Regular Path Queries with Constraints

Serge Abiteboul*

INRIA-Rocquencourt, 78153, Le Chesnay Cedex, France
E-mail: Serge.Abiteboul@inria.fr

and

Victor Vianu†

CSE 0114, University of California San Diego, La Jolla, California 92093-0114
E-mail: vianu@cs.ucsd.edu

Received December 29, 1997; revised October 7, 1998

The evaluation of path expression queries on semistructured data in a distributed asynchronous environment is considered. The focus is on the use of local information expressed in the form of path constraints in the optimization of path expression queries. In particular, decidability and complexity results on the implication problem for path constraints are established.

1. INTRODUCTION

Navigational queries on data represented in a graph-like manner have proven to be useful in a variety of database contexts, ranging from hypertext data to object-oriented databases. Typically, navigational queries are expressed using regular expressions denoting paths in the graph representing the data. Such path queries have assumed renewed interest in the context of semistructured data [2, 28, 5, 11, 22, 30, 27] as found, for instance, in the Web and in particular for querying XML [1]. We focus on a path query evaluation that takes advantage of local knowledge about the data graph. We consider such local knowledge represented as path constraints. The main contribution of the paper is the study of the implication problem for path constraints, and its use in optimizing the evaluation of path queries.

* This work was performed while the author was visiting the CS department at Stanford U. Work supported in part by CESDIS, NASA Goddard Space Flight Center, by the Air Force Wright Laboratory Aeronautical Systems Center under ARPA Contract F33615-93-1-1339, and by the Air Force Rome Laboratories under ARPA Contract F30602-95-C-0119.

† This author supported in part by the National Science Foundation under Grant IRI-9221268.
We use here an abstraction of semistructured data as a set of objects linked by labeled directed edges. In Web terminology, an object can be viewed as a page, and the labeled edges as hypertext links. We focus on path queries \([12, 20, 13, 3, 24, 5, 11, 22, 30]\), which have emerged as an important class of browsing-style queries on graph data. Path queries are of the form \(\text{find all objects reachable by paths whose labels form a word in } r\), where \(r\) is a regular expression over an alphabet of labels. In the context of the Web, we believe that queries involving path expressions will be useful, even if not present explicitly in query languages available to naive users.

We present a basic scenario for evaluating such queries in a distributed context, based on simple communication between sites. (This can be viewed as an agent-based algorithm, in the sense that the query is evaluated by software agents traveling among sites, executing simple code, and spawning other agents.) We show that our technique correctly evaluates the answer, and we provide a protocol that also detects termination whenever possible. We also point to an analogy between our evaluation technique and the \textit{magic-set} [9] or \textit{query-subquery} [31] evaluation of a datalog program (see also [4]).

The distributed processing of path queries can be greatly enhanced by taking advantage of \textit{path constraints}. Path constraints are local; they may capture, for instance, structural information about a Web site (or a collection of sites) or about its physical organization (e.g., cached information). A path constraint is an expression of the form \(p \sqsubseteq q\) or \(p = q\), where \(p\) and \(q\) are regular expressions. A path constraint \(p \sqsubseteq q\) holds at a given site if the answer to query \(p\) applied to that site is included in the answer to \(q\) applied to the same site (and similarly for \(p = q\)). The following are some self-explanatory examples of path constraints:

\begin{align*}
\text{CS-Department DB-group Ullman Classes cs345} \\
=\text{CS-Department Courses cs345} \\
\text{CS-Department Faculty Publications} \\
\subseteq \text{netscape cache 07}.
\end{align*}

Taking advantage of such information in query processing turns out to be non-trivial. A difficulty is to decide equivalence (or inclusion) of regular path queries under such constraints. This is the central technical problem that we address. This problem lies at the confluence of language theory, rewriting systems, and logic. We are able to prove that the general implication problem for regular path constraints is decidable in \(2\text{-expspace}\) \(^1\) (with respect to the size of the constraints). This result is rather surprising, since closely related problems in logic and rewriting systems are known to be undecidable. We obtain improved decision procedures of complexity \(\text{ptime}\) and \(\text{pspace}\) for two important special cases and develop along the way several technical tools related to implication. We lastly apply these techniques to

\(^1\) \(2\text{-expspace}\) denotes the functions computable in space bounded by \(2^{2^k}\) for some \(k\), where \(n\) is the size of the input.
the boundedness problem for regular path expressions. We show that it is decidable whether a given regular path query is equivalent to a path query without recursion, assuming that a given set of equalities among words is satisfied.

Related work. Path queries in graphs have been studied formally in [10, 12, 8, 24]. The language Graphlog, introduced in [12], expresses queries using graph patterns, where paths are specified by regular expressions. Graphlog is shown equivalent to stratified linear Datalog and other languages. The complexity of path queries in graphs is studied in [24]. Specifically, the problem of finding all pairs of nodes connected by a simple path satisfying a given regular expression is shown to be NP-complete in the size of the graph, and tractable subcases are identified. Cycle constraints which restrict cycles in a graph by regular expressions are also discussed. Path queries (regular, linear and context-free) and their complexity are also considered in [32]. It is shown there how regular path queries on graphs can be reduced to the transitive closure query using the cross-product of the input graph with the query automaton. Connections with Datalog are also discussed.

In [8], the problem of finding paths in a labeled graph that spell some word in a regular language is considered. They propose a data structure that can be incrementally maintained when arcs are inserted and deleted. The problem is different from ours (the regular language is fixed and the focus is on incremental evaluation). However, the techniques they develop are relevant to the evaluation of path expressions described in Section 2. In particular, the processing of Datalog programs with one-sided recursion [26], discussed in [8], is in the spirit of our evaluation of path queries using Datalog. The connection between path queries and recursive query processing in deductive databases is also discussed in [10], which relates regular path queries to chain programs in Datalog, where the recursive predicates are monadic.

Query languages for semistructured data that include path expressions are considered in [21, 11, 22, 30]. The language UnQL and its optimization are discussed in [11]; the optimizations involve loop fusion and a form of pushing selection; [30] provides an evaluation procedure of UnQL queries in a distributed Web-like environment. Using a decomposition technique, it is shown that UnQL queries can be evaluated by shipping the query exactly once to every site, returning the local results to the client site, and assembling the final result at the client site. Reference [22] considers the language WebSQL, which also incorporates path expressions and provides a theory of query cost, based on the notion of query locality. Path queries in object-oriented databases are considered in [13]; they focus on the concise specification of path queries and the inference of completions of partially specified paths from schema information.

Recently, path constraints similar to ours and their use in rewrite rules for query optimization were considered by Mecca, Mendelzon, and Merialdo [23]. Their constraints, unlike ours, apply to intensional descriptions of Web data rather than the extensional data itself.

The paper is organized as follows. Section 2 provides the background and motivation and relates path queries to datalog. Section 3 presents our basic distributed evaluation scenario for path queries and discusses path constraints and their use in query optimization.
Our main results on the implication problem for path constraints are presented in Section 4. Brief conclusions are provided in Section 5.

2. PATH QUERIES

We first present a simple abstraction of semistructured data and introduce path queries expressed using regular expressions. We then explore the connection with datalog, which provides a well-studied framework for expressing and evaluating recursive database queries. We show how regular path queries are expressed as datalog with certain special properties and consider the evaluation of such programs on both finite and infinite instances. Finally, we briefly consider more general path queries and show how their evaluation can be reduced to the evaluation of standard path queries.

2.1. Semistructured Data

We view a (semistructured) database as a labeled graph, i.e., as an instance of the relational schema,

\[
\text{Ref}(\text{source: oid, label: label, destination: oid}),
\]

where \text{oid} and \text{label} are (countable) infinite, disjoint sorts. In the context of the Web, an object is an abstraction of a Web page. Labeled edges model labeled links among pages. More precisely, \text{Ref}(o_1, l, o_2) indicates that there is an edge labeled \(l\) from \(o_1\) to \(o_2\). For reasons discussed below, it is sometimes interesting to also consider the case when the graph represented by \text{Ref} is not restricted to be finite. However, in either the finite or infinite case, objects are “small;” i.e., each vertex is of finite outdegree. This is in agreement with what is found on the Web, where a page may be referenced arbitrarily many times but only references a fixed, generally small number of pages. More precisely, for each object \(o\) there are finitely many tuples in \(I\) with \(o\) in the first column. The description of \(o\) in \(I\) consists of this finite set of tuples. Thus, the description of an object provides its outgoing links. On the other hand, there may be infinitely many objects pointing to some object \(o\); i.e., \(o\) may have an infinite indegree.

We call a relation \(I\) over \text{Ref} restricted as above an \textit{instance}. We say that object \(o'\) is reachable from object \(o\) if there is a directed path from \(o\) to \(o'\) in the labeled graph given by \(I\). The \textit{distance} between two objects is also defined with respect to the graph \(I\).

The motivation for considering infinite instances is discussed at length in [6]. Intuitively, a query that results in exhaustive exploration of the Web is unreasonable. However, this notion seems hard to formalize in standard complexity terms. On the other hand, infinite graphs yield a sharp distinction between such queries and more tractable ones, by penalizing exhaustive exploration of the Web with a nonterminating computation. This aspect is formalized and studied at length in [6]. The model leads to a focus on querying and computation where exploration of the Web is controlled. Unlike [6], most of the investigation in the present paper is not tied
to the infiniteness assumption. We consider instead both the finite and infinite cases. It turns out that most of our results are independent of (in)finiteness assumptions.

2.2. Regular Path Queries

We next recall the notion of regular path query. In the paper, we assume familiarity with basic notions of formal language theory, such as regular expressions and regular languages, (nondeterministic) finite state automata ((n)fsa), context-free languages, and pushdown automata (pda); see [18]. We will use the following terminology and notation. An fsa is written as a 5-tuple \((Q, s, A, \Sigma, \delta)\), where \(Q\) is the finite set of states, \(s\) the start state, \(A\) the set of accepting states, \(\Sigma\) the input alphabet, and \(\delta: Q \times \Sigma \rightarrow Q\) the transition function. The language accepted by an fsa \(F\) is denoted by \(L(F)\). Regular expressions are denoted \(p, q, r, \ldots\) and the language defined by a regular expression \(p\) is denoted by \(L(p)\). We will use the following facts on quotients of regular languages. For each regular language \(L\) over some alphabet \(\Sigma\) and each \(l \in \Sigma\), the quotient of \(L\) by \(l\) is the language \([w \mid lw \in L]\), denoted \(L/l\).

It turns out that \(L/l\) is regular if \(L\) is regular. To see this, consider an fsa \(F = (Q, s, A, \Sigma, \delta)\) accepting \(L\). For each state \(q\) in \(Q\), let \(F_q\) be the fsa \((Q, q, A, \Sigma, \delta)\) (the only difference with \(F\) is that the start state of \(F_q\) is \(q\)). Clearly, \(L(F)/l\) equals \(L(F_q)\), where \(q = \delta(s, l)\). It also follows that by taking repeated quotients of a regular language one only obtains a finite set of regular languages, since \(F\) has finitely many states. If \(p\) is a regular expression over \(\Sigma\) and \(l \in \Sigma\), \(p/l\) denotes a regular expression for the regular language \(L(p)/l\).

Let \(I\) be an instance. A (regular) path query is a regular expression over some finite alphabet \(\Sigma\) included in \(label\). In keeping with usual notation for regular expressions, “+” represents union and “*” the Kleene closure. Examples of path queries are:

\[
\text{section (paragraph + figure) caption}
\]
\[
\text{engine (subpart)\_name}
\]

Path queries are navigational and are posed relative to some designated source vertex. Thus, the semantics of a path query is determined by an input pair \((o, I)\), where \(I\) is an instance and \(o\) is an object in \(I\). The answer of a path query \(p\) on input \((o, I)\) is the set of all objects \(o'\) reachable from \(o\) by some path whose labels spell a word in \(p\). More precisely, \(o'\) is in \(p(o, I)\) if there is a directed path from \(o\) to \(o'\) whose edges are labeled \(l_1, \ldots, l_m\) for some word \(l_1 \ldots l_m\) in \(L(p)\). Two path queries \(p\) and \(q\) are equivalent if \(p(o, I) = q(o, I)\) for every input \((o, I)\). Clearly, this holds if and only if \(L(p) = L(q)\).

Note that if \(I\) is finite, \(p(o, I)\) is finite and computable (in polynomial time). It is easy to design an algorithm to evaluate path queries. The following observations provide the ingredients of a basic recursive evaluation procedure:

- if \(e \in L(p)\) then \(o \in p(o, I)\); and
- if \(\langle o, l, o' \rangle \in I\) and \(x \in (q/l)(o', I)\) then \(x \in q(o, I)\).
In fact, it is easy to verify that

\[ p(o, I) = \{ o \ | \ z \in L(p) \} \cup \{ p/\|a', I \} \text{ Ref}(o, I, a') \}. \]  

Thus, a recursive procedure based on the above would repeatedly evaluate quotients of the original query. The needed quotients could be constructed explicitly; however, this may be exponential in \( p \), since it requires constructing the \( \text{fsa} \) for \( p \). A more economical approach is to construct the \( \text{nfsa} \) for \( p \) and carry along the set of states of the \( \text{nfsa} \) corresponding to the path traveled so far (basically, this constructs a portion of the product of the \( \text{nfsa} \) for \( p \) and the instance \( I \)). The resulting algorithm has polynomial-time combined data and query complexity and \( \text{nlogspace} \) (and therefore \( \text{NC} \)) data complexity.

It will be interesting to show how path queries can be expressed in datalog, the standard declarative language for recursive database queries. We do this next. The main benefit of this exercise is to place path queries in the broader framework of recursive queries and to establish connections with known results about datalog. For example, the restricted form of the datalog programs needed to express path queries (linear datalog) immediately yields an upper bound of \( \text{NC} \) for the complexity of path queries, using known results about datalog [19] (note, however, that the earlier direct algorithm establishes the better \( \text{nlogspace} \) data complexity).

2.3. Path Queries and Datalog

We show how path queries can be expressed as linear datalog programs with monadic idb’s. We assume familiarity with datalog (e.g., see [4]). Let \( p \) be a regular expression and \( F_p \) an \( \text{fsa} \) accepting \( p \). We describe a datalog program \( D_p \) evaluating \( p(o, I) \) by a recursive procedure based on (†) above. The datalog program \( D_p \) has two edb relations: \( \text{Ref} \) providing the input graph and \( \text{source} \) providing the source node. The idb relations are all unary. Let \( \mathcal{P} \) consist of one regular expression for each of the languages obtained by taking repeated quotients of \( p \) by symbols in the alphabet. As noted above, \( \mathcal{P} \) is finite. For each \( q \in \mathcal{P} \), the program \( D_p \) has one idb relation \( \text{still-left}_q \). Intuitively, \( \text{still-left}_q(x) \) holds if \( x \) is reachable from \( o \) (the source node) by a path \( l_1 \cdots l_k \) and \( q = p/l_1/l_2/\cdots/l_k \) (thus, \( q \) is the subquery “still left” to evaluate from \( x \)). Additionally there is one idb relation \( \text{answer} \). The program \( D_p \) consists of the rules

\[
\begin{align*}
\text{still-left}_q(o) & : \text{source}(o) & \text{initialization} \\
\text{still-left}_q(x) & : \text{still-left}_q(y), \text{Ref}(y, l, x) & \text{for every } q \in \mathcal{P}, r = q/l \\
\text{answer}(x) & : \text{still-left}_q(x) & \text{for every } q \in \mathcal{P}, \\
& & \text{such that } e \in L(q).
\end{align*}
\]

In some cases, it is more convenient to think in terms of states of the \( \text{fsa} \) for \( p \) rather than the quotient queries. The two approaches are, of course, syntactic variants of each other. To illustrate, let \( F_p = (Q, s, A, \Sigma, \delta) \). A datalog program
computing \( p(o, l) \) has the same edb's as the above: \( \text{Ref} \) and \( \text{source} \). For each state \( h \in Q \), the program uses a unary idb relation \( \text{state}_h \). Intuitively, \( \text{state}_h(x) \) holds if \( x \) is reachable from \( o \) (the source) by some path \( l_1 \cdots l_k \) such that \( h = \delta(s, l_1 \cdots l_k) \).

The program consists of the rules

\[
\begin{align*}
\text{state}_o(o) :&= \text{source}(o) & \text{initialization} \\
\text{state}_x(y, \text{Ref}(y, l, x)) :&= \text{state}_j(y), & \text{for every state } j \in Q \text{ and } h = \delta(j, l) \\
\text{answer}(x) :&= \text{state}_h(x) & \text{for each accepting state } h.
\end{align*}
\]

The datalog programs that are obtained (by either the quotient or state approach) are very particular. First, they are linear; i.e., at most one intensional predicate occurs in the body of each rule. Since the evaluation of linear datalog programs is in \( \text{nc} \) [19], it follows that the evaluation of path queries is also in \( \text{nc} \). The programs are also monadic [19] (the recursion is over a unary predicate), another restriction of datalog programs of importance in query optimization.

**Remark 2.1 (Infinite Web).** The point of view that the Web is infinite is adopted in [6] to capture the intuition that exhaustive exploration of the Web is—or will soon become—prohibitively expensive. The infiniteness assumption can be viewed here as a convenient metaphor, much like Turing machines with infinite tapes are useful abstractions of computers with finite (but potentially very large) memory. The infiniteness assumption also has the advantage of clearly identifying queries that require exhaustive exploration of the Web, as opposed to more controlled navigation. Indeed, this assumption penalizes exhaustive exploration of the Web by a nonterminating computation. Whether there exists a finitary approach that results in the same classification of queries (e.g., based on complexity), remains an interesting open problem.

In [6], machine-based models of computation for infinite Webs and notions of computability over such infinite structures are considered. It is shown in [7] that path queries are computable by so-called Browser machines, which model the prevalent style of accessing the Web navigationally. Using the terminology of [6], this implies that path queries are eventually computable; i.e., each path query can be evaluated by considering increasing but finite portions of the infinite Web, and every single answer to the query will be produced eventually. We do not further pursue these issues in the present paper.

### 2.4. More General Path Queries

Our model provides a bare-bones abstraction of the Web and of some query languages recently proposed for semistructured data and the Web. It is worth noting that our framework can be easily adapted to capture some additional aspects not explicitly included in the model. For example, some languages with path expressions (such as Lorel [5]) view labels as strings of characters and use regular expressions that work at two levels of granularity: the label (viewed as a
string of characters) and the path (viewed as a sequence of labels). For instance, consider the following general path expression:

```
"doc"("[sS]ections?" "text"+"[pP]aragraph")
```

which specifies a path starting with an edge labeled doc, either followed by an edge labeled section(s) (possibly starting with a capital S) and a text-edge, or followed by an edge labeled paragraph (possibly with a capital P).

We used here a syntax based on grep E-regular expressions for string patterns, and quotes to separate labels (strings of characters) from paths (sequences of labels).

Call such queries general path queries. We claim that these can essentially be captured by our framework, modulo some preprocessing of labels. Let \( q \) be a general path query and let \( \Sigma \) be the set of string patterns occurring in \( q \). We will reduce the problem of the evaluation of the general path query \( q \) on an instance \( I \) with possibly infinitely many labels, to the problem of the evaluation of a regular path query \( \mu(q) \) on an instance \( \mu(I) \) with finitely many labels. To do this, consider the equivalence relation on strings defined by:

\[ v \sim u \text{ if } v \text{ and } u \text{ satisfy precisely the same patterns in } \Sigma. \]

For each equivalence class \([ v ]\), choose a particular label say \( l([ v ]) \) in \([ v ]\) and let \( \Sigma \) be the set of such labels. Observe that \( \Sigma \) is finite. Now \( \mu \) is defined as

1. for each label/string \( v \), \( \mu(v) = l([ v ]) \) and \( \mu(o) = o \) for each vertex \( o \) in \( I \); this defines \( \mu(I) \).
2. for each string pattern \( s \) occurring in \( q \), let \( \mu(s) = l([ v_1 ]) + \cdots + l([ v_k ]) \), where \([ v_1 ], \ldots, [ v_k ]\) are the equivalence classes of words satisfying \( s \) (i.e., \([ v_j ] \subseteq L(s)\)); this defines \( \mu(q) \).

The construction is illustrated next.

**Example 2.1.** Consider the general path expression:

\[ q = ("a*b" + "ba""") + ("a*b" + "c") + ("ba" + "c") + ("dd"""). \]

One can find six equivalence classes:

\[
[b] = a*b \cap ba* = b, \\
[ab] = a*b - ba* = aa*b, \\
[ba] = ba* - a*b = ba*, \\
[c] = c, \\
[d] = dd*, \\
[h] = \Sigma* - ([b] \cup [ab] \cup [ba] \cup [c] \cup [d]).
\]

Let “b,” “ab,” “ba,” “c,” “d,” “h” be the labels representing the equivalence classes. An instance \( I \) and its corresponding “translation” \( \mu(I) \) are represented in Fig. 1. The query \( \mu(q) \) is

\[ \mu(q) = ("b" + "ab") ("b" + "ba") + ("b" + "ab") "c" + ("b" + "ba") "c" + ("d")+. \]
FIG. 1. A graph and its translation.

Clearly, a vertex is in $q(o, I)$, if and only if it is in $\mu(q)(o, \mu(I))$. In general, we have

**Proposition 2.2.** For each $q$, $o$, $I$, and $\mu$ defined as above,

$$q(o, I) = \mu(q)(o, \mu(I)).$$

**Proof.** By induction, on the evaluation of $q(o, I)$ and $\mu(q)(o, \mu(I))$.

This allows us to reduce the problem of the evaluation of a general path query involving potentially infinitely many labels to the evaluation of a regular path query on a finite alphabet of labels, via preprocessing of labels.

To conclude this section, we mention another extension. Consider the Web. Our basic model only captures links among pages. However, pages also have content, in the form of text. A vertex $o$ with “content” $w$ (where $w$ is a string) can be modeled in our context by having an edge labeled “content=$w$” outgoing from $o$ and pointing to $o$ itself. Now content-based selections can be specified using the general path expressions just discussed. For instance, the reachable vertices that contain the word “SGML” can be retrieved using the general path query

$$('('*.*'))* 'content=(' . *)*SGML(' . )*'$',

where “(*.)*” indicates some arbitrary sequence of characters.

In the remainder of the paper, we only consider regular path queries.

### 3. DISTRIBUTED EVALUATION AND OPTIMIZATION

In this section, we outline a distributed evaluation algorithm for path queries, motivated by the distributed nature of the Web. We then briefly discuss some optimization techniques that motivate our main results.
3.1. Distributed Evaluation of Path Queries

As outlined in the introduction, we are motivated by a natural scenario for processing path queries in a distributed environment with asynchronous communication. In this scenario, objects represent sites. The processing of a path query involves local processing at each site and simple communication between sites. To simplify, here we ignore node or network failures. In particular, we assume that every message eventually reaches its destination.

We also assume that a single query is evaluated at a time. (Many queries may be treated by appending a global query identifier to all messages.) Communications between nodes consist in messages of the form:

\[
\text{subquery}(\text{mid}, \text{sender}, \text{receiver}, \text{destination}, q)
\]

\[
\text{done}(\text{mid}, \text{sender}, \text{receiver})
\]

\[
\text{answer}(\text{mid}, \text{sender}, \text{receiver})
\]

\[
\text{akn}(\text{mid}, \text{sender}, \text{receiver}).
\]

Each of these messages is sent by a sender to a receiver. The semantics of a subquery message is as follows. The answers to path subquery \( q \) evaluated at node receiver must be sent to node destination. An answer message is used to send to the node that initiated the query (i.e., the destination of the subquery) a new answer that has been found. The identifier \( \text{mid} \) is used to uniquely identify each particular subquery or answer message. A done message is used to notify that a subquery has been completed and an ack message to acknowledge the reception of an answer. In both cases, \( \text{mid} \) is used to identify the particular subquery or answer message.

This will be best seen in an example that we discuss below. A possible run of this algorithm with the graph \( I \) represented in Fig. 2 and with the query \( ab^*(o_1, I) \) asked by node \( d \) is shown in Fig. 3.

In the example, node \( d \) asks node \( o_1 \) query \( ab^* \) by sending the message subquery(\#d, d, o_1, d, ab*) to \( o_1 \). (This is the first message in Fig. 3.) This results in a series of messages.

Suppose some node \( r \) receives a subquery message subquery(\( m, s, r, d, q \)). If \( e \) is in the language described by \( q \), then \( r \) detects that it is itself an answer to the global query, so \( r \) sends an answer message to the destination node \( d \) that initiated the

![Graph I](https://via.placeholder.com/150)

**FIG. 2.** Graph \( I \).
global query; e.g., \( o_2 \) sends message \( \text{answer}(\#12, o_2, d) \) to \( d \) to tell \( d \) that \( o_2 \) is an answer. Besides eventually returning itself as answer, node \( r \) processes the first (possible) letter(s) of the query and asks its neighbors, accessible by links labeled with the appropriate letter to continue the work. This is what \( o_2 \) does by calling \( o_3 \) with subquery \( \text{subquery}(\#23, o_2, o_3, d, b^*) \). This results in possibly initiating new subqueries. When node \( r \) has been notified that all subqueries it initiated have been processed (\( \text{done} \) messages) and if (eventually) the destination acknowledged receiving the new answer (\( \text{ack} \) message), then \( r \) notifies \( s \) with a \( \text{done} \) message that its subtask has been performed.

The time that is relevant here is the time of message reception. For instance, to process subquery \( \#12 \), node \( o_2 \) sends the answer message \( \text{answer}(\#12, o_2, d) \) and the subquery message \( \text{subquery}(\#23, o_2, o_3, d, b^*) \). It can send back to node \( o_1 \) a done message (\( \text{done}(\#12, o_2, o_1) \)) only when it has received the \( \text{ack} \) message for \( \#12, o_2, d \) and the \( \text{done} \) message for \( \#23 \) from \( o_3 \).

Each node maintains a list of the subqueries it has been asked to perform. If node \( r \) is asked to perform a subquery that it is already processing or that it has already processed, \( r \) immediately returns to \( s \) a \( \text{done} \) message for that particular task. This is what happens when \( o_1 \) asks \( o_2 \) to evaluate \( b^* \). Node \( o_2 \) immediately answers that this is \( \text{done} \). In general, the node may still be processing the particular subquery. However, this is simply to avoid duplicating effort, is essential to guarantee termination, and may not result in “losing” any answer.

Note that \( d \) receives all the answers. Indeed, one can show that the distributed algorithm previously outlined always terminates and computes the proper answer. (On an infinite Web, it would terminate if and only if the set of \( o' \) such that for some prefix \( u \) of some word in \( p \) there is a path from the source of the query to \( o' \) is finite.)

Remark 3.1. The distributed algorithm outlined above requires extending the capabilities of current crawlers. Indeed, existing Web crawlers take, for the time
being, a centralized approach. In particular, the http protocol does not allow carrying information when traveling from site to site.

3.2. Optimization of Path Queries

The basic distributed processing algorithm can be improved in many ways by taking into account additional information that might be available. In keeping with the spirit of the distributed scenario, we assume such information is local to each site. More precisely, we assume that an object \( o \) may have local information of the form \( p = q \) or \( p \leq q \), meaning that \( p(o, I) = q(o, I) \) or \( p(o, I) \leq q(o, I) \). We refer to such properties as path constraints.

Path constraints may reflect various kinds of information. First, they may reflect structural information about neighboring Web pages. For example, consider the two paths:

\[
\begin{align*}
  p_1 &= \text{CS-Department DB-group Ullman Classes cs345} \\
  p_2 &= \text{CS-Department Courses cs345}.
\end{align*}
\]

It may be the case that starting from some site \( \text{Stanford} \), the paths \( p_1 \) and \( p_2 \) lead to the same object. Thus, the path constraint \( p_1 = p_2 \) holds at site \( \text{Stanford} \). Similarly, at the site \( \text{CS-Department} \) one could have the constraint

\[
\Sigma^* \text{Stanford-CS-Main} = \epsilon,
\]

stating that all paths starting at site \( \text{CS-Department} \) whose final label is \( \text{Stanford-CS-Main} \) lead back to that site.

Path constraints also naturally arise from caching frequently asked queries. More precisely, the answer to query \( q \) at site \( o \) could be saved and accessed from \( o \) by links labeled \( l_q \). This would yield the path constraint \( q = l_q \) and a rapid way to evaluate \( q \) by simply evaluating \( l_q \). Similar constraints arise from the presence of “mirror sites,” which are duplications of frequently accessed sites.

How can path constraints be used? The hope is that they may allow more efficient evaluation of path queries. For instance, the query may ask for the page \( p_1 \) (as above) and the system may decide to substitute it with the page \( p_2 \) if this page is available locally and it is known that it contains the same information.

So, in general, the query processor at each site may use the path constraints holding at the site to replace the query to be executed by a simpler query. We are not concerned here with what “simpler” means; this could potentially involve a cost measure using information not captured by our basic model, such as locality information, cost of accessing different sites in the network, etc. Regardless of the cost measure, the basic problem laying at the core of this approach is testing the implication of relationships among queries by the given constraints. Thus, we must be able to answer the question:

Given a finite set \( E \) of path constraints of the form \( p_i = q_i \) or \( p_i \leq q_i \) and two path queries \( p, q \), is it true that \( p(o, I) = q(o, I) \) or \( p(o, I) \leq q(o, I) \) for each \( (o, I) \) satisfying \( E \)?
We examine this problem in detail in the next section. The following illustrates further how such inferences might be used in query optimization.

**Examples.**

1. Suppose we know that every path ending by label $l$ returns to the source site, i.e., $\Sigma^*l = e$. Suppose query $p = (la + lb)^* d$ must be executed at this site. It can be shown that $p$ is equivalent to $(a + b) d$. This query is likely to be simpler than the original; in particular, it is nonrecursive and so it is guaranteed to terminate.

2. Suppose the path constraint $ll \subseteq l$ holds at the source site. Consider the query $p = l^*$. It can be shown that $l^* = l + e$ so $p$ can be replaced by the query $l + e$.

3. Suppose the query $(ab)^*$ has been cached and labeled $l$, so that the constraint $l = (ab)^*$ holds. Consider the query $p = a(ba)^* c$. One can show that $p = lac$. In other words, $p$ can be evaluated by sending the query $ac$ to the cached objects.

### 4. IMPLICATION OF REGULAR PATH CONSTRAINTS

In this section, we consider the implication problem for path constraints. We first formalize the problem and relate it to well-known problems in rewrite systems and logic. We then study several natural special cases. These concern constraints between "words" instead of arbitrary regular expressions. We are able to obtain decision procedures of complexity \textit{PTime} for the implication of word constraints and of complexity \textit{PSpace} for implication of path constraints by word constraints. As a side effect we develop tools that are of interest in their own right. For example, we use them to show that, given a finite set of word equalities, the boundedness problem for path queries is decidable; that is, it is decidable if a path query is equivalent to a non-recursive path query, given a finite set of word equalities.

**Path constraints.** We now formalize the implication problem for path constraints and mention related problems in logic and rewriting systems. In the following we fix a finite set of labels $\Sigma$ (see Section 2).

**Definition 4.1.** A (regular) path inclusion is an expression of the form $p \subseteq q$ where $p, q$ are regular expressions over $\Sigma$. An instance $(o, I)$ satisfies a path inclusion $p \subseteq q$, denoted $(o, I) \models [p \subseteq q]$, if $p(o, I) \subseteq q(o, I)$; $(o, I)$ satisfies a set $E$ of path inclusions, denoted $(o, I) \models E$, if it satisfies each inclusion in $E$. A finite set $E$ of path inclusions implies a constraint $p \subseteq q$, denoted $E \models [p \subseteq q]$, if each instance $(o, I)$ satisfying $E$ also satisfies $[p \subseteq q]$.

If $p, q$ are words, i.e., simply sequences of labels, the path inclusion $p \subseteq q$ is called a word inclusion (e.g., $a b c \subseteq d e$). The expressions obtained by replacing $\leq$ by $=$ are called, respectively, path equalities (e.g., $a(b + c)^* = d e$) and word equalities (e.g., $a b c = d e$). A path constraint is a path inclusion or a path equality, and similarly for word constraint. Equality constraints can of course be expressed by inclusion constraints, but equality is an important and well-behaved special case.

We start by pointing to two problems in rewrite systems and logic that are related to the implication problem for path constraints.
**Rewrite systems.** Consider first word inclusions. Suppose that we know \( u_1 \subseteq u_2 \) and \( u_2 \subseteq u_4 \). Then it seems natural to infer, for instance, that \( u_1 \subseteq u_3 \subseteq u_5 \subseteq u_2 \subseteq u_3 \subseteq u_5 \subseteq u_4 \). We will present a rewrite system that is sound and complete for word constraints. This will then be used to obtain a decision procedure for this case. Note that in the general case, one cannot decide whether a word can rewrite into another word using an arbitrary system of rewrite rules (a semi-Thue system) \([18]\). Our case differs from the general case in that rewrite rules are applied only to prefixes of words. See \([14]\) for a comprehensive survey of rewrite systems.

**First-order logic with two variables.** In the particular context of word constraints, the implication problem can be stated in terms of first-order logic. Moreover, only two variables are needed. Then the decidability of the implication problem for word constraints follows from known results about first-order logic with two variables (\(\text{FO}^2\)). Indeed, satisfiability of \(\text{FO}^2\) sentences (with relational vocabulary and constants) is decidable \([25]\), and the implication problem for word constraints can be reduced to satisfiability of such an \(\text{FO}^2\) sentence. However, the complexity of testing \(\text{FO}^2\) satisfiability is doubly exponential in the formula \([25]\) and exponential in the model size \([16]\). In contrast, our direct proof provides a \(\text{PTime}\) test for word constraint implication (in the size of the words). Furthermore, results about \(\text{FO}^2\) and its extensions are no longer of help for implication of full path constraints, where recursion is present in the form of the Kleene closure. Indeed, for the extensions of \(\text{FO}^2\) with recursion/fixpoint that have so far been studied, satisfiability was shown to be undecidable \([17]\). In this light, decidability of implication for path constraints comes as a welcome surprise.

4.1. Path Constraint Implication

In this section, we study path constraint implication. We first prove that implication of path constraints is decidable (in \(2\text{-expspace}\)). The idea of the proof is to show that if an implication \( E \models p \subseteq q \) is violated by an instance (finite or infinite), then one can find a finite instance witnessing the violation whose size is doubly exponential in the size of \( E, p, q \). Observe that this also demonstrates that for path constraints, finite and unrestricted implication coincide.

**Theorem 4.2:** (1) If \( \bigwedge_{i \in [1 \ldots m]} p_i \subseteq q_i \) \( \not\models p_0 \subseteq q_0 \), then there is some instance \((\alpha, J)\), of size doubly exponential in the total size of \( \big\{ p_i, q_i \big\}_{i \in [0 \ldots m]} \), such that \( (\alpha, J) \models \bigwedge_{i \in [1 \ldots m]} p_i \subseteq q_i \) and \((\alpha, I) \not\models p_0 \subseteq q_0 \). (2) Implication of path constraints is decidable in \(2\text{-expspace}\).

**Proof.** Let \((\alpha, I)\) be an instance (possibly infinite) such that \((\alpha, I) \models \bigwedge_{i \in [1 \ldots m]} p_i \subseteq q_i \) and \((\alpha, I) \not\models p_0 \subseteq q_0 \).

Consider standard nfa’s for the \( p_i, q_i, i \in [0 \ldots m] \), and the nfa \( F \) that is the product of these nfa’s. Let \( f \) be the start state of \( F \) (i.e., the product of the start states for the nfa’s for the \( p_i, q_i \)) and \( \delta_F \) be its transition function. For each vertex \( o' \) in \( I \), let

\[
\text{states}(o') = \{ s \mid \text{there is a path } u \text{ from } o \text{ to } o' \text{ such that } s \in \delta_F(f, u) \}.
\]
For each set $S$ of states in $F$, let $o_S$ be a distinct new vertex. Consider the graph homomorphism $\mu$ that replaces each vertex $o'$ in $I$ by $o_S$, where $S = states(o')$. Let $o = \mu(o)$ and $J = \mu(I)$.

We prove that for each $p = p'$ or $q_i$, $0 \leq i \leq m$, and each $o'$,

$$o' \in p(o, I) \iff \mu(o') \in p(o, J).$$

For suppose that $(\dagger)$ holds. Then for $i \geq 1$, $p_i(o, J) = \mu(p_i(o, I)) \subseteq \mu(q_i(o, I)) = q_i(o, J)$; and for $o'$ in $p_d(o, I) - q_d(o, I)$, $\mu(o')$ in $p_d(o, J) - q_d(o, J)$, so $(1)$ is proved.

Consider $(\dagger)$. Clearly, it is sufficient to show that for each vertex $o'$ in $I$, $states(o') = states(\mu(o'))$). The inclusion $states(o') \subseteq states(\mu(o'))$ follows immediately by the definition of homomorphism. Consider the inclusion $states(\mu(o')) \subseteq states(o')$. Let $s \in states(\mu(o'))$. There exists a path $u$ from $\omega$ to $\mu(o')$ such that $s \in \delta_\mu(f, u)$. We prove by induction on $|u|$ that $s \in states(o')$. If $u = v$ then $s \in states(\omega)$ and $\mu(o') = \omega = \mu(o)$. From the latter equalities it follows that $states(\omega) = states(o')$, so $s \in states(o')$. Now suppose $u = va$ with $a \in \Sigma$, and the statement holds for words shorter than $u$. Let $f'$ be a state in $\delta_\mu(f, v)$ such that $s \in \delta_\mu(f', a)$. There exist vertices $o_1, o_2$ in $I$ such that there is an $a$-link from $o_1$ to $o_2$, $\mu(o_2) = \mu(o')$, and $\mu(o_1) \in v(o, J)$. By the induction hypothesis, $states(\mu(o_1)) \subseteq states(o_1)$ so there exists $v'$ such that $f' \in \delta_\mu(f, v')$ and $o_1 \in v'(o, I)$. Consider the path $v'a$ in $I$, we have that $s \in \delta_\mu(f, v'a)$ and $v'a$ is a path from $o$ to $o_2$. Thus, $s \in states(o_2) = states(o')$. This proves $(\dagger)$.

To summarize, we constructed a finite instance $(o, J)$ such that

- $(o, J) = [\bigwedge_{i=1}^{m} p_i \subseteq q_i]$ and
- $(o, J) \not\equiv p_0 \subseteq q_0$.

Furthermore, the size of $(o, J)$ is doubly exponential in $|E| + |p_0| + |q_0|$. Thus, one can test implication by considering all instances up to this size, which takes $2$-EXPSPACE. It is worth noting that the topmost exponential is in the number of constraints, not their size. Thus, the complexity is EXPSPACE if the number of path constraints involved is bounded by a constant.

Although the above result shows the decidability of implication, the test of implication it provides has some drawbacks. First, its complexity is high. Second, it does not provide real insight into the interplay of path constraints. Such insight might be better served by a sound and complete axiomatization of path constraint implication. However, obtaining such an axiomatization appears to be highly non-trivial. Note that even an axiomatization of classical regular expression equivalence (in the absence of constraints) is far from obvious (see the set of axioms provided in [29]).

4.2. Word Constraints

We next consider two particular cases of the implication problem. We show that for word constraints, implication is decidable in PTIME. We are then able to extend this result to implication of full path constraints by word constraints, with PSPACE
complexity. Note that deciding the equivalence of regular expressions is by itself $\text{pspace}$-complete (in absence of constraints) [15], so this is the best one can do for that case. Finally, we consider the special case of word equality.

Whenever we consider a finite set $E$ of word inclusions, we will assume that if $u \subseteq w$ is in $E$, the constraint $e \subseteq u$ is also in $E$. This is convenient because $e(o, I)$ always consists of the single vertex $o$, so $u \subseteq e$ and $e \not\subset u$ would imply that $u = \emptyset$. This would introduce a new category of emptiness constraints that we wish to avoid. We will prove the following.

**Theorem 4.3.** (i) Implication of a word constraint by a set of word constraints can be tested in $\text{ptime}$. (ii) Implication of a path constraint by a set of word constraints can be tested in $\text{pspace}$.

The proof of the theorem requires four lemmas and involves a rewrite system of words. We associate to each inclusion $u \subseteq v$ in $E$ a rewrite rule $u \xrightarrow{E} v$. Let $\xrightarrow{E}$ be the binary relation on words defined as follows: $z \xrightarrow{E} t$ iff there is a finite sequence of words $w_1 \cdots w_n$ (for $n \geq 1$) such that $z = w_1$, $t = w_n$, and for each $i$, $1 \leq i < n$, $w_i = xw$ and $w_{i+1} = yw$ for some $x \subseteq y$ in $E$ and some word $w \in \Sigma^*$. It is useful to note that $\xrightarrow{E}$ is the reflexive, transitive, right-congruent closure of $E$.

The first lemma provides a connection between implication of word constraints and derivation by the corresponding rewrite system.

**Lemma 4.4.** Given a finite set $E$ of word constraints, $\xrightarrow{E}$ is sound and complete for implication of word constraints. That is, for each $E$ and $u, v \in \Sigma^*$, $E \models u \subseteq v$ iff $u \xrightarrow{E} v$.

**Proof.** It is quite obvious that if $u \xrightarrow{E} v$ then $E \models u \subseteq v$ (soundness of rewriting). To prove the converse (completeness), we show that

for each $k$, there is a finite instance $(o, I)$ that satisfies $E$ and such that for each $u, v$ shorter than $k$, if $(o, I) \models u \subseteq v$ then $u \xrightarrow{E} v$.

For suppose $(\dagger)$ holds and $E \models u \subseteq v$. Let $k$ be larger than $u, v$ and $(o, I)$ the instance provided for by $(\dagger)$. Since $E \models u \subseteq v$ and $(o, I) \models E$, $(o, I) \models u \subseteq v$. By $(\dagger)$, $u \xrightarrow{E} v$.

To prove $(\dagger)$, let $\approx$ be the equivalence relation on $\Sigma^*$ defined by $u \approx v$ iff $u \xrightarrow{E} v$ and $v \xrightarrow{E} u$. Let $\hat{u}$ denote the equivalence class of a word $u$ with respect to $\approx$. Let $\prec$ be the partial order on the equivalence classes of $\approx$ defined by $\hat{u} \prec \hat{v}$ iff $u \xrightarrow{E} v$. (note this is well defined).

Let $\mathcal{C} = \{\hat{u} \mid |u| \leq k\}$. We build an instance $(o, I)$ by “populating” each class $\hat{u}$ of $\mathcal{C}$ with a finite set of vertices $\text{obj}(\hat{u})$ such that $u(o, I) = \text{obj}(\hat{u})$, as follows. For each $\sigma \in \mathcal{C}$, let $o_\sigma$ be a distinguished vertex. Let $\text{obj}(\sigma) = \{o_\sigma \mid \psi \in \mathcal{E}, \psi < \sigma\}$, for each $\sigma \in \mathcal{C}$. The instance $(o, I)$ is defined as

1. the vertices are $\{o_\sigma \mid \sigma \in \mathcal{C}\}$;
2. $o$ is $o_e$; and
3. for each $u$, $|u| < k$ and $a$ in $\Sigma$, there is an $a$-edge from $o_a$ to each $o'$ in $\text{obj}(\hat{u})$. 

It is sufficient to show that
\[ u \in \Sigma^*, \ |u| \leq k, u(o,I) = \text{obj}(\hat{u}). \]

For suppose that (+) holds. Then \((o,I) \models u \subseteq v\) implies \(u(o,I) \subseteq v(o,I)\) which in turn implies \(\text{obj}(\hat{u}) \subseteq \text{obj}(\hat{v})\) which again implies \(\hat{u} \prec \hat{v}\) which finally implies \(u \xrightarrow{E} v\).

Before proving (+), we illustrate the construction of the instance \((o,I)\) by an example. Let \(\Sigma = \{a\}, E = \{a^2 \sqsubseteq a\}, \) and \(k = 3\). Clearly, \(\hat{e} = e, \ a^2 \overset{e}{\rightarrow} \{a'\}\) (denoted \(a'\) by slight abuse of notation), \(i > 0\). Also, \(a^1 \prec a^2 \prec a, \) and \(\text{obj}(a_e) = \{a_e\}, \ \text{obj}(a_e^i) = \{a_e^i\}, \ \text{obj}(a_e^o) = \{a_e^o\}, \) \(\text{obj}(a_o) = \{a_o, a_o^2, a_o^3\}\). The instance \((o,I)\) is defined as follows: the set of nodes of \(I\) is \(\{a_e, a_e^2, a_e^3, a_o, a_o^2, a_o^3\}\), \(o = o_e\), and the edges (whose label \(a\) is omitted) are represented in Fig. 4. Clearly, \(e(o,I) = \{a_e\}, \ a(o,I) = \{a_o, a_o^2, a_o^3\}, \ a^2(o,I) = \{a_o^2, a_o^3\}, \) and \(a^3(o,I) = \{a_o^3\}\), as desired.

We prove (+) by induction. First observe that \(e\) is a least element for \(\prec\) since for each \(u \in e\), we also have \(e \subseteq u\). So \(\text{obj}(\hat{e}) = \{o\}\) and \(e(o,I) = \text{obj}(\hat{e}) = \{o\}\). Now suppose (by induction on the size of \(u\)) that \(u(o,I) = \text{obj}(\hat{u})\) for \(|u| < k\) and let \(a\) be in \(\Sigma\). Then \(ua(o,I)\) contains \(\text{obj}(\hat{ua})\) by construction of \((o,I)\). Now, let \(o'\) be in \(ua(o,I)\). Then there exists \(v \prec u\) (so that \(a_x\) is in \(\text{obj}(\hat{u})\)) and an \(a\)-edge from \(a_x\) to \(o'\). By (iii), \(o'\) is in \(\text{obj}(\hat{ua})\). But since \(v \prec u, \ ua \prec \hat{ua}, \) so \(\text{obj}(\hat{ua}) \subseteq \text{obj}(\hat{ua})\). Thus, \(o'\) is in \(\text{obj}(\hat{ua})\), and \(ua(o,I) = \text{obj}(\hat{ua})\). This proves (+).

Note that the boundedness restriction in the construction of \((o,I)\) in the preceding proof cannot be removed. Indeed, there exists a finite set of constraints \(E\) such that there is no \((\text{finite or infinite})\) instance satisfying exactly the constraints implied by \(E\). To see an example, consider again \(E = \{a^2 \sqsubseteq a\}\). Observe that \(E\) implies in particular: \(\cdots \sqsubseteq a' \sqsubseteq a^2 \cdots \sqsubseteq a\) but not \(a' = a^2 \cdots\). However, in each fixed instance there are only finitely many outgoing edges from the source, so each instance satisfying \(a^2 \sqsubseteq a\) must also satisfy \(a' = a^2 \cdots\) for some \(i\).

The \textsc{ptime} bound on testing word implication is obtained using the next lemma that focuses on the set of all words that rewrite to a particular word \(v\). More precisely, consider the set of words \(\text{RewriteTo}(v) = \{u \mid u \in \Sigma^*, u \xrightarrow{E} v\}\). The lemma shows that \(\text{RewriteTo}(v)\) is a regular language. To see this, note first that one can easily build a pushdown automaton (pda) that accepts \(\text{RewriteTo}(v)\). The pda works as follows. It first puts the input word \(a\) on the stack, then starts simulating the rewrite rules by rewriting prefixes of the stack (using pda moves). The pda is very particular in that it first reads its entire input and places it on the stack. The crux of the proof consists in showing that such a pda can actually be simulated by an nfa.
Lemma 4.5. Let $E$ be a finite set of word constraints and $v$ a word in $\Sigma^*$. The set

$$\text{RewriteTo}(v) = \{ u \mid u \in \Sigma^*, u \xrightarrow{E} v \}$$

is a regular language recognized by an nfas constructible in polynomial time from $E$ and $v$. In particular, $u \xrightarrow{E} v$ can be decided in $\text{ptime}$.

Proof. It is convenient to consider the language that consists of the reverse of the words in $\text{RewriteTo}(v)$: $L_a = \{ u^R \mid u \in \text{RewriteTo}(v) \}$. We show that $L_a$ is context-free, so $\text{RewriteTo}(v)$ is context-free. Indeed, a pushdown automaton (pda) $A$ accepting $L_a$ works as follows. First, $A$ places the input $u^R$ on the stack, thus reversing it. Once $u$ is on the stack, the pda $A$ nondeterministically performs a sequence of prefix substitutions using the rewrite rules of $E$. Last, $A$ guesses that $v$ has been constructed and pops the stack to check this. The computation accepts if $v$ is found on the stack.

To see that $\text{RewriteTo}(v)$ is regular, consider the computation of the pda $A$ once $u$ is on the stack. Suppose that at this time $A$ is in some state $q_0$. Then $A$ adds and removes symbols from the stack until the stack is empty, at which time the pda accepts or rejects. Thus, each symbol $x$ in $u$ eventually becomes the top of the stack and is popped. So, consider the triples $\text{move}(q, x, q')$ meaning that the pda $A$, starting in state $q$ with $x$ as the top of the stack, can reach state $q'$ after popping $x$. Consider the nfas $N_A$ whose states are those of $A$, with start state $q_0$ and the same accepting states as $A$, and with $\text{move}$ as transition function. By construction, $N_A$ accepts precisely $\text{RewriteTo}(v)$.

We still have to show that $N_A$ can be constructed from $E$ and $v$ in $\text{ptime}$. The construction of the pda $A$ described above is clearly in $\text{ptime}$ with respect to $E$ and $v$. To construct the transition function $\text{move}$ of $N_A$, we have to decide, for each pair of states $q, q'$ of $A$, and symbol $x$, whether $A$ is in state $q$ and with $x$ as top of the stack may reach state $q'$ after popping $x$. To do this, we build from $A, q, q'$, and $x$ another pda $A(q, q', x)$. The pda $A(q, q', x)$ never reads any input (all of its moves are $\varepsilon$-moves with respect to the input). $A(q, q', x)$ first writes $x$ on the stack, then simulates $A$ starting from state $q$. $A(q, q', x)$ accepts when it reaches an empty stack and is in state $q'$. Clearly, such a pda $A(q, q', x)$ can be build from $A$ in $\text{ptime}$. Next, observe that $\text{move}(q, x, q')$ holds iff $\varepsilon$ is accepted by $A(q, q', x)$, which can be checked in $\text{ptime}$. In summary, the nfas $N_A$ accepts $\text{RewriteTo}(v)$ and can be constructed in $\text{ptime}$ from $E$ and $v$. Finally, one can check whether a word $u$ is accepted by the nfas $N_A$ in $\text{ptime}$.

Lemmas 4.4 and 4.5 together provide the $\text{ptime}$ test for implication of word constraints, and thus prove (i) of Theorem 4.3. To show part (ii) of the theorem, we use two additional lemmas. The first relates implication of path constraints to implication of word constraints:

Lemma 4.6. Let $E$ be a finite set of word constraints and $p, q$ regular expressions. If $E \models p \subseteq q$ then for each $u \in L(p)$ there exists $v \in L(q)$ such that $E \models u \subseteq v$.

Proof. Suppose $E \models p \subseteq q$. Consider a word $u \in L(p)$. Evidently, $E \models u \subseteq q$. We must show that there is some $v \in L(q)$ such that $E \models u \subseteq v$. Let $k$ be an integer larger
than the lengths of $u$ and of any word in $E$. Consider the instance $(o, I)$ constructed for $E$ and $k$ in the proof of Lemma 4.4 and recall the notation developed there. Since $(o, I)$ satisfies $E$, it must also satisfy $u \subseteq q$. Note that $u(o, I) \neq \emptyset$ and $w(o, I) = \emptyset$ for each $w$ such that $w \notin \mathcal{C}$. It follows that

$$u(o, I) \subseteq \cup \{ w(o, I) \mid w \in L(q), w \notin \mathcal{C} \}. \tag{*}$$

Recall that by construction there is a distinguished vertex $o^u$ in $u(o, I)$ such that for all words $w$ with $w \notin \mathcal{C}$, $o^u \in w(o, I)$ if $u(o, I) \subseteq w(o, I)$. This together with $(*)$ imply that there must exist $v \in q$ such that $v \notin \mathcal{C}$ and $o^u \in v(o, I)$, so $u(o, I) \subseteq v(o, I)$. It follows that $u \not< v$, so $u \not\rightarrow v$ and $E \models u \subseteq v$.

It is worth noting that generally an instance $(o, I)$ may satisfy $p \subseteq q$ without it being the case that each word $u$ in $L(p)$ is included in some word $v$ in $L(q)$ (e.g., consider $a \leq b + c$, or $a \leq b^*$). The above lemma shows, however, that this must happen if $p \subseteq q$ is implied by a finite set of word constraints.

The pspace bound is obtained using an extension of Lemma 4.5.

**Lemma 4.7.** Let $E$ be a finite set of word constraints and $p$ a regular expression over $\Sigma$. The set

$$\text{RewriteTo}(p) = \{ u \mid u \in \Sigma^*, \exists v \in L(p)(u \rightarrow^E v) \}$$

is a regular language recognized by an nfsa constructible in polynomial time from $E$ and $p$.

**Proof.** To show that $\text{RewriteTo}(p)$ is regular, we use the same technique as in Lemma 4.5. We place $u^R$ on the stack, nondeterministically perform rewritings using $E$, guess that a word $v$ in $L(p)$ has been constructed, and finally, check whether $v$ is in $L(p)$. It follows, as in the proof of Lemma 4.5, that $\text{RewriteTo}(p)$ is a regular language and that an nfas recognizing it can be constructed in ptime from $E$ and $p$.

By Lemma 4.6 and 4.4,

$$E \models p \subseteq q \quad \text{iff} \quad L(p) \subseteq \text{RewriteTo}(q).$$

We can construct an nfas $F_p$ for $L(p)$ and (by Lemma 4.7) an nfas $F_q$ for $\text{RewriteTo}(q)$ in ptime with respect to $p$ and $q$. Next, we can construct in ptime an nfas $F_{p+q}$ for $L(p) \cup \text{RewriteTo}(q)$. It now suffices to check whether $L(F_p) = L(F_{p+q})$. This can be achieved in pspace. (Note that fsa inequivalence is pspace-complete by reduction from regular expression non universality [15].) This provides the pspace test of implication of path constraints by word constraints and proves part (ii) of Theorem 4.3.

### 4.3. Word Equalities

What is different about equality? Obviously, decidability of the implication problem for path inclusions yields a decision procedure for implication of path equalities.
More precisely, this yields a \textit{ptime} test for implication of word equalities, a \textit{pspace} test for the implication of path equalities by word equalities, and a 2-\textit{expspace} test for implication of path equalities. However, it turns out that equality has some remarkably nice properties. These are due to the following technical facts:

1. for each finite set $E$ of word equalities there exists a “true” Armstrong instance for $E$, i.e., an infinite instance that satisfies \textit{precisely} the path equalities implied by $E$; and

2. the interesting information contained in the Armstrong instance for $E$ occurs at a bounded distance from the source, which allows us to compute a finite “summary” of the Armstrong instance.

The existence of the Armstrong instance is shown next.

**Proposition 4.8.** Let $E$ be a finite set of word equalities. There exists an instance $(o, I)$ such that for each $u, v$,

\[ u(o, I) = v(o, I) \quad \text{iff} \quad E \models u = v. \]

**Proof.** The instance $(o, I)$ is built as follows. Let $\equiv$ be the smallest equivalence relation over $\Sigma^*$ that contains $E$ and is a right-congruence. The set of vertices in $I$ consists of the equivalence classes of $\equiv$, and $o = o_e$. For each $u, a$, there is an $a$-edge from $u$ to $ua$.

First observe that this is, indeed, an instance; i.e., it has finitely many outgoing edges from every vertex; if $\hat{u} = \hat{v}$, then $\hat{u}a = \hat{v}a$ (right congruence), so there is a single outgoing $a$-edge from every vertex. Also note that $u(o, I) = \hat{u}$. Therefore $u(o, I) = v(o, I)$ iff $\hat{u} = \hat{v}$ iff $u \models E \models v$. By Lemma 4.4 $u \models E \models v$ iff $E \models u = v$. Thus, $u(o, I) = v(o, I)$ iff $E \models u = v$. \qed

We call the instance $(o, I)$ constructed in the above proof the \textit{Armstrong instance} of $E$. This is generally an infinite instance. However, we show that all interesting information in $(o, I)$ is contained at some bounded distance from the source. Given an instance $(o, I)$, let the $K$-sphere (around $o$) consist of the restriction of $I$ to vertices at distance at most $K$ from $o$. Using this, the structure of the Armstrong instance is very particular and can be captured as follows (see Fig. 5).

**Lemma 4.9.** Let $E$ be a finite set of word equalities and $(o, I)$ the Armstrong instance for $E$. There exists an integer $K$ such that each vertex outside the $K$-sphere has indegree 1, and there is no edge with tail outside and head inside the $K$-sphere.

**Proof.** We proceed in two stages. We build a first sphere and then extend it to the desired one (see Fig. 5). Let $M$ be the maximum length of words in $E$. We prove that each vertex outside the $M$-sphere has indegree 1. \quad (*

Suppose $\hat{u}$ is outside the $M$-sphere and has an incoming $a_1$-edge from $\hat{v}_1$ and another incoming $a_2$-edge from $\hat{v}_2$. We can assume without loss of generality that $v_1$ and $v_2$ are the shortest words in their equivalence class. Observe that $|v_1| \geq M$
FIG. 5. The Armstrong instance.

and $|v_2| \geq M$; otherwise $\hat{u}$ would be in the $M$-sphere. Since $v_1a_1 = v_2a_2 = \hat{u}$, $v_1a_1 \rightarrow v_2a_2$. Consider the derivation of $v_2a_2$ from $v_1a_1$. Since $|v_1|$ is larger or equal than the maximum length of words occurring in $E$, and since it is also the shortest word in its equivalence class, the rewriting of $v_1a_1$ never replaces the last letter. Thus, $a_1 = a_2$ and $v_1w \in E v_2$, so $v_1 = v_2$.

From $(\ast)$, one can see that each vertex out of the $M$-sphere has indegree 1. The proof is not yet complete, since a path may return to the $M$-sphere after having left it. However, we show that such paths have bounded length. Then we can find a yet larger sphere that satisfies the lemma.

Consider a path leaving the $M$-sphere and returning back to it. Let $\hat{u}$ be the last vertex on the path which is in the $M$-sphere before the path leaves the $M$-sphere, and let $\hat{v}$ be the first vertex on the path which is next inside the $M$-sphere. We may assume without loss of generality that $u$ and $v$ are shortest in their equivalence class, so in particular $|u| = M$ and $|v| \leq M$. Suppose the path from $\hat{u}$ to $\hat{v}$ spells the word $w$. Thus, no vertex along $w$ is in the $M$-sphere except the last one which is $\hat{v}$. We know that the regular language $\text{RewriteTo}(v)$ is accepted by an nfa $F$ that can be constructed from $E$ and $v$ in $\text{ptime}$. The number $N$ of states in $F$ is polynomial in $E$ and $v$, i.e., it is polynomial in $M$. Let us fix $K = M + N$.

We will prove that $|w| \leq N$, so that any point on the path is within the $K$-sphere. Assume towards a contradiction that $|w| > N$. Observe that $\bar{w} = \hat{w}$, so $ww \in \text{RewriteTo}(v)$. Consider the run of $F$ on input $ww$. The automaton accepts $ww$ and since $|w|$ is larger than the number of states of $F$, $F$ goes twice through the same state, say after reading $ux$ and $uxy$, where $w = xyz$ and $y \neq \epsilon$. We use a pumping argument. The word $uxyz$ is also accepted by $F$ so $uxyz \rightarrow uxyyz$. Consider the derivation of $uxyyz$ from $uxyz$. Recall that $|u| = M$ and $u$ is the shortest word in $\hat{u}$. Thus $u$ cannot shrink in the derivation and $u \rightarrow uv'$ for some $v'$ such that $uxyz = uyyz$. It follows that $v'$ is a prefix of $xy$ and so $uv'$ leads to a vertex along the path $w$. But $uv' = \hat{u}$ so that vertex is in the $M$-sphere, which contradicts our assumption that all vertices along $w$ are outside the $M$-sphere. Thus $|w| \leq N$, which concludes the proof. $\blacksquare$
The significance of the above property is that all word equalities implied by $E$ follow by right-congruence from equalities of words leading to vertices inside the $K$-sphere. Thus, all “interesting” information can be found within the $K$-sphere around $o$.

This provides a valuable tool for reasoning about implication of path equalities by word equalities and dramatically simplifies a number of problems. We next show how the Armstrong relation can be used to solve the question of boundedness of a path query under given word equalities. We prove that it is decidable whether a path query is equivalent to a query without recursion assuming a given set of word equalities. Furthermore, an equivalent finite query can be effectively constructed if such a query exists. As illustrated in Section 2, this a problem of significant practical interest.

**Theorem 4.10.** It is decidable, given a finite set $E$ of word equalities and a regular path expression $p$, whether $E \models [p = q]$ for some regular path query $q$ where $L(q)$ is finite. Furthermore, if such $q$ exists, it can be constructed in ExpTime from $E$ and $p$.

**Proof.** Consider the Armstrong relation $(o, I)$ associated with $E$, and the $K$-sphere constructed in Lemma 4.9. Recall that all vertices outside the $K$-sphere have indegree one and no path that leaves the $K$-sphere ever returns. This means that all paths leaving the $K$-sphere which are distinct outside the $K$-sphere lead to distinct vertices. So $p$ is bounded iff the set of words in $p$ that yield distinct paths outside the $K$-sphere is finite. This can be tested by associating an fsa $F$ with $(o, I)$ as follows. Its states are all the vertices in the $K$-sphere plus one new state out. Its transitions are all the labeled edges within the $K$-sphere plus, for each $o$, one $a$-edge $(o$, out$)$ if there is an $a$-edge going from $o$ to some vertex outside the $K$-sphere. Finally, for each $a$, there is an $a$-edge from out to out. The start state of $F$ is $o$ and the accepting-state is out. Note that the size of $F$ is bounded by an exponential in the size of $E$. Next, it is easily seen that $p$ is bounded iff the language

$$\{ v \mid uv \in L(p), u \in L(F) \}$$

is finite. This language is the quotient of $L(p)$ by $L(F)$, and it is regular. Its finiteness can be tested in PTime with respect to $F$ and $p$. Altogether, the above yields an algorithm of complexity ExpTime (with respect to $E$ and $p$) to test if $p$ is bounded assuming $E$ and, if so, to construct a finite path query equivalent to it under $E$. □

It remains open whether boundedness of a path query assuming a set of full path constraints is decidable.

**5. CONCLUSION**

Several technical questions relating to implication of path constraints remain open. They include

- Extending the results to more expressive path constraints and queries, such as those used in languages like Lorel [5] and UnQL [11]. This includes, for example, the use of variables at intermediate points in the paths.
• Investigating whether the 2-EXPSPACE complexity of path constraint implication in the general case can be improved.

• Devising a sound and (if possible) complete axiomatization for path constraint implication. Besides allowing to better understand the interplay of path constraints, such an axiomatization may yield rewrite rules of practical use in simplifying path queries under given path constraints.

• Investigating special cases of practical interest. For example, instances whose nodes have at most one outgoing edge with a given label seem to be of practical interest. This property may simplify some of the problems studied here.

While the path constraint implication problem is clearly central to optimization of path queries in the presence of path constraints, specific evaluation strategies making use of the techniques developed here remain to be explored. Some of these were mentioned in the paper, such as replacing a query by a nonrecursive equivalent, if such exists (the boundedness problem; see Theorem 4.10). Another problem briefly raised in our discussion of optimization in Section 2 is the use of cached path queries to answer a given path query, assuming certain path constraints are satisfied. Clearly, this can also be solved using our results, by exhaustive search of Boolean combination of the cached queries and testing equivalence to the given query under the constraints. The problem can be refined to making partial use of cached queries rather than using them to fully answer the given query. Other problems which could benefit from our techniques include minimizing multiple visits to the same site, preemptive caching of subqueries, allowing software agents to carry along information accumulated during their traversal of the graph, and using more sophisticated communication among them.

In our basic distributed query evaluation scenario, we assumed that software agents evaluating the query have certain capabilities, such as maintaining persistent bookkeeping information at the sites they visit and running optimization algorithms on host sites. However, such capabilities are not available in existing software agents implementing crawlers. It would be interesting to study the interplay between distributed query evaluation strategies and the capabilities of software agents. Understanding the trade-offs involved should provide useful information in the design of future software agents and, conversely, for developing query evaluation strategies subject to restricted agent capabilities.

REFERENCES


27. Y. Papakonstantinou, S. Abiteboul, and H. Garcia-Molina, Object fusion in mediator systems, in “Proceedings of the Twenty-Second International Conference on Very Large Data Bases, Bombay, India, 1996.”


