The Complexity of Evaluating Path Expressions in SPARQL

Katja Losemann*
Universität Bayreuth

Wim Martens
Universität Bayreuth

ABSTRACT

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 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

F.2 [**Analysis of Algorithms and Problem Complexity**]: General; H.2.3 [**Database Management**]: Languages—*query languages*

Keywords

Graph data, regular expression, query evaluation

*Supported by grant number MA 4938/2–1 from the Deutsche Forschungsgemeinschaft (Emmy Noether Nachwuchsgruppe).

PODS'12, May 21–23, 2012, Scottsdale, Arizona, USA.

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 by the introduction of *property paths* [25]. Property paths closely correspond to regular expressions and are a crucial tool in SPARQL if one wants to perform non-trivial navigation through RDF data. In the current working draft, 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 [6]. 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

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

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 an expression with numerical occurrence indicators r, we can test whether there is a path from x to y that matches r in polynomial time combined complexity.

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 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.

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. In contrast, under simple walk semantics, EVALUATION is already NP-complete for the regular expression $(aa)^*$. (So, it is NP-complete under data complexity.) We also identify a fragment of expressions for which EVALUATION under simple walk semantics is in PTIME 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 *deterministic*, then COUNTING can be solved in polynomial time. However, even for expressions with a very limited amount of non-determinism, COUNTING becomes #P-complete. Under simple walk semantics, 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 2. 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 PTIME in Theorem 3.3.

Since the W3C's specification for SPARQL 1.1 is still under development, we want to send a strong message to the W3C that informs them of the computational complexity repercussions of their 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. We propose some concrete ideas in Section 6.

Related Work and Further Literature.

This paper studies evaluation problems of regular expressions on 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 [1, 10, 16, 17, 19, 41]. Various problems for regular path queries have been investigated in the database community, such as optimization [2], query rewriting and query answering using views [13, 12], and containment [11, 18, 20]. Recently, there has been a renewed interest in path expressions on graphs, for example, on expressions with data value comparisons [31].

Regular path queries have also been studied in the context of program analysis. For example, evaluation of path queries on graphs has been studied by Liu et al. [32]. However, their setting is different 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 [15, 14, 22, 23, 29, 28] since they are a part of the W3C XML Schema Language [21]. One of our PTIME upper bounds builds directly on Kilpeläinen and Tuhkonen's algorithm for membership testing of a regular expression with numerical occurrence indicators [28].

To the best of our knowledge, the present paper 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 [37, 3, 5]. Most closely related to us is Arenas et al. [5], which is conducted independently from us and which complements our work in several respects. The authors study the complexity of computing the answer to SPARQL `SELECT` queries using property paths without numerical occurrence indicators. They focus their study on the ALP procedure in [25], which defines the semantics of property paths on a very detailed level. Instead, we focus on a more high-level semantics of property paths, namely the definition of the operators ZeroOrMorePath and OneOrMorePath in the SPARQL Algebra in [25]. Further work on the complexity of SPARQL query evaluation can be found in [36, 38]. We refer to [6] for further references on research on RDF databases and query languages.

2. PRELIMINARIES

For the rest of the paper, 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-*string* (or simply string) 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 string by ε. The set of *positions of* w is $\{1,\ldots,n\}$ and the *symbol of* w *at position* i is a_i. By $w_1 \cdot w_2$ we denote the *concatenation* of two strings w_1 and w_2. For readability, we usually denote the concatenation of w_1 and w_2 by w_1w_2. The set of all strings is denoted by Δ^*. A *string language* is a subset of Δ^*. For two string 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: ε and every Δ-symbol is a regular expression; and when r and s are regular expressions, then (rs), $(r+s)$, $(r?)$, (r^*), and (r^+) are also regular expressions. (Usually we omit braces to improve readability.) 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 $!\{a_1,\ldots,a_n\}$ is a regular expression.

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

By RE($\mathcal{X}$) we denote the set of regular expressions with additional features $\mathcal{X} \subseteq \{\#, \neg, \bullet, !\}$ where "#" stands for numerical occurrence indicators, "¬" for negation, "!" for the negated label test, and "•" for the single-symbol wildcard. For example, RE(#) denotes the set of regular expressions with numerical occurrence indicators and RE(#, ¬, •) is the set of regular expressions with numerical occurrence indicators, negation, and wildcard. We are particularly interested in the following class of expressions.

DEFINITION 2.1. The set of *SPARQL Regular Expressions* or *SPARQL Property Paths* is the set RE(#, !, •).[2]

We consider edge-labeled graphs. 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 don't care about the label of an edge, 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 $p = v_0v_1 \ldots v_n$. We say that path p has *length* n. Notice that a path of length zero does not follow any edges. The *labeled string* 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 $\text{lab}(p)$. 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 $p_1p_2 := v_0[a_1]v_1 \cdots v_{n-1}[a_n]v_n[a_{n+1}]v_{n_1} \cdots v_{n+m-1}[a_{n+m}]v_{n+m}$.

Regular Path Semantics.

The language defined by an expression r, denoted by $L(r)$, is inductively defined as follows: $L(\varepsilon) = \{\varepsilon\}$; $L(a) = \{a\}$; $L(!\{a_1,\ldots,a_n\}) = \Delta - \{a_1,\ldots,a_n\}$; $L(\bullet) = \Delta$; $L(rs) = L(r) \cdot L(s)$; $L(r+s) = L(r) \cup L(s)$; $L(r^*) = \{\varepsilon\} \cup \bigcup_{i=1}^{\infty} L(r)^i$, $L(r^{k,\ell}) = \bigcup_{i=k}^{\ell} L(r)^i$; and, $L(\neg r) = \Delta^* - L(r)$. Furthermore, $L(r?) = \varepsilon + L(r)$ and $L(r^+) = L(r)L(r^*)$.[3] The *size* of a regular expression r over Δ, denoted by $|r|$, is the number of occurrences of Δ-symbols, •-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 $\text{lab}(p) \in L(r)$.

Simple Walk Semantics (Semantics in SPARQL).

A *simple path* is a path $v_0v_1 \cdots v_{n-1}v_n$, where each node v_i occurs exactly once. 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. We say that a *simple walk* is either a simple path or a simple cycle. In the SPARQL 1.1 definition, the W3C specifies the following constraint, which we call "simple walk requirement":

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

The W3C SPARQL algebra (Section 18.4 of [25]) defines the semantics of r^* and r^+ through its operators *ZeroOrMorePath* and *OneOrMorePath*. It is not clear to us whether the currently stated definition only allows *simple paths* or *simple walks* to match. Our complexity results hold for both options.

Let $p = v_0[a_1]v_1[a_2]v_2 \cdots v_{n-1}[a_n]v_n$ be a path and r be a SPARQL regular expression. Then p *matches* r *under simple walk semantics* if one of the following holds:

- If $r = \varepsilon$, $r = a \in \Delta$, $r = \bullet$, or $r = !\{a_1,\ldots,a_n\}$ 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.
- If $r = s^{k,\ell}$ with $\ell \neq \infty$, then there exists 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 $i = 1,\ldots,m$.
- If $r = s^{k,\infty}$, then there exist paths p_1 and p_2 such that $p = p_1p_2$, p_1 matches $s^{k,k}$ under simple walk semantics, and p_2 matches s^* under simple walk semantics.

Notice that, 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.

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

[2] In this paper, we mostly refer to these expressions as "SPARQL regular expressions" to avoid confusion between expressions and paths.

[3] We do not define r^+ as an abbreviation of rr^* since r^+ and rr^* have different semantics in SPARQL.

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*, if 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 $\mathrm{lab}(p_1) \neq \mathrm{lab}(p_2)$. 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?

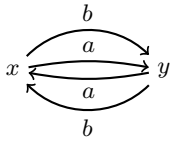
COUNTING: Given a graph (V, E, x, y), a regular expression r and a natural number max in unary, how many different paths of length at most max between x and y match r?

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 papers on (1) and (2) (e.g., [27, 4, 40]). Furthermore, it strengthens our hardness results. However, our polynomial-time results for COUNTING still hold when the number max is given in binary (Theorems 4.4 and 3.11).

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$(\#, \neg)$.

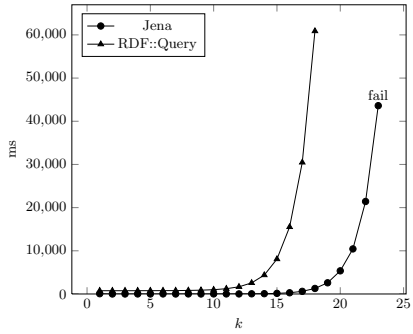
3. THE EVALUATION PROBLEM

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 (which is used in, e.g., ARQ), Sesame, RDF::Query, and Corese 3.0.[4] 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

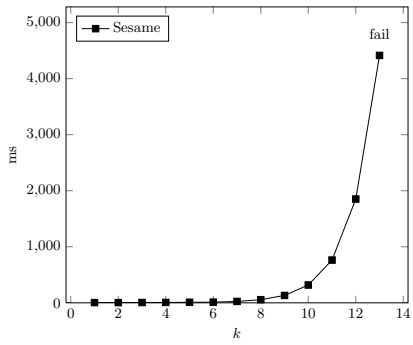


consisting of two nodes and four labeled edges. Formally, this corresponds to answering the EVALUATION problem on the above graph for the expression $(a + b)^{1,k}$. Notice that the answer is always "true". Furthermore, notice that this query has the same semantics under regular path semantics as under 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 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. We note that Arenas, Conca, and Pérez [5] observed double exponential behavior for `SELECT` queries in an independent study. However, `ASK` queries are easier to evaluate since they only ask for a Boolean answer.



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



(b) Evaluation time for Sesame.

Figure 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.

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. In particular, we present a polynomial-time algorithm for EVALUATION of SPARQL regular expressions.

We briefly discuss some basic results on evaluating regular expressions on graphs. EVALUATION is in PTIME for stan-

[4]RDF3X was also recommended to us as a benchmark system but, as far as we could see, it does not support property paths.

dard regular expressions.[5] 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. Pérez et al. have shown 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 [37].

The 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 an RE($\#, !, \bullet$)-expression and let $G = (V, E)$ be a graph. Our 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 to their corresponding subexpressions. We store, for each node in the syntax tree with associated subexpression s, a binary relation $R_s \subseteq V \times V$ such that

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

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.

If s is a Δ-symbol, then $R_s := \{(u, v) \mid (u, s, v) \in E\}$.

If $s = \varepsilon$, then $R_s := \{(u, u) \mid u \in V\}$.

If $s = !\{a_1, \ldots, a_n\}$, then $R_s := \{(u, v) \mid \exists a \in \Delta - \{a_1, \ldots, a_n\}$ with $(u, a, v) \in E\}$.

If $s = \bullet$, then $R_s := \{(u, v) \mid \exists a \in \Delta$ with $(u, a, v) \in E\}$.

If $s = s_1 + s_2$, then $R_s = R_{s_1} \cup R_{s_2}$.

If $s = s_1 \cdot s_2$, then $R_s := \pi_{1,3}(R_{s_1} \underset{R_{s_1}.2=R_{s_2}.1}{\bowtie} R_{s_2})$, where $\underset{R_{s_1}.2=R_{s_2}.1}{\bowtie}$ denotes the ternary relation obtained by joining R_{s_1} and R_{s_2} by pairing tuples that agree on the right column of R_{s_1} and the left column on R_{s_2}. Furthermore, $\pi_{1,3}$ denotes the projection of these triples onto the leftmost and rightmost column.

If $s = s_1^*$, then R_s is the reflexive and transitive closure of R_{s_1}.

If $s = s_1^+$, then R_s is the transitive closure of R_{s_1}.

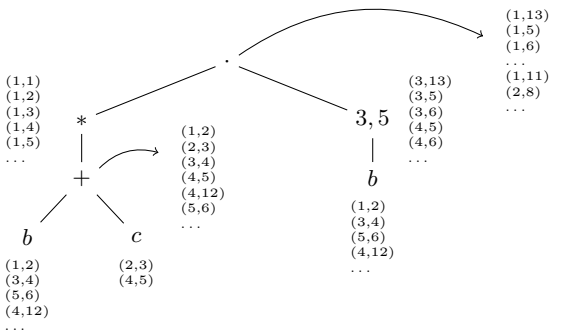
If $s = s_1^k$, then consider the connectivity matrix M_{s_1} of pairs that match s_1 in G. That is, for each pair of nodes (u, v) in G, we have that $M_{s_1}[u, v] = 1$ if and only if $(u, v) \in R_{s_1}$ and $M_{s_1}[u, v] = 0$ otherwise. Notice that M_{s_1} is a $|V| \times |V|$ matrix. Then $R_s := \{(u, v) \mid M_{s_1}^k[u, v] \neq 0\}$, where $M_{s_1}^k$ denotes the matrix M_{s_1} to the power of k.

If $s = s_1^{k,\infty}$, then R_s is the relation for the expression $s_1^k \cdot s_1^*$.

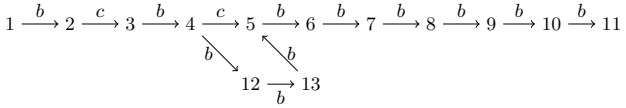
If $s = s_1^{k,\ell}$ and $\ell \neq \infty$, then let M_{s_1} be the same matrix as we used in the s_1^k case. Let M'_{s_1} be the matrix obtained from M_{s_1} by setting $M'_{s_1}[u, v] := 1$ if $u = v$. Therefore we have that $M'_{s_1}[u, v] := 1$ if and only if v is reachable from u by a path that matches s_1 zero or one times. Let $M_s := (M_{s_1})^k \cdot (M'_{s_1})^{\ell - k}$. Then, $R_s := \{(u, v) \mid M_s[u, v] \neq 0\}$.

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 R_r contains the pair (x, y).

[5]This has already been observed in the literature several times, e.g., as Lemma 1 in [35], on p.7 in [2], and in [3].



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



(b) An edge-labeled graph.

Figure 2: Illustration of the polynomial-time dynamic programming algorithm.

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.*

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

THEOREM 3.2. EVALUATION *for SPARQL regular expressions under regular path semantics is in polynomial time.*

PROOF SKETCH. We prove that the dynamic programming algorithm 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.

First, we argue correctness. The following invariant holds for every relation R_s that is calculated: For each subexpression s of r, we have $(u, v) \in R_s \Leftrightarrow \exists$ path p in G from u to v : $\text{lab}(p) \in L(s)$.

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 and that each relation R_s has at most quadratic size in G. We therefore only need to prove that we can implement each separate case in the algorithm in polynomial time.

The cases where $s \in \Delta$, $s = \varepsilon$, $s = \bullet$, $s = s_1^*$, $s = s_1^+$, and $s = s_1 \cdot s_2$ are trivial. For the case $s = s_1^k$, we need to argue that, for a given $|V| \times |V|$ matrix M and a number k given in

binary, we can compute M^k in polynomial time. However, this is well-known to be possible in $\lceil \log k \rceil$ iterated squarings (sometimes also called successive squaring) ([7], page 74). In the case $s = s_1^{k,\infty}$, we only need to compute the relation for s_1 once, copy it, compute $M_{s_1}^k$ as before, compute the transitive and reflexive closure of R_{s_1} and join the results. All of this can be performed in polynomial time. Finally, also the case of $s = s_1^{k,\ell}$ can be computed in polynomial time by using the same methods. This concludes our proof. □

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 [30]. Dynamic programming for testing whether a string belongs to a language of a regular expression has been demonstrated in [26] (p.75–76). Kilpeläinen and Tuhkonen adapted this approach for evaluating RE(#) on strings [28]. However, the algorithm from [28] does not naïvely work on graphs: it would need time exponential in the expression.[6]

We conclude this section with a few observations on the dynamic programming algorithm. Most notably: if we want to evaluate expressions on strings instead of graphs, we can also incorporate negation into the algorithm. By MEMBERSHIP we denote the following decision problem: Given a string w and a regular expression r, is $w \in L(r)$?

THEOREM 3.3. MEMBERSHIP *for RE(#, !, ¬, •) is in polynomial time.*

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

3.2 Negation Makes Evaluation Hard

The negated label test seems to be harmless for the efficiency of evaluating SPARQL regular expressions. On strings, even the full-fledged negation operator "¬" 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 3.4. *Let C be a class of regular expressions over a finite alphabet Σ. Then there exists a 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 $E = \{(x, a, x) \mid a \in \Sigma\}$. □

Since the emptiness problem of *star-free generalized regular expressions*[7] is non-elementary [39], we therefore also immediately have that EVALUATION is non-elementary for RE(¬)-expressions, by Lemma 3.4.

[6]It uses the fact that the length of the *longest match* of the expression on the string cannot exceed the length of the string. For example, the regular expression a^{42} can only match a string if it contains 42 a's. So the fact that 42 is represented in binary notation does not matter for the combined complexity the problem. This assumption no longer holds in graphs.

[7]A star-free generalized regular expression is a regular expression with concatenation, disjunction, and negation.

THEOREM 3.5. EVALUATION *under regular path semantics is non-elementary for RE(¬).*

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

THEOREM 3.6. EVALUATION *under regular path semantics is decidable but non-elementary for RE(#, !, ¬, •)*

3.3 SPARQL Semantics

We study how the complexity of EVALUATION changes when SPARQL's simple walk semantics rather than regular path semantics is applied.

NP-Complete Fragments.

THEOREM 3.7 ([35]). EVALUATION *under simple walk semantics is NP-complete for the expression $(aa)^*$ and for the expression $(aa)^+$.*

The lower bound immediately follows from Theorem 1 in [35], where it is shown that it is NP-hard to decide whether there exists a simple path of even length between two given nodes x and y in a graph G. The upper bound is trivial.

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

THEOREM 3.8. EVALUATION *under simple walk semantics is NP-complete for RE(#, !, •)-expressions.*

PROOF SKETCH. The NP lower bound is immediate from Theorem 3.7. The NP upper bound follows from an adaptation of the dynamic programming algorithm of Section 3 where, in the cases for $s = s_1^*$ and $s = s_1^+$, simple walks are guessed between nodes to see if they belong to R_s. □

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

COROLLARY 3.9. EVALUATION *under simple walk semantics is NP-complete for RE.*

Polynomial Time Fragments.

Theorem 3.7 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 [33, 34]. We will also use these expressions in Section 4.

DEFINITION 3.10 (CHAIN REGULAR EXPRESSION [34]). A *base symbol* is a regular expression w, w^*, w^+, or $w?$, where w is a non-empty string; 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 strings. An *(extended) chain regular expression (CHARE)* is $\emptyset$, ε, or a concatenation of factors.

We use the same shorthand notation for CHAREs as in [34]. The shorthands we use for the different kind of factors is illustrated in Table 1. For example, the regular expression

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

Table 1: 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 strings in Δ^+.

$((abc)^* + b^*)(a + b)?(ab)^+(ac + b)^*$ 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 and the nesting of Kleene star with concatenation. Notice that each kind of factor that is *not* listed in Table 1 can be simulated through one of 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 [8] 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^+, $(+w)^*$).

The following theorem shows that it is possible to use the *- and $^+$-operators and have a fragment for which evaluation is in polynomial time. However, below it, one is only allowed to use a disjunction of single symbols.

THEOREM 3.11. EVALUATION *under simple walk semantics for CHARE*$((+a)^*, (+a)^+, (+w), (+w)?)$ *is in PTIME.*

PROOF SKETCH. This theorem follows from the observation that, for each regular expression r from this class, there exists a path p from a node x to y that matches r under simple walk semantics if and only if there exists a path p' from x to y that matches r under regular path semantics. □

Notice that the range of possibilities between the expressions in CHARE$((+a)^*, (+a)^+, (+w), (+w)?)$ and the expressions in Theorem 3.7 is quite limited. Furthermore, notice the relationship between Theorem 3.11 and Theorem 1 in [35]: Whereas testing the existence of a simple path that matches the expression a^*ba^* is NP-complete [35], testing the existence of a path that matches the expression a^*ba^* under simple walk semantics is in PTIME (Theorem 3.11).

A limitation of Theorem 3.11 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 3.12. EVALUATION *under simple walk semantics is in PTIME for RE-expressions that do not use the *- or $^+$-operators.*

Theorem 3.11 and Observation 3.12 seem to make one central point apparent: unless P = NP, simple walk semantics is tractable as long as it is essentially the same than regular path semantics.

4. THE COUNTING PROBLEM

In this section we study the complexity of COUNTING and FINITENESS. Our motivation for COUNTING comes from the SPARQL definition that requires that, for simple SPARQL queries of the form `SELECT ?x, ?y WHERE {?x r ?y}` for a path expression r, 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.

4.1 Regular Path Semantics

We first show that it is possible to count paths for deterministic patterns and then that allowing even the slightest amount of non-determinism makes the counting problem #P-complete.

Counting for Deterministic Patterns.

We consider finite automata that read Δ-strings. The automata behave very similarly to standard finite automata (see, e.g., [26]), but they can make use of a wildcard symbol "$\circ$" to deal with the infinite set of labels. More formally, an NFA N 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 set of input symbols, $\delta : Q \times (\Sigma \uplus \{\circ\}) \times Q$ is the transition relation, q_0 is the initial state, and Q_f is the set of final states. The *size* of an NFA is $|Q|$, i.e., its number of states.

When the NFA 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 $!\{a_1, \ldots, a_n\}$ in automata.[8] Nevertheless, an expression from RE can still be translated into an equivalent NFA in polynomial time. More formally, a *run* r of N on

[8]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 our definition would be clearer since it defines the semantics of every $\circ$-transition the same across the whole automaton.

a Δ-word $w = a_1 \cdots a_n$ is a string $q_0 q_1 \cdots q_n$ in Q^* such that, for every $i = 1, \ldots, n$, if $a_i \in \Sigma$, then $(q_{i-1}, a_i, q_i) \in \delta$; otherwise, $(q_{i-1}, \circ, q_i) \in \delta$. 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 N if there exists an accepting run of N on w. The *language* $L(N)$ of N is the set of words accepted by N. A *path* p *matches* N if $\mathrm{lab}(p) \in L(N)$.

We say that an NFA is *deterministic*, or a DFA, 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 $(q_1, a, q_2) \in \delta$. In this case, we will also slightly abuse notation and write $\delta(q_1, a) = q_2$.

We discuss the relationship between SPARQL regular expressions and DFAs by (informally) revisiting the *Glushkov-automaton* of a regular expression (see also [9, 24]). Let r be a SPARQL regular expression and let Σ_r be the set of Δ-symbols occurring in r. By $\mathrm{num}(r)$ we denote the *numbered regular expression* obtained from r by replacing each subexpression of the form $!S$, $\bullet$, 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 $\mathrm{num}(r) = 1\ 2\ 3\ (4\ +5\ 6)^*7\ 8$. Formally, $\mathrm{num}(r)$ can be obtained by traversing the parse tree of r depth-first left-to-right and making the appropriate replacements. By denum_r we denote the mapping that maps each number i to the subexpression it replaced in r. In the above example, $\mathrm{denum}_r(1) = a$, $\mathrm{denum}_r(2) = !\{a\}$, $\mathrm{denum}_r(3) = \bullet$, 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 $\Sigma' \subseteq \mathbb{N}$. Let $\mathrm{first}(r_m)$ be the set of all symbols $i \in \Sigma'$ such that $L(r_m)$ contains a word iz, where $z \in (\Sigma')^*$. Furthermore, let $\mathrm{follow}(r_m, i)$ be the set of symbols $j \in \Sigma'$ such that there exists Σ'-strings v, w with $vijw \in L(r_m)$, and let $\mathrm{last}(r_m)$ 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 $Q_f = \mathrm{last}(r_m) \uplus \{q_0\}$. Otherwise, $Q_f = \mathrm{last}(r_m)$. For each $a \in \Sigma_r$ and $i \in Q_r$, the transition function δ_r is defined as follows: (1) $\delta_r(q_0, a) = \{i \in \mathrm{first}(r_m) \mid \mathrm{denum}(i) = a, \mathrm{denum}(i) = \bullet$, or $\mathrm{denum}(i) = !S$ with $a \notin S\}$ and (2) $\delta_r(i, a) = \{j \in \mathrm{follow}(r_m, i) \mid \mathrm{denum}(j) = a, \mathrm{denum}(j) = \bullet$, or $\mathrm{denum}(j) = !S$ with $a \notin S\}$. Furthermore, (3) $\delta_r(q_0, \circ) = \{i \in \mathrm{first}(r_m) \mid \mathrm{denum}(i) = \bullet$ or $\mathrm{denum}(i) = !S$ for some $S\}$ and (4) $\delta_r(i, \circ) = \{j \in \mathrm{follow}(r_m, i) \mid \mathrm{denum}(j) = \bullet$ or $\mathrm{denum}(j) = !S$ for some $S\}$.

Proposition 4.1. *For each SPARQL regular expression r, the Glushkov automaton of r can be constructed in polynomial time. Furthermore, $L(r) = L(G_r)$.*

We say that a SPARQL regular expression r is *deterministic*, or a *Det-RE*, if G_r is a DFA.

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 a DFA. 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 DFA, then there is a strong correspondence between paths from x to y in G and paths from $q_0^{G,A}$ to $Q_f^{G,A}$ in $G^{x,y} \times A$. We formalize this correspondence by a mapping $\varphi_{\textsc{Paths}}$, which we define inductively as follows:

- $\varphi_{\textsc{Paths}}(x) := (x, q_0)$;
- for each v_1 such that $\varphi_{\textsc{Paths}}(v_1) = (v_1, q_1)$ and for each edge $e = (v_1, a, v_2)$ in G, we define
  - $\varphi_{\textsc{Paths}}(v_2) := (v_2, q_2)$, where q_2 is the unique state such that $\delta_A(q_1, a) = q_2$ or $\delta_A(q_1, \circ) = q_2$; and
  - $\varphi_{\textsc{Paths}}(e) := \big((v_1, q_1), a, (v_2, q_2)\big)$.

We extend the mapping $\varphi_{\textsc{Paths}}$ in the canonical manner to paths in G starting from x. Notice that $\varphi_{\textsc{Paths}}$ is only well-defined if A is a DFA.

Lemma 4.2. *If A is a DFA, then $\varphi_{\textsc{Paths}}$ 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, $\varphi_{\textsc{Paths}}$ preserves the length of paths.*

We recall the following graph-theoretical result that states that the number of arbitrary paths between two nodes in a graph can be counted quickly (see, e.g., [7], page 74):

Theorem 4.3. *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.*

Again, the reason why the number of paths can be counted so quickly is due to the fast squaring method that can compute, for a given (square) matrix M, the matrix M^k in $O(\log k)$ matrix multiplications.

Theorem 4.4. Counting *for DFAs is in polynomial time, even if the number max in the input is given in binary.*

Proof. We reduce Counting for DFAs 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 4.3.

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

- Let $G^{x,y} \times A$ be the product of (V, E, x, y) and A.
- Return $\sum_{q_f \in Q_f} \textsc{Paths}\big((x, q_0), (y, q_f)\big)$ in $G^{x,y} \times A$.

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 4.2, 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 $G^{x,y} \times A$. $\square$

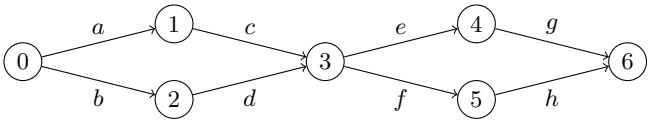


Figure 3: An edge-labeled graph $(V, E, 0, 6)$.

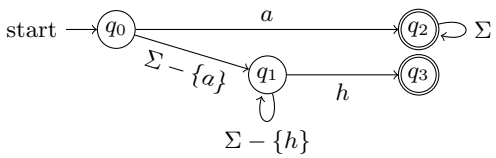


Figure 4: A DFA A for the regular expression $(a\Sigma^* + \Sigma^+h)$, with $\Sigma = \{a, b, c, d, e, f, g, h\}$.

Actually, Theorem 4.4 even holds for automata that are *unambiguous*, i.e., automata that only allow exactly one accepting run for each word in the language.

We illustrate the algorithm of Theorem 4.4 on an example. Consider the DFA 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 start state to an accepting state in P.

Counting for Non-Deterministic Patterns.

We start by observing that Counting is in #P for standard regular expressions.

Theorem 4.5. Counting *is in #P for all REs.*

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

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. 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 DFAs 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 can not be trivially converted to polynomial-size DFAs.

Theorem 4.6. 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 4.5. The lower bounds can be proved by reductions from #DNF. We show the reduction for case (1). Our technique is inspired by a proof in [34], 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

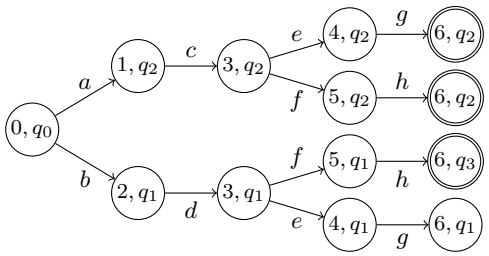


Figure 5: Fragment of the product $G^{0,6} \times A$ of $(V, E, 0, 6)$ from Figure 3 and DFA A from Figure 4. The nodes in $Y_{G,A}$ are in double circles.

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_k is "true". If it chooses ba, it means that x_k 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 $j = 1, \ldots, n$,

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

This concludes the reduction for case (1). □

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

Theorem 4.7. Counting *for RE(#, ¬, •) is #P-complete.*

4.2 SPARQL Semantics

We investigate how the complexity of Counting changes when we apply SPARQL's 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 4.8. Counting *under simple walk semantics for CHARE$(a, (+a))$ is in PTIME*

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 DFAs in polynomial time.

Theorem 4.9. Counting *under simple walk semantics is #P-complete for the expressions a^* and a^+.*

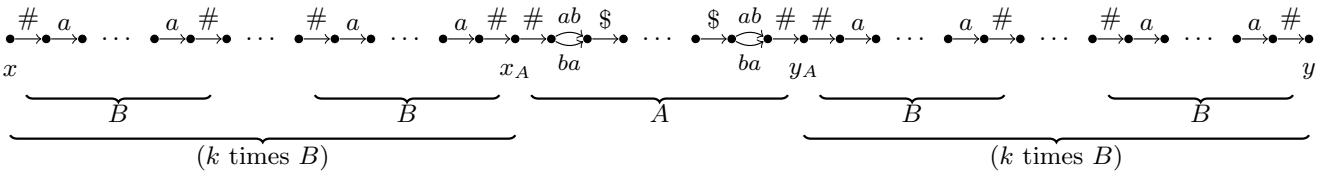


Figure 6: The graph G from the proof of Theorem 4.6.

PROOF. We reduce from the problem of counting the number of simple s-t paths in a graph, which was shown to be #P-complete by Valiant [40]. □

Theorem 4.9 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 4.9, but it is immediate from the observation that the reduction for regular path semantics applies here as well.

THEOREM 4.10. 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 expressions in RE(#, !, •).

THEOREM 4.11. COUNTING *under simple walk semantics for RE(#, !, •) is #P-complete.*

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. Therefore we complete the picture of our complexity analysis by looking at the FINITENESS problem.

Using the product construction (Section 4.1), 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 PTIME for RE.*

By adapting the dynamic programming algorithm to also annotate the length of the longest paths associated to a pair in each relation, we can even decide FINITENESS for RE(#, !, •) in PTIME.

THEOREM 5.2. FINITENESS *for RE(#, !, •) is in PTIME.*

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 over a finite alphabet Σ, such that membership testing of ε for expressions in C is in polynomial time. Then there exists a polynomial reduction from the emptiness problem for C-expressions to the* FINITENESS *problem with C-expressions.*

For $r \in$ RE(¬), one can test whether $\varepsilon \in L(r)$ in linear time traversing the syntax tree of r. If $\varepsilon \notin L(r)$, then testing emptiness for r is equivalent to FINITENESS on the expression r^* and the graph $G = (V, E, x, x)$ with $V = \{x\}$ and $E = \{(x, a, x) \mid a \in \Sigma\}$. Therefore, by [39] and Lemma 5.3 we have the following.

THEOREM 5.4. FINITENESS *is decidable but non-elementary for RE(¬).*

6. DISCUSSION

An overview of our results is presented in Table 2. CHAREs are defined in Section 3.3. By "star-free RE", we denote standard regular expressions that do not use the operators "*" and "+". 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, except for the result on CHARE($a, a?$). Indeed, if the CHARE($a, a?$) is fixed, we can translate it to a DFA 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 and perform the algorithm for DFAs. When considering data complexity, the difference between regular path semantics and simple walk semantics therefore becomes even more striking.

Possible alternatives for the W3C.

The NP-complete and #P-complete data complexities make the current semantics of W3C property paths highly problematic from a computational complexity perspective, especially on a Web scale. There are two orthogonal requirements in the current W3C proposal 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.

By removing the simple walk requirement and the path counting requirement, the answer to the above SELECT query would be the set of pairs (x, y) in the graph such that there exists a path from x to y that matches r under regular path semantics. As such, each pair is returned at most once. Similar to [5], which is work conducted independently from ours, we believe that the W3C should use this semantics as a default semantics for property paths in SELECT queries. Our results show that SPARQL property

Problem	Fragment	Regular path semantics	Simple walk semantics
EVALUATION	CHARE($(+a)^*$,$(+a)^+$,$(+w)$,$(+w)?$)	in PTIME	**in PTIME** (3.11)
	star-free RE	in PTIME	**in PTIME** (3.12)
	$(aa)^*$	in PTIME	NP-complete [35]
	RE	in PTIME	**NP-complete**
	RE($\#, !, \bullet$)	**in PTIME** (3.2)	**NP-complete** (3.8)
	RE($\neg$)	**non-elementary** (3.5)	—
	RE($\#, !, \neg, \bullet$)	**non-elementary** (3.6)	—
COUNTING	DFA	**in FPTIME** (4.4)	—
	CHARE(a, $(+a)$)	**in FPTIME**	**in FPTIME** (4.8)
	a^+	**in FPTIME**	#P-complete (4.9,[40])
	a^*	**in FPTIME**	#P-complete (4.9,[40])
	Det-RE	**in FPTIME**	**#P-complete**
	CHARE(a, a^*)	**#P-complete** (4.5,4.6)	**#P-complete**
	CHARE($a, a?$)	**#P-complete** (4.5,4.6)	**#P-complete**(4.10)
	CHARE($a, (+a^+)$)	**#P-complete** (4.5,4.6)	**#P-complete**
	CHARE($a, (+a)^+$)	**#P-complete** (4.5,4.6)	**#P-complete**
	CHARE(a, w^+)	**#P-complete** (4.5,4.6)	**#P-complete**
	CHARE($(+a), a^+$)	**#P-complete** (4.5,4.6)	**#P-complete**
	RE	**#P-complete** (4.5,4.6)	**#P-complete**
	RE($\#, !, \bullet$)	**#P-complete** (4.7)	**#P-complete** (4.11)
	RE($\#, !, \neg, \bullet$)	**#P-complete** (4.7)	—
FINITENESS	RE	**in PTIME** (5.1)	—
	RE($\#, !, \bullet$)	**in PTIME** (5.2)	—
	RE($\neg$)	**non-elementary** (5.4)	—
	RE($\#, !, \neg, \bullet$)	**non-elementary** (5.4)	—

Table 2: An overview of most of our complexity results. The results printed in bold are new, to the best of our knowledge. 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. These results concern combined complexity. In data complexity, the entire column "Regular path semantics" drops to PTIME/FPTIME. For simple walk semantics, only the complexity of CHARE(a,a?) drops to FPTIME under data complexity.

paths (cfr. Def. 2.1), which can be exponentially more succinct than standard regular expressions, can then be evaluated in polynomial time combined complexity by a simple dynamic programming algorithm. Preliminary results indicate that we can even leverage this technique to evaluate *nested regular expressions* [37] with numerical occurrence indicators in polynomial time combined complexity.

However, it is possible that in some scenarios one would like to have a bag semantics for property paths and, therefore, paths would need to be counted. We think that removing the simple walk requirement would be wise here as well. As we showed, doing so would drop the data complexity of COUNTING from #P-complete to polynomial time for almost all non-trivial queries. However, in this case, it is still less clear *how* one would like to count paths. At the moment, the W3C has a procedural definition for counting paths which is studied in depth in [5], where it is proved that it leads to massive computational problems. Furthermore, we believe that this definition is rather opaque and that it should be much more transparent to end-users and researchers.

So, what could we do? When one would naïvely adopt regular path semantics for counting paths, one would need to find a way to deal with the case where there are infinitely many paths between two nodes that match an expression. In principle, it is possible to deal with this case efficiently. We proved that deciding whether this case applies, i.e., solving the FINITENESS problem, is possible for SPARQL regular expressions in polynomial time. One could also avoid the need to decide this case entirely. An ad-hoc solution could be to simply not count paths anymore beyond a certain number. Such a solution may be sufficient for many practical purposes but is theoretically not very elegant. Perhaps more elegant would be to only count paths that are, in some sense, *shortest paths*.[9] Again, various options are possible. One could, e.g., first compute the length of the shortest path and then count all paths that have this length. Another option is to count all the paths p from x to y such that there does not exist a sub-path of p from x to y that also matches the expression r. We do not think that the last word has been said on this topic and that further research is needed. Essentially, we need to find a semantics that is intuitive, easy to understand, and efficient to compute. Unfortunately, the present semantics fulfills neither condition.

We strongly believe that a feasible solution for property paths in SPARQL should avoid the simple walk requirement due to complexity reasons: from our perspective, the choice between NP-complete data complexity already for the query $(aa)^*$; or polynomial time combined complexity for the full fragment of SPARQL property paths seems to be a rather easy one to make.

[9]This idea was pitched by Serge Abiteboul during a Dagstuhl seminar on foundations of distributed data management.

Acknowledgments

Several people contributed to this work through inspiring discussions. We thank Marcelo Arenas for sending us a draft of [5] and for his comments on a draft of this paper; Thomas Schwentick for pointing out that emptiness testing of star-free regular expressions with negation is non-elementary (i.e, Theorem 3.5); and Pekka Kilpeläinen for giving us the 1979 reference for dynamic programming for evaluating regular expressions on strings. We thank the anonymous reviewers of PODS 2012 for their insightful comments that helped to improve the presentation of the paper.

7. REFERENCES

[1] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. L. Wiener. The Lorel query language for semistructured data. *Int. J. on Digital Libraries*, 1(1):68–88, 1997.

[2] S. Abiteboul and V. Vianu. Regular path queries with constraints. *J. Comput. Syst. Sci.*, 58(3):428–452, 1999.

[3] F. Alkhateeb, J.-F. Baget, and J. Euzenat. Extending SPARQL with regular expression patterns (for querying RDF). *J. Web Sem.*, 7(2):57–73, 2009.

[4] C. Álvarez and B. Jenner. A very hard log-space counting class. *Theor. Comput. Sci.*, 107:3–30, 1993.

[5] M. Arenas, S. Conca, and J. Pérez. Counting beyond a yottabyte, or how SPARQL 1.1 property paths will prevent the adoption of the standard. In *World Wide Web Conference (WWW)*, 2012. To appear.

[6] M. Arenas and J. Pérez. Querying semantic web data with SPARQL. In *Principles of Database Systems (PODS)*, p. 305–316, 2011.

[7] C. Berge. *Graphs and Hypergraphs*. North-Holland Publishing Company, 1973.

[8] G. J. Bex, F. Neven, T. Schwentick, and S. Vansummeren. Inference of concise regular expressions and DTDs. *ACM Trans. Database Syst.*, 2010.

[9] R. Book, S. Even, S. Greibach, and G. Ott. Ambiguity in graphs and expressions. *IEEE Trans. Comput.*, 20:149–153, 1971.

[10] P. Buneman, S. B. Davidson, G. G. Hillebrand, and D. Suciu. A query language and optimization techniques for unstructured data. In *SIGMOD Conference*, p. 505–516, 1996.

[11] D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Y. Vardi. Containment of conjunctive regular path queries with inverse. In *Principles of Knowledge Representation and Reasoning (KR)*, p. 176–185, 2000.

[12] D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Y. Vardi. View-based query processing for regular path queries with inverse. In *Principles of Database Systems (PODS)*, pages 58–66, 2000.

[13] D. Calvanese, G. De Giacomo, M. Lenzerini, and M.Y. Vardi. Rewriting of regular expressions and regular path queries. *J. Comput. Syst. Sci.*, 64(3):443–465, 2002.

[14] D. Colazzo, G. Ghelli, and C. Sartiani. Efficient asymmetric inclusion between regular expression types. In *International Conference on Database Theory (ICDT)*, pages 174–182, 2009.

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

[16] M. P. Consens and A. O. Mendelzon. GraphLog: a visual formalism for real life recursion. In *Principles of Database Systems (PODS)*, p. 404–416, 1990.

[17] I. F. Cruz, A. O. Mendelzon, and P. T. Wood. A graphical query language supporting recursion. In *SIGMOD Conference*, p. 323–330, 1987.

[18] A. Deutsch and V. Tannen. Optimization properties for classes of conjunctive regular path queries. In *Database Programming Languages (DBPL)*, p. 21–39, 2001.

[19] M. F. Fernández, D. Florescu, A. Y. Levy, and D. Suciu. Declarative specification of web sites with strudel. *VLDB J.*, 9(1):38–55, 2000.

[20] D. Florescu, A. Y. Levy, and D. Suciu. Query containment for conjunctive queries with regular expressions. In *Principles of Database Systems (PODS)*, p. 139–148, 1998.

[21] S. Gao, C. M. Sperberg-McQueen, H.S. Thompson, N. Mendelsohn, D. Beech, and M. Maloney. W3C XML Schema Definition Language (XSD) 1.1 part 1: Structures. Tech. report, World Wide Web Consortium, April 2009.

[22] W. Gelade, M. Gyssens, and W. Martens. Regular expressions with counting: Weak versus strong determinism. *SIAM J. Comput.*, 41(1):160–190, 2012.

[23] W. Gelade, W. Martens, and F. Neven. Optimizing schema languages for XML: Numerical constraints and interleaving. *SIAM J. Comput.*, 38(5), 2009.

[24] V. M. Glushkov. The abstract theory of automata. *Russian Math. Surveys*, 16(5(101)):1–53, 1961.

[25] S. Harris and A. Seaborne. SPARQL 1.1 query language. Tech. report, World Wide Web Consortium (W3C), January 2012.

[26] J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.

[27] S. Kannan, Z. Sweedyk, and S. R. Mahaney. Counting and random generation of strings in regular languages. In *Symp. on Discrete Algorithms (SODA)*, p. 551–557, 1995.

[28] P. Kilpeläinen and R. Tuhkanen. Regular expressions with numerical occurrence indicators — preliminary results. In *Symp. on ProgLang. and Software Tools (SPLST)*, p. 163–173, 2003.

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

[30] S. C. Kleene. *Automata Studies*, chapter Representations of events in nerve sets and finite automata, p. 3–42. Princeton Univ. Press, 1956.

[31] L. Libkin and D. Vrgoč. Regular path queries on graphs with data. In *International Conference on Database Theory (ICDT)*, 2012. To appear.

[32] Y. A. Liu and F. Yu. Solving regular path queries. In *Intl. Conf. on Mathematics of Program Construction (MPC)*, p. 195–208, 2002.

[33] W. Martens, F. Neven, and T. Schwentick. Complexity of decision problems for simple regular expressions. In *Mathematical Foundations of Computer Science (MFCS)*, p. 889–900, 2004.

[34] W. Martens, F. Neven, and T. Schwentick. Complexity of decision problems for XML schemas and chain regular expressions. *SIAM J. Comput.*, 39(4):1486–1530, 2009.

[35] A. O. Mendelzon and P. T. Wood. Finding regular simple paths in graph databases. *SIAM J. Comput.*, 24(6):1235–1258, 1995.

[36] J. Pérez, M. Arenas, and C. Gutierrez. Semantics and complexity of SPARQL. *ACM Trans. Database Syst.*, 34(3), 2009.

[37] J. Pérez, M. Arenas, and C. Gutierrez. nSPARQL: A navigational language for RDF. *J. Web Sem.*, 8(4):255–270, 2010.

[38] M. Schmidt, M. Meier, and G. Lausen. Foundations of SPARQL query optimization. In *International Conference on Database Theory (ICDT)*, pages 4–33, 2010.

[39] L. Stockmeyer. *The complexity of decision problems in automata theory and logic*. PhD thesis, Massachusetts Institute of Technology, 1974.

[40] L. G. Valiant. The complexity of enumeration and reliability problems. *SIAM J. Comput.*, 8(3):410–421, 1979.

[41] M. Yannakakis. Graph-theoretic methods in database theory. In *Principles of Database Systems (PODS)*, p. 230–242, 1990.