THE UNIVERSITY of EDINBURGH

Edinburgh Research Explorer

Regular path queries on graphs with data

Citation for published version:
Libkin, L & Vrgoc, D 2012, Regular path queries on graphs with data. in 15th International Conference on Database Theory, ICDT '12. ACM, pp. 74-85. DOI: 10.1145/2274576.2274585

Digital Object Identifier (DOI):
10.1145/2274576.2274585

Link:
Link to publication record in Edinburgh Research Explorer

Document Version:
Early version, also known as pre-print

Published In:
15th International Conference on Database Theory, ICDT '12

OPEN ACCESS

Regular Path Queries on Graphs with Data*

Leonid Libkin   Domagoj Vrgoč

ABSTRACT

Graph data models received much attention lately due to applications in social networks, semantic web, biological databases and other areas. Typical query languages for graph databases retrieve their topology, while actual data stored in them is usually queried using standard relational mechanisms.

Our goal is to develop techniques that combine these two modes of querying, and give us query languages that can ask questions about both data and topology. As the basic querying mechanism we consider regular path queries, with the key difference that conditions on paths between nodes now talk not only about labels but also specify how data changes along the path. Paths that combine edge labels with data values are closely related to data words, so for stating conditions in queries, we look at several data-word formalisms developed recently. We show that many of them immediately lead to intractable data complexity for graph queries, with the notable exception of register automata, which can specify many properties of interest, and have NLOGSPACE data and PSPACE combined complexity. As register automata themselves are not easy to use in querying, we define two types of extensions of regular expressions that are more user-friendly, and develop query evaluation techniques for them. For one class, regular expressions with memory, we achieve the same bounds as for automata, and for the other class, regular expressions with equality, we also obtain tractable combined complexity of query evaluation. In addition, we show that results extends to analogs of conjunctive regular path queries.

*Authors' address: School of Informatics, University of Edinburgh, email: {libkin,domagoj.vrgoc}@inf.ed.ac.uk.

1. Introduction

Querying graph-structured data has been actively studied in recent years, due to numerous applications in areas including biological networks [31, 32, 36], social networks [38, 39], and the semantic Web [27, 37]. Such databases are represented as graphs in which nodes are objects, and edge labels specify relationships between them [1, 3]. Typical queries over such databases look for reachability patterns. A very common and well studied class of queries is that of *regular path queries*, or RPQs. An RPQ selects nodes connected by a path that belongs to a regular language over the labeling alphabet [13, 14, 15]. Their extensions have been studied extensively too; for example, conjunctive RPQs state the existence of several paths [12, 18, 22], and extended conjunctive RPQs add comparisons of paths [4].

These standard queries over graph databases talk about their topology, and do not mention data values. But graph databases do contain data. For example, in a social network, one would expect each node to correspond to a person, with his/her attributes such as name, age, city, email, etc.; labels can specify types of connections between people, e.g., like/dislike, professional, etc. The querying mechanisms one deals with are generally of one of these categories:

- queries about topology such as finding nodes connected by a path with a certain label (e.g., people who are connected via professional links), or

- queries about data, i.e., essentially relational queries (e.g., finding pairs of people of the same age).

What these languages are incapable of doing is *combining* data and topology. As an example of a query that involves such a combination, consider a query looking for people who are connected via professional links and are of the same age. This query states the existence of a path with a certain property and then relates data values at the end of the path. Another example is a query that finds people who are connected via professional links restricted to people of the same age. In this case, comparison of data values (having the same age) is done for every node along the path.

Extending languages that handle structure to languages that handle both structure and data is not new in database theory. For very simple types of paths it was considered in

graph object-oriented models [42], but most notably it happened in the study of XML [8, 40, 41]. For example, languages such as XPath exist in their structural variants as well as extensions that handle data comparisons [6, 9, 20, 34]. A standard abstraction one uses for extending from structure to data in the case of XML is *data trees*, in which data values are attached to tree nodes [9, 29, 40]. The focus of the study of such extensions has been both on querying, where one is concerned with efficient evaluation [7, 24], and on reasoning, where one is concerned with the decidability of the satisfiability problem [9, 10].

So likewise, we consider graph databases where nodes can carry data values. An example of such a graph database is shown in Fig. 1. It has five nodes, $v_1, \ldots, v_5$; data values are shown inside the nodes, and edge labels next to the edges. As an initial assumption, we assume that each node carries just one data value. This is not a real restriction for two reasons. First, if a node has a tuple of data values (e.g., person's name, age, email, etc., in a social network) this could be modeled by extra edges to nodes with those data values. And second, the way we design languages for querying graph databases with data values will make it very easy to extend them to such a setting.

An RPQ may ask for pairs of nodes connected by a path from the regular language $(ab)^*$. In the graph in Fig. 1, one possible answer is (v_1, v_3), another – (v_1, v_5). To combine this with data values, we may ask queries of the following kind:

- Find nodes connected by a path from $(ab)^*$ such that the data values at the beginning and at the end of the path are the same. In this case, (v_1, v_3) is still in the answer but (v_1, v_5) is not.
- We may extend comparisons to other nodes on the path, not only to the first and last nodes. For example, we may ask for nodes connected by paths along which the data value remains the same, or on which all data values are different from the first one. The pair (v_1, v_3) is in the answer to the first query (the path $v_1 v_4 v_3$ witnesses it), while the pair (v_1, v_5) is in the answer to the second, as witnessed by the path $v_1 v_2 v_5$.

What kind of languages can we use in place of regular languages to specify paths with data? To answer this, consider, for example, a path $v_1 v_2 v_5 v_3$ in the graph. If we traverse it by starting in v_1, reading its data value, then reading the label of (v_1, v_2), then the data value in v_2, etc., we end up with the following sequence: $1a2b3a1$. We shall refer to them as *data paths*. They are extremely close to an object that has been actively studied in the XML context – namely, *data words* [8, 10, 40, 41]. A data word is a word in which every position is labeled by both a letter from a finite alphabet (e.g., a or b) and a data value (e.g., a number). Data paths are essentially data words with an extra data value. We can represent the data path $1a2b3a1$ as a data word $\binom{\#}{1}\binom{a}{2}\binom{b}{3}\binom{a}{1}$, where $\#$ is a special symbol reserved for the extra data value.

We can thus use multiple formalisms developed for data words (with a minor adjustment for the extra value) to spec-
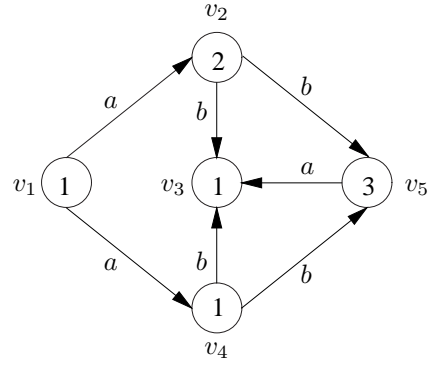

Figure 1: A graph database with data values

ify data paths. Such formalisms abound in the literature, and include first-order and monadic second-order logic with data comparisons [9, 10], LTL with freeze quantifiers [16], XPath fragments [8, 20], and various automata models such as pebble and register automata [11, 28, 29, 30, 33].

The question is then, which one to choose? To answer this, we look at data complexity of query answering for each of these formalisms. We show that as long as the formalism is capable of expressing what is perhaps the most primitive language with data value comparisons (two data values are equal) and is closed under complementation, then *data* complexity is NP-hard. Clearly one cannot tolerate such high data complexity, and this rules out most of the formalisms except *register automata*.

We then study query answering with register automata (adjusted for data paths from data words). We present an algorithm that is based, as expected, on computing products of automata; with nonemptiness performed on-the-fly, this gives us an NLOGSPACE data complexity bound, and PSPACE-completeness for combined complexity. The bound for data complexity is good (it matches the usual RPQs) and the bound for combined complexity is tolerable (equivalent to that of FO, but higher than the NP bound for conjunctive RPQs or the PTIME bound for RPQs).

However, automata are not an ideal way of specifying conditions in queries. In RPQs, we use regular expressions rather than NFAs. While some regular expressions have been considered for register automata [30], they are very far from intuitive. So we propose two types of regular expressions that can be used in queries.

The first, close in spirit to automata themselves, lets one bind a data value and use it later. For example, to express the query "connected by a path along which the data value remains the same", we would use the expression $\downarrow x.(\Sigma[x^=])^*$. This expression says: bind x in the beginning of the path (i.e., to the first data value), and then go along, if labels are arbitrary (Σ) and the condition $x^=$, meaning that the value is equal to x, holds. These expressions are much easier to write than the automata, and at the same time they can be translated into register automata; thus data complexity of queries remains in NLOGSPACE. We show that the com-

bined complexity remains the same as for automata, i.e., PSPACE-complete (except in a rather limited case when the Kleene star is not used: then it drops to NP-complete).

This motivates a second class of expressions that restrict the ability to compare data values along the path; instead, one can only do comparisons for chosen subexpressions. A simple example of such an expression is $\Sigma_{=}^{+}$, which denotes nonempty data paths that have same data value at the beginning and at the end of the path: Σ^{+} indicates the label of the path, and the subscript $=$ states the condition for the first and the last data values. A slightly more elaborate example is $\Sigma^{*} \cdot \Sigma_{=}^{+} \cdot \Sigma^{*}$. It says that a subpath conforms to $\Sigma_{=}^{+}$, i.e., it denotes data paths on which two data values are equal. For expressions of this kind, we give a polynomial-time algorithm for combined complexity. The key idea is to translate expressions into push-down automata and then take the product with an automaton obtained efficiently from the graph database.

Finally, we show that our results extend to analogs of conjunctive regular path queries that use data comparisons. There is no penalty to pay in terms of complexity except one case, where we have to deal with the same increase of complexity as in going from the usual RPQs to their conjunctive analogs [12, 14].

Organization In Section 2 we define data graphs and generic queries over them. In Section 3 we rule out several formalisms for specifying data paths due to prohibitively high data complexity for them. In Section 4 we define register automata and study complexity of query evaluation for them. We do the same in Section 5 for regular expressions with memory and in Section 6 for regular expressions with equality. Finally in Section 7 we look at conjunctive queries based on the formalisms proposed in the previous sections. Due to space limitations, most proofs are only sketched, and complete proofs are given in the appendix.

2. Preliminaries

Let Σ be a finite alphabet, and $\mathcal{D}$ a countably infinite set of data values. Data graphs will have edges labeled by letters from Σ and nodes that store data values from $\mathcal{D}$.

Definition 2.1 (Data graphs). *A* data graph *(over Σ and $\mathcal{D}$) is a triple $G = \langle V, E, \rho \rangle$, where:*

- *V is a finite set of nodes;*
- *$E \subseteq V \times \Sigma \times V$ is a set of labeled edges; and*
- *$\rho : V \to \mathcal{D}$ is a function that assigns a data value to each node in V.*

A path between nodes v_1 and v_n in a graph is a sequence

$$\pi = v_1 a_1 v_2 a_2 v_3 \ldots v_{n-1} a_{n-1} v_n \qquad (1)$$

such that each (v_i, a_i, v_{i+1}), for $i < n$, is an edge in E. Corresponding to the path π (1) we have a *data path*

$$w_\pi = \rho(v_1) a_1 \rho(v_2) a_2 \rho(v_3) \ldots \rho(v_{n-1}) a_{n-1} \rho(v_n) \quad (2)$$

which is a sequence of alternating data values and labels, starting and ending with data values. The set of all *data paths*, i.e., such alternating sequences over Σ and $\mathcal{D}$, will be denoted by $\Sigma[\mathcal{D}]^{*}$. For both paths and data paths, we use the notation $\lambda(\pi)$ or $\lambda(w_\pi)$ to denote their label, i.e. the word $a_1 \ldots a_{n-1} \in \Sigma^{*}$.

Returning to Figure 1 from the Introduction, one example that we used was the path $\pi = v_1 a v_2 b v_5 a v_3$. The corresponding data path w_π is $1a2b3a1$ since data values of v_1, v_2, v_5, and v_3 are $1, 2, 3$, and 1, respectively. Its label is aba.

Recall that *regular path queries*, or *RPQs*, over usual labeled graphs are queries of the form $Q = x \xrightarrow{L} y$, where $L \subseteq \Sigma^{*}$ is a regular language. Given a graph G (the data part is irrelevant for RPQs), $Q(G)$ is the set of pairs of nodes (v, v') such that there is a path π from v to v' whose label $\lambda(\pi)$ is in L.

By analogy, we define *data path queries*. Syntactically they are expressions $Q = x \xrightarrow{L} y$, as before, but now $L \subseteq \Sigma[\mathcal{D}]^{*}$ is a set of data paths. If G is a data graph, then $Q(G)$ is the set of pairs of nodes (v, v') such that there is a path π from v to v' whose associated data path w_π is in L.

As with relational queries and RPQs, we will be interested in *data and combined complexity* of query evaluation problem, i.e. checking, for a data path query Q, a data graph G and a pair of nodes (v, v'), whether $(v, v') \in Q(G)$ (for data complexity, of course, the query Q will be fixed).

3. Language for paths: ruling out bad alternatives

To talk about data path queries, as just defined, we need to express properties of paths with data. As we already mentioned, these are essentially data words, with an extra data value attached. Quite a few languages and automata models have been developed for data words over the past few years, mainly in connection with the study of XML, especially XPath. We now give a quick overview of them. A more extensive survey can be found in [40].

FO($\sim$) **and** MSO($\sim$) These are first-order logic and monadic second-order logic extended with the binary predicate $\sim$ saying that data values in two positions are the same. For example, $\exists x \exists y\, a(x) \wedge a(y) \wedge x \sim y$ says that there are two a-labeled positions with the same data value. Two-variable fragments of FO($\sim$) and existential MSO with the $\sim$ predicate have been shown to have decidable satisfiability problem [9, 10].

Pebble automata These are basically finite state automata equipped with a finite set of pebbles. To ensure regular behavior pebbles are required to adhere to a stack discipline. The automata are modeled in such a way that the last placed pebble acts as the automaton head and we are allowed to drop and lift pebbles over the current position. In addition to this we can also compare the

current data value to the one that already has a pebble placed over it. Algorithmic properties and connections with logics have been extensively studied in [33].

LTL$_\downarrow$ This the is standard LTL expanded with a freeze operator that allows us to store the current data value into a memory location and use it for future comparisons. The full logic has undecidable satisfiability problem, but various decidable restrictions are known [16, 17].

Register automata These are in essence finite state automata extended with a finite set of registers allowing us to store data values. Although first studied only on words over infinite alphabet [28, 33, 35] they are easily extended to handle data words, as illustrated in [16, 40]. They act as usual finite state automata in the sense that they move from one position to another by reading the appropriate letter from the finite alphabet, but are also allowed to compare the current data value with ones already stored in the registers.

XPath fragments XPath is the standard language for navigating in XML documents, i.e., for describing paths in a way that may also include conditions on data values that occur in documents. Fragments of XPath (with and without data values) have been extensively studied, see, e.g., [6, 9]. While in general the satisfiability problem is undecidable, several decidable restrictions are known, e.g., [20, 21].

In deciding which formalism to choose, we look at the *data complexity* of evaluating data path queries, and try to rule out those for which data complexity is intractable. Technically, a formalism just defines a set of allowed languages $L \subseteq \Sigma[\mathcal{D}]^*$. It turns out that most of the formalisms for data words/paths are actually not suitable for graph querying. This is implied by the following result. Let L_{eq} be the language of data paths that contain two equal data values.

Theorem 3.1. *Assume that we have a formalism for data paths that can define $\overline{L_{eq}}$. Then data complexity of evaluating data path queries is NP-hard.*

The proof is by showing that with $\overline{L_{eq}}$, one can encode the 2-disjoint-paths problem which is NP-complete [23].

Note that L_{eq} is about the simplest property one can express about data paths/words; it would be hard to imagine a formalism that cannot check for the equality of data values. The corollary below effectively rules out closure under complement for such formalisms if they are to be used in graph querying.

Corollary 3.2. *Assume that we have a formalism for data paths that can define L_{eq} and that is closed under complement. Then data complexity of evaluating data path queries is NP-hard.*

This immediately rules out FO($\sim$) and its two-variable fragment, LTL with the freeze quantifier, XPath fragments closed under complement, and pebble automata.

The only hope we have among standard formalisms is *register automata*, since they are not closed under complementation [28]. In the next sections we show that we can achieve good query answering complexity with them, as well as sufficient expressivity.

4. Data path queries with register automata

As stated in the previous section, register automata are the only standard formalism for defining classes of data words that does not immediately lead to NP-hard data complexity of queries on graphs with data. In this section we define them and study query evaluation for data path queries based on these automata. We will slightly alter the definition of register automata used in e.g. [16, 40] to work on data paths rather than data words, without affecting their desirable properties.

As mentioned earlier register automata move from one state to another by reading the appropriate letter from the finite alphabet and comparing the data value to one previously stored into the registers. Our version of register automata will use slightly more involved comparisons which will be boolean combinations of atomic $=, \neq$ comparisons of data values.

To define such conditions formally, assume that, for each $k > 0$, we have variables $x_1, \ldots, x_k$. Then conditions in $\mathcal{C}_k$ are given by the grammar:

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

The satisfaction is defined with respect to a data value $d \in \mathcal{D}$ and a tuple $\tau = (d_1, \ldots, d_k) \in \mathcal{D}^k$ as follows:

- $d, \tau \models x_i^=$ iff $d = d_i$;
- $d, \tau \models x_i^{\neq}$ iff $d \neq d_i$;
- $d, \tau \models c_1 \wedge c_2$ iff $d, \tau \models c_1$ and $d, \tau \models c_2$ (and likewise for $c_1 \vee c_2$);
- $d, \tau \models \neg c$ iff $d, \tau \not\models c$.

In what follows, $[k]$ is a shorthand for $\{1, \ldots, k\}$.

Definition 4.1 (Register data path automata). *Let Σ be a finite alphabet, and k a natural number. A k-register data path automaton is a tuple $\mathcal{A} = (Q, q_0, F, \tau_0, \delta)$, where:*

- *$Q = Q_w \cup Q_d$, where Q_w and Q_d are two finite disjoint sets of word states and data states;*
- *$q_0 \in Q_d$ is the initial state;*
- *$F \subseteq Q_w$ is the set of final states;*
- *$\tau_0 \in \mathcal{D}^k$ is the initial configuration of the registers;*
- *$\delta = (\delta_w, \delta_d)$ is a pair of transition relations:*
  - *$\delta_w \subseteq Q_w \times \Sigma \times Q_d$ is the word transition relation;*
  - *$\delta_d \subseteq Q_d \times \mathcal{C}_k \times 2^{[k]} \times Q_w$ is the data transition relation.*

The intuition behind this definition is that since we alternate between data values and word symbols in data paths, we also alternate between data states (which expect data value as the next symbol) and word states (which expect alphabet letters as the next symbol). We start with a data value, so q_0 is a data state, end with a data value, so final states, seen after reading that value, are word states.

In a word state the automaton behaves like the usual NFA (but moves to a data state). In a data state, the automaton checks if the current data value and the configuration of the registers satisfy a condition, and if they do, moves to a word state and updates some of the registers with the read data value.

Given a data path $w = d_0 a_0 d_1 a_1 \ldots a_{n-1} d_n$, where each d_i is a data value and each a_l is a letter, a configuration of $\mathcal{A}$ on w is a tuple (j, q, τ), where j is the current position of the symbol in w that $\mathcal{A}$ reads, q is the current state and $\tau \in \mathcal{D}^k$ is the current state of the registers. The initial configuration is $(0, q_0, \tau_0)$ and any configuration (j, q, τ) with $q \in F$ is a final configuration.

From a configuration $C = (j, q, \tau)$ we can move to a configuration $C' = (j + 1, q', \tau')$ if one of the following holds:

- the jth symbol is a letter a, there is a transition $(q, a, q') \in \delta_w$, and $\tau' = \tau$; or
- the current symbol is a data value d, and there is a transition $(q, c, I, q') \in \delta_d$ such that $d, \tau \models c$ and τ' coincides with τ except that the ith component of τ' is set to d whenever $i \in I$.

A data path w is accepted by $\mathcal{A}$ if $\mathcal{A}$ can move from the initial configuration to a final configuration after reading w. The language of data paths accepted by $\mathcal{A}$ is denoted by $L(\mathcal{A})$.

Data paths vs data words

Register automata have been previously studied for data words [16, 40] and we now briefly explain the connection. A data word is a word in $(\Sigma \times \mathcal{D})^*$, i.e., each position carries a label from Σ and a data value from $\mathcal{D}$. A k-register data word automaton $\mathcal{A}$ is a tuple (Q, q_0, F, τ_0, T) where Q is a finite set of states (no longer split into two), $q_0 \in Q$ is the initial state, $F \subseteq Q$ is the set of final states, $\tau_0 \in \mathcal{D}^k$ is the initial register assignment, and T is a finite set of transitions of the form $(q, a, c) \rightarrow (I, q')$, where q, q' are states, a is a label, $I \subseteq [k]$, and c is a condition in $\mathcal{C}_k$.

The automaton traverses a data word from left to right, starting in q_0 with τ_0 as the register configuration. If it reads $\binom{a}{d}$ in state q with register configuration τ, it may apply a transition $(q, a, c) \rightarrow (I, q')$ if $d, \tau \models c$; it then enters state q' and changes contents of registers i, with $i \in I$, to d.

The relationship between automata models, as needed for our purposes, is described by the lemma below. With each data path $w = d_1 a_1 \ldots a_{n-1} d_n \in \Sigma[\mathcal{D}]^*$ we associate a data word $s_w = \binom{\#}{d_1} \binom{a_1}{d_2} \ldots \binom{a_{n-1}}{d_n}$ over $(\Sigma \cup \{\#\}) \times \mathcal{D}$, where

$\# \notin \Sigma$ is a new alphabet symbol.

Lemma 4.2. *Given a k-register data path automaton $\mathcal{A}$, one can construct, in* DLOGSPACE*, a k-register data word automaton $\mathcal{A}'$ such that a data path w is in $L(\mathcal{A})$ iff the data word s_w is in $L(\mathcal{A}')$.*

It is known [16] that nonemptiness problem for data word register automata is PSPACE-complete. The above lemma shows that the PSPACE upper bound applies to data path automata. Moreover, one can verify that the PSPACE-hardness reduction applies to such automata as well. Hence, we have

Corollary 4.3. *The nonemptiness problem for register data path automata is* PSPACE*-complete.*

4.1 Regular data path queries

Our basic class of regular path queries on graphs with data is based on register data path automata.

Definition 4.4. *A regular data path query (RDPQ) is an expression $Q = x \xrightarrow{\mathcal{A}} y$ where $\mathcal{A}$ is a register data path automaton.*
Given a data graph G, the result of the query $Q(G)$ consists of pairs of nodes (v, v') such that there is a data path w from v to v' that belongs to $L(\mathcal{A})$.

To evaluate RDPQs, we transform both a data graph G and a k-register data path automaton $\mathcal{A}$ into NFAs over an extended alphabet and reduce query evaluation to NFA nonemptiness. More precisely, to evaluate $Q(G)$, we do the following:

1. Let D be the set of all data values in G.
2. Transform $G = \langle V, E, \rho \rangle$ into a graph $G' = \langle V', E' \rangle$ over the alphabet $\Sigma \cup D$ as follows:
   - $V' = \{v_s, v_t \mid v \in V\}$
   - $E' = \{(v_t, a, v'_s) \mid (v, a, v') \in E\}$ $\bigcup \{(v_s, \rho(v), v_t) \mid v \in V\}$

   Basically, we split each node v with a data value d into a source node v_s and a target node v_t and add a d-labeled edge between them; after that we restore the edges from E so that they go from target to source nodes. This is illustrated below.
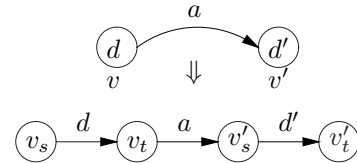
3. Transform the automaton $\mathcal{A} = (Q, q_0, F, \tau_0, (\delta_w, \delta_d))$ into an NFA $\mathcal{A}_D = (Q', q'_0, F', \delta')$ as follows:
   - $Q' = Q \times D^k$;
   - $q'_0 = (q_0, \tau_0)$;
   - $F' = F \times D^k$;

- δ' includes two types of transitions.
  - (a) Whenever we have a transition (q, a, q') in δ_w, we add transitions $((q, \tau), a, (q', \tau))$ to δ' for all $\tau \in D^k$.
  - (b) Whenever we have a transition (q, c, I, q') in δ_d, we add transitions $((q, \tau), d, (q', \tau'))$ if $d, \tau \models c$ and τ' is obtained from τ by putting d in positions from the set I.

For two nodes v, v' of G, we turn G' into an NFA $\mathcal{A}_{G', v, v'}$ by letting v_s be its initial state and v'_t be its final state. Then we have the following.

Proposition 4.5. *Let $Q = x \xrightarrow{\mathcal{A}} y$ be an RDPQ, and G a data graph whose data values form a set $D \subseteq \mathcal{D}$. Then*

$$(v, v') \in Q(G) \iff L(\mathcal{A}_{G', v, v'} \times \mathcal{A}_D) \neq \emptyset.$$

Thus, query evaluation, like in the case of the usual RPQs, is reduced to automata nonemptiness, although this time the automata are over larger alphabets. Since the construction is polynomial in the size of G and exponential in the size of $\mathcal{A}$ (as k gets into the exponent), we immediately get a PTIME upper bound for data complexity and an EXPTIME upper bound for combined complexity. By performing on-the-fly nonemptiness checking for the product, we can lower these bounds.

Theorem 4.6. *Data complexity of RDPQs over data graphs is in* NLOGSPACE*, and the combined complexity of RDPQs over data graphs is* PSPACE*-complete.*

The bound for data complexity cannot be lowered as there exist simple RPQs for which data complexity is NLOGSPACE-complete.

5. Queries based on regular expressions with memory

Regular data path queries based on register automata have acceptable complexity bounds: data complexity is the same as for RPQs, and combined complexity, although exceeding the bounds on conjunctive queries and RPQs, is the same as for relational calculus or for RPQs extended with regular relations. Despite this, RDPQs as we defined them have no chance to lead to a practical language as it is inconceivable that the user will specify a register automaton over data words. Even for queries such as RPQs and their extensions, conditions are normally specified via regular expressions.

Our goal now is to introduce regular expressions that can be used in place of register automata in data path queries. Note that as long as they express languages accepted by register automata, we shall achieve an NLOGSPACE bound on data complexity by Theorem 4.6.

The first class of queries, studied in this section, is based on an extension of regular expressions with *memory* that lets us specify when data values are remembered and when they are used. The basic idea is this: we can write expressions like $\downarrow x.a^+[x^=]$ saying: store the current data value in x and check that after reading a word from a^+ we see the same data value (condition $x^=$ is true). This will define data words of the form $da \dots ad$. Such expressions are relatively easy to write and understand (much easier than automata), and the complexity of their query evaluation will not exceed that of register automata.

Definition 5.1 (Expressions with memory). *Let Σ be a finite alphabet and $x_1, \dots, x_k$ a set of variables. Then* regular expressions with memory *are defined by the grammar:*

$$e \quad := \quad \varepsilon \mid a \mid e + e \mid e \cdot e \mid e^+ \mid e[c] \mid \downarrow \bar{x}.e \quad (3)$$

where a ranges over alphabet letters, c over conditions in $\mathcal{C}_k$, and $\bar{x}$ over tuples of variables from $x_1, \dots, x_k$.

A regular expression with memory e is well-formed if it satisfies two conditions:

- *Subexpressions e^+, $e[c]$, and $\downarrow \bar{x}.e$ are not allowed if e reduces to ε. Formally, e reduces to ε if it is ε, or it is $e_1 + e_2$ or $e_1 \cdot e_2$ or e_1^+ or $e_1[c]$ or $\downarrow \bar{x}.e_1$ where e_1 (and e_2) reduce to ε.*
- *No variable appears in a condition before it appears in $\downarrow \bar{x}$.*

The class of well-formed regular expressions with memory is denoted by $\text{REG}(\Sigma[x_1, \dots, x_k])$.

The extra condition of being well-formed is to rule out pathological cases like $\varepsilon[c]$ for checking conditions over empty subexpressions, or $a[x^=]$ for checking equality with a variable that has not been defined. In what follows we always assume that regular expressions with memory are well-formed.

The intuition behind the expressions is that they process a data path in the same way that the register automaton would, by storing data values in variables, using these variables for comparisons and moving through the word by reading a letter from the finite alphabet.

Example 5.2. We now give four examples of such expressions and languages they recognize, before formally defining their semantics.

1. The expression $\downarrow x.(a[x^{\neq}])^+$ defines the language of data paths where all edges are labeled a and the first data value is different from all other data values. It starts by binding x to the first data value; then it proceeds checking that the letter is a and condition $x^{\neq}$ is satisfied, which is expressed by $a[x^{\neq}]$; the expression is then put in the scope of $+$ to indicate that the number of such values is arbitrary.

2. The expression $\downarrow x.(ab)^+[x^{\neq}]$ denotes the language of data paths whose label is of the form $ab \dots ab$ and for which the first data value is different from the last. Note that the order of $+$ and condition is now different: the condition is checked after verifying that the label is in $(ab)^+$, i.e., at the end of the word.

3. The expression $\downarrow x.a^+[x^=] + \varepsilon$ denotes the language of data paths where all labels are a and the first data value is equal to the last. Note that one such data path is simply of the form d, for $d \in \mathcal{D}$, with label ε.

4. The language L_{eq} of data paths in which two data values are the same (see Section 3) is given by the expression $\Sigma^* \cdot \downarrow x. \Sigma^+ [x^=] \cdot \Sigma^*$, where Σ is the shorthand for $a_1 + \ldots + a_l$, whenever $\Sigma = \{a_1, \ldots, a_l\}$ and Σ^* is the shorthand for $\Sigma^+ + \varepsilon$. It says: at some point, bind x, and then check that after one or more edges, we have the same data value.

Semantics First, we define the *concatenation* of two data paths $w = d_1 a_1 \ldots a_{n-1} d_n$ and $w' = d_n a_n \ldots a_{m-1} d_m$ as $w \cdot w' = d_1 a_1 \ldots a_{n-1} d_n a_n \ldots a_{m-1} d_m$. Note that it is only defined if the last data value of w equals the first data value of w'. The definition naturally extends to concatenation of several data paths. If $w = w_1 \cdots w_l$, we shall refer to it as a *splitting* of a data path (into $w_1, \ldots, w_l$).

The semantics is defined by means of a relation $(e, w, \sigma) \vdash \sigma'$, where $e \in \mathrm{REG}(\Sigma[x_1, \ldots, x_k])$ is a regular expression with memory, w is a data path, and both σ and σ' are k-tuples over $\mathcal{D} \cup \{\bot\}$ (the symbol $\bot$ means that a register has not been assigned yet). The intuition is as follows: one can start with a memory configuration σ (i.e., values of $x_1, \ldots, x_k$) and parse w according to e in such a way that at the end the memory configuration is σ'. The language of e is then defined as

$$L(e) = \{w \mid (e, w, \bar{\bot}) \models \sigma \text{ for some } \sigma\},$$

where $\bar{\bot}$ is the tuple of k values $\bot$.

The relation $\vdash$ is defined inductively on the structure of expressions. Recall that the empty word corresponds to a data path with a single data value d (i.e., a single node in a data graph). We use the notation $\sigma_{\bar{x}=d}$ for the valuation obtained from σ by setting all the variables in $\bar{x}$ to d.

- $(\varepsilon, w, \sigma) \vdash \sigma'$ iff $w = d$ for some $d \in \mathcal{D}$ and $\sigma' = \sigma$.
- $(a, w, \sigma) \vdash \sigma'$ iff $w = d_1 a d_2$ and $\sigma' = \sigma$.
- $(e_1 \cdot e_2, w, \sigma) \vdash \sigma'$ iff there is a splitting $w = w_1 \cdot w_2$ of w and a valuation σ'' such that $(e_1, w_1, \sigma) \vdash \sigma''$ and $(e_2, w_2, \sigma'') \vdash \sigma'$.
- $(e_1 + e_2, w, \sigma) \vdash \sigma'$ iff $(e_1, w, \sigma) \vdash \sigma'$ or $(e_2, w, \sigma) \vdash \sigma'$.
- $(e^+, w, \sigma) \vdash \sigma'$ iff there are a splitting $w = w_1 \cdots w_m$ of w and valuations $\sigma = \sigma_0, \sigma_1, \ldots, \sigma_m = \sigma'$ such that $(w, w_i, \sigma_{i-1}) \vdash \sigma_i$ for all $i \in [m]$.
- $(\downarrow \bar{x}.e, w, \sigma) \vdash \sigma'$ iff $(e, w, \sigma_{\bar{x}=d}) \vdash \sigma'$, where d is the first data value of w.
- $(e[c], w, \sigma) \vdash \sigma'$ iff $(e, w, \sigma) \vdash \sigma'$ and $\sigma', d \models c$, where d is the last data value of w.

Take note that in the last item we require that σ', and not σ, satisfies c. The reason for this is that our initial assignment might change before reaching the end of the expression and we want this change to be reflected when we check that condition c holds.

Translation into automata We now show that regular expressions with memory can be efficiently translated into register automata.

Proposition 5.3. *For each regular expression with memory* $e \in \mathrm{REG}(\Sigma[x_1, \ldots, x_k])$ *one can construct, in* DLOGSPACE, *a* k-*register data path automaton* $\mathcal{A}_e$ *such that* $L(e) = L(\mathcal{A}_e)$.

More precisely, the automaton $\mathcal{A}_e = (Q, q_0, F, \bar{\bot}, \delta)$ *(over data domain* $\mathcal{D} \cup \{\bot\}$*) has the property that for any two valuations* σ, σ' *and a data path* w*, we have* $(e, w, \sigma) \vdash \sigma'$ *iff the automaton* $(Q, q_0, F, \sigma, \delta)$ *has an accepting run on* w *that ends with the register configuration* σ'.

5.1 Query evaluation

We now deal with the following queries.

Definition 5.4. *A regular data path query with memory is an expression* $Q = x \xrightarrow{e} y$, *where* e *is regular expression with memory.*

Given a data graph G, *the result of the query* $Q(G)$ *consists of pairs of nodes* (v, v') *such that there is a data path* w *from* v *to* v' *that belongs to* $L(e)$.

The class of these queries is denoted by $\mathrm{RDPQ}_{\mathrm{mem}}$.

Using Proposition 5.3 combined with Theorem 4.6 we immediately obtain:

Corollary 5.5. *Data complexity of* $\mathrm{RDPQ}_{\mathrm{mem}}$ *queries is in* NLOGSPACE.

From the same connection we also get the upper bound (PSPACE) for combined complexity. It turns out that we can achieve PSPACE-hardness with expressions with memory (see the appendix for the proof). Thus, we have

Theorem 5.6. *Combined complexity of evaluating* $\mathrm{RDPQ}_{\mathrm{mem}}$ *queries is* PSPACE-*complete*.

The question is whether we can reduce this complexity – ideally to PTIME, but at least to NP, to match the combined complexity of conjunctive queries. The following corollary (to the proof of Theorem 5.6) shows that many restrictions will not work.

Corollary 5.7. *Combined complexity of evaluating* $\mathrm{RDPQ}_{\mathrm{mem}}$ *queries remains* PSPACE-*hard for expressions that use at most one* $^+$ *and* $\neq$ *symbol, are specified over a singleton alphabet* $\Sigma = \{a\}$, *and are evaluated over a fixed graph*.

In one case, we can lower the complexity.

Proposition 5.8. *Combined complexity of* $\mathrm{RDPQ}_{\mathrm{mem}}$ *queries whose regular expressions do not have subexpressions of the form* e^+ *is* NP-*complete*.

The restriction, while achieving better combined complexity, is too strong, as it effectively restricts one to languages of data paths whose projections on Σ^* are finite. All the examples we saw earlier use subexpressions e^+. So if we want to achieve tractability, we need to look at a very different way of restricting expressions. This is what we do in the next section.

6. Queries based on regular expressions with equality

The class of regular expressions for data paths that lets us lower the combined complexity of queries to PTIME permits testing for equality or inequality of data values at the beginning or the end of a data (sub)path. For example, $(\Sigma^+)_{\neq}$ denotes the set of all data paths having different first and last data values. The language L_{eq} of data paths on which two data values are the same is given by $\Sigma^* \cdot (\Sigma^+)_= \cdot \Sigma^*$: it checks for the existence of a nonempty subpath (with label in Σ^+) such that the nodes at the beginning and at the end of this subpath have the same data value, indicated by subscript $=$.

Definition 6.1 (Expressions with equality). *Let Σ be a finite alphabet. Then* regular expressions with equality *are defined by the grammar:*

$$e \;:=\; \varepsilon \mid a \mid e + e \mid e \cdot e \mid e^+ \mid e_= \mid e_{\neq} \quad (4)$$

where a ranges over alphabet letters.

The language $L(e)$ of data paths denoted by a regular expression with equality e is defined as follows.

- $L(\varepsilon) = \{d \mid d \in \mathcal{D}\}$.
- $L(a) = \{dad' \mid d, d' \in \mathcal{D}\}$.
- $L(e \cdot e') = L(e) \cdot L(e')$.
- $L(e + e') = L(e) \cup L(e')$.
- $L(e^+) = \{w_1 \cdots w_k \mid k \geq 1 \text{ and each } w_i \in L(e)\}$.
- $L(e_=) = \{d_1 a_1 \ldots a_{n-1} d_n \in L(e) \mid d_1 = d_n\}$.
- $L(e_{\neq}) = \{d_1 a_1 \ldots a_{n-1} d_n \in L(e) \mid d_1 \neq d_n\}$.

These expressions sacrifice the ability to check conditions as one goes along the path, making it only possible to check conditions at the start and the end of chosen subexpressions. Looking at Example 5.2, all languages except the first can be defined by regular expressions with memory. We already saw how to do the language L_{eq}; the expression $\downarrow x.(ab)^+[x^{\neq}]$ is equivalent to $(ab)^+_{\neq}$. The expression $\downarrow x.(a[x^{\neq}])^+$ describing the language of data paths in which all data values are different from the first one, requires checking a condition multiple times. We now show that this goes beyond the power of expressions with equality, which are strictly weaker than expressions with memory.

Proposition 6.2. *1. For each regular expression with equality, there is an equivalent regular expression with memory.*

2. For the regular expression with memory $\downarrow x.(a[x^{\neq}])^+$ there is no equivalent regular expression with equality.

6.1 Query evaluation

We now deal with the following queries.

Definition 6.3. *A regular data path query with equality is an expression $Q = x \xrightarrow{e} y$, where e is regular expression with equality.*

Given a data graph G, the result of the query $Q(G)$ consists of pairs of nodes (v, v') such that there is a data path w from v to v' that belongs to $L(e)$.

The class of these queries is denoted by $\mathrm{RDPQ}_=$.

Combining Propositions 5.3 and 6.2 we see that the power of regular expressions with equality is subsumed by register automata; hence combined with Theorem 4.6 we immediately obtain:

Corollary 6.4. *Data complexity of $\mathrm{RDPQ}_=$ queries is in* NLOGSPACE.

We now show that combined complexity for $\mathrm{RDPQ}_=$ queries is tractable, i.e., is even better than the combined complexity of conjunctive queries. Our outline of the polynomial-time algorithm is as follows. We start with a data graph $G = \langle V, E, \rho \rangle$ whose data values form a (finite) set $D \subset \mathcal{D}$ and a regular expression with equality e.

1. We first show that we can efficiently generate a context-free grammar $\mathcal{G}_{e,D}$ whose language corresponds to the set of all data paths from $L(e)$ whose data values are in D. More precisely, every word in $L(\mathcal{G}_{e,D})$ will be of the form $d_1 a_1 d_2 d_2 a_2 d_3 d_3 \ldots d_{n-1} d_{n-1} a_{n-1} d_n$, where $d_i \in D$ and $a_i \in \Sigma$. We say that this word, in which each data value, except the first and the last, appears twice, corresponds to the data path $d_1 a_1 d_2 a_2 d_3 \ldots a_{n-1} d_n$.

2. We then convert $\mathcal{G}_{e,D}$, in polynomial time, into an equivalent PDA $\mathcal{A}(\mathcal{G}_{e,D})$.

3. Given two nodes v, v' in G, we construct an NFA $\mathcal{A}_{G,v,v'}$. To do so we first define a graph $G' = \langle V', E' \rangle$ that will reflect the fact that all data values from G have to be doubled if they appear on a path as intermediate nodes. We define $G' = \langle V', E' \rangle$ as follows:

   - $V' \;=\; V \cup \{\tilde{u}, \hat{u} \mid u \in V\} \cup \{s, t\}$
   - $E' \;=\; \{(v_1, a, \tilde{v}_2) \mid (v_1, a, v_2) \in E\}$
     $\bigcup \{(\tilde{u}, \rho(u), \hat{u}), (\hat{u}, \rho(u), u) \mid u \in V\}$

Similarly as when dealing with register automata we triple each node and add an edge between new nodes that will reflect the fact that every intermediate data value will have to be doubled. This is illustrated below.
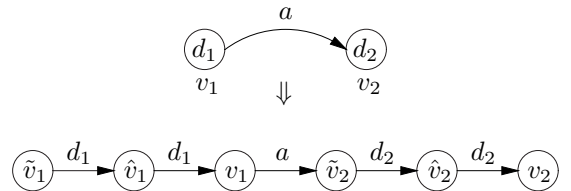
In addition, we also add edges $(s, \rho(v), v)$ and $(\tilde{v}', \rho(v'), t)$ to E'. We now get the automaton $\mathcal{A}_{G,v,v'}$ as the automaton obtained from G' by setting s as the initial and t as the final state. Note that the construction of the automaton $\mathcal{A}_{G,v,v'}$ is polynomial.

4. Finally, for $Q = x \xrightarrow{e} y$ we have $(v, v') \in Q(G)$ iff the language $\mathcal{A}_{G,v,v'}$ has nonempty intersection with the language generated by the grammar $\mathcal{G}_{e,D}$. This follows by an argument similar to the proof of Proposition 4.5.

Since the intersection of a context-free language and a regular language is context-free and can be obtained by the product construction of a PDA and an NFA, this means that $(v, v') \in Q(G)$ iff the product $\mathcal{A}(\mathcal{G}_{e,D}) \times \mathcal{A}_{G,v,v'}$ defines a nonempty language. This product is a PDA, so we can check its nonemptiness in polynomial time, giving us a polynomial algorithm for query evaluation.

Steps 2, 3, and 4 above use the standard constructions of converting CFGs into PDAs, taking products, and checking PDAs for nonemptiness. So what is missing is the construction of the CFG $\mathcal{G}_{e,D}$, which we show next.

Regular expressions with equality into CFGs Assume that we have a finite set D of data values. We now inductively construct CFGs $\mathcal{G}_{e,D}$ for all regular expressions with equality. The terminal symbols of these CFGs will be Σ plus all elements of D. All nonterminals in $\mathcal{G}_{e,D}$ will be of the form $A_{e'}$ and $A_{e'}^{dd'}$, where e' ranges over subexpressions of e and $d, d' \in D$. Intuitively, words derived from $A_{e'}^{dd'}$ will correspond to (in a way previously described) data paths in $L(e')$ with data values from D that start with d and end with d'; words derived from $A_{e'}$ will correspond to data paths in $L(e')$ with data values from D. The start symbol for the grammar corresponding to the expression e will be A_e.

The productions of the grammars $\mathcal{G}_{e,D}$ are now defined inductively as follows.

- If $e = \varepsilon$, we have productions $A_\varepsilon \to \bigvee_{d \in D} A_\varepsilon^{dd}$ and $A_\varepsilon^{dd} \to d$ for each $d \in D$.
- If $e = a$, for $a \in \Sigma$, we have productions $A_e \to \bigvee_{d,d' \in D} A_e^{dd'}$ and $A_e^{dd'} \to dad'$ for all $d, d' \in D$.
- If $e = e_1 \cdot e_2$, we have productions $A_e \to \bigvee_{d,d' \in D} A_e^{dd'}$ and $A_e^{dd'} \to \bigvee_{d'' \in D} A_{e_1}^{dd''} A_{e_2}^{d''d'}$ for all $d, d' \in D$ together with all the productions of the grammars $\mathcal{G}_{e_1,D}$ and $\mathcal{G}_{e_2,D}$.
- If $e = e_1 + e_2$, we have productions $A_e \to \bigvee_{d,d' \in D} A_e^{dd'}$ and $A_e^{dd'} \to A_{e_1}^{dd'} | A_{e_2}^{dd'}$ for all $d, d' \in D$ together with all the productions of the grammars $\mathcal{G}_{e_1,D}$ and $\mathcal{G}_{e_2,D}$.
- If $e = (e_1)^+$, we have productions $A_e \to \bigvee_{d,d' \in D} A_e^{dd'}$ and $A_e^{dd'} \to A_{e_1}^{dd'} | \bigvee_{d'' \in D} A_{e_1}^{dd''} A_e^{d''d'}$ for all $d, d' \in D$ together with all the productions of the grammar $\mathcal{G}_{e_1,D}$.
- If $e = (e_1)_=$, we have productions $A_e \to \bigvee_{d \in D} A_e^{dd}$ and $A_e^{dd} \to A_{e_1}^{dd}$ for all $d \in D$ together with all the productions of the grammar $\mathcal{G}_{e_1,D}$.
- If $e = (e_1)_{\neq}$, we have productions $A_e \to \bigvee_{d,d' \in D,\ d \neq d'} A_e^{dd'}$ and $A_e^{dd'} \to A_{e_1}^{dd'}$ for all $d, d' \in D$

with $d \neq d'$, together with all the productions of the grammar $\mathcal{G}_{e_1,D}$.

It is clear from the construction that all words generated by this grammar(with the sole exception of the empty word) have all of their intermediate data values (i.e. letters corresponding to values in D) doubled, except the first and the last one.

Note that with these expressions we assume that ε can appear only when denoting the empty word and will be removed otherwise. We require this, so that we would not get productions that produce objects that are not data paths, such as e.g. ddd for the expression $\varepsilon \cdot \varepsilon \cdot \varepsilon$. Note that this is not a problem, since all expressions can be rewritten to be of this form in DLOGSPACE.

The main result connecting these CFGs with languages of regular expressions with equality is this. Recall that when we say that a word over Σ and D corresponds to a data path with values in D, we mean that it equals the data path with all the data values, except the first and the last, doubled.

Proposition 6.5. *The language of words derived by each CFG $\mathcal{G}_{e,D}$ corresponds to the set of data paths in $L(e)$ whose data values come from D. Furthermore, the set of words derived from each nonterminal $A_e^{dd'}$ corresponds to the set of data paths in $L(e)$ which start with d, end with d', and whose data values come from D.*
Moreover, the CFG $\mathcal{G}_{e,D}$ can be constructed in polynomial time from e and D.

This, together with the algorithm shown above, finally gives us tractability of combined complexity.

Theorem 6.6. *Combined complexity of* RDPQ$_=$ *queries is in* PTIME.

The correctness of the procedure shown in this section is proved in the appendix.

7. Conjunctive regular path queries with data

A standard extension of RPQs is that to *conjunctive RPQs*, or CRPQs [12, 18, 22]. These add conjunctions of RPQs and existential quantification over variables, in the same way as conjunctive queries extend atomic formulae of relational calculus. We now look at similar extensions of RPQs with data.

Formally, a *conjunctive regular data path query (CRDPQ)* is an expression of the form

$$Ans(\bar{z}) := \bigwedge_{1 \leq i \leq m} x_i \xrightarrow{L_i} y_i, \tag{5}$$

where $m > 0$, each $x_i \xrightarrow{L_i} y_i$ is a regular data path query (in one of the formalisms studied here), and $\bar{z}$ is a tuple of variables among $\bar{x}$ and $\bar{y}$. A query with the head $Ans()$ (i.e., no

Query answering	RDPQ	RDPQ$_{mem}$	RDPQ$_{mem}$ over finite words	RDPQ$_=$
data complexity	NLOGSPACE-complete	NLOGSPACE-complete	NLOGSPACE-complete	NLOGSPACE-complete
combined complexity	PSPACE-complete	PSPACE-complete	NP-complete	PTIME

(a) for single data path query

Query answering	CRDPQ	CRDPQ$_{mem}$	CRDPQ$_=$
data complexity	NLOGSPACE-complete	NLOGSPACE-complete	NLOGSPACE-complete
combined complexity	PSPACE-complete	PSPACE-complete	NP-complete

(b) for conjunctive queries

Figure 2: Summary of complexity results for classes of queries

variables in the output) is called a *Boolean* query. Depending on which RDPQs are used in (5) we shall be referring to CRDPQs, or CRDPQs with memory, or CRDPQs with equality.

These queries extend RDPQs with conjunction, as well as existential quantification: variables that appear in the body but not in the head (i.e., variables in $\bar{x}$ and $\bar{y}$ but not $\bar{z}$) are assumed to be existentially quantified.

The semantics of a CRDPQ Q of the form (5) over a data graph $G = \langle V, E, \rho \rangle$ is defined as follows. Given a valuation $\nu : \bigcup_{1 \leq i \leq m}\{x_i, y_i\} \to V$, we write $(G, \nu) \models Q$ if $(\nu(x_i), \nu(y_i))$ is in the answer of $x_i \xrightarrow{L_i} y_i$ on G, for each $i = 1, \ldots, m$. Then $Q(G)$ is defined as the set of all tuples $\nu(\bar{z})$ such that $(G, \nu) \models Q$. If Q is Boolean, we let $Q(G)$ be true if $(G, \nu) \models Q$ for some ν (that is, as usual, the empty tuple models the Boolean constant true, and the empty set models the Boolean constant false).

As with RDPQs, we study data and combined complexity of the query evaluation problem, i.e. checking, for a CRDPQ Q, a data graph G and a tuple of nodes $\bar{v}$, whether $\bar{v} \in Q(G)$ (for data complexity the query Q is fixed).

First, we show that for all the three formalisms based on register automata and regular expressions for them, no cost is incurred by going from RDPQs to CRDPQs as far as data complexity is concerned.

Theorem 7.1. *Data complexity of conjunctive regular data path queries remains* NLOGSPACE-*complete if they are specified using register automata, regular expressions with memory, or regular expressions with equality.*

PROOF. Consider a query of the form (5) and let $\bar{z}'$ be the tuple of variables from $\bar{x}$ and $\bar{y}$ that is not present in $\bar{z}$. To check whether $\bar{v} \in Q(G)$, we need to check whether there exists a valuation $\bar{v}'$ for $\bar{z}'$ so that under that valuation each of the RDPQs in the conjunction in (5) is true.

We know from the previous sections that checking whether $v \xrightarrow{L} v'$ evaluates to true for some nodes v, v' can be done with NLOGSPACE data complexity for all the formalisms mentioned in the theorem. Thus, given a data graph $G = \langle V, E, \rho \rangle$, we can enumerate all the tuples from $V^{|\bar{z}'|}$, and for each of them check the truth of all the RDPQs in conjunction (5). Since we deal with data complexity,

$|\bar{z}'|$ is fixed, and thus such an enumeration can be done in logarithmic space, showing that query evaluation remains in NLOGSPACE. □

For combined complexity, we have the same bounds for CRDPQs given by register automata and expressions with memory as in the case of a single RDPQ. For regular expressions with equality we get NP-completeness, which matches the combined complexity of conjunctive queries and CR-PQs.

Theorem 7.2. *Combined complexity of conjunctive regular data path queries remains* PSPACE-*complete if they are specified using register automata or regular expressions with memory. It is* NP-*complete if they are specified using regular expressions with equality.*

PROOF. PSPACE-hardness follows from the corresponding results for RDPQs and RDPQs with memory, and NP-hardness follows from NP-hardness of relational conjunctive queries. Thus we show upper bounds. The algorithm (using notations from the proof of Theorem 7.1) is the same in all three cases: guess a tuple $\bar{v}'$ of nodes for $\bar{z}'$, and check whether all the RDPQs in conjunction (5) are true. We know that for register automata and regular expressions with memory the latter can be done in PSPACE; since PSPACE is closed under nondeterministic guesses we have the PSPACE upper bound for combined complexity. For regular expressions with equality, an NP upper bound for the algorithm follows from the PTIME bound for combined complexity for RDPQs with equality. □

8. Summary and future work

The tables in Figure 2 give the summary of data and combined complexity for various query languages studied in this paper. As we introduced models that expand the usual RPQs and CRPQs that handle only edge labels and can now manipulate data in the nodes, we get, as expected, a slightly higher complexity bounds for combined complexity. However, using a large class of regular expressions that can express many properties of interest, we can match the usual bound of RPQs. For CRPQs with data, the bounds are only slightly higher than those for data-free CRPQs; in some cases they coincide with bounds for CRPQs extended with

comparisons of paths, and for some, there is no price to pay for incorporating data comparisons into queries.

This is an initial investigation on combining data and topology in graph query languages, and we plan to extend this work in several directions. One of them has to do with optimizing queries, in particular, with studying containment and equivalence as in [18, 25]. We are also interested in handling constraints in graph query languages [2, 26]. Another direction is to study extensions with path comparisons as in [4], combined with querying data. We also plan to study incomplete data, by extending patterns in [5] with data, potentially incomplete.

Yet another direction we intend to pursue is to define our expressions over data words, a setting usually treated in the literature, and try to study their classical language theoretic properties, such as membership testing, nonemptiness, containment, etc. To lower complexity we might even consider restricting regular expressions with memory in such a way that equality tests are more explicit, while still allowing them to be far more expressive than expressions with equality. We would also like to specify a class of expressions that precisely capture register automata in the same manner that regular expressions capture finite state automata. We have strong indications that we will be able to do so with regular expressions with memory.

9. 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, V. Vianu. Regular path queries with constraints. *J. Comput. Syst. Sci.* 58(3):428-452 (1999).

[3] R. Angles, C. Gutiérrez. Survey of graph database models. *ACM Comput. Surv.* 40(1): (2008).

[4] P. Barceló, C. Hurtado, L. Libkin, P. Wood. Expressive languages for path queries over graph-structured data. In *PODS'10*, pages 3–14.

[5] P. Barceló, L. Libkin, J. Reutter. Querying graph patterns. In *PODS'11*, pages 199–210.

[6] M. Benedikt, W. Fan, F. Geerts. XPath satisfiability in the presence of DTDs. *J. ACM* 55(2): (2008).

[7] M. Bojanczyk, P. Parys. XPath evaluation in linear time. In *PODS'08*, pages 241-250.

[8] M. Bojanczyk. Automata for data words and data trees. In *RTA 2010*.

[9] M. Bojanczyk, A. Muscholl, T. Schwentick, L. Segoufin. Two-variable logic on data trees and XML reasoning. *J. ACM* 56(3): (2009).

[10] M. Bojanczyk, C. David, A. Muscholl, T. Schwentick, L. Segoufin. Two-variable logic on words with data. *ACM TOCL* 12(4): (2011).

[11] P. Bouyer, A. Petit, D. Thérien. An algebraic characterization of data and timed languages. *CONCUR'01*, pages 248–261.

[12] D. Calvanese, G. de Giacomo, M. Lenzerini, M. Y. Vardi. Containment of conjunctive regular path queries with inverse. In *KR'00*, pages 176–185.

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

[14] M. P. Consens, A. O. Mendelzon. GraphLog: a visual formalism for real life recursion. In *PODS'90*, pages 404–416.

[15] I. Cruz, A. Mendelzon, P. Wood. A graphical query language supporting recursion. In *SIGMOD'87*, pages 323-330.

[16] S. Demri, R. Lazic. LTL with the freeze quantifier and register automata. *ACM TOCL* 10(3): (2009).

[17] S. Demri, R. Lazić, D. Nowak. On the freeze quantifier in constraint LTL: Decidability and complexity. *Inf. Comput.* 205(1): 2–24 (2007).

[18] A. Deutsch, V. Tannen. Optimization properties for classes of conjunctive regular path queries. *DBPL'01*, pages 21–39.

[19] W. Fan, J. Li, S. Ma, N. Tang, Y. Wu. Graph pattern matching: from intractable to polynomial time. *PVLDB* 3(1): 264-275 (2010).

[20] D. Figueira. Satisfiability of downward XPath with data equality tests. *PODS'09*, 197-206.

[21] D. Figueira and L. Segoufin. Bottom-up automata on data trees and vertical XPath. *STACS'11*, pages 93–104.

[22] D. Florescu, A. Levy, D. Suciu. Query containment for conjunctive queries with regular expressions. In *PODS'98*, pages 139–148.

[23] S. Fortune, J. Hopcroft, and J. Wyllie. The directed homeomorphism problem. *Theoretical Computer Science*, 10:111-121, 1980.

[24] G. Gottlob, C. Koch, and R. Pichler. Efficient algorithms for processing XPath queries. *ACM Trans. Database Syst*, 30(2):444-491, 2005.

[25] G. Grahne, A. Thomo. Algebraic rewritings for optimizing regular path queries. In *ICDT'01*, pages 301–315.

[26] G. Grahne, A. Thomo. Query containment and rewriting using views for regular path queries under constraints. In *PODS'03*, pages 111–122.

[27] C. Gutierrez, C. Hurtado, A. Mendelzon. Foundations of semantic Web databases. *J. Comput. Syst. Sci.* 77(3): 520-541 (2011).

[28] M. Kaminski and N. Francez. Finite memory automata. *Theoretical Computer Science*, 134(2):329-363, 1994.

[29] M. Kaminski, T. Tan. Tree automata over infinite alphabets. In *Pillars of Computer Science*, 2008, pages 386–423.

[30] M. Kaminski and T. Tan. Regular expressions for languages over infinite alphabets. *Fundam. Inform.*, 69(3):301-318, 2006.

[31] U. Leser. A query language for biological networks. *Bioinformatics* 21 (suppl 2) (2005), ii33–ii39.

[32] R. Milo, S. Shen-Orr, S. Itzkovitz, N. Kashtan, D. Chklovskii, U. Alon. Network motifs: simple building blocks of complex networks. *Science* 298(5594) (2002), 824–827.

[33] F. Neven, Th. Schwentick, V. Vianu. Finite state machines for strings over infinite alphabets. *ACM TOCL* 5(3):403-435 (2004).

[34] F. Neven, Th. Schwentick. XPath containment in the presence of disjunction, DTDs, and variables. *Logical Methods in Computer Science* 2(3) (2006).

[35] H. Sakamoto and D. Ikeda., Intractability of decision problems for finite-memory automata. *Theor. Comput. Sci.* 231, 2, 297-308, 2000.

[36] F. Olken. Graph data management for molecular biology. *OMICS* 7(1): 75-78 (2003).

[37] J. Pérez, M. Arenas, C. Gutierrez. Semantics and complexity of SPARQL. *ACM TODS* 34(3): 2009.

[38] R. Ronen and O. Shmueli. SoQL: a language for querying and creating data in social networks. In *ICDE 2009*.

[39] M. San Martín, C. Gutierrez. Representing, querying and transforming social networks with RDF/SPARQL. In *ESWC 2009*, pages 293–307.

[40] L. Segoufin. Automata and logics for words and trees over an infinite alphabet. In *CSL'06*, pages 41-57.

[41] L. Segoufin. Static analysis of XML processing with data values. *SIGMOD Record* 36(1): 31-38 (2007).

[42] J. Van den Bussche, G. Vossen. An extension of path expressions to simplify navigation in object-oriented queries. In *DOOD'93*, pages 267–282.

APPENDIX

Proofs

10. Proof of Theorem 3.1

We do a reduction from 2-DISJOINT PATHS QUERY problem proven to be NP-complete in [23]. This problem is to check, for a graph G and four nodes s_1, t_1, s_2, t_2 in G, whether there exist two paths in G, one from s_1 to t_1 and the other from s_2 to t_2 that have no nodes in common.

Assume that $G = \langle V, E \rangle$ is a graph and s_1, t_1, s_2, t_2 are four nodes in G. Here we assume that all four nodes are distinct. It is easy to see that with this assumption the problem remains NP-complete, because we can always add two new nodes for each repeated node and connect them with all the nodes the repeated node was connected to, thus modifying our problem to have all source and target nodes different. We let our query be $Q = x \xrightarrow{\overline{L_{eq}}} y$. Since our query will disregard edge labels we can take $\Sigma = \{a\}$. We will construct a data graph G' and two nodes $s, t \in G'$ such that $(s, t) \in Q(G')$ if and only if there are two disjoint paths in G from s_1 to t_1 and from s_2 to t_2.

Let $V = \{v_1, \ldots, v_n\}$. The graph G' will contain two disjoint isomorphic copies of G (with data values and labels attached) connected by a single edge. We define the two isomorphic copies G_1 and G_2 by:

- $G_k = \langle V_k, E_k, \rho_k \rangle$, where
- $V_k = \{v'_1, \ldots, v'_n\}$,
- $E_k = \{(v'_i, a, v'_j) : (v_i, v_j) \in E\}$ and
- $\rho_k(v'_i) = i$, for $i = 1 \ldots n$

for $k = 1, 2$, and then let $G' = \langle V', E', \rho' \rangle$, where

- $V' = V_1 \cup V_2$,
- $E' = E_1 \cup E_2 \cup \{(t'_1, a, s''_2)\}$ and
- $\rho' = \rho_1 \cup \rho_2$.

Note that ρ' is well defined since V_1 and V_2 are disjoint.

Finally we define $s = s'_1$ and $t = t''_2$.

We claim that $(s, t) \in Q(G')$ if and only if there are two disjoint paths in G from s_1 to t_1 and from s_2 to t_2 in G. To see this assume first that $(s, t) \in Q(G')$. This means that we have a path in G' which starts in s'_1 and ends in t''_2. In particular, it must pass the edge between t'_1 and s''_2, since this is the only edge connecting the two graphs. Also, since all data values on this path are different we know that no node can repeat. But then we simply split this path into two disjoint paths in G since the structure of edges in G' is the same as the one in G with the exception of edge between t'_1 and s''_2. Also, no node can be repeated, since the corresponding nodes in G_1 and G_2 have the same data values.

Conversely, if we have two disjoint paths from s_1 to t_1 and from s_2 to t_2 in G, we simply follow the corresponding path from s'_1 to t'_1 in G_1 (and thus in G'), traverse the edge between t'_1 and s''_2 and then follow the path in G_2 (and thus in G') from s''_2 to t''_2 corresponding to the path from s_2 to t_2 in G.

This completes the proof of the theorem.

11. Proof of Lemma 4.2

First we fix some notation. An accepting run of a register data path automaton $\mathcal{A}$ on $w = d_1 a_1 d_2 \ldots a_{n-1} d_n$ is a sequence of configurations $C_0, C_1, \ldots, C_{2n-1}$ starting with the initial configuration, ending in some final configuration and such that for every $i < 2n - 1$, the automaton can move from C_i to C_{i+1} by reading the appropriate symbol of w. For each accepting run there is also a sequence of transitions from δ_d and δ_w witnessing this run. This sequence always starts and ends with a

transition from δ_d and the consecutive transitions are alternating between δ_d and δ_w. We will often identify accepting run of $\mathcal{A}$ on w with its witnessing sequence. An example of such an accepting sequence for $\mathcal{A}$ on $w = d_1 a_1 d_2 \ldots a_{n-1} d_n$ is a sequence

$$(q_0, c_1, I_1, q_1) \to (q_1, a_1, q_2) \to (q_2, c_2, I_2, q_3) \to (q_3, a_2, q_4) \to \ldots \to (q_{2n-2}, c_n, I_n, q_{2n-1}),$$

with $q_{2n-1} \in F$.

A run of a data word automaton is defined analogously.

We are now ready to prove Lemma 4.2.

Let $\mathcal{A} = (Q, q_0, F, \tau_0, \delta = (\delta_w, \delta_d))$ be our automaton over data paths. We know that $Q = Q_w \cup Q_d$. We define $\mathcal{A}' = (Q', q_{-1}, F', \tau'_0, T)$, an automaton over data words, as follows:

- $Q' = \{q_{-1}\} \cup Q_w$, where q_{-1} is the new initial state;
- $F' = F$;
- $\tau'_0 = \tau_0$;
- For every transition $(q_0, c, I, q) \in \delta_d$ we add $(q_{-1}, \#, c, I, q)$ to T. Also, for every pair of transitions $(q_1, a, q_2), (q_2, c, I, q_3)$, with $q_1, q_3 \in Q_w$ and $q_2 \in Q_d$ we add the transition (q_1, a, c, I, q_3) to T.

To see the equivalence, assume that $w = d_1 a_1 d_2 \ldots a_{n-1} d_n$ is in $L(\mathcal{A})$. Then there exists an accepting run

$$(q_0, c_1, I_1, q_1) \to (q_1, a_1, q_2) \to (q_2, c_2, I_2, q_3) \to (q_3, a_2, q_4) \to \ldots \to (q_{2n-2}, c_n, I_n, q_{2n-1}),$$

with $q_{2n-1} \in F$. But then

$$(q_{-1}, \#, c_1, I_1, q_1) \to (q_1, a_1, c_2, I_2, q_3) \to \ldots \to (q_{2n-3}, a_n, c_n, I_n, q_{2n-1})$$

is an accepting run of $\mathcal{A}'$ on $\binom{\#}{d_1}\binom{a_1}{d_2} \ldots \binom{a_{n-1}}{d_n}$.

Conversely, assume that

$$(q_{-1}, \#, c_1, I_1, q_1) \to (q_1, a_1, c_2, I_2, q_3) \to \ldots \to (q_{2n-3}, a_n, c_n, I_n, q_{2n-1})$$

is an accepting run of $\mathcal{A}'$ on $\binom{\#}{d_1}\binom{a_1}{d_2} \ldots \binom{a_{n-1}}{d_n}$. Since every transition of $\mathcal{A}'$, except for the first one is made up from two transitions of $\mathcal{A}$, we know that for each (q, a, c, I, q') in this accepting run there exists $q'' \in Q_d$ such that $(q, a, q'') \in \delta_w$ and $(q'', c, I, q') \in \delta_d$. This pair of transitions will process some pair $\binom{a}{d}$. From this we get an accepting run of $\mathcal{A}$ on $d_1 a_1 \ldots d_n$ starting with (q_0, c_1, I_1, q_1) (processing d_1, since the condition here is for A' to accept $\binom{\#}{d_1}$) and continuing through following transitions as described above.

The DLOGSPACE bound is also immediate.

12. Proof of Corollary 4.3

We prove PSPACE-hardness by doing a reduction from regular automata nonuniversality. This problem requires us to determine, given a finite state automaton $\mathcal{A}$, whether $L(\mathcal{A}) \neq \Sigma^*$.

The proof we give here is similar to the proof for data words given in [16]. It is easy, though quite tedious, to check that the described algorithm can indeed be implemented on a register automaton. Here we give a high level description of how to do so.

Assume we are given a regular automaton $\mathcal{A} = (Q, \Sigma, \delta, q_1, F)$, where $Q = \{q_1, \ldots, q_n\}$ and $F = \{q_{i_1}, \ldots, q_{i_k}\}$. Using an automaton with $2n + 2$ registers we will emulate the following algorithm which solves reachability in the powerset automaton for the complement $\overline{\mathcal{A}}$.

We initialize our automaton by storing two different data values, which we denote by t and f, in the first two registers.

Our algorithm stores two states of $\overline{\mathcal{A}}$, both of which are encoded as an n-bit sequence of t/f. If the ith bit of the sequence is set to t, it means that q_i is included in our state of $\overline{\mathcal{A}}$. The state we start in is $tf \ldots f$, where t corresponds to state q_1, the initial state. It is easy to code this into our automaton.

In what follows we will refer to the two n-bit sequences coding the two states of $\overline{\mathcal{A}}$ as the current state tape and the next state tape. These will be used to test reachability in the powerset automaton for $\overline{\mathcal{A}}$ and will work in the usual manner. This means that we are trying to guess a word in the complement automaton by guessing a letter from the alphabet at each step and

updating the next state tape (which will code the next state) with respect to the current state tape (holding the description of the current state). At the end of each step we simply copy the next state tape to the current state tape and start from there in the next step.

To simulate this in every step we reset the next state tape to contain all false and we nondeterministically pick out a letter of the accepting word for $\overline{\mathcal{A}}$ and apply all possible transitions from the current state (i.e. from the states where we have t on our tape). We remember the result on the new tape and at the end copy it to the current state tape. That is, our register automaton loops over the following set of instructions: it first sets its next state tape (stored in registers $n + 3 \dots 2n + 2$) to contain all f values, then nondeterministically picks a letter of the alphabet and updates the next state tape according to the values it reads on the current state tape (i.e. it only updates if the value is true). After going through the entire current state tape and updating according to δ it copies the next state tape to the current state tape.

The algorithm stops if it reaches a state where all states in F are tagged with f, or it has exhausted all 2^n states. Since we use only two tapes and polynomially many operations in each step our algorithm is in PSPACE. That is, our automaton can chose to nondeterministically enter a block of states specifying that all states in F are coded by f on the current state tape.

We now claim that there is a word in the language of the constructed register automaton if and only if language of $\mathcal{A}$ is not universal.

It is clear from the description that the language of our given automaton is not universal if and only if there is a sequence of tape descriptions starting from the initial state description and moving according to the previous algorithm such that the last state description has f in all positions corresponding to the states in F. But this simply means that the language of the constructed automaton is nonempty. Conversely, if the language of the constructed automaton is nonempty it clearly describes such a set of transitions that leads to a word not in the language of original $\mathcal{A}$.

13.   Proof of Proposition 4.5

It follows immediately from the construction that the automaton $\mathcal{A}_D$ accepts precisely those data paths form $L(\mathcal{A})$ that have data values from D. To see this it suffices to show that every accepting run of $\mathcal{A}_D$ corresponds to an accepting run of $\mathcal{A}$ and vice versa, in the case of paths whose data values come form D. But this follows easily since $\mathcal{A}_D$ has all possible configurations of registers at it's disposal.

To see that the statement of Proposition holds assume first that $(v, v') \in Q(G)$. Then there is a data path $w_\pi = d_0 a_0 d_1 a_1 \dots a_{n-1} d_n$ from v to v' such that $w_\pi \in L(\mathcal{A})$. Since this is a data path in G starting with v and ending with v' it must also be a word in the language of $\mathcal{A}_{G', v, v'}$. On the other hand, since it is in $L(\mathcal{A})$, it must also be in $L(\mathcal{A}_D)$, since $\mathcal{A}_D$ is simply restriction of $\mathcal{A}$ to alphabet in which data values come only from the set D. Thus $L(\mathcal{A}_{G', v, v'} \times \mathcal{A}_D) \neq \emptyset$.

Conversely, assume that $L(\mathcal{A}_{G', v, v'} \times \mathcal{A}_D) \neq \emptyset$. Then there is a data path $w_\pi = d_0 a_0 d_1 a_1 \dots a_{n-1} d_n$ such that $w_\pi \in L(\mathcal{A}_{G', v, v'})$ and $w_\pi \in L(\mathcal{A}_D)$. But then by construction w_π must be a data path in G from v to v'. Also $w_\pi \in L(\mathcal{A})$, since $L(\mathcal{A}_D)$ is simply a restriction of language of $\mathcal{A}$ to data paths whose data values come from D. But this implies that $(v, v') \in Q(G)$.

14.   Proof of Theorem 4.6

We only need to prove PSPACE-hardness, since upper PSPACE bound follows from on-the-fly method for checking nonemptiness of exponential size automata. But this is an immediate consequence of Proposition 5.3 and Theorem 5.6, which are proved for a more restricted language.

15.   Proof of Proposition 5.3

We prove this by induction on the structure of e. Note that the initial assignment of $\mathcal{A}_e$ is not specified in advance. We will simply put the assignment in as needed, since it does not change the structure of the underlying automaton. In what follows we will identify the vector $\overline{x}$ of variables with the set of registers (i.e. positions) it corresponds to. For example the vector (x_3, x_5) will correspond to the set $I = \{3, 5\}$ of registers.

If $(e, w, \sigma) \vdash \sigma'$, we will write $w \in L(e, \sigma, \sigma')$ and similarly if $\mathcal{A}_e = (Q, q_0, F, \overline{\bot}, \delta)$ started with σ accepts w with σ' in the registers, we write $w \in L(\mathcal{A}_e, \sigma, \sigma')$.

- If $e = \varepsilon$, then $\mathcal{A}_e = (Q, q_0, F, \overline{\bot}, \delta)$, where $Q = \{d\} \cup \{w\}$ is the set of states, $q_0 = d$ is the initial state, $F = \{w\}$ the set of final states and the only transition is $(d, \varepsilon, \emptyset, w)$.

- If $e = a$, for some $a \in \Sigma$ then $\mathcal{A}_e = (Q, q_0, F, \bar{\bot}, \delta)$, where $Q = \{d_1, d_2\} \cup \{w_1, w_2\}$ is the set of states, $q_0 = d_1$ the initial state, $F = \{w_2\}$ the final state and the transition functions are as follows: $\delta_w = \{(w_1, a, d_2)\}$ is the word transition relation, and $\delta_d = \{(d_1, \varepsilon, \emptyset, w_1), (d_2, \varepsilon, \emptyset, w_2)\}$ is the data transition relation.

- If $e = e_1 + e_2$ then by the inductive hypothesis we already have automata $\mathcal{A}_{e_1} = (Q_1, d_1, F_1, \bar{\bot}, \delta_1)$ and $\mathcal{A}_{e_2} = (Q_2, d_2, F_2, \bar{\bot}, \delta_2)$ with the desired property. The registers of $\mathcal{A}_e$ will be the union of registers of $\mathcal{A}_{e_1}$ and $\mathcal{A}_{e_2}$. To obtain the desired automaton we set $\mathcal{A}_e = (Q, d_0, F, \bar{\bot}, \delta)$, where
  - $Q = Q_1 \cup Q_2 \cup \{d_0\}$, where d_0 is a new data state,
  - $F = F_1 \cup F_2$,
  - To δ we add all transitions from $\mathcal{A}_{e_1}$ and $\mathcal{A}_{e_2}$ and in addition, for every transition $(d, c, I, w) \in \delta_1 \cup \delta_2$, where $d = d_1$, or $d = d_2$, we add a transition (d_0, c, I, w).

  To see that this automaton has the desired property assume that $w \in L(e_1 + e_2, \sigma, \sigma')$. This means $(e_1 + e_2, w, \sigma) \vdash \sigma'$. By definition, $(e_1, w, \sigma) \vdash \sigma'$ or $(e_2, w, \sigma) \vdash \sigma'$. By the induction hypothesis it follows that either $\mathcal{A}_{e_1}$, or $\mathcal{A}_{e_2}$ accepts w and halts with σ' in the registers (when started with σ). From this it is clear that $\mathcal{A}_e$ can simulate the same accepting run when started with σ in the registers(by using the transition from d_0 to the appropriate automaton and continuing on the same run there). (Note that all conclusions here are equivalences.)

- If $e = \downarrow\bar{x}.e_1$ then again by the induction hypothesis we have $\mathcal{A}_{e_1} = (Q_1, d_1, F_1, \bar{\bot}, \delta_1)$ with the desired property. The automaton for $\mathcal{A}_e$ is defined as $\mathcal{A}_e = (Q_1 \cup \{d_0\}, d_0, F_1, \bar{\bot}, \delta)$, where d_0 is a new data state and δ contains all the transitions of $\mathcal{A}_{e_1}$ and in addition, for every transition (d_1, c, I, w), going from the initial state of $\mathcal{A}_{e_1}$, we add a transition $(d_0, c, I \cup \bar{x}, w)$ to δ. The registers of $\mathcal{A}_e$ are the union of registers of $\mathcal{A}_{e_1}$ and $|\bar{x}|$ new registers.

  To see the equivalence, assume that $w \in L(e, \sigma, \sigma')$. By definition $(e, w, \sigma) \vdash \sigma'$. It follows that $(e_1, w, \sigma_{\bar{x}=v_1}) \vdash \sigma'$, where v_1 is the first data value in w and $\sigma_{\bar{x}=v_1}$ is the same as σ except that every register in $\bar{x}$ contains v_1. By the induction hypothesis we know that $\mathcal{A}_{e_1}$ with $\sigma_{\bar{x}=v_1}$ as initial assignment has an accepting run on w ending with σ' in the registers. But then $\mathcal{A}_e$ starting with σ in the registers can go through the same run with the exception that the first transition will change σ to $\sigma_{\bar{x}=v_1}$ and since all other transitions are the same we have the desired result. (Note that all conclusions here are equivalences.) It is important to note that potential confusion of the variables will cause no conflicts. To see this assume we have a transition (d_1, c, I, w) in $\mathcal{A}_{e_1}$ and we start with σ as initial assignment. If I and $\bar{x}$ have variables in common it will not matter, since all of them will get replaced by the same value, namely the first data value of w. This means that the first step of the run will end up with the same result. Also note that no transition in δ_d with d_1 as the first component will have $c \neq \varepsilon$, since this would amount to an expression starting with a condition, something disallowed by our syntax.

- If $e = e_1[c]$ then let $\mathcal{A}_{e_1} = (Q_1, d_1, F_1, \bar{\bot}, \delta_1)$ be an automaton for e_1 as before. We define $\mathcal{A}_e = (Q, d_1, F, \bar{\bot}, \delta)$ where $Q = Q_1 \cup \{w_f\}$, with w_f a new state, $F = \{w_f\}$ and for every transition (d, c', I, w) where $w \in F_1$ we add a transition $(d, c' \wedge c, I, w_f)$ to $\mathcal{A}_e$. We have to add a new state simply because our original automaton could have looped back from some final state.

  To get the equivalence assume again that $w \in L(e, \sigma, \sigma')$. By definition $(e_1, w, \sigma) \vdash \sigma'$ and $\sigma', v \models c$, where v is the last data value in w. From the induction hypothesis we get an accepting run of $\mathcal{A}_{e_1}$ with σ as initial configuration and σ' as final one. But since $\sigma', v \models c$ instead of the last transition we can simply make a transition to w_f in $\mathcal{A}_e$ (since all other transitions are the same). We again notice that all the implications can be reversed, i.e. we can prove the equivalence.

- If $e = e_1 \cdot e_2$, take again $\mathcal{A}_{e_1}$ and $\mathcal{A}_{e_2}$ as above. The automaton for e is simply the union of the previous two automata, but in addition to the already existing transitions we add the following: for every (d, c, I, w) in $\mathcal{A}_{e_1}$, where $w \in F_1$ and for every (d_2, c', I', w') in $\mathcal{A}_{e_2}$, where d_2 is the initial state of $\mathcal{A}_{e_2}$, we add $(d, c \wedge c', I \cup I', w')$ to δ. Note that I is going to be an empty set, since we work with well formed expressions. We also make d_1 the initial state and F_2 the set of final states. The registers of $\mathcal{A}_e$ are again the union of registers of $\mathcal{A}_{e_1}$ and $\mathcal{A}_{e_2}$.

  To get the desired result once again assume that $w \in L(e, \sigma, \sigma')$. This means $(e, w, \sigma) \vdash \sigma'$, which implies that there exists some σ'' and a splitting $w = w_1 \cdot w_2$ of w such that $(e_1, w_1, \sigma) \vdash \sigma''$ and $(e_2, w_2, \sigma'') \vdash \sigma'$. By the induction hypothesis we know that there is an accepting run of $\mathcal{A}_{e_1}$ on w_1 starting with σ and ending with σ'' in the registers and also an accepting run of $\mathcal{A}_{e_2}$ on w_2 starting with σ'' and ending with σ' in the registers. But we can simply combine these two runs into an accepting run of $\mathcal{A}_e$ on w. We do so by setting σ as initial assignment and tracing the run of $\mathcal{A}_{e_1}$ till the final state. Now instead of taking the last transition we will take one of the newly added transitions from the next to final state in $\mathcal{A}_{e_1}$ to the next to first state in $\mathcal{A}_{e_2}$. Note that we can do this since we know there is an accepting run of $\mathcal{A}_{e_2}$ on w_2 and since $w = w_1 \cdot w_2$, so their last and first data value, respectively, coincide. Note that at this point we end up with σ'' in the registers and can continue the accepting run of $\mathcal{A}_{e_2}$ and thus $\mathcal{A}_e$.

  Conversely, if we have an accepting run of $\mathcal{A}_e$ on w, we split the run, and thus the path, into the part before and after taking the new transition added while constructing the automaton. Note that we have to take this transition in order to pass from the initial state, which is in $\mathcal{A}_{e_1}$ part of $\mathcal{A}_e$, to a final state, which is in a $\mathcal{A}_{e_2}$ part of $\mathcal{A}_e$. From this it follows that $w \in L(e)$.

- If $e = e_1^+$, then let again $\mathcal{A}_{e_1}$ be the automaton from the induction hypothesis. Note first that this automaton has at least four states, since $\text{Proj}(e_1) \neq \varepsilon$, where $\text{Proj}(e)$ denotes the projection to the finite alphabet Σ, and transitions going directly

from initial to final state can only accept the empty word, so they will not alter computations or acceptance. We let the automaton for e be the same as the one for e_1, but we add the following transitions: for every (d, c, I, w) with $w \in F_1$ and for every (d_1, c', I', w'), where d_1 is the initial state of $\mathcal{A}_{e_1}$, we add $(d, c \wedge c', I \cup I', w')$ to our transition function, thus bypassing the last and the first state.

Assume now that $(e, w, \sigma) \vdash \sigma'$. Then either $(e_1, w, \sigma) \vdash \sigma'$, so we are done by the induction hypothesis, or $w = w_1 \cdots w_k$ with $k \geq 2$ and valuations $\sigma_1, \ldots, \sigma_{k+1}$ exist such that $(e_1, w_i, \sigma_i) \vdash \sigma_{i+1}$ for $i = 1, \ldots, k$. But then by the induction hypothesis we have computations of $\mathcal{A}_{e_1}$ with σ_i as the initial assignment and σ_{i+1} as final assignment that accept w_i, for $i = 1, \ldots, k$. Note that this actually means that we start with σ, do a computation for w_1, end with σ_2 in the registers, then take the new transition bypassing the end state for this computation and thus starting the computation with σ_2 in the registers(and updating the registers as dictated by the first transition in the new cycle), etc., until we reach σ' after reading w_k, thus accepting w.

For the converse, if $\mathcal{A}_e$ accepts w when started with σ and ended with σ' then we simply split the data path for every time we take the additional transitions added in the construction of $\mathcal{A}_e$. From this we get computations of $\mathcal{A}_{e_1}$ on sub-paths with intermediate valuations. By the induction hypothesis we have acceptance of these subpaths by e_1 with appropriate valuations and thus the membership of the entire path w in $L(e, \sigma, \sigma')$.

This concludes the proof. It is straightforward to check that all the constructions can be carried in DLOGSPACE.

16.   Proof of Theorem 5.6

The PSPACE upper bound follows from Theorem 4.6 and Proposition 5.3. Thus we only have to prove PSPACE-hardness. For this we do a reduction from regular automata nonuniversality problem. The idea is, similarly to proof of Corollary 4.3, to simulate on the fly reachability testing in the powerset automaton by using two sets of variables, each of the size of the automaton, for coding the current and next state.

Let $\mathcal{A} = (Q, \Sigma, \delta, q_1, F)$ be a finite state automaton, where $Q = \{q_1, \ldots, q_n\}$ and $F = \{q_{i_1}, \ldots, q_{i_k}\}$. We will construct a fixed graph G with 5 nodes, containing two distinguished nodes s and t in G and construct, in polynomial time, a regular expression with memory e,of length $O(n \times |\Sigma|)$, such that $(s, t) \in Q(G)$ if and only if $L(\mathcal{A}) \neq \Sigma^*$, where $Q = x \xrightarrow{e} y$.

The graph G is shown below:
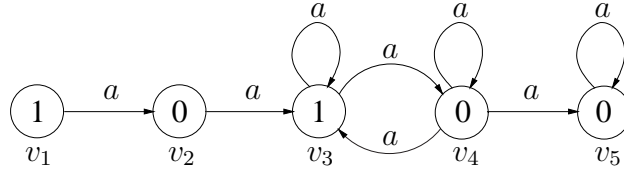
We now set $s = v_1$ and $t = v_5$.

Since we are trying to demonstrate nonuniversality of the automaton $\mathcal{A}$ we simulate reachability checking in the powerset automaton for $\overline{\mathcal{A}}$. To do so we designate two distinct data values, t and f, and code each state of the powerset automaton as an n-bit sequence of t/f values, where the ith bit of the sequence is set to t if the state q_i is included in our state of $\overline{\mathcal{A}}$. Since we are checking reachability we will need only to remember the current and the next state of $\overline{\mathcal{A}}$. In what follows we will code those two states using variables $s_1, \ldots, s_n$ and $w_1, \ldots, w_n$ and refer to them as stable tape and work tape. Our expression e will code data paths that describe successful runs of $\overline{\mathcal{A}}$ by demonstrating how one can move from one state of this automaton to another (as witnessed by their codes in stable and work tapes), starting with the initial and ending in a final state.

We will define several expressions and explain their role. We will use two sets of variables, s_1 through s_n and $w_1, \ldots, w_n$ to denote stable and work tape (i.e. current and next state in the powerset automaton). All of these variables will only contain two values, t and f, which are bound in the beginning and that will correspond to 0 and 1 in the graph G.

The first expression we need is:

$$\texttt{init} := \downarrow t.a[t^{\neq}] \downarrow f.a[t^{=}] \downarrow s_1.a[f^{=}] \downarrow s_2.\ldots.a[f^{=}] \downarrow s_n.a.$$

This expression codes two different values as t and f and initializes stable tape to contain encoding of initial state (the one where only initial state from $\mathcal{A}$ can be reached). That is, a data path is in the language of this expression if and only if it starts with two different data values and continues with n data values that form a sequence in 10^*.

$$\texttt{end} := a[f^{=} \wedge s_{i_1}^{=}] \cdot a[f^{=} \wedge s_{i_2}^{=}] \cdots a[f^{=} \wedge s_{i_k}^{=}], \text{ where } F = \{q_{i_1}, \ldots, q_{i_k}\}.$$

16

This expression is used to check that we have reached a state not containing any final state from the original automaton. That is, a data path is in $L(\text{end})$ if and only if it consists of k data values, all equal to f and where value stored in s_{i_j} also equals f, for $j = 1 \ldots k$.

Next we define expressions that will reflect updating of the work tape according to the transition function of $\mathcal{A}$. Assume that $\delta(q_i, b) = \{q_{j_1}, \ldots, q_{j_l}\}$. We define

$$u_{\delta(q_i, b)} := \left(a[t^= \wedge s_i^=] \cdot a[t^=] {\downarrow} w_{j_1} . \ldots . a[t^=] {\downarrow} w_{j_l} . a\right) + a[f^= \wedge s_i^=].$$

Also, if $\delta(q_i, b) = \emptyset$ we simply put $u_{\delta(q_i, b)} := \varepsilon$.

This expression will be used to update the work tape by writing true to corresponding variables if the state q_i is tagged with t on the work tape (and thus contained in the current state of $\overline{\mathcal{A}}$). If it is false we skip the update.

Since we have to define update according to all transitions from all the states corresponding to chosen letter we get:

$$\text{update} := \bigvee_{b \in \Sigma} \bigwedge_{q_i \in Q} u_{\delta(q_i, b)}.$$

This simply states that we non deterministically pick the next symbol of the word we are guessing and move to the next state accordingly.

We still have to ensure that the tapes are copied at the beginning and end of each step, so we define:

$$\text{step} := (a[f^=] {\downarrow} w_1 . \ldots . a[f^=] {\downarrow} w_n . a) \cdot \text{update} \cdot (a[w_1^=] {\downarrow} s_1 . \ldots . a[w_n^=] {\downarrow} s_n . a).$$

This simply initializes the work tape at the beginning of each step, proceeds with the update and copies the new state to stable tape. Note the few odd a's at the end of the expressions. These will not affect what we what to achieve and are here for syntactical reasons(to get a proper expression).

Finally we have

$$e := \text{init} \cdot (\text{step})^* \cdot \text{end}.$$

Here we use step^* as abbreviation for $\text{step}^+ + \varepsilon$.

We claim that for $Q = x \xrightarrow{e} y$, we have $(s, t) \in Q(G)$ if and only if $L(\mathcal{A}) \neq \Sigma^*$.

Assume first that $L(\mathcal{A}) \neq \Sigma^*$. This means that there is a path from the initial to the final state in the powerset automaton for $\overline{\mathcal{A}}$. That is, there is a word w from Σ^* not in the language of $\mathcal{A}$. This path can in turn be described by pairs of assignment of values t/f to stable and work tape, where each transition is witnessed by the corresponding letter of the alphabet. But then the path from s to t in G that belongs to $L(e)$ is the one that first initializes the stable tape (i.e. the variables $s_1, \ldots, s_n$) to initial state of the powerset automaton, then runs the updates of the tape according to w and finally ends in a state where all variable corresponding to end states of $\mathcal{A}$ are tagged f. Note that we can describe this path in G, since we start in s and put 1 into t in node v_1, 0 into f in node v_2. After that 1 is assigned to s_1 in v_3 and 0 to $s_2, \ldots, s_n$ by looping through v_4. After that each transition is reflected by going through v_3 and v_4 as necessary, to update tapes with t/f and finally going to v_5 and looping there to check that all s_i's corresponding to end states are tagged f.

Conversely, each path from s to t in $L(e)$ corresponds to a run of the powerset automaton for $\overline{\mathcal{A}}$. That is, the part of path corresponding to init sets the initial state. Then the part of this path that corresponds to step^* corresponds to updating our tapes in a way that properly codes one step of powerset automaton. Finally, end denotes that we have reached a state where all end states of $\mathcal{A}$ have been tagged by f, thus, an accepting state for $\overline{\mathcal{A}}$.

17.   Proof of Proposition 5.8

Recall that for $e \in \text{REG}(\Sigma[x_1, \ldots, x_k])$, by $\text{Proj}(e)$ we denote the projection of e to the finite alphabet Σ.

First we show NP-membership. Since we do not use $^+$ we know that every data path in the language of expression e uses at most $|\text{Proj}(e)|$ letters and one more data value. Assume now that we are given a data graph G, two nodes $s, t \in G$ and an expression with memory e. To see if $(s, t) \in Q(G)$, for $Q = x \xrightarrow{e} y$, we use the following algorithm. First compute the register automaton $\mathcal{A}_e$ for e. Note that this can be done in DLOGSPACE. Then nondeterministically guess a data path w_π in G from s to t that is of length at most $|\text{Proj}(e)|$. Now also guess $2|\lambda(w_\pi)| + 1$ states of $\mathcal{A}_e$ and check that the path w_π is

accepted by $\mathcal{A}_e$, as witnessed by this sequence of states, and thus is in $L(e)$. It is straightforward to see that this can be done in polynomial time and since our guesses are of polynomial (in fact linear) size we get the desired result.

For hardness we do a reduction from k-CLIQUE. This problem asks for a given 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 data graph G' with $|G| + 2$ nodes, select two nodes $s, t \in G'$ and construct a regular expression with memory e_k of size $O(k^2)$ such that G contains a k-clique if and only if there is a data path from s to t in G' that satisfies e_k.

Take $\Sigma = \{a, b\}$ and make G directed by adding edges in both directions for every edge in G. Label all the edges by a and add two more nodes s and t. Add an edge from s to every other node except s, t and label them with b. Also add an edge from every node in G to t and label them by b. To finish the construction just add a different data value to every node. We call the resulting graph G'.

To define e_k we use an auxiliary expression δ_i defined as:

$$\delta_i := a[x_1^=] \cdot a[x_i^=] \cdot a[x_2^=] \cdot a[x_i^=] \ldots a[x_{i-1}^=] \cdot a[x_i^=].$$

This expression will simply allow us to test that the current node is connected to all nodes previously selected in our potential clique.

Now we can define e_k inductively as follows:

- $e_1 := b \cdot \downarrow x_1.a[x_1^{\neq}]$,
- $e_2 := e_1 \cdot \downarrow x_2.a[x_1^{\neq} \wedge x_2^{\neq}]$,
- $e_i := e_{i-1} \cdot \downarrow x_i.\delta_i \cdot a[x_1^{\neq} \wedge \ldots \wedge x_i^{\neq}]$, for $i = 3, \ldots, k-1$ and
- $e_k := e_{k-1} \cdot \downarrow x_k.\delta_k \cdot b$.

Next we show that there is a k-clique in G iff there is a data path form s to t in G' that satisfies e_k.

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 labeled 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 satisfy δ_i, 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 labeled edge.

Now suppose that there is a data path from s to t in G' that satisfies e_k. This means that we will be able to select k different nodes $n_1, \ldots, n_k$ in G with data values stored in $x_1, \ldots, x_k$. Since all data values in the graph are different they also act as ids. Now take any two n_i, n_j with $i < j \leq k$. Then we know that n_i and n_j are connected in G because after selecting n_j we have to go through δ_j which contains $a[x_i^=] \cdot a[x_j^=]$ and since no two data values in G are the same this means that we have an edge between n_i and n_j. This completes the proof.

18. Proof of Proposition 6.2

For first item it is enough to observe that for expressions of the kind $e_=$ and $e_{\neq}$, where e is an ordinary regular expression, the expressions with memory $\downarrow x.e[x^=]$ and $\downarrow x.e[x^{\neq}]$ denote the same language of data paths. From this it is straightforward to construct a translation of arbitrary regular expression with equality e to regular expression with memory by doing the above mentioned construction bottom-up, starting from subexpressions of e and using a new variable for each subexpression of the form $e'_=$ or $e'_{\neq}$.

To prove the second claim we introduce a new kind of automata, called weak register automata, show that they capture regular expressions with equality and that they can not express the language $\downarrow x.(a[x^{\neq}])^+$ of a-labeled data paths on which all data values are different from the first one.

The main idea behind weak register automata is that they erase the data value that was stored in the register once they make a comparison, thus rendering the register empty. We denote this by putting a special symbol $\perp$ from $\mathcal{D}$ in the register. Since they have a finite number of registers, they can keep track of only finitely many positions in the future, so in the case of our language, they can only check that a fixed finite number of data values is different from the first one. We proceed with formal definitions.

The definition of weak k-register data path automaton is the same as in the Definition 4.1. The only explicit change we make

is that we now assume that $\mathcal{C}_k$ contains a special symbol ε, that will allow us to simply skip the data value, without doing any comparisons (previously we have been using a simple tautology such as $x_1^= \vee x_1^{\neq}$, or an additional register to emulate this). Thus we simply add $\tau, d \models \varepsilon$, for every valuation τ and data value d, to semantics of $\mathcal{C}_k$. We will also assume that the initial configuration is always empty.

Definition of configuration remains the same as before, but the way we move from one configuration to another changes.

From a configuration $c = (j, q, \tau)$ we can move to a configuration $c' = (j+1, q', \tau')$ if one of the following holds:

- the jth symbol is a letter a, and there is a transition $(q, a, q') \in \delta_w$; or
- the current symbol is a data value d, and there is a transition $(q, c, I, q') \in \delta_d$ such that $d, \tau \models c$ and τ' coincides with τ except that every register mentioned in c is set to be empty (i.e. to contain $\bot$) and the ith component of τ' is set to d whenever $i \in I$.

The second item simply tells us that if we used a condition like $c = x_3^= \wedge x_7^{\neq}$ in our transition, we would afterwards erase data values that were stored in registers 3 and 7. Note that we can immediately rewrite these registers with the current data value.

The notion of acceptance and an accepting run is the same as before.

We now show that weak register automata can not recognize the language L of all data paths where first data value is different from all other data values, i.e. the language denoted by the expression $\downarrow x.(a[x^{\neq}])^+$.

Assume to the contrary, that there is some weak k−register data path automaton $\mathcal{A}$ recognizing L. Since data path $w_\pi = 1a2a \ldots ka(k+1)a(k+2)$ is in L, there is an accepting run of $\mathcal{A}$ on w_π. The idea behind the proof is that $\mathcal{A}$ can check that only the first $k+1$ positions have different data value from the first.

First we note a few things. Since every data value in the path w_π is different, no $=$ comparisons can be used in conditions appearing in this run (otherwise the condition test would fail and the automaton would not accept).

Now note that since we have only k registers, and with every comparison we empty the corresponding register one of the following must occur:

- There is a data value $1 < i < k+2$ such that the condition used when processing this data value is ε. In this case we simply replace i by 1 and get an accepting run on a word that has the first data value repeated – a contradiction. Note that we could store i in that transition, but since afterwards we only test for inequality this will not alter the rest of the computation.
- There is a data value such that when the automaton reads it it does not use any register with the first data value, i.e. 1, stored. Note that this must happen, because at best we can store the first data value in all the registers at the beginning of our run, but after that each time we read a data value and compare it to the first we lose the first data value in this register. But then again we can simply replace this data value with 1 and get an accepting run (just as before, if this data value gets stored in this transition and then used later it can only be used in $\neq$ comparison, which is also true for 1, so the run remains accepting). Again we arrive at a contradiction.

This shows that no weak register automaton can recognize the language L.

To complete the proof of Proposition 6.2 we still have to show the following:

Lemma 18.1. *For every regular expression with equality e there exists a weak k-register automaton $\mathcal{A}_e$, recognizing the same language of data paths, where k is the number of times $=, \neq$ symbols appear in e.*

PROOF. The proof is almost identical to the proof of Proposition 5.3. We can view this as introducing a new variable for every $=, \neq$ comparison in e and act as the subexpression $e'_=$ reads $\downarrow x.e'[x^=]$ and analogously for $\neq$. Note that in this case all variables come with their scope, so we do not have to worry about transferring register configurations from one side of the construction to another (for example when we do concatenation). The underlying automata remain the same. $\square$

19. Proof of Proposition 6.5

We prove the proposition by induction on the structure of e. Note that it is enough to show the second claim, i.e. we will show that the set of words derived from each nonterminal $A_e^{dd'}$ corresponds to the set of data paths in $L(e)$ which start with d, end with d', and whose data values come from D. This means that a word $d_1 a_1 d_2 d_2 a_2 d_3 d_3 \ldots a_{n-1} d_n$ in which all values but first and last are doubled is derived from $A_e^{dd'}$ if and only if data path $d_1 a_1 d_2 a_2 d_3 \ldots a_{n-1} d_n$ is in $L(e)$ and uses data values from D. We prove this by induction on the structure of the expression.

- If $e = \varepsilon$, or $e = a$, with $a \in \Sigma$, the claim is immediate.
- If $e = e_1 + e_2$ then $A_e^{dd'} \to A_{e_1}^{dd'} | A_{e_2}^{dd'}$. But then each word in $A_e^{dd'}$ is either in $A_{e_1}^{dd'}$ or in $A_{e_2}^{dd'}$, so the claim follows from the induction hypothesis.
- If $e = e_1 \cdot e_2$, we have a production $A_e^{dd'} \to \bigvee_{d'' \in D} A_{e_1}^{dd''} A_{e_2}^{d''d'}$. To see the equivalence assume first that w is generated by $A_e^{dd'}$. This means that there exists $d'' \in D$ such that w is generated by $A_{e_1}^{dd''} A_{e_2}^{d''d'}$. By definition this means that $w = w_1 \cdot w_2$ such that w_1 is generated by $A_{e_1}^{dd''}$ and w_2 is generated by $A_{e_2}^{d''d'}$. By the induction hypothesis this implies that data path w_1' corresponding to w_1, is in the language of e_1, starts with d and ends with d''. Likewise w_2', a data path corresponding to w_2 starts with d'', ends with d' and is in the language of e_2. Note that the induction hypothesis also implies that the splitting of the word is correct. Since w_1' ends with d'' and w_2' begins with it we can concatenate these two data paths to get w', a data path corresponding to w, that is in the language of e, begins with d and ends with d' as required.

  Conversely, suppose that $w' \in L(e)$ is a data path that begins with d, ends with d' and takes only data values from the set D. By definition of concatenation there exists a splitting $w' = w_1' \cdot w_2'$ such that $w_1' \in L(e_1)$ and $w_2' \in L(e_2)$. Since w' takes data values from D there is some d'' such that w_1' ends with d'' and w_2' begins with d''. But then by the induction hypothesis w_1, word obtained from w_1' by doubling all intermediate data values, will be generated by $A_{e_1}^{dd''}$, while w_2, a word obtained from w_2' by doubling all intermediate data values, will be generated by $A_{e_2}^{d''d'}$. But then their concatenation $w = w_1 \cdot w_2$ is precisely the word corresponding to data path w' and is generated by $A_{e_1}^{dd''} A_{e_2}^{d''d'}$ and thus $A_e^{dd'}$.

- If $e = (e_1)^+$, we have a production $A_e^{dd'} \to A_{e_1}^{dd'} | \bigvee_{d'' \in D} A_{e_1}^{dd''} A_e^{d''d'}$. This implies that every word is generated either by $A_{e_1}^{dd'}$, in which case the claim follows immediately from the induction hypothesis, or is generated by $\bigvee_{d'' \in D} A_{e_1}^{dd''} A_e^{d''d'}$, in which case the proof mimics the proof for the concatenation case, taking into account that recursion will terminate after finitely many steps and thus the final expression will be a multiple concatenation of terms for which the induction hypothesis holds.
- If $e = (e_1)_=$, we have $A_e^{dd} \to A_{e_1}^{dd}$, which by the induction hypothesis corresponds to all words in $L(e)$ with data values from D.
- If $e = (e_1)_{\neq}$, we have $A_e^{dd'} \to A_{e_1}^{dd'}$, where $d \neq d'$, which by the induction hypothesis corresponds to all words in $L(e)$ with data values from D.

To see that the grammar for an expression e can be constructed in polynomial time observe that there are at most $O(n^2)$ subexpressions of e, where the length of e is n. Since the grammar for e is constructed by starting from subexpressions and taking unions of already constructed subgrammars and every new rule adds at most $O(|D|^3)$ productions to our grammar we get a grammar of the size at most $O(n^2 \cdot |D|^3)$. Note that we reuse old subgrammars so we do not get exponential blow-up.

20. Proof of Proposition 6.6

It is clear from the description that algorithm runs in polynomial time. It remains to prove that it is correct, i.e. that for $Q = x \xrightarrow{e} y$ we have $(v, v') \in Q(G)$ iff the language of $\mathcal{A}_{G,v,v'}$ has nonempty intersection with the language generated by $\mathcal{A}(\mathcal{G}_{e,D})$.

To see this assume first that $(v, v') \in Q(G)$. This means that there is a data path w_π form v to v' in G such that $w_\pi \in L(e)$. By Proposition 6.5 this implies that the corresponding word with all intermediate data values doubled is in the language of $\mathcal{G}_{e,D}$ and thus $\mathcal{A}(\mathcal{G}_{e,D})$. Also, since w_π is a path in G it is of the form $d_1 a_1 \ldots a_{n-1} d_n$, where $d_i = \rho(v_i)$, for $i = 1, \ldots, n$, for some nodes $v_1, \ldots, v_n$ in G such that $v_1 = v$ and $v_n = v'$. This implies that (v_i, a_i, v_{i+1}) is an edge in E, for $i = 1, \ldots, n-1$. This again implies that $a_i d_{i+1} d_{i+1}$ enables us to change the state of $\mathcal{A}_{G,v,v'}$ from v_i to v_{i+1} (by going through $\tilde{v}_{i+1}$ and $\hat{v}_{i+1}$), for $i = 2, \ldots, n-1$. Since (s, d_1, v_1) and $(\tilde{v}_n, d_n, v_n)$ are also transitions in $\mathcal{A}_{G,v,v'}$ (as well as $(v_{n-1}, a_{n-1}, \tilde{v}_n)$) we see that $\mathcal{A}_{G,v,v'}$ accepts the word $d_1 a_1 d_2 d_2 a_2 d_3 d_3 \ldots a_{n-1} d_n$, i.e. the word corresponding to w_π. It follows that the intersection of $\mathcal{A}(\mathcal{G}_{e,D})$ and $\mathcal{A}_{G,v,v'}$ is nonempty.

Conversely, assume that the product $\mathcal{A}_{G,v,v'} \times \mathcal{A}(\mathcal{G}_{e,D})$ defines a nonempty language and that $w' = d_1 a_1 d_2 d_2 a_2 d_3 d_3 \ldots a_{n-1} d_n$ is some word in that language. If we delete doubled data values from w' (remember the discussion before the statement of Proposition 6.5 where we show that all words in $L(\mathcal{G}_{e,D})$ are of this form) we get a word w. By Proposition 6.5, w will be in the language of e. On the other hand, since $w' \in L(\mathcal{A}_{G,v,v'})$ we know that there is a run from s to t in $\mathcal{A}_{G,v,v'}$ that accepts this word. Then by the construction of this automaton there exists a sequence $v_1, \ldots, v_n$ of nodes from G such that $d_i = \rho(v_i)$ are the appropriate data values, $(v_i, a_i, v_{i+1}) \in E$ the corresponding edges and $v = v_1$, while $v' = v_n$. It is clear that w coincides with data path defined by this path and is thus a data path in G starting in v and ending in v'. We conclude that $(v, v') \in Q(G)$.