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.

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).

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

Recently, the W3C decided to boost SPARQL 1.1 with extensive navigational capabilities by the introduction of *property paths* [24]. 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 [5]. 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.

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

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 7.

Related Work.

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, 8, 15, 16, 18, 40]. Various problems for regular path queries have been investigated in the database community, such as optimization [2], query rewriting and query answering using views [12, 9, 11], and containment [10, 17, 19]. Recently, there has been a renewed interest in path expressions on graphs. For example, expressions that add data value comparisons are studied in [29].

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. [31]. 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. In their further work, Liu et al. extended regular path queries with variables or parameters in order to boost their expressiveness [30].

This paper has a strong focus on regular expressions with numerical occurrence indicators. Such expressions have been investigated in the context of XML schema languages [13, 14, 22, 23, 21, 28, 26, 27] since they are a part of the W3C XML Schema Language [20].

There is a wide body of research on RDF databases and query languages. A nice overview of this work can be found in [5]. The complexity of SPARQL has been studied as well, from different perspectives as ours [35, 36, 3, 37]. To the best of our knowledge, the present paper is the first one that studies the complexity of full property paths (i.e. including numerical occurrence indicators) in SPARQL.

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 $w_1 w_2$. 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 $(\neg r)$.

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, "$\neg$" for negation, 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.

The language defined by an expression r, denoted by $L(r)$, is inductively defined as follows: $L(\varepsilon) = \{\varepsilon\}$; $L(a) = \{a\}$; $L(\bullet) = \Delta$; $L(rs) = L(r)\cdot L(s)$; $L(r+s) = L(r)\cup L(s)$; $L(r^*) = \{\varepsilon\} \cup \bigcup_{i=1}^{\infty} L(r)^i$, $L(r^{k,\ell}) = \bigcup_{i=k}^{\ell} L(r)^i$; and, $L(\neg r) = \Delta^* - L(r)$. Furthermore, $L(r?) = \varepsilon + L(r)$ and $L(r^+) = L(r)L(r^*)$.[2] The *size* of a regular expression r over Δ, denoted by $|r|$, is the number occurrences of Δ-symbols, $\bullet$-symbols, and operators occurring in r, plus the sizes of the binary representations of the numerical occurrence indicators.

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 $\text{lab}(p)$. A path p *matches* a regular expression r if $\text{lab}(p) \in L(r)$. 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 $p_1p_2 := v_0[a_1]v_1 \cdots v_{n-1}[a_n]v_n[a_{n+1}]v_{n_1} \cdots v_{n+m-1}[a_{n+m}]v_{n+m}$.

We consider two paths $p_1 = v_0^1[a_1^1]v_1^1 \cdots [a_n^1]v_n^1$ and $p_2 = v_0^2[a_1^2]v_1^2 \cdots [a_m^2]v_m^2$ to be *different*, if either the sequences of nodes or the sequences of labels are different, i.e., $v_0^1v_1^1 \cdots v_n^1 \neq v_0^2v_1^2 \cdots v_m^2$ 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 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 are mainly interested in the following problems:

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 n between x and y match r?

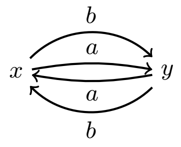
For the COUNTING problem, we chose to have the number max in unary because this was also the case in the classical path counting problems we found in the literature, so it makes results easier to compare. Furthermore, it strengthens our hardness results. However, our polynomial-time results for COUNTING still hold when the number max is given in binary (Theorems 5.3 and 4.6).

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($\#, \neg$).

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

3. THE EFFICIENCY OF EVALUATION

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.[3] 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".)

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 similar double exponential behavior for `SELECT` queries is also observed by Arenas, Conca, and Pérez [4].

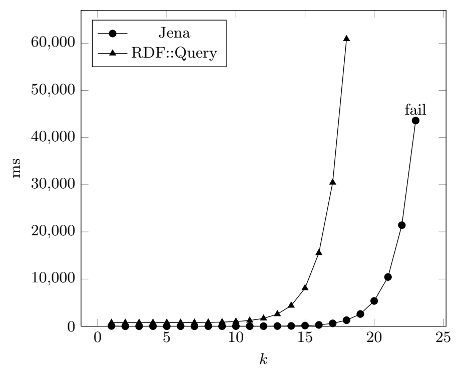
3.1 An Efficient Algorithm for Evaluation

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 RE($\#, \bullet$).

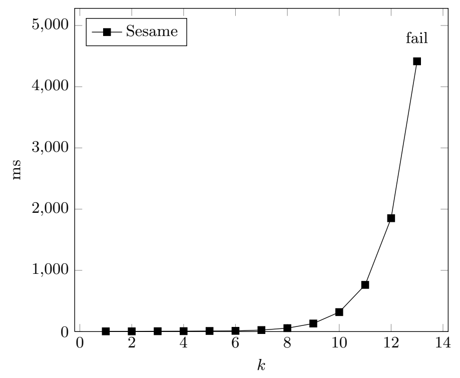
We briefly discuss some basic results on evaluating regular expressions on graphs. EVALUATION is in PTIME for standard regular expressions.[4] In this case, the problem basically boils down to testing intersection emptiness of two finite automata: one converts the graph G with the given nodes x and y 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

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

[4]This has already been observed in the literature several times, e.g., as Lemma 1 in [34], on p.7 in [2], and in [3].



(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.

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 [36].

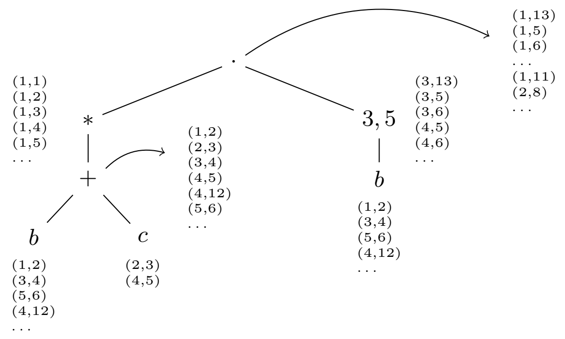
The polynomial time algorithm for EVALUATION of RE(#, •)-expressions follows a dynamic programming approach. We first discuss the main idea of the algorithm and then discuss its complexity. Let r be an RE(#, •)-expression and let $G = (V, E)$ be a graph. Our algorithm traverses the syntax tree of r in 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 $R_s := \{(u, u) \mid u \in V\}$.



(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.

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

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

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(#, •)-expression r, we return the answer "true" if and only if R_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.*

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

THEOREM 3.2. EVALUATION *for RE(#, $\bullet$) 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. In particular, we prove in the Appendix that 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) ([6], 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. $\square$

We note that we are not the first to think of dynamic programming for evaluating regular expressions. Dynamic programming for evaluating expressions on strings has been demonstrated in [25] (p.75–76). Kilpeläinen and Tuhkonen used the approach for evaluating RE(#) on strings [27]. However, the algorithm from [27] does not naïvely work on graphs: it would need time exponential in the expression.[5]

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(#, $\neg$, $\bullet$) 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.

[5]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.

3.2 Negation Makes Evaluation Hard

Unfortunately, the complexity of EVALUATION becomes non-elementary once unrestricted negation is allowed in regular expressions. 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 $E = \{(x, a, x) \mid a \in \Sigma\}$. $\square$

Since the emptiness problem of star-free generalized regular expressions is non-elementary [38], we therefore also immediately have that EVALUATION is non-elementary for RE($\neg$)-expressions, by Lemma 3.4.

THEOREM 3.5. EVALUATION *is non-elementary for RE($\neg$).*

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

THEOREM 3.6. EVALUATION *is non-elementary for RE(#, $\neg$, $\bullet$)*

4. SEMANTICS IN SPARQL

In this section we formally define the semantics of regular expressions according to SPARQL 1.1 [24].

A *simple path* is a path $v_0 v_1 \cdots v_{n-1} v_n$, where each node v_i occurs exactly once. A *simple cycle* is a path $v_0 v_1 \cdots v_{n-1} v_n$ 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.

We discuss in the Appendix where this requirement can be found in [24]. The W3C is not completely clear on the matter whether it only allows *simple paths* or *simple walks* to match, which we also discuss in the Appendix. However, our complexity results hold for both options.

DEFINITION 4.1 (SPARQL REGULAR EXPRESSION). A *SPARQL regular expression*[6] is a regular expression as defined in Section 2 with the difference that the negation operator is only allowed to occur in the form $\neg(a_1 + \cdots + a_n)$, where, for each $i = 1, \ldots, n$, $a_i \in \Delta$.

In order to avoid confusion between the negation operator used before and the more restricted negation from SPARQL, we denote the SPARQL negation operator as "!". The semantics of property paths are defined as follows. 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:

[6]In the SPARQL working draft, SPARQL regular expressions are called "property paths". We refer to them as "SPARQL regular expressions" to avoid confusion between expressions and paths.

- If $r = \varepsilon$, $r = a \in \Delta$, $r = \bullet$, 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.
- 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 $a^{1,\infty}$.

In order to avoid confusion between the semantics of regular expressions we defined in Section 2 and the simple walk semantics, we henceforth refer to the former as "regular path semantics" and to the latter as "simple walk semantics". We note that the query on which we tested the practical engines has the same meaning under simple walk semantics as under regular path semantics.

4.1 NP-Complete Fragments

We study how the complexity of Evaluation changes when simple walk semantics rather than regular path semantics is applied. We start by recalling a result by Mendelzon and Wood.

Theorem 4.2 ([34]). 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 [34], 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 4.3. Evaluation *under simple walk semantics is NP-complete for RE($\#, !, \bullet$)-expressions.*

Proof sketch. The NP lower bound is immediate from Theorem 4.2. 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. The full algorithm is in the Appendix. □

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

Corollary 4.4. Evaluation *under simple walk semantics is NP-complete for RE.*

4.2 Polynomial Time Fragments

Theorem 4.2 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 [32, 33]. These classes of expressions will also come in handy in Section 5.

Definition 4.5 (Chain Regular Expressions [33]). 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 [33]. The shorthands we use for the different kind of factors is illustrated in Table 1. 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 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 [7] and that they are convenient to model classes that allow only a limited amount of non-determinism, which becomes pivotal in Section 5. 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 4.6. 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. A complete proof is given in the Appendix. □

Notice that the range of possibilities between the expressions in CHARE($(+a)^*, (+a)^+, (+w), (+w)?$) and the expressions in Theorem 4.2 is quite limited. Furthermore, notice the relationship between Theorem 4.6 and Theorem 1 in [34]: Whereas testing the existence of a simple path that matches the expression a^*ba^* is NP-complete [34], testing the existence of a path that matches the expression a^*ba^* under simple walk semantics is in PTIME (Theorem 4.6).

A limitation of Theorem 4.6 is that CHAREs do not allow arbitrary nesting of disjunctions. However, since simple

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 words in Δ^+.

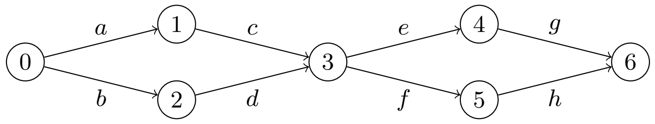


Figure 3: An edge-labeled graph $(V, E, 0, 6)$.

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 4.7. EVALUATION *under simple walk semantics is in PTIME for RE-expressions that do not use the* **- or* $^+$*-operators.*

Theorem 4.6 and Observation 4.7 seem to make one central point apparent: unless P = NP, simple walk semantics is tractable as long as it is essentially not different from regular path semantics.

5. 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.

5.1 Regular Path Semantics

We first want to give the reader some general insight and some intuition behind the challenges of COUNTING for regular path semantics. In particular, we want to convey that, although dynamic programming seems to be a good approach to attack EVALUATION, this is not longer so for COUNTING, at least not in an obvious manner.

Consider the graph $(V, E, 0, 6)$ in Figure 3. We are interested in how many paths from node 0 to node 6 match a regular expression r. An obvious approach to try to tackle this problem is through divide-and-conquer, as we did in Section 3.1: we could try to count partial results for subexpressions and subgraphs; and combine these partial results to obtain the overall result. Here, we give two examples why such an approach will not work naïvely.

The expression $r = (a\bullet\bullet\bullet) + (\bullet c \bullet\bullet)$ shows that recursively counting the number of paths that match $s_1 = (a \bullet \bullet\bullet)$ and $s_2 = (\bullet c \bullet \bullet)$ and adding these numbers together to obtain the number of paths for r does not work. Indeed, there are two paths from 0 to 6 that match s_1, two paths that match s_2 (namely, the same two paths), and yet there are only two paths that match r. The problem is that we counted the paths labeled $aceg$ and $acfh$ twice.

An idea that would solve the problem for the above expression would be to color all the edges that are on some path that matches the subexpressions; and count the number of colored paths from 0 to 6 afterwards. For the above expression, this approach would color the edges labeled a, c, e, g, f, and h and would therefore lead to the correct result. However, this approach also does not work in general. Consider the expression $r = \big(ac(eg + fh)\big) + \big((ac + bd)fh\big)$. With this expression, we would color all the edges in the graph. The number of colored paths from node 0 to node 6 would be four; but the number of paths that match r is only three. Here the problem is that we erroneously add the path $bdeg$ to the result.

A key for obtaining a polynomial-time algorithm therefore seems to lie in some way to decompose the paths of the graph such that matching paths are not counted twice and such that we do not erroneously combine partial paths that are not allowed to occur together. We show that this is possible for deterministic patterns in Section 5.1.1. Furthermore, we show in Section 5.1.2 that allowing even the slightest amount of non-determinism turns the counting problem #P-complete.

5.1.1 *Counting for Deterministic Patterns*

We consider finite automata that read Δ-strings. The automata behave very similarly to standard finite automata (see, e.g., [25]), but they can make use of the wildcard symbol "$\bullet$" 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, $\delta : Q \times (\Sigma \uplus \bullet) \times Q$ 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 $\bullet$-label in outgoing transitions is used to deal with everything else, i.e., the $\bullet$-transitions can be followed when reading $a \notin \Sigma$. Notice that the semantics of the $\bullet$-symbol is therefore different in automata than in regular expressions. The reason for the difference here is that we want to define a natural notion of determinism for automata. Nevertheless, an expression from RE can still be translated into an equivalent NFA in polynomial time. More formally, a *run* r of N on 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, $(q_{i-1}, \bullet, q_i) \in \delta$. 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 $\text{lab}(p) \in L(N)$.

We say that an NFA is *deterministic*, or a DFA, when the relation δ is a function from $Q \times (\Sigma \uplus \bullet)$ to Q. That is, for every $q_1 \in Q$ and $a \in \Sigma \uplus \{\bullet\}$, 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 $\delta(q_1, a) = q_2$.

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 transition $(q_1, a, q_2) \in \delta$ or a transition $(q_1, \bullet, q_2) \in \delta$ 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 φ_{PATHS}, which we define inductively as follows:

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

We extend the mapping φ_{PATHS} in the canonical manner to paths in G starting from x. Notice that φ_{PATHS} is only well-defined if A is a DFA.

LEMMA 5.1. *If A is a DFA, 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.*

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., [6], page 74):

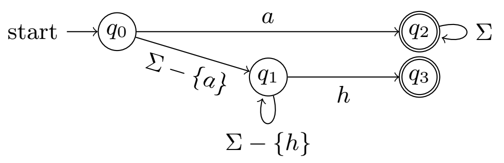


Figure 4: A DFA A for the regular expression $(a\Sigma^* + \Sigma^+ h)$, with $\Sigma = \{a, b, c, d, e, f, g, h\}$.

THEOREM 5.2. *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.*

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 5.3. 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 5.2.

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 $\sum_{q_f \in Q_f} \text{PATHS}\big((x, q_0), (y, q_f)\big)$ in $G^{x,y} \times A$.

Here, $\text{PATHS}\big((x, q_0), (y, q_f)\big)$ 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 5.1, 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 $G^{x,y} \times A$. $\square$

Actually, Theorem 5.3 even holds for automata that are *unambiguous*, i.e., automata that only allow exactly one accepting run for each word in the language. We prove this in the Appendix.

We illustrate the algorithm of Theorem 5.3 on an example. Consider the DFA A in Figure 4. The product of A and $(V, E, 0, 6)$ 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.

5.1.2 Counting for Non-Deterministic Patterns

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

THEOREM 5.4. 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 non-deterministic Turing machine for the $\#P$ procedure simply guesses a path of length at most max and tests whether it matches $L(r)$. $\square$

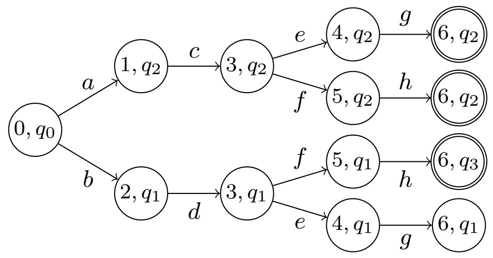


Figure 5: Fragment of the product $G^{0,6} \times A$ of $(V, E, 0, 6)$ the and DFA A from Figure 4. The nodes in $Y_{G,A}$ are in double circles.

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 4.2.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 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 5.5. 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. The upper bound for all cases is immediate from Theorem 5.4. We prove the lower bounds by reductions from #DNF. 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. Our proof technique is inspired by a reduction in [33], 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 prove 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 (and which can be written as "$B^k A B^k$"), 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.

Here, the subgraph α of B is itself also a path which we will instantiate differently in each of the cases (1)–(3). By the second point above we mean 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 $\{x_1, \ldots, x_n\}$. That is, for each truth assignment α, there is a path p_α 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 α_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 (i) each path labeled lab$(B)^i$, for $i = 1, \ldots, k$, matches N and (ii) 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 α_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. For fragment (1), we define: (i) lab$(\alpha) = a$, (ii) lab$(p^{\text{true}}) = ab$, (iii) lab$(p^{\text{false}}) = ba$, (iv) N to be k concatenations of $(\#^*a^*\$^*\cdots\$^*a^*\#^*)$, (v) $r^{\text{true}} = aa^*b^*a^*$, (vi) $r^{\text{false}} = b^*a^*$, and (vii) $r^{\text{all}} = a^*b^*a^*$. Notice that, in this case, r is a CHARE(a, a^*). Finally, we define the number max to be the number of nodes in G. The reductions for the cases (2)–(6) and their proof of correctness is in the Appendix. This concludes the proof of Theorem 5.5. $\square$

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

THEOREM 5.6. COUNTING *for RE$(\#, \neg, \bullet)$ is #P-complete.*

5.2 Semantics in SPARQL

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 4. COUNTING already turns #P-complete as soon as the Kleene star or plus are used. We start by mentioning a polynomial-time result.

THEOREM 5.7. 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 5.8. COUNTING *under simple walk semantics is #P-complete for the expressions a^* and a^+.*

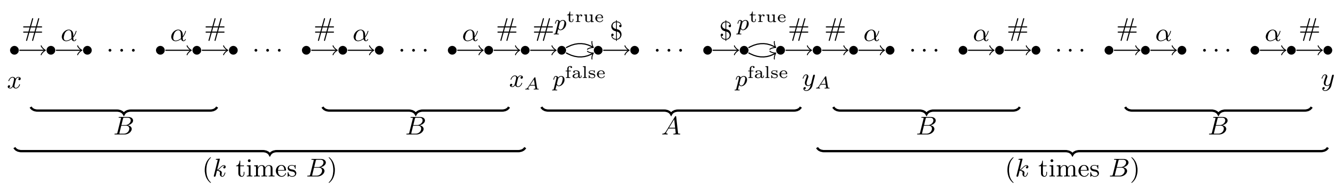


Figure 6: The graph G from the proof of Theorem 5.5.

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 [39]. □

Theorem 5.8 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 5.8, but it is immediate from the observation that the reduction for regular path semantics applies here as well.

Theorem 5.9. 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($\#, !, \bullet$). A proof is provided in the Appendix.

Theorem 5.10. Counting *under simple walk semantics for RE($\#, !, \bullet$) is #P-complete.*

6. 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 5.1.1), 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 6.1. 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($\#, !, \bullet$) in PTIME.

Theorem 6.2. Finiteness *for RE($\#, !, \bullet$) 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 6.3. *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.*

Since, for $r \in$ RE($\neg$), one can test whether $\varepsilon \in L(r)$ in linear time in the size of r by traversing the syntax tree of r, we get the following result, once again by the result of Stockmeyer[38] and Lemma 6.3.

Theorem 6.4. Finiteness *is non-elementary for RE($\neg$).*

7. DISCUSSION

An overview of our results is presented in Table 2. CHAREs are defined in Section 4.2. By "star-free RE", we denote standard regular expressions that do not use the operators "*" and "+". The SPARQL-negation operator "!" is defined in Section 4. By Det-RE, we denote the expressions that can be translated in polynomial time into the DFAs (with single-symbol wildcard) that we defined in Section 5.

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 only becomes larger.[7]

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 [4], which is work conducted independently

[7]For completeness, we provide a Table summarizing data complexity in the Appendix (Table 3).

Problem	Fragment	Regular path semantics	Simple walk semantics
EVALUATION	CHARE($(+a)^*$,$(+a)^+$,$(+w)$,$(+w)?$)	in PTIME	**in PTIME** (4.6)
	star-free RE	in PTIME	**in PTIME** (4.7)
	$(aa)^*$	in PTIME	NP-complete [34]
	RE	in PTIME	**NP-complete**
	RE($\#, !, \bullet$)	**in PTIME** (3.2)	**NP-complete** (4.3)
	RE($\neg$)	**non-elementary** (3.5)	—
	RE($\#, !, \neg, \bullet$)	**non-elementary** (3.6)	—
COUNTING	DFA	**in FPTIME** (5.3)	—
	CHARE(a, $(+a)$)	**in FPTIME**	**in FPTIME** (5.7)
	a^+	**in FPTIME**	#P-complete (5.8,[39])
	a^*	**in FPTIME**	#P-complete (5.8,[39])
	Det-RE	**in FPTIME**	**#P-complete**
	CHARE(a, a^*)	**#P-complete** (5.4,5.5)	**#P-complete**
	CHARE($a, a?$)	**#P-complete** (5.4,5.5)	**#P-complete**(5.9)
	CHARE($a, (+a^+)$)	**#P-complete** (5.4,5.5)	**#P-complete**
	CHARE($a, (+a)^+$)	**#P-complete** (5.4,5.5)	**#P-complete**
	CHARE(a, w^+)	**#P-complete** (5.4,5.5)	**#P-complete**
	CHARE($(+a), a^+$)	**#P-complete** (5.4,5.5)	**#P-complete**
	RE	**#P-complete** (5.4,5.5)	**#P-complete**
	RE($\#, !, \bullet$)	**#P-complete** (5.6)	**#P-complete** (5.10)
	RE($\#, !, \neg, \bullet$)	**#P-complete** (5.6)	—
FINITENESS	RE	**in PTIME** (6.1)	—
	RE($\#, !, \bullet$)	**in PTIME** (6.2)	—
	RE($\neg$)	**non-elementary** (6.4)	—
	RE($\#, !, \neg, \bullet$)	**non-elementary** (6.4)	—

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 irrelevant or not defined. E.g. SPARQL-semantics is not defined for RE($\neg$)-expressions and Finiteness is irrelevant for SPARQL-semantics because the answer is always "yes". 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.

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 paths (cfr. Def. 4.1), 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* [36] 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 [4], 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*.[8] 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.

[8]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 [4] 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.

8. 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] M. Arenas, S. Conca, and J. Pérez. Counting beyond a yottabyte, or how SPARQL 1.1 property paths would prevent the adoption of the standard. Manuscript under submission. Received through personal communication with the authors.

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

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

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

[8] 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.

[9] D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Y. Vardi. Answering regular path queries using views. In *International Conference on Data Engineering (ICDE)*, p. 389–398, 2000.

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

[11] 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.

[12] 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.

[13] D. Colazzo, G. Ghelli, and C. Sartiani. Efficient inclusion for a class of XML types with interleaving and counting. *Inf. Syst.*, 34(7):643–656, 2007.

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

[15] 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.

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

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

[18] 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.

[19] 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.

[20] 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. World Wide Web Consortium, April 2009.

[21] W. Gelade. Succinctness of regular expressions with interleaving, intersection and counting. *Theor. Comput. Sc.*, 411(31–33):2987–2998, 2010.

[22] W. Gelade, M. Gyssens, and W. Martens. Regular expressions with counting: Weak versus strong determinism. In *Mathematical Foundations of Computer Science (MFCS)*, p. 369–381, 2009.

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

[24] S. Harris and A. Seaborne. SPARQL 1.1 query language. Technical report, World Wide Web Consortium, 2010.

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

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

[27] P. Kilpeläinen and R. Tuhkanen. Towards efficient implementation of XML schema content models. In *Symp. on Document Engineering (DOCENG)*, pages 239–241, 2004.

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

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

[30] Y. A. Liu, T. Rothamel, F. Yu, S. D. Stoller, and N. Hu. Parametric regular path queries. In *Programming Language Design and Implementation (PLDI)*, p. 219–230, 2004.

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

[32] 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.

[33] 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.

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

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

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

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

[38] L. Stockmeyer. *The complexity of decision problems in automata theory and logic.* PhD thesis, MIT, 1974.

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

[40] M. Yannakakis. Graph-theoretic methods in database theory. In *Principles of Database Syst.*, p. 230–242, 1990.

APPENDIX

A. SIMPLE WALK SEMANTICS

In its working draft (http://www.w3.org/TR/sparql11-query/#defn_algZeroOrMorePath and http://www.w3.org/TR/ sparql11-query/#defn_algOneOrMorePath, visited on Oct. 28, 2011) the W3C defines the semantics of the Kleene star (*) and Kleene plus ($^+$) operators on graphs through "ZeroOrMorePath" and "OneOrMorePath", respectively. In their definition, they assume that the semantics of the subexpression named "*path*" is already known and then define as follows [24]:

- ZeroOrMorePath: An arbitrary length path $P = (X\ (path)^*\ Y)$ is all solutions from X to Y by repeated use of *path* such that any nodes in the graph are traversed once only. ZeroOrMorePath includes X.
- OneOrMorePath: An arbitrary length path $P = (X\ (path)^+\ Y)$ is all solutions from X to Y by repeated use of path 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.

In the definition of ZeroOrMorePath, the W3C seems to require that the path from X to Y is a *simple path*, as we defined it in Section 4. However, in the definition of OneOrMorePath, simple cycles also seem to be allowed. Therefore, we find the informal definition of the W3C to be unclear on the matter of whether it allows simple cycles. Examples on the W3C pages seem to suggest that simple cycles are allowed. We therefore chose in the body of the paper to consider *simple walks*.

B. MISSING PROOFS

This appendix contains some material for which there was no room in the body of the paper.

We denote by $\boxed{\cdots}$ the positions where we continue proofs from the body of the paper.

Proofs for Section 3

Proof of Theorem 3.2: Evaluation *for RE(#, •) is in polynomial time.*

Proof. We prove that the dynamic programming algorithm can be implemented to decide Evaluation for RE(#, •) in polynomial time. That is, given a graph (G, x, y) and RE(#, •)-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. In particular, we prove that the following invariant holds for every relation R_s that is calculated:

$$\text{For every pair } (u,v) \in V \times V : \big((u,v) \in R_s \Leftrightarrow \exists \text{ path } p \text{ in } G \text{ from } u \text{ to } v : \text{lab}(p) \in L(s)\big). \qquad \text{(I)}$$

$\boxed{\cdots}$ If (I) is correct, then Evaluation for (G, x, y) and r is true if and only if $(x, y) \in R_r$. Therefore it is sufficient that we show that (I) holds for every such relation R_s.

We prove the invariant by structural induction on the syntax tree of r. First, we treat the base cases. These are subexpressions s that correspond to leaves in the syntax tree, i.e., $s = \varepsilon$, $s = a$, and $s = \bullet$.

If $s = \varepsilon$ then our algorithm computes $R_s = \{(u, u) \mid u \in V\}$. Notice that, for all nodes u and v, there exists a path p in G from u to v with $\text{lab}(p) \in L(\varepsilon)$ if and only if p has length zero (and therefore $u = v$). Therefore, condition (I) is fulfilled.

If $s = a$, then our algorithm computes $R_s = \{(u, v) \mid (u, a, v) \in E\}$. For all nodes u and v there exists a path p in G from u to v with $\text{lab}(p) = a$ if and only if p is an edge from u to v labeled with a. Thus, condition (I) is fulfilled.

If $s = \bullet$, then our algorithm computes $R_s = \{(u, v) \mid \exists a \in \Delta \text{ s.t. } (u, a, v) \in E\}$. There exists a path p in G from u to v that matches some symbol if and only if there exists an edge between u and v in G. Therefore, (I) is fulfilled.

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

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

If $s = s_1 \cdot s_2$, then our algorithm computes

$$R_s := \pi_{1,3}(R_{s_1} \underset{R_{s_1}.2=R_{s_2}.1}{\bowtie} R_{s_2}),$$

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} and $\pi_{1,3}$ denotes the projection of these triples onto the leftmost and rightmost column. Take an arbitrary $(u, v) \in V \times V$. First we show that, if (u, v) in R_s, then there exists a path p from u to v in G such that $\text{lab}(p) \in L(s)$. By the definition of the join operator, we know that for every tuple $(u, v) \in R_s$ there exist tuples (u, w) and (w, v), such that $(u, w) \in R_{s_1}$ and $(w, v) \in R_{s_2}$. Because by induction R_{s_1} and R_{s_2} are calculated correctly, there exists a path p_1 from u to w in G with $\text{lab}(p_1) \in L(s_1)$ and a path p_2 from w to v with $\text{lab}(p_2) \in L(s_2)$. Thus the concatenation of these two paths $p = p_1.p_2$, is a path from u to v in G with $\text{lab}(p) \in L(s_1 \cdot s_2)$. Second, we show that, if there exists a path p from u to v in G with $\text{lab}(p) \in L(s_1 \cdot s_2)$, then (u, v) in R_s. Therefore, let p be such a path. Since $\text{lab}(p) \in L(s_1 \cdot s_2)$, we know that there exists paths p_1 and p_2, such that $p = p_1.p_2$ and p_1 is a path from u to some node w with $\text{lab}(p_1) \in L(s_1)$ and p_2 is a path from w to v with $\text{lab}(p_2) \in L(s_2)$. Because R_{s_1} and R_{s_2} are calculated correctly by induction, we know that $(u, w) \in R_{s_1}$ and $(w, v) \in R_{s_2}$. Thus the tuple (u, v) is in R_s by the definition of the join and R_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.

If $s = s_1^*$, then our algorithm computes R_s as the reflexive and transitive closure of R_{s_1}. Let (u, v) be an arbitrary pair in $V \times V$. By construction, we have that $(u, v) \in R_s$ if and only if (u, v) is in the reflexive and transitive closure of R_{s_1}. By definition of reflexive and transitive closure, this either means that $u = v$, $(u, v) \in R_{s_1}$, or there exist nodes $v_1, \ldots, v_k$ with $k \geq 1$ such that $(u, v_1), (v_1, v_2), \ldots, (v_k, v) \in R_{s_1}$. In the first case, we have that (u, v) matches ε and therefore there is a path from u to v that matches s. In the second case, there is a path p from u to v that matches s_1 (and therefore also s) by the induction hypothesis. Finally, in the third case, we know, by induction, that, there are paths

- p_0 from u to v_1 that matches s_1;
- p_i from v_i to v_{i+1} that match s_1, for every $i = 1, \ldots, k - 1$; and
- p_k from v_k to v that match s_1.

Therefore, the path $p = p_0 p_1 \cdots p_k$ matches $s = s_1^*$. Conversely, if there is a path p from u to v that matches $s = s_1^*$, then, by definition of the Kleene star, we either have that $u = v$ or there exists a $k \geq 0$ and paths $p_0, p_1, \ldots, p_k$ with $p = p_0 p_1 \cdots p_k$ such that, for each $i = 0, \ldots, k$, p_i matches s_1. By definition of reflexive and transitive closure, and since R_{s_1} is computed correctly by induction, we then have that $(u, v) \in R_s$. Therefore, R_s fulfills (I).

If $s = s_1^+$, then our algorithm computes the transitive closure of R_{s_1}. The proof is analogous to the proof of the case $s = s_1^*$.

The last three cases concern the numerical expressions. If $s = s_1^k$, then our algorithm computes

$$R_s := \{(u, v) \mid (M_{s_1})^k[u, v] \neq 0\},$$

where $M_{s_1}[u, v] = 1$ if there exists a path p from u to v with $\text{lab}(p) \in L(s_1)$ and $M_{s_1}[u, v] = 0$ otherwise. When we interpret the relation R_{s_1} as a Boolean $|V| \times |V|$ matrix by setting $M_{s_1}[u, v] = 1$ if and only if $(u, v) \in R_{s_1}$, then we can obtain the matrix $M_{s_1 \cdot s_1}$ for $R_{s_1 \cdot s_1}$ by computing $M' = M_{s_1} \cdot M_{s_1}$ and setting $M_{s_1 \cdot s_1}[u, v] := \min(1, M'[u, v])$ for every $(u, v) \in V \times V$ [6]. Since we can write s_1^k as k concatenations of s_1 and R_{s_1} is calculated correctly by induction, (I) follows directly from the concatenation case.

If $s = s_1^{k,\infty}$, then our algorithm computes R_s as the relation for the expression $s_1^k \cdot s_1^*$. Because by induction R_{s_1} is calculated correctly, (I) follows directly from the s_1^k case, the concatenation case, and the Kleene star case.

If $s = s_1^{k,\ell}$, then our algorithm computes $R_s := \{(u, v) \mid M_s[u, v] \neq 0\}$ with $M_s := (M_{s_1})^k \cdot (M'_{s_1})^{\ell-k}$ where M_{s_1} is the matrix representation of s_1 (as in the $s = s_1^k$ case) and M'_{s_1} is the matrix obtained from M_{s_1} by additionally setting $M'_{s_1}[u, v] := 1$ if $u = v$. Hence, M' is the matrix representation of $R_{(s_1+\varepsilon)}$. Correctness again follows analogously to the $s = s_1^k$ case by the equivalence of the expressions s and $s_1^k \cdot (s_1 + \varepsilon)^{k-\ell}$, and the correctness of the concatenation case and the $s = s_1^k$ case. Therefore (I) holds also for the last case.

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 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) ([6], 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. $\square$

Proof of Theorem 3.3: Membership *for RE($\#, \neg, \bullet$) is in polynomial time.*

Proof. $\boxed{\cdots}$ To simplify the technical presentation in this proof, we abstract a string 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 3.2. We already showed in the proof of Theorem 3.2 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 dynamic programming algorithm can be extended to also take negation into account, if we evaluate over strings instead of graphs. The change in our algorithm consists of considering one extra case:

- If $s = \neg s_1$, then $R_s := \{(u, v) \mid (u, v) \notin R_{s_1}\}$.

Clearly, this case can also be implemented to run in polynomial time.

We prove that the algorithm is correct. To this end, let r be an expression from RE($\#, \neg, \bullet$). We prove the same invariant than in the proof of Theorem 3.2, that is, for each subexpression s of r:

$$\text{For every pair } (u, v) \in V \times V : \big((u, v) \in R_s \Leftrightarrow \exists \text{ path } p \text{ in } G \text{ from } u \text{ to } v : \text{lab}(p) \in L(s)\big). \qquad \text{(I)}$$

Note that we are considering only words now. That means, in the following, a tuple (u, v) represents two positions in the string, such that $(u, v) \in R_s$ if and only if the substring 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 3.2. So, it only remains to consider the following case:

If $s = \neg s_1$, then our algorithm computes $R_s = \{(u,v) \mid (u,v) \notin R_{s_1}\}$. By definition of the negation operator and since we are dealing with a string, the (unique) path from u to v matches $\neg s_1$ if and only if it does not match s_1. Since by induction R_{s_1} is calculated correctly, a tuple (u,v) is in R_{s_1} if and only if the substring from u to v in the considered word matches $L(s_1)$. Then $(u,v) \in R_s$ if and only if $(u,v) \notin R_{s_1}$. Therefore R_s fulfills (I). □

Proofs for Section 4

Proof of Theorem 4.3: Evaluation *under simple walk semantics is NP-complete for RE(#, !, •)-expressions.*

Proof. The NP lower bound is immediate from Theorem 4.2.

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. $\boxed{\cdots}$ 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.

In particular, the NP algorithm takes the syntax tree of r, traverses it in bottom-up fashion, and guesses a polynomial amount of information for each node that corresponds to a subexpression s.

The type of information that the algorithm guesses depends on the type of the node. We discuss all possible cases next.

- If s is a Δ-symbol, $s = \varepsilon$, or $s = \bullet$, then R_s is defined exactly as in the dynamic programming algorithm in Section 3.
- If $s = !(a_1 + \cdots + a_n)$ for $a_1, \ldots, a_n \in \Delta$, then $R_s := \{(u,v) \mid (u,a,v) \in E \text{ and } a \notin \{a_1, \ldots, a_n\}\}$.
- If $s = s_1^*$ or if $s = s_1^+$ then, for each pair (u,v) of nodes of G, we compute the following. First, we test whether v is reachable from u. If v is reachable from u, then we guess a simple walk $p = uv_1 \cdots v_{n-1}v$ from u to v. We then test whether $\text{lab}(p) \in L(s)$. We define $R_s := \{(u,v) \mid v \text{ is reachable from } u \text{ and } \text{lab}(p) \in L(s)\}$.
- If $s = s_1 \cdot s_2$ or if $s = s_1 + s_2$, then R_s is computed from R_{s_1} and R_{s_2} exactly as in Section 3.
- If $s = s_1?$, then $R_s := R_{s_1} \cup \{(u,u) \mid u \in V\}$.
- If $s = s_1^k$, then consider the connectivity matrix M_{s_1} of pairs that are in R_{s_1}. 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. 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 $s = s_1^{k,\ell}$ and $\ell \neq \infty$, then let M_{s_1} be the same matrix 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 $u = v$. Let $M_s := (M_{s_1})^k \cdot (M'_{s_1})^{\ell-k}$. Then, $R_s := \{(u,v) \mid M_s[u,v] \neq 0\}$.
- If $s = s_1^{k,\infty}$, then R_s is the relation for the expression $s_1^k \cdot s_1^*$.

Finally, we accept if and only if R_r contains the pair (x,y).

We want to argue correctness similar to the proof of Theorem 3.2. In particular, we have proved for Theorem 3.2 that the following invariant holds for every subexpression s with corresponding relation R_s:

$$\text{For every pair } (u,v) \in V \times V : \big((u,v) \in R_s \Leftrightarrow \exists \text{ path } p \text{ in } G \text{ from } u \text{ to } v : \text{lab}(p) \in L(s)\big). \qquad \text{(I)}$$

Here we prove the following statements:

$$\text{There is a run of our algorithm, such that, for every subexpression } s \text{ of } r,\ R_s \text{ fulfills (I).} \qquad \text{(I1)}$$

and

for every run of our algorithm and every subexpression s of r :
$$\forall (u,v) \in V \times V : \big((u,v) \in R_s \Rightarrow \exists \text{ path } p \text{ in } G \text{ from } u \text{ to } v : \text{lab}(p) \in L(s)\big). \qquad \text{(I2)}$$

Notice that (I1) implies that our algorithm is sound and that (I2) implies that it is complete.

The proof of (I2) is very similar to the proof of Theorem 3.2, with a slightly altered induction hypothesis.

We prove (I1) in more detail, by structural induction on the syntax tree of r. First we treat the base cases. These are subexpressions s that correspond to leaves in the syntax tree, i.e., $s = a$, $s = \varepsilon$ and $s = \bullet$.

If s is an Δ-symbol, $s = \varepsilon$, or $s = \bullet$, then our algorithm is defined exactly as in the dynamic programming algorithm in Section 3. In the proof of Theorem 3.2 we have shown that (I) holds for every relation R_s. Since our algorithm runs deterministically in this step, s fulfills (I1).

If $s = !(a_1 + \cdots + a_n)$ for $a_1, \ldots, a_n \in \Delta$, then our algorithm computes $R_s := \{(u,v) \mid (u,a,v) \in E \text{ and } a \notin \{a_1, \ldots, a_n\}\}$. The !-operator is defined by the same semantics as the $\neg$-operator. Therefore, for an arbitrary tuple $(u,v) \in R_s$, it holds that $(u,v) \in R_s$ if and only if there exists an edge $e = (u,a,v)$ in G, such that $a \notin L(a_1 + \cdots + a_n)$. It follows that $R_s = \{(u,v) \mid (u,a,v) \in E \text{ and } a \notin \{a_1, \ldots, a_n\}\}$. Thus R_s fulfills (I). Since R_s is calculated deterministically, s fulfills (I1).

We now prove the induction cases. To this end, let s be a node in the syntax tree such that (I1) is already known to be true for all of its children. Again, we make the case distinction based on the type of subexpression that s represents.

If $s = s_1^*$ or if $s = s_1^+$, then our algorithm guesses a simple walk $p = uv_1 \cdots v_{n-1}v$ for every pair of nodes (u,v), such that v is reachable from u. Only if $\text{lab}(p) \in L(s)$, the tuple (u,v) is added to R_s. We know that there exists a run of the algorithm such that R_{s_1} is calculated correctly by induction. This means that, in this run, if we guess the right information, then R_s fulfills (I). Then there exists a run for s, such that R_s fulfills (I) and, therefore, s fulfills (I1).

If $s = s_1 + s_2$ or $s = s_1.s_2$, then our algorithm computes R_s as it is defined in Section 3. By the proof of Theorem 3.2, we know that R_s is calculated correctly from R_{s_1} and R_{s_2}, if they fulfill (I). By induction we know that there exists a run, such that R_{s_1} and R_{s_2} fulfill (I). Therefore there exists a run, such that R_s fulfills (I). Thus s fulfills (I1).

If $s = s_1?$, then our algorithm computes $R_s := R_{s_1} \cup \{(u,u) \mid u \in V\}$. By definition it holds that $s_1? = (s_1 + \varepsilon)$. Therefore, from the correctness of the cases $s = s_1 + s_2$ and $s = \varepsilon$, it follows that R_s fulfills (I1).

If $s = s_1^k$, $s = s_1^{k,l}$, or $s = s_1^{k,\infty}$, then it follows, analogously to the cases $s = s_1 + s_2$ and $s = s_1 \cdot s_2$, that R_s fulfills (I1). $\square$

In the following proof, we need the notion of a *match*.

Definition B.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 mapping from pairs* (i,j), $0 \le i \le j \le n$, *of nodes of* p *to sets of subexpressions 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* $i+1 = j$ *and* $a_j = a$;

(3) if $r? \in m(i,j)$, *then* $r \in m(i,j)$ *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 $r^* \in m(i,j)$, *then there exists a* $t \in \mathbb{N}$ *and numbers* $k_1, \ldots, k_t$ *such that,* $r \in m(i,k_1)$, $r \in m(k_t,j)$ *and* $r \in m(k_\ell, k_{\ell+1})$, *for all* ℓ *such that* $1 \le \ell < t$;

(7) if $r^+ \in m(i,j)$, *then there exists a* $t \ge 1$ *and numbers* $k_1, \ldots, k_t$ *such that,* $r \in m(i,k_1)$, $r \in m(k_t,j)$ *and* $r \in m(k_\ell, k_{\ell+1})$, *for all* ℓ *such that* $1 \le \ell < t$.

Furthermore, m *has to be minimal with these properties. That is, if* m' *fulfills (1)–(5) and* $m'(i,j) \subseteq m(i,j)$ *for each* i,j, *then* $m' = m$. *We say that* m matches *a subpath* $v_{i-1}[a_i] \cdots [a_j]v_j$ *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* $v_{i-1}[a_i] \cdots [a_j]v_j$ *matches* r'.

Proof of Theorem 4.6: Evaluation *under simple walk semantics for CHARE*$((+a)^*, (+a)^+, (+w), (+w)?)$ *is in PTIME.*

Proof. This theorem immediately 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.

$\boxed{\cdots}$

This can be seen as follows. When a path p matches a CHARE$((+a)^*, (+a)^+, (+w), (+w)?)$ under simple walk semantics, then it trivially also matches the CHARE under regular path semantics. Conversely, let p be a path that matches a CHARE$((+a)^*, (+a)^+, (+w), (+w)?)$ r under regular path semantics. Let m be a match between p and r. Let p_0 be an arbitrary subpath of p such that m matches p_0 onto a factor of the form $(+a)^*$. Suppose that p_0 is of the form $u_0[a_1]u_1 \cdots u_{k-1}[a_k]u_k$. If p_0 has a loop, then let i be the smallest number such that $u_i = u_j$ for some $j > i$. Note that also the path $p_0' = u_0[a_1]u_1 \cdots u_i[a_{j+1}]u_{j+1} \cdots u_{k-1}[a_k]u_k$ matches the factor of the form $(+a)^*$. The argument for factors of the form $(+a)^+$ is similar: the difference is that we do not need to cut away anything if p_0 is a simple cycle. This shows that, for each path p that matches r under regular path semantics, we can cut away loops from p that are not allowed in simple walk semantics, thereby obtaining a path p' that matches r under simple walk semantics.

We note that the above argument only holds for simple walk semantics that allows simple cycles to match expressions of the form r^+.

However, a more general argument shows that also in the case where simple cycles are *not* allowed, Evaluation is in PTIME. Indeed, we can compute, for every factor f of the form $(+w)$ or $(+w)?$, a $|V| \times |V|$ matrix M_f such that, for each pair of nodes (u,v), we have that $M_f[u,v] = 1$ if v is reachable from u by a path p that matches f; and $M_f[u,v] = 0$ otherwise. Similarly, we can compute, for each factor f of the form $(a_1 + \cdots + a_n)^*$ a matrix M_f for which $M_f[u,v] = 1$ if v is reachable from u by a path that only contains symbols from $\{a_1, \ldots, a_n\}$; and $M_f[u,v] = 0$ otherwise. Finally, for each factor f of the form $(a_1 + \cdots + a_n)^+$ we can compute a matrix M_f for which $M_f[u,v] = 1$ if v is reachable from u by a path of length at least one that only contains symbols from $\{a_1, \ldots, a_n\}$; and $M_f[u,v] = 0$ otherwise. Let $r = f_1 f_2 \cdots f_m$. Then there exists a path from u to v that matches r if and only if $((\cdots(M_{f_1} M_{f_2}) \cdots M_{f_{m-1}}) M_{f_m})[u,v] \ge 1$. $\square$

Proofs for Section 5

Proof of Lemma 5.1: *If* A *is a DFA, then the mapping* 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 the product* $G^{x,y} \times A$.

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

$P(G)$: the set of paths from x to y in G that match A; to

$P(G \times A)$: the set 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.

Let $A = (Q, \Sigma, \Delta, \delta, q_0, Q_f)$.

First, we show that φ_{PATHS} is surjective. To this end, 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 with $(y, q_n) \in Y_{G,A}$ in $P(G \times A)$. 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 φ_{PATHS}, we have that $\varphi_{\text{PATHS}}(x) = (x, q_0)$.

Since $\big((x, q_0), a_1, (v_1, q_1)\big)$ is an edge in $G^{x,y} \times A$, we know (by definition of $G^{x,y} \times A$) that there is an edge (x, a_1, v_1) in G and

- a transition $\delta(q_0, a_1) = q_1$ in A (if $a_1 \in \Sigma$) or
- a transition $\delta(q_0, \bullet) = q_1$ in A (if $a_1 \notin \Sigma$).

By definition of φ_{PATHS}, we then have that $\varphi_{\text{PATHS}}((x, a_1, v_1)) = \big((x, q_0), a_1, (v_1, q_1)\big)$. It now follows by a straightforward induction on the length of p that $\varphi_{\text{PATHS}}(p^G) = p$. Finally, since $p^G \in P(G)$, we have that φ_{PATHS} is surjective from $P(G)$ to $P(G \times A)$.

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 $\varphi_{\text{PATHS}}(p) = (x, q_0)[a_1](v_1, q_1)[a_2](v_2, q_2) \cdots (v_{n-1}, q_{n-1})[a_n](y, q_n)$. By definition, $\varphi_{\text{PATHS}}(x) = (x, q_0)$. Furthermore, since φ_{PATHS} preserves the labels on edges, we have that $a_i = a'_i$ for every $i = 1, \ldots, n$. Since, for every node $v \in G$ we have that $\varphi_{\text{PATHS}}(v) \in \{v\} \times Q$ and since $\varphi_{\text{PATHS}}(v_1) = \varphi_{\text{PATHS}}(v'_1)$, we also know that $v_1 = v'_1$. It now follows by straightforward induction on the lengths of p and p' that $p = p'$. □

We can obtain a result similar to Lemma 5.1 for unambiguous NFAs. An NFA $A = (Q, \Sigma, \Delta, \delta, q_0, Q_f)$ is *unambiguous*, when, for each word w in $L(A)$, there exists exactly one accepting run of A on w.

However, since non-determinism is involved, we need to define the mapping φ_{PATHS} in a slightly different way. Let $G^{x,y} \times A$ denote the product of (V, E, x, y) and A, as defined in the body of the paper. Again, we formalize the correspondence between paths from $q_0^{G,A}$ to $Q_f^{G,A}$ in $G^{x,y} \times A$ and paths from x to y that match A in G 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 such 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 string $a_1 \ldots a_n$. 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.

LEMMA B.2. *If A is an unambiguous NFA, 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

$P(G)$: the set of paths from x to y in G that match A; to

$P(G \times A)$: the set 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.

Let $A = (Q, \Sigma, \Delta, \delta, q_0, Q_f)$ be unambiguous.

First, we show that φ_{PATHS} is surjective. To this end, 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 $P(G \times A)$, i.e., $(y, q_n) \in Y_{G,A}$. 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 $q_0 \cdots q_n$ in 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

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

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 $p = x[a_1]v_1[a_2]v_2 \cdots v_{n-1}[a_n]y = p'$. □

PROOF OF THEOREM 5.5 (CORRECTNESS OF THE REDUCTION): 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 5.4. We prove the lower bounds by reductions from #DNF. 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 (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

> 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.

Here, the subgraph α of B is itself also a path which we will instantiate differently in each of the cases (1)–(3). By the second point above we mean that all paths from x to y will enter A through the node x_A and leave A through y_A

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 α, there is a path p_α 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 α_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 α_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.

$\boxed{\cdots}$ 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:

$$\text{lab}(\alpha) \in L(r^{\text{false}}) \cap L(r^{\text{true}}) \quad \text{(P1)}$$

$$\text{lab}(p^{\text{true}}) \in L(r^{\text{true}}) - L(r^{\text{false}}) \quad \text{(P2)}$$

$$\text{lab}(p^{\text{false}}) \in L(r^{\text{false}}) - L(r^{\text{true}}) \quad \text{(P3)}$$

$$\{\text{lab}(p^{\text{true}}), \text{lab}(p^{\text{false}}), \text{lab}(\alpha)\} \subseteq L(r^{\text{all}}) \quad \text{(P4)}$$

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 B.3. *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 B.3. 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 $V_w \models C$, $V_w(x_j) = \text{false}$ and 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 $\text{lab}(p^{\text{true}})$. By condition (P2), $w_j \notin L(r^{\text{false}})$. Hence, $w_V \notin L(F(C))$.

This concludes the proof of Claim B.3. $\square$

We now fill in some details to complete the reductions for the cases (1)–(3).

(1) We instantiate the paths and subexpressions as follows:

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

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

- $\text{lab}(\alpha) = aa$
- $\text{lab}(p^{\text{true}}) = aaa$
- $\text{lab}(p^{\text{false}}) = a$
- 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

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

- $r^{\text{all}} = a^+$

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 B.4. *Let w be a label of a path from x_A to y_A in A. Then*

(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 B.4. 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 strings u, w, and every string in every $L(F_i)$ is of the form $\#y\#$ where y is a non-empty string over the alphabet $\{a, b, \$\}$. Also, $L(N)$ only contains strings of the form $\#y_1\#\#y_2\# \cdots \#y_\ell\#$, where $y_1, \ldots, y_\ell$ are non-empty strings over $\{a, \$\}$ (and, possibly, subsequences thereof). Hence, as $u^k w u^k \in L(r)$ and as none of the strings $y, y_1, \ldots, y_\ell$ 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 string $u^k w u^k$ matches r. Therefore, we use the notion of a *match* (Definition B.1). For a path p_0 and regular expression r, we also say that the string lab(p_0) *matches* r if p_0 matches r.

Let m be a match between p and r. Notice that 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 superstring of $u^k w$ to the left occurrence of N in r. Note that $u^k w$ is a string of the form $\#y'_1\#\#y'_2\# \cdots \#y'_{k+1}\#$, where $y'_1, \ldots, y'_{k+1}$ are strings over $\{a, b, \$\}$. But, since $\ell \leq k$, no superstring of $u^k w$ can match N, which is a contradiction. Analogously, no superstring 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 superstring of $u^k w$ onto the left occurrence of the expression N in r. Observe that every string 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 superstring 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 B.4. $\square$

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

Claim B.5. *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 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 B.4 and B.3 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. $\square$

Thus the reduction used in the proof of Theorem 5.5 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 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 $j = 1, \ldots, n$. 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 7 containing the paths, $p_1^{\text{true}}, p_1^{\text{false}}, \ldots, p_n^{\text{true}}, p_n^{\text{false}}$;
- B is a path, consisting of n concatenations of the subpath α; and

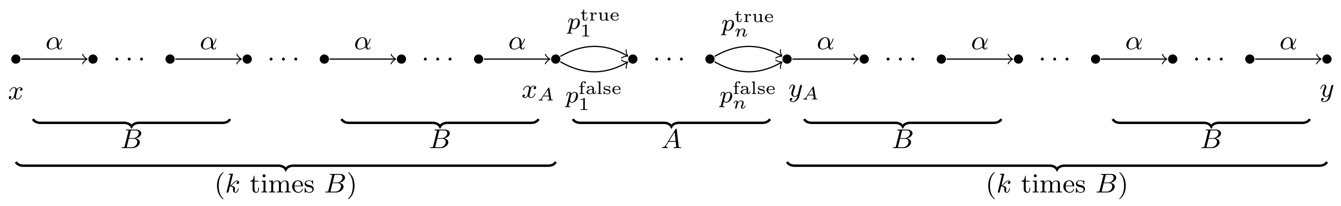


Figure 7: The graph G from the proof of Theorem 5.5 for the cases (4)–(6).

- $F(C)$ is defined as

  $$e_1 \cdots e_n,$$

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

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

Furthermore The new structure of graph G is also shown in Figure 7.

(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 strings 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 B.3 and B.4 hold. For Claim B.3 this can be done similarly as before, and for Claim B.4 (a) and (b) it is again easy to see. We complete the proof of Theorem 5.5 by showing Claim B.4 (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 string w is of the form $y_1 \cdots y_n$, where, for every $j = 1, \ldots, n$, y_j is a string in Σ_j^+. Moreover, no strings in $L(N)$ contain symbols from Σ_j for any $j = 1, \ldots, n$. Hence, m cannot match any symbol of the string w onto N. Consequently, m matches the entire string w onto a subexpression of $F_1 \cdots F_k$ in r.

Further, observe that every string in every F_i, $i = 1, \ldots, k$, is of the form $y'_1 \ldots y'_n$, where each y'_j is a string 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 Claim B.3 and B.4 hold for the fragments (4)–(6), the proof of $(*)$ is analogous to the proof of Claim B.5.

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 5.5. □

Proof of Theorem 5.6: Counting *for RE(#, ¬, •) is #P-complete.*

Proof. The lower bound follows from Theorem 5.5. Since Membership for RE(#, ¬, •) is in polynomial time by Theorem 3.3 and since the number max is given in unary, the upper bound follows analogously to the proof in Theorem 5.4. □

Proof of Theorem 5.8: 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 [39]. 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. □

Proof of Theorem 5.10: Counting *under simple walk semantics for RE(#, !, •) 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 3.3) 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 R_{s^*} such that the subpath from x to y is not a simple walk, are removed from the relation. 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. □

Proofs for Section 6

Proof of Theorem 6.2: Finiteness *for RE(#, •) is in PTIME.*

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. This can be seen as follows: if we translate the RE(#, •) expression r to a normal RE(•) 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 N be an NFA for $L(r)$. Again, N can be constructed in time exponential in the size of r. If we consider the product $G^{x,y} \times N$ (defined in the paper), 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 $x_{G,N}$ to some node in $Y_{G,N}$. The latter holds if and only if there exists a path from $x_{G,N}$

to some node in $Y_{G,N}$ of length at least $|G| \cdot |N| + 1$, due to a pumping argument. Since the size $|N|$ of N is exponential in r, we therefore only need to consider lengths of paths that are exponential in the input.

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

The dynamic programming algorithm works as follows. Let r be an RE(#, •)-expression and let $G = (V, E)$ be a graph. We store, for each node in the syntax tree with associated subexpression s, a ternary relation $R_s \subseteq V \times V \times \{0, \ldots, |G| \cdot |N| + 1\}$ such that all of the following hold:

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

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 $s = a$ with $a \in \Delta$, then $R_s := \{(u, v, 1) \mid (u, a, v) \in E\}$.
- If $s = \varepsilon$, then $R_s := \{(u, u, 0) \mid u \in V\}$.
- If $s = \bullet$, then $R_s := \{(u, v, 1) \mid \exists a \in \Delta \text{ s.t. } (u, a, v) \in E\}$.
- If $s = s_1 + s_2$, then R_s represents the union of R_{s_1} and R_{s_2} where we remember the longest paths. That is, R_s is the set of all triples of the form (u, v, i) such that
  - $(u, v, k) \in R_{s_1}$ and $(u, v, \ell) \in R_{s_2}$ and $i = \min(\max(k, \ell), |G| \cdot |N| + 1)$, or
  - $(u, v, i) \in R_{s_1}$, $\nexists j.\ (u, v, j) \in R_{s_2}$, or
  - $(u, v, i) \in R_{s_2}$, $\nexists j.\ (u, v, j) \in R_{s_1}$.
- If $s = s_1 \cdot s_2$, then R_s is the set of all triples (u, v, i) such that there is a node z such that $(u, z, k) \in R_{s_1}$, $(z, v, \ell) \in R_{s_2}$, and $i = \min(k + \ell, |G| \cdot |N| + 1)$.
- If $s = s_1?$, then R_s is the relation for the expression $(s_1 + \varepsilon)$.
- If $s = s_1^k$, then we perform fast exponentiation, while keeping track of the maximal lengths of paths similarly as in the concatenation case. For computing the relation $R_{s_1^j}$ for some $j \in \mathbb{N}$, we can recursively compute the following:
  - If $j = 1$, then $R_{(s_1^j)} = R_{s_1}$.
  - Otherwise, if j is even, then $R_{(s_1^j)} = R_{(s_1^{j/2} \cdot s_1^{j/2})}$.
  - Otherwise, if j is odd, then $R_{(s_1^j)} = R_{(s_1 \cdot s_1^{j/2} \cdot s_1^{j/2})}$.

  Of course, we store every relation that we compute, in order to avoid recomputations. The relation R_s is then obtained by calling the above procedure with $j = k$.
- If $s = s_1^*$, then R_s is the relation for the expression $(s_1?)^k$, for $k = |G| \cdot |N| + 1$.
- If $s = s_1^+$, then R_s is the relation for the expression $s_1 \cdot s_1^*$.
- 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 R_s is the relation for the expression $s_1^k \cdot (s_1?)^{\ell - k}$.

Finally, if the input for GRAPHEVAL is G, nodes x and y, and RE(#, •)-expression r, we return the answer "true" if and only if R_r contains the pair $(x, y, |G| \cdot |N| + 1)$.

The proof of correctness is a straightforward induction that follows the same lines as the proof of Theorem 3.2. □

PROOF OF LEMMA 6.3: *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.*

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

Problem	Fragment	Regular path semantics	simple walk semantics
EVALUATION	CHARE($(+a)^*$,$(+a)^+$,$(+w)$,$(+w)?$)	in PTIME	in PTIME
	star-free RE	in PTIME	in PTIME
	$(aa)^*$	in PTIME	NP-complete [34]
	RE	in PTIME	NP-complete
	RE($\#, !, \bullet$)	in PTIME	NP-complete
COUNTING	DFA	in FPTIME	—
	CHARE(a, $(+a)$)	in FPTIME	in FPTIME
	a^+	in FPTIME	#P-complete [39]
	a^*	in FPTIME	#P-complete [39]
	Det-RE	in FPTIME	#P-complete
	CHARE(a, a^*)	in FPTIME	#P-complete
	CHARE($a, a?$)	in FPTIME	in FPTIME
	CHARE($a, (+a^+)$)	in FPTIME	#P-complete
	CHARE($a, (+a)^+$)	in FPTIME	#P-complete
	CHARE(a, w^+)	in FPTIME	#P-complete
	CHARE($(+a), a^+$)	in FPTIME	#P-complete
	RE	in FPTIME	#P-complete
	RE($\#, !, \bullet$)	in FPTIME	#P-complete
FINITENESS	RE	in PTIME	—
	RE($\#, !, \bullet$)	in PTIME	—

Table 3: Our results, when interpreted under data complexity.