
=>	"is an ancestor of" relation
->	"is the parent of" relationship
/	Root element
/	Child axis
//	Descendant axis
BNF	Backus Naur Form

CHAPTER 1

INTRODUCTION

In this thesis, we study the problem of testing the satisfiability of some XPath queries with node identity constraints. A query is satisfiable if there exists a database of XML documents that will produce a non-empty answer to that query. Research has shown that in many cases the decidability of whether a query is satisfiable or not is NP-hard (Hidders 2003) (Geerts and Fan 2005), and in some cases it is actually undecidable (Benedikt, Fan et al. 2005). We will focus on special cases where the problem is decidable and solutions can be found in polynomial time. Further, this thesis examines the use of topological sort to determine some conditions, which will result in an XPath query being unsatisfiable.

The queries we consider are XPath 2.0 (Kay 2004) queries that involve only child and descendant relationships amongst the elements of the XML document. We will examine how Node Identity Constraints (NICs) impose structural constraints on the documents; this will help us to determine the satisfiability of the query. Other researchers have used predicate logic to examine satisfiability (Lakshmanan, Ramesh et al. 2004), and also using non-deterministic finite automata (Fernández, Hidders et al. 2004). We present an algorithm based on topological sorting of the query vertices to determine if a query is satisfiable. If a total ordering of the child and descendant edges among the query nodes can be obtained, then the query is satisfiable.

In Chapter 2 we provide examples of XML documents and discuss some of the key features of XML documents and XPath queries. Tree Pattern Queries are defined and examples provided for sample XML documents and XPath queries.

In Chapter 3 the satisfiability problem is examined with examples that highlight specific structural aspects that are evident in unsatisfiable queries. Constraint graphs (Lakshmanan, Ramesh et al. 2004) are used to explain the impact of increasing the number of vertices in a Tree Pattern Query. The impact on satisfiability determination is explored by adding c-edges to a constraint graph and comparing this to the impact of adding d-edges.

In Chapter 4 the algorithm for determining satisfiability for a Tree Pattern Query with a single node identity constraint is presented, first as an outline followed by a formal theorem. Four cases are considered which cover the various possible structures in these special Tree Pattern Queries.

In the last chapter, theoretical implications are considered and the extension of the algorithm to Tree Pattern Queries with multiple Node Identity Constraints is discussed.

CHAPTER 2

XML, XPATH AND XQUERY

XML, eXtensible Mark-up Language, was developed as an open standard for marking up documents by the World Wide Web Consortium (W3C) XML Working Group in 1996 (W3C 1996). Due to its flexibility, it has become widely adopted in the electronic exchange of information across the Internet and also for the long-term storage of electronic information (Lewis, Bernstein et al. 2002). XML is a meta-language that allows a user to define a markup language to describe their data. Data in XML format is both human readable and machine readable.

2.1 XML Documents

Mark-up languages are used to define extra information about text within the same document. For example, chapter and section headings in a book are an example of mark-up information within a piece of writing. The most well known mark-up language is HTML, where the tags are used to provide additional information about how a web browser should display the data content. By using XML, a user can assign meaning to their data within the document itself. Each piece of information is surrounded by a description of its meaning. XML is particularly suited to documents which often contain semi-structured content. This has resulted in XML documents becoming a popular format for representing, storing and sharing information by many different computer applications. XML documents can be accompanied by optional schema documents for type definitions and other constraints; however this thesis focuses on XML documents and databases without the presence of schemas.

```
<?xml version="1.0" ?>

    <StudentList Date="2006-04-15">
        <Student>
                <Personal>
                        <Name>
                                <FamilyName>Aardvark </FamilyName>
                                <GivenName>Alexander</GivenName>
                                <GivenName>Stuart</GivenName>
                        </Name>
                        <StudentID>123-456-789</StudentID>
                        <Address Type="Mailing" Status="Confirmed">
                                <AddressLine>P.O. Box 12</AddressLine>
                                <AddressLine>Any Town, VIC 3130</AddressLine>
                        </Address>
                        <Address Type="Residential" Status="Previous">
                                <AddressLine>12 Some Street</AddressLine>
                                <AddressLine>Any Town, VIC 3130</AddressLine>
                        </Address>
                </Personal>
                <Courses>
                        <Semester ID= "Spring 2006">
                                <Course>
                                        <Code>CS311</Code>
                                        <Instructor>
                                                <FamilyName>Zebra></FamilyName>
                                        </Instructor>
                                </Course>
                                <Course>
                                        <Code>MA412</Code>
                                        <Instructor>
                                                <FamilyName>Smith></FamilyName>
                                        </Instructor>
                                </Course>
                                <Course>
                                        <Code>CH101</Code>
                                        <Instructor>
                                                <FamilyName>Brown></FamilyName>
                                        </Instructor>
                                </Course>

                        </Semester>
                        <Semester ID= "Fall 2005">
                                <Course>
                                        <Code>CS211</Code>
                                </Course>
```

```
                    <Course>
                        <Code>MA333</Code>
                    </Course>
                </Semester>
            </Courses>
        </Student>
        <Student>
            <Personal>
                <Name>
                    <FamilyName>Zebra </FamilyName>
                    <GivenName>Zoe</GivenName>
                </Name>
                <BirthDate>1993-05-05</BirthDate>
                <Address Type="Email">zzebra@cox.net</Address>
            </Personal>
        </Student>
    </StudentList>
```

Figure 1. Example of an XML document.

Figure 1 shows an example of an XML document for a list of students. Notice how the document is self-describing in that the name for each piece of data is immediately adjacent to the actual data content. Each piece of data is enclosed by opening and closing tags of the form <tag>data</tag>. The combination of the opening tag, optional data, and a closing tag is called an “element” of the XML document. Other data with its tags can be nested within another element; however the opening and closing tags of the elements must be properly nested. A nested element cannot extend beyond its outer enclosing element, for example the <*FamilyName*> tag is nested properly within the <*Name*> tag which is properly nested within the outer element of <Personal>, because the opening tag <*FamilyName*> is after the opening tag <Name> and the closing tag

</FamilyName> is before the closing tag *</Name>*. There is one root element for the document and all other elements are nested within the root element.

The data is called semi-structured because each element can repeat a varying number of times or be absent altogether. A student can have a varying number of given names, they can have a variety of different type of addresses with varying lines of address information. Similarly, a student can be enrolled for courses for different semesters. Some students may not be enrolled in any courses, particularly if they are new at the University.

Some data is included inside the opening tag itself, and this type of data is called an Attribute. The *Date*, *Type* and *ID* information above are examples of attributes. Attributes usually contain information about the element itself, in well-designed XML documents. The *Address* opening tag has an attribute '*Type*' which holds information about the nature of the address instead of direct address data.

Well-formed XML documents contain:

- one or more elements identifying the data which they surround with opening and closing tags
- an unique root element
- properly nested elements
- no repetition of attributes within one opening tag, and
- attribute values are enclosed in quotes.

A formal data model is used to define XML, and in this data model an XML document is represented as an ordered, labeled tree of nodes, where the nodes represent the elements (Kay 2004). Each node of an XML document tree may have a number of child nodes, but each node, except for the root node has exactly one parent node. The nodes are numbered sequentially so each node has a unique identifier in addition to the tag-name of the XML element. Tag-names can be repeated throughout the document.

The XML document in Figure 1 can be represented as an XML document tree as shown in Figure 2. Each element, such as *Student*, in the document is represented as a node of the tree, where the tag name is the node name. Nested elements of an outer element are shown as child nodes of the parent element. For example the nested elements *Name, StudentID* and *Address* with the element *Personal* are shown as separate child nodes of the node labeled *Personal*. Edges are used to link the parent node to its nested child nodes to show the hierarchy. The root node, called the document node, of the XML document tree is different from the root element of the XML document. The root node is represented by ***/*** and it has a single child node representing the root element of the document, in this example, *StudentList*. The attributes and textual information of an element are represented as the contents of the node.

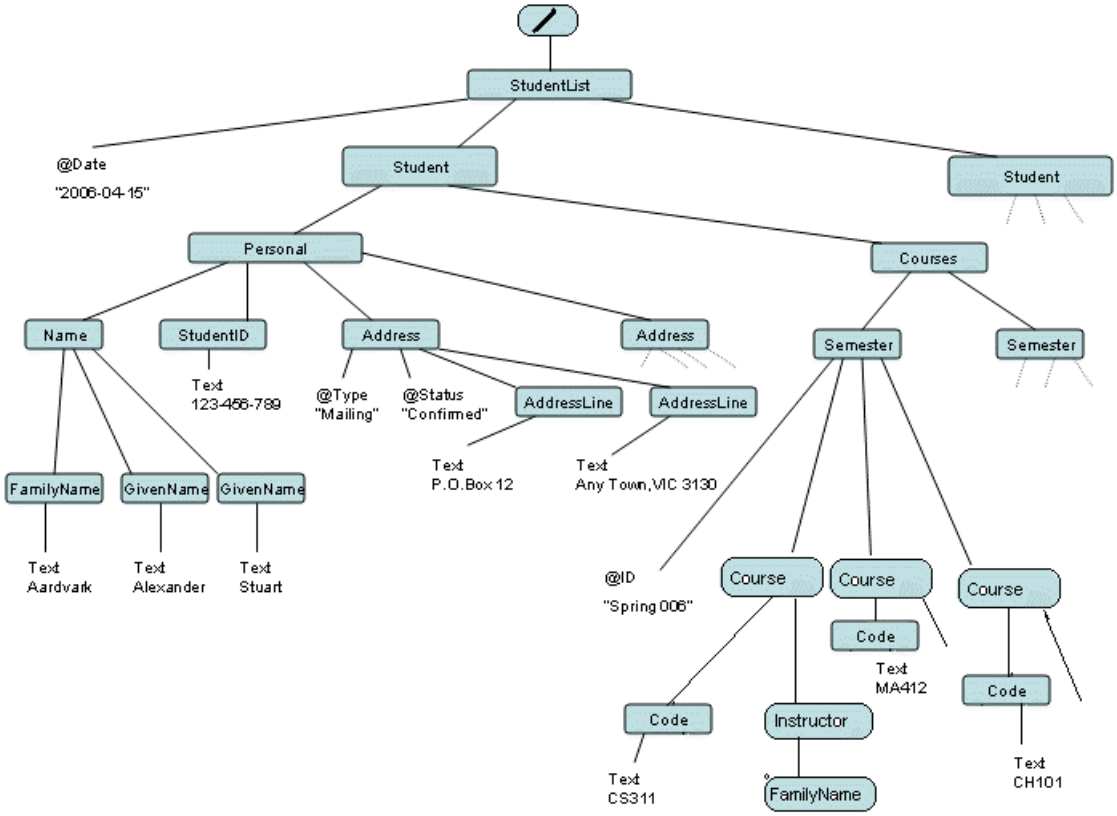


Figure 2. XML document tree.

The XML document tree can be represented by a 4-tuple (Lakshmanan, Ramesh et al. 2004) T = (N, P, r, λ), where

N is the set of element nodes

P represents the parent-child relationships

r is the root element of the document, and

λ is the labeling function which assigns a unique tag name to each node

All nodes of the tree have a parent node except for the root node. For an element or attribute node $x \varepsilon D$, $\gamma(x)$ denotes its tag-name. The special root node of the document, *root(D)*, does not correspond to any element of the document and γ *(root(D)) = /*.

2.2 XPath

XPath is a query language for selecting a set of nodes in an XML document. The XPath data model views an XML document as a rooted tree of nodes, as in section 2.1. The input to an XPath expression is an XML document tree and the output is a set of nodes of the tree.

An XPath query *Q* consists of a sequence of location steps, $Q = L_1 L_2 \ldots L_n$. Each location step L_i specifies an axis, node test, and predicates,

L_i = <axis> <node test> <predicates>

XPath provides thirteen relative axes which are defined for a given context node:

- *self, attribute, namespace*,
- *child, descendant, descendant-or-self*,
- *parent, ancestor, ancestor-or-self*,
- *preceding-siblings, following-siblings*,
- *preceding* (nodes that end before the context node) and *following*.

The most common axes used are child and descendant, which are represented by / and // respectively in XPath expressions. In this thesis, only these two axes are considered. Node test is a test on the name or type of the element.

We consider a subclass of XPath 2.0 called Conjunctive XPath (*C XPath)* consisting of queries, where each *predicate* is either an *and* of predicates, or a relative query. This class of queries is defined by the following grammar in Backus-Naur Form:

<query> ::= <loc_step> | <loc_step> <query>

<loc_step> ::= <axis> <node_test> <predicates>

<axis> :: = / | //

<node_test> ::= elem_label | *

<predicates> ::= Є | [<predicate>]

<predicate> ::= <predicate> and <predicate> | . <query>

elem_label belongs to the alphabet Σ of tag names; .<*query*> indicates a relative query; * is the wildcard label that matches any tag name; *axis* is either / and // which correspond to child or descendant axis respectively. Let *axis(L_i), nodeTest(L_i) and predicate(L_i)* denote the axis, node test and predicate in step L_i, respectively.

XPath expressions can either be absolute path expressions, which always start the navigation at the root of the document tree, or relative path expressions which start the navigation from the current node which is called the context node. For example, //Address will select all *Address* elements in the document. In comparison, the expression *.//Address* will select all *Address* elements that are descendants of the context node, while the expression *Address* will return only

the *Address* elements which are children of the current node (Lewis, Bernstein et al. 2002).

In an XPath expression, every step evaluates a sequence of nodes. For example, */doc/bib/book* will find all book nodes within all 'bib' nodes in a document with root element "doc". The data model also defines an ordered collection of nodes, called a NodeList, for the traversal of the list of nodes. Using the NodeList, nodes can be located using path expressions.

In this example of an XPath expression,

//Student//Address[@status = "Confirmed"]/AddressLine

the node *Address* is a descendant node of the *Student* nodes, and *AddressLine* is a child node of the *Address* nodes. The query will search for *Address* nodes with an attribute of *status = "Confirmed"* and return the *AddressLine* nodes immediately below those *Address* nodes.

2.3 Tree Pattern Queries

(Amer-Yahia, Cho et al. 2001) proposed a model to represent a C Xpath query in the form of a tree-shaped pattern. A query $Q \varepsilon$ C XPath can be represented by a tree *tree(Q)* = (*V*,*A*) where *V* is a set of vertices and *A* is a set of arcs (Ramanan 2003). Each vertex $v \varepsilon V$ has a tag $\gamma(v) \varepsilon \Sigma \cup \{/, *\}$ associated with it. $\gamma(v)$ is the element name associated with *v*; / is the tag of *root*(*Q*), and * denotes the wildcard tag. Each arc $a \varepsilon A$ is either a child arc (*c-arc*) or a descendant arc (*d-arc*), corresponding to a *child* or *descendant* axis in *Q*,

respectively. In our figures, *c-arcs* and *d-arcs* are represented by single lines and double lines, respectively.

For example, Figure 3 shows *tree(Q)* for *Q = //a[b and .//c]/ * [a and .//b].* The vertex of *tree(Q)* that corresponds to the node test in the last location step in *Q* is called the *output vertex* of *Q*, and is denoted by *opv(Q)*; it is marked by a # sign in Figure 3 (a). For an arc *r = (u, v)*: If *r* is a *c-arc*, we say that *v* is a *c-child* of *u*; if *r* is *d-arc* , *v* is a *d-child* of *u*.

In general, |*tree(Q)*| is linear in |*Q*|. From now onwards, we will not distinguish between *Q* and *tree(Q)*. To minimize confusion the terms *vertices* and *arcs* will be used when referring to components of *Q*; and *nodes* and *edges* to refer to components of *D*, an XML document.

For a vertex $u \in Q$, let Q_u denote the subtree of *Q* that is rooted at *u*. For a node $n \in D$, let D_n denote the subtree of *D* rooted at *n*. An embedding Γ of Q_u in D_n is a mapping from the vertices of Q_u to the nodes D_n, that satisfies the following conditions:

1. Preserve vertex types: For each vertex *v* in Q_u:
   - If $\gamma(v) = /$, then $\Gamma(v) = root(D)$. In this case,

     $v = u = root(Q)$ and $n = root(D)$.
   - If $\gamma(v) \in \Sigma$, then $\gamma(\Gamma(v)) = \gamma(v)$.
2. Preserve arc types: For each vertex *v* in Q_u:
   - If v´ is a *c-child* of *v*: $\Gamma(v')$ is a child of $\Gamma(v)$ in *D*.
   - If *v´* is a *d-child* of *v*: $\Gamma(v')$ is a descendant of $\Gamma(v)$ in *D*.

The output of *Q* on *D* is *Q(D)* = { Γ (*opv*(*Q*)) | Γ is an embedding of *Q* in *D*}. The answer to the query is the set *Q*(*D*) of nodes that result from all the possible ways of "embedding" the tree pattern into the database. For example, using the XML document from Figure 1, a query to find for all students only the AddressLines for Addresses which have a status of "Confimed", can be represented by the XPath query

//Student//Address[@status = "Confirmed"]/AddressLine

and as a tree pattern as shown below in Figure 4. Q(D) =

{ <AddressLine>P.O. Box 12</AddressLine>,
<AddressLine>Any Town, VIC 3130</AddressLine>}

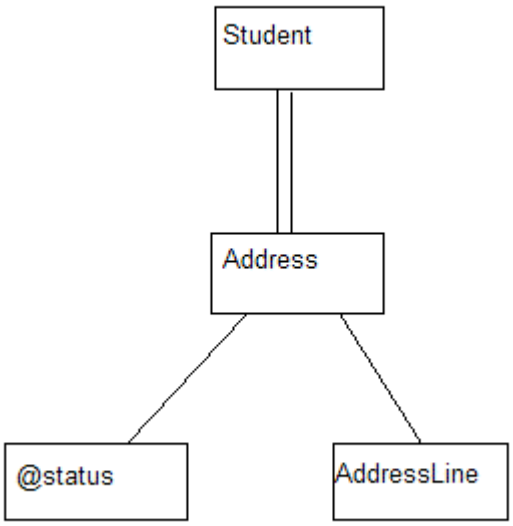


Figure 3. Tree Pattern Query.

2.4 XQuery

XQuery is a declarative language that can be used to formulate queries for both XML documents and XML databases. Declarative languages state what needs to be computed instead of how it is to be computed. XPath is a subset of XQuery, which is a more powerful query language. XQuery, like XPath, uses a tree-structured data model for the XML data and for navigation.

XQuery was developed by the World Wide Web Consortium (W3C) XML Working Group to be a concise and flexible query language for XML (Brundage 2004). Structured Query Language, SQL, was developed to query relational data, unordered sets of "flat" rows of data. In contrast, XQuery focuses on ordered sequences of values and hierarchical nodes and supports node identity. XQuery can construct temporary XML results within a query. New XML documents can also be constructed using XQuery.

The central expression in XQuery is the FLWOR (For, Let, Where, Order by, and Return) expression. Simple FLWOR expressions can be expressed by using XPath expressions to navigate the NodeList using the relative axes.

As an example of a FLOWR expression, consider the XML document from Figure 1 and the following query, which will produce a document of student names who took CS311 in the Spring 2006 semester. The document will be sorted by the student's last name.

FOR $s IN doc("http://www.wsu. Studentlist.xml")//Student

LET $c := $s//Semester[ID="Spring 2006"]/Course

WHERE $c/Code = "CS311"

```
ORDERBY ($s//FamilyName)
RETURN
    <StudentSummary
        LastName=$s//FamilyName
        FirstName=$s//GivenName
      />
```

2.5 Tree Pattern Queries with Node Identity Constraints

Tree Pattern Queries with Node Identity Constraints arise in XQuery. XQuery provides the operator '*is*' to determine whether two nodes have the same identity or not. Tree Pattern Queries with node identity constraints can be represented as Structural Constraint Graphs, where the node pairs with identity constraints are represented as a single node.

In a tree and therefore in an XML document, a node can have only one parent. In a tree pattern query, paths are represented with child edges and descendant edges. For example, suppose node *x* has 2 descendants *e* and *f*. Then both *e* and *f* share a common ancestor *x*. There are 2 possibilities, either *e* and *f* are on the same path from node *x*, or they are cousins. If *e* and *f* share a common ancestry path (or pedigree), it could be *x* => *e* => *f* or it could be *x* => *f* => *e* where => is used to represent "is a descendant of" relation.

In the other possibility, "cousins" share a common ancestor but are the descendants of sibling nodes. Sibling nodes are child nodes of the same parent node and so "cousin" nodes cannot occur on the same common path. Therefore

nodes *e* and *f* do not lie on the same path. If *e* and *f* are cousin nodes, then a query with a node identity constraint on them is unsatisfiable.

The least common ancestor of two nodes *e* and *f* in a TPQ *T* is the node *x* that is an ancestor of both *e* and *f* and that has the greatest depth in *T* (Bender, Pemmasani et al. 2001). Therefore any query that requires two nodes to be identical but the nodes do not share a common path from their least common ancestor is not satisfiable (Lakshmanan, Ramesh et al. 2004).

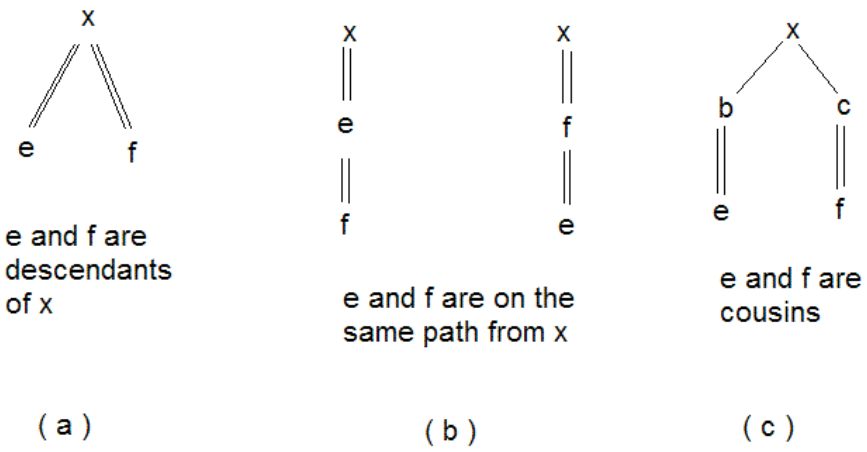


Figure 4. Two examples of descendant nodes

Consider the following XQuery in Figure 4 from (Lakshmanan, Ramesh et al. 2004):

```
FOR $a in document("doc.xml")//a,
        $e IN $a/b//e,  $f IN $a/d//f,
        $c IN $a//c, $e1 IN $c//e, $f1 IN $c//f
WHERE $e = $e1  AND  $f = $f1
RETURN {$a}
```

The corresponding TPQ with two node identity constraints is shown in Figure 5. The query can be represented by the structural constraint graphs in Figure 6 (a) and (b). The pairs of node identity constraints are represented by a single vertex for each NIC as shown in Figure 6(a). However the Structural Constraint Graph for this TPQ may alternatively be represented by Figure 6(b).

Vertices *B* and *C* are both ancestors of vertex *E* and also descendants of vertex *A*, but *B* is a child of *A*. This implies that vertex *C* must be a descendant of vertex *B*. Similarly vertex *C* is also a descendant of vertex *D*. Therefore the implicit constraints on vertex *C* may be represented more clearly by explicitly showing vertex *C* as a descendant of vertex *B* and also as a descendant of vertex *D*.

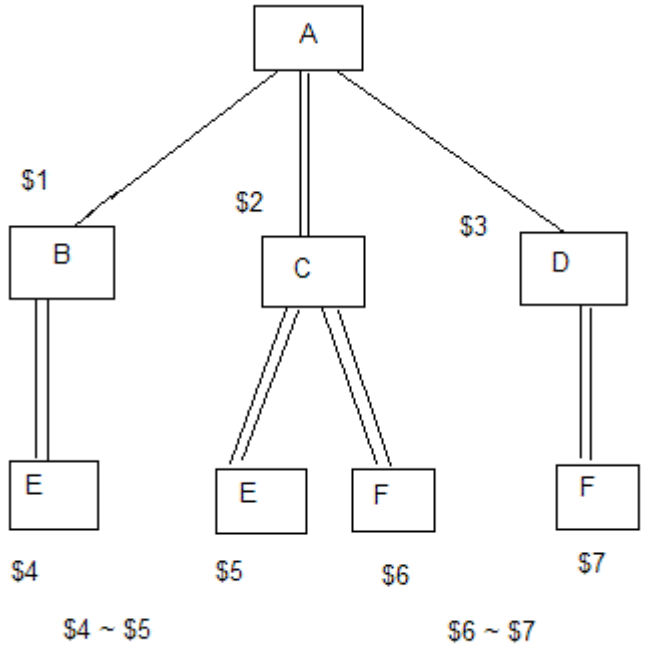


Figure 5. Tree Pattern Query from (Lakshmanan, Ramesh et al. 2004).

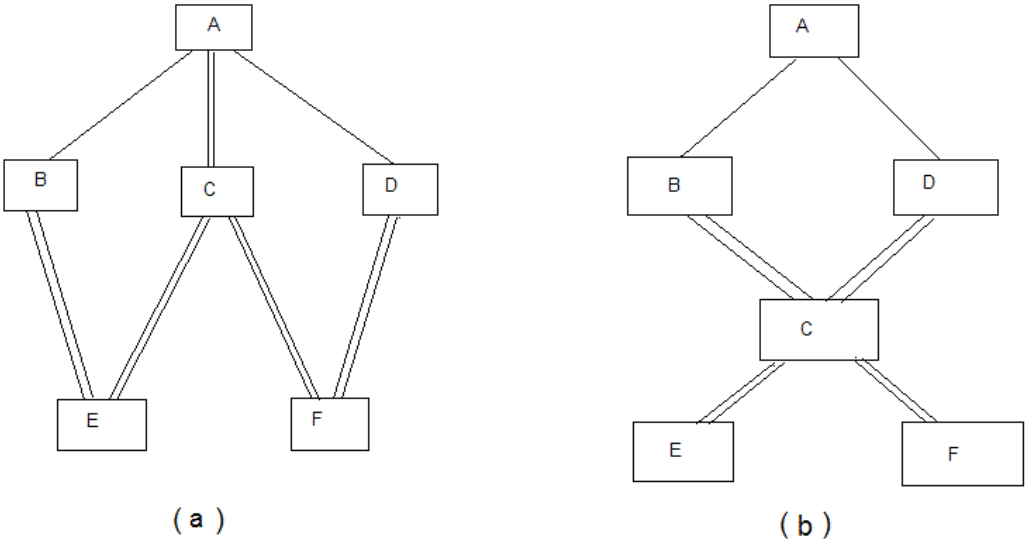


Figure 6. Structural Constraint Graphs for TPQ of Figure 5.

Vertex *A* has child vertices *B* and *D* and so *B* and *D* are "siblings". Therefore *B* and *D* cannot share any common descendants of *A* as this would require the descendants to lie on the same path. However vertex *C* is a common descendant of both *B* and *D*. Therefore this query is unsatisfiable.

2.6 Topological Sort of Directed Acyclic Graphs

A directed graph, *G = (V, A)*, where *V* is a set of vertices, and *A* is a set of arcs. An arc is represented by a tuple *(u,v)* where *u, v* are elements of *V*. In a directed graph, the first vertex in the arc is the start of the arc, and the second vertex is the end of the arc. Therefore in a directed graph, arc *(u,v)* is different from arc *(v,u).* A directed graph is acyclic if it contains no cycles, that is, it is possible to produce a linear ordering of the vertices in *G* consistent with the total order $u < v$ if *(u,v)* is an element of *A*. Any such ordering is a topological sort. For any directed acyclic graph, there will be one or more topological sorts. The topological sort of *G* can be found using a variation of the Depth First Search algorithm (Cormen, Leiserson et al. 2001).

Tree pattern queries can be represented by a special type of directed graph, Structural Constraint Graphs, where there are two types of arcs representing the child and descendant relationships of the query. For the purposes of a topological sort, descendant arcs behave the same as directed arcs in a graph, and it is possible that other vertices may appear in the linear

ordering between the two vertices of the descendant arc. However a child arc means that the second vertex of the arc must appear immediately after the first vertex of a child arc in a linear ordering. Standard topological sort algorithms cannot be used on directed graphs with child arcs. For Graph 1 shown in Figure 7, there are many possible linear orderings, such as ABDFCEGH, ABCDEFGH etc. However for Graph 2, which has both *c-arcs* and *d-arcs*, there is only one possible linear ordering which is ABCEGDFH.

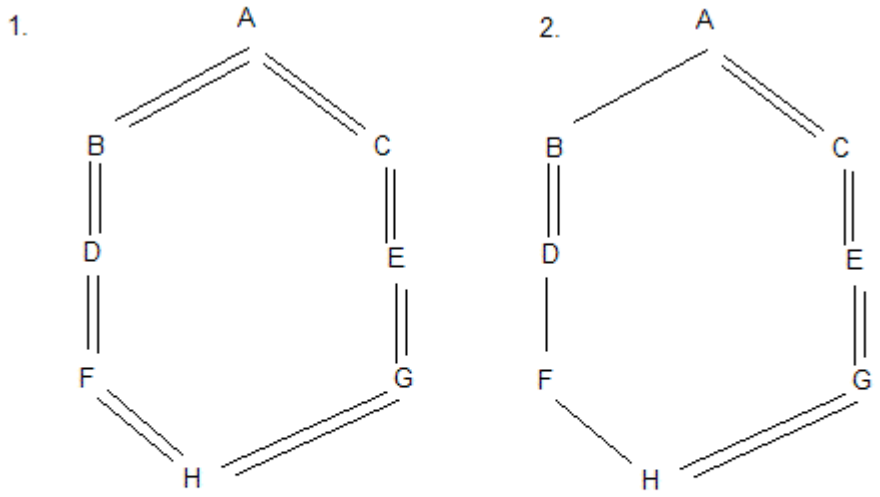


Figure 7. Directed Acyclic Graphs.

CHAPTER 3

THE SATISFIABILITY PROBLEM

Increasingly, more electronic information is managed by XML or stored as XML documents. In part this is due to the adoption of XML for data exchange between organizations and the use of XML by archival agencies for the long-term storage of digital information (Quenault 2004). When terabytes of XML documents need to be searched, it is essential that queries are optimized.

A key step in query evaluation is to determine if there is an answer to the query before running the query against a database. A query is satisfiable if there is a database that has a non-empty answer. (Lakshmanan, Ramesh et al. 2004) showed that a satisfiability check can make substantial savings in query evaluation. Some queries are unsatisfiable because there is no database that can return a non-empty answer to the query. In these cases, if the query is determined to be unsatisfiable, then the query does not need to be evaluated against the given database.

This chapter examines some conditions which will result in the query being unsatisfiable. First we examine the complexity of deciding whether a query is satisfiable. Next we examine some structural constraints on a query that if broken causes a query to be unsatisfiable. In the next chapter we present an efficient algorithm to test for satisfiability in some special cases.

3.1 Complexity of Deciding Satisfiability

(Hidders 2003) examined the problem of deciding the satisfiability of XPath 2.0 expressions. He showed the problem of deciding the satisfiability of XPath expressions is NP-hard when all axes are allowed and predicates may contain set intersection, set union, and set difference. Subsequent researchers have examined special situations where the problem of deciding satisfiability may be more tractable.

Although XML data is semi-structured, an XML document is often associated with a set of rules, a grammar. These rules impose structural constraints on the database and may be represented by a Document Type Definition (DTD) or an XML Schema. The presence of a DTD or XML Schema provides an additional factor in determining the satisfiability of an XPath expression. In the presence of recursion in the DTD, (Benedikt, Fan et al. 2005) showed that it is undecidable to determine if an XPath expression with negation is satisfiable.

XPath queries can be constructed using vertical axes, such as child, descendant, parent, and ancestor axes, which are similar to file path navigation in many operating systems. However XPath queries can also be constructed to access the order of XML data. These types of queries, using horizontal or siblings axes, allow access to the ordered sequence of the nodes. This makes it possible to construct queries such as find the third course taken by students, or find the second author of books. In the absence of DTDs, (Benedikt, Fan et al. 2005) showed that for queries with vertical axes and without negation, it was possible to test for satisfiability in time ($O\ |Q|^3$). In contrast, (Geerts and Fan

2005) found that for queries with sibling axes and without a DTD, that satisfiability is undecidable.

3.2 Unsatisfiability Due To Structural Constraints

Given the difficulty of deciding satisfiability, we shall focus our examination on queries which contain only descendant and child axes. Queries with negation are excluded from this study. These queries shall be considered in the absence of DTDs and XML Schemas.

There are 3 constraints that can be represented in a Tree Pattern Query: tag, value and node identity. Tag constraints are used when searching for particular types of nodes, for example author nodes in a book database. Value constraints are used for searching for particular content of a node, for example books with date published after a particular year. Node identity constraints are particularly common in XQuery FLOWR expressions (see Figure 4) when searching for nodes that have several conditions in common. Node Identity Constraints are similar in concept to join conditions in SQL queries. In TPQ, two nodes, b1 and b2, have a Node Identity Constraint, NIC, when there exists one or more paths from an ancestor node, x, to the two nodes, b1 and b2 which are equated by the node identity constraint. The nodes must have the same tag.

The interaction between these various constraints can be evaluated to determine if a query is satisfiable or not. Queries with just tag and value constraints are satisfiable, providing the value based constraints are consistent and in the absence of a DTD or XML Schema (Lakshmanan, Ramesh et al.

2004). Queries that contain Node Identity Constraints may be satisfiable under certain conditions.

Violations of node identity constraints, tag constraints, and value-based constraints can make a query unsatisfiable, through a pair of conflicting predicates. Examples are

(a) node x is identical to node y and node x is not identical to node y

(b) node x is the ancestor of y and y is the ancestor of x, referred to as a cycle

(c) node x and node y are on the same path and node x and node y are cousins

(d) node x is a child of node y and node x is a child of node z (a node cannot have 2 parents).

The conflicts in examples (a) and (b) are self-evident.

Example 1 – Violation (c)

//x[b//d = c//d] is an example of a query that is unsatisfiable. This is because the query is asking for 2 distinct children of node x that have a common descendant.

Let NIC-node be used to refer to the node with the node identity constraint, in this example this is the node with tag name D. In a constraint graph, each NIC-node will have at least 2 incoming arcs. There will be a vertex which is the start of these incoming paths that end with the NIC-node, and this vertex will be the least common ancestor of the NIC-node. In the constraint graph above, X is the least common ancestor of the NIC-node. In the XPath query, this node is called the context node. In general, in a constraint graph for a query, where the least

common ancestor of the NIC-node has 2 or more child arcs that form paths to the same NIC-node, the query will be unsatisfiable.

There is one special case where it is possible for the query to be satisfiable. Consider the query,

//x[c//f//d = c//d],

where *d* is the NIC-node and x is the context node. Due to the way the query is written, the references to node *c* are referring to all nodes with tag *c* which have a parent with tag *x*. This is because *c* occurs inside the square brackets. If *c* was the context node of the two d nodes then the query would be written as

//x/c[.//f/d = .//d]

and this query is satisfiable.

Therefore in the special case above, the query could be satisfied if the two child nodes of *x* were in fact the same node. This is possible only if the nodes have the same tags which they do in this special case.

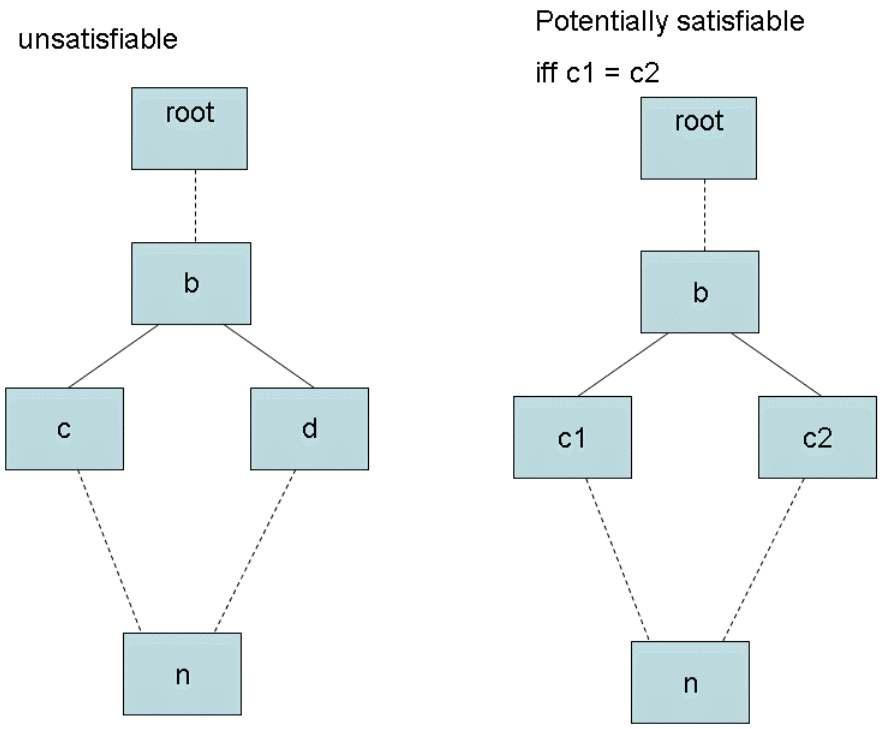


Figure 8. Nodes cannot be on the same path and also be cousins.

Example 2 – violation (d)

//a[.//b/d = .//c/d] is an example of a query that is unsatisfiable, as it requires *d* to be the child of two different nodes, which is not possible in an XML document.

In general, any query is unsatisfiable where the NIC-node has two or more parents. There is one special case where it is possible for the query to be satisfiable. Consider the query,

//a[.//b//c/d = .//f//c/d],

where d is the NIC-node with 2 parents. However in this particular query, it would

be possible for the query to be satisfied if both the parent nodes were in fact the same node.

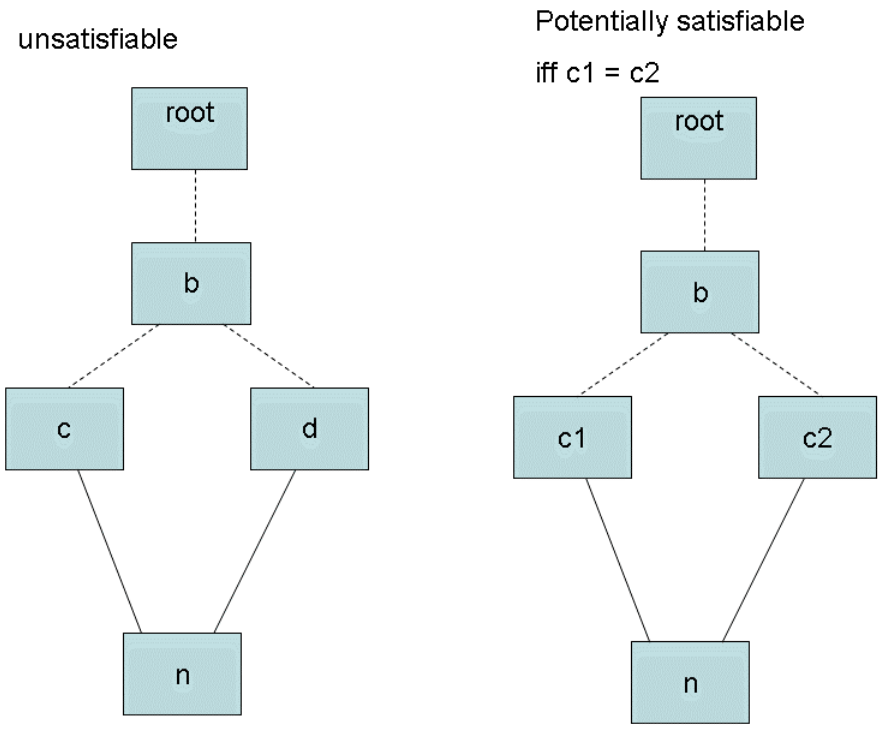


Figure 9. Nodes cannot have 2 parents.

Example 3 – complex violation of (c)

//a[/b/c/d/e/f = //g//f] is another example of a query that is unsatisfiable, as the path from *a* to *f* (the pedigree) is completely defined by the child edges and it does not contain *g*.

Consider the query *//a[b/c/d/e/f = .//c/d//f]* which can be satisfied if the nodes *c* and *d* on both paths are the same nodes.

In general, any query is unsatisfiable where one path in the query contains only child edges from the least common ancestor to the NIC-node, unless it is possible to "map" or embed the other path(s) onto the path of child edges. Therefore in cases where the length of the path with descendant edges is shorter than or equal to the length of the path with all child edges, it may be possible for the query to be satisfiable in the following situation.

Let the sequence of nodes defining the full pedigree, *P*, (that is, the path containing all the child edges) be written as a string starting with the first node, followed by the next node, etc until the string is finished with the node with the Node Identity Constraint. (NIC). So *a/b/c/d/e/f* would be written as the string *abcedf*.

Let the other path, *Q*, be written as the string containing all the nodes on the other path to the NIC, such that where there is a child edge, the parent node is followed by the child node tab, so *a/b* would be written as *ab*, and where there is a descendant edge the wildcard * is inserted into the string, so *a//b* would be written as *a*b*, where the wildcard can match 0 or more nodes from string *P*. Let the length of path *P* be represented by |p| and the length of path *Q* by |q|. If |p| < |q| then the query is unsatisfiable. However if |p| >= |q| and q is a substring of p then the query is satisfiable. There are various algorithms for determining if a string is a substring of another string, (Stephen 1994).

Example 4 – satisfiable queries

//a[b//d//e = .//f//e] is a query that can be satisfied as node f can be embedded in the path either between *b* and *d* or between *d* and *e*. The query can be rewritten as

//a/[b[//d//e = //f//e]] and queries in this form are satisfiable.

Example 5 – cycle in a TPQ

Query that results in a cycle, for example the following FLWOR expression from (Lakshmanan, Ramesh et al. 2004), will be unsatisfiable.

```
FOR $a in document("doc.xml")//a
    $e IN $a/b//e, $f IN $a/d//f,
    $c IN $a//c, $e1 IN $c//e, $f1 IN $c//f
WHERE $e = $e1 AND $f = $f1
RETURN {$a}
```

Figure 4 shows the tree pattern query and the constraint graph for this FLOWR expression. The constraint that $e = $e1 requires a/b/c to lie on the same path to node e, and similarly $f = $f1 requires a/d/c to lie on the path to node f, thus requiring node *a* to have a child with two different labels. Different representations of the constraint graph will also contain a cycle, which is a violation of condition (b) which makes the query unsatisfiable.

Example 6 – satisfiable query

//a[.//b//c//d/e = .//f//e] is a query that can be satisfied as node *f* can be embedded in the path as an ancestor of *d*. In general queries of this form, with only descendant edges and no child edges, can be satisfied. A constraint graph with only descendant edges is comparable to a directed acyclic graph. A total ordering, or a topological sort, of the nodes of the TPQ can be matched directly to at least an XML document that contains nodes in the same sequence as the result of the topological sort of the constraint graph. Therefore there is at least one database which will satisfy the query, so the query is satisfiable.

3.3 Determining Satisfiability from Structural Constraint Graph

In this section we use the structural constraint graphs for minimized tree patterns as described by (Lakshmanan, Ramesh et al. 2004), but with a difference. Nodes with node identity constraints are shown as the same node, which allows a DAG to be used instead of the more specialized tree. XML queries that initially appear to be very similar can vary in terms of satisfiability. We shall examine various structural constraint graphs and compare the ability to produce a topological sort of the DAG with *c-arcs* and *d-arcs*, with the satisfiability of the query with reference to the structural constraints described in 3.2. In terms of the constraint graph, these constraints can be redefined to the following rules:

(a) a vertex can have only one parent but many ancestors

(b) constraint graph does not contain a cycle

(c) if a path between two vertices contains only child arcs, then any alternative path must be able to be “embedded” into that c-arc only path.

In the case of a single vertex, a topological sort is possible and the query is satisfiable. Similarly in the case of 2 vertices, a topological sort is possible and the query is satisfiable. Now to consider some more interesting cases.

3.3.1 Constraint Graph with 3 vertices

There are only 3 possible constraint graphs (ignoring isomorphic graphs) consisting of 3 vertices and 2 arcs as shown in Figure 10. Note that the direction of the arcs is assumed by the relative hierarchy of the nodes, that is all arcs are in the downward direction. Single lines represent *c-arcs* and double lines represent *d-arcs*.

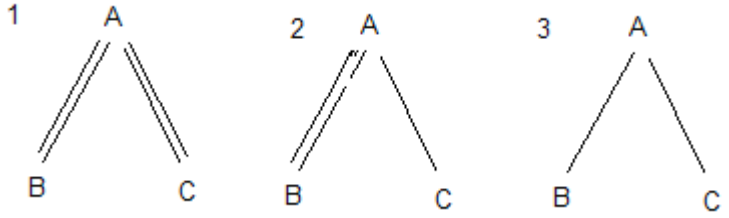


Figure 10. Constraint graphs with 3 vertices and 2 arcs.

We will not consider graphs with fewer arcs than vertices, as we are interested in exploring the impact of node identity constraints on the satisfiability of queries. By drawing the constraint graph with only one vertex to represent the

node with the node identity constraint, then the node representing the NIC node will have at least 2 incoming arcs. Any nodes that are not on the path the NIC nodes are effectively “pruned” from the constraint graph as they have no impact on the satisfiability of the query.

If we consider 3 nodes and 3 edges in the TPQ, there are 4 graphs which represent satisfiable queries (Figure 11) and 4 graphs that represent unsatisfiable queries (Figure 12). In the case of the graphs representing the unsatisfiable queries, it is not possible to produce a topological sort of the graph. The most noticeable point of difference is that the constraint graphs of the satisfiable queries all have a *d-arc* between the highest vertex and the lowest vertex. Whereas the unsatisfiable queries all have a *c-arc* from the highest vertex to the lowest vertex.

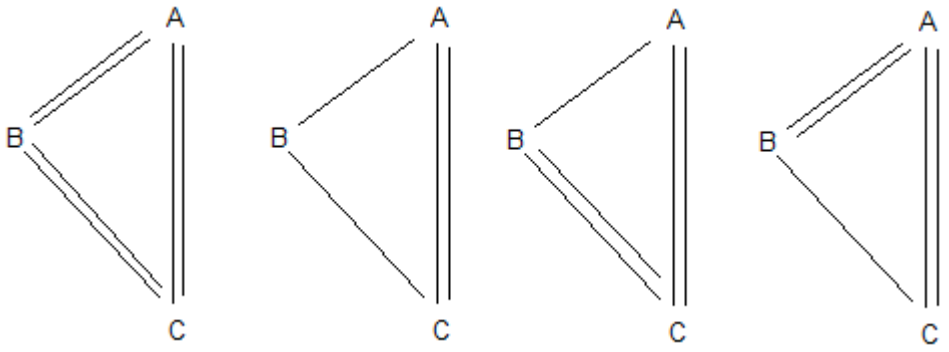


Figure 11. Constraint graphs – 3 vertices, 3 arcs, satisfiable.

If we compare the topological sort to inequality expressions, then a *c-arc* between *b* and *c* can be considered the same as $b < c$ where $c = b + 1$. In contrast, *d-arcs* represent the relationship $b < c$.

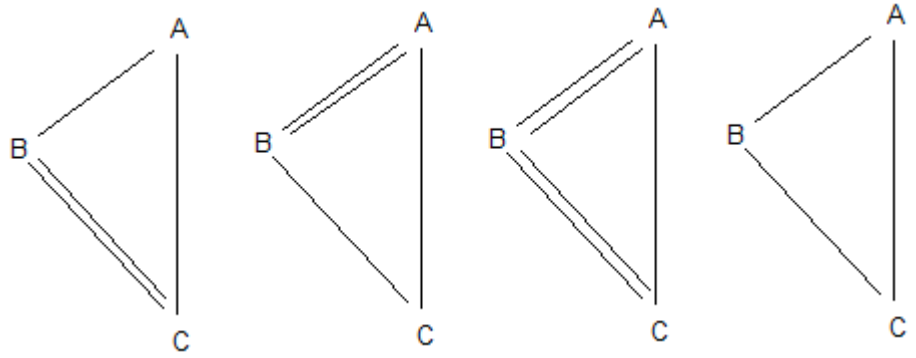


Figure 12. Constraint graphs – 3 vertices, 3 arcs, unsatisfiable.

3.3.2. Constraint Graph for TPQs with 4 nodes

With 4 nodes in a TPQ and one Node Identity Constraint, then it is possible for the restraint graphs to have 4 or 5 arcs. Initially graphs with 4 arcs are examined. Out of the 10 distinct graphs, 4 are satisfiable and 6 are not.

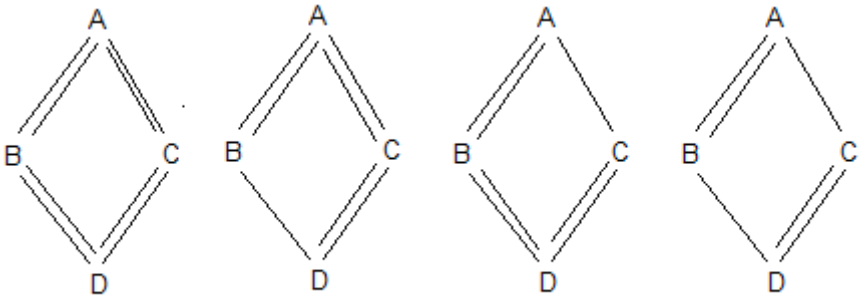


Figure 13. Constraint graphs – 4 vertices, 4 arcs, satisfiable.

Notice that as the number of *c-arcs* increases, then more possible constraint graphs are for queries that are unsatisfiable. Once half the arcs are *c-arcs*, there is only one variation where the query is satisfiable. With the addition of another arc, then it is possible to determine unsatisfiability with the presence of an unsatisfiable sub-graph within the constraint graph.

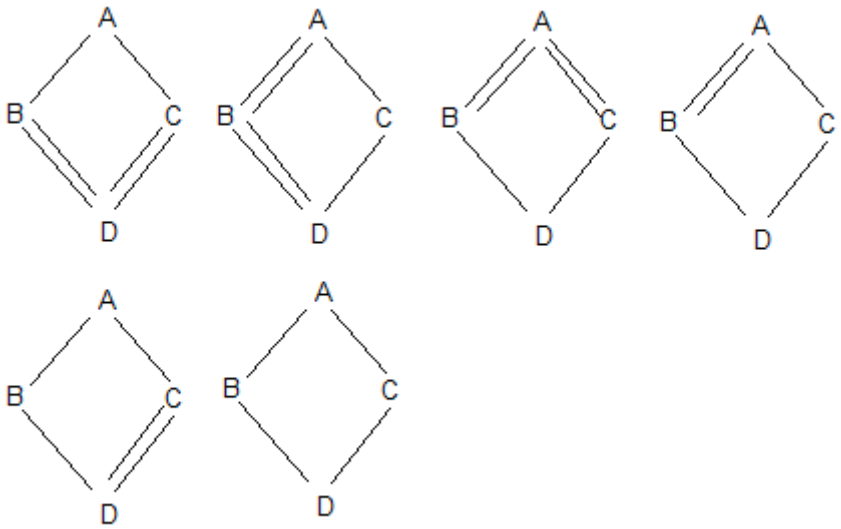


Figure 14. Constraint graphs – 4 vertices, 4 arcs, unsatisfiable.

There are 4 possible ways to add a 5th arc to the constraint graphs of 4 vertices and 4 arcs.

a. Add a *d-arc* from A to D which will not affect the satisfiability of the 4 arc graph
b. Add a *c-arc* from A to D which will make all the graphs unsatisfiable
c. Add a *d-arc* from B to C which will make some previously satisfiable queries unsatisfiable.
d. Add a *c-arc* from B to C which will make all the queries unsatisfiable except for where there are d-arcs from A to C and from B to D.

The first 4 constraint graphs in Figure 15 show that the addition of a new *d-arc* from A to D has no impact on the satisfiability of the query as this arc is redundant as it contains no additional constraints within the query. The new arc does not increase the number of NICs in the constraint graph. Expressed as inequality expressions, we have added a < d to the existing expressions of a < b and b < d, which has added no further information.

However the addition of a “sideways” *d-arc* from B to C does affect the satisfiability of the query. Vertex C and Vertex D represent separate NIC nodes in the constraint graph. Notice that the constraint graph now contains sub-graphs for each NIC. Only 4 graphs represent a satisfiable query with the addition of this additional d-arc. These 4 graphs also contain satisfiable sub-graphs. Alternatively if a *c-arc* is added from Node B to Node C, these same constraint graphs are still satisfiable. In contrast, the addition of a *c-arc* from A to D makes all the queries

unsatisfiable, as shown in Figure 16. This additional c-arc places an additional constraint, namely that the lowest node must be a child of the highest node.

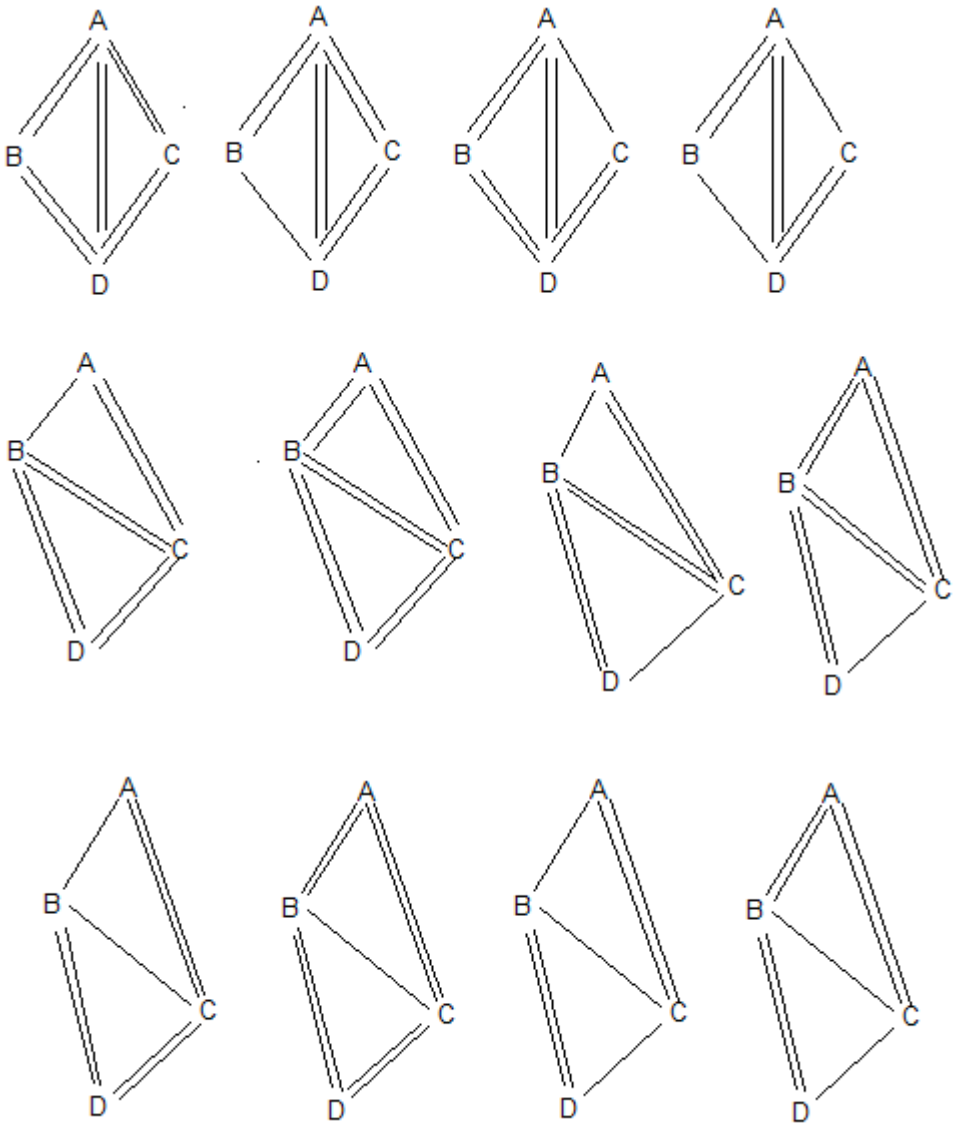


Figure 15. Constraint graphs – 4 vertices, 5 arcs, satisfiable.

The addition of further edges to queries that are already unsatisfiable has no effect on satisfiability. If a topological sort is not possible for a graph, the addition of further arcs cannot make the sort possible.

The unsatisfiable queries with the addition of c-edge from A to D are shown in Figure 16.

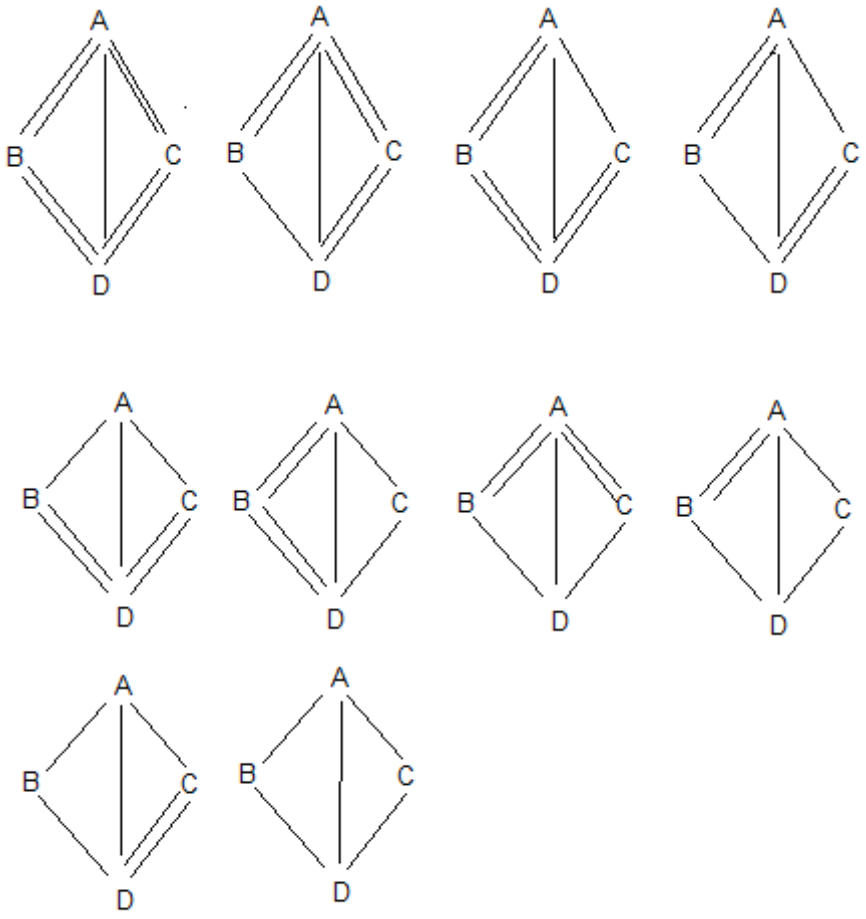


Figure 16. Constraint graphs – 4 vertices, 5 arcs, 1 NIC, unsatisfiable.

The addition of an arc from B to C adds a new separate NIC node to the constraint graph. The last 8 graphs in Figure 15, show the queries that are still

satisfiable after adding a *d-edge* or *c-edge* from *B* to *C*. Figure 17 shows the queries that are satisfiable with 4 edges but are unsatisfiable with an additional edge from *B* to *C*.

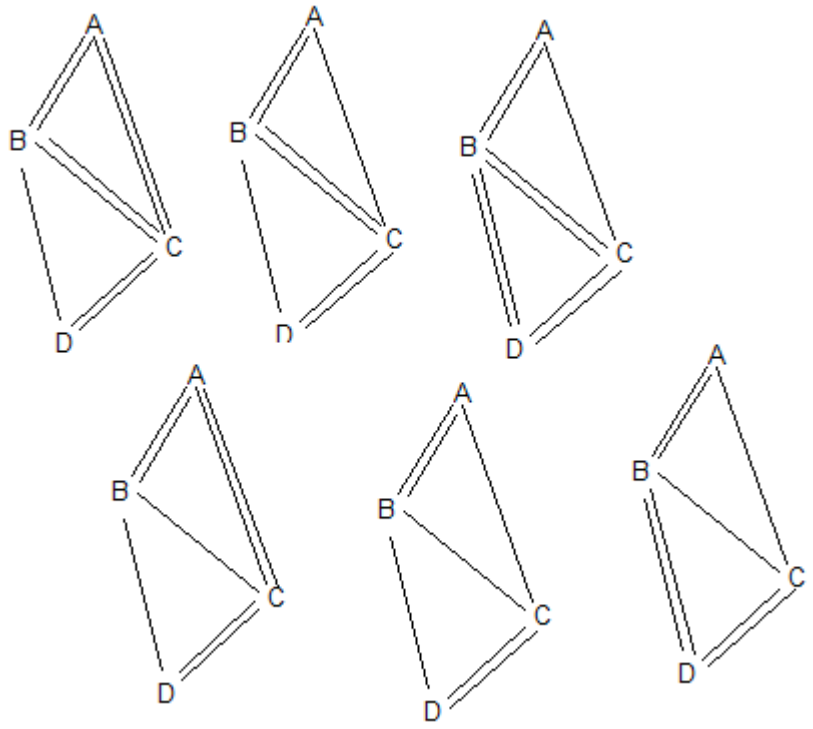


Figure 17. Constraint graphs – 4 vertices, 5 arcs, 2 NICs, unsatisfiable.

The addition of 6th edge can be accomplished in 2 ways: either have 2 or 3 NIC nodes. First we will examine the scenario with 2 NIC nodes and consider the impact on the satisfiable queries shown in Figure 15. We can add a *d-arc* from A to D but we have shown that this arc is redundant as a constraint. Similarly we have previously shown that adding a *c-arc* from A to D makes the queries unsatisfiable, basically because it requires node D to be both a child and a grand-child of Node A at the same time which is not possible.

The addition of a third NIC Node will lead to a cycle in the graph. No ordering of the vertices is possible if there is a cycle in the constraint graph since it means that in the TPQ, node B is a child or descendant of Node C which is also a child or descendant of node B.

3.3.3. Constraint Graph with 6 vertices

With 6 nodes in a TPQ, the number of possible structural constraint graphs is quite large. However there are some general principles that can be applied to assist with determining whether a constraint graph has a topological sort and hence whether the query is satisfiable. If there exists a sub-graph in the query that it unsatisfiable, then the larger query is unsatisfiable. If there is just one NIC in the graph then it is relatively easy to determine satisfiability. If the ancestor of the path has 2 c-edges or if the NIC node has 2 incoming c-edges then the query will be unsatisfiable as either a node has 2 parents, or we are asking for nodes to be cousin and also have the same parent, which is not possible.

More complex issues need to be considered when there are 2 or 3 NICs in the constraint graph. Both of the constraint graphs shown in Figure 18 are satisfiable and also each graph has only one possible topological sort that satisfies the constraints of the c-arcs and d-arcs. For the first graph the only possible ordering is A => C -> E => B -> D => F

For the second graph the only possible ordering is A => B => C -> D => E => F

If the arc from C to D is omitted from both graphs then there are several possible orderings for each graph. Once there is only one possible ordering for a graph, then for the query to remain satisfiable as further constraints are added, it means

that these additional constraints are redundant as they do not add any additional constraints on the query.

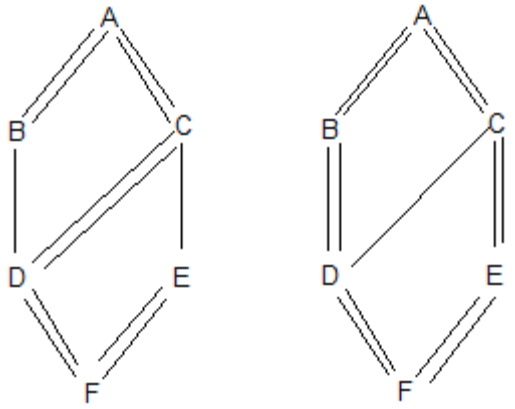


Figure 18. Constraint graphs – 6 vertices, 7 arcs, 2 NICs, satisfiable.

Using the graphs from Figure 18, let us examine the impact of adding another NIC to the constraint graph at node E. For the first graph a d-arc is added from B to E, and for the second graph a c-arc is added from B to E. In both cases the query becomes unsatisfiable and there is no ordering possible of the constraint graph. These graphs are shown in Figure 19.

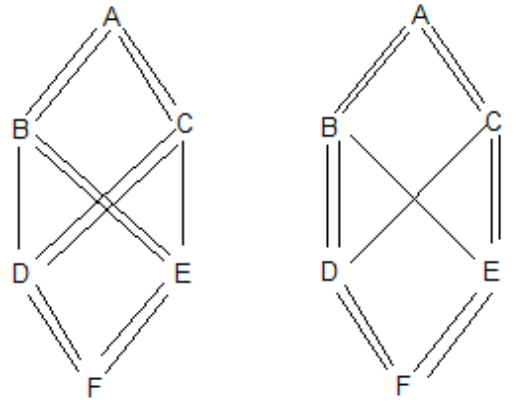


Figure 19. Constraint graphs – 6 vertices, 8 arcs, 3 NICs, unsatisfiable.

3.4 Summary of the Problem

Determining satisfiability of XPath expressions with Node Identity Constraints is a complex problem and in some situations the problem is undecidable. We examined the satisfiability of query expressions in the absence of DTD and XML schemas and in the absence of sibling axes, wildcard and negative expressions within the query. This restricted group of queries has been shown to be have PTIME solutions for determining satisfiability.

We examined conditions that cause a query to be unsatisfiable and compared the related constraint graph and the examined whether it was possible to produce a topological sort of the nodes. We found that iff a topological sort of the constraint graph could be produced then the query was satisfiable. Similarly we found that if it was not possible to produce a topological sort of the constraint graph then the equivalent query was unsatisfiable.

In the next chapter we present an efficient algorithm to test for satisfiability in these special cases of queries without negation and wildcards, with only child and descendant axes and consider these queries in the absence of DTDs and XML Schemas.

CHAPTER 4

ALGORITHM

In this chapter, we present an algorithm for testing the satisfiability of a Tree Pattern Query. First we present an algorithm for a TPQ with a single node identity constraint. First we will provide an outline of the algorithm and then examine how the algorithm handles various cases. This algorithm can be used to test for satisfiability of queries with more than one node identity constraint, if the query vertices involved in the various NICs are disjoint.

4.1 Outline of Algorithm

The general structure for a structural constraint graph for a Conjunctive XPath query with one node identity constraint is shown in Figure 20. In the generalized Structural Constraint Graph, the least common ancestor of the nodes with the Node Identity Constraint is represented by the vertex *b*; the Node identity constraint is shown as vertex *v(1+2)* indicating that node v_1 is identical to node v_2 in the document. The path from node *b* to node v_1 is represented by e_1, p_1, f_1 and similarly the path from node *b* to node v_2 is represented by e_2, p_2, f_2 .

The algorithm determines if there is at least one possible way to create a single path from *b* to *v(1+2)* that satisfies the structural constraints. In the case where arcs e_1 , e_2 , and f_1 , f_2 are all d-arcs, it is possible to embed the path e_2, p_2, f_2 into the d-arc e_1 and produce a single path from *b* to *v(1+2)* and thus the query is satisfiable as it is possible to find an XML document that satisfies the query

which has its elements listed in the same order as the nodes on the path. Other embeddings are also possible that will also produce a single path.

The algorithm considers 4 special cases where it is possible that the query may not be satisfiable:

1) arcs e_1 and e_2 are both c-arcs
2) arcs f_1 and f_2 are both c-arcs
3) arc e_1 is a c-arc and e_2 is a d-arc and arc f_1 is a d-arc and f_2 is a c-arc
4) arc e_1 is a c-arc and e_2 is a d-arc but arc f_1 is a c-arc and f_2 is a d-arc.

   In this case 2 sub-cases are considered

   4.1) path *p1* contains a d-arc

   4.1) path *p1* contains no d-arcs, that is it contains only c-arcs.

In case 1, we are requiring nodes *c* and *d* to be cousins but also to be on the same path to *v(1+2)*. Generally this would cause the query to be unsatisfiable. However there is one special case where this constraint conflict can be resolved: namely nodes *c* and *d* are the same node which would require them to have the same tag. In this special case, we can “merge” these 2 nodes and this new node is now the least common ancestor of *v(1+2)*, then testing for satisfiability can be done recursively.

In case 2, we are requiring nodes *m* and *n* to be the parents of *v(1+2)*. This is a conflict as a node cannot have 2 parents in a TPQ and so the query would be unsatisfiable. However there is a special case where this conflict can be resolved namely nodes *m* and *n* are the same node which would require them to have the

same tag. In this special case, we can “merge” these 2 nodes and this new node is now the node with the node identity constraint instead of *v(1+2)* which is now the child of the new merged node, then testing for satisfiability can be done recursively.

From Case 1, we can assume that at least one of e_1 and e_2 is a d-arc, (otherwise we would apply the algorithm recursively until we had this situation, and if we cannot do this then the query is unsatisfiable). Without loss of generality, we can assume that e_2 is a d-arc. From Case 2, we can assume that at least one of f_1 and f_2 is a d-arc. If f_1 is a d-arc, then there is a path

b -> c => m => d => n -> v(1+2) which satisfies the constraints arising from the possible c-arcs, and so the query is satisfiable.

However in Case 4 we consider the situation where f_1 is a c-arc and also e_1 is a c-arc. If path *p1* contains a d-arc, then it is possible to embed the path e_2, p_2, f_2 into this d-arc in *p1* as both e_2 and f_2 are d-arcs. However if *p1* does not contain a d-arc, then *p2* must be a substring of *p1* that conforms to the c-arc constraints.

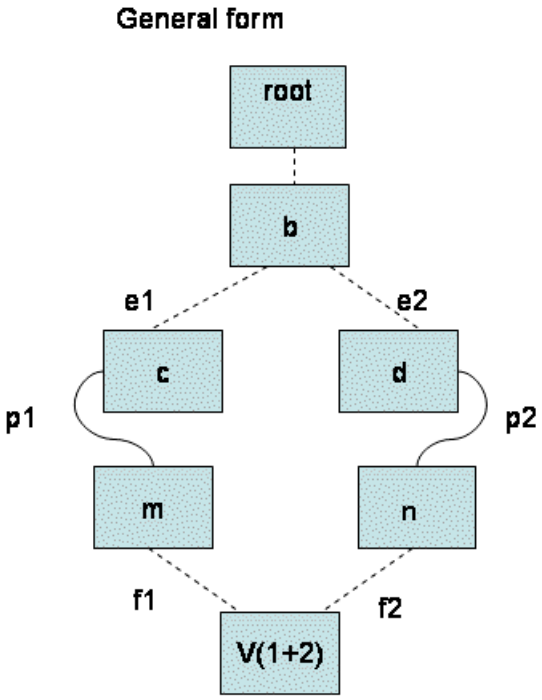


Figure 20. Generalized Structural Constraint Graph for a C XPath query with 1 NIC.

4.2 Formalization of Algorithm

For any TPQ *Q* (no node identity constraint), let *Δ(Q)* be the XML document obtained from *Q* as follows:

- Replace all *c*-arcs and *d*-arcs by parent-child edges;
- Replace each * vertex with any tag name in Σ.

There is a *natural* embedding of *Q* in *Δ(Q)*: Each vertex is mapped to the corresponding node. So TPQ *Q* (no node identity constraint) is definitely satisfiable as it can be embedded in *Δ(Q).* For example, see Figure 21.

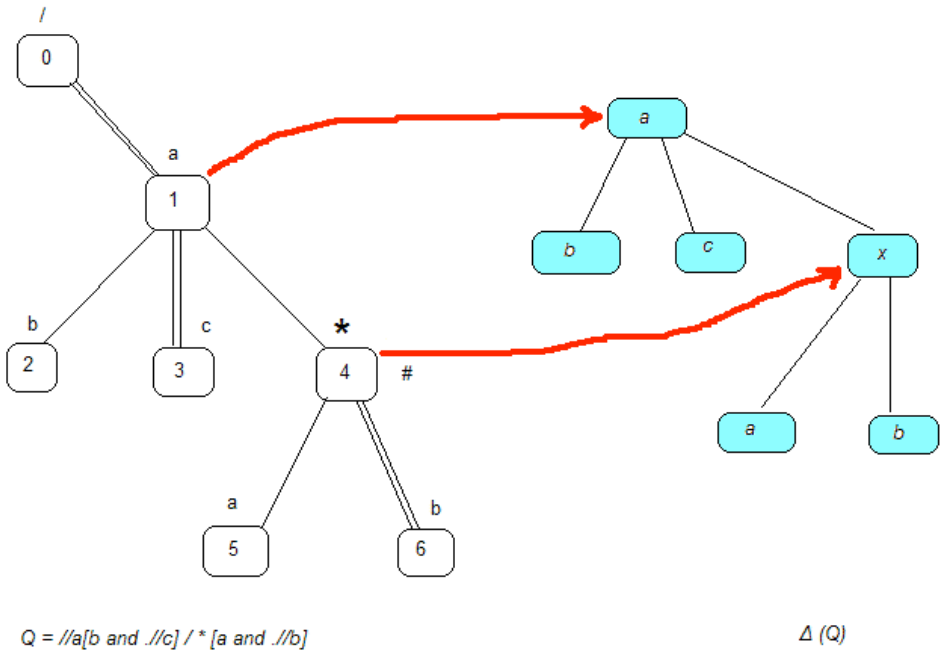


Figure 21. Query Q, document Δ(Q), and an embedding of Q in Δ(Q).

Now consider a TPQ Q with one Node Identity Constraint (NIC). Let *u* and *v* be the vertices of *Q* that are equated by the node identity constraint. Let *b* be the least common ancestor of *u* and *v* in *Q*. Let $P_u = (b, u_1, u_2, \ldots, u_m, u_{m+1} = u)$ be the path from *b* to *u* in *Q, and let* $P_v = (b, v_1, v_2, \ldots, v_n, v_{n+1} = v)$ be the path from *b* to *v* in *Q*.

Theorem 4.1

A TPQ *Q* with one NIC that equates vertices *u* and *v* is satisfiable iff there exists a path P of XML nodes and an embedding Γ of P_u and P_v in *P* such that

- Γ (b) has the same value for P_u and P_v
- Γ (u) = Γ (v)

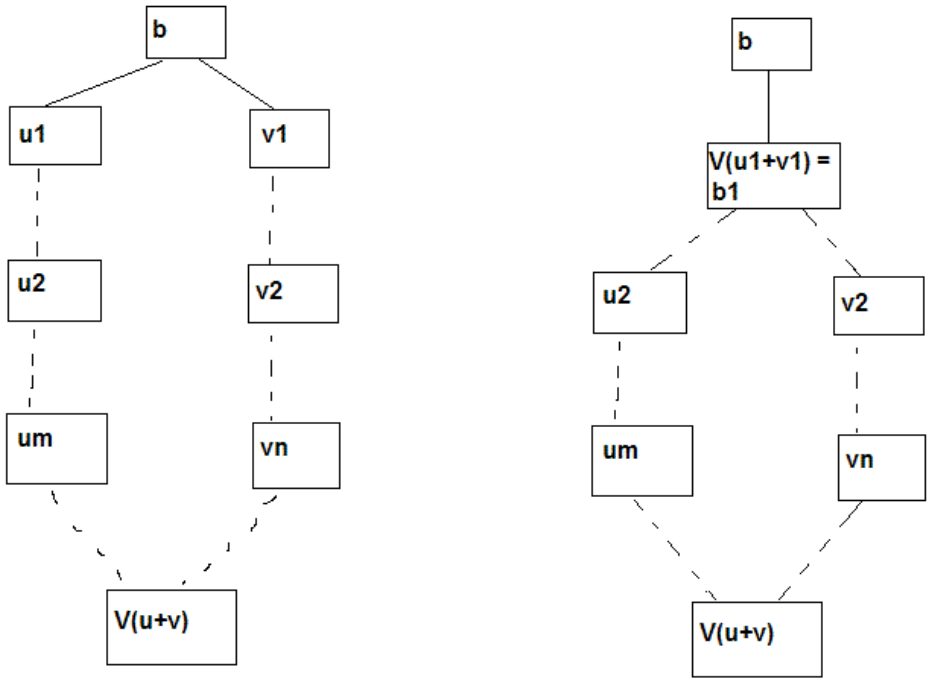
Proof. First consider the "only-if" case. Consider any possible embedding Γ of *Q* in an XML document. Clearly, *D* must contain a path satisfying the constraints in the theorem. Now consider the "if" case. Suppose that there exists a path *P* and an embedding Γ satisfying the constraints in the theorem. Then *P* can be extended by attaching extra nodes and subtrees, to embed the whole query *Q*. □

Consider a possible embedding Γ of P_u and P_v in path *P*. We need to find the conditions under which such *P* and Γ exist. In the Figures in this section, a solid single line represents a c-arc, a solid double line represents a d-arc, and a dotted line represents either a c-arc or a d-arc.

We consider the following cases.

Case 1. The arcs (*b*, u_1) and (*b*, v_1) are both c-arcs. Then, in any possible embedding of P_u and P_v in a path *P*, we must have $\Gamma(u_1) = \Gamma(v_1)$. So, for *Q* to be satisfiable, we must have $\gamma(u_1) = \gamma(v_1)$; so let $\gamma(u_1) = \gamma(v_1)$. Let *Q*' be the TPQ obtained by merging u_1 and v_1 into a single vertex b_1; the node identity

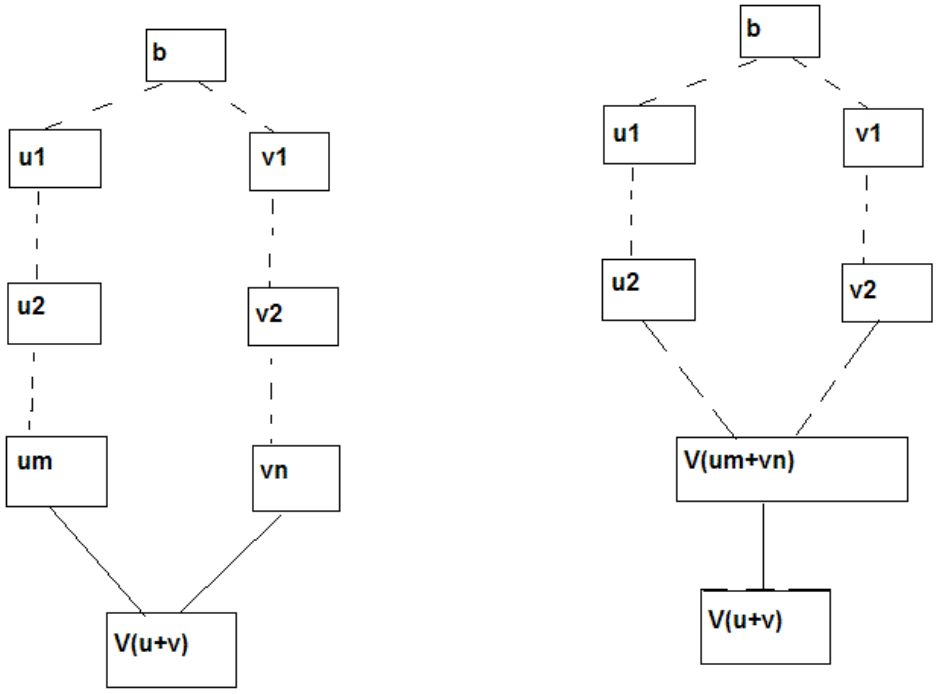
constraint is rephrased in terms of b_1 (see Figure 21 (b)). Q is satisfiable iff Q' is; satisfiability of Q' is tested recursively.



(a) (b)

Figure 22. Case 1: before and after the merge.

Case 2. The arcs (u_m, u) and (v_n, v) are both c-arcs. Then, in any possible embedding of P_u and P_v in a path P, we must have $\Gamma(u_m) = \Gamma(v_n)$. So, for Q to be satisfiable, we must have $\gamma(u_m) = \gamma(v_n)$; so let $\gamma(u_m) = \gamma(v_n)$. Let Q' be the same as Q, except that the node identity constraint is rephrased to equate u_m and v_n (see Figure 23 (b)). Q is satisfiable iff Q' is; satisfiability of Q' is tested recursively.



(a) (b)

Figure 23. Case 2: before and after the merge.

By Case1, we can assume that at least one of (b, u_1) and (b, v_1) is a d-arc. Without loss of generality, let (b, v_1) be a d-arc.

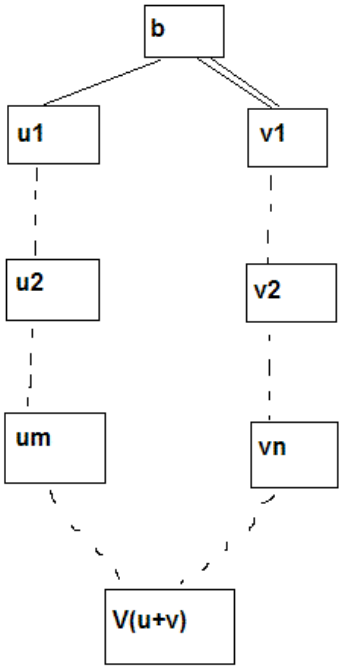


Figure 24. At least one of arcs (b,u_1) and (b,v_1) must be a d-arc

By Case 2, we can assume that at least one of arcs (u_m, u) and (v_n, v) is a d-arc. We consider two cases.

Case 3. The arc (u_m, u) is a d-arc (see Figure 24 (a)). Then, Q is satisfiable. Let $P'_u = P_u - \{u\} = (b, u_1, u_2, \ldots, u_m)$, and let $P'_v = P_v - \{b\} = (v_1, v_2, \ldots, v_n, v_{n+1} = v)$.

Let $P = \Delta(P'_u) \bullet \Delta(P'_v)$, where $\bullet$ denotes the concatenation operator. There is a natural embedding of P_u and P_v in P (see Figure 24 (b)). So by Theorem 4.1, Q is satisfiable.

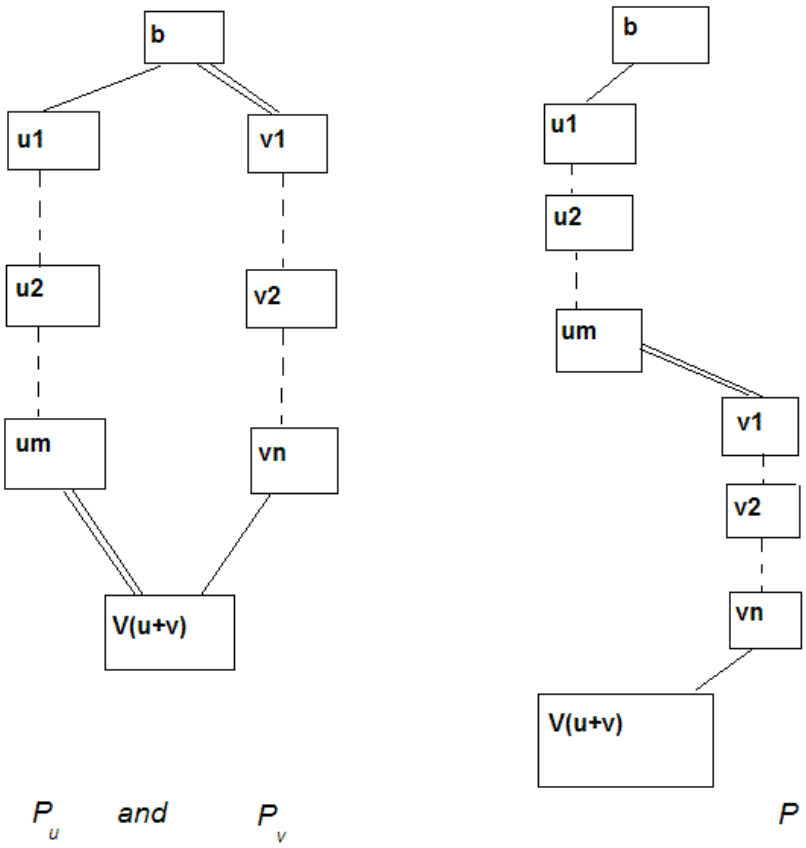


Figure 25. Case 3: before and after an embedding.

Case 4. The arc (u_m, u) is a c-arc, but the arc (v_n, v) is a d-arc (see Figure 26).

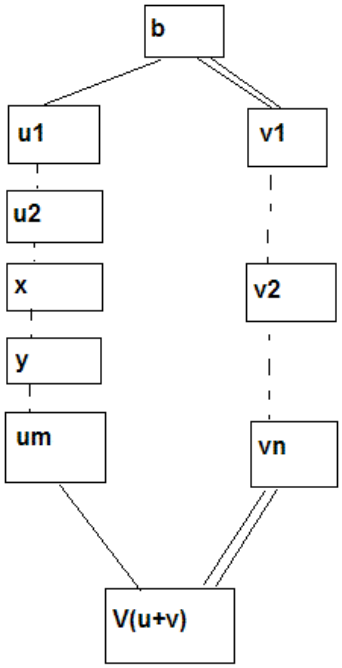


Figure 26. Case 4: arcs (b, u_1) and (u_m,V_{u+v}) are c-arcs.

We consider two sub-cases.

Case 4.1. P_u contains a d-arc (x,y). Then, Q is satisfiable. Let $P'_{u,1}$ be the part of P_u from b to x; $P'_{u,1} = (b, u_1, u_2 \ldots, u_i = x)$, for some $i < m$. Let $P'_{u,2}$ be the part of P_u from y to u; $P'_{u,2} = (y = u_{i+1}, u_{i+2}, \ldots, u_m, u_{m+1} = u)$. Let $P'_v = P_v - \{b, v\} = (v_1, v_2, \ldots, v_n)$. Let $P = \Delta(P'_{u,1}) \bullet \Delta(P'_v) \bullet \Delta(P'_{u,2})$. There is a natural embedding of of P_u and P_v in P (see Figure 27). So by Theorem 4.1, Q is satisfiable.

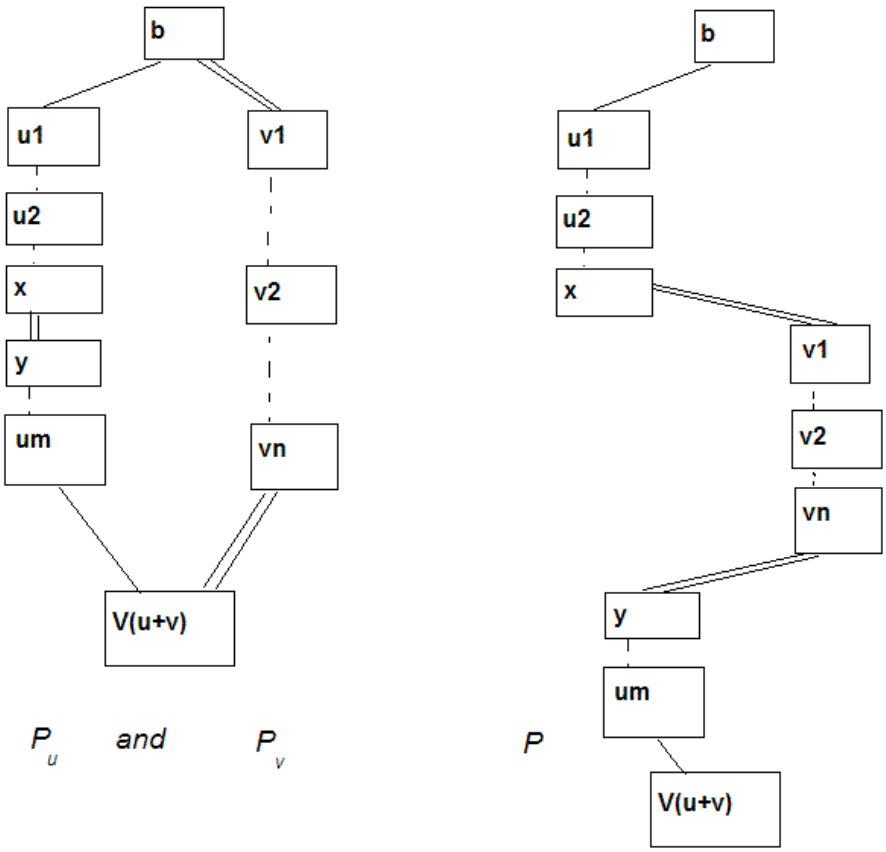


Figure 27. Case 4.1: before and after an embedding.

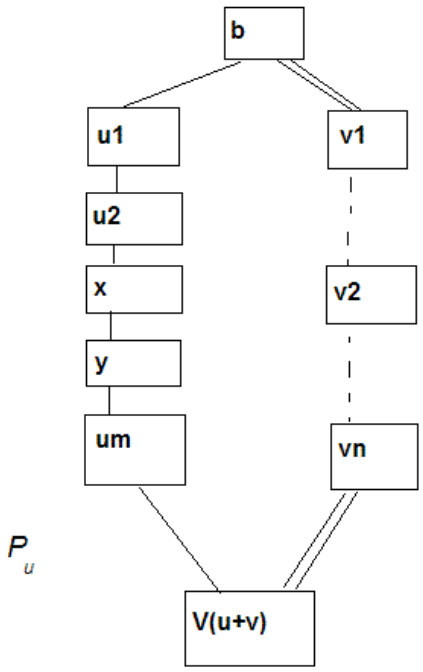


Figure 28. Case 4.2: P_u contains no d-arcs.

Case 4.2. P_u contains no d-arcs (see Figure 28). In this case, we must take P to be $\Delta(P_u)$, which is essentially P_u except that we can replace each * vertex label by any tag name in Σ. So, Q is satisfiable iff P_v can be embedded in P_u. To determine whether such an embedding exists, we partition P_v into maximal length chains such that each chain consists either of only c-arcs or only d-arcs; they will be referred to as c-chains and d-chains, respectively. Let $P_v = (P_{v,1}, P_{v,2}, \ldots, P_{v,k})$ be the partition. $P_{v,i}$ is a d-chain if i is odd, and is a c-chain if i is even, Since (b, v_1) and (v_n, v) are both d-arcs, both $P_{v,1}$ and $P_{v,k}$ are d-chains, and k must be odd. Let the vertices assigned to $P_{v,i}$, denoted by $V(P_{v,i})$, be defined as follows: If $P_{v,i}$

is a d-chain, $V(P_{v,i})$ is the sequence of vertices on $P_{v,i}$, *excluding* the two endpoints; so, if $P_{v,i}$ consists of a single d-arc, $V(P_{v,i})$ would be empty. If $P_{v,i}$ is a c-chain, $V(P_{v,i})$ is the sequence of vertices on $P_{v,i}$, *including* the two endpoints.

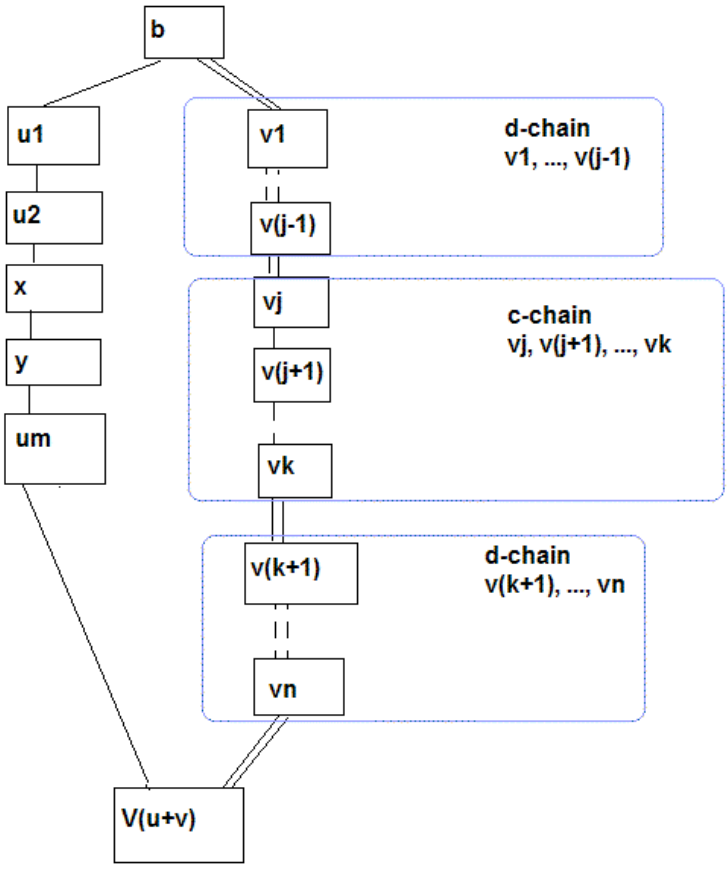


Figure 29. Case 4.2: P_v partitioned into c-chains and d-chains.

Embedding P_v in P_u reduces to embedding each $P_{v,i}$, one below the other. Suppose that we have embedded $P_{v,i}$ and let x be the image in P_u of the last vertex in $V(P_{v,i})$. Now consider the embedding of $P_{v,i+1}$ .If $P_{v,i+1}$ is a c-chain, we need to find the first sequence of consecutive vertices (not necessarily immediately) after x in P_u , whose tag names match the sequence of tag names

in $V(P_{v,i+1})$. If $P_{v,i+1}$ is a d-chain, we need to find the first sequence of not necessarily consecutive vertices (not necessarily immediately) after x in P_u, whose tag names match the sequence of tag names in $V(P_{v,i+1})$. Q is satisfiable iff each of the c-chains and d-chains in P_v can be embedded in P_u, in this manner, one below the other.

4.3 Application of Algorithm to Examples for Case 4.2

Figure 30 shows an example of a query that is satisfiable. By applying the algorithm, we partition the path P_v = (b, c, e, g) into a *c-chain* consisting of the *c-arc* (c,e) and into two *d*-chains containing no vertices. Next the path P_u = (b,c,d,c,e,f,g) is checked to determine if there is a possible embedding. The first vertex *c* is a match but the next vertex *d* does not match the next vertex *e* in the *c-chain*. So this is not a possible embedding so the checking continues. Vertex *d* is not a match but the next vertex *c* is a possible match. The subsequent vertex *e* is a match to the next vertex in the *c-chain*. All the vertices of the *c-chain* have been embedded into the path of *c-arcs* from b to g. The two *d-chains* do not contain any vertices so no further embedding is required.

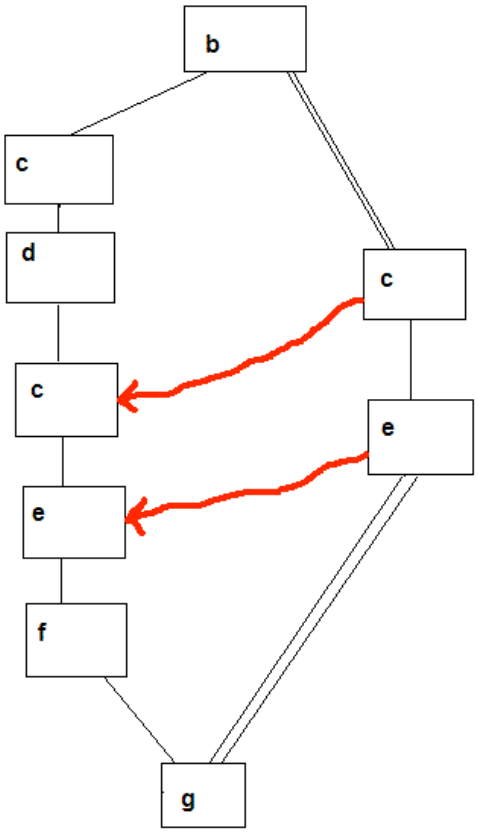


Figure 30. Case 4.2: Example of satisfiable query with c-chain.

Figure 31 shows a more complex example with two *d-chains* and 1 *c-chain* where the d-chains contain a vertex. The chains are shown together with their “match” in the long path, which shows that an embedding is possible and so the query is satisfiable.

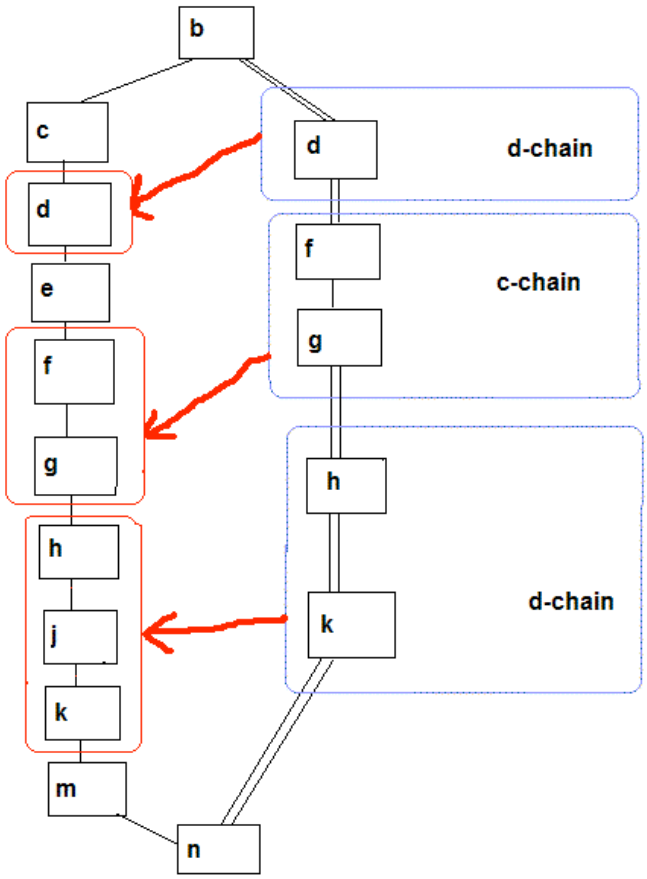


Figure 31. Case 4.2: Example of satisfiable query with d-chains and c-chain

If an embedding is not possible, then the query is unsatisfiable. Figure 32 shows an example of such a query. In this example, it is not possible to match the *c-chain* with a set of *c-arcs* in the long path. The vertices *x* and *h* do not match and so an embedding is not possible.

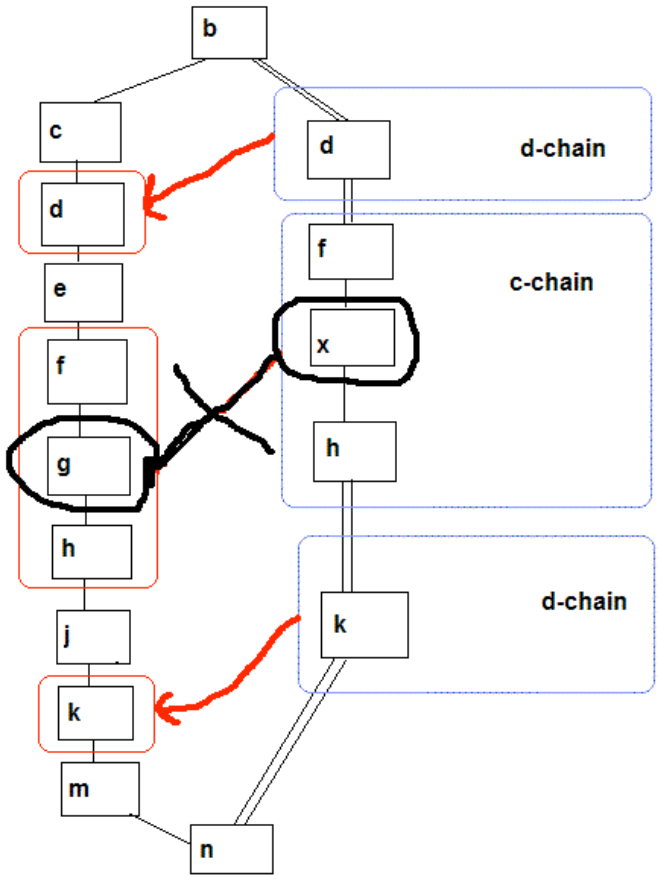


Figure 32. Case 4.2: Example of unsatisfiable query

4.4 Runtime of the Algorithm

Let $n = |P_u|$ and $m = |P_v|$. Consider embedding a chain $P_{v,i}$ in P_u. If $P_{v,i}$ is a d-chain, the time spent is equal to the length of P_u that is used up by the embedding; this is because we never look at a vertex in P_u more than once. So, the total time spent over all d-chains is $O(n)$. Now consider the c-chains. For a c-chain $P_{v,i}$, we spend $O(|P_{v,i}|)$ time for each starting point we try in P_u. So, total time spent on embedding $P_{v,i}$ is $O(m\,|P_{v,i}|)$. Over all the c-chains, the total time

is *O(nm).* This can be improved to *O(n+m)* using the string matching algorithm of Knuth, Morris and Pratt (Cormen, Leiserson et al. 2001).

CHAPTER 5

SUMMARY AND CONCLUSIONS

5.1 Theoretical Implications

Efficient algorithms for evaluating XPath and XQuery expressions are an important research area of Computer Science. Techniques developed for SQL queries are only partially applicable to the more complex queries for XML data. The ability to determine if a query is satisfiable before running a query against a large dataset is useful. Although it is not always possible to decide if an XPath or XQuery query is satisfiable, there is a subset of queries where it is possible to determine if a query is satisfiable. One subset is the Conjunctive XPath subclass and we have shown an efficient algorithm for determining satisfiability for this class of query with one Node Identity Constraint.

5.2 Conclusion

Algorithms for determining satisfiability may also be applicable to other problems. XPath in many ways resembles the path structures for storing and retrieving data in file systems. The structural constraint graphs are also similar to the representation of scheduling problems: this activity must occur after another but immediately before another. The concept of child and descendant axis could be used to represent the scheduling constraints and hence the algorithm developed for XML queries could be applied to other problems that can be represented by structural constraint graphs.

5.3 Extension of the Algorithm

The algorithm developed in Chapter 4 is for a single node identity constraint. However in more complex XQuery queries, the TPQ may contain multiple Node Identity Constraints. In section 3.3.3, it was noted that if there exists a sub-graph in the query that it unsatisfiable, then the larger query is unsatisfiable. Therefore one method to extend this algorithm to multiple node identity constraints would be to determine if the various node identity constraints are satisfiable. A rough outline of such an algorithm is provided below, for the case where the various NICs involve disjoint sets of vertices.

Step 1:

Partition the graph into sub-graphs for each Node identity Constraint.

Step 2:

Choose a node identity constraint and apply the algorithm from Theorem 4.3. If an embedding is possible, then continue applying the algorithm for any remaining node identity constraints, however do not make the embedding. It is only necessary to determine if an embedding is possible. If an embedding is possible for all the node identity constraints continue to Step 3, else the query is unsatisfiable.

In the situation where the NICS involved overlapping sets of vertices, these multiple node identity constraints can introduce a cycle into the graph. A

topological sort is one method of determining if a graph contains a cycle. If a topological sort can be created for a DAG, then the graph does not contain a cycle.

5.4 Areas of Further Research

The extended algorithm requires further research in the area of nested and overlapping Node Identity Constraints. Where the Node Identity Constraints do not overlap or are not nested within each other, then the extended algorithm would be suitable. In the case of nested or overlapping Node Identity Constraints, an embedding for the inner most NIC, may make an embedding for the outermost or overlapping NIC impossible to achieve. This is because our algorithm selects just one of many possible embeddings to determine satisfiability. When the NICs overlap or are nested, it may be that a different choice of embeddings need to be considered.

REFERENCES

LIST OF REFERENCES

Amer-Yahia, S., S. Cho, et al. (2001). Minimization of Tree Pattern Queries Proceedings of ACM SIGMOD, Santa Barbara California.

Bender, M. A., G. Pemmasani, et al. (2001). Finding Least Common Ancestors in Directed Acyclic Graphs. Proceedings of the 12th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), Washington, D.C.

Benedikt, M., W. Fan, et al. (2005). XPath Satisfiability in the Presence of DTDs ACM Symposium on Principles of Database Systems (PODS), Baltimore, Maryland.

Brundage, M. (2004). XQuery: The XML Query Language. Boston, Addison-Wesley.

Cormen, T., C. Leiserson, et al. (2001). Introduction to Algoritms. Cambridge, MA, MIT Press/McGraw Hill.

Fernández, M., J. Hidders, et al. (2004). Automata for Avoiding Unnecessary Ordering Operations in XPath Implementations, University of Antwerp Technical Report TR UA 2004-02.

Geerts, F. and W. Fan (2005). Satisfiability of XPath Queries with Sibling Axes The 10th International Workshop on Database Programming Languages, Trondheim, Norway.

Hidders, J. (2003). Satisfiability of XPath Expressions. Proceedings of the 9th International Conference on Data Base Programming Languages, Potsdam, Germany.

Kay, M. (2004). XPath 2.0 Programmer's Reference (Programmer to Programmer). Indianapolis, IN, Wiley Publishing.

Lakshmanan, L. V. S., G. Ramesh, et al. (2004). On Testing Satisfiability of Tree Pattern Queries. Proceedings of the 30th International Conference on Very Large Data Bases, Toronto, Canada.

Lewis, P. M., A. Bernstein, et al. (2002). XML and Web Data. Databases and Transaction Processing: An Application-Oriented Approach. Boston, Addison-Wesley**:** 537-623.

Quenault, H. S. (2004). VERS: Building a Digital Heritage. VALA, Victorian Association for Library Automation, Melbourne, Australia.

Ramanan, P. (2003). Covering Indexes for XML Queries: Bisimulation - Simulation = Negation. VLDB.

Stephen, G. A. (1994). String Searching Algorithms. Singapore, World Scientific.
W3C (1996). Extensible Markup Language (XML).