Using variable automata for querying data graphs

Domagoj Vrgoč

University of Edinburgh and PUC Chile
Vicuna Mackenna 4860, Edificio San Agustin, Macul, Santiago, Chile
domagojvrgoc@gmail.com

Abstract

Thus far query languages for graphs databases that, in addition to navigating the structure of a graph, also consider data values encountered along the paths they traverse, seem to exhibit somewhat dual behaviour in terms of the efficiency of their query evaluation problem. Namely, their combined complexity is either tractable, or are at least PSPACE-hard. In this paper we show how to use the model of variable automata to get a query language with intermediate (NP-complete) combined complexity of query evaluation. Since variable automata are incomparable in terms of expressive power with previously studied querying mechanisms for data graphs we also show how to join their capabilities with the ones of previously used languages without an increase in the complexity of query evaluation, thus getting the best of both worlds.

Keywords: Graph databases, query languages, variable automata

1. Introduction

Querying graph databases has become an important topic in the database community, fuelled by applications such as social networks, biological databases and the Semantic Web. There are now several vendors offering graph database systems [4, 12] and a growing body of research literature on the subject (for a survey see [1]). In all of these applications the data is naturally modelled by graphs, with nodes representing entities in the database and edges representing various connections these entities can form. For example if we are describing a social network it is natural to represent users by nodes, with edges symbolizing the connection between two users, such as friends, coworkers, relatives and so on. In this model a node carries information about a specific user in the usual (attribute name, attribute value) format, where the name of the attribute is drawn from a finite alphabet of labels, while the attribute value comes from an infinite domain. For example we can store information about user's name, phone number, etc. Furthermore, since nodes can form different types of connections, it is usual to assign labels to the edges connecting them, as well as some additional information such as the time of the edge creation, or how the edge was modified.

Over the years several querying mechanisms for graph data have been developed, both for navigating graphs and for dealing with the stored data, and their evaluation properties were studied in detail. Most notable among these are *regular path queries*, or RPQs [3] and their extensions with conjunction and two-way navigation [2], or the ability to define more complex graph patterns [13]. All of them have in common the fact that they query the graph structure without the ability to access data values stored in the nodes. More recently languages that in addition to topology also consider data values have been studied [9, 11]. For both these classes of languages one of the main concern is the efficiency of their *query evaluation* – that is the problem of checking, given a data graph, a query and a tuple of nodes, if this tuple belongs to the answer of the query on this given graph. This problem is often referred to as the *combined complexity* of query evaluation. When the query itself is fixed and not considered as part of the input we are talking about *data complexity*. By now the consensus is that while navigational languages can be designed with very low combined complexity in mind, for languages that mix topological properties with data features the problem is either tractable, or at least PSPACE-hard.

Indeed, it appears that models that use memory, such as register automata and their expression equivalent [11], make the query evaluation PSPACE-hard, while XPath-based approaches bring the complexity down to PTIME, but lose the ability to store values into separate memory locations. However, the panorama of languages that mix topology and data is far from being completely understood and it therefore makes sense to look for other query formalisms that might lead to languages with lower complexity of query evaluation while still retaining some of the desirable properties related to manipulation of memory locations.

One such model that was not considered previously for querying graphs is that of variable automata. These were originally introduced in [6] to reason about words over (countable) infinite alphabets, but here we show how they can also be used to define a graph query language with NP-complete combined complexity. We also show that data complexity remains NL-complete, matching the bound for RPQs. Furthermore, since variable automata are incomparable in terms of expressive power with the well established model of register automata, we show how the two can be joined together to get a graph querying formalism whose evaluation complexity (both data and combined) does not exceed that of register automata, while at the same time giving them more expressive power.

Remark. Note that some of the results presented here were announced previously in [10].

Organization. We review notation in Section 2. In Section 3 we introduce variable automata and show how they can be used to query graph databases, while in Section 4 we extend register automata in a way that subsumes properties definable by variable automata. We conclude in Section 5.

2. Preliminaries

Let Σ be a finite alphabet and $\mathcal{D}$ a countable infinite set of data values.

Definition 2.1. *A data graph (over Σ) is pair $G = (V, E)$, where*

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

We write $V(G)$ and $E(G)$ to denote the set of nodes and edges of G, respectively. An edge e from a node u to a node u' is written in the form $(u, \binom{a}{d}, u')$, where $a \in \Sigma$ and $d \in \mathcal{D}$. We call a the label of the edge e and d the data value of the edge e. We write $\mathcal{D}(G)$ to denote the set of data values in G. A sample data graph is given in Figure 1.

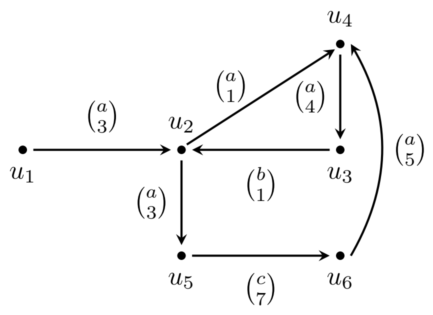

Figure 1: A graph database with data values

A path from a node v to a node v' in G is a sequence $\pi = v_1 \binom{a_1}{d_1} v_2 \binom{a_2}{d_2} v_3 \binom{a_3}{d_3} \cdots v_n \binom{a_n}{d_n} v_{n+1}$ such that each $(v_i, \binom{a_i}{d_i}, v_{i+1})$ is an edge for each $i \leq n$, and $v_1 = v$ and $v_{n+1} = v'$.

Each path π defines a *data word* $w(\pi) = \binom{a_1}{d_1}\binom{a_2}{d_2}\binom{a_3}{d_3}\cdots\binom{a_n}{d_n}$. Data words are commonly studied in XML literature [5], where they are used to describe paths in XML trees. We use them in a similar manner to describe paths in data graphs.

Remark. Note that we use the model where both labels and data values appear in the edges. Several different approaches have been used in the past, for example with data values in the nodes and labels on edges [11], or both labels and data values in nodes and edges [12], but it is easily shown that all of these variations are essentially equivalent [14]. Our choice is dictated by the ease of notation primarily, as it identifies paths with data words.

3. Variable automata as a graph language

In this section we show how to use the model of variable automata introduced in [6] to query graph databases. These automata can be viewed as less procedural than register automata [8]; in fact they can be seen as NFAs with a guess of values to be assigned to variables, with the run of the automaton verifying correctness of the guess. Originally they were defined on words over infinite alphabets [6], but it is straightforward to extend them to the setting of data words. In what follows we define variable automata as a query language, give examples of some queries one can ask using them and show that their query evaluation problem can be solved in NP-time.

We begin by defining variable automata formally.

Definition 3.1. *Let Σ be a finite alphabet and $\mathcal{D}$ a countable infinite domain of data values. We will also assume that we have a countable set V of variables. A* variable finite automaton *(or VFA for short) over $\Sigma \times \mathcal{D}$ is a pair $\mathcal{A} = (\Gamma, A)$, where A is an NFA over the alphabet $\Sigma \times \Gamma$, and $\Gamma = C \cup X \cup \{\star\}$ such that:*

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

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

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

1. $\lambda(v_i) = \lambda(w_i)$, for $i = 1, \dots, n$,
2. $\delta(v_i) = \delta(w_i)$ whenever $\delta(v_i) \in C$,
3. if $\delta(v_i), \delta(v_j) \in X$, then $\delta(w_i), \delta(w_j) \notin C$ and $\delta(w_i) = \delta(w_j)$ iff $\delta(v_i) = \delta(v_j)$,
4. $\delta(v_i) = \star$ and $\delta(v_j) \neq \star$, then $\delta(w_i) \neq \delta(w_j)$.

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

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

Example 3.2. *Here we give two examples of languages accepted by VFAs.*

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

$$\begin{tikzcd}
\textit{start} \arrow[r] & q_a \arrow[r, "\binom{a}{x}"] & q_b \arrow[loop above, "\binom{a}{\star}"] \arrow[r, "\binom{a}{x}"] & q_c
\end{tikzcd}$$

2. *The language where the last data value differs from all other data values is defined by the following VFA:*

$$\begin{tikzcd}
\textit{start} \arrow[r] & q_a \arrow[loop above, "\binom{a}{\star}"] \arrow[rr, "\binom{a}{x}"] & & q_b
\end{tikzcd}$$

Note that the last example is not expressible by register automata [8]. It was shown in [7] that the language $L = \{\binom{a}{d_1}\binom{a}{d_1}\binom{a}{d_2}\binom{a}{d_2}\cdots\binom{a}{d_k}\binom{a}{d_k} \mid k \geq 1\}$ is not expressible by VFAs. However, it is straightforward to show that it is expressible by register automata. We thus conclude that VFA and register automata are incomparable in terms of expressive power.

3.1. Using variable automata to query graph databases

We now show how to use VFAs as a query language for data graphs and study the complexity of the query evaluation problem. As is standard when using a language theoretic formalism to define a query over graph databases [3, 2], given a data graph G and a VFA $\mathcal{A}$ we define the relation $\mathcal{A}(G) \subseteq V \times V$ that consists of all pairs (s, t) of nodes in G such that there is a path π between them with the property that $w(\pi) \in L(\mathcal{A})$. The relation $\mathcal{A}(G)$ is then the answer to the query posted by $\mathcal{A}$ over G.

Now we study the following problem.

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

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

Theorem 3.3.

- *Query Evaluation for VFAs is* NP*-complete.*
- *For each fixed $\mathcal{A}$, the problem* Query Evaluation for VFAs *is* NL*-complete.*

To prove the theorem we will use the following claim:

Claim 3.4. *Assume we are given a graph G, two nodes $s, t \in G$ and a VFA $\mathcal{A}$. If there exists a word $w \in L(\mathcal{A})$ that is a label of a path in G from s to t, then there is a path in G from s to t, with the label w' and of length at most $|G| \cdot |\mathcal{A}| + 1$ such that $w' \in L(\mathcal{A})$, where $|\mathcal{A}|$ denotes the number of states in $\mathcal{A}$.*

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

By the assumption there is a path $n_0, \dots, n_l$ of nodes in G with the label w and such that $n_0 = s$ and $n_l = t$. By the pigeon hole principle there exists $i, j \leq n$ such that $n_i = n_j$ and $q_i = q_j$. Observe that $n_0, \dots, n_i, n_{j+1}, \dots, n_l$ is still a path in G from s to t with the label $w' = w_1 \dots w_i w_{j+1} \dots w_l$ and that $q_0 \dots q_i q_{j+1} \dots q_l$ is an accepting run on $v' = v_1 \dots v_i v_{j+1} \dots v_l$. Also note that v' is a witnessing pattern for w', as witnessed by the same assignment of data values to variables in v' as it was in v. By repeating this cutting procedure we get the desired result. $\square$

Proof of Theorem 3.3. We start by showing the NP upper bound. For the NP-algorithm we simply guess a path of length at most $|G| \cdot |\mathcal{A}| + 1$–a polynomial in the size of the input. We then check membership of this word in the language of $\mathcal{A}$ in NP [6]. Since the nondeterministic guessing can be carried out at the same time as guessing the actual word we obtain the desired result.

The NP-hardness follows from NP-hardness of the membership problem for variable automata from [7]. However, since there are some minor differences between the automata model in [7] and the one presented here, we provide an independent NP-hardness proof in order to make the presentation self contained. We do this by showing a reduction from k-CLIQUE. This problem asks, given an undirected, unlabelled graph G and a number k, to determine if G has a clique of size at least k.

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

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

We define our VFA as a linear path with transitions:

- $(q_0, \binom{b}{x_1}, q_1), (q_1, \binom{a}{x_2}, q_2)$ (this collect the first two nodes in the clique),
- $(q_{i-1}, \binom{a}{x_i}, q'_i), (q'_i, \binom{a}{x_1}, q^i_1), (q^i_1, \binom{a}{x_i}, p^1_i),$
  $(p^1_i, \binom{a}{x_2}, q^i_2), (q^i_2, \binom{a}{x_i}, p^2_i), \dots,$
  $(p^{i-2}_i, \binom{a}{x_{i-1}}, q^i_{i-1}), (q^i_{i-1}, \binom{a}{x_i}, q_i)$, $3 \leq i \leq k$,
- $(q_k, \binom{b}{d_t}, q_{k+1})$ (to get the target node).

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

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

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

Now suppose that there is a path from s to t in G' whose label belongs to $L(\mathcal{A})$. This means that we will be able to select k different nodes $n_1, \ldots, n_k$ 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_l, n_m with $l < m \leq k$. Then we know that n_l and n_m are connected in G because after selecting n_m we have to go through the transitions stating $(p_m^{l-1}, \binom{a}{x_l}, q_l^m), (q_l^m, \binom{a}{x_m}, p_m^l)$ and similarly for when l, m are at the beginning or the end of the transition chain. Since no two data values in G are the same this means that we have an edge between n_l and n_m. This completes the proof of NP-hardness.

Next we show the NL-bound for data complexity. Assume that the automaton $\mathcal{A}$ is fixed and only G and s, t are part of the input. Let $X = \{x_1, \ldots, x_l\}$ be the set of bound variables used by $\mathcal{A}$ and $C = \{c_1, \ldots, c_m\}$ the set of constants. For the NL-upper bound we simply guess a unique data value d_i^x appearing in G that will be assigned to the variable x_i. We can also decide that the value $d_i^x = \bot$, denoting that the variable x_i is unassigned. Since the automaton is fixed we can do this using a logarithmic amount of space.

Next we do the usual on-the-fly nonemptiness testing for our automaton. Namely, we start with $s_0 = s$ and guess a node $s_1 \in G$. If there is an edge $(s_0, \binom{a_1}{d_1}, s_1)$ in G or some a_1, d_1 our algorithm checks that:

1. $d_1 \in C$ and $(q_0, \binom{a_1}{d_1}, q_1)$ is a transition in $\mathcal{A}$, for some state q_1, or
2. $d_1 \notin C$, and for some i we have $d_1 = d_i^x$ and $(q_0, \binom{a_1}{x_i}, q_1)$ is a transition in $\mathcal{A}$, or
3. $d_1 \notin \{d_1^x, \ldots, d_l^x\}$ and $(q_0, \binom{a_1}{\star}, q_1)$ is a transition in $\mathcal{A}$.

If any of the conditions above are satisfied the algorithm proceeds with s_1 in place of s_0, otherwise it rejects. The algorithm halts and accepts if it reaches t. If it performs more than $|G| \cdot |\mathcal{A}| + 1$ steps the algorithm halts and rejects.

By Claim 3.4 and definition of acceptance for queries defined by VFAs we get the desired result. The NL-lower bound follows from the fact that graph reachability is already NL-hard. $\square$

Note that the combined complexity dropped from PSpace to NP, which is viewed as much more acceptable for query evaluation, at least over large databases. This is the complexity of relational conjunctive queries, for instance, or conjunctive regular path queries over graphs [3].

4. Adding registers to variable automata

We know that variable automata are incomparable in expressive power with register automata. In particular we showed that they can express a property that all data values differ from the last. On the other hand, bound variables in variable automata behave like a limited version of registers that are capable of storing a data value only once. As the result, variable automata are not able to express some simple properties definable even by formalisms more restrictive than register automata [11].

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

To formalize storing and comparing data values into registers we will use *conditions*. Assume that for each $k > 0$, we have registers $x_1, \ldots, x_k$ at our disposal. Then the set of conditions $\mathcal{C}_k$ is given by the grammar:

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

The satisfaction of a condition is defined with respect to a data value $d \in \mathcal{D}$ and an assignment $\tau : \{x_1, \ldots, x_k\} \to \mathcal{D}$ of registers as follows:

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

Note that an assignment of registers can also be viewed as a tuple $\tau \in \mathcal{D}^k$. We can now formally define register automata with variables.

Definition 4.1. *Let Σ be a finite alphabet, k a natural number and C a finite set of data values. A k-*register automaton with variables *(or varRA for short) is a tuple $\mathcal{A} = (Q, q_0, F, \tau_0, T, \{\star\}, \Sigma, C)$, where:*

- *Q is a finite set of states;*
- *$q_0 \in Q$ is the initial state;*
- *$F \subseteq Q$ is the set of final states;*
- *τ_0 is the initial assignment of the registers;*

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

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

- If $t_i = (q_{i-1}, a, c, I, q_i)$ then $\tau_i, \delta(w_i) \models c$, $\lambda(w_i) = a$, and τ_{i+1} is obtained from τ_i by putting $\delta(w_i)$ in registers from I. That is, we define $\tau_{i+1}(j) = \tau_i(j)$, if $j \notin I$, and $\tau_{i_1}(j) = \delta(w_i)$, if $j \in I$;
- If $t_i = (q_{i-1}, a, d, \emptyset, q_i)$, then $\lambda(w_i) = a$, $\delta(w_i) = d$ and $\tau_{i+1} = \tau_i$;
- If $t_i = (q_{i-1}, a, \star, \emptyset, q_i)$ then $\delta(w_i) = \delta(w_j)$ iff t_j is a $\star$-transition.

Notice that register automata with variables extend both register and variable automata in a natural way. Moreover, if we restrict the registers by allowing them to store values only once and restrict conditions to single equality tests, we get variable automata (to prohibit equality with constants we simply test for this in the conditions attached to each transition). On the other hand if we disallow the usage of the free variable $\star$ we get register automata (note here that constants can be easily simulated by additional registers storing the constant values in the initial configuration τ_0).

Example 4.2. *The following varVFA defines the language* $L = \{\binom{a}{d_1}\binom{a}{d_1}\binom{a}{d_2}\binom{a}{d_2}\cdots\binom{a}{d_k}\binom{a}{d_k}\binom{a}{d} \mid k \geq 1, d \neq d_1, \ldots, d_k\}$ *that is not expressible using neither register nor variable automata.*

$$\begin{tikzcd}
\text{start} \arrow[r] & q_a \arrow[r, "{a, \top, \{x\}}"] & q_b \arrow[l, bend left, "{a, x^=, \emptyset}"] \arrow[r, "{a, x^=, \emptyset}"] & q_c \arrow[r, "{a, \star, \emptyset}"] & \boxed{q_d}
\end{tikzcd}$$

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

	QUERY EVALUATION FOR VARRAS
Input:	A data graph G, two nodes s, t from $V(G)$ and a varRA $\mathcal{A}$.
Task:	Decide whether $(s, t) \in \mathcal{A}(G)$.

Despite the increased expressive power, this model still retains the evaluation complexity of register automata.

Theorem 4.3. • QUERY EVALUATION FOR VARRAS *is* PSPACE*-complete.*

- *For each fixed* $\mathcal{A}$*, the problem is* NL*-complete.*

Proof. To prove this we use a similar construction to the one in [11]. That is for a finite set of data values D and a k-register automaton with variables $\mathcal{A}$ we produce a variable automaton $\mathcal{A}_D$ that accepts precisely the same data words as $\mathcal{A}$ does when restricted to data words whose values come from the set D. We start by proving the PSPACE upper bound.

Let $\mathcal{A} = (Q, q_0, F, \tau_0, T, \{\star\}, \Sigma, C)$ be a k-register automaton with variables and D a finite set of data values.

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

- $Q' = Q \times D_\perp^k$, where $\perp$ is a new data value not in D and $D_\perp = D \cup \{\perp\}$
- $q_0' = (q_0, \tau_0)$
- $F' = F \times D_\perp^k$
- For the transitions:
  - If $(q, a, c, I, q') \in T$ we add $((q, \tau), \binom{a}{d}, (q', \tau'))$ to T' iff $\tau, d \models c$ and we have $\tau'(i) = \begin{cases} \tau(i) & \text{for } i \notin I \\ d & \text{for } i \in I \end{cases}$
  - If $(q, a, d, \emptyset, q') \in T$, with d a constant in C we add $((q, \tau), \binom{a}{d}, (q', \tau))$ to T'
  - If $(q, a, \star, \emptyset, q') \in T$ we add $((q, \tau), \binom{a}{\star}, (q', \tau))$ to T'

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

Claim 4.4. *Let w be a data word whose data values come from D. Then $w \in L(\mathcal{A}_D)$ if and only if $w \in L(\mathcal{A})$.*

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

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

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

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

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

To complete the proof of Theorem 4.3 we use the algorithm for data complexity from Theorem 3.3.

We are given our k-varRA $\mathcal{A}$, a data graph G and s, t in G. Let $D = \mathcal{D}(G)$ be the set of all data values appearing in G. Note that $|D| = O(|G|)$. Observe now that for this $\mathcal{A}$ and D the size of $\mathcal{A}_D$ is bounded by $O(|\mathcal{A}| \times |D|^k)$.

Recall that the algorithm from Theorem 3.3 checks for the existence of a path that belongs to the language of a variable automaton using standard on-the-fly technique. Since each of the states of $\mathcal{A}_D$ can be described using only polynomial space and since the size of $\mathcal{A}_D$ is exponential in the size of the input the algorithm runs in PSpace. For data complexity we have $\mathcal{A}$ fixed, so each state of $\mathcal{A}_D$ can be described using logarithmic space and the number of states is polynomial in the size of the input, so the running time is NL.

The lower bounds follow from the complexity of query evaluation of register automata [11]. □

5. Conclusions

In this paper we examined how variable automata behave when used as a graph query language. In particular we showed that combined complexity of query evaluation for them is NP-complete, a bound lower than that for other graph query languages that allow usage of memory [11, 14]. We also established a NL-complete bound for data complexity, matching that of RPQs. As variable automata are incomparable in terms of expressive power with other formalisms previously studied, we also constructed a graph query language subsuming both variable automata and register automata. Surprisingly, for this highly expressive language querying complexity does not increase when compared to register automata alone. We therefore believe that using variable automata is an interesting approach to querying graph databases and that some of their abilities can be added to register automata to increase expressive power without hindering their efficiency.

Acknowledgements. The author would like to thank the reviewers for their comments. This work was supported by EPSRC grants G049165 and J015377 and Millennium Nucleus Center for Semantic Web Research Grant NC120004.

References

[1] Barceló, P., 2013. Querying graph databases. In: PODS.
[2] Calvanese, D., De Giacomo, G., Lenzerini, M., Vardi, M., 2000. Containment of conjunctive regular path queries with inverse. In: KR.
[3] Consens, M., Mendelzon, A., 1990. Graphlog: A visual formalism for real life recursion. In: PODS.
[4] Dex, 2013. DEX query language, Sparsity Technologies. `http://www.sparsity-technologies.com/dex.php/`.
[5] Figueira, D., 2010. Reasoning on words and trees with data. Ph.D. thesis, ÉNS de Cachan.
[6] Grumberg, O., Kupferman, O., Sheinvald, S., 2010. Variable automata over infinite alphabets. In: LATA.
[7] Grumberg, O., Kupferman, O., Sheinvald, S., 2010. Variable automata over infinite alphabets. Manuscript.
[8] Kaminski, M., Francez, N., 1994. Finite memory automata. Theoretical Computer Science 134 (2).
[9] Libkin, L., Martens, W., Vrgoč, D., 2013. Querying Graph Databases with XPath. In: ICDT.
[10] Libkin, L., Tan, T., Vrgoč, D., 2013. Regular expressions with binding over data words for querying graph databases. In: DLT.
[11] Libkin, L., Vrgoč, D., 2012. Regular Path Queries on Graphs with Data. In: ICDT. pp. 74–85.
[12] Neo4j, 2013. Neo4j, The graph database. `http://www.neo4j.org/`.
[13] Pérez, J., Arenas, M., Gutierrez, C., 2010. nSPARQL: A navigational language for RDF. JWS 8 (4).
[14] Vrgoč, D., 2014. Querying graphs with data. Ph.D. thesis, School of Informatics, University of Edinburgh.