Conjunctive Regular Path Queries with String Variables

Markus L. Schmid
MLSchmid@MLSchmid.de
Humboldt-Universität zu Berlin
Germany

ABSTRACT

We introduce the class CXRPQ of conjunctive xregex path queries, which are obtained from conjunctive regular path queries (CRPQs) by adding string variables (also called backreferences) as found in practical implementations of regular expressions. CXRPQs can be considered user-friendly, since they combine two concepts that are well-established in practice: pattern-based graph queries and regular expressions with backreferences. Due to the string variables, CXRPQs can express inter-path dependencies, which are not expressible by CRPQs. The evaluation complexity of CXRPQs, if not further restricted, is PSpace-hard in data complexity. We identify three natural fragments with more acceptable evaluation complexity: their data complexity is in NL, while their combined complexity varies between ExpSpace, PSpace and NP. In terms of expressive power, we compare the CXRPQ-fragments with CRPQs and unions of CRPQs, and with extended conjunctive regular path queries (ECRPQs) and unions of ECRPQs.

CCS CONCEPTS

• **Theory of computation** → *Database query languages (principles)*; *Regular languages.*

KEYWORDS

Graph Databases; Conjunctive Regular Path Queries

ACM Reference Format:
Markus L. Schmid. 2020. Conjunctive Regular Path Queries with String Variables. In *Proceedings of the 39th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems (PODS'20), June 14–19, 2020, Portland, OR, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3375395.3387663

1 INTRODUCTION

The popularity of graph databases (commonly abstracted as directed, edge-labelled graphs) is due to their applicability in a variety of settings where the underlying data is naturally represented as graphs, e. g., Semantic Web and social networks, biological data, chemical structure analysis, pattern recognition, network traffic, crime detection, object oriented data. The problem of querying graph-structured data has been studied over the last three decades and still receives a lot of attention. For more background information, we refer to the introductions of the recent papers [8, 9, 18, 34], and to the survey papers [4–6, 45].

Many query languages for graph databases (for practical systems as well as those studied in academia) follow an elegant and natural declarative approach: a query is described by a *graph pattern*, i. e., a graph $G = (V, E)$ with edge labels that represent some path-specifications. The evaluation of such a query consists in *matching* it to the graph database $\mathcal{D} = (V_\mathcal{D}, E_\mathcal{D})$, i. e., finding a mapping $h : V \to V_\mathcal{D}$, such that, for every $(x, s, y) \in E$, in $\mathcal{D}$ there is a path from $h(x)$ to $h(y)$ whose edge labels satisfy the path-specification s. In the literature, such query languages are also called *pattern-based*. Let us now briefly summarise where this concept can be found in theory and practice.

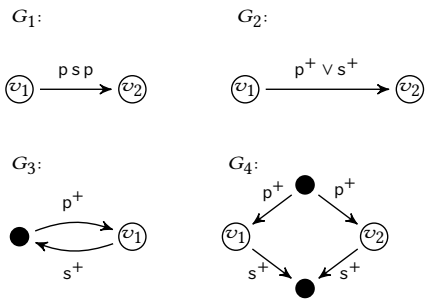

Figure 1: Examples of simple graph patterns.

The most simple graph-patterns (called *basic* in [4]) have just fixed relations (i. e., edge-labels) from the graph database as their edge labels. A natural extension are *wildcards*, which can match any edge-label of the database (e. g., as described in [18]), or *label variables*, which are like wildcards, but different occurrences of the same variable must match the same label (see, e. g., [9]). It is common to extend such basic graph patterns with relational features like, e. g., projection, union, and difference (see [4]). In order to implement *navigational features* that can describe more complex connectivities between nodes via longer paths instead of only single arcs, we need more complicated path specifications.

Navigational features are popular, since they allow to query the *topology* of the data and, if transitivity can be described, exceed the power of the basic relational query languages. Using regular expressions as path specifications is the most common way of implementing navigational features. The *regular path queries* (RPQs) given by *single-edge* graph patterns ($\{x, y\}, \{(x, s, y)\}$), where s is a regular expression, can be considered the simplest navigational graph patterns. General graph patterns labelled by regular expressions are called *conjunctive regular path queries* (CRPQs).

PODS'20, June 14–19, 2020, Portland, OR, USA

ACM ISBN 978-1-4503-7108-7/20/06…$15.00
https://doi.org/10.1145/3375395.3387663

For example, consider a graph database with nodes representing persons, arcs (u, p, v) meaning "u is a (biological) parent of v" and arcs (u, s, v) meaning "v is u's PhD-supervisor". We consider the graph patterns from Figure 1 (labelled nodes are considered as free variables of the query). Then G_1 describes pairs (v_1, v_2), where v_1's child has been supervised by v_2's parent; G_2 describes pairs (v_1, v_2), where v_1 is a biological ancestor or an academical descendant of v_2; G_3 describes v_1 that have a biological ancestor that is also their academical ancestor; G_4 describes pairs (v_1, v_2), where v_1 and v_2 are biologically related as well as academically. Note that G_1, G_2 represent RPQs, while G_3, G_4 represent CRPQs.

The classes of RPQs and CRPQs (and modifications of them) have been intensively studied in the literature (see, e. g., [1, 2, 10, 13, 14, 16, 17, 19, 24, 39]). The former can be evaluated efficiently (see, e. g., [16]), while evaluation for the latter it is NP-complete in combined complexity, but NL-complete in data complexity (see [8]).

Despite their long-standing investigation, these basic classes still pose several challenges that are currently studied. For example, [35–37] provide an in-depth analysis of the complexity of RPQs for different path semantics. So far in this introduction, we implicitly assumed *arbitrary path* semantics, but since there are potentially infinitely many such paths, query languages that also retrieve paths often restrict this by considering simple paths or trails. However, such semantics make the evaluation of RPQs much more difficult (see [35–37] for details). Much effort has also been spent on extending RPQs and CRPQs to the setting where the data-elements stored at nodes of the graph database can also be queried (see [33, 34]). In [9], the authors represent partially defined graph data by graph patterns and then query them with CRPQs (among others). In the very recent paper [7], the authors study the boundedness problem for unions of CRPQs (i. e., the problem of finding an equivalent union of (relational) conjunctive queries).

Also in the practical world, pattern-based query languages for graph databases play a central role. Most prominently, the *W3C Recommendation* for *SPARQL 1.1* "is based around graph pattern matching" (as stated in Section 5 of [30]), and *Neo4J Cypher* also uses graph patterns as a core functionality (see [43]). Moreover, the graph computing framework *Apache TinkerPop*^{TM} contains the graph database query language *Gremlin* [44], which is more based on the navigational graph traversal aspect, but nevertheless supports pattern-based query mechanisms. Note that [4] surveys the main features of these three languages.

1.1 Main Goal of this Work

CRPQs are not expressive enough for many natural querying tasks (see, e. g., the introduction of [8]). The most obvious shortcoming is that we cannot express any *inter-path* dependencies, i. e., relations between the paths of the database that are matched by the arcs of the graph pattern, except that they must start or end with the same node. The main goal of this work is to extend CRPQs in order to properly increase their expressive power. In particular, we want to meet the following objectives:

(1) The increased expressive power should be reasonable, i. e., it should cover natural and relevant querying tasks.
(2) The extensions should be user-friendly, i. e., the obtained query language should be intuitive.
(3) The evaluation complexity should still be acceptable.

The main idea is to allow *string variables* in the edge labels of the graph patterns. For example, in G_1 of Figure 2, the $\mathsf{x}\{\mathsf{a} \vee \mathsf{b}\}$ label sets variable x to some word matched by regular expression $\mathsf{a} \vee \mathsf{b}$ and the occurrence of x on the other edge label then refers to the value of x. Hence, G_1 describes all pairs (v_1, v_2) such that v_1 has a direct a-predecessor that has v_2 as a transitive successor with respect to a or c, or v_1 has a direct b-predecessor that has v_2 as a transitive successor with respect to b or c. Similarly, G_2 of Figure 2 describes triangles (v_1, v_2, v_3) with a complicated connectivity relation: v_1 reaches v_2 with aa or b, v_2 reaches v_3 with some path labelled only with symbols different than a and b (Σ is the set of edge labels), while v_3 reaches v_1 either in the same way as v_1 reaches v_2, or in the same way as v_2 reaches v_3.

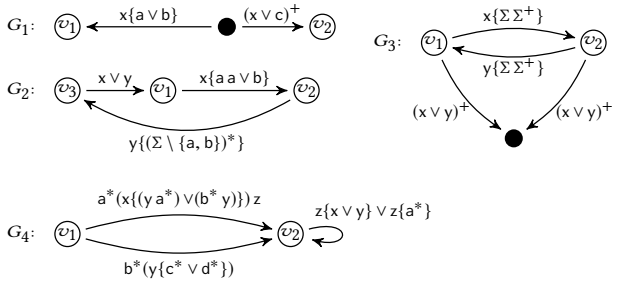

Figure 2: Examples of CRPQs with string variables.

The graph patterns G_1 and G_2 of Figure 2 could also be considered as CRPQs with some syntactic sugar. More precisely, both these graph patterns can be transformed into a union of CRPQs by simply "spelling out" all combinations that are possible for the variables x and y. However, we can easily build examples that would translate into an exponential number of CRPQs, and, moreover, it can be argued that the necessity for explicitly listing *all* possible combinations, even though they can conveniently be stated in a concise way, is exactly what a user should not be bothered with.

We move on to an example, where a simple application of string variables adds substantial expressive power to a CRPQ. Let us assume that the nodes in a graph database represent persons and arcs represent text messages sent by mobile phone (let Σ be the set of messages). The idea is that some individuals try to hide their direct communication by encoding their messages by sequences of simple text messages that are sent via intermediate senders and receivers. In particular, we want to discover individuals who are likely to be involved in such a hidden communication network. In this regard, G_3 of Figure 2 describes pairs (v_1, v_2) such that v_1 reaches v_2 (and v_2 reaches v_1) by a sequence x (y, respectively) of at least 2 messages, and there is some person that has been contacted by v_1 and v_2 by paths that are repetitions of these message-sequences. Note that, in this example, both the length of the message-sequences x and y, as well as the number of their repetitions in order to reach the mutual friend of v_1 and v_2, are unbounded.

Finally, G_4 of Figure 2 shows a feature that has not been used in the previous examples: references to variables could also occur in the definitions of other variables, e. g., y is defined on one edge,

and used in the definition of both x and z on the other two edges. Moreover, note that the same variable z has two definitions, which are mutually exclusive and therefore do not cause ambiguities.

This formalism obviously extends CRPQ; moreover, it is easy to see that it also covers wildcards and edge variables described above, as well as the fragment of *extended conjunctive regular path queries* (ECRPQs) [8] that only have equality-relations as non-unary relations (ECRPQs shall be discussed in more detail below).

1.2 Conjunctive Xregex Path Queries

Regular expressions of the kind used in the graph patterns of Figure 2 are actually a well-established concept, which, in the theoretical literature, is usually called *regex* or *xregex*, and string variables are often called *backreferences*. Xregex have been investigated in the formal language community [15, 25, 28, 40, 41], and, despite the fact that allowing them in regular expressions has many negative consequences (see [3, 21–23, 25]), regular expression libraries of almost all modern programming languages (like, e. g., Java, PERL, Python and .NET) support backreferences (although they syntactically and even semantically slightly differ from each other (see the discussion in [28])), and they are even part of the POSIX standard [32]. The syntax of xregex is quite intuitive: on top of normal regular expressions, we can *define* variables by the construct $\mathsf{x}\{\ldots\}$ and *reference* them by occurrences of x (see Figure 2). Formally defining their semantics is more tricky (mainly due to nested variable definitions and undefined variables), but for simple xregex their meaning is intuitively clear.

Obviously, we can define various query classes by replacing the regular expressions in CRPQ by some more powerful language descriptors (see, e. g., [31], where this has been done with respect to context-free grammars). However, this will not remedy the lack of a means to describe inter-path dependencies, and, furthermore, regular expressions seem powerful enough to describe the desired navigational features (in fact, the analyses carried out in [11, 12, 38] suggest that the regular expressions used in practical queries are even rather simple). What makes xregex interesting is that defining a variable on some edge and referencing it on another is a convenient way of describing inter-path dependencies, while, syntactically, staying in the realm of graph patterns with regular expressions.

To define such a query class, we first have to lift xregex to (multi-dimensional) *conjunctive xregex*, i. e., tuples $\bar{\alpha} = (\alpha_1, \alpha_2, \ldots, \alpha_m)$ of xregex that can generate tuples $\bar{w} = (w_1, w_2, \ldots, w_m)$ of words, but such that the w_i match the α_i in a *conjunctive* way with respect to the string variables (i. e., occurrences of the same variable x in different α_i and α_j must refer to the same string). Labeling graph patterns with xregex and interpreting the edge labels as conjunctive xregex yields our class of *conjunctive xregex path queries* (CXRPQs).

With respect to Objective 2 from above, we note that CXRPQs are purely pattern-based, i. e., the queries are just graph patterns with edge labels (as pointed out above, such queries are widely adapted in practice and we can therefore assume their user-friendliness), and the xregex used as path-specifications are a well-known practical tool (in fact, the use of string variables (backreferences) in regular expressions is a topic covered by standard textbooks on practical application of regular expressions (see, e. g., [29])). Regarding Objective 1, string variables add some mechanism to describe inter-path relationships and we already saw some illustrating examples. We will further argue in favour of their expressive power in the remainder of the introduction (see also Figure 5).

1.3 Barcelo et al.'s ECRPQ

An existing class of graph queries that is suitable for comparison with the class CXRPQ are the extended conjunctive regular path queries (ECRPQs) introduced in [8]. While CRPQs can be seen as graph patterns with unary regular relations on the edges (i. e., the regular expressions), the class ECRPQ permits regular relations of arbitrary arity, which allows to formulate a wide range of inter-path dependencies. ECRPQs have acceptable evaluation complexity: NL-complete in data complexity and PSpace-complete in combined complexity (see [8] for details).

In terms of expressive power, CXRPQs completely cover the fragment $\mathrm{ECRPQ}^{\mathsf{er}}$ of those ECRPQs that have only unary relations or equality relations (i. e., relations requiring certain paths to be equal). On the other hand, we can show that there are CXRPQ that are not expressible by $\mathrm{ECRPQ}^{\mathsf{er}}$. From an intuitive point of view CXRPQ and ECRPQ are incomparable in the sense that ECRPQ can describe inter-path dependencies beyond simple equality, while CXRPQ can describe equality of arbitrarily many paths as, e. g., by $(\mathsf{x} \vee \mathsf{y})^+$ in the query G_3 of Figure 2.

The class of ECRPQ is not purely pattern-based anymore, since also the relations must be given as regular expressions. In this regard, the authors of [8] mention that "[...] specifying regular relations with regular expressions is probably less natural than specifying regular languages (at least it is more cumbersome) [...]" and suggest that any practical standard would rather provide some reasonable regular relations as built-in predicates.

1.4 Technical Contributions

In terms of Objective 3, the best evaluation complexity that we can hope for is NL in data complexity and NP in combined complexity (since CRPQs have these lower bounds). Higher combined complexity (e. g., PSpace) is still acceptable, as long as the optimum of NL in data complexity is reached (this makes sense if we can assume our queries to be rather small in comparison to the data).

Unfortunately, CXRPQs have a surprisingly high evaluation complexity: even for the fixed xregex $\alpha_{\mathsf{ni}} = \#\mathsf{x}\{(\mathsf{a} \vee \mathsf{b})^*\}\,(\#\#\,\mathsf{x})^*\#\#\#$, it is PSpace-hard to decide whether a given graph database contains a path labelled by a word from $\mathcal{L}(\alpha_{\mathsf{ni}})$ (so Boolean evaluation is PSpace-hard in data complexity). This hardness result has nevertheless a silver lining: it directly points us to restrictions of CXRPQ that might lead to more tractable fragments. More precisely, for PSpace-hardness it seems vital that references for variable x are subject to the star-operator, and that the variable x can store words of unbounded length. Our main positive result will be that by restricting CXRPQs accordingly, we can tame their evaluation complexity and obtain more tractable fragments.

Let $\mathrm{CXRPQ}^{\mathsf{vsf}}$ be the class of *variable-star free* CXRPQs, i. e., no variable can be used under a star or plus (but normal symbols still can). For example, G_2, G_4 of Figure 2 are in $\mathrm{CXRPQ}^{\mathsf{vsf}}$. This

restriction is enough to make the data complexity drop from PSpace-hardness to the optimum of NL-completeness (although combined complexity is ExpSpace). The upper bound is obtained by showing that $q \in \text{CXRPQ}^{\text{vsf}}$ can be transformed into equivalent q' in a certain *normal form*, which can be evaluated in nondeterministic space $O(|q'| \log(|\mathcal{D}|))$. However, $|q'| = O(2^{2^{|q|}})$ and we only get the single exponential space upper bound for combined complexity by handling one exponential size blow-up with nondeterminism. A closer look at the normal form construction reveals that the (seemingly unavoidable) exponential size blow-up is caused by chains of the following form: a reference of x occurs in the definition of y, a reference of y occurs in the definition of z and so on (e. g., this happens with respect to x, y and z in G_4 of Figure 2). If we require that every variable definition only contains references of variables that themselves have definitions without variables, then we obtain the fragment $\text{CXRPQ}^{\text{vsf,fl}}$ which have normal forms of polynomial size and therefore the combined complexity drops to PSpace (i. e., the same complexity as ECRPQ).

The second successful approach is to add a constant upper-bound k on the *image size* of CXRPQs, i. e., the length of the words stored in variables. Let $\text{CXRPQ}^{\leq k}$ be the corresponding fragments. For every $\text{CXRPQ}^{\leq k}$, the evaluation complexity drops to the optimum of NL-completeness in data complexity and NP-completeness in combined complexity (i. e., the same as for CRPQs). Unlike for $\text{CXRPQ}^{\text{vsf}}$, bounding the image size does not impose any syntactical restrictions; it is rather a restriction of how a CXRPQ can match a graph database. For example, we can treat G_3 of Figure 2 as a $\text{CXRPQ}^{\leq 10}$, i. e., we only have a successful match if the paths between v_1 and v_2 are of length at most 10 (note that the paths from v_1 and v_2 to their mutual friend can have unbounded size). For G_1 of Figure 2, on the other hand, the image size of variables is necessarily bounded by 1 and therefore it does not matter whether we interpret it as CXRPQ or $\text{CXRPQ}^{\leq k}$ with $k \geq 1$. We stress the fact that this bound only applies to strings stored in variables; we can still specify paths of arbitrary length with regular expressions, i. e., $\text{CRPQ} \subseteq \text{CXRPQ}^{\leq k}$. Moreover, evaluating $\text{CXRPQ}^{\leq k}$ is not as simple as just replacing all variables by fixed words of length at most k and then evaluating a CRPQ, since we also have to take care of dependencies between variable definitions.

Finally, there are two more noteworthy results about $\text{CXRPQ}^{\leq k}$ (a negative and a positive one). While in combined complexity CRPQ can be evaluated in polynomial-time if the underlying graph pattern is acyclic (see [6, 8, 10]), $\text{CXRPQ}^{\leq k}$ remain NP-hard in combined complexity even for single-edge graph patterns (and $k = 1$). This also demonstrates the general difference of $\text{CXRPQ}^{\leq k}$ and CRPQ. On the positive side, if instead of a constant upper bound, we allow images bounded logarithmically in the size of the database, then the NP upper bound in combined complexity remains, while the data complexity slightly increases to $O(\log^2(|\mathcal{D}|))$.

The question is whether these fragments are still interesting with respect to Objectives 1 and 2. We believe the answer is yes. First observe that the restrictions are quite natural: Not using the star-operator over variables is a rule not difficult for users to comply with, if they are familiar with regular expressions; it is also easily to be checked algorithmically, and the same holds for the additional restriction required by $\text{CXRPQ}^{\text{vsf,fl}}$. The class $\text{CXRPQ}^{\leq k}$ does not require any syntactical restriction; when interpreting the query result, the user only has to keep in mind that the paths corresponding to images of variables are bounded in length.

Regarding expressive power, all these fragments contain non-trivial examples of CXRPQs. With respect to the examples from Figure 2, $G_4 \in \text{CXRPQ}^{\text{vsf}}$ and $G_2 \in \text{CXRPQ}^{\text{vsf,fl}}$; any CXRPQ can be interpreted as $\text{CXRPQ}^{\leq k}$ for any k. Both $\text{CXRPQ}^{\text{vsf}}$ and $\text{CXRPQ}^{\text{vsf,fl}}$ still cover the fragment ECRPQ^{er}. It is tempting to misinterpret queries from $\text{CXRPQ}^{\leq k}$ as CRPQ with mere syntactic sugar (since string variables range over finite sets of words). However, it can be proven that even $\text{CXRPQ}^{\leq 1}$ contains queries that are not expressible by CRPQs.

We either omit formal proofs (results marked with ⋆) or only provide proof sketches. A full version with all technical details is in preparation (see [42] for a preliminary full version of this paper).

2 PRELIMINARIES

Let $\mathbb{N} = \{1, 2, 3, \ldots\}$ and $[n] = \{1, 2, \ldots, n\}$ for $n \in \mathbb{N}$. A^+ denotes the set of non-empty words over an alphabet A and $A^* = A^+ \cup \{\varepsilon\}$ (where ε is the empty word). For a word $w \in A^*$, $|w|$ denotes its length, and for $k \in \mathbb{N}$, $A^{\leq k} = \{w \in A^* \mid |w| \leq k\}$. For $w_1, w_2, \ldots, w_n \in A^*$, we set $\prod_{i=1}^{n} w_i = w_1 w_2 \ldots w_n$, and if $w = w_i$, for every $i \in [n]$, then we also write w^n instead of $\prod_{i=1}^{n} w_i$.

We fix a finite *terminal* alphabet Σ and an enumerable set $\mathcal{X}_s$ of *string variables*, where $\mathcal{X}_s \cap \Sigma = \emptyset$. As a convention, we use symbols $\mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{d}, \ldots$ for elements from Σ, and $\mathsf{x}, \mathsf{y}, \mathsf{z}, \mathsf{x}_1, \mathsf{x}_2, \ldots, \mathsf{y}_1, \mathsf{y}_2, \ldots$ for variables from $\mathcal{X}_s$. We consistently use sans-serif font for string variables to distinguish them from *node variables* to be introduced later. We use *regular expressions* and (*nondeterministic*) *finite automata* (NFA for short) as commonly defined in the literature (see Section 3.1 and the remainder of this section for more details).

2.1 Ref-Words

The following *ref-words* (first introduced in [41]) are convenient for defining the semantics of xregex (Section 3). They have also been used in [28] and for so-called document spanners in [20, 26, 27]. Ref-words will be vital in our definition of conjunctive xregex (Section 3.1), which are the basis of the class CXRPQ.

For every $\mathsf{x} \in \mathcal{X}_s$, we use the pair of symbols $\vdash^{\mathsf{x}}$ and $\dashv^{\mathsf{x}}$, which are interpreted as opening and closing parentheses.

Definition 2.1 (Ref-Words). A *subword-marked* word (*over terminal alphabet Σ and variables $\mathcal{X}_s$*) is a word $w \in (\Sigma \cup \{\vdash^{\mathsf{x}}, \dashv^{\mathsf{x}} \mid \mathsf{x} \in \mathcal{X}_s\} \cup \mathcal{X}_s)^*$ that, for every $\mathsf{x} \in \mathcal{X}_s$, contains the parentheses $\vdash^{\mathsf{x}}$ and $\dashv^{\mathsf{x}}$ at most once and all these parentheses form a well-formed parenthesised expression. For every $\mathsf{x} \in \mathcal{X}_s$, a subword $\vdash^{\mathsf{x}} v \dashv^{\mathsf{x}}$ in w is called a *definition* (of variable x), and an occurrence of symbol x is called a *reference* (of variable x). For a subword-marked word w over Σ and $\mathcal{X}_s$, the binary relation $\preceq_w$ over $\mathcal{X}_s$ is defined by setting $\mathsf{x} \preceq_w \mathsf{y}$ if in w there is a definition of y that contains a reference or a definition of x. A *ref-word* (*over terminal alphabet Σ and variables $\mathcal{X}_s$*) is a subword-marked word over Σ and $\mathcal{X}_s$, such that the transitive closure of $\preceq_w$ is acyclic.

Ref-words are just words over alphabet $\Sigma \cup \mathcal{X}_s$, in which some subwords are uniquely marked by means of the parentheses $\vdash^{\mathsf{x}}$ and $\dashv^{\mathsf{x}}$. Moreover, the marked subwords are not allowed to overlap, i. e.,

$\mathsf{x}\triangleright\ \mathsf{y}\triangleright\ \triangleleft^{\mathsf{x}}\ \triangleleft^{\mathsf{y}}$ must not occur as subsequence. For every variable $\mathsf{x} \in \mathcal{X}_s$, all occurrences of x in a ref-word are interpreted as references to the definition of x. Since definitions may contain itself references or definitions of other variables, there are chains of references, e. g., the definition of x contains references of y, but the definition of y contains references of z and so on. Therefore, in order to make this implicit referencing process terminate, we have to require that it is acyclic, which is done by requiring the transitive closure of $\preceq_w$ to be acyclic. For example, $\mathsf{axb}\ \mathsf{x}\triangleright\ \mathsf{ab}\ \triangleleft^{\mathsf{x}}\ \mathsf{c}\ \mathsf{y}\triangleright\ \mathsf{xaa}\ \triangleleft^{\mathsf{y}}\ \mathsf{y}$ is a valid ref-word, while $\mathsf{axb}\ \mathsf{x}\triangleright\ \mathsf{ab}\ \triangleleft^{\mathsf{x}}\ \mathsf{c}\ \mathsf{y}\triangleright\ \mathsf{xaay}\ \triangleleft^{\mathsf{y}}\ \mathsf{y}$, or $\mathsf{axa}\ \mathsf{x}\triangleright\ \mathsf{ayb}\ \triangleleft^{\mathsf{x}}\ \mathsf{c}\ \mathsf{y}\triangleright\ \mathsf{xa}\triangleleft^{\mathsf{y}}$ are not.

For ref-words over Σ and $\mathcal{X}_s$, it is therefore possible to successively substitute references by their definitions until we obtain a word over Σ. This can be formalised as follows.

Definition 2.2 (Deref-Function). For a ref-word w over Σ and $\mathcal{X}_s$, $\mathsf{deref}(w) \in \Sigma^*$ is obtained from w by the following procedure:

(1) Delete all occurrences of $\mathsf{x} \in \mathcal{X}_s$ without definition in w.
(2) Repeat until we have obtained a word over Σ:
   (a) Let $\mathsf{x}\triangleright\ v_{\mathsf{x}}\triangleleft^{\mathsf{x}}$ be a definition such that $v_{\mathsf{x}} \in \Sigma^*$.
   (b) Replace $\mathsf{x}\triangleright\ v_{\mathsf{x}}\triangleleft^{\mathsf{x}}$ and all occurrences of x in w by v_{x}.

Proposition 2.3 (★). *The function* $\mathsf{deref}(w)$ *is well-defined.*

For a ref-word w, the procedure of Definition 2.2 that computes $\mathsf{deref}(w)$ also uniquely allocates a subword v_{x} of $\mathsf{deref}(w)$ to each variable x that has a definition in w (i. e., the subwords v_{x} defined in the iterations of Step 2a). In this way, a ref-word w over terminal alphabet Σ and variables $\mathcal{X}_s$ describes a *variable mapping* $\mathsf{vmap}_{w,\mathcal{X}_s} : \mathcal{X}_s \to \Sigma^*$, i. e., we set $\mathsf{vmap}_{w,\mathcal{X}_s}(x) = v_{\mathsf{x}}$ if x has a definition in w and we set $\mathsf{vmap}_{w,\mathcal{X}_s}(x) = \varepsilon$ otherwise. The elements $\mathsf{vmap}_{w,\mathcal{X}_s}(x)$ for $x \in \mathcal{X}_s$ are called *variable images*.

Note that even if x has a definition in w, $\mathsf{vmap}_{w,\mathcal{X}_s}(x) = \varepsilon$ is possible due to a definition $\mathsf{x}\triangleright\ \triangleleft^{\mathsf{x}}$ or to a definition $\mathsf{x}\triangleright\ v_{\mathsf{x}}\triangleleft^{\mathsf{x}}$, where v_{x} is a non empty word with only references of variables y with $\mathsf{vmap}_{w,\mathcal{X}_s}(y) = \varepsilon$. Also note that any ref-word w over terminal alphabet Σ and variables $\mathcal{X}_s$ is also a ref-word over any terminal alphabet $\Sigma' \supset \Sigma$ and variables $\mathcal{X}'_s \supset \mathcal{X}_s$ ($\mathsf{vmap}_{w,\mathcal{X}'_s}$ then equals the extension of $\mathsf{vmap}_{w,\mathcal{X}_s}$ by $\mathsf{vmap}_{w,\mathcal{X}'_s}(x') = \varepsilon$ for every $\mathsf{x}' \in \mathcal{X}'_s \setminus \mathcal{X}_s$). If the set of variables $\mathcal{X}_s$ is clear from the context or negligible, we shall also denote the variable mapping by vmap_w and if there is some obvious implicit order on $\mathcal{X}_s$, e. g., given by indices as in the case $\mathcal{X}_s = \{\mathsf{x}_1, \mathsf{x}_2, \ldots, \mathsf{x}_m\}$, then we also write vmap_w as a tuple $(\mathsf{vmap}_w(\mathsf{x}_1), \mathsf{vmap}_w(\mathsf{x}_2), \ldots, \mathsf{vmap}_w(\mathsf{x}_m))$.

A set L of ref-words is called *ref-language* and we extend the deref-function to ref-languages in the obvious way, i. e., $\mathsf{deref}(L) = \{\mathsf{deref}(w) \mid w \in L\}$. Let us illustrate ref-words with an example.

Example 2.4. Let $\Sigma = \{\mathsf{a}, \mathsf{b}, \mathsf{c}\}$ and let $\mathsf{x}_1, \mathsf{x}_2, \mathsf{x}_3, \mathsf{x}_4 \in \mathcal{X}_s$.

$$w = \mathsf{ax}_4\mathsf{a}\ \mathsf{x}_1\triangleright\ \mathsf{ab}\ \mathsf{x}_2\triangleright\ \mathsf{acc}\ \triangleleft^{\mathsf{x}_2}\ \mathsf{ax}_2\mathsf{x}_4\ \triangleleft^{\mathsf{x}_1}\ \ \mathsf{x}_3\triangleright\ \mathsf{x}_1\mathsf{ax}_2\ \triangleleft^{\mathsf{x}_3}\ \mathsf{x}_3\mathsf{bx}_1\,.$$

The procedure of Definition 2.2 will first delete all occurrences of x_4. Then $\mathsf{x}_2\triangleright\ \mathsf{acc}\triangleleft^{\mathsf{x}_2}$ and all references of x_2 are replaced by acc. After this step, the definition for variable x_1 is $\mathsf{x}_1\triangleright\ \mathsf{abaccaacc}\triangleleft^{\mathsf{x}_1}$, so abaccaacc can be substituted for the definitions and references of x_1. After replacing the last variable x_3, we obtain $\mathsf{deref}(w)$. Note that $\mathsf{vmap}_w = (\mathsf{abaccaacc}, \mathsf{acc}, \mathsf{abaccaaccaacc}, \varepsilon)$.

2.2 Graph-Databases

A *graph-database* (*over* Σ) is a directed, edge labelled multigraph $\mathcal{D} = (V_\mathcal{D}, E_\mathcal{D})$, where $V_\mathcal{D}$ is the set of *vertices* (or *nodes*) and $E_\mathcal{D} \subseteq V_\mathcal{D} \times \Sigma \times V_\mathcal{D}$ is the set of *edges* (or *arcs*). A path from $u \in V_\mathcal{D}$ to $v \in V_\mathcal{D}$ of length $k \geq 0$ is a sequence

$$p = (w_0, a_1, w_1, a_2, w_2 \ldots, w_{k-1}, a_k, w_k)$$

with $(w_{i-1}, a_i, w_i) \in E_\mathcal{D}$ for every $i \in [k]$. We say that p is *labelled* with the *word* $a_1 a_2 \ldots a_k \in \Sigma^*$. According to this definition, for every $v \in V_\mathcal{D}$, (v) is a path from v to v of length 0 that is labelled by ε. Hence, every node of every graph-database has an ε-labelled path to itself (and these are the only ε-labelled paths in $\mathcal{D}$).

Nondeterministic finite automata (NFAs) are just graph databases, the nodes of which are called *states*, and that have a specified *start state* and a set of specified *final states*. Moreover, we allow the empty word as edge label as well (which is not the case for graph databases). The language $\mathcal{L}(M)$ of an NFA M is the set of all labels from paths from the start state to some final state.

In the following, $\mathcal{X}_n$ is an enumerable set of *node-variables*; we shall use symbols $x, y, z, x_1, x_2, \ldots, y_1, y_2, \ldots$ for node variables (in contrast to the string variables $\mathcal{X}_s$ in sans-serif font).

2.3 Conjunctive Path Queries

Let $\mathfrak{R}$ be a class of language descriptors, and, for every $r \in \mathfrak{R}$, let $\mathcal{L}(r)$ denote the language represented by r. An $\mathfrak{R}$*-graph pattern* is a directed, edge-labelled graph $G = (V, E)$ with $V \subseteq \mathcal{X}_n$ and $E \subseteq V \times \mathfrak{R} \times V$; it is an $\mathfrak{R}$-graph pattern *over alphabet* Σ, if $\mathcal{L}(\alpha) \subseteq \Sigma^*$ for every $(x, \alpha, y) \in E$. For an $\mathfrak{R}$-graph pattern $G = (V, E)$ over Σ and a graph-database $\mathcal{D} = (V_\mathcal{D}, E_\mathcal{D})$ over Σ, a mapping $h : V \to V_\mathcal{D}$ is a *matching morphism* for G and $\mathcal{D}$ if, for every $e = (x, \alpha, y) \in E$, $\mathcal{D}$ contains a path from $h(x)$ to $h(y)$ that is labelled with a word $w_e \in \mathcal{L}(\alpha)$. The tuple $(w_e)_{e \in E}$ is a tuple of *matching words* (*with respect to* h). In particular, a matching morphism can have several different tuples of matching words.

A *conjunctive* $\mathfrak{R}$*-path query* ($\mathfrak{R}$-CPQ for short) is a query $q = \bar{z} \leftarrow G_q$, where $G_q = (V_q, E_q)$ is an $\mathfrak{R}$-graph pattern and $\bar{z} = (z_1, z_2, \ldots, z_\ell)$ with $\{z_1, z_2, \ldots, z_\ell\} \subseteq V_q$. We say that q is an $\mathfrak{R}$-CPQ *over alphabet* Σ if G_q is an $\mathfrak{R}$-graph pattern over Σ. The query q is a *single-edge* query, if $|E_q| = 1$.

For an $\mathfrak{R}$-CPQ $q = \bar{z} \leftarrow G_q$ over Σ with $\bar{z} = (z_1, z_2, \ldots, z_\ell)$, a graph-database $\mathcal{D} = (V_\mathcal{D}, E_\mathcal{D})$ over Σ and a matching morphism h for G_q and $\mathcal{D}$ (we also call h a *matching morphism for* q *and* $\mathcal{D}$), we define $q_h(\mathcal{D}) = (h(z_1), h(z_2), \ldots, h(z_\ell))$ and we set $q(\mathcal{D}) = \{q_h(\mathcal{D}) \mid h \text{ is a matching morphism for } q \text{ and } \mathcal{D}\}$. The mapping $\mathcal{D} \mapsto q(\mathcal{D})$ from the set of graph-databases to the set of relations over Σ of arity ℓ that is defined by q shall be denoted by $[\![q]\!]$, and for any class A of conjunctive path queries, we set $[\![A]\!] = \{[\![q]\!] \mid q \in A\}$.

A *Boolean* $\mathfrak{R}$-CPQ has the form $\bar{z} \leftarrow G_q$ where $\bar{z}$ is the empty tuple. In this case, we also denote q just by G_q instead of $() \leftarrow G_q$. For a Boolean $\mathfrak{R}$-CPQ q and a graph-database $\mathcal{D}$, we either have $q(\mathcal{D}) = \{()\}$ or $q(\mathcal{D}) = \emptyset$, which we shall also denote by $\mathcal{D} \models q$ and $\mathcal{D} \not\models q$, respectively. For Boolean queries q, the set $[\![q]\!]$ can also be interpreted as $\{\mathcal{D} \mid \mathcal{D} \models q\}$. Two $\mathfrak{R}$-CPQs q and q' are *equivalent*, denoted by $q \equiv q'$, if $[\![q]\!] = [\![q']\!]$, i. e., $q(\mathcal{D}) = q'(\mathcal{D})$ for every graph-database $\mathcal{D}$ (or, in the Boolean case, $\mathcal{D} \models q \Leftrightarrow \mathcal{D} \models q'$).

For a class Q of conjunctive path queries, Q-Bool-Eval is the problem to decide, for a given Boolean $q \in Q$ and a graph database $\mathcal{D}$, whether $\mathcal{D} \models q$. By Q-Check, we denote the problem to check $\bar{t} \in q(\mathcal{D})$ for a given $q \in Q$, a graph database $\mathcal{D}$ and a tuple $\bar{t}$.

As common in database theory, the *combined complexity* for an algorithm solving Q-Bool-Eval or Q-Check is the time or space needed by the algorithm measured in both $|q|$ and $|\mathcal{D}|$, while for the *data complexity* the query q is considered constant. For simplicity, we assume $|q| = O(|\mathcal{D}|)$ throughout the paper.

Conjunctive regular path queries (CRPQ) are $\mathfrak{R}$-CPQ where $\mathfrak{R}$ is the class of regular expressions (which are defined in Section 3). For some of our results, we need the following result about CRPQs.

Lemma 2.5 ([8]). CRPQ-*Bool-Eval is* NP*-complete in combined complexity and* NL*-complete in data complexity.*

3 XREGEX

We next define the underlying language descriptors for our class of conjunctive path queries. Complete formalisations of the class of xregex can also be found elsewhere in the literature (e. g., [25, 28, 41]). However, since we will extend xregex to conjunctive xregex (the main building block for CXRPQ), we give a full definition.

Definition 3.1 (Xregex). The set $\mathsf{XRE}_{\Sigma,\mathcal{X}_s}$ of *regular expressions with backreferences* (*over* Σ *and* $\mathcal{X}_s$), also denoted by *xregex*, for short, is recursively defined as follows:

(1) $a \in \mathsf{XRE}_{\Sigma,\mathcal{X}_s}$ and $\mathrm{var}(a) = \emptyset$, for every $a \in \Sigma \cup \{\varepsilon\}$,
(2) $\mathsf{x} \in \mathsf{XRE}_{\Sigma,\mathcal{X}_s}$ and $\mathrm{var}(\mathsf{x}) = \{\mathsf{x}\}$, for every $\mathsf{x} \in \mathcal{X}_s$,
(3) $(\alpha \cdot \beta) \in \mathsf{XRE}_{\Sigma,\mathcal{X}_s}$, $(\alpha \vee \beta) \in \mathsf{XRE}_{\Sigma,\mathcal{X}_s}$, and $(\alpha)^+ \in \mathsf{XRE}_{\Sigma,\mathcal{X}_s}$, for every $\alpha, \beta \in \mathsf{XRE}_{\Sigma,\mathcal{X}_s}$;
furthermore, $\mathrm{var}((\alpha \cdot \beta)) = \mathrm{var}((\alpha \vee \beta)) = \mathrm{var}(\alpha) \cup \mathrm{var}(\beta)$ and $\mathrm{var}((\alpha)^+) = \mathrm{var}(\alpha)$,
(4) $\mathsf{x}\{\alpha\} \in \mathsf{XRE}_{\Sigma,\mathcal{X}_s}$ and $\mathrm{var}(\mathsf{x}\{\alpha\}) = \mathrm{var}(\alpha) \cup \{\mathsf{x}\}$, for every $\alpha \in \mathsf{XRE}_{\Sigma,\mathcal{X}_s}$ and $\mathsf{x} \in \mathcal{X}_s \setminus \mathrm{var}(\alpha)$.

For technical reasons, we also add $\varnothing$ to $\mathsf{XRE}_{\Sigma,\mathcal{X}_s}$. For $\alpha \in \mathsf{XRE}_{\Sigma,\mathcal{X}_s}$, we use r^* as a shorthand form for $r^+ \vee \varepsilon$, and we usually omit the operator '$\cdot$', i. e., we use juxtaposition. If this does not cause ambiguities, we often omit parenthesis. For example, x, $\mathsf{x}\{\mathsf{ya}\}$ and $\mathsf{x}\{(\mathsf{y}\{\mathsf{z}\{\mathsf{a}^* \vee \mathsf{bc}\}\mathsf{a}\}\mathsf{y})^+\mathsf{b}\}\mathsf{x}$ are all valid xregex, while neither $\mathsf{x}\{\mathsf{ax}\}\mathsf{b}$ nor $\mathsf{x}\{\mathsf{ax}\{\mathsf{b}^*\}\mathsf{a}\}\mathsf{b}$ is a valid xregex. The operations $\cdot$, $\vee$, $+$ and $*$ are also called *concatenation*, *alternation*, *plus* and *star*, respectively.

We call an occurrence of $\mathsf{x} \in \mathcal{X}_s$ a *reference of variable* x and a subexpression $\mathsf{x}\{\alpha\}$ a *definition of variable* x. The set $\mathsf{XRE}_{\Sigma,\emptyset}$ is exactly the set of regular expressions over Σ, which shall be denoted by RE_Σ in the following. We also use the term *classical* regular expressions for a clearer distinction from xregex. If the underlying alphabet Σ or set $\mathcal{X}_s$ of variables is negligible or clear form the context, we also drop these and simply write XRE and RE.

We next define the semantics of xregex, for which we heavily rely on the concept of ref-words (see Section 2). First, for any $\alpha \in \mathsf{RE}$, let $\mathcal{L}(\alpha)$ be the regular language described by the (classical) regular expression α defined as usual: $\mathcal{L}(a) = \{a\}$, $\mathcal{L}(\alpha \cdot \beta) = \mathcal{L}(\alpha) \cdot \mathcal{L}(\beta)$, $\mathcal{L}(\alpha \vee \beta) = \mathcal{L}(\alpha) \cup \mathcal{L}(\beta)$, $\mathcal{L}(\alpha^+) = \mathcal{L}(\alpha)^+$. Now in order to define the language described by an xregex $\alpha \in \mathsf{XRE}_{\Sigma,\mathcal{X}_s}$, we first define a classical regular expression α_{ref} over the alphabet $\Sigma \cup \mathcal{X}_s \cup \{{}^{\mathsf{x}}\triangleright, \triangleleft^{\mathsf{x}} \mid \mathsf{x} \in \mathcal{X}_s\}$ that is obtained from α by iteratively replacing all variable definitions $\mathsf{x}\{\beta\}$ by ${}^{\mathsf{x}}\triangleright\, \beta \triangleleft^{\mathsf{x}}$. For example,

$$\alpha = \mathsf{x}\{(\mathsf{y}\{\mathsf{z}\{\mathsf{a}^* \vee \mathsf{bc}\}\mathsf{a}\}\mathsf{y})^+\mathsf{b}\}\mathsf{x},$$
$$\alpha_{\mathrm{ref}} = {}^{\mathsf{x}}\triangleright\, (\, {}^{\mathsf{y}}\triangleright\, {}^{\mathsf{z}}\triangleright\, (\mathsf{a}^* \vee \mathsf{bc}) \triangleleft^{\mathsf{z}} \mathsf{a} \triangleleft^{\mathsf{y}} \mathsf{y})^+ \mathsf{b} \triangleleft^{\mathsf{x}} \mathsf{x}.$$

We say that α is *sequential* if every $w \in \mathcal{L}(\alpha_{\mathrm{ref}})$ contains for every $\mathsf{x} \in \mathcal{X}_s$ at most one occurrence of ${}^{\mathsf{x}}\triangleright$ (if not explicitly stated otherwise, all our xregex are sequential). If an xregex α is sequential, then every $w \in \mathcal{L}(\alpha_{\mathrm{ref}})$ must be a ref-word. Indeed, this directly follows from the fact that the definitions $\mathsf{x}\{\ldots\}$ are always subexpressions. Hence, for sequential xregex, $\mathcal{L}(\alpha_{\mathrm{ref}})$ is a ref-language, which we shall denote by $\mathcal{L}_{\mathrm{ref}}(\alpha)$. If a ref-word $v \in \mathcal{L}_{\mathrm{ref}}(\alpha)$ contains a definition ${}^{\mathsf{x}}\triangleright\, v_{\mathsf{x}} \triangleleft^{\mathsf{x}}$, then we say that the corresponding definition $\mathsf{x}\{\gamma_x\}$ in α is *instantiated* (by v). In particular, we observe that sequential xregex can nevertheless have several definitions for the same variable x, but at most one of them is instantiated by any ref-word. Finally, the language described by α is defined as $\mathcal{L}(\alpha) = \mathrm{deref}(\mathcal{L}_{\mathrm{ref}}(\alpha))$. As a special case, we also define $\mathcal{L}(\varnothing) = \emptyset$. For $\alpha \in \mathsf{XRE}_{\Sigma,\mathcal{X}_s}$ and $w \in \Sigma^*$, we say that w *matches* α *with witness* $u \in \mathcal{L}_{ref}(\alpha)$ *and variable mapping* vmap_u, if $\mathrm{deref}(u) = w$.

Example 3.2. Let $\alpha \in \mathsf{XRE}_{\{\mathsf{a},\mathsf{b}\},\mathcal{X}_s}$ with $\mathsf{x}_1, \mathsf{x}_2 \in \mathcal{X}_s$:

$$\alpha = \mathsf{a}^*\mathsf{x}_1\{\mathsf{a}^*\mathsf{x}_2\{(\mathsf{a} \vee \mathsf{b})^*\}\mathsf{b}^*\mathsf{a}^*\}\mathsf{x}_2^*(\mathsf{a} \vee \mathsf{b})^*\mathsf{x}_1,$$
$$\alpha_{\mathrm{ref}} = \mathsf{a}^*\, {}^{\mathsf{x}_1}\triangleright\, \mathsf{a}^*\, {}^{\mathsf{x}_2}\triangleright\, (\mathsf{a} \vee \mathsf{b})^*\, \triangleleft^{\mathsf{x}_2}\, \mathsf{b}^*\mathsf{a}^*\, \triangleleft^{\mathsf{x}_1}\, \mathsf{x}_2^*(\mathsf{a} \vee \mathsf{b})^*\mathsf{x}_1,$$
$$u_1 = \mathsf{aaaa}\, {}^{\mathsf{x}_1}\triangleright\, {}^{\mathsf{x}_2}\triangleright\, \mathsf{ba} \triangleleft^{\mathsf{x}_2} \mathsf{baa} \triangleleft^{\mathsf{x}_1} \mathsf{x}_2\mathsf{x}_2\mathsf{bbax}_1 \in \mathcal{L}_{\mathrm{ref}}(\alpha),$$
$$u_2 = \mathsf{aaa}\, {}^{\mathsf{x}_1}\triangleright\, \mathsf{a}\, {}^{\mathsf{x}_2}\triangleright\, \mathsf{bab} \triangleleft^{\mathsf{x}_2} \mathsf{aa} \triangleleft^{\mathsf{x}_1} \mathsf{x}_2\mathsf{abbx}_1 \in \mathcal{L}_{\mathrm{ref}}(\alpha),$$
$$w_\alpha = \mathrm{deref}(u_1) = \mathrm{deref}(u_2) = \mathsf{a}^4(\mathsf{ba})^2(\mathsf{ab})^3(\mathsf{ba})^3\mathsf{a} \in \mathcal{L}(\alpha),$$
$$\mathsf{vmap}_{u_1} = (\mathsf{babaa}, \mathsf{ba}), \mathsf{vmap}_{u_2} = (\mathsf{ababaa}, \mathsf{bab}).$$

For $\gamma = \mathsf{x}_1\{\mathsf{c}^*(\mathsf{x}_2\{\mathsf{a}^*\} \vee \mathsf{x}_3\{\mathsf{b}^*\})\}\mathsf{cx}_2\mathsf{cx}_3\mathsf{bx}_1, \mathsf{c}^2\mathsf{a}^2\mathsf{ca}^2\mathsf{cbc}^2\mathsf{a}^2 \in \mathcal{L}(\gamma)$ is witnessed by ref-word $u_5 = {}^{\mathsf{x}_1}\triangleright\, \mathsf{c}^2\, {}^{\mathsf{x}_2}\triangleright\, \mathsf{a}^2 \triangleleft^{\mathsf{x}_2} \triangleleft^{\mathsf{x}_1} \mathsf{cx}_2\mathsf{cx}_3\mathsf{bx}_1$, which has the variable mapping $\mathsf{vmap}_{u_5} = (\mathsf{c}^2\mathsf{a}^2, \mathsf{a}^2, \varepsilon)$. Note that empty variable images can result from variables without definitions in the ref-word (as in this example), but also from definitions of variables that define the empty word, as, e. g., in the ref-word ${}^{\mathsf{x}_1}\triangleright\, \triangleleft^{\mathsf{x}_1} \mathsf{cx}_1 \in \mathcal{L}_{\mathrm{ref}}(\mathsf{x}_1\{(\mathsf{a} \vee \mathsf{b})^*\}\mathsf{cx}_1)$.

For $\alpha \in \mathsf{XRE}_{\Sigma,\mathcal{X}_s}$, we define the relation $\preceq_\alpha$ analogously how it is done for ref-words, i. e., $\mathsf{x} \preceq_\alpha \mathsf{y}$ if in α there is a definition of y that contains a reference or a definition of x. Even though the transitive closure of $\preceq_v$ is acyclic for every $v \in \mathcal{L}_{\mathrm{ref}}(\alpha)$, the transitive closure of $\preceq_\alpha$ is not necessarily acyclic. For example, $\alpha = \mathsf{x}\{\mathsf{a}^*\}\mathsf{y}\{\mathsf{x}\} \vee \mathsf{y}\{\mathsf{a}^*\}\mathsf{x}\{\mathsf{y}\}$ is an xregex, but the transitive closure of $\preceq_\alpha$ is not acyclic. We call xregex *acyclic* if $\preceq_\alpha$ is acyclic.

3.1 Conjunctive Xregex

Syntactically, conjunctive xregex are tuples of xregex. Their semantics, however, is more difficult and we spend some more time with intuitive explanations before giving a formal definition.

Definition 3.3 (Conjunctive Xregex). A tuple $\bar{\alpha} = (\alpha_1, \ldots, \alpha_m) \in (\mathsf{XRE}_{\Sigma,\mathcal{X}_s})^m$ is a *conjunctive xregex of dimension* m, if $\alpha_1\alpha_2\ldots\alpha_m$ is an acyclic xregex.

By m-$\mathsf{CXRE}_{\Sigma,\mathcal{X}_s}$, we denote the set of conjunctive xregex of dimension m (over Σ and $\mathcal{X}_s$) and we set $\mathsf{CXRE}_{\Sigma,\mathcal{X}_s} = \bigcup_{m\geq 1} m\text{-}\mathsf{CXRE}_{\Sigma,\mathcal{X}_s}$. Note that $1\text{-}\mathsf{CXRE}_{\Sigma,\mathcal{X}_s} = \mathsf{XRE}_{\Sigma,\mathcal{X}_s}$. If the terminal alphabet Σ or set $\mathcal{X}_s$ of variables are negligible or clear from the

context, we also drop the corresponding subscripts. We also write $\bar{\alpha}[i]$ to denote the i^{th} element of some $\bar{\alpha} \in m\text{-CXRE}$.

Before defining next the semantics of conjunctive xregex, we first give some intuitive explanations. The central idea of a conjunctive xregex $\bar{\alpha} = (\alpha_1, \alpha_2, \ldots, \alpha_m)$ is that the definition of some x in some α_i also serves as definition for possible references of x in some α_j with $i \neq j$ (which, due to the requirement that $\alpha_1\alpha_2 \ldots \alpha_m$ is an xregex, cannot contain a definition of x). Let us illustrate the situation with the concrete example $\bar{\gamma} = (\gamma_1, \gamma_2)$, with $\gamma_1 = (\mathsf{x}\{\mathsf{a}^*\} \vee \mathsf{b}^*)\mathsf{y}$ and $\gamma_2 = \mathsf{y}\{\mathsf{xaxb}\}\mathsf{by}^*$.

As an intermediate step, we first add dummy definitions for the undefined variables, i. e., we define $\langle\gamma_1\rangle_{\text{int}} = \mathsf{y}\{\Sigma^*\}\#\gamma_1$ and $\langle\gamma_2\rangle_{\text{int}} = \mathsf{x}\{\Sigma^*\}\#\gamma_2$, which are both xregex in which all variables have a definition. Now, we can treat $(\langle\gamma_1\rangle_{\text{int}}, \langle\gamma_2\rangle_{\text{int}})$ as a generator for pairs of ref-words (and therefore as pairs of words over Σ), but we only consider those pairs of ref-words that have the same variable mapping. For example, $u_1 = {}^{\mathsf{y}}\triangleright\, \mathsf{a}^5\mathsf{b} \triangleleft^{\mathsf{y}} \# {}^{\mathsf{x}}\triangleright\, \mathsf{aa} \triangleleft^{\mathsf{x}} \mathsf{y} \in \mathcal{L}_{\text{ref}}(\langle\gamma_1\rangle_{\text{int}})$ and $u_2 = {}^{\mathsf{x}}\triangleright\, \mathsf{aa} \triangleleft^{\mathsf{x}} \# {}^{\mathsf{y}}\triangleright\, \mathsf{xaxb} \triangleleft^{\mathsf{y}} \mathsf{byy} \in \mathcal{L}_{\text{ref}}(\langle\gamma_2\rangle_{\text{int}})$ and, furthermore, since u_1 and u_2 have the same variable mapping $(\mathsf{aa}, \mathsf{a}^5\mathsf{b})$, they are witnesses for the conjunctive match $(w_1, w_2) = (\mathsf{aaa}^5\mathsf{b}, \mathsf{a}^5\mathsf{bb}(\mathsf{a}^5\mathsf{b})^2)$ of $\bar{\gamma}$, since $\text{deref}(u_1) = \mathsf{a}^5\mathsf{b}\#w_1$ and $\text{deref}(u_2) = \mathsf{aa}\#w_2$. On the other hand, $v_1 = {}^{\mathsf{y}}\triangleright\mathsf{a}\triangleleft^{\mathsf{y}}\#{}^{\mathsf{x}}\triangleright\mathsf{a}\triangleleft^{\mathsf{x}}\mathsf{y}$ and $v_2 = {}^{\mathsf{x}}\triangleright\mathsf{a}\triangleleft^{\mathsf{x}}\#{}^{\mathsf{y}}\triangleright\mathsf{xaxb}\triangleleft^{\mathsf{y}}\mathsf{by}$ are also ref-words from $\mathcal{L}_{\text{ref}}(\langle\alpha_1\rangle_{\text{int}})$ and $\mathcal{L}_{\text{ref}}(\langle\alpha_2\rangle_{\text{int}})$, respectively, but, even though $\text{deref}(v_1) = \mathsf{a}\#\mathsf{aa}$ and $\text{deref}(v_2) = \mathsf{a}\#\mathsf{a}^3\mathsf{bba}^3\mathsf{b}$, the tuple $(\mathsf{aa}, \mathsf{a}^3\mathsf{bba}^3\mathsf{b})$ is not a conjunctive match for $\bar{\alpha}$, since $\text{vmap}_{v_1}(\mathsf{y}) = \mathsf{a} \neq \mathsf{a}^3\mathsf{b} = \text{vmap}_{v_2}(\mathsf{y})$. We shall now generalise this idea to obtain a sound definition of the semantics of conjunctive xregex.

For any xregex $\gamma \in \text{XRE}_{\Sigma,\mathcal{X}_s}$, let $\triangledown_\gamma = \Pi_{x\in A}x\{\Sigma^*\}$, where $A \subseteq \mathcal{X}_s$ is the set of variables that have no definition in γ (the order of the definitions $\mathsf{x}\{\Sigma^*\}$ with $\mathsf{x} \in A$ in $\triangledown_\gamma$ shall be irrelevant), and we also define $\langle\gamma\rangle_{\text{int}} = \triangledown_\gamma\#\gamma$, where $\#$ is a new symbol with $\# \notin \Sigma$. Finally, a tuple $\bar{w} = (w_1, w_2, \ldots, w_m) \in (\Sigma^*)^m$ is a (*conjunctive*) *match* for $\bar{\alpha} = (\alpha_1, \alpha_2, \ldots, \alpha_m) \in (\text{XRE}_{\Sigma,\mathcal{X}_s})^m$ with variable mapping ψ if, for every $i \in [m]$, there is a ref-word $v_i = v_i'\#v_i'' \in \mathcal{L}_{\text{ref}}(\langle\alpha_i\rangle_{\text{int}})$ with $\text{deref}(v_i) = \text{deref}(v_i')\#w_i$ and $\text{vmap}_{v_i} = \psi$.

Conjunctive xregex behave similar to xregex in terms of how ε can be allocated to a variable. It might happen that α_i contains the definition of some variable x (and therefore $\triangledown_{\alpha_i}$ does not contain $\mathsf{x}\{\Sigma^*\}$), but the corresponding ref-word $v_i \in \mathcal{L}_{\text{ref}}(\langle\alpha_i\rangle_{\text{int}})$ does not contain a definition of x. This means that $\text{vmap}_{v_i}(\mathsf{x}) = \varepsilon$ and therefore all other v_j with $i \neq j$ must also allocate ε to x in their definitions $\mathsf{x}\{\Sigma^*\}$ contained in $\triangledown_{\alpha_j}$. On the other hand, it is possible that $\text{vmap}_{v_i}(\mathsf{x}) = \varepsilon$ even though v_i contains a definition of x.

For an $\bar{\alpha} \in \text{CXRE}$, $\mathcal{L}(\bar{\alpha})$ is the *set of conjunctive matches* for $\bar{\alpha}$. We say that conjunctive xregex $\bar{\alpha}$ and $\bar{\beta}$ are *equivalent* if $\mathcal{L}(\bar{\alpha}) = \mathcal{L}(\bar{\beta})$ (note that $\bar{\alpha}$ and $\bar{\beta}$ having the same dimension is necessary for this). We stress the fact that the order of the elements α_i of a conjunctive xregex $\bar{\alpha} = (\alpha_1, \alpha_2, \ldots, \alpha_m)$ is vital.

A special class of conjunctive xregex is $m\text{-CXRE}_{\Sigma,\emptyset}$, i. e., the class of all m-dimensional tuples of classical regular expressions. For every $\bar{\alpha} \in m\text{-CXRE}_{\Sigma,\emptyset}$, $\mathcal{L}(\bar{\alpha}) = \mathcal{L}(\bar{\alpha}[1]) \times \mathcal{L}(\bar{\alpha}[2]) \times \ldots \times \mathcal{L}(\bar{\alpha}[m])$.

Example 3.4. Consider the xregex

$$\alpha_1 = \mathsf{x}_2\{\mathsf{x}_1 \vee \mathsf{a}^*\}\mathsf{b}, \qquad \alpha_2 = \mathsf{x}_1\{(\mathsf{a}\vee\mathsf{b})^*\}\mathsf{x}_3\{\mathsf{c}^*\}\mathsf{bx}_3,$$
$$\alpha_3 = \mathsf{x}_2^*\mathsf{a}^*\mathsf{x}_1, \qquad \alpha_4 = \mathsf{x}_4\{\mathsf{a}^*\}\mathsf{bx}_4\mathsf{x}_1\{\mathsf{x}_2\mathsf{a}\}.$$

Then (α_2, α_4) is not a conjunctive xregex since $\alpha_2\alpha_4$ is not sequential, but both (α_3, α_4) and $(\alpha_1, \alpha_2, \alpha_3)$ are conjunctive xregex.

Obviously, $w_1 = \mathsf{aab}$, $w_2 = \mathsf{bbacbc}$ and $w_3 = \mathsf{aa}$ are matches for α_1, α_2 and α_3, respectively, with variable mappings $(\varepsilon, \mathsf{aa}, \varepsilon)$, $(\mathsf{bba}, \varepsilon, \mathsf{c})$ and $(\varepsilon, \varepsilon, \varepsilon)$, respectively. However, for (w_1, w_2, w_3) to be a conjunctive match for $(\alpha_1, \alpha_2, \alpha_3)$, there must be words u_1, u_2, u_3 such that the following w_i' match the $\langle\alpha_i\rangle_{\text{int}}$ all with the same variable mapping (u_1, u_2, u_3).

$$w_1' = u_1u_3\#w_1 \qquad \langle\alpha_1\rangle_{\text{int}} = \mathsf{x}_1\{\Sigma^*\}\mathsf{x}_3\{\Sigma^*\}\#\alpha_1$$
$$w_2' = u_2\#w_2 \qquad \langle\alpha_2\rangle_{\text{int}} = \mathsf{x}_2\{\Sigma^*\}\#\alpha_2$$
$$w_3' = u_1u_2u_3\#w_3 \qquad \langle\alpha_3\rangle_{\text{int}} = \mathsf{x}_1\{\Sigma^*\}\mathsf{x}_2\{\Sigma^*\}\mathsf{x}_3\{\Sigma^*\}\#\alpha_3$$

This can be easily seen to be impossible: from the fact that w_3' matches $\langle\alpha_3\rangle_{\text{int}}$ with variable mapping (u_1, u_2, u_3) it follows that u_1 cannot contain any occurrences of b, which makes it impossible for w_2' to match $\langle\alpha_2\rangle_{\text{int}}$ (since w_2 contains 3 occurrences of b).

However, it can be verified that $(\mathsf{abb}, \mathsf{abccbcc}, \mathsf{ababaaab})$ is a conjunctive match for $(\alpha_1, \alpha_2, \alpha_3)$ with variable mapping $(\mathsf{ab}, \mathsf{ab}, \mathsf{cc})$.

4 CONJUNCTIVE XREGEX PATH QUERIES

Finally, we can define conjunctive xregex path queries:

*Definition 4.1 (*CXRPQ*).* A *conjunctive xregex path query* (CXRPQ for short) is an $\mathfrak{R}$-CPQ $q = \bar{z} \leftarrow G_q$, where $\mathfrak{R}$ is the class of xregex, $G_q = (V_q, E_q)$ with $E_q = \{(x_i, \alpha_i, y_i) \mid i \in [m]\}$ and $\bar{\alpha} = (\alpha_1, \alpha_2, \ldots, \alpha_m)$ is a conjunctive xregex (which is also called *the conjunctive xregex of* q).

The semantics are defined by combining the semantics of conjunctive path queries and conjunctive xregex in the obvious way. More precisely, let $q = \bar{z} \leftarrow G_q$ be a CXRPQ, where $G_q = (V_q, E_q)$ with $E_q = \{(x_i, \alpha_i, y_i) \mid i \in [m]\}$, and let $\mathcal{D}$ be a graph database. Then $h : V_q \to V_\mathcal{D}$ is a matching morphism for q and $\mathcal{D}$ if there is a conjuctive match $(w_1, w_2, \ldots, w_m)$ of $\bar{\alpha}$, such that, for every $i \in [m]$, $\mathcal{D}$ contains a path from $h(x_i)$ to $h(y_i)$ labelled with w_i. Note that this definition of a matching morphism h for CXRPQ also implies definitions for $q_h(\mathcal{D})$, $q(\mathcal{D})$, $\mathcal{D} \models q$, $[\![q]\!]$ and $[\![\text{CXRPQ}]\!]$.

Next, we observe that evaluating CXRPQ is surprisingly difficult. Let $\Delta = \{\mathsf{a}, \mathsf{b}, \#\}$ and let $\alpha_{\text{ni}} = \#\mathsf{z}\{(\mathsf{a}\vee\mathsf{b})^*\}(\#\#\mathsf{z})^*\#\#\# \in \text{XRE}_{\Delta,\mathcal{X}_s}$.

Theorem 4.2. *Deciding whether a given graph-database over $\Sigma \supset \Delta$ contains a path labelled with some $w \in \mathcal{L}(\alpha_{ni})$ is* PSpace*-hard.*

Proof Sketch. Given NFA $M_1, \ldots, M_k$ over alphabet $\{\mathsf{a}, \mathsf{b}\}$ with initial states $q_{0,i}$ and accepting states $q_{f,i}$, we construct a graph database $\mathcal{D}$ over alphabet Δ as follows. We add new nodes s and t with a $\#$-labelled arc from s to $q_{0,1}$, a $\#\#\#$-labelled path from $q_{f,k}$ to t, and $\#\#$-labelled paths from $q_{f,i}$ to $q_{f,i+1}$ for every $i \in [k-1]$. Then in $\mathcal{D}$ there is a path labelled with $w \in \mathcal{L}(\alpha_{\text{ni}})$ if and only if there is some $v \in \bigcap_{i=1}^k \mathcal{L}(M_i)$. □

Theorem 4.2 constitutes a rather strong negative result: CXRPQ-Bool-Eval is PSpace-hard in data complexity, even for a single-edge query with a conjunctive xregex $\alpha \in \text{XRE}_{\Sigma,\mathcal{X}_s}$, where $|\Sigma| = 3$ and $|\mathcal{X}_s| = 1$. However, as explained in the introduction, α_{ni} gives a hint how to restrict CXRPQ to obtain more tractable fragments. Such fragments are investigated in the following Sections 5 and 6.

5 VARIABLE-STAR FREE CXRPQ

An xregex α is *variable-star free* (*vstar-free*, for short), if in α no +-operator applies to a variable reference.[1] See Example 5.3 for an illustration. A conjunctive xregex $\bar{\alpha}$ of dimension m is vstar-free if, for every $i \in [m]$, $\bar{\alpha}[i]$ is vstar-free, and a $q \in$ CXRPQ is vstar-free, if its conjunctive xregex is vstar-free. Finally, let $\text{CXRPQ}^{\text{vsf}}$ be the class of vstar-free CXRPQ.

The main result of this section is as follows.

Theorem 5.1. $\text{CXRPQ}^{\text{vsf}}$*-Bool-Eval is in* ExpSpace *w. r. t. combined-complexity and* NL*-complete w. r. t. data complexity.*

Before discussing at greater length how to prove the upper bounds of Theorem 5.1, we first observe the following lower bound.

Theorem 5.2. $\text{CXRPQ}^{\text{vsf}}$*-Bool-Eval is* PSpace*-hard in combined complexity, even for single-edge queries with xregex* $\alpha \in \text{XRE}_{\Sigma,\mathcal{X}_s}$*, where* $|\Sigma| = 3$ *and* $|\mathcal{X}_s| = 1$.

Proof Sketch. This can be shown similarly to the proof of Theorem 4.2, with the only difference that we use the xregex $\alpha^{\text{k}}_{\text{ni}} = \#\text{z}\{(\text{a} \vee \text{b})^*\}\,(\#\#\,\text{z})^{k-1}\#\#\#$, where k is the number of NFAs. □

The upper bounds require more work. Before we can give an intuitive idea of our procedure, we have to introduce some more restrictions of xregex. An xregex α is

- *variable-alternation free* (*valt-free*) if, for every subexpression $(\beta_1 \vee \beta_2)$ of α, neither β_1 nor β_2 contain any variable definition or variable reference.
- *variable-simple* if it is vstar-free and valt-free.
- *simple* if it is variable-simple and every variable definition $\text{x}\{\gamma\}$ is *basic*, i. e., γ is a classical regular expression or a single variable reference.
- in *normal form* if $\alpha = \alpha_1 \vee \alpha_2 \vee \ldots \vee \alpha_m$ and, for every $i \in [m]$, α_i is simple.

Let us clarify these definitions with some intuitive explanations and examples. The condition of being vstar-free or valt-free can also be interpreted as follows: an xregex is vstar-free (or valt-free), if every subtree of its syntax tree rooted by a node for a +-operation (for a $\vee$-operation, respectively) does not contain any nodes for variable definitions or references. If this is true for all +-operation nodes *and* all $\vee$-operation nodes, then the xregex is variable-simple. Equivalently, α is variable-simple if $\alpha = \beta_1\beta_2 \ldots \beta_k$, where each β_i is a classical regular expression, a variable reference or a variable definition $\text{x}\{\gamma\}$, where γ is also variable-simple. If additionally each variable definition $\text{x}\{\gamma\}$ is such that γ is a classical regular expressions or a single variable reference, then α is simple. Finally, an xregex is in normal form, if it is an alternation of simple xregex.

Example 5.3. $\text{x}\{\text{a}^*\}(\text{bx}(\text{c} \vee \text{a}))^*\text{b}$ is not vstar-free, but valt-free; $\text{x}\{\text{a}^*\}\text{y}((\text{bx}) \vee (\text{ca}))\text{b}^*\text{y}$ is vstar-free, but not valt-free. The xregex $\text{ax}\{(\text{b} \vee \text{c})^*\text{by}\{\text{dxa}^*\}\}\text{bxa}^*\text{z}\{\text{d}^*\}\text{zy}$ is variable-simple, but not simple, and $\text{ax}\{(\text{b} \vee \text{c})^*\text{da}\}\text{bxa}^*\text{y}\{\text{z}\}\text{xy}$ is simple.

We extend these restrictions to conjunctive xregex and CXRPQ in the obvious way, i. e., as also done above for vstar-free xregex.

[1] Since we use the Kleene-star r^* only as short hand form for $r^+ \vee \varepsilon$, the term "variable-*plus* free" seems more appropriate. We nevertheless use the term "star free", since it is much more common in the literature on regular expressions and languages.

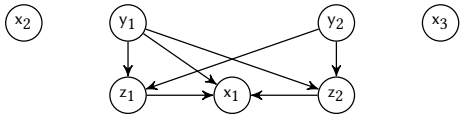

Figure 3: The DAG $G_{\bar{\gamma}}$ at Step 3.

We can now give a high-level "road map" for the proof of Theorem 5.1. As an important building block, we first prove that CXRPQ that are simple (in the sense defined above) can be evaluated in nondeterministics space $\text{O}(|q| \log(|\mathcal{D}|) + |q| \log(|q|))$ (see Lemma 5.4 below). Then, in Subsection 5.1, we show that every variable star-free conjunctive xregex can be transformed into an equivalent one in normal form (Lemmas 5.6, 5.7 and 5.8), which also means that $\text{CXRPQ}^{\text{vsf}}$ can be transformed into equivalent CXRPQ in normal form. A CXRPQ in normal form has a conjunctive xregex whose elements are alternations of simple xregex; thus, they can be evaluated by using Lemma 5.4. This directly yields a nondeterministic log-space algorithm with respect to data complexity. For combined complexity, we have to deal with the problem that the normal form causes a double exponential size blow-up. In order to get the ExpSpace upper bound, we observe that the first exponential blow-up can be handled by nondeterminism.

After this, in Subsection 5.3, we discuss more thoroughly what exactly causes the exponential blow-up in our normal form and how it can be avoided. Our observations can be transformed into the fragment $\text{CXRPQ}^{\text{vsf,fl}}$ (already mentioned in the introduction) that avoids the exponential blow-up, and therefore has a PSpace-complete evaluation problem in combined complexity.

We proceed with Lemma 5.4, which can be proven similar as the corresponding result for CRPQ (see, e. g., [8]).

Lemma 5.4 (★). *Given a Boolean* $q \in$ CXRPQ *that is simple and a graph-database* $\mathcal{D}$*, we can nondeterministically decide whether or not* $\mathcal{D} \models q$ *in space* $\text{O}(|q| \log(|\mathcal{D}|))$.

5.1 Construction of the Normal Form

Theorem 5.5. *Let* $\bar{\alpha} \in m\text{-CXRE}_{\Sigma,\mathcal{X}_s}$ *be vstar-free. Then there is an equivalent* $\bar{\beta} \in m\text{-CXRE}_{\Sigma,\mathcal{X}'_s}$ *in normal form with* $|\bar{\beta}| = \text{O}\left(2^{2^{poly|\bar{\alpha}|}}\right)$.

Our construction to transform vstar-free $\bar{\alpha} \in$ CXRE in normal form has three main steps, represented by Lemmas 5.6, 5.7 and 5.8. Before stating these lemmas, we will first explain the three main steps on an intuitive level with an example.

Let $\bar{\gamma} = (\gamma_1, \gamma_2)$ be a variable star free conjunctive xregex with

$$\gamma_1 = \text{x}\{\text{a}^*\text{y}\{\text{b}^*\}\text{az}\} \vee (\text{x}\{\text{b}^*\} \cdot (\text{z} \vee \text{y}\{\text{c}^*\})$$
$$\gamma_2 = (\text{a}^* \vee \text{x}) \cdot \text{z}\{\text{y} \cdot (\text{a} \vee \text{b})\}$$

Step 1: We will "multiply-out" alternations with definitions or variables, which transforms γ_1 and γ_2 into alternations of variable-simple xregex.

$$\gamma_1 = [\text{x}\{\text{a}^*\text{y}\{\text{b}^*\}\text{az}\}] \vee [\text{x}\{\text{b}^*\} \cdot \text{z}] \vee [\text{x}\{\text{b}^*\} \cdot \text{y}\{\text{c}^*\}]$$
$$\gamma_2 = [\text{a}^* \cdot \text{z}\{\text{y} \cdot (\text{a} \vee \text{b})\}] \vee [\text{x} \cdot \text{z}\{\text{y} \cdot (\text{a} \vee \text{b})\}]$$

This only results in an equivalent xregex because $\bar{\gamma}$ is vstar-free. This transformation may cause an exponential size blow-up.

Step 2: Next, we relabel the variables in such a way, that every variable has at most one definition in $\bar{\gamma}$. This also requires that variable references are substituted by several variable references (e. g., references of x by $x_1x_2x_3$). The size blow-up is at most quadratic.

$$\gamma_1 = [x_1\{a^*y_1\{b^*\}az_1z_2\}] \vee [x_2\{b^*\} \cdot z_1z_2] \vee [x_3\{b^*\} \cdot y_2\{c^*\}]$$

$$\gamma_2 = [a^* \cdot z_1\{y_1y_2 \cdot (a \vee b)\}] \vee [x_1x_2x_3 \cdot z_2\{y_1y_2 \cdot (a \vee b)\}]$$

Step 3: We are now able to remove non-basic variable definitions, which remains to be done in order to obtain the normal form. The main idea is to split γ of a variable definition $x\{\gamma\}$ into smaller parts and (if they are not already definitions) store them in new variables. Instead of x, we can then reference the new variables, which allows to remove x. This, however, has to be done in the order given by the DAG induced by the relation $\preceq_{\bar{\alpha}}$ (see Figure 3). More precisely, we first consider the roots of the DAG representing variables x_2, x_3, y_1, y_2, which have basic variable definitions and therefore can be left unchanged. After deleting them from the DAG, the nodes for z_1, z_2 are roots. The non-basic definition $z_1\{y_1y_2 \cdot (a \vee b)\}$ is replaced by $u_1\{y_1\}u_2\{y_2\}u_3\{a \vee b\}$ and $z_2\{y_1y_2 \cdot (a \vee b)\}$ is replaced by $u_4\{y_1\}u_5\{y_2\}u_6\{a \vee b\}$. All references for z_1 and z_2 are replaced by $\bar{z}_1 = u_1u_2u_3$ and $\bar{z}_2 = u_4u_5u_6$, respectively.

$$\gamma_1 = [x_1\{a^*y_1\{b^*\}a\bar{z}_1\bar{z}_2\}] \vee [x_2\{b^*\} \cdot \bar{z}_1\bar{z}_2] \vee [x_3\{b^*\} \cdot y_2\{c^*\}]$$

$$\gamma_2 = [a^* \cdot u_1\{y_1\}u_2\{y_2\}u_3\{a \vee b\}] \vee [x_1x_2x_3 \cdot u_4\{y_1\}u_5\{y_2\}u_6\{a \vee b\}]$$

Finally, after deleting the nodes for z_1 and z_2 from the DAG, the node for x_1 is the only root left. Therefore, we can now replace

$$x_1\{a^*y_1\{b^*\}a\bar{z}_1\bar{z}_2\} = x_1\{a^*y_1\{b^*\}au_1u_2u_3u_4u_5u_6\}$$

by the following concatenation of variable definitions:

$$u_7\{a^*\}y_1\{b^*\}u_8\{a\}u_9\{u_1\}u_{10}\{u_2\}u_{11}\{u_3\}$$
$$u_{12}\{u_4\}u_{13}\{u_5\}u_{14}\{u_6\}\,.$$

Moreover, all references of x_1 are replaced by $\bar{x}_1 = u_7y_1u_8u_9 \dots u_{14}$. The thus obtained conjunctive xregex is in fact in normal form.

We shall discuss a few particularities. While Steps 1 and 2 are more or less straightfoward, Step 3 is more complicated (latter we also provide a proof sketch for the correctness of this step). In particular, it is not easy to see why it is necessary to use new variables u with definition $u\{x\}$ instead of just using the existing x. For example, if a variable definition $z\{y_1a^*y_2\}$ would be replaced by $y_1u\{a^*\}y_2$ and references of z by y_1uy_2, then y_1uy_2 must refer to ε in the case that z is not instantiated, which is not the case if y_1 is instantiated elsewhere. So we use $u_1\{y_1\}u_2\{a^*\}u_3\{y_2\}$ and replace references of z by $u_1u_2u_3$, which means that if z is not instantiated, then also none of the u_1, u_2 and u_3 are instantiated in the modified xregex. Why it is necessary to apply these modifications according to the order given by $\preceq_{\bar{\gamma}}$ will also be discussed in the proof sketch of the corresponding lemma further below. The possible exponential size blow-up of this step is discussed in Subsection 5.3.

We now give the lemmas for the three steps of the construction. The correctness of the simpler first two steps is sufficiently sketched by the example. For the third, we provide a proof sketch.

Lemma 5.6 (⋆). ***(Step 1)*** *Let $\bar{\alpha} \in m\text{-CXRE}_{\Sigma,\mathcal{X}_s}$ be vstar-free. Then there is an equivalent $\bar{\beta} \in m\text{-CXRE}_{\Sigma,\mathcal{X}_s}$ with $|\bar{\beta}| = O(2^{|\bar{\alpha}|})$, such that each $\bar{\beta}[i]$ is an alternation of variable-simple xregex.*

Lemma 5.7 (⋆). ***(Step 2)*** *Let $\bar{\alpha} \in m\text{-CXRE}_{\Sigma,\mathcal{X}_s}$ such that, for every $i \in [m]$, $\bar{\alpha}[i]$ is an alternation of variable-simple xregex. Then there is an equivalent $\bar{\beta} \in m\text{-CXRE}_{\Sigma,\mathcal{X}'_s}$ such that, for every $i \in [m]$, $\bar{\beta}[i]$ is an alternation of variable-simple xregex, and every $x \in \mathcal{X}'_s$ has at most one definition in $\bar{\beta}$. Moreover, $|\bar{\beta}| = O(|\bar{\alpha}|^2)$.*

Lemma 5.8. ***(Step 3)*** *Let $\bar{\alpha} \in m\text{-CXRE}_{\Sigma,\mathcal{X}_s}$ such that, for every $i \in [m]$, $\bar{\alpha}[i]$ is an alternation of variable-simple xregex, and every $x \in \mathcal{X}_s$ has at most one definition in $\bar{\alpha}$. Then there is an equivalent $\bar{\beta} \in m\text{-CXRE}_{\Sigma,\mathcal{X}'_s}$ in normal form with $|\bar{\beta}| = O(|\bar{\alpha}|^{|\mathcal{X}_s|+1})$.*

Proof Sketch. Let $\bar{\alpha} = (\alpha_1, \alpha_2, \dots, \alpha_m)$, and, for every $i \in [m]$, let $\alpha_i = \alpha_{i,1} \vee \alpha_{i,2} \vee \dots \vee \alpha_{i,t_i}$, where, for every $j \in [t_i]$, $\alpha_{i,j}$ is variable-simple, and, for every $x \in \mathcal{X}_s$, there is at most one definition for x in $\bar{\alpha}$. The core of the whole procedure to obtain $\bar{\beta}$ is the following *main modification step*, which can be applied to any variable definition $z\{\gamma\}$ of $\bar{\alpha}$.

Since γ is variable-simple, $\gamma = \gamma_1\gamma_2 \dots \gamma_p$ such that each γ_ℓ with $\ell \in [p]$ is either a classical regular expression, a variable definition, or a variable reference. For every $\ell \in [p]$, we set $\gamma'_\ell = \gamma_\ell$, if $\gamma_\ell = y_\ell\{\dots\}$, and we set $\gamma'_\ell = y_\ell\{\gamma_\ell\}$ for a *new* variable $y_\ell \notin \mathcal{X}_s$, if γ_ℓ is a classical regular expression or $\gamma_\ell \in \mathcal{X}_s$. Note that $\gamma'_1\gamma'_2 \dots \gamma'_p = y_1\{\dots\}y_2\{\dots\} \dots y_p\{\dots\}$. Next, we replace $z\{\gamma\}$ by $\gamma'_1\gamma'_2 \dots \gamma'_p$, and we replace all references z by $y_1y_2 \dots y_p$. Let the thus modified version of $\bar{\alpha}$ be denoted by $\bar{\alpha}' = (\alpha'_1, \alpha'_2, \dots, \alpha'_m)$.

Formally proving $\mathcal{L}(\bar{\alpha}) = \mathcal{L}(\bar{\alpha}')$ is technically involved, since the construction changes the ref-language of $\bar{\alpha}$. Therefore, we have to show how ref-words of $\bar{\alpha}$ translate into suitable ref-words $\bar{\alpha}$, and vice versa. Informally speaking, we have to factorise the image of z into images of the variables $y_1, y_2, \dots, y_p$ (or concatenating the single images of variables $y_1, y_2, \dots, y_p$ into to one image of variable z, respectively). For this, the property provided by Lemma 5.7 is crucial, since it means that the variables y_i that have not been newly introduced by the main modification step are also not used in any other way in $\bar{\alpha}$.

If we apply the main modification step to a non-basic variable definition $z\{\gamma\}$, then it will be removed and all variable definitions that are introduced are basic. However, we also replace references of z by $y_1y_2 \dots y_p$, which may turn a basic variable definition $x\{z\}$ into a new *non-basic* variable definition $x\{y_1y_2 \dots y_p\}$. To deal with this problem, we construct the DAG $G_{\bar{\alpha}}$ with node set $\mathcal{X}_s$ and edge-relation $\preceq_{\bar{\alpha}}$, and then repeatedly apply the main modification step only to roots of the DAG, which are afterwards deleted.

Each application of the main modification step will replace single variable references of z by concatenations $y_1y_2 \dots y_p$ of variable references, where p corresponds to the size of some other variable reference. However, since we apply the main modification steps as given by the DAG $G_{\bar{\alpha}}$, we know that p can never exceed the size of an original variable definition that was already present in $\bar{\alpha}$, so $p \leq |\bar{\alpha}|$, which implies the claimed single exponential size bound. Examples that actually yield an exponential size blow-up can be easily found (see the example in Subsection 5.3). □

5.2 Proof Sketch of Theorem 5.1

Theorem 5.1 is a consequence from the following lemma.

Lemma 5.9. *Given a Boolean $q \in \text{CXRPQ}^{\text{vsf}}$ and a graph database $\mathcal{D}$, we can nondeterministically check whether $\mathcal{D} \models q$ in space $O(2^{poly(|q|)} \log(|\mathcal{D}|))$.*

Proof Sketch. We transform the conjunctive xregex $\bar{\alpha}$ of q into normal form according to the construction of Subsection 5.1, but instead of performing Step 1, we nondeterministically make each $\bar{\alpha}[i]$ valt-free as follows: as long as $\bar{\alpha}[i]$ is not valt-free, we choose some subexpression $(\gamma_1 \vee \gamma_2)$ where γ_1 or γ_2 contains a variable definition or a variable reference, and we nondeterministically replace it by γ_1 or γ_2. Each component of the thus obtained conjunctive xregex is even variable-simple, and not only an alternation of variable-simple xregex; thus, Step 2 and 3 yield a $q' \in \text{CXRPQ}$ that is simple, and $|q'| = O(2^{\text{poly}(|q|)})$. Finally, we nondeterministically check whether $\mathcal{D} \models q'$ with Lemma 5.4 in space $O(2^{\text{poly}(|q|)} \log(|\mathcal{D}|)))$. □

5.3 PSpace Combined Complexity

Step 3 of the normal form construction (Lemma 5.8) works for $\bar{\alpha}$, where, for every $i \in [m]$, $\bar{\alpha}[i]$ is an alternation of variable-simple xregex and every $x \in \mathcal{X}_s$ has at most one definition. However, in the proof of Lemma 5.9, when we apply Step 3, we can make the much stronger assumption that $\bar{\alpha}$ is even variable-simple and every $x \in \mathcal{X}_s$ has at most one definition in $\bar{\alpha}$. Unfortunately, as illustrated by the following example, this does not make any difference with respect to the possible exponential size blow-up caused by Step 3.

$$\alpha = x_1\{a\}x_2\{x_1x_1\}x_3\{x_2x_2\}x_4\{x_3x_3\}\ldots x_n\{x_{n-1}x_{n-1}\}$$

is a conjunctive xregex that is variable-simple and every variable has at most one definition. However, the procedure of Lemma 5.8 will apply the main modification step with respect to all variables $x_2, x_3, \ldots, x_n$ in this order (note that $G_{\bar{\alpha}}$ is a path), which replaces each references of x_2 by 2 variables, each references of x_4 by 4 variables, each references of x_5 by 8 variables, and so on.

The crucial point seems to be non-basic definitions for variables with references in other non-basic definitions. If we restrict vstar-free conjunctive xregex accordingly, then the exponential size blow-up does not occur in Step 3 of the normal form construction.

A variable $x \in \mathcal{X}_s$ is *flat* (in some $\bar{\alpha} \in \text{CXRE}_{\Sigma,\mathcal{X}_s}$) if its definition is basic, or it has no reference in any other definition. For example, let $\alpha_1 = ub^*x\{y\{a^*\}(a \vee b)^*zy\}$ and $\alpha_2 = u\{cbz\{a^*(b \vee ca)\}\}ax$, then in $\bar{\alpha} = (\alpha_1, \alpha_2)$ every variable is flat. Finally, let $\text{CXRPQ}^{\text{vsf,fl}}$ be the class of vstar-free CXRPQ with only flat variables.

Theorem 5.10 (★). $\text{CXRPQ}^{\text{vsf,fl}}$*-Bool-Eval is* PSpace*-complete in combined complexity.*

6 CXRPQ WITH BOUNDED IMAGE SIZE

The fragment to be defined in this section directly follows from the definition of the subsets $\mathcal{L}^{\leq k}(\alpha) \subseteq \mathcal{L}(\alpha)$ that contain the words that match xregex α with a witness v such that $|\text{vmap}_v(x)| \leq k$ for every $x \in \mathcal{X}_s$. This can be easily done as follows.

Let $\alpha \in \text{XRE}_{\Sigma,\mathcal{X}_s}$ with $\mathcal{X}_s = \{x_1, x_2, \ldots, x_n\}$ and $\bar{v} \in (\Sigma^*)^n$. We define $\mathcal{L}^{\bar{v}}_{\text{ref}}(\alpha) = \{u \in \mathcal{L}_{\text{ref}}(\alpha) \mid \text{vmap}_{u,\mathcal{X}_s} = \bar{v}\}$ and $\mathcal{L}^{\bar{v}}(\alpha) = \text{deref}(\mathcal{L}^{\bar{v}}_{\text{ref}}(\alpha))$. For every $k \geq 1$, let $\mathcal{L}^{\leq k}_{\text{ref}}(\alpha) = \bigcup_{\bar{v} \in (\Sigma^{\leq k})^n} \mathcal{L}^{\bar{v}}_{\text{ref}}(\alpha)$. Finally, we define $\mathcal{L}^{\leq k}(\alpha) = \text{deref}(\mathcal{L}^{\leq k}_{\text{ref}}(\alpha))$.

The notions $\mathcal{L}^{\bar{v}}_{\text{ref}}$ and $\mathcal{L}^{\leq k}_{\text{ref}}$ also extend to conjunctive xregex in the obvious way, and therefore to CXRPQ as follows. For a $q \in \text{CXRPQ}$ with conjunctive xregex $\bar{\alpha} \in \text{CXRE}_{\Sigma,\mathcal{X}_s}$, a $\bar{v} \in (\Sigma^*)^n$, and a graph database $\mathcal{D}$, by $q^{\bar{v}}(\mathcal{D})$ we denote the subset of $q(\mathcal{D})$ that contains those $q_h(\mathcal{D}) \in q(\mathcal{D})$, where h is a matching morphism with respect to some matching words $\bar{w} = (w_1, w_2, \ldots, w_m) \in \mathcal{L}^{\bar{v}}(\bar{\alpha})$; $q^{\leq k}(\mathcal{D})$ is defined analogously by restricting the matching words to be from $\mathcal{L}^{\leq k}(\bar{\alpha})$. In the Boolean case, we also set $\mathcal{D} \models^{\bar{v}} q$ and $\mathcal{D} \models^{\leq k} q$ to denote that $q^{\bar{v}}(\mathcal{D})$ or $q^{\leq k}(\mathcal{D})$, respectively, contains the empty tuple.

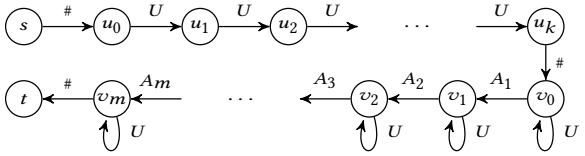

Figure 4: Sketch for the reduction from Theorem 6.2.

The above defined restrictions do not restrict the class CXRPQ (as it is the case for $\text{CXRPQ}^{\text{vsf}}$ considered in Section 5), but rather how the results of queries from CXRPQ are defined. However, it shall be convenient to allow a slight abuse of notation and define, for every $k \in \mathbb{N}$, the class $\text{CXRPQ}^{\leq k}$, which is the same as CXRPQ, but for every $q \in \text{CXRPQ}^{\leq k}$ it is understood that $q(\mathcal{D})$ actually means $q^{\leq k}(\mathcal{D})$ (and $\mathcal{D} \models q$ actually means $\mathcal{D} \models^{\leq k} q$). In this way, for every $k \in \mathbb{N}$, also the problems $\text{CXRPQ}^{\leq k}$-Bool-Eval and $\text{CXRPQ}^{\leq k}$-Check are defined.

For the classes $\text{CXRPQ}^{\leq k}$, we can show the following upper complexity bounds (which shall be discussed in Subsection 6.1).

Theorem 6.1. *For every $k \in \mathbb{N}$, $\text{CXRPQ}^{\leq k}$-Bool-Eval is in* NP *w. r. t. combined complexity and in* NL *w. r. t. data complexity.*

Since matching lower bounds directly carry over from CRPQs, the evaluation complexity of $\text{CXRPQ}^{\leq k}$ and CRPQ seems to be the same. However, we can state a much stronger hardness result, which points out an important difference between $\text{CXRPQ}^{\leq k}$ and CRPQ in terms of evaluation complexity: for $\text{CXRPQ}^{\leq 1}$, we have NP-hardness in combined complexity even for single-edge graph patterns (with even simple xregex). This is not the case for CRPQ, which can be evaluated in polynomial-time if the structure of the underlying graph pattern is acyclic (see [6, 8, 10]).

Theorem 6.2. $\text{CXRPQ}^{\leq k}$*-Bool-Eval is* NP*-hard in combined complexity, even for $k = 1$ and single-edge queries with simple xregex $\alpha \in \text{XRE}_{\Sigma,\mathcal{X}_s}$ and $|\Sigma| = 3$.*

Proof Sketch. The problem Hitting Set is to decide, for given subsets $A_1, A_2, \ldots, A_m$ of some universe $U = \{z_1, z_2, \ldots, z_n\}$ and $k \in \mathbb{N}$, whether there is a *hitting set* of size at most k, i. e., a set $B \subseteq U$ with $|B| \leq k$ and $B \cap A_i \neq \emptyset$ for every $i \in [m]$. Let $\mathcal{D} = (V_{\mathcal{D}}, E_{\mathcal{D}})$ be the graph database over $\Sigma = \{a, b, \#\}$, which is illustrated in Figure 4 (an arc labelled with U or A_i stands for paths labelled by ba^jb for every $z_j \in U$ or $z_j \in A_i$, respectively). Finally, let the Boolean $q \in \text{CXRPQ}$ be defined by a single-edge graph pattern with xregex $\alpha = \# \prod_{i=1}^{(n+2)k} x_i\{a \vee b \vee \varepsilon\} \# \left(\prod_{i=1}^{(n+2)k} x_i\right)^m \#$.

If in $\mathcal{D}$ there is a path p labelled with a word $w \in \mathcal{L}(\alpha)$, then, due to the occurrences of # in α, this must be a path from s to t. Moreover, by the structure of $\mathcal{D}$, we have that $w = \#w_1\#w_2\#$, where $w_1 =$

$\langle z_{j_1}\rangle\langle z_{j_2}\rangle \dots \langle z_{j_k}\rangle$ and $w_2 = u_1\langle z_{r_1}\rangle u_2\langle z_{r_2}\rangle u_3 \dots u_m\langle z_{r_m}\rangle u_{m+1}$, such that the set $\{z_{r_1}, z_{r_2}, \dots, z_{r_m}\}$ is a hitting set. Finally, since $\mathcal{L}(\alpha)$ contains exactly the words $w = \#w_1\#w_2\#$, where $w_1 \in \{\mathsf{a},\mathsf{b}\}^*$ with $|w_1| \leq (n+2)k$, and $w_2 = (w_1)^m$, we can conclude that $\{z_{r_1}, z_{r_2}, \dots, z_{r_m}\} \subseteq \{z_{j_1}, z_{j_2}, \dots, z_{j_k}\}$. Hence, we can conclude that $|\{z_{r_1}, z_{r_2}, \dots, z_{r_m}\}| \leq k$.

If $\{z_{j_1}, z_{j_2}, \dots, z_{j_k}\}$ is a hitting set of size k, then we can construct a word $w = \#w_1\#w_2\#$ with $w_1 = \langle z_{j_1}\rangle\langle z_{j_2}\rangle \dots \langle z_{j_k}\rangle$ and $w_2 = (w_1)^m$. It can be easily verified that $w \in \mathcal{L}(\alpha)$. Moreover, in $\mathcal{D}$ there is obviously a path from s to v_0 labelled with $\#w_1\#$ and, since $\{z_{j_1}, z_{j_2}, \dots, z_{j_k}\}$ is a hitting set and each of the m occurrences of factor w_1 in w_2 contains a factor $\langle z_{j_i}\rangle$ for every $i \in [k]$, w_2 can be factorised into $w_2 = u_1\langle z_{r_1}\rangle u_2\langle z_{r_2}\rangle u_3 \dots u_m\langle z_{r_m}\rangle u_{m+1}$, such that, for every $i \in [m]$, $\langle z_{r_i}\rangle \in A_i$. Thus, there is a path from v_0 to t labelled with $w_2\#$ and therefore a path from s to t labelled with w.

Note that, for every $k \geq 1$, $\mathcal{L}^{\leq k}(\alpha) = \mathcal{L}(\alpha)$; thus, it does not matter for what k we interpret q to be a $\mathrm{CXRPQ}^{\leq k}$. □

6.1 Upper Bounds

The main building block of the NP-algorithm is that if we have fixed some variable mapping $\bar{v} = (v_1, v_2, \dots, v_m)$, then the subset $\mathcal{L}^{\bar{v}}(\bar{\alpha})$ of $\mathcal{L}(\bar{\alpha})$ can be represented by a conjunctive xregex without variable definitions, i. e., a tuple of classical regular expressions (see Lemma 6.3). However, the corresponding procedure is not as simple as "replace each $\mathsf{x}_i\{\dots\}$ and x_i by v_i". We shall now illustrate this with an example and some intuitive explanations. Let $\alpha = (\alpha_1, \alpha_2) \in \mathrm{CXRE}_{\Sigma,\mathcal{X}_s}$ be defined by

$$\alpha_1 = \mathsf{x}_3\{\mathsf{x}_1\{\mathsf{ca}^*\mathsf{c}\}\mathsf{x}_2^*\} \vee [\,(\mathsf{x}_1\{\mathsf{cb}^*\} \vee \mathsf{x}_1\{\mathsf{x}_4\mathsf{c}^*\})(\mathsf{b} \vee \mathsf{x}_2^*)\mathsf{x}_3\{\mathsf{x}_1\mathsf{x}_2\mathsf{x}_1^*\}\,],$$

$$\alpha_2 = (\mathsf{x}_1 \vee \mathsf{x}_2)^*\mathsf{x}_4\{(\mathsf{b} \vee \mathsf{c})^*\mathsf{x}_2^*\}\mathsf{x}_2\{(\mathsf{a} \vee \mathsf{b})^*\mathsf{a}\},$$

and let $\bar{v} = (v_1, \dots, v_4) = (\mathsf{ca}, \mathsf{a}, \mathsf{caaca}, \mathsf{ca})$.

For computing $\beta = (\beta_1, \beta_2) \in \mathrm{CXRE}_{\Sigma,\emptyset}$ with $\mathcal{L}(\bar{\beta}) = \mathcal{L}^{\bar{v}}(\bar{\alpha})$, replacing $\mathsf{x}_i\{\gamma\}$ by v_i can only be correct, if γ can produce v_i (in a match with variable mapping $\bar{v}$). So we should treat the variable definitions and references of γ as their intended images and then check whether the thus obtained classical regular expression can produce v_i. For example, we transform $\mathsf{x}_3\{\mathsf{x}_1\{\mathsf{ca}^*\mathsf{c}\}\mathsf{x}_2^*\}$ in α_1 to $\mathsf{x}_3\{v_1v_2^*\} = \mathsf{x}_3\{\mathsf{caa}^*\}$ and check $v_3 = \mathsf{ca} \in \mathcal{L}(\mathsf{caa}^*)$. However, this assumes that $\mathsf{x}_1\{\mathsf{ca}^*\mathsf{c}\}$ can really produce v_1, which is not the case, so we should rather remove $\mathsf{x}_3\{\mathsf{x}_1\{\mathsf{ca}^*\mathsf{c}\}\mathsf{x}_2^*\}$ altogether, since it can never be involved in a conjunctive match from $\mathcal{L}^{\bar{v}}(\bar{\alpha})$. Hence, we also need to cut whole alternation branches in $\bar{\alpha}$, and, moreover, we have to make sure that if $\mathsf{x}_i \neq \varepsilon$, then at least one definition of x_i is necessarily instantiated, while this is not required if $\mathsf{x}_i = \varepsilon$. Let us illustrate the correct transformations with this example.

For every variable definition $\mathsf{x}_i\{\gamma\}$, where γ is a classical regular expression, we check whether $v_i \in \mathcal{L}(\gamma)$, and we mark the definition accordingly with $\mathtt{1}$ or $\mathtt{0}$, respectively. Then, we cut all branches that necesssarily produce a definition marked with $\mathtt{0}$. This transform α_1 as follows

$$\begin{aligned}\alpha_1 &\rightsquigarrow \mathsf{x}_3\{\mathsf{x}_1\{\mathtt{0}\}\mathsf{x}_2^*\} \vee [\,(\mathsf{x}_1\{\mathtt{0}\} \vee \mathsf{x}_1\{\mathtt{1}\})(\mathsf{b} \vee \mathsf{x}_2^*)\mathsf{x}_3\{\mathtt{1}\}\,]\\ &\rightsquigarrow \mathsf{x}_1\{\mathtt{1}\}(\mathsf{b} \vee \mathsf{x}_2^*)\mathsf{x}_3\{\mathtt{1}\}.\end{aligned}$$

If there were variable definitions left, we would repeat this step, but for checking whether v_i can be generated by γ, we would treat variable definitions marked with $\mathtt{1}$ as their intended images. With respect to α_2, we get

$$(\mathsf{x}_1 \vee \mathsf{x}_2)^*\mathsf{x}_4\{(\mathsf{b} \vee \mathsf{c})^*\mathsf{x}_2^*\}\mathsf{x}_2\{(\mathsf{a} \vee \mathsf{b})^*\mathsf{a}\} \rightsquigarrow (\mathsf{x}_1 \vee \mathsf{x}_2)^*\mathsf{x}_4\{\mathtt{1}\}\mathsf{x}_2\{\mathtt{1}\}.$$

Note that for $\mathsf{x}_4\{(\mathsf{b} \vee \mathsf{c})^*\mathsf{x}_2^*\}$, we check whether $(\mathsf{b} \vee \mathsf{c})^*(v_2)^* = (\mathsf{b} \vee \mathsf{c})^*\mathsf{a}^*$ can produce $v_4 = \mathsf{ca}$, which is the case.

Finally, we replace all definitions and references by the intended images to obtain $(\beta_1, \beta_2) = (\mathsf{ca}(\mathsf{b} \vee \mathsf{a}^*)\mathsf{caaca}, ((\mathsf{ca}) \vee \mathsf{a})^*\mathsf{caa})$.

In the general case, the situation can be slightly more complicated, since we also have to make sure that every ref-word necessarily instantiates a definition for x_i if $v_i \neq \emptyset$, while for $v_i = \emptyset$ ref-words without definition for x_i should still be possible. It can also happen that this procedure reduces an xregex to $\varnothing$, which means that $\mathcal{L}^{\bar{v}}(\bar{\alpha}) = \emptyset$.

These exemplary observations can be turned into a general procedure, which yields the following result.

Lemma 6.3 (★). *For every $\bar{\alpha} \in m\text{-}\mathrm{CXRE}_{\Sigma,\mathcal{X}_s}$ with $\mathcal{X}_s = \{\mathsf{x}_1, \mathsf{x}_2, \dots, \mathsf{x}_n\}$ and every $\bar{v} \in (\Sigma^*)^n$, there is a $\bar{\beta} \in m\text{-}\mathrm{CXRE}_{\Sigma,\emptyset}$ such that $\mathcal{L}(\bar{\beta}) = \mathcal{L}^{\bar{v}}(\bar{\alpha})$. Furthermore, $|\bar{\beta}| = \mathrm{O}(|\bar{\alpha}|k)$, where $k = \max\{|\bar{v}[i]| \mid i \in [n]\}$, and $\bar{\beta}$ can be computed in time polynomial in $|\bar{\alpha}|$ and $|\bar{v}|$.*

We note that applying Lemma 6.3 to a $q \in \mathrm{CXRPQ}^{\leq k}$ yields a CRPQ. Sketching a proof for Theorem 6.1 can be done as follows:

Let $q \in \mathrm{CXRPQ}^{\leq k}$ be Boolean with a conjunctive xregex $\bar{\alpha} = (\alpha_1, \alpha_2, \dots, \alpha_m)$ over Σ and $\mathcal{X}_s = \{\mathsf{x}_1, \mathsf{x}_2, \dots, \mathsf{x}_n\}$, and let $\mathcal{D}$ be a graph-database. Obviously, $\mathcal{D} \models q$ if and only if there is a $\bar{v} = (v_1, v_2, \dots, v_n) \in (\Sigma^{\leq k})^n$ such that $\mathcal{D} \models^{\bar{v}} q$. Thus, a nondeterministic algorithm that checks whether $\mathcal{D} \models q$ is as follows:

(1) Nondeterministically guess $\bar{v} = (v_1, v_2, \dots, v_n) \in (\Sigma^{\leq k})^n$.
(2) Compute $q' \in \mathrm{CRPQ}$ such that, for every database $\mathcal{D}$, we have that $q^{\bar{v}}(\mathcal{D}) = q'(\mathcal{D})$ (Lemma 6.3).
(3) Check whether $\mathcal{D} \models q'$ (Lemma 2.5).

We conclude this section by observing that, analogously to $q^k(\mathcal{D})$ for every $k \geq 1$, $q \in \mathrm{CXRPQ}$ and graph database $\mathcal{D}$, we can also define $q^{\log}(\mathcal{D}) = \bigcup_{k=0}^{\log(|\mathcal{D}|)} q^k(\mathcal{D})$. Just like we derived the classes $\mathrm{CXRPQ}^{\leq k}$, this gives rise to the class $\mathrm{CXRPQ}^{\log}$. Note that a Boolean $q \in \mathrm{CXRPQ}^{\log}$ matches a graph database $\mathcal{D}$, if and only if there is a matching morphism with image size bounded by $\log(|\mathcal{D}|)$. The next corollary follows from Theorem 6.1.

Corollary 6.4 (★). *$\mathrm{CXRPQ}^{\log}$-Bool-Eval can nondeterministically be solved in polynomial time in combined complexity and in space $\mathrm{O}(\log^2(|\mathcal{D}|))$ in data complexity.*

7 EXPRESSIVE POWER

In this section, we compare the expressive power of CXRPQs and their fragments with other established classes of graph queries. The *extended conjunctive regular path queries* (ECRPQs), already mentioned in the introduction, have been introduced in [8]. While CRPQ can only use regular expressions to impose unary constraints on the paths that are matched by the edges of the graph pattern, ECRPQs can use *regular relations* of arbitrary arity to describe constraints on tuples of paths, e. g., that certain paths are all equal, or have equal length or that one path is a prefix of the other etc. We are mainly interested in $\mathrm{ECRPQ}^{\mathrm{er}}$, i. e., the fragment obtained by only allowing unary relations or equality relations (i. e., relations

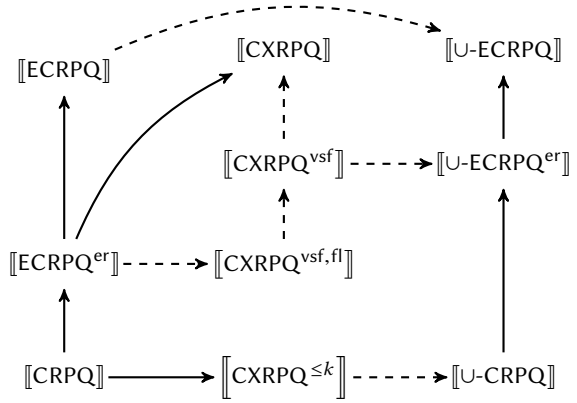

Figure 5: Illustration of the relations between the considered classes of conjunctive path queries. A dashed or solid arrow from A to B denotes $A \subseteq B$ or $A \subsetneq B$, respectively.

requiring certain paths to be equal). For a more detailed definition of ECRPQ, we refer to [8].

For a query class Q, a *union of Qs* (or $\cup$-Q, for short) is a query of the form $q = q_1 \vee q_2 \vee \ldots \vee q_k$, where, for every $i \in [k]$, $q_i \in Q$. For a graph-database $\mathcal{D}$, we define $q(\mathcal{D}) = \bigcup_{i\in[k]} q_i(\mathcal{D})$.

Theorem 7.1 (★). *The inclusion-diagram of Figure 5 is correct.*

We shall discuss these results in more detail and also provide some prove ideas for the non-trivial subset relations of Figure 5.

The diagram of Figure 5 shows three vertical layers of query classes of increasing expressive power. More precisely, on the left side we have the *classical* conjunctive path query classes CRPQ, ECRPQ$^{\text{er}}$ and ECRPQ. Then, our new fragments of CXRPQ follow. Finally, we have the classes of unions of the classical conjunctive path query classes. All these layers are naturally ordered by the subset relation, and these subset relations follow all directly by definition. The strictness of the inclusions

$$\llbracket \text{CRPQ} \rrbracket \subsetneq \llbracket \text{ECRPQ}^{\text{er}} \rrbracket \subsetneq \llbracket \text{ECRPQ} \rrbracket$$

(which are not mentioned in the literature) can be shown by interchange arguments, which, if slightly adapted, also show the strictness of the inclusions

$$\llbracket \cup\text{-CRPQ} \rrbracket \subsetneq \llbracket \cup\text{-ECRPQ}^{\text{er}} \rrbracket \subsetneq \llbracket \cup\text{-ECRPQ} \rrbracket .$$

The more interesting inclusion relations are the vertical ones. In this regard, we first note that $\llbracket \text{ECRPQ}^{\text{er}} \rrbracket \subseteq \llbracket \text{CXRPQ}^{\text{vsf,fl}} \rrbracket$ follows from a simple transformation: if in a $q \in \text{ECRPQ}^{\text{er}}$, we simply replace arcs $(x_1, \alpha_1, y_1), (x_2, \alpha_2, y_2), \ldots, (x_k, \alpha_k, y_k)$ of q's graph pattern that are all subject to the same equality relation by arcs $(x_1, \text{z}\{\beta\}, y_1), (x_2, \text{z}, y_2), \ldots, (x_k, \text{z}, y_k)$, where β is a regular expression for the language $\bigcap_{i=1}^{k} \mathcal{L}(\alpha_i)$, then we obtain an equivalent $q' \in \text{CXRPQ}^{\text{vsf,fl}}$. While the inclusion $\llbracket \text{ECRPQ}^{\text{er}} \rrbracket \subseteq \llbracket \text{CXRPQ}^{\text{vsf,fl}} \rrbracket$ is probably not very unexpected, it nevertheless points out that also quite restricted classes of CXRPQ still cover CRPQ that are extended by equality relations.

Less obvious are the inclusions $\llbracket \text{CXRPQ}^{\text{vsf}} \rrbracket \subseteq \llbracket \cup\text{-ECRPQ}^{\text{er}} \rrbracket$, and $\llbracket \text{CXRPQ}^{\leq k} \rrbracket \subseteq \llbracket \cup\text{-CRPQ} \rrbracket$, which are both more or less a result

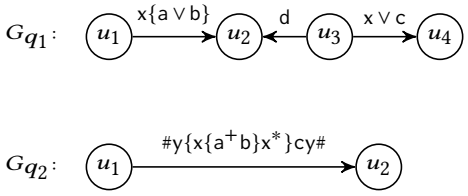

Figure 6: The graph patterns for q_1 and q_2 of Lemma 7.2.

from the upper bounds for these CXRPQ-fragments: $\llbracket \text{CXRPQ}^{\text{vsf}} \rrbracket \subseteq \llbracket \cup\text{-ECRPQ}^{\text{er}} \rrbracket$ is a consequence of the normal form of Section 5, while $\llbracket \text{CXRPQ}^{\leq k} \rrbracket \subseteq \llbracket \cup\text{-CRPQ} \rrbracket$, for every $k \geq 1$, follows from Lemma 6.3.

In a sense, these inclusion relations mean that the queries of our CXRPQ-fragments can be "decomposed" into unions of the more classical CRPQs and ECRPQ$^{\text{er}}$s. Thus, semantically, the CXRPQ$^{\leq k}$ and CXRPQ$^{\text{vsf}}$ can still be described in the formalisms of CRPQ and ECRPQ$^{\text{er}}$. However, there is a significant syntactical difference: our conversions of CXRPQ$^{\leq k}$ or CXRPQ$^{\text{vsf}}$ into $\cup$-CRPQ or $\cup$-ECRPQ$^{\text{er}}$, respectively, require exponential or even double exponential size blow-ups, respectively.

From an intuitive point of view, it is to be expected that the inclusion of ECRPQ$^{\text{er}}$ in CXRPQ$^{\text{vsf}}$ is strict. However, we can only show that CXRPQ are strictly more powerful than ECRPQ$^{\text{er}}$: the query $q_2 \in \text{CXRPQ}^{\text{vsf}}$ represented by the graph pattern G_{q_2} of Figure 6 cannot be represented by an ECRPQ$^{\text{er}}$ (see Lemma 7.2). On the other hand, it is much less obvious why CRPQ should be strictly less powerful than CXRPQ$^{\leq k}$, since the string variables of CXRPQ$^{\leq k}$s, which syntactically set them apart from CRPQs, can only range over finite sets of possible images (and therefore all possible combinations of images for these string variables should be expressible by a regular expression). It can therefore be considered surprising that, in fact, even CXRPQ$^{\leq 1}$ contains queries that are not expressible as CRPQ. More precisely, the query $q_1 \in \text{CXRPQ}^{\leq 1}$ represented by the graph pattern G_{q_1} of Figure 6 cannot be represented by an CRPQ (see Lemma 7.2).

Lemma 7.2 (★). $\llbracket q_1 \rrbracket \notin \llbracket \text{CRPQ} \rrbracket$ *and* $\llbracket q_2 \rrbracket \notin \llbracket \text{ECRPQ}^{\text{er}} \rrbracket$.

8 CONCLUSIONS AND OPEN PROBLEMS

The fact that evaluation for CXRPQ is at least PSpace-hard even in data complexity is reason enough to look at fragments of CXRPQ. The fragments presented in this work and their upper complexity bounds are summarised in Table 1; the classical classes CRPQ and ECRPQ$^{\text{er}}$ are also mentioned for comparison (note that in Table 1, ECRPQ$^{\text{er}}$ could also be replaced by ECRPQ).

An upper bound for evaluating unrestricted CXRPQs would surely be interesting from a theoretical point of view. All the upper bounds of Table 1 also have matching lower bounds, except for the combined complexity of CXRPQ$^{\text{vsf}}$-evaluation, for which we only know PSpace-hardness (see Theorem 5.2).

Several of our results implicitly pose conciseness questions. Each CXRPQ$^{\leq k}$ can be represented as the union of $\text{O}(|\Sigma| + 1)^{|\mathcal{X}_s|k})$ many CRPQs, and each CXRPQ$^{\text{vsf}}$ can be represented as the union of exponentially many ECRPQ$^{\text{er}}$s of exponential size. Are these

CRPQ	ECRPQ$^{\text{er}}$	CXRPQ$^{\text{vsf}}$	CXRPQ$^{\text{vsf,fl}}$	CXRPQ$^{\leq k}$
NP	PSpace	ExpSpace	PSpace	NP
NL	NL	NL	NL	NL

Table 1: A summary of the evaluation complexity upper bounds in combined complexity (upper row) and data complexity (lower row).

exponential blow-ups necessary? Theorem 6.1 gives some evidence that for CXRPQ$^{\leq k}$ this is the case.

All our algorithms for Bool-Eval can be extended to the problem Check. With respect to also extracting paths instead of only nodes from the graph-database, we can use the general techniques of [8] to some extent. More precisely, for a $q \in$ CXRPQ$^{\text{vsf}}$ (or $q \in$ CXRPQ$^{\leq k}$), a graph database $\mathcal{D}$ and a tuple $\bar{t} \in (V_{\mathcal{D}})^{\ell}$, our evaluation algorithms can be adapted to produce an automaton that represents all tuples of paths corresponding to the matching morphisms of q and $\mathcal{D}$, but they are rather large. A thorough analysis of CXRPQ-fragments with respect to how they can be used as queries that also extract paths is left for future work.

Interestingly enough, restrictions that lower the complexity for matching xregex to strings do not seem to help at all if we use them for querying graphs, and vice versa. In the string case, the NP-complete matching problem for xregex trivially becomes polynomial-time solvable, if the number of variables is bounded by a constant (see [41]), while this does not help for graphs (see Theorem 4.2). Variable-star freeness (Section 5) or bounding the image size (Section 6) has a positive impact with respect to graphs. However, the matching problem for xregex remains NP-hard, even if we require variable-star freeness *and* that variables can only range over words of length at most 1 (see [22, Theorem 3]).

ACKNOWLEDGMENTS

The author thanks Nicole Schweikardt for helpful discussions, and the anonymous referees for their careful evaluation and feedback. The author is supported by DFG-grant SCHM 3485/1-1.

REFERENCES

[1] Serge Abiteboul, Peter Buneman, and Dan Suciu. 1999. *Data on the Web: From Relations to Semistructured Data and XML.* Morgan Kaufmann.
[2] Serge Abiteboul, Dallan Quass, Jason McHugh, Jennifer Widom, and Janet L. Wiener. 1997. The Lorel Query Language for Semistructured Data. *Int. J. on Digital Libraries* 1, 1 (1997), 68–88.
[3] Alfred V. Aho. 1990. Algorithms for Finding Patterns in Strings. In *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity (A).* 255–300.
[4] Renzo Angles, Marcelo Arenas, Pablo Barceló, Aidan Hogan, Juan L. Reutter, and Domagoj Vrgoc. 2017. Foundations of Modern Query Languages for Graph Databases. *ACM Comput. Surv.* 50, 5 (2017), 68:1–68:40.
[5] Renzo Angles, Juan L. Reutter, and Hannes Voigt. 2019. Graph Query Languages. In *Encyclopedia of Big Data Technologies.*
[6] Pablo Barceló. 2013. Querying graph databases. In *Proceedings of the 32nd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2013, New York, NY, USA - June 22 - 27, 2013.* 175–188.
[7] Pablo Barceló, Diego Figueira, and Miguel Romero. 2019. Boundedness of Conjunctive Regular Path Queries. In *46th International Colloquium on Automata, Languages, and Programming, ICALP 2019, July 9-12, 2019, Patras, Greece.* 104:1–104:15.
[8] Pablo Barceló, Leonid Libkin, Anthony Widjaja Lin, and Peter T. Wood. 2012. Expressive Languages for Path Queries over Graph-Structured Data. *ACM Transactions on Database Systems (TODS)* 37, 4 (2012), 31:1–31:46.
[9] Pablo Barceló, Leonid Libkin, and Juan L. Reutter. 2014. Querying Regular Graph Patterns. *J. ACM* 61, 1 (2014), 8:1–8:54.
[10] Pablo Barceló, Miguel Romero, and Moshe Y. Vardi. 2016. Semantic Acyclicity on Graph Databases. *SIAM J. Comput.* 45, 4 (2016), 1339–1376.
[11] Angela Bonifati, Wim Martens, and Thomas Timm. 2017. An Analytical Study of Large SPARQL Query Logs. *PVLDB* 11, 2 (2017), 149–161.
[12] Angela Bonifati, Wim Martens, and Thomas Timm. 2019. Navigating the Maze of Wikidata Query Logs. In *The World Wide Web Conference, WWW 2019, San Francisco, CA, USA, May 13-17, 2019.* 127–138.
[13] Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Moshe Y. Vardi. 2000. Containment of Conjunctive Regular Path Queries with Inverse. In *KR 2000, Principles of Knowledge Representation and Reasoning Proceedings of the Seventh International Conference, Breckenridge, Colorado, USA, April 11-15, 2000.* 176–185.
[14] Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Moshe Y. Vardi. 2003. Reasoning on regular path queries. *SIGMOD Record* 32, 4 (2003), 83–92.
[15] Cezar Câmpeanu, Kai Salomaa, and Sheng Yu. [n.d.]. A Formal Study Of Practical Regular Expressions. *Int. J. Found. Comput. Sci.* 14, 6 ([n. d.]).
[16] Mariano P. Consens and Alberto O. Mendelzon. 1990. GraphLog: a Visual Formalism for Real Life Recursion. In *Proceedings of the Ninth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, April 2-4, 1990, Nashville, Tennessee, USA.* 404–416.
[17] Isabel F. Cruz, Alberto O. Mendelzon, and Peter T. Wood. 1987. A Graphical Query Language Supporting Recursion. In *Proceedings of the Association for Computing Machinery Special Interest Group on Management of Data 1987 Annual Conference, San Francisco, California, May 27-29, 1987.* 323–330.
[18] Wojciech Czerwinski, Wim Martens, Matthias Niewerth, and Pawel Parys. 2018. Minimization of Tree Patterns. *J. ACM* 65, 4 (2018), 26:1–26:46.
[19] Alin Deutsch and Val Tannen. 2001. Optimization Properties for Classes of Conjunctive Regular Path Queries. In *Database Programming Languages, 8th International Workshop, DBPL 2001, Frascati, Italy, September 8-10, 2001, Revised Papers.* 21–39.
[20] Johannes Doleschal, Benny Kimelfeld, Wim Martens, Yoav Nahshon, and Frank Neven. 2019. Split-Correctness in Information Extraction. In *Proceedings of the 38th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019.* 149–163.
[21] Henning Fernau, Florin Manea, Robert Mercas, and Markus L. Schmid. 2015. Pattern Matching with Variables: Fast Algorithms and New Hardness Results. In *32nd International Symposium on Theoretical Aspects of Computer Science, STACS 2015, March 4-7, 2015, Garching, Germany.* 302–315.
[22] Henning Fernau and Markus L. Schmid. 2015. Pattern matching with variables: A multivariate complexity analysis. *Information and Computation (I&C)* 242 (2015), 287–305.
[23] Henning Fernau, Markus L. Schmid, and Yngve Villanger. 2016. On the Parameterised Complexity of String Morphism Problems. *Theory of Computing Systems (ToCS)* 59, 1 (2016), 24–51.
[24] Daniela Florescu, Alon Y. Levy, and Dan Suciu. 1998. Query Containment for Conjunctive Queries with Regular Expressions. In *Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 1-3, 1998, Seattle, Washington, USA.* 139–148.
[25] Dominik D. Freydenberger. 2013. Extended Regular Expressions: Succinctness and Decidability. *Theory of Computing Systems (ToCS)* 53, 2 (2013), 159–193.
[26] Dominik D. Freydenberger. 2019. A Logic for Document Spanners. *Theory Comput. Syst.* 63, 7 (2019), 1679–1754.
[27] Dominik D. Freydenberger, Benny Kimelfeld, and Liat Peterfreund. 2018. Joining Extractions of Regular Expressions. In *Proceedings of the 37th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, Houston, TX, USA, June 10-15, 2018.* 137–149.
[28] Dominik D. Freydenberger and Markus L. Schmid. 2019. Deterministic regular expressions with back-references. *J. Comput. Syst. Sci.* 105 (2019), 1–39. https://doi.org/10.1016/j.jcss.2019.04.001
[29] Jeffrey E. F. Friedl. 2006. *Mastering regular expressions - understand your data and be more productive: for Perl, PHP, Java, .NET, Ruby, and more (3. ed.).* O'Reilly.
[30] Steve Harris and Andy Seaborne. 2013. *SPARQL 1.1 Query Language. W3C Recommendation.* Technical Report https://www.w3.org/TR/sparql11-query/. W3C.
[31] Jelle Hellings. 2014. Conjunctive Context-Free Path Queries. In *Proc. 17th International Conference on Database Theory (ICDT), Athens, Greece, March 24-28, 2014.* 119–130.
[32] The IEEE and The Open Group. 2016. IEEE Std 1003.1-2008, 2016 Edition, Chapter 9. http://pubs.opengroup.org/onlinepubs/9699919799/.
[33] Egor V. Kostylev, Juan L. Reutter, and Domagoj Vrgoc. 2018. Containment of queries for graphs with data. *J. Comput. Syst. Sci.* 92 (2018), 65–91.
[34] Leonid Libkin, Wim Martens, and Domagoj Vrgoc. 2016. Querying Graphs with Data. *Journal of the ACM* 63, 2 (2016), 14:1–14:53.
[35] Katja Losemann and Wim Martens. 2013. The complexity of regular expressions and property paths in SPARQL. *ACM Transactions on Database Systems (TODS)* 38, 4 (2013), 24:1–24:39.
[36] Wim Martens, Matthias Niewerth, and Tina Trautner. 2020. A Trichotomy for Regular Trail Queries. In *37th Symposium on Theoretical Aspects of Computer Science, STACS 2020, March 10-13, 2020, Montpellier, Germany.* Accepted for

publication.
[37] Wim Martens and Tina Trautner. 2018. Evaluation and Enumeration Problems for Regular Path Queries. In *21st International Conference on Database Theory, ICDT 2018, March 26-29, 2018, Vienna, Austria*. 19:1–19:21.
[38] Wim Martens and Tina Trautner. 2019. Bridging Theory and Practice with Query Log Analysis. *SIGMOD Record* 48, 1 (2019), 6–13.
[39] Alberto O. Mendelzon and Peter T. Wood. 1995. Finding Regular Simple Paths in Graph Databases. *SIAM Journal on Computing (SICOMP)* 24, 6 (1995), 1235–1258.
[40] Markus L. Schmid. 2013. Inside the Class of REGEX Languages. *International Journal of Foundations of Computer Science (IJFCS)* 24, 7 (2013), 1117–1134.
[41] Markus L. Schmid. 2016. Characterising REGEX languages by regular languages equipped with factor-referencing. *Information and Computation (I&C)* 249 (2016), 1–17.
[42] Markus L. Schmid. 2019. Conjunctive Regular Path Queries with String Variables. *CoRR* abs/1912.09326 (2019). arXiv:1912.09326 http://arxiv.org/abs/1912.09326
[43] The Neo4j Team. 2016. The Neo4j Cypher Manual v3.5. retrieved from https://neo4j.com/docs/developer-manual/current/cypher/.
[44] Apache TinkerPop^{TM}. 2013. Gremlin Language. retrieved from https://tinkerpop.apache.org/gremlin.html.
[45] Peter T. Wood. 2012. Query languages for graph databases. *SIGMOD Record* 41, 1 (2012), 50–60.