Regular Expressions with Binding over Data Words for Querying Graph Databases

Regular Expressions with Binding over Data Words for Querying Graph Databases

Leonid Libkin[1], Tony Tan[2], and Domagoj Vrgoč[3]

[1] University of Edinburgh, email: libkin@inf.ed.ac.uk
[2] Hasselt University and Transnational University of Limburg, email: tony.tan@uhasselt.be
[3] University of Edinburgh, email: domagoj.vrgoc@ed.ac.uk

Abstract. Data words assign to each position a letter from a finite alphabet and a data value from an infinite set. Introduced as an abstraction of paths in XML documents, they recently found applications in querying graph databases as well. Those are actively studied due to applications in such diverse areas as social networks, semantic web, and biological databases. Querying formalisms for graph databases are based on specifying paths conforming to some regular conditions, which led to a study of regular expressions for data words.
Previously studied regular expressions for data words were either rather limited, or had the full expressiveness of register automata, at the expense of a quite unnatural and unintuitive binding mechanism for data values. Our goal is to introduce a natural extension of regular expressions with proper bindings for data values, similar to the notion of freeze quantifiers used in connection with temporal logics over data words, and to study both language-theoretic properties of the resulting class of languages of data words, and their applications in querying graph databases.

1 Introduction

Data words, unlike the usual words over finite alphabet, assign to each position both a letter from a finite alphabet and an element of an infinite set, referred to as a data value. An example of a data word is $\binom{a}{1}\binom{b}{2}\binom{a}{3}\binom{b}{1}$. This is a data word over the finite alphabet $\{a, b\}$, with data elements coming from an infinite domain, in this case, $\mathbb{N}$. Investigations of data words picked up recently due to their importance in the study of XML documents. Those are naturally modeled as ordered unranked trees in which every node has both a label and a datum (these are referred to as data trees). Data words then model paths in data trees, and as such are essential for investigations of many path-based formalisms for XML, for instance, its navigational query language XPath. We refer the reader to [30, 7, 14, 29] for recent surveys.

While the XML data format dominated the data management landscape for a while, primarily in the 2000s, over the past few years the focus started shifting towards the *graph* data model. Graph-structured data appears naturally in a variety of applications, most notably social networks and the Semantic Web (as

it underlies the RDF format). Its other applications include biology, network traffic, crime detection, and modeling object-oriented data [13, 20, 23, 25–28]. Such databases are represented as graphs in which nodes are objects and the edge labels specify relationships between them; see [1, 4] for surveys.

Just as in the case of XML, a crucial building block in queries against graph data deals with properties of paths in them. The most basic formalism is that of *regular path queries*, or *RPQs*, which select nodes connected by a path described by a regular language over the labeling alphabet [11]. There are multiple extensions with more complex patterns, backward navigation, regular relations over paths, and non-regular features [3, 5, 6, 8–10]. In real applications we deal with both navigational information and data, so it is essential that we look at properties of paths that also describe how data values change along them. Since such paths (as we shall explain later) are just data words, it becomes necessary to provide expressive and well-behaved mechanisms for describing languages of data words.

One of the most commonly used formalisms for describing the notion of regularity for data words is that of *register automata* [18]. These extend the standard NFAs with registers that can store data values; transitions can compare the currently read data value with values stored in registers.

However, register automata are not convenient for specifying properties – ideally, we want to use regular expressions to define languages. These have been looked at in the context of data words (or words over infinite alphabets), and are based on the idea of using *variables* for binding data values. An initial attempt to define such expressions was made in [19], but it was very limited. Another formalism, called *regular expressions with memory*, was shown to be equivalent to register automata [21, 22]. At the first glance, they appear to be a good formalism: these are expressions like $a \downarrow_x (a[x^=])^*$ saying: read letter a, bind data value to x, and read the rest of the data word checking that all letters are a and the data values are the same as x. This will define data words $\binom{a}{d} \cdots \binom{a}{d}$ for some data value d. This is reminiscent of freeze quantifiers used in connection with the study of data word languages [12].

The serious problem with these expressions, however, is the *binding* of variables. The expression above is fine, but now consider the following expression: $a \downarrow_x (a[x^=]a \downarrow_x)^* a[x^=]$. This expression re-binds variable x inside the scope of another binding, and then crucially, when this happens, the original binding of x is *lost*! Such expressions really mimic the behavior of register automata, which makes them more procedural than declarative. (The above expression defines data words of the form $\binom{a}{d_1}\binom{a}{d_1} \cdots \binom{a}{d_n}\binom{a}{d_n}$.)

Losing the original binding of a variable when reusing it inside its scope goes completely against the usual practice of writing logical expressions, programs, etc., that have bound variables. Nevertheless, this feature was essential for capturing register automata [21]. So natural questions arise:

– Can we define regular expressions for data words that use the acceptable scope/binding policies for variables? Such expressions will be more declarative than procedural, and more appropriate for being used in queries.

– Do these fall short of the full power of register automata?
– What are their basic properties, and what is the complexity of querying graph data with such expressions?

Contributions Our main contribution is to define a new formalism of *regular expressions with binding*, or REWBs, to study its properties, and to show how it can be used in the context of graph querying. The binding mechanism of REWBs follows the standard scoping rules, and is essentially the same as in LTL extensions with freeze quantifiers [12]. We also look at some subclasses of REWBs based on the types of conditions one can use: in *simple* REWBs, each condition involves at most one variable (all those shown above were such), and in *positive* REWBs, negation and inequality cannot be used in conditions.

We show that the class of languages defined by REWBs is strictly contained in the class of languages defined by register automata. The separating example is rather intricate, and indeed it appears that for most reasonable languages one can think of, if they are definable by register automata, they would be definable by REWBs as well. At the same time, REWBs lower the complexity of some key computational tasks related to languages of data words. For instance, nonemptiness is PSPACE-complete for register automata [12], but we show that it is NP-complete for REWBs (and trivializes for simple and positive REWBs).

We consider the containment and universality problems for REWBs. In general they are undecidable, even for simple REWBs. However, the problem becomes decidable for positive REWBs.

We look at applications of REWBs in querying graph databases. The problem of query evaluation is essentially checking whether the intersection of two languages of data words is nonempty. We use this to show that the complexity of query evaluation is PSPACE-complete (note that it is higher than the complexity of nonemptiness alone); for a fixed REWB, the complexity is tractable.

At the end we also sketch some results concerning a model of data word automaton that uses variables introduced in [15]. We also comment on how these can be combined with register automata to obtain a language subsuming all the previously used ones while still retaining good query evaluation bounds.

Organization We define data words and data graphs in Section 2. In Section 3 we introduce our notion of regular expression with binding (REWB) and study their nonemptiness and universality problems in Section 4 and Section 5, respectively. In Section 6 we study REWBs as a graph database query language and in Section 7 we consider some possible extensions that could be useful in graph querying. Due to space limitations, complete proofs of all the results are in the appendix.

2 Data words and data graphs

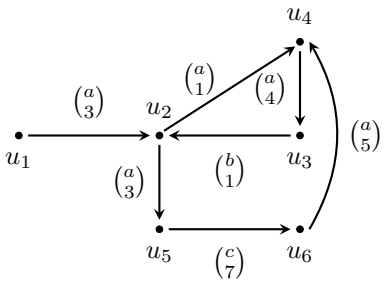
Let Σ be a finite alphabet and $\mathcal{D}$ a countable infinite set of data values. A **data word** is simply a finite string over the alphabet $\Sigma \times \mathcal{D}$. That is, in each position a data word carries a letter from Σ and a data value from $\mathcal{D}$. We will denote data words by $\binom{a_1}{d_1} \dots \binom{a_n}{d_n}$, where $a_i \in \Sigma$ and $d_i \in \mathcal{D}$.

A **data graph** (over Σ) is pair $G = (V, E)$, where

- V is a finite set of nodes;
- $E \subseteq V \times \Sigma \times \mathcal{D} \times V$ is a set of edges where each edge contains a label from Σ and a data value from $\mathcal{D}$.

We write $V(G)$ and $E(G)$ to denote the set of nodes and edges of G, respectively. An edge e from a node u to a node u' is written in the form $(u, \binom{a}{d}, u')$, where $a \in \Sigma$ and $d \in \mathcal{D}$. We call a the label of the edge e and d the data value of the edge e. We write $\mathcal{D}(G)$ to denote the set of data values in G.

The following is an example of a data graph, with nodes $u_1, \ldots, u_6$ and edges $(u_1, \binom{a}{3}, u_2)$, $(u_3, \binom{b}{1}, u_2)$, $(u_2, \binom{a}{3}, u_5)$, $(u_6, \binom{a}{5}, u_4)$, $(u_2, \binom{a}{1}, u_4)$, $(u_4, \binom{a}{4}, u_3)$ and $(u_5, \binom{c}{7}, u_6)$.



A path from a node v to a node v' in G is a sequence

$$\pi = v_1 \binom{a_1}{d_1} v_2 \binom{a_2}{d_2} v_3 \binom{a_3}{d_3} \cdots v_n \binom{a_n}{d_n} v_{n+1}$$

such that each $(v_i, \binom{a_i}{d_i}, v_{i+1})$ is an edge for each $i \leq n$, and $v_1 = v$ and $v_{n+1} = v'$.

A path π defines a data word $w(\pi) = \binom{a_1}{d_1}\binom{a_2}{d_2}\binom{a_3}{d_3}\cdots\binom{a_n}{d_n}$.

Remark Note that we have chosen a model in which labels and data values appear in edges. Of course other variations are possible, for instance labels appearing in edges and data values in nodes. All of these easily simulate each other, very much in the same way as one can use either labeled transitions systems or Kripke structures as models of temporal or modal logic formulae. In fact both models – with labels in edges and labels in nodes – have been considered in the context of semistructured data and, at least from the point of view of their expressiveness, they are viewed as equivalent. Our choice is dictated by the ease of notation primarily, as it identifies paths with data words.

3 Regular expressions with binding

We now define regular expressions with binding for data words. As explained already, expressions with variables for data words were previously defined in [22]

but those were really designed to mimic the transitions of register automata, and had very procedural, rather than declarative flavor. Here we define them using proper scoping rules.

Variables will store data values; those will be compared with other variables using conditions. To define them, assume that, for each $k > 0$, we have variables $x_1, \ldots, x_k$. Then the set of conditions $\mathcal{C}_k$ is given by the grammar:

$$c \;:=\; \top \mid \bot \mid x_i^{=} \mid x_i^{\neq} \mid c \wedge c \mid c \vee c \mid \neg c, \qquad 1 \leq i \leq k.$$

The satisfaction of a condition is defined with respect to a data value $d \in \mathcal{D}$ and a (partial) valuation $\nu : \{x_1, \ldots, x_k\} \to \mathcal{D}$ of variables as follows:

- $d, \nu \models \top$ and $d, \nu \not\models \bot$;
- $d, \nu \models x_i^{=}$ iff $d = \nu(x_i)$;
- $d, \nu \models x_i^{\neq}$ iff $d \neq \nu(x_i)$;
- the semantics for Boolean connectives $\vee, \wedge$, and $\neg$ is standard.

Next we define regular expressions with binding.

Definition 1. *Let Σ be a finite alphabet and $\{x_1, \ldots, x_k\}$ a finite set of variables.* Regular expressions with binding *(REWB) over $\Sigma[x_1, \ldots, x_k]$ are defined inductively as follows:*

$$r \;:=\; \varepsilon \;\mid\; a \;\mid\; a[c] \;\mid\; r + r \;\mid\; r \cdot r \;\mid\; r^* \;\mid\; a \downarrow_{x_i} (r) \qquad (1)$$

where $a \in \Sigma$ and c is a condition in $\mathcal{C}_k$.

A variable x_i is bound if it it occurs in the scope of some $\downarrow_{x_i}$ operator and free otherwise. More precisely, free variables of an expression are defined inductively: ε and a have no free variables, in $a[c]$ all variables occurring in c are free, in $r_1 + r_2$ and $r_1 \cdot r_2$ the free variables are those of r_1 and r_2, the free variables of r^* are those of r, and the free variables of $a \downarrow_{x_i} (r)$ are those of r except x_i. We will write $r(x_1, \ldots, x_l)$ if $x_1, \ldots, x_l$ are the free variables in r.

A valuation on the variables $x_1, \ldots, x_k$ is a partial function $\nu : \{x_1, \ldots, x_k\} \mapsto \mathcal{D}$. We denote by $\mathcal{F}(x_1, \ldots, x_k)$ the set of all valuations on $x_1, \ldots, x_k$. For a valuation ν, we write $\nu[x_i \leftarrow d]$ to denote the valuation ν' obtained by fixing $\nu'(x_i) = d$ and $\nu'(x) = \nu(x)$ for all other $x \neq x_i$. Likewise, we write $\nu[\bar{x} \leftarrow \bar{d}]$ for a simultaneous substitution of values from $\bar{d} = (d_1, \ldots, d_l)$ for variables $\bar{x} = (x_1, \ldots, x_l)$. Also notation $\nu(\bar{x}) = \bar{d}$ means that $\nu(x_i) = d_i$ for all $i \leq l$.

Semantics Let $r(\bar{x})$ be an REWB over $\Sigma[x_1, \ldots, x_k]$. A valuation $\nu \in \mathcal{F}(x_1, \ldots, x_k)$ is compatible with r, if $\nu(\bar{x})$ is defined.

A regular expression $r(\bar{x})$ over $\Sigma[x_1, \ldots, x_k]$ and a valuation $\nu \in \mathcal{F}(x_1, \ldots, x_k)$ compatible with r define a language $L(r, \nu)$ of data words as follows.

- If $r = a$ and $a \in \Sigma$, then $L(r, \nu) = \{\binom{a}{d} \mid d \in \mathbb{N}\}$.
- If $r = a[c]$, then $L(r, \nu) = \{\binom{a}{d} \mid d, \nu \models c\}$.
- If $r = r_1 + r_2$, then $L(r, \nu) = L(r_1, \nu) \cup L(r_2, \nu)$.

– If $r = r_1 \cdot r_2$, then $L(r, \nu) = L(r_1, \nu) \cdot L(r_2, \nu)$.
– If $r = r_1^*$, then $L(r, \nu) = L(r_1, \nu)^*$.
– If $r = a \downarrow_{x_i} (r_1)$, then $L(r, \nu) = \bigcup_{d \in \mathcal{D}} \left\{ \binom{a}{d} \right\} \cdot L(r_1, \nu[x_i \leftarrow d])$.

A REWB r defines a language of data words as follows.

$$L(r) = \bigcup_{\nu \text{ compatible with } r} L(r, \nu).$$

In particular, if r is without free variables, then $L(r) = L(r, \emptyset)$. We will call such REWBs *closed*.

Register automata and expressions with memory As mentioned earlier, *register automata* extend NFAs with the ability to store and compare data values. Formally, an automaton with k registers is $\mathcal{A} = (Q, q_0, F, T)$, where:

– Q is a finite set of states;
– $q_0 \in Q$ is the initial state;
– $F \subseteq Q$ is the set of final states;
– T is a finite set of transitions of the form $(q, a, c) \to (I, q')$, where q, q' are states, a is a label, $I \subseteq \{1, \ldots, k\}$, and c is a condition in $\mathcal{C}_k$.

Intuitively the automaton traverses a data word from left to right, starting in q_0, with all registers empty. If it reads $\binom{a}{d}$ in state q with register configuration $\tau : \{1, \ldots, k\} \to \mathcal{D}$, it may apply a transition $(q, a, c) \to (I, q')$ if $d, \tau \models c$; it then enters state q' and changes contents of registers i, with $i \in I$, to d. For more details on register automata we refer reader to [18, 22].

Expressions introduced in [21] had a similar syntax but rather different semantics. They were built using $a \downarrow_x$, concatenation, union and Kleene star. That is, no binding was introduced with $a \downarrow_x$; rather it directly matched the operation of putting a value in a register. In contrast, we use proper bindings of variables; expression $a \downarrow_x$ appears only in the context $a \downarrow_x (r)$ where it binds x inside the expression r only. This corresponds to the standard binding policies in logic, or in programs.

Example 1. We list several examples of languages expressible with our expressions. In all cases below we have a singleton alphabet $\Sigma = \{a\}$.

– The language that consists of data words where the data value in the first position is different from the others is given by: $a \downarrow_x ((a[x^{\neq}])^*)$.
– The language that consists of data words where the data values in the first and the last position are the same is given by: $a \downarrow_x (a^* \cdot a[x^{=}])$.
– The language that consists of data words where there are two positions with the same data value: $a^* \cdot a \downarrow_x (a^* \cdot a[x^{=}]) \cdot a^*$.

Note that in REWBs in the above example the conditions are very simple: they are either $x^{=}$ or $x^{\neq}$. We will call such expressions *simple* REWBs.

We shall also consider *positive* REWBs where negation and inequality are disallowed in conditions. That is, all the conditions c are constructed using the following syntax: $c := \top \mid x_i^{=} \mid c \wedge c \mid c \vee c$,, where $1 \leq i \leq k$.

We finish this section by showing that REWBs are strictly weaker than register automata (i.e., proper binding of variables has a cost – albeit small – in terms of expressiveness).

Theorem 1. *The class of languages defined by REWBs is strictly contained in the class of languages accepted by register automata.*

That the class of languages defined by REWBs is contained in the class of languages defined by register automata can be proved by using a similar inductive construction as in [21, Proposition 5.3]. The idea behind the construction of the separating example follows the intuition that defining scope of variables restricts the power of the language, compared to register automata where once stored, the value remains in the register until rewritten. As the proof is rather technical and lengthy, we present it in the appendix.

We note that the separating example is rather intricate, and certainly not a natural language one would think of. In fact, all natural languages definable with register automata that we used here as examples – and many more, especially those suitable for graph querying – are definable by REWBs.

4 The nonemptiness problem

We now look at the standard language-theoretic problem of nonemptiness:

	NONEMPTINESS FOR REWBS
Input:	A REWB r over $\Sigma[x_1, \ldots, x_k]$.
Task:	Decide whether $L(r) \neq \emptyset$.

More generally, one can ask if $L(r, \nu) \neq \emptyset$ for a REWB r and a compatible valuation ν.

Recall that for register automata, the nonemptiness problem is PSPACE-complete [12] (and the same bound applied to regular expressions with memory [22]). Introducing proper binding, we lose little expressiveness and yet can lower the complexity.

Theorem 2. *The nonemptiness problem for REWBs is* NP*-complete.*

The proof is in the appendix. Note that for simple and positive REWBs the problem trivializes.

Proposition 1.
- *For every simple REWB r over $\Sigma[x_1, \ldots, x_k]$, and for every valuation ν compatible with r, we have $L(r, \nu) \neq \emptyset$.*
- *For every positive REWB r over $\Sigma[x_1, \ldots, x_k]$, there is a valuation ν such that $L(r, \nu) \neq \emptyset$.*

5 Containment and universality

We now turn our attention to language containment. That is we are dealing with the following problem:

Containment for REWBs	
Input:	Two REWBs r_1, r_2 over $\Sigma[x_1, \ldots, x_k]$.
Task:	Decide whether $L(r_1) \subseteq L(r_2)$.

When r_2 is a fixed expression denoting all data words, this is the universality problem. We show that both are undecidable.

In fact, we show a stronger statement, that *universality* of simple REWBs that use just a single variable is already undecidable.

Universality for one-variable REWBs	
Input:	An REWB r over $\Sigma[x]$.
Task:	Decide whether $L(r) = (\Sigma \times \mathcal{D})^*$.

Theorem 3. Universality for one-variable REWBs *is undecidable. In particular, containment for REWBs is undecidable too.*

While restriction to simple REWBs does not make the problem decidable, the restriction to positive REWBs does: as is often the case, static analysis tasks become easier without negation.

Theorem 4. *The containment problem for positive REWBs is decidable.*

Proof. It is rather straightforward to show that any positive REWB can be converted into a register automaton without inequality [19]. The decidability of the language containment follows from the fact that the containment problem for register automata without inequality is decidable [31].

6 REWBs as a query language for data graphs

Standard mechanisms for querying graph databases are based on *regular path queries*, or RPQs: those select nodes connected by a path belonging to a given regular language [4, 11, 9, 10]. For data graphs, we follow the same idea, but now paths are specified by REWBs, since they contain data. In this section we study the complexity of this querying formalism.

We first explain how the problem of query evaluation can be cast as a problem of checking nonemptiness of language intersection.

Note that a data graph G can be viewed as an automaton, generating data words. That is, given a data graph $G = (V, E)$, and a pair of nodes s, t, we let $\mathcal{L}(G, s, t)$ be $\{w(\pi) \mid \pi \text{ is a path from } s \text{ to } t \text{ in } G\}$; this is a set of data words.

Let $r(\bar{x})$ be a REWB over $\Sigma[x_1, \ldots, x_k]$. For ν compatible with r, we let $\mathcal{L}(G, s, t, r, \nu)$ be $\mathcal{L}(G, s, t) \cap \mathcal{L}(r, \nu)$. Then for a graph $G = (V, E)$, we define the

answer to r over G as the set $\mathcal{Q}(r, G)$ of triples $(s, t, \bar{d}) \in V \times V \times \mathcal{D}^k$, such that $\mathcal{L}(G, s, t, r, \nu[\bar{x} \leftarrow \bar{d}])) \neq \emptyset$. In other words, there is a path π in G from s to t such that $w(\pi) \in L(r, \nu)$, where $\nu(\bar{x}) = \bar{d}$.

If r is a closed REWB, we do not need a valuation in the above definition. That is, $\mathcal{Q}(r, G)$ is the set of pairs of nodes (s, t) such that $\mathcal{L}(G, s, t) \cap \mathcal{L}(r) \neq \emptyset$, i.e., there is a path π in G from s to t such that $w(\pi) \in L(r)$.

In what follows we are interested in the query evaluation and query containment problems. For simplicity we will work with closed REWBs only. We start with query evaluation.

	Query Evaluation for REWB
Input:	A data graph G, two nodes $s, t \in V(G)$ and a REWB r.
Task:	Decide whether $(s, t) \in \mathcal{Q}(r, G)$.

Note that in this problem, both the data graph and the query, given by r, are inputs; this is referred to as the *combined complexity* of query evaluation. If the expression r is fixed, we are talking about *data complexity*.

Recall that for the usual graphs (without data), the combined complexity of evaluating RPQs is polynomial, but if conjunctions of RPQs are taken, it goes up to NP (and could be NP-complete, in fact [11, 10]). When we look at data graphs and specify paths with register automata, combined complexity jumps to Pspace-complete [21].

However, we have seen that REWBs are less expressive than register automata, so perhaps a lower NP bound would apply to them? One way to try to do it is to find a polynomial bound on the length of a minimal path witnessing a REWB in a data graph. The next proposition shows that this is impossible, since in some cases the shortest witnessing path will be exponentially long, even if the REWB uses only one variable.

Proposition 2. *Let $\Sigma = \{\$, ¢, a, b\}$ be a finite alphabet. There exists a family of data graphs $\{G_n(s, t)\}_{n>1}$ with two distinguished nodes s and t, and a family of closed REWBs $\{r_n\}_{n>1}$ such that*

- *each $G_n(s, t)$ is of size $O(n)$;*
- *each r_n is a closed REWB over $\Sigma[x]$ of length $O(n)$; and*
- *every data word in $\mathcal{L}(G_n, s, t, r_n)$ is of length $\Omega(2^{\lfloor n/2 \rfloor})$.*

The proof of this is rather involved and can be found in the appendix.

Next we describe the complexity of the query evaluation problem. It turns out that it matches that for register automata.

Theorem 5.
- *The complexity of query evaluation for REWB is* Pspace*-complete.*
- *For each fixed r, the complexity of query evaluation for REWB is in* NLogspace*.*

In other words, the combined complexity of queries based on REWBs is Pspace-complete, and their data complexity is in NLogspace (and of course it

can be NLOGSPACE-complete even for very simple expressions, e.g., Σ^*, which just expresses reachability). Note that the combined complexity is acceptable (it matches, for example, the combined complexity of standard relational query languages such as relational calculus and algebra), and that data complexity is the best possible for a language that can express the reachability problem.

We prove PSPACE membership by showing how to transform REWBs into regular expressions when only finitely many data values are considered. Since the expression in question is of length exponential in the size of the input, standard on-the-fly construction of product with the input graph (viewed as an NFA) gives us the desired bound. Details of this construction, as well as the proof of hardness, can be found in the appendix. The same proof, for a fixed r, gives us the bound for data complexity.

Note that the upper bound follows from the connection with register automata. In order to make our presentation self contained we opted to present a different proof in the appendix.

By examining the proofs of Theorem 5 and Theorem 3 we observe that lower bounds already hold for both simple and positive REWBs. That is we get the following.

Corollary 1. *The following holds for simple REWBs.*

- *Combined complexity of simple (or positive) REWB queries is* PSPACE*-complete.*
- *Data complexity of simple (or positive) REWB queries is* NLOGSPACE*-complete.*

Another important problem in querying graphs is query containment. In general, the query containment problem asks, for two REWBs r_1, r_2 over $\Sigma[x_1, \ldots, x_k]$, whether $\mathcal{Q}(r_1, G) \subseteq \mathcal{Q}(r_2, G)$ for every data graph G. For REWB-based queries we look at, this problem is easily seen to be equivalent to language containment. Using this fact and the results of Section 5 we obtain the following.

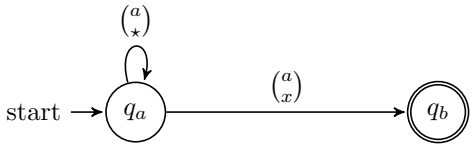
Corollary 2. *Query containment is undecidable for REWBs and simple REWBs. It becomes decidable if we restrict our queries to positive REWBs.*

7 Conclusions and Extensions

After conducting an extensive study of their language-theoretic properties and their ability to query graph data we conclude that REWBs can serve as a highly expressive language that still retains good query evaluation properties. Although weaker than register automata and their expression counterpart – regular expressions with memory, REWBs come with a more natural and declarative syntax and have a lower complexity of some language-theoretic properties such as nonemptiness. They also complete a picture of expressions that relate to register automata – a question that often came up in the discussions about the connection of regular expressions with memory (REMs) and register automata [21, 22], as they can be seen as a natural restriction of REMs with proper scoping rules.

As we have seen, both in this paper and in previous work on graph querying, all of the considered formalisms have a combined complexity of query evaluation that is either a low degree polynomial, or PSPACE-complete. A natural question to ask is if there is a formalism whose combined complexity lies between these two classes.

An answer to this can be given using a model of automata that extends NFAs in a similar way that REWBs extend regular expressions – by allowing usage of variables. These automata, called *variable automata*, were introduced in [15] and although originally defined for words over an infinite alphabet, they can easily be modified to handle data words. Intuitively, they can be viewed as NFAs with a guess of data values to be assigned to variables, with the run of the automaton verifying correctness of the guess. An example of a variable automaton recognizing the language of all words where the last data value is different from all others is given in the following image.



Here we observe that variable automata use two sorts of variables – an ordinary bound variable x that is assigned a unique value, and a special free variable $\star$, whose every occurrence is assigned a value different from the ones assigned to the bound variables.

It can be show that variable automata, used as a graph querying formalism, have NP-complete combined complexity of query evaluation and that their deterministic subclass [15] has coNP query containment. Due to space limitations we defer the technical details of these results to the appendix.

The somewhat synthetic nature of variable automata and their usage of the free variable makes them incomparable with REWBs and register automata, as the example above demonstrates. A natural question then is whether there is a model that encompasses both and still retains the same good query evaluation bounds. It can be shown that by allowing variable automata to use the full power of registers we get a model that subsumes all of the previously studied models and whose combined complexity is no worse that the one of register automata. As the details of the construction are rather lengthy we defer them to the appendix.

References

1. S. Abiteboul, P. Buneman, D. Suciu. *Data on the Web: From Relations to Semistructured Data and XML.* Morgan Kauffman, 1999.
2. S. Abiteboul, R. Hull, V. Vianu. *Foundations of Databases.* Addison-Wesley, 1995.
3. S. Abiteboul, V. Vianu. Regular path queries with constraints. *JCSS* 58 (1999), 428–452.
4. R. Angles, C. Gutiérrez. Survey of graph database models. *ACM Comput. Surv.* 40(1): (2008).

5. P. Barceló, D. Figueira, L. Libkin. Graph logics with rational relations and the generalized intersection problem. In *LICS 2012*.
6. P. Barceló, L. Libkin, A. W. Lin, P. Wood. Expressive languages for path queries over graph-structured data. *ACM TODS*, 37(4) (2012).
7. M. Bojanczyk. Automata for Data Words and Data Trees. In *RTA* 2010, pages 1–4.
8. D. Calvanese, G. de Giacomo, M. Lenzerini, M. Y. Vardi. Containment of conjunctive regular path queries with inverse. In *KR'00*, pages 176–185.
9. D. Calvanese, G. de Giacomo, M. Lenzerini, M. Y. Vardi. Rewriting of regular expressions and regular path queries. *JCSS*, 64(3):443–465 (2002).
10. M. P. Consens, A. O. Mendelzon. GraphLog: a visual formalism for real life recursion. In *PODS'90*, pages 404–416.
11. I. Cruz, A. Mendelzon, P. Wood. A graphical query language supporting recursion. In *SIGMOD'87*, pages 323–330.
12. S. Demri, R. Lazić. LTL with the freeze quantifier and register automata. *ACM TOCL* 10(3): (2009).
13. W. Fan. Graph pattern matching revised for social network analysis. In *ICDT 2012*, pages 8–21.
14. D. Figueira. Reasoning on words and trees with data. PhD thesis, 2010.
15. O. Grumberg, O. Kupferman, S. Sheinvald. Variable automata over infinite alphabets. In *LATA'10*, pages 561–572.
16. O. Grumberg, O. Kupferman, S. Sheinvald. Variable automata over infinite alphabets. Manuscript, 2011.
17. C. Gutierrez, C. Hurtado, A. Mendelzon. Foundations of semantic Web databases. *J. Comput. Syst. Sci.* 77(3): 520–541 (2011).
18. M. Kaminski, N. Francez. Finite-memory automata. *TCS* 134(2): 329–363 (1994).
19. M. Kaminski and T. Tan. Regular expressions for languages over infinite alphabets. *Fundamenta Informaticae*, 69(3):301–318 (2006).
20. U. Leser. A query language for biological networks. *Bioinformatics* 21 (suppl 2) (2005), ii33–ii39.
21. L. Libkin, D. Vrgoč. Regular path queries on graphs with data. In *ICDT'12*, pages 74–85.
22. L. Libkin, D. Vrgoč. Regular expressions for data words. *LPAR'12*, pages 274–288.
23. R. Milo, S. Shen-Orr, et al. Network motifs: simple building blocks of complex networks. *Science* 298(5594) (2002), 824–827.
24. F. Neven, T. Schwentick, V. Vianu. Finite state machines for strings over infinite alphabets. *ACM TOCL* 5(3): 403–435 (2004).
25. F. Olken. Graph data management for molecular biology. *OMICS* 7: 75–78 (2003).
26. J. Pérez, M. Arenas, C. Gutierrez. Semantics and complexity of SPARQL. *ACM TODS* 34(3): 1–45 (2009).
27. R. Ronen and O. Shmueli. SoQL: a language for querying and creating data in social networks. In *ICDE 2009*, pages 1595–1602.
28. M. San Martín, C. Gutierrez. Representing, querying and transforming social networks with RDF/SPARQL. In *ESWC 2009*, pages 293–307.
29. T. Schwentick. A Little Bit Infinite? On Adding Data to Finitely Labelled Structures. In *STACS* 2008, pages 17–18.
30. L. Segoufin. Automata and logics for words and trees over an infinite alphabet. In *CSL'06*, pages 41-57.
31. A. Tal. *Decidability of Inclusion for Unification Based Automata.* M.Sc. thesis (in Hebrew), Technion, 1999.

APPENDIX

Proofs

Proof of Theorem 1

To prove the Theorem we define the language $\mathcal{P}$ that will separate register automata from REWBs.

For a positive integer $m \geq 1$, we define a language $\mathcal{P}_m$ over the unary alphabet $\Sigma = \{a\}$ which consists of data words of the form:

$$\binom{a}{d_0}\binom{a}{d_1}\binom{a}{e_0}\binom{a}{e_1}\underbrace{\cdots}_{v_1}\binom{a}{d_1}\binom{a}{d_2}\underbrace{\cdots}_{w_1}\binom{a}{e_1}\binom{a}{e_2}\underbrace{\cdots}_{v_2}\binom{a}{d_2}\binom{a}{d_3}\underbrace{\cdots}_{w_2}\binom{a}{e_2}\binom{a}{e_3}\cdots\cdots$$

$$\cdots\cdots\binom{a}{e_{m-2}}\binom{a}{e_{m-1}}\underbrace{\cdots}_{v_{m-1}}\binom{a}{d_{m-1}}\binom{a}{d_m}\underbrace{\cdots}_{w_{m-1}}\binom{a}{e_{m-1}}\binom{a}{e_m}$$

where $m \geq 1$ and for each $i = 1, \ldots, m$, the data value d_i does not appear in v_i and the data value e_i does not appear in w_i.

We then define the language $\mathcal{P}$ as

$$\mathcal{P} := \bigcup_{m \geq 1} \mathcal{P}_m$$

Now Theorem 1 follows immediately from Lemmas 1 and 2 below.

Lemma 1. *The language $\mathcal{P}$ is accepted by a two-register automaton.*

Proof. It is rather straightforward to show that the language $\mathcal{P}$ is accepted by two-register automaton. One register is to take care of the d_i's and the other the e_i's.

Lemma 2. *The language $\mathcal{P}$ is not definable by REWBs.*

Next we prove Lemma 2. Note that for simplicity we prove the Lemma for the case of simple REWBs. It is straightforward to see that the same proof works in the case of REWBs that use multiple comparisons in one condition.

The proof is rather technical and will require a few auxiliary notions.

Let r be an REWB over $\Sigma[x_1, \ldots, x_k]$. A *derivation tree* t with respect to r is a tree whose internal nodes are labeled with (r', ν) where r' is an subexpression of r and $\nu \in \mathcal{F}[x_1, \ldots, x_k]$ constructed as follows. The root node is labeled with $(e, \emptyset)$. The other nodes are labeled as follows. For a node u labeled with (e', ν), its children are labeled as follows.

- If $r' = a$, then u has only one child: a leaf node labeled with $\binom{a}{d}$ for some $d \in \mathcal{D}$.

- If $r' = a[x^{=}]$, then u has only one child: a leaf node labeled with $\binom{a}{\nu(x)}$.
- If $r' = a[x^{\neq}]$, then u has only one child: a leaf node labeled with $\binom{a}{d}$ for some $d \neq \nu(x)$.
- If $r' = r_1 + r_2$, then u has only one child: a leaf node labeled with either (r_1, ν) or (r_2, ν).
- If $r' = r_1 \cdot r_2$, then u has only two children: the left child is labeled with (r_1, ν) and the right child is labeled with (r_2, ν).
- If $r' = r_1^*$, then u has either only one child: a leaf node labeled with ϵ; or at least one child labeled with (r_1, ν).
- If $r' = a \downarrow_x \cdot(r_1)$, then u has only two children: the left child is labeled with $\binom{a}{d}$ and the right child is labeled with $(r_1, \nu[x \leftarrow d])$, for some data value $d \in \mathcal{D}$.

A derivation tree t defines a data word $w(t)$ as the word read on the leaf nodes of t from left to right.

Proposition 3. *A data word $w \in L(r, \emptyset)$ if and only if there exists a derivation tree t such that $w = w(t)$.*

Proof. We start with the "only if" direction. Suppose that $w \in L(r, \emptyset)$. By induction on the length of e, we can construct the derivation tree t such that $w = w(t)$. It is a rather straightforward induction, where the induction step is based on the recursive definition of REWB, where r is either a, $a[x^{=}]$, $a[x^{\neq}]$, $r_1 + r_2$, $r_1 \cdot r_2$, r_1^* or $a \downarrow_x .(r_1)$.

Now we prove the "if" direction. We are going to show that for every node u in t, if u is labeled with (r', ν), then $w_u(t) \in L(r', \nu)$. This can be proved by induction on the *height* of the node u, which is defined as follows.

- The height of a leaf node is 0.
- The height of a node u is the maximum between the heights of its children nodes.

It is a rather straightforward induction, where the base case is the nodes with zero height and the induction step is carried on nodes of height h with the induction hypothesis assumed to hold on nodes of height $< h$.

For a node u in a derivation tree t, the word induced by the node u is the subword made up of the leaf nodes in the subtree rooted at u. We denote such subword by $w_u(t)$. Suppose $w(t) = w_1 w_u(t) w_2$, the *index pair* of the node u is the pair of integers (i, j) such that $i = \text{length}(w_1) + 1$ and $j = \text{length}(w_1 w_u(t))$.

A derivation tree t induces a binary relation R_t as follows.

$$R_t = \{(i, j) \mid (i, j) \text{ is the index pair of a node } u \text{ in } t \text{ labeled with } a \downarrow x_i \cdot (r')\}.$$

Note that R_t is a partial function from the set $\{1, \ldots, \text{length}(w(t))\}$ to itself, where if $R_t(i)$ is defined, then $i < R_t(i)$.

For a pair $(i, j) \in R_t$, we say that the variable x is associated with (i, j), if (i, j) is the index pair of a node u in t labeled with a label of the form $a \downarrow x \cdot (r')$. Two binary tuples (i, j) and (i', j'), where $i < j$ and $i' < j'$, *cross each other* if either $i < i' < j < j'$ or $i < i' < j < j'$.

Proposition 4. *For any derivation tree t, the binary relation R_t induced by it does not contain any two pairs (i,j) and (i',j') that cross each other.*

Proof. Suppose $(i,j),(i',j') \in R_t$. Then let u and u' be the nodes whose index pairs are (i,j) and (i',j'), respectively. There are two cases.

- The nodes u and u' are descendants of each other.
  Suppose u is a descendant of u'. Then, we have $i' < i < j < j'$.
- The nodes u and u' are not descendants of each other.
  Suppose the node u' is on the right side of u, that is, $w_{u'}(t)$ is on the right side of $w_u(t)$ in w. Then we have $i' < j' < i < j$.

In either case (i,j) and (i',j') do not cross each other. This completes the proof of our claim.

Now we are ready to prove Lemma 2.

Proof of Lemma 2. Suppose to the contrary that there is an REWB r over $\Sigma[x_1,\ldots,x_k]$ such that $L(r) = \mathcal{P}$, where $\Sigma = \{a\}$. Consider the following word $w \in \mathcal{P}_m$, where $m = k+2$:

$$w = \binom{a}{d_0}\binom{a}{d_1}\binom{a}{e_0}\binom{a}{e_1}\underbrace{\cdots}_{v_1}\binom{a}{d_1}\binom{a}{d_2}\underbrace{\cdots}_{w_1}\binom{a}{e_1}\binom{a}{e_2}\underbrace{\cdots}_{v_2}\binom{a}{d_2}\binom{a}{d_3}\underbrace{\cdots}_{w_2}\binom{a}{e_2}\binom{a}{e_3}\cdots\cdots$$

$$\cdots\cdots\binom{a}{e_{m-2}}\binom{a}{e_{m-1}}\underbrace{\cdots}_{v_{m-1}}\binom{a}{d_{m-1}}\binom{a}{d_m}\underbrace{\cdots}_{w_{m-1}}\binom{a}{e_{m-1}}\binom{a}{e_m}$$

where

- each of the data values in $v_1, w_1, \ldots, v_{m-1}, w_{m-1}$ appear exactly once in w;
- $d_0, d_1, \ldots, d_m, e_0, e_1, \ldots, e_m$ are pairwise different.

Let t be the derivation tree of w. Consider the binary relation R_t and the following sets A and B.

$$A = \{\text{length}(w') \mid w' \text{ is the prefix } \tbinom{a}{d_0}\tbinom{a}{d_1}\cdots\tbinom{a}{d_{l-1}}\tbinom{a}{d_l} \text{ of } w \text{ where } 1 \le l \le m-1\}$$
$$B = \{\text{length}(w') \mid w' \text{ is the prefix } \tbinom{a}{d_0}\tbinom{a}{d_1}\cdots\tbinom{a}{e_{l-1}}\tbinom{a}{e_l} \text{ of } w \text{ where } 1 \le l \le m-1\}$$

Claim. The relation R_t is a function on $A \cup B$. That is, for every $h \in A \cup B$, there is h' such that $(h,h') \in R_t$.

Proof. Suppose there exists $h \in A \cup B$ such that $R_t(h)$ is not defined. Assume that $h \in A$. Let l be the index $1 \le l \le m-1$ where $h = \text{length}(w')$ and w' is the prefix $\tbinom{a}{d_0}\tbinom{a}{d_1}\cdots\tbinom{a}{d_{l-1}}\tbinom{a}{d_l}$.

If $R_t(h)$ is not defined, then for any valuation ν found in the nodes in t, $d_l \notin \text{Image}(\nu)$. So, the word

$$w'' = \binom{a}{d_0}\binom{a}{d_1}\binom{a}{e_0}\binom{a}{e_1}\cdots\cdots\binom{a}{d_{l-1}}\binom{a}{f}\cdots\binom{a}{e_{l-1}}\binom{a}{e_l}\cdots\binom{a}{d_l}\binom{a}{d_{l+1}}\cdots\cdots$$

is also in $L(r)$, where f is a new data value. That is, the word w'' is obtained by replacing the first appearance of d_l with f. This contradicts the fact that $\mathcal{P} = L(r)$, since $w'' \notin \mathcal{P}$. The same reasoning goes for the case if $h \in B$. This completes the proof of our claim.

Remark 1. Without loss of generality, we can assume that each variable in the REWB r is introduced only once. Otherwise, we can rename the variable.

Claim. There exist $(h_1, h_2), (h_1', h_2') \in R_t$ such that $h_1 < h_2 < h_1' < h_2'$ and $h_1, h_1' \in A$ and both $(h_1, h_2), (h_1', h_2')$ have the same associated variable.

Proof. The cardinality $|A| = k + 1$. So there exists a variable $x \in \{x_1, \ldots, x_k\}$ and $(h_1, h_2), (h_1', h_2') \in R_t$ such that $(h_1, h_2), (h_1', h_2')$ are associated with the variable x. By Remark 1, no variable is written twice in e, so the nodes u, u' associated with $(h_1, h_2), (h_1', h_2')$ are not descendants of each other, so we have $h_1 < h_2 < h_1' < h_2'$, or $h_1' < h_2' < h_1 < h_2$. This completes the proof of our claim.

¿From the following claim we immediately get that $\mathcal{P} \neq L(r)$.

Claim. There exists a word $w'' \notin \mathcal{P}$, but $w'' \in L(r)$.

Proof. The word w'' is constructed from the word w. By Claim 7, there exist $(h_1, h_2), (h_1', h_2') \in R_t$ such that $h_1 < h_2 < h_1' < h_2'$ and $h_1, h_1' \in A$ and both h_1, h_1' have the same associated variable.

By definition of the language $\mathcal{P}$, between h_1 and h_1', there exists an index $l \in B$ such that $h_1 < l < h_1'$. (Recall that the set A contains the positions of the data values d's, and the set B the positions of the data values e's.)

Let h be the maximum of such indices. The index h is not the index of the last e, hence $R_t(h)$ exists and $R_t(h) < h_2$, by Proposition 4. Now the data value in $R_t(h)$ is different from the data value in position h. To get w'', we change the data value in the position h with a new data value f, and it will not change the acceptance of the word w'' by the REWB r.

However, the word w'' given by

$$w'' = \binom{a}{d_0}\binom{a}{d_1}\binom{a}{e_0}\binom{a}{e_1}\cdots\cdots\binom{a}{e_{l-1}}\binom{a}{f}\cdots\binom{a}{e_l}\binom{a}{e_{l+1}}\cdots\cdots$$

is not in $\mathcal{P}$, by definition.

Thus, this completes the proof of our claim.

This completes the proof of Lemma 2.

Proof of Theorem 2

To prove the NP-upper bound we will need the following Proposition.

Proposition 5. *For every REWB r over $\Sigma[x_1, \ldots, x_k]$ and every valuation ν compatible with r, if $L(r, \nu) \neq \emptyset$, then there exists a data word $w \in L(r, \nu)$ of length $\mathcal{O}(|r|)$.*

Proof. The proof is by induction on the length of r. The basis is when the length of r is 1. There are two cases: $a[c]$ and a; and it is trivial that our proposition holds.

Let r be an REWB and ν a valuation compatible with r. For the induction hypothesis, we assume that our proposition holds for all REWBs of shorter length than r. For the induction step, we prove our proposition for r. There are four cases.

- Case 1: $r = r_1 + r_2$.
  If $L(r, \nu) \neq \emptyset$, then by the induction hypothesis, either $L(r_1, \nu)$ or $L(r_2, \nu)$ are not empty. So, either
  - there exists $w_1 \in L(r_1, \nu)$ such that $|w_1| = \mathcal{O}(|r_1|)$; or
  - there exists $w_2 \in L(r_2, \nu)$ such that $|w_2| = \mathcal{O}(|r_2|)$.

  Thus, by definition, there exists $w \in L(r, \nu)$ such that $|w| = \mathcal{O}(|r|)$.
- Case 2: $r = r_1 \cdot r_2$.
  If $L(r, \nu) \neq \emptyset$, then by the definition, $L(r_1, \nu)$ and $L(r_2, \nu)$ are not empty. So by the induction hypothesis
  - there exists $w_1 \in L(r_1, \nu)$ such that $|w_1| = \mathcal{O}(|r_1|)$; and
  - there exists $w_2 \in L(r_2, \nu)$ such that $|w_2| = \mathcal{O}(|r_2|)$.

  Thus, by definition, $w_1 \cdot w_2 \in L(r, \nu)$ and $|w_1 \cdot w_2| = \mathcal{O}(|r|)$.
- Case 3: $r = (r_1)^*$.
  This case is trivial since $\varepsilon \in L(r, \nu)$.
- Case 4: $r = a \downarrow_{x_i} (r_1)$.
  If $L(r, \nu) \neq \emptyset$, then by the definition, $L(r_1, \nu[x_i \leftarrow d])$ is not empty, for some data value d. By the induction hypothesis, there exists $w_1 \in L(r_1, \nu[x_i \leftarrow d])$ such that $|w_1| = \mathcal{O}(|r_1|)$. By definition, $\binom{a}{d} w_1 \in L(r, \nu)$.

This completes the proof of Proposition 5.

The NP membership follows from Proposition 5, where given a REWB r, we simply guess a data word $w \in L(r)$ of length $\mathcal{O}(|r|)$. The verification that $w \in L(r)$ can be done deterministically in polynomial time.

Note that the data values here can be made small as well. It is well known that in a word accepted by a register automaton one can replace the data values with the ones from the set $1, \ldots k+1$, where k is the number of registers [18, 22], while retaining the acceptance condition. Thus we can always assume that the values appearing in our word are not bigger than the number of variables in our expression plus one.

We prove NP hardness via a reduction from 3-SAT.

Assume that $\varphi = (\ell_{1,1} \vee \ell_{1,2} \vee \ell_{1,3}) \wedge \cdots \wedge (\ell_{n,1} \vee \ell_{n,2} \vee \ell_{n,3})$ is the given 3-CNF formula, where each $\ell_{i,j}$ is a literal. Let $x_1, \ldots x_k$ denote the variables occurring in φ. We say that the literal $\ell_{i,j}$ is negative, if it is a negation of a variable. Otherwise, we call it a positive literal.

We will define a REWB r over $\Sigma[y_1, z_1, y_2, z_2, \ldots, y_k, z_k]$ of length $\mathcal{O}(n)$ such that φ is satisfiable if and only if $L(r) \neq \emptyset$.

Let r be the following REWB.

$$r := a \downarrow_{y_1} (a \downarrow_{z_1} (a \downarrow_{y_2} (a \downarrow_{z_2} (\cdots (a \downarrow_{y_k} (a \downarrow_{z_k} (\\ (r_{1,1} + r_{1,2} + r_{1,3}) \cdots (r_{n,1} + r_{n,2} + r_{n,3}) \ldots),$$

$$r_{i,j} := \begin{cases} b[y_k^{=} \wedge z_k^{=}] & \text{if } \ell_{i,j} = x_k \\ b[y_k^{=} \wedge z_k^{\neq}] + b[z_k^{=} \wedge y_k^{\neq}] & \text{if } \ell_{i,j} = \neg x_k \end{cases}$$

Obviously, $|r| = \mathcal{O}(n)$. We are going to prove that φ is satisfiable if and only if $L(r) \neq \emptyset$.

Assume first that φ is satisfiable. Then there is an assignment $f : \{x_1, \ldots, x_k\} \mapsto \{0, 1\}$ making φ true. We define the evaluation $\nu : \{y_1, z_1, \ldots y_n, z_n\} \mapsto \{0, 1\}$ as follows.

- If $f(x_i) = 1$, then $\nu(y_i) = \nu(z_i) = 1$.
- If $f(x_i) = 0$, then $\nu(y_i) = 0$ and $\nu(z_i) = 1$.

We define the following data word.

$$w := \binom{a}{\nu(y_1)}\binom{a}{\nu(z_1)} \cdots \binom{a}{\nu(y_k)}\binom{a}{\nu(z_k)} \underbrace{\binom{b}{1} \cdots \binom{b}{1}}_{n \text{ times}}$$

To see that $w \in L(r)$, we observe that the first $2k$ labels are parsed to bind values $y_1, z_1, \ldots y_k, z_k$ to corresponding values determined by ν. To parse the remaining $\binom{b}{1} \cdots \binom{b}{1}$, we observe that for each $i \in \{1, \ldots, n\}$, $\ell_{i,1} \vee \ell_{i,2} \vee \ell_{i,3}$ is true according to the assignment f if and only if $\binom{b}{1} \in L(r_{i,1} + r_{i,2} + r_{i,3}, \nu)$.

Conversely, assume that $L(r) \neq \emptyset$. Let

$$w = \binom{a}{d_{y_1}}\binom{a}{d_{z_1}} \cdots \binom{a}{d_{y_k}}\binom{a}{d_{z_k}}\binom{b}{d_1} \cdots \binom{b}{d_n} \in L(r).$$

We define the following assignment $f : \{x_1, \ldots, x_k\} \mapsto \{0, 1\}$.

$$f(x_i) = \begin{cases} 1 \text{ if } d_{y_i} = d_{z_i} \\ 0 \text{ if } d_{y_i} \neq d_{z_i} \end{cases}$$

We are going to show that f is a satisfying assignment for φ. Now since $w \in L(r)$, we have

$$\binom{b}{d_1} \cdots \binom{b}{d_n} \in L((r_{1,1} + r_{1,2} + r_{1,3}) \cdots (r_{n,1} + r_{n,2} + r_{n,3}), \nu),$$

where $\nu(y_i) = d_{y_i}$ and $\nu(z_i) = d_{z_i}$. In particular, we have for every $j = 1, \ldots, n$,

$$\binom{b}{d_j} \in L(r_{j,1} + r_{j,2} + r_{j,3}, \nu).$$

W.l.o.g, assume that $\binom{b}{d_j} \in L(r_{j,1})$. There are two cases.

- If $r_{j,1} = b[y_i^{=} \wedge z_i^{=}]$, then by definition, $\ell_{j,1} = x_i$, hence the clause $\ell_{j,1} \vee \ell_{j,2} \vee \ell_{j,3}$ is true under the assignment f.
- If $r_{j,1} = b[y_i^{=} \wedge z_i^{\neq}] + b[z_i^{=} \wedge y_i^{\neq}]$, then by definition, $\ell_{j,1} = \neg x_i$, hence the clause $\ell_{j,1} \vee \ell_{j,2} \vee \ell_{j,3}$ is true under the assignment f.

Thus, the assignment f is a satisfying assignment for the formula φ. This completes the proof of our theorem.

Proof of Proposition 1

First we consider the case of simple REWBs.

The proof is by induction on the length of r. The basis is when the length of r is 1. There are three cases: $a[x_i^{=}]$, $a[x_i^{\neq}]$ and a; and it is trivial that our proposition holds.

Let r be an REWB and ν a valuation compatible with r. For the induction hypothesis, we assume that our proposition holds for all REWBs of shorter length than r. For the induction step, we prove our proposition for r. There are four cases.

- Case 1: $r = r_1 + r_2$.
  By the induction hypothesis, both $L(r_1, \nu)$ and $L(r_2, \nu)$ are not empty, thus, by definition, $L(r, \nu)$ is also not empty.
- Case 2: $r = r_1 \cdot r_2$.
  By the induction hypothesis, both $L(r_1, \nu)$ and $L(r_2, \nu)$ are not empty, thus, by definition, $L(r, \nu)$ is also not empty.
- Case 3: $r = (r_1)^*$.
  This case is trivial, since $\varepsilon \in L(r, \nu)$, thus, $L(r, \nu) \neq \emptyset$.
- Case 4: $r = a \downarrow_{x_i} (r_1)$.
  By the induction hypothesis, $L(r_1, \nu[x_i \leftarrow d])$ is not empty for some arbitrary data value d. Thus, by definition, $L(r, \nu)$ is also not empty.

Next we prove the claim for positive REWBs.

Namely what we show is that if for any $d \in \mathcal{D}$ we define $\nu_d(x) := d$, with x a variable in our expression, we will have $L(r, \nu_d) \neq \emptyset$.

The proof is by induction on the length of r. The basis is when the length of r is 1. There are two cases: $a[c]$ and a; and it is trivial that our proposition holds.

Let r be a positive REWB. For the induction hypothesis, we assume that our proposition holds for all REWBs of shorter length than r. For the induction step, we prove our proposition for r. There are four cases.

- Case 1: $r = r_1 + r_2$.
  By the induction hypothesis, $L(r_1, \nu_d)$ and $L(r_2, \nu_d)$ are nonempty, thus, by definition, $L(r, \nu_d)$ is also not empty.
- Case 2: $r = r_1 \cdot r_2$.
  By the induction hypothesis, both $L(r_1, \nu_d)$ and $L(r_2, \nu_d)$ are not empty, thus, by definition, $L(r, \nu_d)$ is also not empty.

– Case 3: $r = (r_1)^*$.
  This case is trivial, since $\varepsilon \in L(r, \nu_d)$, thus, $L(r, \nu_d) \neq \emptyset$.
– Case 4: $r = a \downarrow_{x_i} (r_1)$.
  By the induction hypothesis, for any d we have $L(r_1, \nu_d)$ is not empty. Take any $w \in L(r_1, \nu_d)$. Then by the definition we have that $\binom{a}{d} \cdot w \in L(r, \nu_d)$, so we have that $L(r, \nu_d) \neq \emptyset$.

This completes the proof of Proposition 1.

Proof of Theorem 3

We are first going to prove that given an REWB r over $\Sigma[x_1, \ldots, x_k]$, checking whether $L(e) = (\Sigma \times \mathcal{D})^*$ is undecidable. This immediately implies that given r_1, r_2, checking whether $L(r_1) \subseteq L(r_2)$ is undecidable, hence, the second item of our theorem.

The proof is similar to the proof of the universality of register automata in [24]. The reduction is via Post Correspondence Problem (PCP), which is defined as follows. An instance of PCP is a set of of pair of strings

$$I = \{(u_1, v_1), \ldots, (u_n, v_n)\},$$

where $u_i, v_i \in \Sigma^*$. A solution of the instance I is a sequence $l_1, \ldots, l_m$ such that $u_{l_1} \cdots u_{l_m} = v_{l_1} \cdots v_{l_m}$.

Let $\$, \#$ be two special symbols not in Σ. Now a solution $l_1, \ldots, l_m$ of the PCP instance I can be encoded into data word $w_1 \binom{\#}{h} w_2$ over $\Sigma \cup \{\$, \#\}$, where

$$w_1 = \tbinom{\$}{e_1}\tbinom{a_1}{d_1} \cdots \tbinom{a_{\ell_1}}{d_{\ell_1}}\tbinom{\$}{e_2}\tbinom{a_{\ell_1+1}}{d_{\ell_1+1}} \cdots \tbinom{a_{\ell_1+\ell_2}}{d_{\ell_1+\ell_2}}\tbinom{\$}{e_3} \cdots\cdots \tbinom{\$}{e_m}\tbinom{a_{\ell_1+\cdots+\ell_{m-1}}}{d_{\ell_1+\cdots+\ell_{m-1}}} \cdots \tbinom{a_\ell}{d_\ell}$$

$$w_2 = \tbinom{\$}{g_1}\tbinom{b_1}{f_1} \cdots \tbinom{b_{\ell_1}}{f_{\ell_1}}\tbinom{\$}{g_2}\tbinom{b_{\ell_1+1}}{f_{\ell_1+1}} \cdots \tbinom{b_{\ell_1+\ell_2}}{f_{\ell_1+\ell_2}}\tbinom{\$}{g_3} \cdots\cdots \tbinom{\$}{g_m}\tbinom{b_{\ell_1+\cdots+\ell_{m-1}}}{f_{\ell_1+\cdots+\ell_{m-1}}} \cdots \tbinom{b_\ell}{f_\ell}$$

where $\ell = \ell_1 + \ell_2 + \cdots + \ell_m$, and

(C1) The symbol # appears only once.
(C2) $\mathsf{Proj}_\Sigma(w_1) \in (\$ \cdot u_1 + \cdots + \$ \cdot u_n)^*$.
(C3) $\mathsf{Proj}_\Sigma(w_2) \in (\$ \cdot v_1 + \cdots + \$ \cdot v_n)^*$.
(C4) The data values e_i's and d_i's are pairwise different.
(C5) The data values g_i's and f_i's are pairwise different.
(C6) $e_1 = g_1$ and $e_m = g_m$.
(C7) $d_1 = f_1$ and $d_{\ell_m} = f_{\ell_m}$.
(C8) For all $i \in \{1, \ldots, m-1\}$, there exists $j \in \{1, \ldots, m-1\}$ such that $e_i = g_j$ and $e_{i+1} = g_{j+1}$.
(C9) For all $i \in \{1, \ldots, \ell_m - 1\}$, there exists $j \in \{1, \ldots, \ell_m - 1\}$ such that $d_i = f_j$ and $d_{i+1} = f_{j+1}$.
(C10) For all $i, j \in \{1, \ldots, \ell_m\}$, if $d_i = f_j$, then $a_i = b_j$.
(C11) For all $i, j \in \{1, \ldots, m\}$, if $e_i = g_j$, then $(a_{\ell_{i-1}+1} \cdots a_{\ell_i}, b_{\ell_{j-1}+1} \cdots b_{\ell_j}) \in I$.

Now it is straightforward to show that there exists a solution to the PCP instance I if and only if there exists a data word over $\Sigma \cup \{\$, \#\}$ that satisfies Conditions (C1)–(C11) above.

We now construct an REWB e over $\Sigma_1[x_1, \ldots, x_k]$ where $\Sigma_1 = \Sigma \cup \{\$, \#\}$ that accepts a data word w that does not satisfies at least one of the Conditions (C1) to (C11) above. Such REWB e can be constructed by taking the union of the negation of each of Conditions (C1) to (C11), and it is a rather straightforward observation that the negation of each of them can be stated as an REWB. Hence, we have that the PCP instance I has no solution if and only if $L(r) = (\Sigma_1 \times \mathcal{D})^*$. This concludes our proof in the case of multiple variables.

We now prove that we get undecidability even when using expressions with only one variable. The proof is a slight modification of the proof in multi-variable case and for completeness we present it here.

Let r be an REWB over $\Sigma[x]$.

Let $\$, \#$ be two special symbols not in Σ. Let $\Gamma = \Sigma \cup \{\$, \#\}$. Now a solution $l_1, \ldots, l_m$ of the PCP instance I can be encoded into data word $w_1 \binom{\#}{h} \mathrm{REV}(w_2)$ over $\Sigma \cup \{\$, \#\}$, where w_1, w_2 are defined as above and $\mathrm{REV}(w_2)$ is the reversal of w_2.

We then construct an REWB r over $\Gamma[x_1, \ldots, x_k]$ that accepts a data word $w = w_1 \# \mathrm{REV}(w_2)$ such that $w_1 \# w_2$ does not satisfies at least one of the Conditions (C1) to (C11) above. The REWB r is obtained by taking the union of the following.

- The negations of each (C1), (C2), (C3) which can be written in a standard regular expression without variables.
- The negation of (C4) which can be written as:

$$\Big(\Gamma^*\$ \downarrow_x (\Gamma^*\$[x^=]) \quad + \quad \Gamma^* \bigcup_{a \in \Sigma} \big(a \downarrow_x (\Gamma^* a[x^=])\big)\Big) \# \Gamma^*$$

  The negation of (C5) can be written in a similar manner.
- The negation of (C6) which can be written as:

$$\$ \downarrow_x (\Gamma^* \cdot \$[x^{\neq}]) \quad + \quad \Gamma^*\$ \downarrow_x (\# \cdot \Sigma^*\$[x^{\neq}])\Gamma^*.$$

  The negation of (C7) can be written in a similar manner.
- The negation of (C8) which can be written as:

$$\Gamma^*\$ \downarrow_x \left(\Gamma^* \#(\$[x^{\neq}]\Sigma)^* + \Sigma^*\$ \downarrow_x (\Gamma * \#\Gamma * \$[x^{\neq}])\Sigma^*\$[x^=]\right).$$

  Note that here we use the fact that (C8) can be paraphrased as follows:
  1. For all $i \in \{1, \ldots, m-1\}$ exists $j \in \{1, \ldots, m-1\}$ such that $e_i = g_j$
  2. For all $i \in \{1, \ldots, m-1\}$ and for all $j \in \{1, \ldots, m-1\}$ if $e_i = g_j$ then $e_{i+1} = g_{j+1}$.

  (Recall that by (C6) we have that $e_1 = g_1$.)

  The negation of (C9) can be written in a similar manner.

– The negation of (C10) and the negation of (C11), which can be written in a straightforward manner using only one variable.

It is straightforward to see that the PCP instance I has no solution if and only if $L(r) = (\Sigma_1 \times \mathcal{D})^*$. This concludes our proof of Theorem 3.

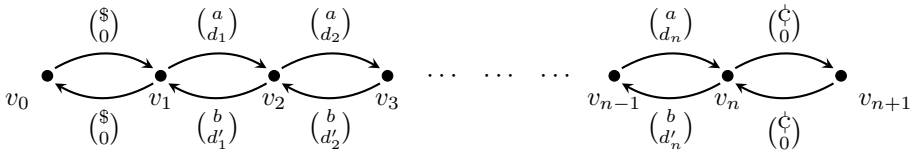
Proof of Proposition 2

To make the proof more precise we will introduce some additional notation. For the sake of readability, we will first prove it for REWB using multiple variables.

We write $\mathsf{Path}(G, r, \nu)$ for the set of all paths π in G such that $w(\pi) \in L(r, \nu)$ and ν is compatible with r; and $\mathsf{Path}(G, r, s, t, \nu)$ for the set of all paths π in G from s to t such that $w(\pi) \in L(r, \nu)$ and ν is compatible with r. Similarly, for a closed expression r, we can write $\mathsf{Path}(G, r)$ to denote the set of all paths π in G such that $w(\pi) \in L(r)$; and $\mathsf{Path}(G, r, s, t)$ the set of all paths π in G from s to t such that $w(\pi) \in L(r)$.

We can now continue with the proof.

For $n \geq 2$, the graph G is defined as follows, where $0, d_1, \ldots, d_n, d'_1, \ldots, d'_n$ are pairwise different.



Obviously, G has $n + 2$ vertices and $2n + 2$ edges.

We define the following auxiliary REWBs $r_{k,a}$ and $r_{k,b}$ for each $k = 0, 1, \ldots, n$ as follows.

$$
\begin{aligned}
r_{0,a} &:= a \\
r_{0,b} &:= b \\
r_{k,a} &:= a \downarrow_{x_k} \left((r_{k-1,b})^* \cdot \$ \cdot \$ \cdot a^* \cdot a[x_k^{=}] \right) \\
r_{k,b} &:= b \downarrow_{x_k} \left((r_{k-1,a})^* \cdot \dot{\varsigma} \cdot \dot{\varsigma} \cdot b^* \cdot b[x_k^{=}] \right)
\end{aligned}
$$

The REWB r is defined as $\$ \cdot \varphi_{n,a}^* \cdot \dot{\varsigma}$. The two nodes s and t are v_0 and v_{n+1}, respectively. Note that the length of $r_{k,a}$ and $r_{k,b}$ is $\mathcal{O}(n)$.

We claim that every path $\pi \in \mathsf{Path}(G, r, s, t)$ is of length $\Omega(2^{\lfloor n/2 \rfloor})$. For this, we need a few auxiliary claims.

Claim. For every $k \in \{0, 1, \ldots, n\}$ and for every node $v_i, v_j \in \{v_1, \ldots, v_n\}$ and for every valuation ν, the following holds.

1. There exists a path π from v_i to v_j such that $w(\pi) \in L(r_{k,a}, \nu)$ if and only if $j = i + 1$.

2. There exists a path π from v_i to v_j such that $w(\pi) \in L(r_{k,b}), \nu$ if and only if $i = j + 1$.

Proof. The proof is by induction on k. The basis $k = 0$ is trivial, due to the definition that $r_{0,a} = a$ and $r_{0,b} = b$, Thus,

- there exists a path π from v_i to v_j such that $w(\pi) \in L(a, \nu)$ if and only if the path is of consists of one edge labeled with a, which means $j = i + 1$; and
- there exists a path π from v_i to v_j such that $w(\pi) \in L(b, \nu)$ if and only if the path is of consists of one edge labeled with b, which means $i = j + 1$.

For the induction hypothesis we assume that our claim holds for the case of k.

For the induction step, we prove item (1) for the case of $k + 1$. Item (2) can be proved in a similar manner. The "only if" direction is as follows. Suppose there exists a path π from v_i to v_j such that $w(\pi) \in L(r_{k+1,a})$. By definition of $r_{k+1,a}$, we have

$$r_{k+1,a} = a \downarrow_{x_{k+1}} \left((r_{k,b})^* \cdot \$ \cdot \$ \cdot a^* \cdot a[x_{k+1}^{=}] \right)$$

This means that the variable x_{k+1} is assigned with the data value d_i. Since the last step of the expression $\varphi_{k+1,a}$ is $a[x_{k+1}^{=}]$, and all the data values $0, d_1, \ldots, d_n$, $d'_1, \ldots, d'_n$ are all different, the last edge in the path π must be $v_i \binom{a}{d_i} v_{i+1}$, which means $j = i + 1$.

The "if" direction is as follows. We want to show that there exists a path π from v_i to v_{i+1} such that $w(\pi) \in L(r_{k+1,a}, \nu)$. By definition,

$$r_{k+1,a} = a \downarrow_{x_{k+1}} \left((r_{k,b})^* \cdot \$ \cdot \$ \cdot a^* \cdot a[x_{k+1}^{=}] \right)$$

We claim that there are the following paths for any valuation ν.

- There is a path π_1 from v_i to v_{i+1} such that $w(\pi_1) \in L(a, \nu[x_{k+1} \leftarrow d_i])$.
- There is a path π_2 from v_{i+1} to v_1 such that $w(\pi_2) \in L(r_{k,b}^*, \nu[x_{k+1} \leftarrow d_i])$.
- There is a path π_3 from v_1 to v_0 such that $w(\pi_3) \in L(\$, \nu[x_{k+1} \leftarrow d_i])$.
- There is a path π_4 from v_0 to v_1 such that $w(\pi_4) \in L(\$, \nu[x_{k+1} \leftarrow d_i])$.
- There is a path π_5 from v_1 to v_i such that $w(\pi_5) \in L(a^*, \nu[x_{k+1} \leftarrow d_i])$.
- There is a path π_6 from v_i to v_{i+1} such that $w(\pi_6) \in L(a[x_{k+1}^{=}], \nu[x_{k+1} \leftarrow d_i])$.

The existence of all the paths, except π_2, are trivially established. The existence of the path π_2 follows from the induction hypothesis that there exists a path $\pi_{(l+1,l)}$ from v_l to v_{l+1} such that $w(\pi_{(l+1,l)}) \in L(r_{k,b}, \nu[x_{k+1} \leftarrow d_i])$, for every $l = i+1, \ldots, 2$. Thus, we establish the existence of a path π from v_i to v_{i+1} such that $w(\pi) \in L(\varphi_{k+1,a}, \nu)$. This completes the proof of the "if" direction, hence the proof of our claim.

Now Claim 7 immediately implies the following claim.

Claim. For every $k \in \{0, 1, \ldots, n\}$ and for every node $v_i, v_j \in \{v_1, \ldots, v_n\}$ and for every valuation ν, the following holds.

1. There exists a path π from v_i to v_j such that $w(\pi) \in L(r^*_{k,a}, \nu)$ if and only if $j \geq i$.
2. There exists a path π from v_i to v_j such that $w(\pi) \in L(r^*_{k,b}, \nu)$ if and only if $i \geq j$.

We are going to need the following inequality. For any integer $k \geq 1$, for any integer $m \geq 1$,

$$\sum_{1 \leq i \leq m} i^k \geq \frac{m^{k+1}}{k+1}. \tag{2}$$

It can be proved by induction on m. The base case when $m = 1$ is trivial. Assume now that the claim holds for $m \geq 1$. For $m + 1$ we have

$$\sum_{1 \leq i \leq m+1} i^k = \sum_{1 \leq i \leq m} i^k + (m+1)^k \geq \frac{m^{k+1}}{k+1} + (m+1)^k = \frac{m^{k+1} + (k+1)(m+1)^k}{k+1} \geq \frac{(m+1)^{k+1}}{k+1}$$

The first inequality follows from the induction hypothesis. The second inequality is obtained from the binomial expansion of $(m+1)^k$ and from the fact that $(k+1)\binom{k}{l} \geq \binom{k+1}{l}$.

Claim. For every $k \in \{0, 1, \ldots, n\}$, for every node $v_i, v_j \in \{v_1, \ldots, v_n\}$ where $i \leq j$ and for every valuation ν, the following holds.

1. Every path π in G from v_i to v_j such that $w(\pi) \in L(r^*_{a,k}, \nu)$ has length $\geq \frac{(j-i)^{k+1}}{(k+1)!}$.
2. Every path π in G from v_j to v_i such that $w(\pi) \in L(r^*_{b,k}, \nu)$ has length $\geq \frac{(j-i)^{k+1}}{(k+1)!}$.

Proof. The proof is by induction on k. The basis $k = 0$ is trivial. For the induction hypothesis, we assume that our claim holds for the case of k.

For the induction step, we prove item (1) for the case of $k + 1$. Item (2) can be proved in exactly the same manner. Let π be a path from v_i to v_j such that $w(\pi) \in L(r^*_{a,k+1}, \nu)$. By Claim 7, the path π consists of the path $\pi_{l,l+1}$ from the v_l to v_{l+1} such that $w(\pi_{l,l+1}) \in L(r_{a,k+1}, \nu)$ for every $l = i, \ldots, j-1$.

Now for every $l = i, \ldots, j-1$, the path $\pi_{l,l+1}$ consists of the following paths.

- A path π_1 from v_l to v_{l+1} such that $w(\pi_1) \in L(a, \nu[x_{k+1} \leftarrow d_l])$.
  The length of this path is 1.
- A path π_2 from v_{l+1} to v_1 such that $w(\pi_2) \in L(r^*_{k,b}, \nu[x_{k+1} \leftarrow d_l])$.
  By induction hypothesis, the length of this path is $\geq \frac{l^{k+1}}{(k+1)!}$.
- A path π_3 from v_1 to v_0 such that $w(\pi_3) \in L(\$, \nu[x_{k+1} \leftarrow d_l])$.
  The length of this path is 1.

– A path π_4 from v_0 to v_1 such that $w(\pi_4) \in L(\$, \nu[x_{k+1} \leftarrow d_l])$.
  The length of this path is 1.
– A path π_5 from v_1 to v_l such that $w(\pi_5) \in L(a^*, \nu[x_{k+1} \leftarrow d_l])$.
  The length of this path is $l-1$.
– A path π_6 from v_l to v_{l+1} such that $w(\pi_6) \in L(a[x^{=}_{k+1}], \nu[x_{k+1} \leftarrow d_l])$.
  The length of this path is 1.

Thus, the length of the path $\pi_{l,l+1}$ is $\geq \frac{l^{k+1}}{(k+1)!} + l + 3$. Hence,

$$\begin{aligned}
\text{the length of the path } \pi &\geq \sum_{i \leq l \leq j-1} \frac{l^{k+1}}{(k+1)!} + l + 3 \\
&\geq \sum_{i \leq l \leq j-1} \frac{l^{k+1}}{(k+1)!} \\
&\geq \sum_{1 \leq l \leq j-i} \frac{l^{k+1}}{(k+1)!} \\
&\geq \frac{(j-i)^{k+2}}{(k+2)!}
\end{aligned}$$

The last inequality is obtained by applying Formula (2).

Recall that the REWB r is defined as $\$ \cdot r^*_{n,a} \cdot ¢$ and that the two nodes s and t are v_0 and v_{n+1}, respectively. The following claim establishes that every path π from s to t such that $w(\pi) \in L(e)$ have exponential length.

Claim. Every path $\pi \in \mathsf{Path}(G, r, s, t)$ is of length $\Omega(2^{\lfloor n/2 \rfloor})$.

Proof. It is immediate from Claim 7 that every path $\pi \in \mathsf{Path}(G, r, s, t)$ is of length $\Omega(\frac{n^n}{n!})$. Since $\frac{n^n}{n!} \geq 2^{\lfloor n/2 \rfloor}$ for $n \geq 2$, our claim follows immediately.

This completes our proof of Proposition 2 for the case of multiple variables.

Now we remark that the proof above can be easily modified for the case of one variable. We simply modify the definition of $r_{k,a}$'s and $r_{k,b}$'s for each $k = 0, 1, \ldots, n$ as follows.

$$\begin{aligned}
r_{0,a} &:= a \\
r_{0,b} &:= b \\
r_{k,a} &:= a \downarrow_x \left((r_{k-1,b})^* \cdot \$ \cdot \$ \cdot a^* \cdot a[x^{=}] \right) \\
r_{k,b} &:= b \downarrow_x \left((r_{k-1,a})^* \cdot ¢ \cdot ¢ \cdot b^* \cdot b[x^{=}] \right)
\end{aligned}$$

The REWB r is defined as $\$ \cdot \varphi^*_{n,a} \cdot ¢$. Now all the claims above still hold, and hence Proposition 2.

Proof of Theorem 5

We first prove PSPACE membership. Let r be a REWB over $\Sigma[x_1, \ldots, x_k]$ and G be a data graph. We write $\mathcal{D}(G)$ to denote the set of data values appearing in G.

For a valuation $\nu : \{x_1, \ldots, x_k\} \mapsto \mathcal{D}(G)$, we define a standard regular expression $N_{G,\nu}(r)$ over the alphabet $\Sigma \times \mathcal{D}(G)$, called the normalization of r with respect to G and D, as follows.

- If $r = a$, then $N_{G,\nu}(r) = \bigcup_{d \in \mathcal{D}(G)} \binom{a}{d}$.
- If $r = a[c]$, then $N_{G,\nu}(r) = \bigcup_{d \in \mathcal{D}(G) \text{ and } d,\nu \models c} \binom{a}{d}$.
- If $r = a \downarrow_x (r)$, then $N_{G,\nu}(r) = \bigcup_{d \in \mathcal{D}(G)} (a, d) \cdot N_{G,\nu[x \leftarrow d]}(r)$.
- If $r = r_1 \cdot r_2$, then $N_{G,\nu}(r) = N_{G,\nu}(r_1) \cdot N_{G,\nu}(r_2)$.
- If $r = r_1 + r_2$, then $N_{G,\nu}(r) = N_{G,\nu}(r_1) + N_{G,\nu}(r_2)$.
- If $r = r_1^*$, then $N_{G,\nu}(r) = (N_{G,\nu}(r_1))^*$.

First we show that $N_{G,\nu}(r)$ captures the desired semantics of the REWB r.

Claim. For every valuation ν and for every path π, $w(\pi) \in L(r, \nu)$ if and only if $w(\pi) \in L(N_{G,\nu}(r))$.

Proof. We show this by induction on the length of the REWB r.

The base cases: $r = a$ and $r = a[c]$ are straightforward.

For the induction hypothesis, we assume the claim holds for all REWB r' of shorter length than r.

The induction step is as follows. There are a few cases. For the cases where $r = r_1 + r_2$, or $r = r_1 \cdot r_2$ and $r = r_1^*$, the induction step is straightforward. We consider the case when $r = a \downarrow_x (r')$.

We will show that our claim holds for r. We start with the "only if" part. Let π be a path such that

$$w(\pi) = \tbinom{a_1}{d_1}\tbinom{a_2}{d_2} \cdots \tbinom{a_n}{d_n} \in L(r, \nu).$$

By definition of $L(r, \nu)$,

$$w' = \tbinom{a_2}{d_2} \cdots \tbinom{a_n}{d_n} \in L(r', \nu[x \leftarrow d_1]).$$

Then by the induction hypothesis,

$$\tbinom{a_2}{d_2} \cdots \tbinom{a_n}{d_n} \in L(N_{G,\nu[x \leftarrow d_1]}(r')).$$

By the definition of $N_{G,\nu}$, we have $w(\pi) \in L(N_{G,\nu}(r))$.

Now we prove the "if" part. Assume that $w(\pi) \in L(N_{G,\nu}(r))$, where

$$w(\pi) = \tbinom{a_1}{d_1}\tbinom{a_2}{d_2} \cdots \tbinom{a_n}{d_n}.$$

It follows from the definition of $N_{G,\nu}(r)$ that

$$\binom{a_2}{d_2} \cdots \binom{a_k}{d_k} \in L(N_{G,\nu[x \leftarrow d_1]}(r')).$$

By the induction hypothesis,

$$\binom{a_2}{d_2} \cdots \binom{a_k}{d_k} \in L(r', \nu[x \leftarrow d_1]).$$

By the definition of $L(r, \nu)$, we obtain that $w(\pi) \in L(r, \nu)$. This completes the proof of our claim.

Observation 1 *It is readily checked that $|N_{G,\emptyset}(r)|$ is bounded by $O(|D|^{|r|})$, which is exponential in the length of the input for combined complexity (any polynomial for data complexity).*

The PSPACE upper bound now follows from simple reachability argument for regular expressions (that is answering RPQs in graph databases). Assume we are given G, s, t, a tuple $\bar{d} = (d_1, \ldots, d_l)$ and a REWB r with free variables $x_1, \ldots, x_l$. Let ν be a valuation such that $\nu(x_1) = d_1, \ldots, \nu(x_l) = d_l$.

To find a path connecting s and t we check reachability in the product automaton of $N_{G,\nu}(r)$ and G, where we view G and an automaton over $\Sigma \times D$ with the initial state s and the final state t. ¿From Observation 1 and standard on-the fly argument for reachability we get the desired upper bound. This also proves the NLOGSPACE bound in case of data complexity, since the length of normalization is now polynomial in the size of the input.

Now we prove the PSPACE-hardness of our theorem. The reduction is form QBF.

Let

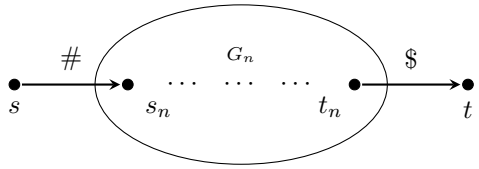
$$\Psi = \forall x_n \exists y_n \ldots \forall x_1 \exists y_1 \ \varphi$$
$$\varphi = (\ell_{1,1} \vee \ell_{1,2} \vee \ell_{1,3}) \wedge (\ell_{2,1} \vee \ell_{2,2} \vee \ell_{2,3}) \wedge \cdots \wedge (\ell_{m,1} \vee \ell_{m,2} \vee \ell_{m,3})$$

where each $\ell_{i,j}$ is a literal. We call a literal $\ell_{i,j}$ a *negative* literal, if it is a negation of a variable. Otherwise, we call it a *positive* literal.

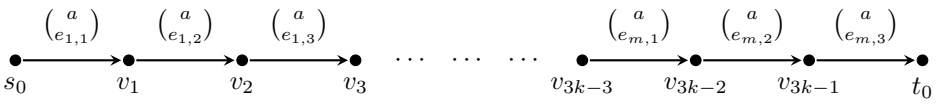
For each $i \in \{0, 1, \ldots, n\}$, we will denote $\Psi_i = \forall x_i \exists y_i \ldots \forall x_1 \exists y_1 \varphi$. Hence, $\Psi_0 = \varphi$ and $\Psi_n = \Psi$. We are going to construct (in polynomial time) a graph G, two nodes $s, t \in V(G)$ and an REWB r such that

$$\Psi \text{ is true if and only if } (s, t) \in \mathcal{Q}(r, G).$$

The construction of graph G and the two nodes $s, t \in V(G)$: The graph G is a data graph over $\Sigma = \{a, b, \#, \$\}$. Its construction is done inductively on $i \in \{0, 1, \ldots, n\}$, where G_i, s_i, t_i are constructed from Ψ_i. The desired graph G and the two nodes $s, t \in V(G)$ is the following graph.



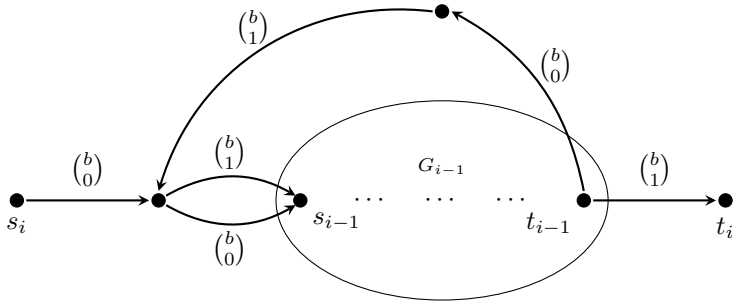
The construction of G_i, s_i, t_i is constructed inductively on i. The graph G_0 and the two vertices s_0, t_0 are as follows.



where

$$e_{i,j} = \begin{cases} 1 \text{ if the literal } \ell_{i,j} \text{ is positive} \\ 0 \text{ if the literal } \ell_{i,j} \text{ is negative} \end{cases}$$

Now we show the construction of G_i, s_i, t_i . Suppose we already constructed $G_{i-1}, s_{i-1}, t_{i-1}$. Then G_i, s_i, t_i is as follows.



The construction of the REWB r: In the following we are going to show the construction of the REWB r. We first show how to construct the auxiliary REWB r_i, for each $i = 0, 1, \ldots, m$, which is based on the formula Ψ_i. The desired REWB r is defined as $r = \# \cdot r_n \cdot \$$.

The REWB r_i is defined inductively on $k = i$. First we set

$$r_0 = \text{clause}_1 \cdot \text{clause}_2 \cdots \text{clause}_m,$$

where each clause_i is defined as follows.

$$\text{clause}_i = a[x^{=}_{i,1}] \cdot a \cdot a + a \cdot a[x^{=}_{i,2}] \cdot a + a \cdot a \cdot a[x^{=}_{i,3}]$$

and $x_{i,1}, x_{i,2}, x_{i,3}$ are the variables in the literals $\ell_{i,1}, \ell_{i,2}, \ell_{i,3}$, respectively.

Now, assuming we have the REWB r_{i-1}, we define r_i as follows.

$$r_i = \Big(b \downarrow_{x_i} \big(b \downarrow_{y_i} (r_{i-1}) \cdot b[x^{=}_i] \big) \Big)^*.$$

Finally we set $r = \# \cdot r_n \cdot \$$.

It is straightforward to verify that the construction of both the data graph G and the REWB r runs in time polynomial in the length of the formula Ψ.

Remark 2. For every $i = 0, 1, \ldots, n$,

– the formula Ψ_i has the free variables $x_{i+1}, y_{i+1}, \ldots, x_n, y_n$;
– the REWB r_i has the free variables $x_{i+1}, y_{i+1}, \ldots, x_n, y_n$.

Moreover, for a tuple $\bar{d} \in \{0,1\}^{2(n-i)}$, we write $\Psi_i(\bar{d})$ to denote the formula Ψ_i in which the variables $x_{i+1}, y_{i+1}, \ldots, x_n, y_n$ are assigned with $\bar{d}$.

To prove that Ψ is true if and only if $(s,t) \in \mathcal{Q}(r,G)$, we prove the following claim.

Claim. For each $i = 0, 1, \ldots, n$ and for every tuple $\bar{d} \in \{0,1\}^{2(n-i)}$, $\Psi_i(\bar{d})$ is true if and only if $((s_i,t_i),\bar{d}) \in \mathcal{Q}(r_i,G_i)$,

Proof. The proof is by induction on i. The basis is $i = 0$. We have to prove that $\Psi_i(\bar{d})$ is true if and only if $((s_0,t_0),\bar{d}) \in \mathcal{Q}(r_0,G_0)$.

Let for each $i = 1, \ldots, m$ and $j = 1, 2, 3$, we write $d_{i,j}$ to denote the 0-1 value assigned to the variable in the literal $\ell_{i,j}$. Let ν denote the valuation where $\nu(x_1), \nu(y_1), \ldots, \nu(x_n), \nu(y_n)$ are assigned with $\bar{d}$, respectively. Then, we have

$$\Psi_0(\bar{d}) \text{ is true}$$
$$\Updownarrow$$
$$\text{every clause } (\ell_{i,1} \vee \ell_{i,2} \vee \ell_{i,3}) \text{ is true under the assignment } \nu$$
$$\Updownarrow$$
$$\text{for each } i = 1, \ldots, m, \text{ there exists } j \in \{1,2,3\} \text{ such that } d_{i,j} = \begin{cases} 1 \text{ if } \ell_{i,j} \text{ is positive} \\ 0 \text{ if } \ell_{i,j} \text{ is negative} \end{cases}$$
$$\Updownarrow$$
$$\text{for each } i = 1, \ldots, m, w(\pi_i) \in L(\text{clause}_i, \nu) \text{ where } \pi_i = v_{3i+0}\tbinom{a}{d_{i,1}} v_{3i+1}\tbinom{a}{d_{i,2}} v_{3i+2}\tbinom{a}{d_{i,3}} v_{3i+3}$$
$$\Updownarrow$$
$$((s_0,t_0),\bar{d}) \in \mathcal{Q}(r_0,G_0)$$

For the induction hypothesis, we assume that $\Psi_i(\bar{d})$ is true if and only if $((s_i,t_i),\bar{d}) \in \mathcal{Q}(r_i,G_i)$. For the induction step, we prove the claim for $i+1$, which follows from the following equality.

$$\Psi_{i+1}(\bar{d}) \text{ is true}$$
$$\Updownarrow$$
$$\text{there exist } e_0, e_1 \in \{0,1\} \text{ such that } \Psi_i(\bar{d}0e_0) \text{ and } \Psi_i(\bar{d}1e_1) \text{ are true}$$
$$\Updownarrow$$
$$\text{there exist } e_0, e_1 \in \{0,1\} \text{ such that } ((s_i,t_i),\bar{d}0e_0), ((s_i,t_i),\bar{d}1e_1) \in \mathcal{Q}(r_i,G_i).$$
$$\Updownarrow$$
$$\text{there exists a path } \pi \text{ from } s_{i+1} \text{ to } t_{i+1} \text{ such that } w(\pi) \in L(r_{i+1},\bar{d})$$

The last inequality follows from the definition of r_{i+1}, where

$$r_{i+1} = \Big(b \downarrow_{x_{i+1}} \big(b \downarrow_{y_{i+1}} (r_i) \cdot b[x_{i+1}^{=}]\big)\Big)^*$$

and to go from the vertex s_{i+1} to t_{i+1}, the path π has to go thorough G_i at least twice: once when the variable x_{i+1} is assigned with 0 and at least once when the variable x_{i+1} is assigned with 1. Thus, we have $\Psi_{i+1}(\bar{d})$ is true if and only if $((s_{i+1}, t_{i+1}), \bar{d}) \in \mathcal{Q}(r_{i+1}, G_{i+1})$.

This completes the proof of our claim.

This concludes the proof of the hardness part, hence, our theorem.

8 Variable automata for querying graphs

In this section we continue our search for query formalisms suitable for data graphs. As we have seen before, both in the previous sections and in e.g. [21], query languages tend to have either polynomial or PSPACE combined complexity when evaluated on graph databases. A natural question to ask is if we can find a reasonable formalism whose combined complexity will be between these two classes.

Here we do so by using variable automata introduced in [15]. These automata can be viewed as less procedural than register automata; in fact they can be seen as NFAs with a guess of values to be assigned to variables, with the run of the automaton verifying correctness of the guess. Thus, they are more likely to be usable as a querying mechanism than register automata. Originally they were defined on words over infinite alphabets [15], but it is straightforward to extend them to the setting of data words. In what follows we define variable automata as a query language, give examples of some queries one can post using them and show that they can be evaluated in NP-time. At the end we look at some restrictions that will allow decidable query containment.

We begin by defining variable automata formally.

Definition 2. *Let Σ be a finite alphabet and $\mathcal{D}$ an infinite domain of data values. We will also assume that we have a countable set V of variables. A* variable finite automaton *(or VFA for short) over $\Sigma \times \mathcal{D}$ is a pair $\mathcal{A} = (\Gamma, A)$, where A is an NFA over the alphabet $\Sigma \times \Gamma$, and $\Gamma = C \cup X \cup \{\star\}$ such that:*

- *$C \subseteq \mathcal{D}$ is a finite set of data values called* constants
- *$X \subseteq V$ is a finite set of* bound variables*, and*
- *$\star$ is a symbol for the* free variable*.*

Next we define when a VFA accepts a data word $w = w_1 w_2 \dots w_n \in (\Sigma \times \mathcal{D})^*$. For each letter $u = \binom{a}{d}$ in $\Sigma \times \mathcal{D}$, we let $\lambda(u) = a$ (label projection) and $\delta(u) = d$ (data projection).

Let $v = v_1 v_2 \dots v_n \in (\Sigma \times \Gamma)^*$ be a word accepted by A. We will say that v is a *witnessing pattern* for w (or that w is a *legal instance* of v) if the following holds:

1. $\lambda(v_i) = \lambda(w_i)$, for $i = 1, \dots, n$,
2. $\delta(v_i) = \delta(w_i)$ whenever $\delta(v_i) \in C$,

3. if $\delta(v_i), \delta(v_j) \in X$, then $\delta(w_i) = \delta(w_j)$ iff $\delta(v_i) = \delta(v_j)$ and $\delta(w_i), \delta(w_j) \notin C$,
4. $\delta(v_i) = \star$ and $\delta(v_j) \neq \star$, then $\delta(w_i) \neq \delta(w_j)$.

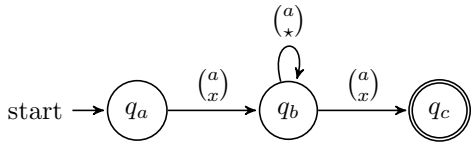
Intuitively the definition states that in a legal instance constants and finite alphabet part will remain unchanged (conditions 1 and 2), while every bound variable is assigned with the same *unique* data value from $\mathcal{D} - C$ (condition 3) and every occurrence of the free variable $\star$ is freely assigned any data value from $\mathcal{D} - C$ that is not assigned to any of the bound variables (condition 4). Note that the condition 4 is a lot stronger that saying that $\star$ is just a wild card.

We now define the *language of* $\mathcal{A}$, or simply $L(\mathcal{A})$ for short, as the set of all data words w for which there exists a witnessing pattern $v \in L(A)$. That is a word is accepted by $\mathcal{A}$ if there is a witnessing pattern for it that is accepted by the underlying NFA A.

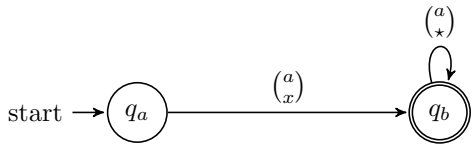
Note that it is straightforward to define regular expressions for VFAs that will simply inherit the associated semantics.

Example 2. Here we give a few examples of languages accepted by VFAs.

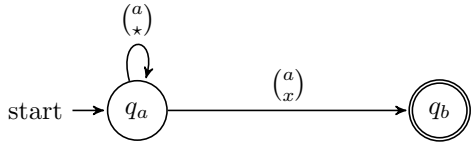
1. The language where the first data value is equal to the last and all other values are different from them (but can be equal among themselves).



2. The language where the first data value is different from all other data values.



3. The language where the last data value differs from all other data values.



Note that the last example is not expressible by register automata [18].

It was shown in [16] that the language $L = \{\binom{a}{d_1}\binom{a}{d_1}\binom{a}{d_2}\binom{a}{d_2}\dots\binom{a}{d_k}\binom{a}{d_k} \mid k \geq 1\}$ is not expressible by VFAs. However, it is straightforward to show that it is expressible by REWBs. Thus, we obtain:

Proposition 6. *VFAs are incomparable in terms of expressive power with register automata and REWBs.*

Query evaluation We now show how to use VFAs as a query language for data graphs and study the complexity of the query evaluation problem.

As before, given a data graph G and a VFA $\mathcal{A}$ we define the relation $\mathcal{A}(G) \subseteq V \times V$ that consists of all pairs (s,t) of nodes in G such that there is a path π between them with the property that $w(\pi) \in L(\mathcal{A})$.

Now we study the following problem.

	Query Evaluation for VFAs
Input:	A data graph G, two nodes $s,t \in V(G)$ and a VFA $\mathcal{A}$.
Task:	Decide whether $(s,t) \in \mathcal{A}(G)$.

Note that this corresponds to the combined complexity of query evaluation; if the automaton $\mathcal{A}$ is fixed, we deal with data complexity.

Theorem 6. – Query Evaluation for VFAs *is* NP*-complete.*
– *For each fixed* $\mathcal{A}$*, the problem* Query Evaluation for VFAs *is solvable in* NLogspace.

As the proof is rather lengthy we present it in the following two sections.

Note that the combined complexity dropped from Pspace to NP, which is viewed as much more acceptable for query evaluation, at least over large databases. This is the complexity of relational conjunctive queries, for instance [2], or conjunctive regular path queries over graphs [10].

Query containment Here we study the query containment problem for variable automata. It is known that the language containment problem for VFAs is undecidable [15]. Hence, the problem of checking, for two VFAs $\mathcal{A}_1, \mathcal{A}_2$, whether $\mathcal{A}_1(G) \subseteq \mathcal{A}_2(G)$ for every data graph G, is undecidable too.

To get a decidable subcase of the query containment problem, we turn to restriction based on *deterministic variable automata – DVFAs.* These are the VFAs with the property that for every word in their language there is only one run accepting it. Note that these are not the same as the ones with underlying deterministic automaton. We can use results from [15] to show

Proposition 7. *The containment problem for queries posted by* deterministic *VFAs is in* coNP.

Although testing if a VFA is deterministic can be done in NLogspace, problem of determinizing VFAs is undecidable. There is however a nice class of determinizable VFAs – the ones with no free variable mentioned in their automaton. It is easy to see that this fragments corresponds to regular expressions with backreferencing – that is grep specifications from the unix systems.

Proof of Theorem 6, data complexity

Here we will use the standard automata product construction used for RPQs and register automata [21].

Assume now that we have fixed a VFA $\mathcal{A}$. We are given G and $s, t \in G$ as input. We can view G as a VFA that uses only constants and with s as initial and t as the final state.

Using Theorem 1 in [15] we build the product of our graph, viewed as a VFA and our fixed VFA $\mathcal{A}$. Theorem 2 in [16] counts the number of states in the product construction as $O(n_1 \cdot n_2 \frac{(d_1+d_2+c_1+c_2)!}{(c_1+c_2)!})$ and the number of transitions as $O(\frac{(d_1+d_2+c_1+c_2)!}{(c_1+c_2)!})$, where n_i is the number of states, d_i the number of bounded variables and c_i the number of constants from $\mathcal{D}$ in each of our automatons.

Note now that since $\mathcal{A}$ is fixed n_2, d_2 and c_2 are constants. Let $M = n_2 + d_2 + c_2$. Also notice that our graph, viewed as an automaton has $d_1 = 0$ and n_1 and c_1 are both bounded by the size of the graph $|G|$. Thus the size of our product automaton is $O(M \cdot |G| \frac{(M+|G|)!}{(c_1+|G|)!}) \leq O(M \cdot |G| \cdot (M + |G|)^M)$, that is polynomial in the size of G and the same calculation applies to the number of transitions.

Using standard on-the-fly technique we check the product automaton for nonemptiness in NLogspace. It is straightforward to see that (s, t) is in the answer to our query $\mathcal{A}$ on G if and only if this product is nonempty. Thus we get the desired upper bound.

Lower bound follows from the same result for RPQs (without data values).

Proof of Theorem 6, combined complexity

Note that the product automaton in the proof of data complexity is exponential in size of the input, so the above algorithm would give us a Pspace upper bound for combined complexity. Next we show that we can do better.

First we prove membership. Assume we are given a graph G, two nodes $s, t \in G$ and a VFA $\mathcal{A}$. We show that if $w \in L(\mathcal{A})$ and w is label of a path in G from s to t, then there is a path in G from s to t, with label w' and of length at most $|G| \cdot |\mathcal{A}| + 1$ such that $w' \in L(\mathcal{A})$, where $|\mathcal{A}|$ denotes the number of states in $\mathcal{A}$.

Assume that $w = w_1 \dots w_l \in L(\mathcal{A})$ is label of a path of length greater than $|G| \cdot |\mathcal{A}| + 1$ as above. Let $v = v_1 \dots v_l$ be a witnessing pattern for w that is accepted by $\mathcal{A}$. Then there is a sequence $q_0, q_1, \dots q_l$ of states of $\mathcal{A}$ such that (q_i, v_{i+1}, q_{i+1}) is a transition in $\mathcal{A}$, with q_l a final state. There is also an assignment of variables in v to values in $\mathcal{D}$ that witness w (as in the definition of a witnessing sequence).

By the assumption there is a path $n_0, \dots, n_l$ of nodes in G with the label w and such that $n_0 = s$ and $n_l = t$.

By the pigeon hole principle there exists $i, j \leq n$ such that $n_i = n_j$ and $q_i = q_j$. Observe that $n_0, \dots, n_i, n_{j+1}, \dots, n_l$ is still a path in G from s to t with

the label $w' = w_1 \dots w_i w_{j+1} \dots w_l$ and that $q_0 \dots q_i q_{j+1} \dots q_l$ is an accepting run on $v' = v_1 \dots v_i v_{j+1} \dots v_l$. Also note that v' is a witnessing pattern for w', as witnessed by the same assignment of data values to variables in v' as it was in v.

By repeating this cutting procedure we get the desired result. Now for the NP-algorithm we simply guess a path of length at most $|G| \cdot |\mathcal{A}| + 1$–a polynomial in the size of the input and verify that it belongs to our language in PTIME.

To show NP-hardness we do a reduction from k-CLIQUE. This problem asks, given a graph G and a number k, to determine if G has a clique of size at least k.

Suppose we are given an undirected graph G and a number k. We will construct a graph G' with $|G| + 2$ nodes , select two nodes $s, t \in G'$ and construct a VFA $\mathcal{A}$ of size $O(k^2)$ such that G contains a k-clique if and only if there is a path from s to t in G' whose label belongs to $L(\mathcal{A})$.

Take $\Sigma = \{a, b\}$ and make G directed by adding edges in both directions for every edge in G. Assume that every vertex v is given an unique data value d_v. Label the edges $(v, v') \in G$ by $\binom{a}{d_{v'}}$ and add two more nodes s and t. Add an edge from s to every other node v except s, t and label them with $\binom{b}{d_v}$. Also add an edge from every node in G to t and label them by $\binom{b}{d_t}$, with d_t a new unique data value. We call the resulting graph G'. (The idea is that every node has a unique data value – its id.)

We define our VFA as a linear path with transitions:

- $(q_0, \binom{b}{x_1}, q_1), (q_1, \binom{a}{x_2}, q_2)$ (this collect the first two nodes in the clique),
- $(q_{i-1}, \binom{a}{x_i}, q'_i), (q'_i, \binom{a}{x_1}, q_1^i), (q_1^i, \binom{a}{x_i}, p_i^1), (p_i^1, \binom{a}{x_2}, q_2^i), (q_2^i, \binom{a}{x_i}, p_i^2), \dots,$ $(p_i^{i-2}, \binom{a}{x_{i-1}}, q_{i-1}^i), (q_{i-1}^i, \binom{a}{x_i}, q_i)$, for $3 \leq i \leq k$ and
- $(q_k, \binom{b}{d_t}, q_{k+1})$ (to get the target node).

Note that here we add a new state for each transition of the automaton.

Next we show that there is a k-clique in G iff there is a data path form s to t in G' whose label belongs to $L(\mathcal{A})$.

Suppose first that there is a k-clique in G. Then we simply move from s to arbitrary point in that clique using the b Labelled edge and traverse the clique back and forth until we reach the k-th element of the clique. Note that starting from the third element, whenever we select a different node in the clique we have to move back and forth between this node and all previously selected ones to match the transitions (we check that they are interconnected), but since we have a clique this is possible. Finally, after selecting the last node and verifying that it is connected to all the others we move to t using a b Labelled edge.

Now suppose that there is a path from s to t in G' whose label belongs to $L(\mathcal{A})$. This means that we will be able to select k different nodes $n_1, \dots, n_k$ in G with data values stored in $x_1, \dots, x_k$. Since all data values in the graph are different they also act as ids. Now take any two n_l, n_m with $l < m \leq k$. Then we

know that n_l and n_m are connected in G because after selecting n_m we have to go through the transitions stating $(p_m^{l-1}, \binom{a}{x_l}, q_l^m), (q_l^m, \binom{a}{x_m}, p_m^l)$ and similarly for when l, m are at the beginning or end of the transition chain. Since no two data values in G are the same this means that we have an edge between n_l and n_m. This completes the proof.

9 Putting the formalisms together

We know that variable automata are incomparable in expressive power with register automata and regular expressions with binding. In particular we showed that they can express a property that all data values differ from the last. On the other hand, bound variables in variable automata behave like a limited version of registers that are capable of storing a data value only once. As the result, variable automata are not able to express some simple properties definable by REWBs.

In this section we define a general model that will encompass both register and variable automata and study its query evaluation problem over graphs. The model is essentially a variable automaton that can use the full power of registers in a same way that an ordinary register automaton would. It will subsume both models, but we shall see that it does not increase the complexity of query evaluation beyond that of only register automata.

Definition 3. *Let Σ be a finite alphabet, k a natural number and C a finite set of data values. A* k-register automaton with variables *(or varRA for short) is a tuple $\mathcal{A} = (Q, q_0, F, T, \{\star\}, \Sigma, C)$, where:*

- *Q is a finite set of states;*
- *$q_0 \in Q$ is the initial state;*
- *$F \subseteq Q$ is the set of final states;*
- *$T \subseteq Q \times \Sigma \times (\mathcal{C}_k \cup C \cup \{\star\}) \times 2^{\{1,\ldots,k\}} \times Q$ is a finite set of transitions, which could be of the following form:*
  - *(q, a, c, I, q') with $c \in \mathcal{C}_k$, or*
  - *$(q, a, d, \emptyset, q')$ with $d \in C$, or*
  - *$(q, a, \star, \emptyset, q')$.*

We now define the notion of acceptance. We refer to transitions $(q, a, \star, \emptyset, q')$ as $\star$-transitions. A k-register automaton $\mathcal{A}$ with variables accepts a data word $w = w_1 \cdots w_n$ if there is a sequence $q_0, \ldots q_n$ of states in Q with $q_n \in F$, a sequence $t_1, \ldots t_n$ such that t_i is a transition from q_{i-1} to q_i, and a sequence $\tau_0, \ldots \tau_n$ of register assignments such that for each $i \in \{1, \ldots, n\}$, we have:

- If $t_i = (q_{i-1}, a, c, I, q_i)$ then: $\tau_i, \delta(w_i) \models c$, $\lambda(w_i) = a$, and τ_{i+1} is obtained from τ_i by putting $\delta(w_i)$ in registers from I;
- If $t_i = (q_{i-1}, a, d, \emptyset, q_i)$, then $\lambda(w_i) = a$, $\delta(w_i) = d$ and $\tau_{i+1} = \tau_i$;
- If $t_i = (q_{i-1}, a, \star, \emptyset, q_i)$ then $\delta(w_i) = \delta(w_j)$ iff t_j is a $\star$-transition.

Notice that register automata with variables extend both register and variable automata in a natural way. Moreover, if we restrict the registers by allowing them to store values only once and restrict conditions to single equality tests only, we get variable automata. On the other hand if we disallow the usage of the free variable $\star$ we get register automata (note here that constants can be easily simulated by additional registers).

Next we study complexity of the query evaluation problem for our automata. Given such an automaton $\mathcal{A}$, and a data graph G, we write $(s,t) \in \mathcal{A}(G)$ if there is a path π from s to t such that $w(\pi)$ is accepted by $\mathcal{A}$.

Query Evaluation for varRAs	
Input:	A data graph G, two nodes $s,t \in V(G)$ and a varRA $\mathcal{A}$.
Task:	Decide whether $(s,t) \in \mathcal{A}(G)$.

Surprisingly, despite the increased expressive power, this model still retains the complexity of register automata.

Theorem 7. – Query Evaluation for varRAs *is* Pspace*-complete.*
– *For each fixed $\mathcal{A}$, the problem is in* NLogspace*.*

To prove this we use a similar construction to the one in [22]. That is for a finite set of data values D and a k-register automaton with variables $\mathcal{A}$ we produce a variable automaton $\mathcal{A}_D$ that accepts precisely the same words as $\mathcal{A}$ does when both use only data values from D.

Let $\mathcal{A} = (Q, q_0, F, T, \{\star\}, \Sigma, C)$ be a k-register automaton with variables and D a finite set of data values.

Next we define our VFA $\mathcal{A}_D = (\Gamma, A)$ with $\Gamma = \{C \cup D\} \cup X \cup \{\star\}$. The NFA $A = (Q', q_0', F', T')$ over $\Sigma \times \Gamma$ is defined as follows:

– $Q' = Q \times D_\perp^k$, where $\perp$ is a new data value not in D and $D_\perp = D \cup \{\perp\}$
– $q_0' = (q_0, \perp^k)$
– $F' = F \times D_\perp^k$
– For the transitions:
  • If $(q, a, c, I, q') \in T$ we add

$$((q,\tau), \binom{a}{d}, (q',\tau'))$$

  to T' iff $\tau, d \models c$ and $\tau' = \tau[I \leftarrow d]$
  • If $(q, a, d, \emptyset, q') \in T$, with d a constant in C we add

$$((q,\tau), \binom{a}{d}, (q',\tau'))$$

  to T'

- If $(q, a, \star, \emptyset, q') \in T$ we add

$$((q, \tau), \binom{a}{\star}, (q', \tau'))$$

to T'

Next we prove that the variable automaton obtained in this construction indeed accepts the same class of data words over D as the original register automaton with variables does.

Claim. Let w be a data word whose data values come from D. Then $w \in L(\mathcal{A}_D)$ if and only if $w \in L(\mathcal{A})$.

Proof. Assume first that $w = \binom{a_1}{d_1} \cdots \binom{a_n}{d_n}$, where $d_1, \ldots d_n$ are from D is accepted by $\mathcal{A}_D$.

Since $\mathcal{A}_D$ is a VFA with constants and free variable only (and no bound variables), this means that there is a word $v = v_1 \cdots v_n \in (\Sigma \times \Gamma)^*$, accepted by the underlying NFA A, such that for $1 \leq i \leq n$ it holds:

- $\lambda(v_i) = \lambda(w_i)$ (finite labels match)
- $\delta(v_i) = \delta(w_i)$, for $v_i = d$, a constant of $\mathcal{A}_D$ (constants match)
- $\delta(v_i) = \star$ and $\delta(v_j) \neq \star$ implies that $\delta(w_i) \neq \delta(w_j)$ (free variable condition is true).

This in turn means that there is a sequence $(q_0, \tau_0), \ldots (q_n, \tau_n)$ of states in $\mathcal{A}_D$ and appropriate transitions that accept v as the witnessing pattern of w. But this same sequence of states and transitions of $\mathcal{A}_D$ can be easily transformed into an accepting run of $\mathcal{A}$ on w (follows from the construction of $\mathcal{A}_D$), thus implying that $w \in L(\mathcal{A})$.

To see that the reverse is true we simply transform the accepting run of $\mathcal{A}$ on w into the matching run of $\mathcal{A}_D$. The witnessing pattern for w will be obtained by converting every data value matched with $\star$ in w by $\star$ itself. All the details easily follow from the definition of acceptance and the construction of $\mathcal{A}_D$.

To complete the proof of Theorem 7 we use the same technique as in the proof of Theorem 6.

That is we are given our k-varRA $\mathcal{A}$, a data graph G and s, t in G. Let $D = \mathcal{D}(G)$ be the set of all data values appearing in G. Note that $|D| \leq |G|^2 \times |\Sigma|$, since each edge can have a distinct data value.

We again view our graph as a VFA (with the initial state s and final state t) and denote it by $\mathcal{A}_G$. We next build a product of $\mathcal{A}_G$ and $\mathcal{A}_D$. Testing his automaton for nonemptiness is the same as answering our query evaluation problem.

Note that the number n_1 of states of $\mathcal{A}_D$ is $|\mathcal{A}| \times |D|^k$, the number of bound variables $d_1 = 0$ and the number of constants c_1 at most $|D| + |\mathcal{A}|$.

For $\mathcal{A}_G$ we have $n_2 = |G|$, while $d_2 = 0$ and $c_2 = |D|$.

By the construction in [16] we know that the size of the product is $O(n_1 \cdot n_2 \cdot \frac{(c_1+c_2+d_1+d_2)!}{(c_1+c_2)!}) = O(n_1 \cdot n_2)$.

Using the values above we get that the size is $O(|\mathcal{A}| \times |D|^k \times |G|)$.

Note that this is polynomial in $|G|$ if the automaton is fixed and exponential if it is part of the input (as the number of registers gets into the exponent). Thus using the standard on-the-fly method for testing nonemptiness we obtain the desired result.