The Complexity of Regular Expressions and Property Paths in SPARQL

KATJA LOSEMANN, University of Bayreuth
WIM MARTENS, University of Bayreuth

The World Wide Web Consortium (W3C) recently introduced property paths in SPARQL 1.1, a query language for RDF data. Property paths allow SPARQL queries to evaluate regular expressions over graph-structured data. However, they differ from standard regular expressions in several notable aspects. For example, they have a limited form of negation, they have numerical occurrence indicators as syntactic sugar, and their semantics on graphs is defined in a non-standard manner.

We formalize the W3C semantics of property paths and investigate various query evaluation problems on graphs. More specifically, let x and y be two nodes in an edge-labeled graph and r be an expression. We study the complexities of (1) deciding whether there exists a path from x to y that matches r and (2) counting how many paths from x to y match r. Our main results show that, compared to an alternative semantics of regular expressions on graphs, the complexity of (1) and (2) under W3C semantics is significantly higher. Whereas the alternative semantics remains in polynomial time for large fragments of expressions, the W3C semantics makes problems (1) and (2) intractable almost immediately.

As a side-result, we prove that the membership problem for regular expressions with numerical occurrence indicators and negation is in polynomial time.

Categories and Subject Descriptors: ??? [**???**]

General Terms: Theory

Additional Key Words and Phrases: Graph data, query evaluation, regular expression

ACM Reference Format:
Katja Losemann and Wim Martens. 2013. The Complexity of Regular Expressions and Property Paths in SPARQL. ACM Trans. Datab. Syst. V, N, Article A (January YYYY), 39 pages.
DOI = 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

1. INTRODUCTION

The Resource Description Framework (RDF) is a data model developed by the World Wide Web Consortium (W3C) to represent linked data on the Web. The underlying idea is to improve the way in which data on the Web is readable by computers and to enable new ways of querying Web data. In its core, RDF represents linked data as an edge-labeled graph. The de facto language developed by the W3C for querying RDF data is the SPARQL Protocol and RDF Query Language (SPARQL).

Recently, the W3C decided to boost SPARQL 1.1 with extensive navigational capabilities through the introduction of *property paths* [Harris and Seaborne 2010]. Property paths closely correspond to regular expressions and are a crucial tool in SPARQL if one

Author's addresses: K. Losemann, W. Martens, Institut für Informatik, University of Bayreuth, 95440 Bayreuth, Germany. This article is the full version of [Losemann and Martens 2012], which was presented at the 31st ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS) in Scottsdale, Arizona, USA, 2012. This work is supported by grant number MA 4938/2–1 from the Deutsche Forschungsgemeinschaft (Emmy Noether Nachwuchsgruppe).

wants to perform non-trivial navigation through RDF data. In the January 2012 working draft of SPARQL 1.1 [Harris and Seaborne 2012], property paths are not defined as standard regular expressions, but some syntactic sugar is added. Notably, property paths can use numerical occurrence indicators (making them exponentially more succinct than standard regular expressions) and a limited form of negation. Furthermore, their semantics is different from usual definitions of regular expressions on graphs. In particular, when evaluating a regular expression, the W3C semantics requires some subexpressions to be matched onto *simple walks*,[1] whereas other subexpressions can be matched onto arbitrary paths.

Property paths are very fundamental in SPARQL. For example, the SPARQL query of the form

```
SELECT ?x,?y WHERE {?x r ?y}
```

asks for pairs of nodes (x, y) such that there is a path from x to y that matches the property path `r`. In fact, according to the SPARQL definition, the output of such a query is a multiset in which each pair of nodes (x, y) of the graph occurs as often as the number of paths from x to y that match r under W3C semantics. By only allowing certain subexpressions to match simple walks, the W3C therefore ensures that the number of paths that match a property path is always finite.

The amount of available RDF data on the Web has grown steadily over the last decade [Arenas and Pérez 2011]. Since it is highly likely to become more and more important in the future, we are convinced that investigating foundational aspects of evaluating regular expressions and property paths over graphs is a very relevant research topic. We therefore make the following contributions.

We investigate the complexity of two problems which we believe to be central for query processing on graph data. In the EVALUATION problem, one is given a graph, two nodes x and y, and a regular expression r, and one is asked whether there exists a path from x to y that matches r. In the COUNTING problem, one is asked *how many* paths from x to y match r. Notice that, according to the W3C definition, the answer to the above SELECT query needs to contain the answer to the COUNTING problem in unary notation.

Our theoretical investigation is motivated by an experimental analysis on several popular SPARQL processors that reveals that they deal with property paths very inefficiently. Already for solving the EVALUATION problem, all systems we found require time double exponential in the size of the queries in the worst case. We show that it is, in principle, possible to solve EVALUATION much more quickly: For a graph G and a regular expression r with numerical occurrence indicators, we can test whether there is a path from x to y that matches r in polynomial time combined complexity. More precisely, we present a rather simple procedure that already gives linear time in the size of r and a low degree polynomial in the size of the graph in the worst case.

We then investigate deeper reasons why evaluation of property paths is so inefficient in practice. In particular, we perform an in-depth study on the influence of some W3C design decisions on the computational complexity of property path evaluation. Our study reveals that the high processing times can be partly attributed already to the SPARQL 1.1 definition from the W3C. We formally define two kinds of semantics for property paths: *regular path semantics* and *simple walk semantics*. Here, *simple walk semantics* is our formalization of the W3C's semantics for property paths. Under *regular path semantics*, a path (possibly containing loops) in an edge-labeled graph matches a regular expression if the concatenation of the labels on the edges is in the language defined by the expression.

[1]A simple walk is a path that does not visit the same node twice, but is allowed to return to its first node.

We prove that, under regular path semantics, EVALUATION remains tractable under combined query evaluation complexity, even when numerical occurrence indicators are added to regular expressions. Our algorithm is cubic in the size of the graph (which means that it is not efficient enough to scale on real-life instances) but it is very naive and we did not try to optimize it in terms of performance. In contrast, under simple walk semantics, EVALUATION is already NP-complete for the regular expression $(aa)^*$ [Mendelzon and Wood 1995]. (So, it is NP-complete even under data complexity.) We also identify a fragment of expressions for which EVALUATION under simple walk semantics is in P but we prove that EVALUATION under simple walk semantics for this fragment is the same problem than EVALUATION under regular path semantics.

The picture becomes perhaps even more striking for the COUNTING problem. Under regular path semantics, we provide a detailed chart of the tractability frontier. When the expressions are *unambiguous*, then COUNTING can be solved in polynomial time. Intuitively, an expression is unambiguous if it can only match a word in one possible way. Therefore, *deterministic* expressions (as used in *Document Type Definitions* [Bray et al. 2008]) would be unambiguous, for example. However, even for expressions with a very limited amount of non-determinism beyond unambiguity, COUNTING becomes #P-complete. This picture changes rather drastically under simple walk semantics. Here, COUNTING is already #P-complete for the regular expression a^*. Essentially, this shows that, as soon as the Kleene star operator is used, COUNTING is #P-complete under simple walk semantics. All fragments we found for which COUNTING is tractable under simple walk semantics are tractable because, for these fragments, simple walk semantics equals regular path semantics.

Our complexity results are summarized in Table I. One result that is not in the table but may be of independent interest is the word membership problem for regular expressions with numerical occurrence indicators and negation. We prove this problem to be in P in Theorem 3.7. Another result that is not in the table is that the complexity of EVALUATION under regular path semantics remains the same if we extend our expressions with a *nesting* operator, thereby obtaining a variant of *nested regular expressions* (see, e.g. [Pérez et al. 2010]). This result is presented in Section 6.

Since the W3C's specification for SPARQL 1.1 is still under development, this article is intended to send a strong message to the W3C that informs it of the computational complexity repercussions of some design decisions; and what could be possible if the semantics of property paths were to be changed. Based on our observations, a semantics for property paths that is based on regular path semantics seems to be recommendable from a computational complexity point of view.

Related Work and Further Literature. This article studies evaluation problems of regular expressions over graphs. Regular expressions as a language for querying graphs have been studied in the database literature for more than a decade, sometimes under the name of regular path queries or general path queries [Abiteboul et al. 1997; Buneman et al. 1996; Consens and Mendelzon 1990; Cruz et al. 1987; Fernández et al. 2000; Yannakakis 1990]. Various problems for regular path queries have been investigated in the database community, such as optimization [Abiteboul and Vianu 1999], query rewriting and query answering using views [Calvanese et al. 2002; 2000b], and containment [Calvanese et al. 2000a; Deutsch and Tannen 2001; Florescu et al. 1998]. Recently, there has been a renewed interest in path queries on graphs, for example, on expressions with data value comparisons [Libkin and Vrgoč 2012].

Regular path queries have also been studied in the context of program analysis [Liu and Yu 2002]. However, the setting from [Liu and Yu 2002] is different from ours in the sense that they are interested in a universal semantics of the queries. That is, they are searching for pairs of nodes in the graph such that *all* paths between them match the given expression.

On a technical level, the most closely related work is on regular expressions with numerical occurrence indicators and on the complexity of SPARQL. Regular expressions with numerical occurrence indicators have been investigated in the context of XML schema languages [Colazzo et al. 2009b; 2009a; Gelade et al. 2012; Gelade et al. 2009; Kilpeläinen and Tuhkanen 2007; 2003] since they are a part of the W3C XML Schema Language [Gao et al. 2009]. One of our polynomial time upper bounds builds directly on Kilpeläinen and Tuhkonen's algorithm for membership testing of a regular expression with numerical occurrence indicators [Kilpeläinen and Tuhkanen 2003].

To the best of our knowledge, this article is the first one that studies the complexity of full property paths (i.e., regular expressions with numerical occurrence indicators) in SPARQL. Property paths without numerical occurrence indicators have been studied in [Pérez et al. 2010; Alkhateeb et al. 2009; Arenas et al. 2012]. Most closely related to us is [Arenas et al. 2012], which is conducted independently from us and which complements our work in several respects. The semantics of property paths in [Harris and Seaborne 2012] was not entirely clear and could be interpreted in different ways. The present article studies one such interpretation (based on the definition of ZeroOrMorePath/OneOrMorePath in Section 18.4 in [Harris and Seaborne 2012]) and Arenas et al. study an alternative one (based on Section 18.5 in [Harris and Seaborne 2012] that gives a procedure for counting paths). The joint main message of [Arenas et al. 2012] and this article is that both interpretations quickly lead to intractability of query evaluation and that something should be changed. In particular, we show that, even if one would abandon the precise counting procedure in Section 18.5 of [Harris and Seaborne 2012] and fall back to the definition in Section 18.4, evaluation would remain intractable very quickly. We should note that, in the meantime, this ambiguity in the definitions has been resolved in the SPARQL 1.1 Recommendation. We discuss these matters in more detail in Section 7. Further work on the complexity of SPARQL query evaluation can be found in [Pérez et al. 2009; Schmidt et al. 2010]. We refer to [Arenas and Pérez 2011] for further references on research on RDF databases and query languages.

2. PRELIMINARIES

For the remainder of this article, Δ always denotes a countably infinite set. We use Δ to model the set of IRIs and prefixed names from the SPARQL specification. We assume that we can test for equality between elements of Δ in constant time.

A Δ-*symbol* (or simply *symbol*) is an element of Δ, and a Δ-*word* (or simply word) is a finite sequence $w = a_1 \cdots a_n$ of Δ-symbols. We define the length of w, denoted by $|w|$, to be n. We denote the empty word by ε. The set of *positions of* w is $\{1, \ldots, n\}$ and the *symbol of* w *at position* i is a_i. By $w_1 \cdot w_2$ or w_1w_2 we denote the *concatenation* of two words w_1 and w_2. The set of all words is denoted by Δ^*. A *word language* is a subset of Δ^*. For two word languages $L, L' \subseteq \Delta^*$, we define their concatenation $L \cdot L'$ to be the set $\{ww' \mid w \in L, w' \in L'\}$. We abbreviate $L \cdot L \cdots L$ (i times) by L^i. The set of *regular expressions* over Δ, denoted by RE, is defined as follows: $\emptyset$, ε and every Δ-symbol is a regular expression; and when r and s are regular expressions, then $(r \cdot s)$, $(r + s)$, $(r?)$, (r^*), and (r^+) are also regular expressions. We assume w.l.o.g. that $\emptyset$ is never used as a subexpression of another regular expression. We consider the following additional operators for regular expressions:

Numerical Occurrence Indicators: If $k \in \mathbb{N}$ and $\ell \in \mathbb{N} \cup \infty$ with $k \leq \ell$, then $(r^{k,\ell})$ is a regular expression.

Negation: If r is a regular expression, then so is $(\neg r)$.

Negated label test: If $\{a_1, \ldots, a_n\}$ is a non-empty, finite subset of Δ, then $!(a_1 + \cdots + a_n)$ is a regular expression.[2]

Wildcard: The symbol $\bullet$ is a regular expression. (We assume that $\bullet \notin \Delta$.)

We often omit braces in regular expressions to improve readability. Furthermore, we often abbreviate $r^{k,k}$ by r^k. By $\text{RE}(\mathcal{X})$ we denote the set of regular expressions with additional features $\mathcal{X} \subseteq \{\#, \neg, \bullet, !\}$ where "#" stands for numerical occurrence indicators, "$\neg$" for negation, "!" for the negated label test, and "$\bullet$" for the single-symbol wildcard. For example, RE(#) denotes the set of regular expressions with numerical occurrence indicators and RE($\#, \neg, \bullet$) is the set of regular expressions with numerical occurrence indicators, negation, and wildcard. We are particularly interested in the following class of expressions, which is inspired by the definition of property paths in the SPARQL 1.1 working draft of January 2012 [Harris and Seaborne 2012].

Definition 2.1. The set of *SPARQL Regular Expressions* or *SPARQL Property Paths* is the set RE($\#, !, \bullet$).[3]

We consider edge-labeled graphs as our data model. A graph G will be denoted as $G = (V, E)$, where V is the set of nodes of G and $E \subseteq V \times \Delta \times V$ is the set of edges. An edge e is therefore of the form (u, a, v) if it goes from node u to node v and bears the label a. When we want to refer to an edge but do not care about its label, we sometimes also write an edge as a pair (u, v) in order to simplify notation. We assume familiarity with basic terminology on graphs. A *path* from node x to node y in G is a sequence $p = v_0[a_1]v_1[a_2]v_2 \cdots v_{n-1}[a_n]v_n$ such that $v_0 = x$, $v_n = y$, and (v_{i-1}, a_i, v_i) is an edge for each $i = 1, \ldots, n$. When we are not interested in the labels on the edges, we sometimes also write $p = v_0 v_1 \ldots v_n$. We say that path p has *length* n. Notice that a path of length zero does not follow any edges. The *labeled word* induced by the path p in G is $a_1 \cdots a_n$ and is denoted by $\text{lab}^G(p)$. If G is clear from the context, we sometimes also simply write lab(p). We define the *concatenation* of paths $p_1 = v_0[a_1]v_1 \cdots v_{n-1}[a_n]v_n$ and $p_2 = v_n[a_{n+1}]v_{n_1} \cdots v_{n+m-1}[a_{n+m}]v_{n+m}$ to be the path $p_1 p_2 := v_0[a_1]v_1 \cdots v_{n-1}[a_n]v_n[a_{n+1}]v_{n_1} \cdots v_{n+m-1}[a_{n+m}]v_{n+m}$.

For two binary relations $R_1 \subseteq V \times V$ and $R_2 \subseteq V \times V$, we denote by $R_1 \bowtie R_2$ the set $\{(u, v) \mid \exists z \in V : (u, z) \in R_1 \wedge (z, v) \in R_2\}$. Likewise, for a binary relation $R \subseteq V \times V$ and $k \in \mathbb{N} - \{0\}$, we define $R^k := R$ if $k = 1$ and $R^k := R \bowtie R^{k-1}$ otherwise.

2.1. Regular Path Semantics

The language defined by an expression r, denoted by $L(r)$, is inductively defined as follows: $L(\emptyset) = \emptyset$; $L(\varepsilon) = \{\varepsilon\}$; $L(a) = \{a\}$ for every $a \in \Delta$; $L(!(a_1 + \cdots + a_n)) = \Delta - \{a_1, \ldots, a_n\}$; $L(\bullet) = \Delta$; $L(rs) = L(r) \cdot L(s)$; $L(r+s) = L(r) \cup L(s)$; $L(r^*) = \{\varepsilon\} \cup \bigcup_{i=1}^{\infty} L(r)^i$, $L(r^{k,\ell}) = \bigcup_{i=k}^{\ell} L(r)^i$; and, $L(\neg r) = \Delta^* - L(r)$. Furthermore, $L(r?) = \varepsilon + L(r)$ and $L(r^+) = L(r)L(r^*)$.[4] The *size* of a regular expression r over Δ, denoted by $|r|$, is the number of occurrences of Δ-symbols, $\bullet$-symbols, and operators occurring in r, plus the sizes of the binary representations of the numerical occurrence indicators. We say that a path p *matches a regular expression* r *under regular path semantics* if $\text{lab}(p) \in L(r)$.

[2]Throughout the article, we will consider $!(a_1 + \cdots + a_n)$ to be an atomic expression. This means that it will often be a base case in inductive proofs.
[3]In this article, we mostly refer to these expressions as "SPARQL regular expressions" to avoid confusion between expressions and paths.
[4]We do not define r^+ as an abbreviation of the expression rr^* since r^+ and rr^* have different semantics in [Harris and Seaborne 2012].

2.2. Simple Walk Semantics

The simple walk semantics is our formalization of the semantics of property paths in the SPARQL working draft of January 2012 [Harris and Seaborne 2012]. In the *SPARQL algebra* (Section 18.4 of [Harris and Seaborne 2012]), the semantics of subexpressions of the form r^* and r^+ is defined through the operators *ZeroOrMorePath* and *OneOrMorePath* respectively. In the definition, it is assumed that the semantics of the subexpression[5] r is already known and reads as follows:

— "ZeroOrMorePath: An arbitrary length path $P = (X\ r^*\ Y)$ is all solutions from X to Y by repeated use of r such that any nodes in the graph are traversed once only. ZeroOrMorePath includes X."
— "OneOrMorePath: An arbitrary length path $P = (X\ r^+\ Y)$ is all solutions from X to Y by repeated use of r such that any nodes in the graph are traversed once only. This does not include X, unless repeated evaluation of the path from X returns to X."

(Here, X and Y are variables that bind to nodes.) These definitions seem to be a bit ambiguous. In the definition of ZeroOrMorePath, the specification seems to require that the path from X to Y is a path that contains each node at most once. We formalize this as a *simple path*.

Definition 2.2. A *simple path* is a path $v_0v_1 \cdots v_{n-1}v_n$, where each node v_i occurs exactly once.

However, in the definition of OneOrMorePath, it seems to be allowed for a path to return to the first node. We formalize these paths as *simple walks*.

Definition 2.3. A *simple cycle* is a path $v_0v_1 \cdots v_{n-1}v_n$ such that $v_0 = v_n$ and every v_i for $i = 1, \ldots, n-1$ occurs exactly once. A *simple walk* is either a simple path or a simple cycle.

We find the informal definition of the W3C to be unclear on the matter of whether it allows simple cycles, but examples in the working draft suggest that simple cycles are allowed. We therefore choose to consider simple walks in the presentation of our proofs and consider the following constraint on the semantics of regular expressions:

Simple Walk Requirement. Subexpressions of the form r^* and r^+ should be matched to simple walks in graphs.

However, our complexity results concerning simple walks also hold if we would consider simple paths instead of simple walks.

We now formally define our abstraction of the semantics of property paths as defined by the W3C in the January 2012 working draft [Harris and Seaborne 2012]. Let $p = v_0[a_1]v_1[a_2]v_2 \cdots v_{n-1}[a_n]v_n$ be a path in a graph and r be a SPARQL regular expression. Then p *matches* r *under simple walk semantics* if one of the following holds:

— If $r = \emptyset$, $r = \varepsilon$, $r = a$ for some $a \in \Delta$, $r = \bullet$, or $r = !(a_1 + \cdots + a_n)$ then $\text{lab}(p) \in L(r)$.
— If $r = s^*$ or $r = s^+$, then $\text{lab}(p) \in L(r)$ and p is a simple walk.
— If $r = s?$, then either $p = v_0$ or p matches s under simple walk semantics.
— If $r = s_1 \cdot s_2$, then there exist paths p_1 and p_2 such that $p = p_1p_2$ and p_i matches s_i under simple walk semantics for all $i = 1, 2$.
— If $r = s_1 + s_2$, then there exists an $i = 1, 2$ such that p matches s_i under simple walk semantics.

[5]Subexpression r is called "path" in [Harris and Seaborne 2012].

— If $r = s^{k,\ell}$ with $\ell \neq \infty$, then there exist paths $p_1, \ldots, p_m$ with $k \leq m \leq \ell$ such that $p = p_1 \cdots p_m$ and p_i matches s under simple walk semantics for each $i = 1, \ldots, m$. (Notice that, if $\ell = 0$ then p has length zero.)
— If $r = s^{k,\infty}$, then there exist paths p_1 and p_2 such that $p = p_1 p_2$, p_1 matches $s^{k,k}$ under simple walk semantics, and p_2 matches s^* under simple walk semantics.

We added the case $r = \emptyset$ for compatibility with regular expressions. Notice that there does not exist a path that matches the expression $\emptyset$ under simple walk semantics. Furthermore, under simple walk semantics, we no longer have that a^* is equivalent to a^*a^*, that $a^{1,\infty}$ is equivalent to a^+, or that aa^* is equivalent to a^+. However, aa^* is equivalent to $a^{1,\infty}$. For the expression $(a + b)^{50,60}$, regular path semantics and simple walk semantics coincide.

2.3. Problems of Interest

We will often consider a graph $G = (V, E)$ together with a *source node* x and a *target node* y, for example, when considering paths from x to y. We say that (V, E, x, y) is the *s-t graph of* G *w.r.t.* x *and* y. Sometimes we leave the facts that x and y are source and target implicit and just refer to (V, E, x, y) as a graph.

We consider two paths $p_1 = v_0^1[a_1^1]v_1^1 \cdots [a_n^1]v_n^1$ and $p_2 = v_0^2[a_1^2]v_1^2 \cdots [a_m^2]v_m^2$ in a graph to be *different*, when either the sequences of nodes or the sequences of labels are different, i.e., $v_0^1 v_1^1 \cdots v_n^1 \neq v_0^2 v_1^2 \cdots v_m^2$ or $\text{lab}(p_1) \neq \text{lab}(p_2)$. Notice that this implies that we consider two paths going through the same sequence of nodes but using different edge labels to be different.

We are mainly interested in the following problems, which we consider under regular path semantics and under simple walk semantics:

EVALUATION: Given a graph (V, E, x, y) and a regular expression r, is there a path from x to y that matches r?

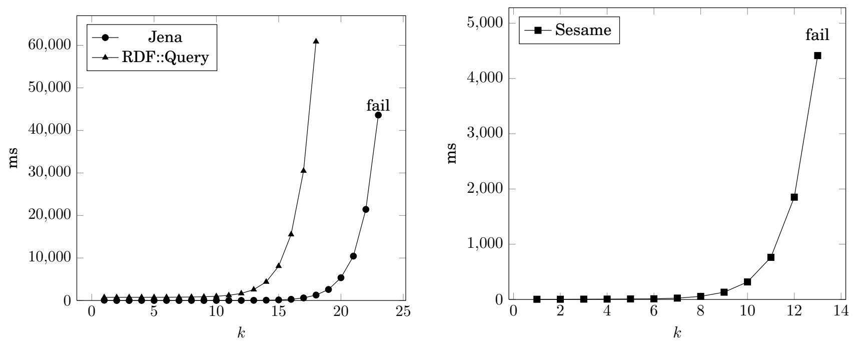
FINITENESS: Given a graph (V, E, x, y) and a regular expression r, are there only finitely many different paths from x to y that match r?

COUNTING: Given a graph (V, E, x, y), a regular expression r and a natural number max in unary, how many different paths of length at most max between x and y match r?

Throughout the article, we will sometimes talk about the query complexity or data complexity of the above problems. When we talk about *query complexity*, we assume the graph (V, E, x, y) to be fixed. Therefore, the only input for the above problems is the regular expression r and, if relevant, the number max in unary. Under *data complexity*, we consider the expression r and number max (if relevant) to be fixed. Therefore, the input of the above problems under data complexity only consists of the graph (V, E, x, y).

The COUNTING problem is closely related to two problems studied in the literature: (1) counting the number of words of a given length in the language of a regular expression and (2) counting the number of paths in a graph that match certain constraints. We chose to have the number max in unary because this was also the case in several highly relevant work on (1) and (2) (e.g., [Kannan et al. 1995; Álvarez and Jenner 1993; Valiant 1979]). Furthermore, it strengthens our hardness results to consider the number max to be encoded in unary. We note that our polynomial-time results for COUNTING still hold when the number max is given in binary (Theorems 4.3 and 3.15).

We will often parameterize the problems with the kind of regular expressions or automata we consider. For example, when we talk about EVALUATION for RE($\#, \neg$), then we mean the EVALUATION problem where the input is a graph (V, E, x, y) and an expression r in RE($\#, \neg$).



(a) Evaluation time for Jena and RDF::Query.

(b) Evaluation time for Sesame.

Fig. 1. Time taken by Jena, Sesame and RDF::Query for evaluating the expression $(a + b)^{1,k}$ for increasing values of k on a graph with two nodes and four edges.

Finally, we note that we use Cook reductions, also known as Turing reductions, in this article. Under these reductions, counting the number of satisfying assignments of a DNF formula and counting the number of simple source-to-target paths in a graph are #P-complete problems.

3. THE EVALUATION PROBLEM

We conducted a practical study on the efficiency in which SPARQL engines evaluate property paths. We evaluated the most prevalent SPARQL query engines which support property paths, namely the Jena Semantic Web Framework, Sesame, RDF::Query, and Corese 3.0[6]. Our experiments were performed in November 2011. We asked the four frameworks to answer the query

```
ASK WHERE { :x (a|b){1,k} :y }
```

for increasing values of k on the graph

$$\begin{tikzcd} x \arrow[r, bend left=50, "b"] \arrow[r, bend left=10, "a"] & y \arrow[l, bend left=10, "a"] \arrow[l, bend left=50, "b"] \end{tikzcd}$$

consisting of two nodes and four labeled edges. An ASK-query in SPARQL returns a Boolean value, which is true if and only if there exists an answer for the query in the graph. (More formally, if the query would return at least one tuple on the graph.) In our formal framework, answering this query therefore corresponds to solving the EVALUATION problem on the above graph for the expression $(a + b)^{1,k}$. Notice that the answer is always "true". Furthermore, notice that this query has the same semantics under regular path semantics as under simple walk semantics.

The performance of three of the four systems is depicted in Figure 1. The results are obtained from evaluation on a desktop PC with 2 GB of RAM. For the Jena and Sesame framework the points in the graph depict all the points we could obtain data on. When

[6]These engines can be found at http://jena.apache.org/ (Jena Semantic Web Framework), http://www.openrdf.org/ (Sesame), http://code.google.com/p/rdfquery/ (RDF::Query), and http://wimmics.inria.fr/corese (Corese 3.0).

we increased the number k by one more as shown on the graphic, the systems ran out of memory. Our conclusion from our measurements is that all three systems seem to exhibit a double exponential behavior: from a certain point, whenever we increase the number k by one (which does not mean that one more bit is needed to represent it), the processing time doubles. Corese 3.0 evaluated queries of the above form very quickly. However, when we asked the query

```
ASK WHERE {x ((a|b)/(a|b)){1,k} y},
```

which asks for the existence of even length paths, its time consumption was the same than the other three systems. In contrast to the other three systems, Corese did not run out of memory so quickly.

In a related study by Arenas, Conca, and Pérez [Arenas et al. 2012] a double exponential behavior is observed for `SELECT` queries that use property paths. `SELECT` queries are more difficult to answer than `ASK` queries because `SELECT` queries should output all tuples that witness the query, whereas an `ASK` query simply asks if there exists such a tuple or not. However, the queries for which [Arenas et al. 2012] observed double exponential behavior did not exploit numerical occurrence indicators as we did. As such, the experiments here and in [Arenas et al. 2012] seem to complement one another.

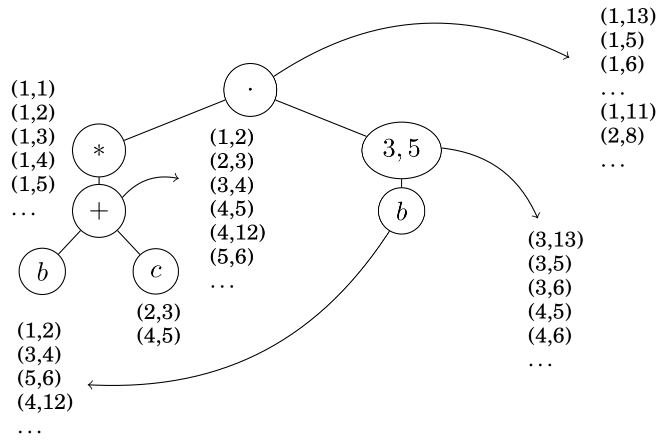
3.1. An Efficient Algorithm for Regular Path Semantics

We show that the double exponential behavior we observed in practice can be improved to polynomial-time combined complexity. More precisely, we present a polynomial-time algorithm for EVALUATION of SPARQL regular expressions under regular path semantics. Later, in Section 3.3, we prove that EVALUATION under simple walk semantics is NP-complete for very simple regular expressions.

We briefly discuss some basic results on evaluating regular expressions on graphs. EVALUATION is in P for standard regular expressions.[7] In this case, the problem basically boils down to testing intersection emptiness of two finite automata: one converts the graph G with the given nodes x and y into a finite automaton A_G by taking the nodes of G as states, the edges as transitions, x as its initial state and y as its accepting state. The expression r is converted into a finite automaton A_r by using standard methods. Then, there is a path from x to y in G that matches r if and only if the intersection of the languages of A_G and A_r is not empty, which can easily be tested in polynomial time. It is known that the product construction of automata can even be used for a linear-time algorithm for evaluating *nested* regular path expressions, which are regular expressions that have the power to branch out in the graph [Alechina and Immerman 2000; Pérez et al. 2010]. In fact, since such nested regular expressions are a fragment of propositional dynamic logic (PDL), linear time evaluation of such expressions already follows from linear time evaluation of PDL [Cleaveland and Steffen 1993; Alechina and Immerman 2000].

Our polynomial time algorithm for EVALUATION of RE$(\#, !, \bullet)$-expressions follows a dynamic programming approach. We first discuss the main idea of the algorithm and then discuss its complexity. Let r be a SPARQL regular expression, that is, a RE$(\#, !, \bullet)$-expression, and let $G = (V, E)$ be a graph. The algorithm traverses the syntax tree of r in a bottom-up fashion. To simplify notation in the following discussion, we identify nodes from the parse tree of r with their corresponding subexpressions. We store, for each node in the syntax tree with associated subexpression s, a binary relation eval$(s) \subseteq V \times V$ such that

[7]This has already been observed in the literature several times, e.g., as Lemma 1 in [Mendelzon and Wood 1995], on p.7 in [Abiteboul and Vianu 1999], and in [Alkhateeb et al. 2009].



(a) Part of a run on the expression $(b+c)^*b^{3,5}$ and the graph in Fig. 2(b).

$$\begin{tikzcd}
1 \arrow[r, "b"] & 2 \arrow[r, "c"] & 3 \arrow[r, "b"] & 4 \arrow[r, "c"] \arrow[dr, "b"'] & 5 \arrow[r, "b"] & 6 \arrow[r, "b"] & 7 \arrow[r, "b"] & 8 \arrow[r, "b"] & 9 \arrow[r, "b"] & 10 \arrow[r, "b"] & 11 \\
 & & & & 12 \arrow[r, "b"'] & 13 \arrow[ul, "b"'] & & & & &
\end{tikzcd}$$

(b) An edge-labeled graph.

Fig. 2. Illustration of the polynomial-time dynamic programming algorithm.

$$(u, v) \in \text{eval}(s) \text{ if and only if there exists a path from } u \text{ to } v \text{ in } G \text{ that matches } s.$$

(Of course, relations for identical subexpressions should not be computed twice.) The manner in which we join relations while going bottom-up in the parse tree depends on the type of the node. We discuss all possible cases next. The sets eval(s) can be computed by structural induction on r as follows:

— eval$(\emptyset) := \emptyset$;
— eval$(\varepsilon) := \{(u,u) \mid u \in V\}$;
— eval$(\bullet) := \{(u,v) \mid \exists a \in \Delta$ with $(u,a,v) \in E\}$;
— for every $a \in \Delta$, eval$(a) := \{(u,v) \mid (u,a,v) \in E\}$;
— eval$(!(a_1 + \cdots + a_n)) := \{(u,v) \mid \exists a \in \Delta - \{a_1,\ldots,a_n\}$ with $(u,a,v) \in E\}$;
— eval$(s_1 + s_2) :=$ eval$(s_1) \cup$ eval(s_2);
— eval$(s_1 \cdot s_2) :=$ eval$(s_1) \bowtie$ eval(s_2);
— eval$(s?) :=$ eval$(s) \cup$ eval(ε);
— eval(s^*) is the reflexive and transitive closure of eval(s);
— eval(s^+) is the transitive closure of eval(s);
— eval$(s^{k,\infty}) :=$ eval$(s^k) \bowtie$ eval(s^*); and
— for $\ell \neq \infty$, eval$(s^{k,\ell}) :=$ eval$(s)^k \bowtie$ eval$(s?)^\ell$.

Finally, if the input for EVALUATION is G, nodes x and y, and RE$(\#, !, \bullet)$-expression r, we return the answer "true" if and only if eval(r) contains the pair (x, y).

Example 3.1. Figure 2 illustrates part of a run of the evaluation algorithm on the graph in Figure 2(b) and the regular expression $r = (b+c)^*b^{3,5}$. Each node of the parse tree of the expression (Fig. 2(a)) is annotated with the binary relation that we compute for it. Finally, the relation for the root node contains all pairs (x, y) such that there is a path from x to y that matches r.

To prove that our algorithm runs in polynomial time, we use the following well-known results.

LEMMA 3.2 (SEE, E.G., [BERGE 1973], PAGE 74). *Let R be a binary relation and $k \in \mathbb{N}$. Then we can compute R^k by performing $O(\log k)$ join operations.*

PROOF (SKETCH). Immediate from the observation that, for every $k \in \mathbb{N}$,

$$R^k = \begin{cases} R, & \text{if } k = 1 \\ R \bowtie \left(R^{\frac{k-1}{2}} \bowtie R^{\frac{k-1}{2}}\right), & \text{if } k \text{ is odd} \\ \left(R^{\frac{k}{2}} \bowtie R^{\frac{k}{2}}\right), & \text{if } k \text{ is even} \end{cases}$$

□

The next lemma states the cost of a single join.

LEMMA 3.3. *For some $n \in \mathbb{N}$, let $R, S \subseteq \{1, \ldots, n\}^2$ be binary relations. Then we can compute $R \bowtie S$ in time $O(n^3)$.*

PROOF. When one represents R and S as boolean $n \times n$ matrices M_R and M_S (i.e., the connectivity matrices of R and S), the matrix representation for $R \bowtie S$ can be obtained by multiplying M_R with M_S, costing time $O(n^3)$. □

We are now ready to show that EVALUATION is correct and can be implemented to run in polynomial time.

THEOREM 3.4. EVALUATION *for SPARQL regular expressions under regular path semantics is in time $O(|r| \cdot |V|^3)$, where $|r|$ is the size of the expression and $|V|$ is the number of nodes in the graph.*

PROOF. We prove that the algorithm that computes the sets eval(s) in the beginning of Section 3.1 can be implemented to decide EVALUATION for RE$(\#, !, \bullet)$ in polynomial time. That is, given a graph $G = (V, E, x, y)$ and RE$(\#, !, \bullet)$-expression r, it decides in polynomial time whether there is a path in G from x to y that matches r (under regular path semantics).

The correctness proof is a straightforward induction on the structure of the expression r. More precisely, the following invariant (I) holds for every relation eval(s) that is calculated:

For each subexpression s of r, we have

$$(u, v) \in \text{eval}(s) \Leftrightarrow \exists \text{ path } p \text{ in } G \text{ from } u \text{ to } v \text{ such that lab}(p) \in L(s). \qquad \text{(I)}$$

Notice that the correctness of the invariant implies the correctness of the algorithm.

We now prove that the invariant is correct. The base cases, i.e., the cases where $s = \emptyset$, $s = \varepsilon$, $s = a$ for some $a \in \Delta$, $s = \bullet$, or $s = !(a_1 + \cdots + a_n)$, are immediate.

We now prove the induction cases. To this end, let s be a node in the syntax tree of r such that (I) is already known to be true for all of its children. We make the case distinction based on the type of subexpression that s represents.

If $s = s_1 + s_2$ then we compute eval(s) = eval$(s_1) \cup$ eval(s_2). Take an arbitrary $(u, v) \in V \times V$. First we show that, if (u, v) in eval(s), then there exists a path p from u to v in G such that lab$(p) \in L(s)$. By construction of eval(s), we know that $(u, v) \in$ eval(s_1) or $(u, v) \in$ eval(s_2). By induction, we know that eval(s_1) and eval(s_2) are calculated correctly. Therefore, there exists a path p from u to v that matches s_1 or s_2. By definition of s, this implies that p also matches s. Second we show that, if there exists a path p from u to v in G with lab$(p) \in L(s)$ then (u, v) in eval(s). To this end, let p be a path from u to v in G with lab$(p) \in L(s)$. By definition of s, we have that p matches s_1 or s_2. Since s_1 and s_2 are calculated correctly by induction, we have that (u, v) is in eval(s_1) or in eval(s_2). Since eval(s) = eval$(s_1) \cup$ eval(s_2), it follows that $(u, v) \in$ eval(s) and eval(s) fulfills (I).

If $s = s_1 \cdot s_2$, then we compute eval(s) := eval$(s_1) \bowtie$ eval(s_2), where eval$(s_1) \bowtie$ eval$(s_2) := \{(u, v) \mid \exists z \in V : (u, z) \in$ eval$(s_1) \wedge (z, v) \in$ eval$(s_2)\}$ is the composition of the relation eval(s_1) and eval(s_2). Take an arbitrary $(u, v) \in V \times V$. First we show that, if (u, v) in eval(s), then there exists a path p from u to v in G such that lab$(p) \in L(s)$. By the definition of the join operator, we know that for every tuple $(u, v) \in$ eval(s) there exist tuples (u, z) and (z, v), such that $(u, z) \in$ eval(s_1) and $(z, v) \in$ eval(s_2). Because eval(s_1) and eval(s_2) are calculated correctly by induction, there exists a path p_1 from u to z in G with lab$(p_1) \in L(s_1)$ and a path p_2 from z to v with lab$(p_2) \in L(s_2)$. Thus the concatenation of these two paths $p = p_1.p_2$, is a path from u to v in G with lab$(p) \in L(s_1 \cdot s_2)$. Second, we show that, if there exists a path p from u to v in G with lab$(p) \in L(s_1 \cdot s_2)$, then (u, v) in eval(s). Therefore, let p be such a path. Since lab$(p) \in L(s_1 \cdot s_2)$, we know that there exist paths p_1 and p_2, such that $p = p_1.p_2$ and p_1 is a path from u to some node z with lab$(p_1) \in L(s_1)$ and p_2 is a path from z to v with lab$(p_2) \in L(s_2)$. Because eval(s_1) and eval(s_2) are calculated correctly by induction, we know that $(u, z) \in$ eval(s_1) and $(z, v) \in$ eval(s_2). Thus the tuple (u, v) is in eval(s) by the definition of the join and eval(s) fulfills (I). Note that if $L(s_1)$ or $L(s_2)$ contain ε, then the relations of s_1 or s_2 are reflexive.

The cases $s = s_1?$, $s = s_1^*$, $s = s_1^+$, and $s = s_1^{k,\ell}$ can be proved similarly to the previous cases. This concludes our proof of correctness.

Next, we argue that the algorithm can be implemented to run in polynomial time. Notice that the parse tree of the input expression s has linear size. Since the algorithm processes the parse tree in a bottom-up fashion we therefore only need to prove that we can implement each separate case in time $O(|V|^3)$ or in time $O(|V|^3 \log k)$ if there is a numerical occurrence operator k. Notice that each relation eval(s) has size $O(|V|^2)$.

The cases where $s \in \Delta$, $s = \varepsilon$, $s = \bullet$, $s =!(a_1 + \cdots + a_n)$, $s = s_1 \cdot s_2$, $s = s_1^*$, $s = s_1^+$, and $s = s_1?$ are either trivial or immediate from Lemma 3.3. The cases $s = s_1^k$ and $s = s_1^{k,\infty}$, can be computed in time $O(|V|^3 \log k)$ by applying Lemmas 3.2 and 3.3. Similarly, we obtain that the case $s = s_1^{k,\ell}$ is in time $O(|V|^3 \log \ell)$. This concludes our proof. □

Although Theorem 3.4 presents a polynomial-time result, the algorithm in the proof is not very practical. The biggest bottleneck is in Lemma 3.3 which joins two relations in time $O(n^3)$. If we represent the relations as matrices, this procedure has a best-case complexity of $\Omega(n^2)$, even if the relations to be joined are much smaller than n.

We therefore look at an alternative algorithm that uses sort-merge join instead of matrix multiplications. For analysing the complexity of joining two relations R and S, we need the following notion. For two finite binary relations R and S, the *number of tuples of $R \bowtie S$ under multiset semantics* is the number of elements in the set $\{(u, z, v) \mid (u, z) \in R$ and $(z, v) \in S\}$.

LEMMA 3.5. *Let R and S be finite binary relations. Let t_R, t_S and t_{RS} denote the number of tuples in R, S, and $R \bowtie S$, respectively. Let m denote the number of tuples of $R \bowtie S$ under multiset semantics. Then we can compute $R \bowtie S$ in time and space $O(t_R \log t_R + t_S \log t_S + m \log t_{RS})$.*

PROOF. The bound is achieved by a variant of the merge join algorithm. Sorting the tuples of R and S on their respective join attributes costs time $O(t_R \log t_R + t_S \log t_S)$.

Once the relations are sorted, we can join the relations similar to merge join (see, e.g., [Ramakrishnan and Gehrke 2003], page 458), but we need the output to be a set of pairs, rather than a multiset of pairs, so we have to eliminate duplicates. In order to do this, we can iterate through the sorted R and S as in the merge join algorithm and maintain the set of pairs we already discovered to be in $R \bowtie S$ in a self-balancing binary search tree (e.g., an AVL tree). When we find a new candidate pair for $R \bowtie S$ we can thus discover in time $O(\log t_{RS})$ if we already found it or not. Since there are m such candidate pairs, this last step costs time $O(t_R + t_S + m \log t_{RS})$. $\square$

In the worst case, the number m in Lemma 3.5 could be $\Theta(n^4)$ in terms of the number n of Lemma 3.3. However, it is expected that m is usually small in practice and that the literature emphasizes that the worst case is very unlikely [Ramakrishnan and Gehrke 2003].

In the following corollary, let $R_{\max}$ denote the maximal number of pairs in any relation eval(s) in the algorithm of Theorem 3.4 (including in the intermediate results for fast squaring in Lemma 3.2). Let m be at least $R_{\max}$ and an upper bound for the largest number of tuples in eval$(s_1) \bowtie$ eval(s_2) under multiset semantics, where eval$(s_1) \bowtie$ eval(s_2) ranges over all joins we perform in the algorithm (again, including the intermediate results for fast squaring). The following corollary is then obtained by taking the algorithm of Theorem 3.4 and replacing the join procedure of Lemma 3.3 with the procedure in Lemma 3.5.

COROLLARY 3.6. EVALUATION *for SPARQL regular expressions under regular path semantics is in time* $O(|r| \cdot m \log R_{max})$.

Since our evaluation algorithm is still rather naive, we feel that further improvements towards better data complexity are very likely to be possible.

Our evaluation algorithm is based on dynamic programming. We are not the first to think of dynamic programming in the context of regular expressions. The connection between dynamic programming and regular expressions goes back at least to Kleene's recursive formulas for extracting a regular expression from a DFA [Kleene 1956]. Dynamic programming for testing whether a word belongs to a language of a regular expression has been demonstrated in [Hopcroft and Ullman 1979] (p.75–76). Kilpeläinen and Tuhkonen adapted this approach for evaluating RE(#) on words [Kilpeläinen and Tuhkanen 2003]. However, the algorithm from Kilpeläinen and Tuhkonen does not naively extend to graphs: it would need time exponential in the expression. It uses the fact that the length of the *longest match* of the expression on the word cannot exceed the length of the word. For example, the regular expression a^{42} can only match a word if the word contains 42 a's. This assumption no longer holds in graphs.

We conclude this section with one more observation on the dynamic programming algorithm, which we will need for later proofs in the article. If we want to evaluate expressions on words instead of graphs, we can also incorporate negation into the algorithm. By MEMBERSHIP we denote the following decision problem: Given a word w and a regular expression r, is $w \in L(r)$?

THEOREM 3.7. MEMBERSHIP *for RE*$(\#, !, \neg, \bullet)$ *is in time* $O(|r| \cdot |w|^3)$, *where* $|r|$ *is the size of the expression and* $|w|$ *the length of the word.*

PROOF. To simplify the technical presentation in this proof, we abstract a word as an acyclic, connected, edge-labeled graph in which every node has at most one incoming or outgoing edge. As such, we can re-use the algorithm from Theorem 3.4. We already showed in the proof of Theorem 3.4 that EVALUATION (and therefore also MEMBERSHIP) for RE$(\#, !, \bullet)$ is in polynomial time by means of a dynamic programming algorithm. Here, we show how the algorithm can be extended to also take negation into account, if we evaluate over words instead of graphs. The change in our algorithm consists of considering one extra case:

— $\text{eval}(\neg s) := \{(u, v) \mid (u, v) \notin \text{eval}(s)\}$.

Clearly, this case can also be implemented to run in time $O(|r| \cdot |w|^3)$.

We prove that the algorithm is correct. To this end, let r be an expression from RE($\#, !, \neg, \bullet$). We prove the same invariant (I) than in the proof of Theorem 3.4, that is:

For each subexpression s of r, we have

$$(u, v) \in \text{eval}(s) \Leftrightarrow \exists \text{ path } p \text{ in } G \text{ from } u \text{ to } v \text{ such that } \text{lab}(p) \in L(s). \qquad \text{(I)}$$

Since we are considering only words now, a tuple (u, v) represents two positions in the word, such that $(u, v) \in \text{eval}(s)$ if and only if the subword from u to v of the considered word is in $L(s)$.

The induction base cases and all operators, except for $\neg$, are exactly the same as in the proof of Theorem 3.4. So, it only remains to consider the case $\text{eval}(\neg s) = \{(u, v) \mid (u, v) \notin \text{eval}(s)\}$. By definition of the negation operator and since we are dealing with a word, the (unique) path from u to v matches $\neg s$ if and only if it does not match s. Since, by induction, $\text{eval}(s)$ is calculated correctly, a tuple (u, v) is in $\text{eval}(s)$ if and only if the subword from u to v in the considered word matches $L(s)$. Then $(u, v) \in \text{eval}(s)$ if and only if $(u, v) \notin \text{eval}(\neg s)$. Therefore $\text{eval}(\neg s)$ fulfills (I). □

However, as we illustrate in the next section, allowing unrestricted negation in expressions does not allow for an efficient algorithm for EVALUATION on graphs.

3.2. Negation Makes Evaluation Hard over Graphs

The negated label test seems to be harmless for the efficiency of evaluating SPARQL regular expressions. On words, even the full-fledged negation operator "$\neg$" can be evaluated efficiently. However, allowing full-fledged negation for evaluation on graphs makes the complexity of EVALUATION non-elementary. The reason is that EVALUATION is at least as hard as satisfiability of the given regular expression.

LEMMA 3.8. *Let C be a class of regular expressions over a finite alphabet Σ. Then there exists a LOGSPACE (and therefore polynomial time) reduction from the non-emptiness problem for C-expressions to the* EVALUATION *problem with C-expressions.*

PROOF. The proof is immediate from the observation that non-emptiness of an expression r over an alphabet Σ is the same decision problem as EVALUATION for r and the graph $G = (V, E)$ with $V = \{x\}$ and $E = \{(x, a, x) \mid a \in \Sigma\}$. □

A *star-free generalized regular expression* is a regular expression that uses the concatenation ($\cdot$), disjunction ($+$), and negation ($\neg$) operators. The language emptiness problem of star-free generalized regular expressions takes such an expression r as input and asks whether $L(r) = \emptyset$. It is well known that this problem is non-elementary [Stockmeyer 1974] and we therefore also immediately have that EVALUATION is non-elementary for RE($\neg$)-expressions, by Lemma 3.8.

THEOREM 3.9. EVALUATION *for RE($\neg$) under regular path semantics is non-elementary.*

For completeness, since RE($\#, !, \neg, \bullet$)-expressions can be converted into RE($\#, !, \bullet$)-expressions with a non-elementary blow-up, we also mention a general upper bound for EVALUATION.

THEOREM 3.10. EVALUATION *for RE($\#, !, \neg, \bullet$) under regular path semantics is decidable.*

3.3. Complexity under Simple Walk Semantics

Section 3.1 showed how SPARQL regular expressions, which have numerical occurrence indicators, can be efficiently evaluated under regular path semantics with a rather simple algorithm. In this section, we will see how the complexity of EVALUATION changes when SPARQL's simple walk semantics rather than regular path semantics is applied.

3.3.1. NP-Complete Fragments. It follows from classical results that EVALUATION under simple walk semantics is NP-complete rather quickly. As Lapaugh and Papadimitriou showed in 1984, given a directed graph and two nodes x and y, it is NP-hard to decide whether there exists a simple path of even length from x to y. The NP upper bound is trivial.

THEOREM 3.11 (MENDELZON, WOOD 1995; LAPAUGH, PAPADIMITRIOU 1984). EVALUATION *under simple walk semantics is NP-complete for the expression* $(aa)^*$ *and for the expression* $(aa)^+$.

Notice that, since the expressions in Theorem 3.11 are fixed, the theorem already shows that the data complexity of EVALUATION under simple walk semantics is NP-hard.

On the other hand, EVALUATION remains in NP even when numerical occurrence indicators are allowed.

THEOREM 3.12. EVALUATION *for RE($\#, !, \bullet$)-expressions under simple walk semantics is NP-complete.*

PROOF. The NP lower bound is immediate from Theorem 3.11.

The NP upper bound follows from an adaptation of the algorithm of Section 3.1 where, in the cases for $s = s_1^*$ and $s = s_1^+$, simple walks are guessed between nodes to see if they belong to eval(s). Let $G = (V, E, x, y)$ be the s-t graph and let r be the regular expression. The NP algorithm guesses a (polynomial size representation of a) path from x to y and tests whether the path matches the expression under simple walk semantics in polynomial time.

For a subexpression s of r and nodes $u, v \in V$, let

$$\text{eval}_{\text{sw}}(s) = \{(u,v) \mid \text{there is a path from } u \text{ to } v \text{ that matches } s \text{ under simple walk semantics }\}$$

The NP algorithm computes eval_{sw} for all subexpressions s of r.

By eval we denote the function for evaluating expressions under regular path semantics, as defined in Section 3.1. The sets $\text{eval}_{\text{sw}}(s)$ can be computed by structural induction on r as follows:

— $\text{eval}_{\text{sw}}(\emptyset) := \emptyset$;
— $\text{eval}_{\text{sw}}(\varepsilon) := \{(u,u) \mid u \in V\}$;
— $\text{eval}_{\text{sw}}(\bullet) := \text{eval}(\bullet)$;
— for every $a \in \Delta$, $\text{eval}_{\text{sw}}(a) := \text{eval}(a)$;
— $\text{eval}_{\text{sw}}(!(a_1 + \cdots + a_n)) := \text{eval}(!(a_1 + \cdots + a_n))$;
— $\text{eval}_{\text{sw}}(s_1 + s_2) := \text{eval}_{\text{sw}}(s_1) \cup \text{eval}_{\text{sw}}(s_2)$;
— $\text{eval}_{\text{sw}}(s_1 \cdot s_2) := \text{eval}_{\text{sw}}(s_1) \bowtie \text{eval}_{\text{sw}}(s_2)$;
— $\text{eval}_{\text{sw}}(s?) := \text{eval}_{\text{sw}}(s) \cup \text{eval}_{\text{sw}}(\varepsilon)$;
— $\text{eval}_{\text{sw}}(s^+) := \{(u,v) \mid$ there is a simple walk from u to v that matches $s^+\}$;
— $\text{eval}_{\text{sw}}(s^*) := \text{eval}_{\text{sw}}(s^+) \cup \text{eval}_{\text{sw}}(\varepsilon)$;
— $\text{eval}_{\text{sw}}(s^{k,\ell}) := \text{eval}_{\text{sw}}(s)^{k,\ell}$; and
— $\text{eval}_{\text{sw}}(s^{k,\infty}) := \text{eval}_{\text{sw}}(s)^k \bowtie \text{eval}_{\text{sw}}(s^*)$.

Table 3.3.2. Possible factors in extended chain regular expressions and how they are denoted. We denote by a and a_i arbitrary symbols in Δ and by w, w_i non-empty words in Δ^+.

Factor	Abbr.
a	a
a^*	a^*
a^+	a^+
$a?$	$a?$
w^*	w^*
w^+	w^+
$w?$	$w?$

Factor	Abbr.
$(a_1 + \cdots + a_n)$	$(+a)$
$(a_1 + \cdots + a_n)^*$	$(+a)^*$
$(a_1 + \cdots + a_n)^+$	$(+a)^+$
$(a_1 + \cdots + a_n)?$	$(+a)?$
$(a_1^* + \cdots + a_n^*)$	$(+a^*)$
$(a_1^+ + \cdots + a_n^+)$	$(+a^+)$

Factor	Abbr.
$(w_1 + \cdots + w_n)$	$(+w)$
$(w_1 + \cdots + w_n)^*$	$(+w)^*$
$(w_1 + \cdots + w_n)^+$	$(+w)^+$
$(w_1 + \cdots + w_n)?$	$(+w)?$
$(w_1^* + \cdots + w_n^*)$	$(+w^*)$
$(w_1^+ + \cdots + w_n^+)$	$(+w^+)$

Finally, we accept if and only if $\text{eval}_{\text{sw}}(r)$ contains the pair (x, y).

It is easy to see that, in all cases above, $\text{eval}_{\text{sw}}(s)$ can be computed in polynomial time using non-determinism. The only place where non-determinism is needed, is in the case $\text{eval}_{\text{sw}}(s^+)$. Here, to decide if a pair (u, v) should be in the result, the algorithm can simply guess a simple path p from u to v, which trivially has polynomial length, and test whether $\text{lab}(p) \in L(s^+)$.

Correctness of the algorithm is proved similarly as in the proof of Theorem 3.4. The only extra thing that should be taken into account in the induction hypothesis is the non-determinism and the simple walk semantics. Formally, the induction hypothesis states:

There is a run of the algorithm such that, for every subexpression s of r

$$\text{eval}_{\text{sw}}(s) = \{(u, v) \mid \text{there is a path from } u \text{ to } v \text{ that matches } s \text{ under simple walk semantics}\}$$

The proof of the induction is rather straightforward and follows the same lines as the proof of Theorem 3.4. □

It follows that EVALUATION under simple walk semantics is also NP-complete for standard regular expressions.

COROLLARY 3.13. EVALUATION *under simple walk semantics is NP-complete for RE.*

3.3.2. Polynomial Time Fragments. Theorem 3.11 restrains the possibilities for finding polynomial time fragments rather severely. In order to find such fragments and in order to trace a tractability frontier, we will look at syntactically constrained classes of regular expressions that have been used to trace the tractability frontier for the regular expression containment problem [Martens et al. 2004; 2009]. We will also use these expressions in Section 4.

Definition 3.14 (*Chain Regular Expression [Martens et al. 2009]*). A *base symbol* is a regular expression w, w^*, w^+, or $w?$, where w is a non-empty word; a *factor* is of the form e, e^*, e^+, or $e?$ where e is a disjunction of base symbols of the same kind. That is, e is of the form $(w_1 + \cdots + w_n)$, $(w_1^* + \cdots + w_n^*)$, $(w_1^+ + \cdots + w_n^+)$, or $(w_1? + \cdots + w_n?)$, where $n \geq 0$ and $w_1, \ldots, w_n$ are non-empty words. An *(extended) chain regular expression (CHARE)* is $\emptyset$, ε, or a concatenation of factors.

We use the same shorthand notation for CHAREs as in [Martens et al. 2009]. The shorthands we use for the different kind of factors are illustrated in Table 3.3.2. For example, the regular expression $((abc)^* + b^*)(a + b)?(ab)^+(ac + b)^*$ is an extended chain regular expression with factors of the form $(+w^*)$, $(+a)?$, w^+, and $(+w)^*$, from left to right. The expression $(a + b) + (a^*b^*)$, however, is not even a CHARE, due to the nested disjunction. Notice that each kind of factor that is *not* listed in Table 3.3.2 can be simulated through one of the other ones. For example, a factor of the form $(a_1^+ + \cdots + a_n^+)?$ is equivalent to $(a_1^* + \cdots + a_n^*)$. For a similar reason, no factor of the form w is listed. Our interest in these expressions is that CHAREs often occur in practical settings [Bex et al. 2010] and that they are convenient to model classes that allow only a limited amount of non-determinism, which becomes pivotal in Section 4. We denote fragments of the class of CHAREs by enumerating the kinds of factors that are allowed. For example, the above mentioned expression is a CHARE($(+w^*)$, $(+a)?$, w^+, $(+w)^*$).

In the next theorem, we will show that it is possible to use the *- and $^+$-operators and have a fragment for which evaluation is in polynomial time. However, below these operators, one is only allowed to use a disjunction of single symbols.

THEOREM 3.15. EVALUATION *for CHARE*$((+a)^*, (+a)^+, (+w), (+w)?)$ *under simple walk semantics is in P.*

PROOF. The theorem immediately follows from the observation that, for every expression $r \in \text{CHARE}((+a)^*, (+a)^+, (+w), (+w)?)$ it holds that $\text{eval}_{\text{sw}}(r) = \text{eval}(r)$, where $\text{eval}_{\text{sw}}(r)$ is the evaluation for r under simple walk semantics as defined in Section 3.3 and $\text{eval}(r)$ is the evaluation for r under regular path semantics as defined in Section 3.1. Therefore, r can be evaluated in P by Theorem 3.4. $\square$

In Section 2.2 we wrote that our complexity results also hold if we would only allow simple paths instead of simple walks in the formalization of ZeroOrMorePath and OneOrMorePath. The choice between the two makes a minor difference in the proof of the above theorem, but the result holds for both variants. Indeed, EVALUATION for CHARE$((+a)^*, (+a)^+, (+w), (+w)?)$ remains in P, even if we would not allow simple cycles to match expressions of the form $(+a)^*$ or $(+a)^+$. Denote by Id the identity relation and by $\text{eval}'(r)$ the set of pairs that match an expression r under this semantics. It now suffices to observe that, for every subexpression s of the form $(a_1+\cdots+a_k)^*$ or $(a_1+\cdots+a_k)^+$, we have $\text{eval}'(s) = \text{eval}(s) - \text{Id}$, for every subexpression s of the form $(w_1 + \cdots + w_k)$ or $(w_1 + \cdots + w_k)?$, we have $\text{eval}'(s) = \text{eval}(s)$, and that $\text{eval}'(s_1 \cdot s_2) = \text{eval}'(s_1) \bowtie \text{eval}'(s_2)$.

Notice the (perhaps striking) relationship between Theorem 3.15 and Theorem 1 in [Mendelzon and Wood 1995], which states that testing the existence of a simple path that matches the expression a^*ba^* is NP-complete. However, according to Theorem 3.15, testing the existence of a path that matches the expression a^*ba^* under simple walk semantics is in P. The difference, of course, is that under simple walk semantics, we do not require the *entire* path to be simple.

When we would search for more polynomial-time cases, we see that the range of possible fragments between the expressions in CHARE$((+a)^*, (+a)^+, (+w), (+w)?)$ and the expressions in Theorem 3.11 is quite limited. For example, a limitation of Theorem 3.15 is that CHAREs do not allow arbitrary nesting of disjunctions. However, since simple walk semantics and regular path semantics are equal for RE-expressions that do not use the Kleene star or the $^+$-operator, EVALUATION for those expressions under simple walk semantics is tractable as well.

OBSERVATION 3.16. EVALUATION *under simple walk semantics is in P for star-free regular expressions, i.e., RE-expressions that do not use the* **- or* $^+$*-operators.*

We conclude this section by noting that Bagan et al. [Bagan et al. 2013] recently studied a variation of the simple walk semantics (in which the whole regular expression should be matched against a simple path). They study the data complexity of the problem and, under the assumption that $P \neq NP$, they characterize precisely for which kinds of regular expressions evaluation on graphs can be done efficiently.

4. THE COUNTING PROBLEM

In this section we study the complexity of COUNTING. Our motivation for COUNTING comes from the W3C SPARQL working draft [Harris and Seaborne 2012] that requires that, for simple SPARQL queries of the form `SELECT ?x, ?y WHERE {?x r ?y}` where r is a property path, the result is a multiset that has n copies of a pair $(x, y) \in V \times V$, when n is the number of paths between x and y that match r. We informally refer to this requirement as the path counting requirement.

Path Counting Requirement.
The number of paths from x to y that match r needs to be counted.

First, we investigate COUNTING under regular path semantics and then under simple walk semantics. Notice that the number of paths that match an expression is always finite under simple walk semantics. Therefore, to complete the picture for the comparison between regular path semantics and simple walk semantics, we discuss the complexity of FINITENESS in Section 5.

4.1. Regular Path Semantics

We show that it is possible to efficiently solve the COUNTING problem for expressions that are *unambiguous*. Intuitively, an expression is unambiguous when words can only match the expression in one possible manner. Every regular expression can be rewritten into an unambiguous one.[8] A well-known class of unambiguous regular expressions is the class of *deterministic* regular expressions (sometimes also called one-unambiguous regular expressions [Brüggemann-Klein and Wood 1998]), which are used to define content models in Document Type Definitions [Bray et al. 2008] or XML Schema Definitions [Fallside and Walmsley 2004] in the context of XML. So, COUNTING can also be solved efficiently for deterministic regular expressions.

Our formal route goes through automata: We will encode expressions as finite automata through a slight adaptation of standard methods. For these automata, we will have natural notions of non-determinism, determinism, and unambiguity. It is precisely this class of unambiguous finite automata for which we can efficiently solve COUNTING.

We then turn to intractability results and show that COUNTING becomes intractable if we allow more non-determinism. We will consider various classes of expressions that allow only slightly more non-determinism and show that COUNTING becomes #P-complete for all these classes.

4.1.1. Counting for Unambiguous Patterns. We consider finite automata that read Δ-words. The automata behave very similarly to standard finite automata (see, e.g., [Hopcroft and Ullman 1979]), but they can make use of a wildcard symbol "$\circ$" to deal with the infinite set of labels. We will therefore define finite automata with a transition function that is defined over a finite subset Σ of Δ and over the symbol $\circ$, which we assume not to be a member of Σ. More formally, a non-deterministic finite automaton with wildcard NFA^w A over Δ is a tuple $(Q, \Sigma, \Delta, \delta, q_0, Q_f)$, where Q is a finite set of states, $\Sigma \subseteq \Delta$ is a finite alphabet, Δ is the (infinite) set of input symbols,

[8]For example, by converting the expression to a deterministic finite automaton and converting this automaton back into an expression by the "standard" algorithm in [Hopcroft and Ullman 1979], page 33–34.

$\delta \subseteq Q \times (\Sigma \uplus \{\circ\}) \times Q$ is the transition relation, q_0 is the initial state, and Q_f is the set of final states. The *size* of an NFA^w is $|Q|$, i.e., its number of states.

When the NFA^w is in a state q and reads a symbol $a \in \Delta$, we may be able to follow several transitions. The transitions labeled with Σ-symbols can be followed if $a \in \Sigma$. The $\circ$-label in outgoing transitions is used to deal with everything else, i.e., the $\circ$-transitions can be followed when reading $a \notin \Sigma$. Notice that the semantics of the $\bullet$-symbol in regular expressions is therefore different from $\circ$ in automata. The reason for the difference is twofold: first, we want to define a natural notion of determinism for automata and second, our definition of $\circ$ makes it easy to represent subexpressions of the form $!(a_1 + \cdots + a_n)$ in automata.[9] Nevertheless, the expressions from RE($\bullet$) that we defined in Section 2 can still be translated into an equivalent NFA^w in polynomial time. The semantics of NFA^{w}s is defined as follows. A *run* r of an NFA^w $A = (Q, \Sigma, \Delta, \delta, q_0, Q_f)$ on a Δ-word $w = a_1 \cdots a_n$ is a word $q_0 q_1 \cdots q_n$ in Q^* such that, for every $i = 1, \ldots, n$, if $a_i \in \Sigma$, then $(q_{i-1}, a_i, q_i) \in \delta$ and, if $a_i \notin \Sigma$, then $(q_{i-1}, \circ, q_i) \in \delta$. Notice that, when $i = 1$, the condition states that we can follow a transition from the initial state q_0 to q_1. A run is *accepting* when $q_n \in Q_f$. A word w is accepted by A if there exists an accepting run of A on w. The *language* $L(A)$ of A is the set of words accepted by A. A *path* p *matches* A if $\text{lab}(p) \in L(A)$.

We say that an NFA^w is *deterministic*, or a DFA^w, when the relation δ is a function from $Q \times (\Sigma \uplus \{\circ\})$ to Q. That is, for every $q_1 \in Q$ and $a \in \Sigma \uplus \{\circ\}$, there is at most one q_2 such that $(q_1, a, q_2) \in \delta$. An NFA^w is *unambiguous*, or a UFA^w, when, for each word w in $L(A)$, there exists exactly one accepting run of A on w.

In the following, we slightly generalize the definition of s-t graphs and overload their notation. For an edge-labeled graph $G = (V, E)$, $x \in V$, and $Y \subseteq V$, the *s-t graph of G w.r.t. x and Y* is the quadruple (V, E, x, Y). As before, we refer to x as the source node and to Y as the (set of) target nodes. Let $G = (V, E, x, y)$ be an s-t graph and $A = (Q, \Sigma, \Delta, \delta, q_0, Q_F)$ be an NFA^w. We define a *product* of (V, E, x, y) and A, denoted by $G_{x,y} \times A$, similar to the standard product of finite automata. More formally, $G_{x,y} \times A$ is an s-t graph $(V_{G,A}, E_{G,A}, x_{G,A}, Y_{G,A})$, where all of the following hold.

— The set of nodes $V_{G,A}$ is $V \times Q$.
— The source node $x_{G,A}$ is (x, q_0).
— The set of target nodes $Y_{G,A}$ is $\{(y, q_f) \mid q_f \in Q_f\}$.
— For each $a \in \Delta$, there is an edge $\big((v_1, q_1), a, (v_2, q_2)\big) \in E_{G,A}$ if and only if there is an edge (v_1, a, v_2) in G and either
  — $a \in \Sigma$ and there is a transition $(q_1, a, q_2) \in \delta$ or
  — $a \notin \Sigma$ and there is a transition $(q_1, \circ, q_2) \in \delta$ in A.

If A is a UFA^w, then there is a strong correspondence between paths from x to y in G and paths from $x_{G,A}$ to $Y_{G,A}$ in $G_{x,y} \times A$. We formalize this correspondence by a mapping φ_{PATHS}. Therefore let $p = x[a_1]v_1[a_2]v_2 \cdots v_{n-1}[a_n]y$ be a path in G. Then

$$\varphi_{\text{PATHS}}(p) = (x, q_0)[a_1](v_1, q_1)[a_2](v_2, q_2) \cdots (v_{n-1}, q_{n-1})[a_n](y, q_n),$$

where $q_0 q_1 \ldots q_n$ is the unique accepting run of A on the word $a_1 \ldots a_n$. Notice that, since A is unambiguous, there exists only one accepting run for $a_1 \ldots a_n$ and therefore φ_{PATHS} is well-defined.

[9]One could also achieve these goals by defining the semantics of $\circ$ to be "all symbols for which the current state has no other outgoing transition". We thought that the current definition would be clearer since it defines the semantics of every $\circ$-transition the same across the whole automaton.

LEMMA 4.1. *If A is a UFA^w, then φ_{PATHS} is a bijection between paths from x to y in G that match A and paths from $x_{G,A}$ to some node in $Y_{G,A}$ in $G_{x,y} \times A$. Furthermore, φ_{PATHS} preserves the length of paths.*

PROOF. Observe that the mapping φ_{PATHS} preserves the length and label of a path by definition. We show that φ_{PATHS} is a bijection from

— the set $P(G)$ of paths from x to y in G that match A; to
— the set $P(G \times A)$ of paths from $x_{G,A}$ to some node in $Y_{G,A}$ in $G_{x,y} \times A$

by proving that φ_{PATHS} is surjective and injective. To this end, let $A = (Q, \Sigma, \Delta, \delta, q_0, Q_f)$ be a UFA^w.

First, we show that φ_{PATHS} is surjective. Let $p = (x, q_0)[a_1](v_1, q_1)[a_2](v_2, q_2) \cdots (v_{n-1}, q_{n-1})[a_n](y, q_n)$ be an arbitrary path in $P(G_{x,y} \times A)$ with $(y, q_n) \in Y_{G,A}$. We prove that $p = \varphi_{\text{PATHS}}(p^G)$, where $p^G = x[a_1]v_1[a_2]v_2 \cdots v_{n-1}[a_n]y$. By definition of $G_{x,y} \times A$, we have that there is an edge $((v_i, q_i), a, (v_j, q_j))$ if and only if there is a transition in δ that takes q_i to q_j by reading a. As such, the existence of p in $G_{x,y} \times A$ implies that there is an accepting run $q_0 \cdots q_n$ of A on the word $a_1 \cdots a_n$. By the unambiguity of A, the run $q_0 \cdots q_n$ is unique. Therefore, by definition of φ_{PATHS}, we have that

$$\begin{aligned}\varphi_{\text{PATHS}}(p^G) &= \varphi_{\text{PATHS}}(x[a_1]v_1[a_2]v_2 \cdots v_{n-1}[a_n]y) \\ &= (x, q_0)[a_1](v_1, q_1)[a_2](v_2, q_2) \cdots (v_{n-1}, q_{n-1})[a_n](y, q_n) = p\end{aligned}$$

We now show that φ_{PATHS} is injective. Let $p = x[a_1]v_1 \cdots v_{n-1}[a_n]y$ and $p' = x[a'_1]v'_1 \cdots v'_{n-1}[a'_n]y$ be two paths in $P(G)$ such that $\varphi_{\text{PATHS}}(p) = \varphi_{\text{PATHS}}(p')$. We prove that $p = p'$. Let $\varphi_{\text{PATHS}}(p) = (x, q_0)[a_1](v_1, q_1)[a_2](v_2, q_2) \cdots (v_{n-1}, q_{n-1})[a_n](y, q_n)$. Since φ_{PATHS} preserves the labels on edges and the nodes in V, it follows that $p = x[a_1]v_1 [a_2]v_2 \cdots v_{n-1}[a_n]y = p'$. □

We recall the following graph-theoretical result that states that the number of arbitrary paths between two nodes in a graph can be counted efficiently (see, e.g., [Berge 1973], page 74):

THEOREM 4.2. *Let G be a graph, let x and y be two nodes of G, and let max be a number given in binary. Then, the number of paths from x to y of length at most max can be computed in time polynomial in G and the number of bits of max.*

The reason why the number of paths can be counted so efficiently is due to fast squaring, of which we summarized a variant in the proof of Lemma 3.2. With fast squaring we can compute, for a square matrix M, the matrix M^k by performing $O(\log k)$ matrix multiplications. Furthermore, if M is the connectivity matrix of $G = (V, E)$ in which $M[u, v] = 1$ if $(u, v) \in E$ and $M[u, v] = 0$ otherwise, then $M^k[x, y]$ is the number of paths from x to y of length at most k.

THEOREM 4.3. COUNTING *for UFA^{w}s is in polynomial time, even if the number max in the input is given in binary.*

PROOF. We reduce COUNTING for UFA^{w}s to the problem of counting the number of paths in a graph, which is in polynomial time even when max is in binary, due to Theorem 4.2.

Let $G = (V, E, x, y)$ be a graph and $A = (Q, \Sigma, \Delta, \delta, q_0, Q_f)$ be a UFA^w. The algorithm works as follows:

— Let $G_{x,y} \times A$ be the product of (V, E, x, y) and A.
— Return $\sum_{q_f \in Q_f} \text{PATHS}\big((x, q_0), (y, q_f)\big)$ in $G_{x,y} \times A$.

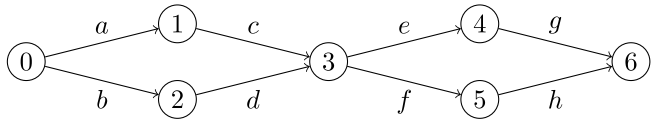


Fig. 3. An edge-labeled graph $(V, E, 0, 6)$.

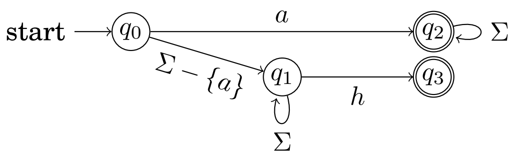


Fig. 4. A UFA^w A for the regular expression $(a\Sigma^* + \Sigma^+ h)$, where Σ is an abbreviation for the expression $(a + b + c + d + e + f + g + h)$.

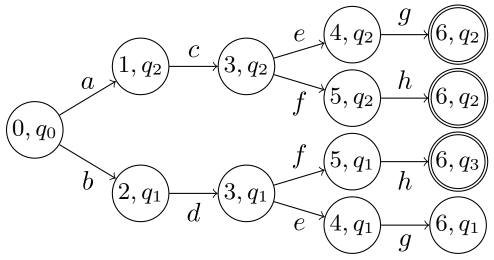


Fig. 5. Fragment of the product $G_{0,6} \times A$ of $(V, E, 0, 6)$ from Figure 3 and UFA^w A from Figure 4. The nodes in $Y_{G,A}$ are in double circles.

Here, $\textsc{Paths}\big((x, q_0), (y, q_f)\big)$ denotes the number of paths of length at most max in $G_{x,y} \times A$ from node (x, q_0) to (y, q_f). By Lemma 4.1, this algorithm is correct. Indeed, the lemma shows that the number of paths of length at most max in G between x and y and that are matched by A equals the number of paths of length at most max from (x, q_0) to some node in $\{y\} \times Q_f$ in $G_{x,y} \times A$. □

Since the class of DFA^{w}s is a strict subset of the UFA^{w}s, the following corollary is immediate.

COROLLARY 4.4. COUNTING *for DFA^{w}s is in polynomial time, even if the number max in the input is given in binary.*

We illustrate the algorithm of Theorem 4.3 on an example. Consider the UFA^w A in Figure 4. The product of A and the s-t graph $(V, E, 0, 6)$ from Figure 3 is depicted in Figure 5. We see that the number of paths in G from node 0 to 6 that match A is precisely the number of paths from the source node to a target node in the product.

From Automata to Expressions. From the automata classes in Theorem 4.3 and Corollary 4.4 we can now infer classes of SPARQL regular expressions for which COUNTING can be solved in polynomial time. In general, one can say that COUNTING can be solved in polynomial time for each class of SPARQL regular expressions that can be converted in polynomial time into UFA^{w}s.

We will present a concrete such class of SPARQL regular expressions by revisiting the *Glushkov-automaton* of a regular expression (see also [Book et al. 1971; Glushkov 1961]). Let r be a SPARQL regular expression in RE$(!, \bullet)$ and let Σ_r be the set of Δ-

symbols occurring in r. By num(r) we denote the *numbered regular expression* obtained from r by replacing each subexpression of the form $!(a_1 + \cdots + a_n)$, $\bullet$, or $a \in \Sigma_r$ (that is not in the scope of an !-operator) with a unique number, increasing from left to right. For example, for $r = a\ !(a) \bullet (a + bc)^* \bullet\ !(a + b)$ we have num(r) $= 1\ 2\ 3\ (4\ \ + 5\ 6)^* 7\ 8$. Formally, num($r$) can be obtained by traversing the parse tree of r depth-first left-to-right and replacing each atomic expression by a unique number. By denum$_r$ we denote the mapping that maps each number i to the subexpression it replaced in r. In the above example, denum$_r(1) = a$, denum$_r(2) =!(a)$, denum$_r(3) = \bullet$, and so on.

Fix an expression r and its numbered expression r_m. Notice that r_m can be seen as a regular expression over a finite alphabet $\Sigma' \subseteq \mathbb{N}$ where $|\Sigma'|$ is the number of leaves in the parse tree of r. Let $\text{first}(r_m)$ be the set of all symbols $i \in \Sigma'$ such that $L(r_m)$ contains a word iz, where $z \in (\Sigma')^*$. Furthermore, for $i \in \Sigma'$, let $\text{follow}(r_m, i)$ be the set of symbols $j \in \Sigma'$ such that there exist Σ'-words v, w with $vijw \in L(r_m)$, and let $\text{last}(r_m)$ be the set of symbols $i \in \Sigma'$ such that there exists a word vi in $L(r_m)$. The *Glushkov-automaton* G_r of r is the tuple $(Q_r, \Sigma_r, \Delta, \delta_r, q_0, Q_f)$ where $Q_r = \{q_0\} \uplus \Sigma'$ is its finite set of states. That is, Q_r contains an initial state and one state for each Σ'-symbol i in the numbered expression r_m. If $\varepsilon \in L(r)$, then the set of accepting states is $Q_f = \text{last}(r_m) \uplus \{q_0\}$; otherwise, $Q_f = \text{last}(r_m)$. For each $a \in \Sigma_r$ and $i \in Q_r$, the transition function δ_r is defined as follows: (1) $\delta_r(q_0, a) = \{i \in \text{first}(r_m) \mid \text{denum}(i) = a, \text{denum}(i) = \bullet$, or $\text{denum}(i) = !(a_1 + \cdots + a_\ell)$ with $a \notin \{a_1, \ldots, a_\ell\}\}$ and (2) $\delta_r(i, a) = \{j \in \text{follow}(r_m, i) \mid \text{denum}(j) = a, \text{denum}(j) = \bullet$, or $\text{denum}(j) = !(a_1 + \cdots + a_\ell)$ with $a \notin \{a_1, \ldots, a_\ell\}\}$. Furthermore, (3) $\delta_r(q_0, \circ) = \{i \in \text{first}(r_m) \mid \text{denum}(i) = \bullet$ or $\text{denum}(i) = !(a_1 + \cdots + a_\ell)$ for some $a_1, \ldots, a_\ell\}$, and (4) $\delta_r(i, \circ) = \{j \in \text{follow}(r_m, i) \mid \text{denum}(j) = \bullet$ or $\text{denum}(j) = !(a_1 + \cdots + a_\ell)$ for some $a_1, \ldots, a_\ell\}$. The following proposition can be proved similarly as the corresponding one for Glushkov automata for ordinary regular expressions.

PROPOSITION 4.5. *For each SPARQL regular expression r in RE($!, \bullet$), the Glushkov-automaton of r can be constructed in polynomial time. Furthermore, $L(r) = L(G_r)$.*

We say that a regular expression $r \in$ RE($!, \bullet$) is *deterministic*, or a *Det-RE*, if G_r is a DFA^w. It is *unambiguous*, if G_r is a UFA^w.

COROLLARY 4.6. COUNTING *for unambiguous (or deterministic) regular expressions under regular path semantics is in polynomial time, even if the number max in the input is given in binary.*

4.1.2. Counting for Ambiguous Patterns. In this section, we investigate the complexity of COUNTING for more general patterns than in the previous section and prove that COUNTING already becomes intractable for very slight extensions. We start by observing that COUNTING is in #P for standard regular expressions.

THEOREM 4.7. COUNTING *is in #P for all REs.*

PROOF. Let $G = (V, E, x, y)$ be a graph, r be an RE, and $max \in \mathbb{N}$ be a number given in unary notation. The non-deterministic Turing machine for the $\#P$ procedure simply guesses a path of length at most max and tests whether it matches $L(r)$. $\square$

We now prove that COUNTING becomes #P-hard for a wide array of restricted REs that allow for a very limited amount of non-determinism. We consider the chain regular expressions introduced in Section 3.3.2. For example, the class of CHARE(a,a?) seems, at first sight, to be very limited. However, such expressions cannot be translated to polynomial-size UFA^{w}s in general. We show that COUNTING is #P-complete for all classes of CHAREs that allow a single label (i.e., "a") as a factor and cannot be trivially converted to polynomial-size DFA^{w}s or UFA^{w}s.

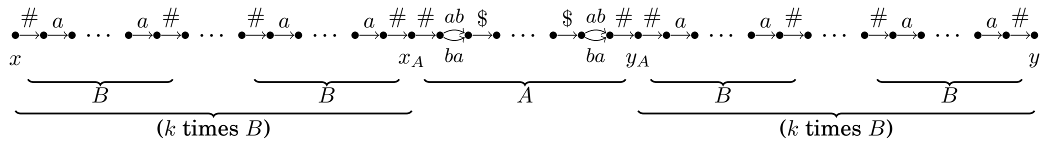


Fig. 6. The graph G from the proof of Theorem 4.8.

THEOREM 4.8. COUNTING *is #P-complete for all of the following classes: (1) CHARE*(a, a^*)*, (2) CHARE*$(a, a?)$*, (3) CHARE*(a, w^+)*, (4) CHARE*$(a, (+a^+))$*, (5) CHARE*$(a, (+a)^+)$*, and (6) CHARE*$((+a), a^+)$*. Moreover, #P-hardness already holds if the graph G is acyclic.*

PROOF SKETCH. The upper bound for all cases is immediate from Theorem 4.7. The lower bounds can be proved by reductions from #DNF. We show the reduction for case (1) and refer to the Appendix for a more general proof that deals with all cases and shows that the reduction is correct. Our technique is inspired by a proof in [Martens et al. 2009], where it is shown that language inclusion for various classes of CHAREs is coNP-hard. Let $\Phi = C_1 \vee \cdots \vee C_k$ be a propositional formula in 3DNF using variables $\{x_1, \ldots, x_n\}$. We encode truth assignments for Φ by paths in the graph. In particular, we construct a graph (V, E, x, y), an expression r, and a number max such that each path of length at most max in G from x to y that matches r corresponds to a unique satisfying truth assignment for Φ and vice versa. Formally, we will have that the number of paths of length at most max in G from x to y that match r is equal to the number of truth assignments that satisfy Φ.

The graph G has the structure as depicted in Figure 6, where *(i)* B is a path labeled $\#a\$a\$\cdots\$a\#$ (with n copies of a) and *(ii)* A is a subgraph as depicted in Figure 6, with n copies of the gadget labeled ab/ba. Notice that all paths from x to y will enter A through the node x_A and leave A through y_A. Notice that G is acyclic.

Each path from x_A to y_A in A corresponds to exactly one truth assignment for the variables $\{x_1, \ldots, x_n\}$: if the path chooses the i-th subpath labeled ab, this means that x_i is "true". If it chooses ba, it means that x_i is "false". This concludes the description of the graph.

The expression r has the form

$$r = (\#^* a^* \$^* \cdots \$^* a^* \#^*)^k F(C_1) \cdots F(C_k) (\#^* a^* \$^* \cdots \$^* a^* \#^*)^k,$$

where, for each $i = 1, \ldots, k$, we define $F(C_i)$ as $\#e_1\$\cdots\$e_n\#$ with, for each $j = 1, \ldots, n$,

$$e_j := \begin{cases} b^*a^*, & \text{if } x_j \text{ occurs negated in } C_i, \\ a^*b^*, & \text{if } x_j \text{ occurs positively in } C_i, \text{ and} \\ a^*b^*a^*, & \text{otherwise.} \end{cases}$$

This concludes the reduction for case (1). □

We conclude this section by stating the general #P upper bound on the counting problem.

THEOREM 4.9. COUNTING *for RE*$(\#, !, \neg, \bullet)$ *is #P-complete.*

PROOF. The lower bound follows from Theorem 4.8. Since MEMBERSHIP for RE$(\#, !, \neg, \bullet)$ is in polynomial time by Theorem 3.7 and since the number max is given in unary, the upper bound follows analogously to the proof of Theorem 4.7. □

4.2. Simple Walk Semantics

We investigate how the complexity of COUNTING changes when we apply simple walk semantics. The picture is even more drastic than in Section 3.3. COUNTING already turns #P-complete as soon as the Kleene star or plus are used. We start by mentioning a polynomial-time result.

THEOREM 4.10. COUNTING *under simple walk semantics for CHARE*($a, (+a)$) *is in P.*

This result trivially holds since, for this fragment, simple walk semantics is the same as regular path semantics and expressions from this fragment can be translated into DFA^{w}s in polynomial time.

THEOREM 4.11. COUNTING *under simple walk semantics is #P-complete for the expressions* a^* *and* a^+.

PROOF. This is a straightforward reduction from the problem of counting the number of simple s-t paths in a graph, which was shown to be #P-complete by Valiant [Valiant 1979]. Since we can assume that the source node and target node are different, the hardness result holds for simple walks as well.

Given an s-t graph $G = (V, E, x, y)$ so that $x \neq y$ and a number max in unary, we construct an edge-labeled s-t graph $G' = (V, E', x, y)$ by labeling each edge with a. The number of simple walks from x to y in G of length at most max is equal to the number of paths from x to y in G' of length at most max that match the regular expression a^*. The reduction for the expression a^+ is similar. □

Theorem 4.11 immediately implies that COUNTING under simple walk semantics is #P-complete for CHARE(a, a^*), CHARE(a, w^+), CHARE(a, $(+a^+)$), CHARE(a, $(+a)^+$), and CHARE(a^+, $(+a)$) as well. The result for CHARE(a,$a?$) is not immediate from Theorem 4.11, but it is immediate from the observation that the reduction for regular path semantics applies here as well.

THEOREM 4.12. COUNTING *under simple walk semantics for CHARE*($a, a?$) *is #P-complete.*

Finally, we mention that COUNTING is in #P for the full fragment of SPARQL regular expressions, i.e., expressions in RE($\#, !, \bullet$).

THEOREM 4.13. COUNTING *under simple walk semantics for RE*($\#, !, \bullet$) *is #P-complete.*

PROOF. The #P algorithm guesses a path from x to y in the graph of length at most max and then tests whether it matches the expression under simple walk semantics. Since the number max is given in unary, the algorithm can guess the entire path. (We need to guess the nodes, as well as the labels.) We then run the dynamic programming algorithm on words (i.e., the one from Theorem 3.7) on the path, but we remove all pairs in all relations that do not correspond to a match under simple walk semantics. In particular, all pairs (x, y) in a relation eval(s^*) such that the subpath from x to y is not a simple walk, are removed from the relation. (Notice that, since we already guessed the path, there is a unique subpath from x to y on this path.) Otherwise, the algorithm is unchanged. Since the only nondeterminism in the algorithm comes from guessing the path, the number of accepting computations of the #P algorithm corresponds to the number of paths of length at most max matching the expression. □

5. THE FINITENESS PROBLEM

Under simple walk semantics, there can never be an infinite number of paths that match a certain regular expression. Under regular path semantics, however, this can be the case. So, in order to be able to make a fair comparison of the complexity of counting paths for regular path semantics versus simple walk semantics, we also need to consider the FINITENESS problem. In this section, we only consider regular path semantics.

Using the product construction (Section 4.1.1), we can test in polynomial time whether there is a path from x to y that is labelled uvw, such that v labels a loop and such that uv^kw matches r for every $k \in \mathbb{N}$. If there is such a loop, then we return that there are infinitely many paths.

OBSERVATION 5.1. FINITENESS *is in P for RE. More precisely, it can be decided in time* $O(|r| \cdot |G|)$.

By adapting the polynomial time algorithm of Section 3.1 to also annotate the length of the longest paths associated to a pair in each relation, we can even decide FINITENESS for RE($\#, !, \bullet$) in P.

THEOREM 5.2. FINITENESS *for RE($\#, !, \bullet$) is in P.*

PROOF. The underlying idea of the proof is a pumping argument: the number of paths from x to y that match r is infinite if and only if there is a very long path from x to y that matches r. Furthermore, it suffices to consider paths of exponential length in $|r|$. This can be seen as follows: if we translate the RE($\#, !, \bullet$) expression r to an RE($!, \bullet$) expression r_0 (by unfolding the counters, i.e., replace subexpressions of the form $s^{k,k}$ with k concatenation of s), then the size of r_0 is exponential in the size of r. Let A be an NFA^w for $L(r)$. Again, A can be constructed in time exponential in the size of r. If we consider the product $G_{x,y} \times A = (V_{G,A}, E_{G,A}, x_{G,A}, Y_{G,A})$ defined in Section 4.1.1, then there are infinitely many paths from x to y in G that match r if and only if there are infinitely many paths from $x_{G,A}$ to some node in $Y_{G,A}$. The latter holds if and only if there exists a path from $x_{G,A}$ to some node in $Y_{G,A}$ of length at least $|G| \cdot |A| + 1$, due to a pumping argument. Since the size $|A|$ is exponential in $|r|$, we therefore only need to consider lengths of paths that are exponential in $|r|$.

The main idea is to remember the lengths of paths in the algorithm of Section 3.1 for EVALUATION for RE($\#, !, \bullet$) while trying to find paths that are as long as possible. Once a path becomes longer than $M := |G| \cdot |A| + 1$, we simply remember that the path is long enough.

The algorithm works as follows. Let r be an RE($\#, !, \bullet$)-expression and let $G = (V, E)$ be a graph. We compute, for all subexpressions s of r the ternary relation

$$\mathrm{eval_c}(s) \subseteq V \times V \times \{0, \dots, |G| \cdot |A| + 1\}$$

such that, intuitively, if $(u, v, i) \in \mathrm{eval_c}(s)$, it means that we have found a path from u to v that matches s and has length at least i. More formally, all of the following hold:

— for each $(u, v) \in V \times V$, there is at most one triple of the form $(u, v, i) \in \mathrm{eval_c}(s)$;
— for each $(u, v, i) \in \mathrm{eval_c}(s)$ with $i \in \{0, \dots, |G| \cdot |A|\}$, there exists a path from u to v of length i in G that matches s;
— if there is a path from u to v in G that matches s, there exists a triple $(u, v, i) \in \mathrm{eval_c}(s)$; and
— there is a path from u to v in G of length at least $|G| \cdot |A| + 1$ that matches s if and only if $(u, v, |G| \cdot |A| + 1) \in \mathrm{eval_c}(s)$.

We now discuss how $\mathrm{eval_c}(s)$ can be defined inductively on the structure of RE($\#, !, \bullet$) expressions:

— $\text{eval}_\mathsf{c}(\emptyset) := \emptyset$;
— $\text{eval}_\mathsf{c}(\varepsilon) := \{(u,u,0) \mid u \in V\}$;
— $\text{eval}_\mathsf{c}(\bullet) := \{(u,v,1) \mid \exists a \in \Delta \text{ s.t. } (u,a,v) \in E\}$;
— for every $a \in \Delta$, we define $\text{eval}_\mathsf{c}(a) := \{(u,v,1) \mid (u,a,v) \in E\}$;
— $\text{eval}_\mathsf{c}(!(a_1 + \cdots + a_n)) := \{(u,v,1) \mid \exists a \in \Delta - \{a_1,\ldots,a_n\} \text{ s.t. } (u,a,v) \in E\}$;
— $\text{eval}_\mathsf{c}(s_1 + s_2)$ is the set of all (u,v,i) such that
  — $(u,v,k) \in \text{eval}_\mathsf{c}(s_1)$ and $(u,v,\ell) \in \text{eval}_\mathsf{c}(s_2)$ and $i = \min(\max(k,\ell), |G| \cdot |N| + 1)$, or
  — $(u,v,i) \in \text{eval}_\mathsf{c}(s_1)$ and there is no j such that $(u,v,j) \in \text{eval}_\mathsf{c}(s_2)$, or
  — $(u,v,i) \in \text{eval}_\mathsf{c}(s_2)$ and there is no j such that $(u,v,j) \in \text{eval}_\mathsf{c}(s_1)$;
— $\text{eval}_\mathsf{c}(s_1 \cdot s_2)$ is the set of all triples (u,v,i) such that there is a node z such that $(u,z,k) \in \text{eval}_\mathsf{c}(s_1)$, $(z,v,\ell) \in \text{eval}_\mathsf{c}(s_2)$, and $i = \min(k + \ell, |G| \cdot |N| + 1)$;
— $\text{eval}_\mathsf{c}(s?) := \text{eval}_\mathsf{c}(s + \varepsilon)$;
— $\text{eval}_\mathsf{c}(s^*) := \text{eval}_\mathsf{c}((s + \varepsilon)^{|G|\cdot|N|+1})$;
— $\text{eval}_\mathsf{c}(s^+) := \text{eval}_\mathsf{c}(s \cdot s^*)$;
— for $k \in \mathbb{N}$, $\text{eval}_\mathsf{c}(s^k)$ is inductively defined as
  — If $k = 1$, then $\text{eval}_\mathsf{c}(s^k) = \text{eval}_\mathsf{c}(s)$.
  — Otherwise, if k is even, then $\text{eval}_\mathsf{c}(s^k) := \text{eval}_\mathsf{c}(s^{k/2} \cdot s^{k/2})$.
  — Otherwise, if k is odd, then $\text{eval}_\mathsf{c}(s^k) := \text{eval}_\mathsf{c}(s \cdot s^{k/2} \cdot s^{k/2})$.
— $\text{eval}_\mathsf{c}(s^{k,\infty}) := \text{eval}_\mathsf{c}(s^k \cdot s^*)$; and
— if $\ell \neq \infty$, then $\text{eval}_\mathsf{c}(s^{k,\ell}) := \text{eval}_\mathsf{c}(s^k \cdot (s?)^{\ell-k})$.

Computing $\text{eval}_\mathsf{c}(r)$ in polynomial time can be done by following the above inductive definition and storing all intermediate results for avoiding recomputations. Finally, if the input for GRAPHEVAL is G, nodes x and y, and RE($\#, !, \bullet$)-expression r, we return the answer "true" if and only if $\text{eval}_\mathsf{c}(r)$ contains the triple $(x, y, |G| \cdot |N| + 1)$.

The proof of correctness is a straightforward induction that follows the same lines as the proof of Theorem 3.4. □

Similar to EVALUATION, the complexity of FINITENESS becomes non-elementary once unrestricted negation is allowed in regular expressions. Analogously to EVALUATION we show that FINITENESS is at least as hard as satisfiability of a given regular expression.

LEMMA 5.3. *Let C be a class of regular expressions r over a finite alphabet Σ such that testing whether $\varepsilon \in L(r)$ is in polynomial time. Then there exists a polynomial reduction from the emptiness problem for C-expressions to the* FINITENESS *problem with C-expressions.*

PROOF. Let r be a C-expression over Σ. We construct a graph (V, E, x, y) and a C-expression s such that $L(r) = \emptyset$ if and only if FINITENESS is "true" for s and (V, E, x, y).

First we test whether $\varepsilon \in L(r)$ or not. Notice that we can test this for the expression r in polynomial time. If $\varepsilon \in L(r)$, then we know that $L(r) \neq \emptyset$. Therefore we return the expression a^* and the graph $G = (V, E, x, x)$ with $V = \{x\}$ and $E = \{(x, a, x)\}$. If $\varepsilon \notin L(r)$, then we return the expression r^* and the graph $G = (V, E, x, x)$ with $V = \{x\}$ and $E = \{(x, a, x) \mid a \in \Sigma\}$. This concludes the reduction.

We now prove that the reduction is correct. In the case where $\varepsilon \in L(r)$, we have that $L(r) \neq \emptyset$ and we constructed an instance for which FINITENESS is always "false", so the reduction is correct.

In the case where $\varepsilon \notin L(r)$ we know that either $L(r) = \emptyset$ or $L(r)$ contains at least one word w with $|w| > 0$. If $L(r) = \emptyset$, then $L(r^*) = \{\varepsilon\}$ and there exists only one path with length 0 in G that matches r^*, i.e., FINITENESS returns "true". If $L(r) \neq \emptyset$, then it follows that $w^i \in L(r^*)$ for all $i \geq 1$. Thus, for every $i \geq 1$, there is a path p_i with $\text{lab}(p_i) = w^i$ in G. Since all paths p_i and p_j are different when $i \neq j$, this means that

$$\begin{array}{rcl}
[\![\varepsilon]\!]_G &=& \{(u,u) \mid u \in V\} \\
[\![\bullet]\!]_G &=& \{(u,v) \mid (u,a,v) \in E \text{ for some } a \in \Delta\} \\
[\![a]\!]_G &=& \{(u,v) \mid (u,a,v) \in E\} \\
[\![a^-]\!]_G &=& \{(u,v) \mid (v,a,u) \in E\} \\
[\![!(a_1 + \cdots + a_n)]\!]_G &=& \{(u,v) \mid \exists a \in \Delta - \{a_1,\ldots,a_n\} \text{ with } (u,a,v) \in E\} \\
[\![r+s]\!]_G &=& [\![r]\!]_G \cup [\![s]\!]_G \\
[\![r \cdot s]\!]_G &=& [\![r]\!]_G \bowtie [\![s]\!]_G \\
[\![r^{k,\ell}]\!]_G &=& ([\![r]\!]_G)^k \bowtie ([\![r+\varepsilon]\!]_G)^\ell \\
[\![\langle r \rangle]\!]_G &=& \{(u,u) \mid \exists z :\ (u,z) \in [\![r]\!]_G\}
\end{array}$$

Fig. 7. Regular paths semantics of NREs with respect to a graph $G = (V, E)$.

there exist infinitely many paths p in G that match r^*, i.e. FINITENESS returns “false”. Therefore, FINITENESS returns “true” if and only if $L(r) = \emptyset$. $\square$

For $r \in$ RE($\neg$), one can test whether $\varepsilon \in L(r)$ in linear time traversing the syntax tree of r. Therefore, by [Stockmeyer 1974] and Lemma 5.3 we have the following.

THEOREM 5.4. FINITENESS *is decidable but non-elementary for RE($\neg$).*

6. NESTED REGULAR EXPRESSIONS

Until now, we have only considered decision problems for single regular expressions in this article. In this section we want to show that our complexity upper bound for EVALUATION also holds for more complex queries. We will consider a variant of *nested regular expressions (NREs)*, which we equip with the numerical occurrence operators, wildcard, and negated label test of Section 2. Nested regular expressions were studied by Pérez et al. who showed that their evaluation problem is in linear time [Pérez et al. 2010]. This shows that NREs are an interesting class of queries because the complexity of evaluating them is essentially not worse than for ordinary regular expressions. In this section we see that the same holds when we build NREs from SPARQL regular expressions.

We now formally define a variant of nested regular expressions which is equipped with the operators from the SPARQL recommendation [Harris and Seaborne 2012]. The syntax of *SPARQL nested regular expressions (or SPARQL NREs)* r, s is defined as follows:

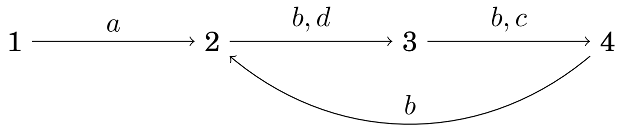
$$r, s := \varepsilon \mid \bullet \mid a \mid a^- \mid !(a_1 + \cdots + a_n) \mid r + s \mid r \cdot s \mid r^{k,\ell} \mid \langle r \rangle,$$

where $a, a_1, \ldots, a_n \in \Delta$, $k \in \mathbb{N}$, and $\ell \in \{k, k+1, \ldots, \infty\}$.

The regular path semantics of SPARQL NREs is defined in Figure 7. We note that we have added one more operator to SPARQL regular expressions which is a^- for $a \in \Delta$. This expression allows to navigate through an a-edge in the reverse direction and is also inspired by a corresponding operator in [Harris and Seaborne 2012]. SPARQL NREs strictly generalize SPARQL regular expressions by extending them with the *nesting operator* $\langle\cdot\rangle$. As such, all the lower bounds from this article immediately transfer to SPARQL NREs. Therefore, we will only consider upper bounds in this section. Furthermore, we only define the regular path semantics for SPARQL NREs since most of our results on simple walk semantics are lower bounds.

We illustrate the semantics of SPARQL NREs by means of an example for the EVALUATION problem.

Example 6.1. Consider the SPARQL NRE $r = a\langle (b^{2,2})^* c \rangle d$ and the graph G from Figure 8. For r and a tuple (x, y) of nodes in G, it holds that, $(x, y) \in$ EVALUATION if and only if there exists a path from x to y over some node z labeled with ad and there



$$\begin{tikzcd}1 \arrow[r, "a"] & 2 \arrow[r, "{b,d}"] & 3 \arrow[r, "{b,c}"] & 4 \arrow[ll, bend left, "b"]\end{tikzcd}$$

Fig. 8. An edge-labeled graph G

exists a path from z to some node in G which is labeled with a word in $L((b^{2,2})^*c)$. Notice that none of these nodes has to be distinct to another under regular path semantics. For example, for the graph from Figure 8, it holds that EVALUATION is true for $(1, 3)$, since there is a path from 1 to 3 labeled with ad and a path from 2 to 4 labeled with $bbbbc \in L((b^{2,2})^*c)$. (The path from 2 to 4 is not a simple walk.)

In the following we generalize the upper bound of Theorem 3.4 to SPARQL NREs.

THEOREM 6.2. EVALUATION *for SPARQL NREs under regular path semantics is in time $O(|r| \cdot |V|^3)$, where $|r|$ is the size of the NRE and $|V|$ is the number of nodes in the graph.*

PROOF. We prove that we can extend the dynamic programming algorithm from Section 3.1 to decide EVALUATION for NREs in polynomial time. That is, given a graph $G = (V, E, x, y)$ and an NRE r, it decides in polynomial time whether (x, y) is selected by r, that is, whether $(x, y) \in [\![r]\!]_G$.

We extend our previous algorithm that computes the relation eval$(s) = [\![s]\!]_G$ with two cases:

— for every $a \in \Delta$, eval$(a^-) := \{(v, u) \mid (u, a, v) \in E\}$; and
— eval$(\langle s \rangle) := \{(u, u) \mid \exists z : (u, z) \in \text{eval}(s)\}$.

Clearly, these additional cases can be added to the algorithm without changing the worst-case time complexity (both steps can be computed in linear time).

We argue correctness similar to the proof of Theorem 3.4. We claim that the following invariants hold:

For each subexpression $s \neq \langle s_1 \rangle$ of r, we have

$$(u, v) \in \text{eval}(s) \Leftrightarrow \exists \text{ path } p \text{ in } G \text{ from } u \text{ to } v \text{ sucht that lab}(p) \in L(s). \qquad \text{(I)}$$

and

For each subexpression $s = \langle s_1 \rangle$ of r, we have

$$(u, v) \in \text{eval}(s) \Leftrightarrow (u = v)$$
$$\wedge (\exists \text{ node } z, \text{ path } p \text{ from } u \text{ to } z \text{ in } G \text{ such that lab}(p) \in L(s_1)). \qquad \text{(I')}$$

If (I) and (I') are correct, then EVALUATION for (V, E, x, y) and r is true if and only if $(x, y) \in \text{eval}(r)$. The proof of invariant (I) is analogous to the proofs of the cases in Theorem 3.4 and the proof of (I') is immediate from the definition of eval$(\langle s \rangle)$. □

Thus, the complexity of the EVALUATION problem does not suffer from considering NREs instead of ordinary regular expressions. In [Libkin et al. 2013] it is shown that this can even be extended to variants of SPARQL NREs that use more powerful negation.

7. DISCUSSION

An overview of our results is presented in Table I. CHAREs are defined in Section 3.3.2. By *star-free RE*, we denote regular expressions that only use "+" (disjunction) and "·"

Table I. An overview of most of our combined complexity results.

Problem	Fragment	Regular path semantics	Simple walk semantics
EVALUATION	CHARE($(+a)^*$,$(+a)^+$, $(+w)$,$(+w)?$)	in P	**in P** (3.15)
	star-free RE	in P	**in P** (3.16)
	$(aa)^*$	in P	NP [LP84, MW95]
	RE, RE($\#, !, \bullet$)	**in P** (3.4)	**NP** (3.12)
	RE($\neg$)	**non-elem.** (3.9,3.10)	—
	RE($\#, !, \neg, \bullet$)	**non-elem.** (3.9,3.10)	—
COUNTING	DFA^w, UFA^w	**in FP** (4.3,4.4)	—
	CHARE(a, $(+a)$)	**in FP**	**in FP** (4.10)
	a^+, a^*	**in FP**	#P (4.11, [V79])
	Det-RE	**in FP**	**#P**
	CHARE($a, a?$)	**#P** (4.7,4.8)	**#P**(4.12)
	CHARE(a, a^*), CHARE($a, (+a^+)$), CHARE($a, (+a)^+$), CHARE(a, w^+), CHARE($(+a), a^+$), RE	**#P** (4.7,4.8)	**#P**
	RE($\#, !, \bullet$)	**#P** (4.9)	**#P** (4.13)
	RE($\#, !, \neg, \bullet$)	**#P** (4.9)	—
FINITENESS	RE, RE($\#, !, \bullet$)	**in P** (5.1,5.2)	—
	RE($\neg$), RE($\#, !, \neg, \bullet$)	**non-elem.** (5.4)	—

Note: The results printed in bold are new, to the best of our knowledge. All complexities are completeness results, unless stated otherwise. The entries marked by "—" signify that the question is either trivial or not defined. We annotated new results with the relevant theorem numbers. If no such number is provided, it means that the result directly follows from other entries in the table.

Table II. Summary of our results when interpreted under data complexity.

Problem	Fragment	Regular path semantics	simple walk semantics
EVALUATION	CHARE($(+a)^*$,$(+a)^+$, $(+w)$,$(+w)?$)	in P	in P
	star-free RE	in P	in P
	$(aa)^*$	in P	NP [LP84, MW95]
	RE, RE($\#, !, \neg, \bullet$)	in P	NP
COUNTING	DFA^w, UFA^w, NFA^w	in FP	—
	CHARE(a, $(+a)$), CHARE($a, a?$)	in FP	in FP
	a^+, a^*	in FP	#P [V79]
	RE, RE($\#, !, \neg, \bullet$)	in FP	#P
FINITENESS	RE, RE($\#, !, \neg, \bullet$)	in P	—

(concatenation). The SPARQL-negation operator "!" is defined in Section 3.3. *Det-RE* stands for the class of deterministic SPARQL expressions that we defined in Section 4.

The table presents complexity results under combined query evaluation complexity. However, for simple walk semantics, all the NP-hardness or #P-hardness results hold under data complexity as well (see Table II), except for the result on CHARE($a, a?$). Indeed, if the CHARE($a, a?$) expression is fixed, we can translate it to a DFA^w and perform the algorithm for COUNTING under regular path semantics. (For this fragment, simple walk semantics equals regular path semantics.) For regular path semantics, all #P-hardness results become tractable under data complexity: when the query is fixed, we can always translate it to a DFA^w and perform the algorithm for DFA^{w}s. When considering data complexity, the difference between regular path semantics and simple walk semantics is therefore rather severe.

Influence in SPARQL 1.1. The NP-complete and #P-complete data complexities make the semantics of W3C property paths in [Harris and Seaborne 2012] highly problematic from a computational complexity perspective, especially on a Web scale. There are two orthogonal requirements that render the evaluation of simple queries of the form `SELECT ?x, ?y WHERE {?x r ?y}` computationally difficult:

Simple Walk Requirement.
Subexpressions of the form r^* and r^+ should be matched to *simple walks*.
Path Counting Requirement.
The number of paths from x to y that match r need to be counted.

Two studies [Arenas et al. 2012; Losemann and Martens 2012] propose that the W3C should use a semantics for property paths without the simple walk requirement or path counting requirement. These proposals were seriously taken into consideration by the W3C, who made the following changes to the definition of property paths:

(1) The ambiguity between the definitions of ZeroOrMorePath/OneOrMorePath in Sections 18.4 and 18.5 in [Harris and Seaborne 2012] has been removed and the simple walk requirement has been dropped.
(2) The path counting requirement has been dropped for subexpressions of the form r^* and r^+.
(3) Subexpressions of the form $r^{k,\ell}$ with $k, \ell \in \mathbb{N}$ are no longer part of the property path syntax.

We feel that change (1) is very welcome. Change (2) will undeniably make the evaluation of SPARQL property paths more efficient. However, we feel that, even after this change, the semantics of property paths may be rather counter-intuitive, since the change does not completely remove multiset semantics for property paths. Instead, change (2) means that, for a given regular expression r, its subexpressions of the form s^* and s^+ are evaluated under set semantics and the others under multiset semantics. As such, it may become difficult to understand the number of occurrences of tuples in the output of a query. For example, the combination of set semantics and multiset semantics has as a result that a^+ is not equivalent to aa^*, since the former always returns tuples with multiplicity one (set semantics) and the latter can return tuples with higher multiplicity due to the partial multiset semantics.

On the other hand, if one would use a set semantics for property paths (and, as such, completely remove the path counting requirement), then the evaluation of property paths becomes rather efficient from a theoretical perspective. Indeed, Theorem 3.4 exhibits a polynomial time upper bound in combined complexity which is further refined in Corollary 3.6.

If set semantics would be used for property paths, then it could even be imaginable to extend property paths to something more expressive such as nested regular expressions or XPath dialects. We studied the extension to nested regular expressions (with wildcards, limited negation, and numerical occurrence indicators) in Section 6 and it turns out that, under set semantics, these expressions can be evaluated equally efficient as RE($\#, !, \bullet$), i.e., SPARQL regular expressions. This result can be strengthened even more by also allowing XPath-like operators, as shown in [Libkin et al. 2013].

Finally, we feel that change (3) is a pity. We showed that property paths with this feature can in principle be evaluated efficiently (Theorem 3.4, Corollary 3.6), even though they can be exponentially more succinct than standard regular expressions. This seems to be a win-win situation and we therefore feel that such expressions should have their place in SPARQL property paths. Expressions with numerical occurrence indicators also seem to have their use: for example, in the regexlib regular expression library,[10] more than half of the expressions numerical occurrence indicators.

APPENDIX

A. PROOF FOR THEOREM 4.8.

To prove the theorem, we need the notion of a *match*.

Definition A.1. A *match* m between a path $p = v_0[a_1]v_1 \cdots [a_n]v_n$ and a regular expression r is a total mapping from pairs (i, j) of nodes of p (with $0 \leq i \leq j \leq n$) to a non-empty set of subexpression of r. This mapping has to be consistent with the semantics of regular expressions, that is,

(1) if $\varepsilon \in m(i, j)$, then $i = j$;
(2) if $a \in m(i, j)$ for $a \in \Delta$, then $j = i + 1$ and $a_j = a$;
(3) if $r_1? \in m(i, j)$, then $r_1 \in m(i, j)$ or $i = j$;
(4) if $(r_1 + r_2) \in m(i, j)$, then $r_1 \in m(i, j)$ or $r_2 \in m(i, j)$;
(5) if $r_1 r_2 \in m(i, j)$, then there is a k such that $r_1 \in m(i, k)$ and $r_2 \in m(k, j)$;
(6) if $r_1^* \in m(i, j)$, then $i = j$ or there exist numbers $k_1, \ldots, k_t$ such that $k_1 = i$, $k_t = j$, and $r_1 \in m(k_\ell, k_{\ell+1})$ for all $\ell = 1, \ldots, t-1$; and
(7) if $r_1^+ \in m(i, j)$, then there exist numbers $k_1, \ldots, k_t$ such that $k_1 = i$, $k_t = j$, and $r_1 \in m(k_\ell, k_{\ell+1})$ for all $\ell = 1, \ldots, t-1$.

Furthermore, m has to be minimal with these properties. That is, if m' fulfills (1)–(7) and $m'(i, j) \subseteq m(i, j)$ for each i, j, then $m' = m$. The minimality requirement on m is for convenience in proofs and ensures that, in case of $r_1 + r_2$, mapping m either matches onto r_1 or on r_2, but not both. We say that m *matches* a subpath $v_i[a_{i+1}] \cdots [a_j]v_j$ of p onto a subexpression r' of r when $r' \in m(i, j)$. We sometimes also leave the matching m implicit and simply say that $v_i[a_{i+1}] \cdots [a_j]v_j$ matches r'.

THEOREM 4.8: COUNTING *is #P-complete for all of the following classes:*

(1) CHARE(a, a^*)
(2) CHARE(a, $a?$)
(3) CHARE(a, w^+)
(4) CHARE(a, $(+a^+)$)
(5) CHARE(a, $(+a)^+$)
(6) CHARE(a^+, $(+a)$)

Moreover, $\#P$-hardness already holds if the graph G is acyclic.

PROOF. The upper bound for all cases is immediate from Theorem 4.7. We prove the lower bounds by reductions from #DNF. The technique is inspired by a proof in [Martens et al. 2009], where it is shown that language inclusion for various classes of CHAREs is coNP-hard. We first perform a meta-reduction for the cases (1)–(3) and then instantiate it with slightly different subgraphs and expressions to deal with the different cases. For the cases (4)–(6) we follow a similar approach.

Let $\Phi = C_1 \vee \cdots \vee C_k$ be a propositional formula in 3DNF using variables $\{x_1, \ldots, x_n\}$. We encode truth assignments for Φ by paths in the graph. In particular, we construct a graph $G = (V, E, x, y)$, an expression r, and a number max such that each path of length at most max in G from x to y that matches r corresponds to a unique satisfying truth assignment for Φ and vice versa. Formally, we will prove that

[10]RegExLib.com (Regular Expression Library)

The number of paths of length at most max in G from x to y that match r is equal to the number of truth assignments that satisfy Φ. (*)

The graph G has the structure as depicted in Figure 6 (and which can be written as "B^kAB^k"), where

— B is a path labeled $\#\alpha\$\alpha\$\cdots\$\alpha\#$ (containing n copies of α) and
— A is a subgraph with a dedicated source node x_A and target node y_A, i.e., all paths from x to y will enter A through the node x_A and leave A through y_A.

Here, the subgraph α of B is itself also a path which we will instantiate differently in each of the cases (1)–(3).

Each path from x_A to y_A in A corresponds to exactly one truth assignment for the variables $\{x_1, \ldots, x_n\}$. That is, for each truth assignment ψ, there is a path p_ψ in A from x_A to y_A and, for each path p in A from x_A to y_A, there is a corresponding truth assignment ψ_p. More precisely, consider the structure of A as depicted in Figure 6, with n occurrences of p^{true} and p^{false}. Here, p^{true} and p^{false} are paths in A whose labels depend on the CHARE fragment we are considering. The paths p^{true} and p^{false} do not use any of the special labels in $\{\$, \#\}$. A path through A from x_A to y_A therefore has n choices of going through p^{true} or p^{false}. If the i-th choice goes through p^{true}, this corresponds to a truth assignment that sets x_i to "true". Similarly for p^{false}.

The expressions r will have the form

$$r = NF(C_1)\cdots F(C_k)N,$$

such that

— each path labeled $\text{lab}(B)^i$, for $i = 1, \ldots, k$, matches N and
— each $F(C_i)$ is a subexpression associated with clause C_i with $i \in \{1, \ldots, k\}$.

Furthermore, the path B can be matched by each $F(C_i)$ and, for each clause C_i and every path p in A, it will hold that p matches $F(C_i)$ if and only if the truth assignment ψ_p associated with p satisfies C_i. We use subexpressions r^{true}, r^{false}, and r^{all} in the definition of the $F(C_i)$. These expressions intuitively correspond to a variable occurring positively, negatively, or not at all in clause C_i. Again, these subexpressions will be instantiated differently in the different fragments. Formally, for each clause C, we define $F(C)$ as

$$\#e_1\$\cdots\$e_n\#$$

where, for each $j = 1, \ldots, n$,

$$e_j := \begin{cases} r^{\text{false}}, & \text{if } x_j \text{ occurs negated in } C, \\ r^{\text{true}}, & \text{if } x_j \text{ occurs positively in } C, \text{ and} \\ r^{\text{all}}, & \text{otherwise.} \end{cases}$$

The subexpression N will be defined differently for each of the fragments.

We prove some properties of the meta-reduction and some assumptions that we will need to make for the properties to hold. Later, in the proofs of the cases (1)–(3), we will prove that the assumptions are met. Our assumptions on the paths and expressions are the following:

$$\text{lab}(\alpha) \in L(r^{\text{false}}) \cap L(r^{\text{true}}) \tag{P1}$$
$$\text{lab}(p^{\text{true}}) \in L(r^{\text{true}}) - L(r^{\text{false}}) \tag{P2}$$
$$\text{lab}(p^{\text{false}}) \in L(r^{\text{false}}) - L(r^{\text{true}}) \tag{P3}$$
$$\{\text{lab}(p^{\text{true}}), \text{lab}(p^{\text{false}}), \text{lab}(\alpha)\} \subseteq L(r^{\text{all}}) \tag{P4}$$

Notice that conditions (P2) and (P3) imply that $\text{lab}(p^{\text{true}}) \neq \text{lab}(p^{\text{false}})$. Due to the structure of G and since p^{true} and p^{false} do not use the symbols \$ or $\#$, we therefore have that paths p_1 and p_2 from x to y are different if and only $\text{lab}(p_1) \neq \text{lab}(p_2)$. Furthermore, since B is a path, the only parts where paths from x to y can differ is in the subgraph A. Therefore, to simplify notation, we identify a path p from x to y in G with the label of the subpath p' of p that goes from x_A to y_A. Now, we associate a truth assighment to each such path:

— For a path labeled $w = \#w_1\$ \cdots \$w_n\#$ in A, let the truth assignment V_w be defined as follows:

$$V_w(x_j) := \begin{cases} \text{true}, & \text{if } w_j = \text{lab}(p^{\text{true}}), \\ \text{false}, & \text{otherwise}. \end{cases}$$

— For a truth assignment V, let $w_V = \#w_1\$ \cdots \$w_n\#$, where, for each $j = 1, \ldots, n$,

$$w_j = \begin{cases} \text{lab}(p^{\text{true}}) & \text{if } V(x_j) = \text{true}, \\ \text{lab}(p^{\text{false}}) & \text{otherwise}. \end{cases}$$

Due to the structure of G, it is immediate that the encoding between truth assignments and paths is unique.

CLAIM A.2. *Let C be a clause from Φ. If $V_w \models C$ then $w \in L(F(C))$. If $w_V \in L(F(C))$ then $V \models C$.*

PROOF OF CLAIM A.2. Let $w = \#w_1\$ \cdots \$w_n\#$ be the label of a path from x_A to y_A in A such that $V_w \models C$ and let $F(C) = \#e_1\$ \cdots \$e_n\#$ be as defined above. We show that $w \in L(F(C))$. To this end, let $j \leq n$. There are three cases to consider:

1. If x_j does not occur in C then $e_j = r^{\text{all}}$. Hence, as $w_j \in \{\text{lab}(p^{\text{true}}), \text{lab}(p^{\text{false}})\}$ and by condition (P4), we have that $w_j \in L(e_j)$.
2. If x_j occurs positively in C, then $e_j = r^{\text{true}}$. As $V_w \models C$, we have $V_w(x_j) = \text{true}$ and, by definition of V_w, we get $w_j = \text{lab}(p^{\text{true}})$. By condition (P2), we have that $w_j \in L(r^{\text{true}}) = L(e_j)$.
3. If x_j occurs negatively in C, then $e_j = r^{\text{false}}$. As $V_w \models C$, $V_w(x_j) = \text{false}$. Thus, $w_j = \text{lab}(p^{\text{false}})$ by definition of V_w and since $w_j \in \{\text{lab}(p^{\text{false}}), \text{lab}(p^{\text{true}})\}$. Therefore, by condition (P3), $w_j \in L(r^{\text{false}}) = L(e_j)$.

Therefore, for each $j = 1, \ldots, n, w_j \in L(e_j)$ and thus $w \in L(F(C))$.

We show the other statement by contraposition. Thereto, let V be a truth assignment such that it does not make clause C true. We show that $w_V \notin L(F(C))$. There are two cases:

1. Suppose there exists an x_j which occurs positively in C and $V(x_j)$ is false. By definition, the e_j component of $F(C)$ is r^{true} and, by definition of w_V, the w_j component of w_V is $\text{lab}(p^{\text{false}})$. By condition (P3), $w_j \notin L(r^{\text{true}})$. Hence, $w_V \notin L(F(C))$.

2. Otherwise, there exists an x_j which occurs negatively in C and $V(x_j)$ is true. By definition, the e_j component of $F(C)$ is r^{false} and, by definition of w_V, the w_j component of w_V is lab(p^{true}). By condition (P2), $w_j \notin L(r^{\text{false}})$. Hence, $w_V \notin L(F(C))$.

This concludes the proof of Claim A.2. □

We now fill in some details to complete the reductions for the cases (1)–(3).
(1) We instantiate the paths and subexpressions as follows:

— lab$(\alpha) = a$
— lab$(p^{\text{true}}) = ab$
— lab$(p^{\text{false}}) = ba$
— N is k concatenations of $(\#^* a^* \$^* \cdots \$^* a^* \#^*)$
— $r^{\text{true}} = a^* b^*$
— $r^{\text{false}} = b^* a^*$
— $r^{\text{all}} = a^* b^* a^*$

Notice that, in this case, r is a CHARE(a, a^*).
(2) The reduction is analogous to the one in case (1), with the differences that

— lab$(\alpha) = aa$
— lab$(p^{\text{true}}) = aaa$
— lab$(p^{\text{false}}) = a$
— N is k concatenations of $(\#?a?a?\$? \cdots \$?a?a?\#?)$
— $r^{\text{true}} = aaa?$
— $r^{\text{false}} = aa?$
— $r^{\text{all}} = aa?a?$

Notice that, in this case, r is a CHARE$(a, a?)$.
(3) The reduction is analogous to the one in case (1), with the differences that

— lab$(\alpha) = aaaa$
— lab$(p^{\text{true}}) = aaa$
— lab$(p^{\text{false}}) = aa$
— $N = (\#aaaa\$ \cdots \$aaaa\#)^+$
— $r^{\text{true}} = a^+(aa)^+$
— $r^{\text{false}} = (aa)^+$
— $r^{\text{all}} = a^+$

Notice that, in this case, r is a CHARE(a, w^+).
It is straightforward to verify that the conditions (P1)–(P4) are fulfilled for each of the fragments. Note that the expressions use the fixed alphabet $\{a, b, \$, \#\}$.
Before we prove (*) for fragments (1)–(3), we need to prove some properties that will be helpful.

CLAIM A.3. *Let w be a label of a path from x_A to y_A in A. Then all of the following hold:*

(a) lab$(B)^i \in L(N)$ for every $i = 1, \ldots, k$.
(b) lab$(B) \in L(F(C_i))$ for every $i = 1, \ldots, k$.
(c) If p is a path in G from x to y with lab(p) = lab$(B)^k \cdot w \cdot$ lab$(B)^k \in L(r)$, then w matches some C_i.

PROOF OF CLAIM A.3. Part (a) can be easily checked for all fragments (1)–(3) and (b) follows immediately from conditions (P1), (P4), and the definition of all expressions $F(C)$.

We now prove (c). In the following, we abbreviate $F(C_i)$ by F_i. Let $u = \text{lab}(B)$. Suppose that $u^k w u^k \in L(r)$ and is the label of some path from x to y in G. We need to show that w matches some F_i. Observe that the words u, w, and every word in every $L(F_i)$ is of the form $\#y\#$ where y is a non-empty word over the alphabet $\{a, b, \$\}$. Also, $L(N)$ only contains words of the form $\#y_1\#\#y_2\#\cdots\#y_\ell\#$, where $y_1, \ldots, y_\ell$ are non-empty words over $\{a, \$\}$ (and, possibly, subsequences thereof). Hence, as $u^k w u^k \in L(r)$ and as none of the words $y, y_1, \ldots, y_\ell$ contain the symbol "#", w either matches some F_i or w matches a sub-expression of N.

We now distinguish between fragments (1–2) and fragment (3), because we need to talk about how the word $u^k w u^k$ matches r. Therefore, we use the notion of a *match* (Definition A.1). For a path p_0 and regular expression r, we also say that the word lab(p_0) *matches* r if p_0 matches r.

Let m be a match between p and r. Notice that $\text{lab}(p) = u^k w u^k$.

— In fragments (1–2) we have that $\ell \leq k$. Towards a contradiction, assume that m matches a superword of $u^k w$ to the left occurrence of N in r. Note that $u^k w$ is a word of the form $\#y'_1\#\#y'_2\#\cdots\#y'_{k+1}\#$, where $y'_1, \ldots, y'_{k+1}$ are words over $\{a, b, \$\}$. But, since $\ell \leq k$, no superword of $u^k w$ can match N, which is a contradiction. Analogously, no superword of wu^k matches the right occurrence of the expression N in r. So, m must match w onto some F_i.
— In fragment (3), we have that $\ell \geq 1$. Again, towards a contradiction, assume that m matches a superword of $u^k w$ onto the left occurrence of the expression N in r. Observe that every word that matches $F_1 \cdots F_k N$ is of the form $\#y''_1\#\#y''_2\#\cdots\#y''_{\ell'}\#$, where $\ell' > k$. As u^k is not of this form, m cannot match u^k onto $F_1 \cdots F_k N$, which is a contradiction. Analogously, m cannot match a superword of wu^k onto the right occurrence of the expression N in r. So, m must match w onto some F_i.

This concludes the proof of Claim A.3. □

We are now ready to prove $(*)$ for fragments (1)–(3).

CLAIM A.4. *For each of the fragments (1)–(3), the number of paths of length at most max in G from x to y that match r is equal to the number of truth assignments that satisfy Φ.*

PROOF. We show that every path of length at most max in G from x to y that matches r corresponds to exactly one truth assignment that satisfy Φ and vice versa.

Formally, we define a bijection φ between paths of length at most max in G from x to y and truth assignments for Φ. Let p be an arbitrary path from x to y in G. Since max is the number of nodes in G and every path in G can visit every node at most once (G is acyclic), every such path p has length at most max. Let $\text{lab}(p) = \text{lab}(B)^k \cdot w \cdot \text{lab}(B)^k$. We define $\varphi(p) := V_w$, where V_w is the truth assignment we defined above. Notice that there are exactly 2^n paths from x to y and the same amount of truth assignments for Φ. From the definition of V_w and the structure if G, it is immediate that φ is a bijection. Notice that φ^{-1} maps each truth assignment V onto the path labeled w_V, where w_V is as defined above.

It now follows from Claims A.3 and A.2 that φ is a bijection between the paths of length at most max from x to y that match r and the truth assignments that satisfy Φ as well. □

Thus the reduction used in the proof of Theorem 4.8 is correct for fragments (1)–(3).

For the last three cases we have to slightly modify the reduction. The main difference of these last fragments with the first ones, is that we will not use a fixed size alphabet. Instead, we use the symbols b_j and c_j, for $j = 1, \ldots, n$. Instead of the paths

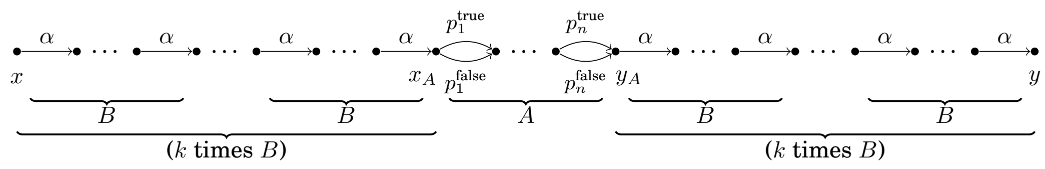


Fig. 9. The graph G from the proof of Theorem 4.8 for the cases (4)–(6).

p^{true} and p^{false} we will have p_j^{true} and p_j^{false} and instead of the regular expressions r^{true}, r^{false}, and r^{all}, we will now have expressions r_j^{true}, r_j^{false}, and r_j^{all}, for every $j = 1, \dots, n$. We will require that these expressions fulfill the properties (P1)–(P4) for each j, where $p^{\text{true}}, p^{\text{false}}, r^{\text{true}}, r^{\text{false}}$, and r^{all} are replaced by p_j^{true}, p_j^{false}, r_j^{true}, r_j^{false}, and r_j^{all}, respectively.

Furthermore, we need different subgraphs A, B and different expressions $F(C)$ for clauses C:

— A is of the form as described in Figure 9 containing the paths, $p_1^{\text{true}}, p_1^{\text{false}}, \dots, p_n^{\text{true}}, p_n^{\text{false}}$;
— B is a path, consisting of n concatenations of the subpath α; and
— $F(C)$ is defined as

$$e_1 \cdots e_n,$$

where for each $j = 1, \dots, n$,

$$e_j := \begin{cases} r_j^{\text{false}}, & \text{if } x_j \text{ occurs negated in } C, \\ r_j^{\text{true}}, & \text{if } x_j \text{ occurs positively in } C, \text{ and} \\ r_j^{\text{all}}, & \text{otherwise.} \end{cases}$$

Furthermore The new structure of graph G is also shown in Figure 9.

(4) For the fourth case the reduction is instantiated by the following expressions.

— $\text{lab}(\alpha) = a$
— $\text{lab}(p_j^{\text{true}}) = b_j$
— $\text{lab}(p_j^{\text{false}}) = c_j$
— $N = a^+$
— $r_j^{\text{true}} = (a^+ + b_j^+)$
— $r_j^{\text{false}} = (a^+ + c_j^+)$
— $r_j^{\text{all}} = (a^+ + b_j^+ + c_j^+)$

Notice that, in this case, r is a CHARE($a, (+a^+)$).

(5) The reduction is analogous to the one in case (4), with the differences that

— $\text{lab}(\alpha) = a$
— $\text{lab}(p_j^{\text{true}}) = b_j$
— $\text{lab}(p_j^{\text{false}}) = c_j$
— $N = a^+$
— $r_j^{\text{true}} = (a + b_j)^+$
— $r_j^{\text{false}} = (a + c_j)^+$
— $r_j^{\text{all}} = (a + b_j + c_j)^+$

Notice that, in this case, r is a CHARE($a, (+a)^+$).

(6) The reduction is analogous to the one in case (4), with the differences that

— $\text{lab}(\alpha) = a$
— $\text{lab}(p_j^{\text{true}}) = b_j$
— $\text{lab}(p_j^{\text{false}}) = c_j$
— $N = a^+$
— $r_j^{\text{true}} = (a + b_j)$
— $r_j^{\text{false}} = (a + c_j)$
— $r_j^{\text{all}} = (a + b_j + c_j)$

Notice that, in this case, r is a CHARE($(+a), a^+$).

We now prove that the reduction is correct for the fragments (4)–(6). It is straightforward to verify that the conditions (P1)–(P4) are fulfilled for each of the fragments.

The association between words and truth assignments is now defined as follows.

— For each path labeled $w = w_1 \cdots w_n$ in A where, for every $j = 1, \ldots, n$, $w_j \in \{p_j^{\text{true}}, p_j^{\text{false}}\}$, let V_w be defined as follows:

$$V_w(x_j) := \begin{cases} \text{true}, & \text{if } w_j \in L(r_j^{\text{true}}), \\ \text{false}, & \text{otherwise.} \end{cases}$$

— For a truth assignment V, let $w_V = w_1 \cdots w_n$, where, for each $j = 1, \ldots, n$,

$$w_j = \begin{cases} p_j^{\text{true}} & \text{if } V(x_j) = \text{true, and} \\ p_j^{\text{false}} & \text{otherwise.} \end{cases}$$

In order to prove $(*)$ for fragments (4)–(6), we have to show that Claims A.2 and A.3 hold. For Claim A.2 this can be done similarly as before, and for Claim A.3 (a) and (b) it is again easy to see. We complete the proof of Theorem 4.8 by showing Claim A.3 (c).

To this end, again denote lab(B) by u. Assume that $u^k w u^k \in L(r)$ and is the label of a path from x to y in G. We need to show that w matches some F_i. Let m be a match between $u^k w u^k$ and r. For every $j = 1, \ldots, n$, let Σ_j denote the set $\{b_j, c_j\}$. Observe that the word w is of the form $y_1 \cdots y_n$, where, for every $j = 1, \ldots, n$, y_j is a word in Σ_j^+. Moreover, no words in $L(N)$ contain symbols from Σ_j for any $j = 1, \ldots, n$. Hence, m cannot match any symbol of the word w onto N. Consequently, m matches the entire word w onto a subexpression of $F_1 \cdots F_k$ in r.

Further, observe that every word in every F_i, $i = 1, \ldots, k$, is of the form $y'_1 \ldots y'_n$, where each y'_j is a word in $(\Sigma_j \cup \{a\})^+$. As m can only match symbols in Σ_j onto subexpressions with symbols in Σ_j, m matches w onto some F_i.

Now that we know that Claim A.2 and A.3 hold for the fragments (4)–(6), the proof of $(*)$ is analogous to the proof of Claim A.4.

Notice that, in all our cases, we have constructed an acyclic graph G. This means that, in all cases, #P-hardness even holds if G is acyclic.

This concludes the proof of Theorem 4.8. □

ACKNOWLEDGMENTS

Many people contributed to this work through inspiring discussions. We thank Marcelo Arenas for sending us a draft of [Arenas et al. 2012] and for his comments on a draft of [Losemann and Martens 2012]; Thomas Schwentick for pointing out that emptiness testing of star-free regular expressions with negation is non-elementary (i.e, Theorem 3.9); Pekka Kilpeläinen for giving us the 1979 reference for dynamic programming for evaluating regular expressions on words; and Christian Cöster for many useful remarks. We thank the anonymous reviewers of PODS 2012 and ACM Transactions on Database Systems for their insightful comments that helped to improve the presentation of the article.

REFERENCES

S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. L. Wiener. 1997. The Lorel Query Language for Semistructured Data. *International Journal on Digital Libraries* 1, 1 (1997), 68–88.

S. Abiteboul and V. Vianu. 1999. Regular Path Queries with Constraints. *Journal of Computer and System Sciences (JCSS)* 58, 3 (1999), 428–452.

N. Alechina and N. Immerman. 2000. Reachability logic: An efficient fragment of transitive closure logic. *Logic Journal of the IGPL* 8, 3 (2000), 325–337.

F. Alkhateeb, J.-F. Baget, and J. Euzenat. 2009. Extending SPARQL with regular expression patterns (for querying RDF). *Journal of Web Semantics* 7, 2 (2009), 57–73.

Carme Álvarez and Birgit Jenner. 1993. A very hard log-space counting class. *Theoretical Computer Science* 107, 1 (1993), 3–30. Issue 1.

M. Arenas, S. Conca, and J. Pérez. 2012. Counting beyond a Yottabyte, or how SPARQL 1.1 property paths will prevent adoption of the standard. In *International World Wide Web Conference (WWW)*. ACM, New York, NY, USA, 629–638.

M. Arenas and J. Pérez. 2011. Querying semantic web data with SPARQL. In *Symposium on Principles of Database Systems (PODS)*. ACM, New York, NY, USA, 305–316.

G. Bagan, A. Bonifati, and B. Groz. 2013. A Trichotomy for Regular Simple Path Queries on Graphs. In *Symposium on Principles of Database Systems (PODS)*. ACM, New York, NY, USA.

Claude Berge. 1973. *Graphs and Hypergraphs*. North-Holland Publishing Company, New York, NY, USA.

G. J. Bex, F. Neven, T. Schwentick, and S. Vansummeren. 2010. Inference of Concise Regular Expressions and DTDs. *ACM Transactions on Database Systems (TODS)* 35, 2 (2010), 11:1–11:47.

R. Book, S. Even, S. Greibach, and G. Ott. 1971. Ambiguity in Graphs and Expressions. *IEEE Trans. Comput.* 20 (February 1971), 149–153. Issue 2.

T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau. 2008. *Extensible Markup Language XML 1.0 (Fifth Edition)* (5 ed.). Technical Report. World Wide Web Consortium (W3C). W3C Recommendation, http://www.w3.org/TR/2008/REC-xml-20081126/.

A. Brüggemann-Klein and D. Wood. 1998. One-Unambiguous Regular Languages. *Information and Computation* 142, 2 (1998), 182–206.

P. Buneman, S. B. Davidson, G. G. Hillebrand, and D. Suciu. 1996. A Query Language and Optimization Techniques for Unstructured Data. In *ACM SIGMOD International Conference on Management of Data*. ACM, New York, NY, USA, 505–516.

D. Calvanese, G. De Giacomo, M. Lenzerini, and M.Y. Vardi. 2002. Rewriting of Regular Expressions and Regular Path Queries. *Journal of Computer and System Sciences (JCSS)* 64, 3 (2002), 443–465.

D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Y. Vardi. 2000a. Containment of Conjunctive Regular Path Queries with Inverse. In *Principles of Knowledge Representation and Reasoning (KR)*. Morgan Kaufmann, San Fransisco, CA, USA, 176–185.

D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Y. Vardi. 2000b. View-Based Query Processing for Regular Path Queries with Inverse. In *Symposium on Principles of Database Systems (PODS)*. ACM, New York, NY, USA, 58–66.

R. Cleaveland and B. Steffen. 1993. A Linear-time model-checking algorithm for the alternation-free modal mu-calculus. *Formal Methods in System Design* 2, 2 (1993), 121–147.

D. Colazzo, G. Ghelli, and C. Sartiani. 2009a. Efficient asymmetric inclusion between regular expression types. In *International Conference Database Theory (ICDT)*. ACM, New York, NY, USA, 174–182.

D. Colazzo, G. Ghelli, and C. Sartiani. 2009b. Efficient inclusion for a class of XML types with interleaving and counting. *Information Systems* 34, 7 (2009), 643–656.

M. P. Consens and A. O. Mendelzon. 1990. GraphLog: a Visual Formalism for Real Life Recursion. In *Symposium on Principles of Database Systems (PODS)*. ACM, New York, NY, USA, 404–416.

I. F. Cruz, A. O. Mendelzon, and P. T. Wood. 1987. A Graphical Query Language Supporting Recursion. In *ACM SIGMOD International Conference on Management of Data*. ACM, New York, NY, USA, 323–330.

A. Deutsch and V. Tannen. 2001. Optimization Properties for Classes of Conjunctive Regular Path Queries. In *International Workshop on Database Programming Languages (DBPL)*. Springer-Verlag, London, UK, UK, 21–39.

D. Fallside and P. Walmsley. 2004. *XML Schema Part 0: Primer (Second Edition)*. Technical Report. World Wide Web Consortium. http://www.w3.org/TR/2004/REC-xmlschema-0-20041028/.

M. F. Fernández, D. Florescu, A. Y. Levy, and D. Suciu. 2000. Declarative Specification of Web Sites with STRUDEL. *The VLDB Journal* 9, 1 (2000), 38–55.

D. Florescu, A. Y. Levy, and D. Suciu. 1998. Query Containment for Conjunctive Queries with Regular Expressions. In *Symposium on Principles of Database Systems (PODS)*. ACM, New York, NY, USA, 139–148.

S. Gao, C. M. Sperberg-McQueen, H.S. Thompson, N. Mendelsohn, D. Beech, and M. Maloney. 2009. *W3C XML Schema Definition Language (XSD) 1.1 Part 1: Structures*. Technical Report. World Wide Web Consortium. W3C Recommendation, http://www.w3.org/TR/2009/CR-xmlschema11-1-20090430/.

W. Gelade, M. Gyssens, and W. Martens. 2012. Regular Expressions with Counting: Weak versus Strong Determinism. *SIAM J. Comput.* 41, 1 (2012), 160–190.

W. Gelade, W. Martens, and F. Neven. 2009. Optimizing Schema Languages for XML: Numerical Constraints and Interleaving. *SIAM J. Comput.* 38, 5 (2009), 2021–2043.

V. M. Glushkov. 1961. The abstract theory of automata. *Russian Mathematical Surveys* 16, 5(101) (1961), 1–53.

S. Harris and A. Seaborne. 2010. *SPARQL 1.1 Query Language*. Technical Report. World Wide Web Consortium (W3C). `http://www.w3.org/TR/2010/WD-sparql11-query-20100601/`

S. Harris and A. Seaborne. 2012. *SPARQL 1.1 Query Language*. Technical Report. World Wide Web Consortium (W3C). `http://www.w3.org/TR/2012/WD-sparql11-query-20120105/`

J.E. Hopcroft and J.D. Ullman. 1979. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, Boston, MA, USA.

Sampath Kannan, Z. Sweedyk, and Stephen R. Mahaney. 1995. Counting and Random Generation of Strings in Regular Languages. In *Symposium on Discrete Algorithms (SODA)*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 551–557.

P. Kilpeläinen and R. Tuhkanen. 2003. Regular Expressions with Numerical Occurrence Indicators — preliminary results.. In *Symposium on Programming Languages and Software Tools (SPLST)*. University of Kuopio, Department of Computer Science, Kuopio, Finland, 163–173.

P. Kilpeläinen and R. Tuhkanen. 2007. One-unambiguity of regular expressions with numeric occurrence indicators. *Information and Computation* 205, 6 (2007), 890–916.

S. C. Kleene. 1956. *Automata Studies*. Princeton Univ. Press, Princeton, NJ, USA, Chapter Representations of events in nerve sets and finite automata, 3–42.

L. Libkin, W. Martens, and D. Vrgoč. 2013. Querying graph databases with XPath. In *International Conference on Database Theory (ICDT)*. ACM, New York, NY, USA, 129–140.

L. Libkin and D. Vrgoč. 2012. Regular Path Queries on Graphs with Data. In *International Conference on Database Theory (ICDT)*. ACM, New York, NY, USA, 74–85.

Yanhong A. Liu and Fuxiang Yu. 2002. Solving Regular Path Queries. In *Mathematics of Program Construction (MPC)*. Springer-Verlag, London, UK, UK, 195–208.

Katja Losemann and Wim Martens. 2012. The complexity of evaluating path expressions in SPARQL. In *Symposium on Principles of Database Systems (PODS)*. ACM, New York, NY, USA, 101–112.

W. Martens, F. Neven, and T. Schwentick. 2004. Complexity of Decision Problems for Simple Regular Expressions. In *Mathematical Foundations of Computer Science (MFCS)*. Springer, Heidelberg, Germany, 889–900.

W. Martens, F. Neven, and T. Schwentick. 2009. Complexity of Decision Problems for XML Schemas and Chain Regular Expressions. *SIAM J. Comput.* 39, 4 (2009), 1486–1530.

A. O. Mendelzon and P. T. Wood. 1995. Finding Regular Simple Paths in Graph Databases. *SIAM J. Comput.* 24, 6 (1995), 1235–1258.

J. Pérez, M. Arenas, and C. Gutierrez. 2009. Semantics and complexity of SPARQL. *ACM Transactions on Database Systems (TODS)* 34, 3 (2009), 16:1–16:45.

J. Pérez, M. Arenas, and C. Gutierrez. 2010. nSPARQL: A navigational language for RDF. *Journal of Web Semantics* 8, 4 (2010), 255–270.

Raghu Ramakrishnan and Johannes Gehrke. 2003. *Database management systems (3. ed.)*. McGraw-Hill, New York, NY, USA.

M. Schmidt, M. Meier, and G. Lausen. 2010. Foundations of SPARQL query optimization. In *International Conference on Database Theory (ICDT)*. ACM, New York, NY, USA, 4–33.

L. Stockmeyer. 1974. *The complexity of decision problems in automata theory and logic*. Ph.D. Dissertation. Massachusetts Institute of Technology.

L. G. Valiant. 1979. The Complexity of Enumeration and Reliability Problems. *SIAM J. Comput.* 8, 3 (1979), 410–421.

M. Yannakakis. 1990. Graph-Theoretic Methods in Database Theory. In *Symposium on Principles of Database Systems (PODS)*. ACM, New York, NY, USA, 230–242.

Received October 2012; revised MM/YYYY; accepted MM/YYYY