Expressive Path Queries on Graphs with Data

Pablo Barceló[1], Gaelle Fontaine[1], and Anthony W. Lin[2,3]

[1] Dept. of Computer Science, University of Chile
[2] Dept. of Computer Science, Oxford University
[3] Academia Sinica, Taipei, Taiwan

Abstract. Graph data models have recently become popular owing to their applications, e.g., in social networks, semantic web. Typical navigational query languages over graph databases — such as Conjunctive Regular Path Queries (CRPQs) — cannot express relevant properties of the interaction between the underlying data and the topology. Two languages have been recently proposed to overcome this problem: *walk logic* (WL) and *regular expressions with memory* (REM). In this paper, we begin by investigating fundamental properties of WL and REM, i.e., complexity of evaluation problems and expressive power. We first show that the data complexity of WL is nonelementary, which rules out its practicality. On the other hand, while REM has low data complexity, we point out that many natural data/topology properties of graphs expressible in WL cannot be expressed in REM. To this end, we propose *register logic*, an extension of REM, which we show to be able to express many natural graph properties expressible in WL, while at the same time preserving the elementariness of data complexity of REMs. It is also incomparable in expressive power against WL.

1 Introduction

Graph databases have recently gained renewed interest due to applications, such as the semantic web, social network analysis, crime detection networks, software bug detection, biological networks, and others (e.g., see [1] for a survey). Despite the importance of querying graph databases, no general agreement has been reached to date about the kind of features a practical query language for graph databases should support and about what can be considered a reasonable computational cost of query evaluation for the aforementioned applications.

Typical navigational query languages for graph databases — including the conjunctive regular path queries [6] and its many extensions [4] — suffer from a common drawback: they are well-suited for expressing relevant properties about the underlying topology of a graph database, but not about how it interacts with the data. This drawback is shared by common specification languages for verification [5] (e.g. CTL*), which are evaluated over a similar graph data model (a.k.a. transition systems). Examples of important queries that combine graph data and topology, but cannot be expressed in usual navigational languages for graph databases, include the following [7, 11]: (Q1) *Find pairs of people in a social network connected by professional links restricted to people of the same*

age. (Q2) *Find pairs of cities x and y in a transportation system, such that y can be reached from x using only services operated by the same company.* In each one of these queries, the connectivity between two nodes (i.e., the topology) is constrained by the data (from an infinite domain, e.g., $\mathbb{N}$), in the sense that we only consider paths in which all intermediate nodes satisfy a certain condition (e.g. they are people of the same age).

Two languages, *walk logic* and *regular expressions with memory*, have recently been proposed to overcome this problem. These languages aim at different goals:

(a) Walk logic (WL) was proposed by Hellings et al. [7] as a unifying framework for understanding the expressive power of path queries over graph databases. Its strength is on the expressiveness side. The underlying data model of WL is that of (node or edge)-labeled directed graphs. In this context, WL can be seen as a natural extension of FO with path quantification, plus the ability to check whether positions p and p' in paths π and π', respectively, have the same data values. In their paper, they assume the restriction that each node carries a distinct data value. However, as we shall see, this makes no difference in terms of the results that we can obtain.

(b) Regular expressions with memory (REMs) were proposed by Libkin and Vrgoč [9] as a formalism for comparing data values along a single path, while retaining a reasonable complexity for query evaluation. The strength of this language is on the side of efficiency. The data model of the class of REMs is that of edge-labeled directed graphs, in which each node is assigned a data value from an infinite domain. REMs define pairs of nodes in the graph database that are linked by a path satisfying a given condition c. Each such condition c is defined in a formalism inspired by the class of *register automata* [8], allowing some data values to be stored in the registers and then compared against other data values. The evaluation problem for REMs is PSPACE-complete (same than for FO over relational databases), and can be solved in polynomial time in *data complexity* [9], i.e., assuming queries to be fixed.[4] This shows that the language is, in fact, well-behaved in terms of the complexity of query evaluation.

The aim of this paper is to investigate the expressiveness and complexity of query evaluation for WL and the class of REMs with the hope of finding a navigational query language for data graphs that strikes a good balance between these two important aspects of query languages.

Contributions. We start by considering WL, which is known to be a powerful formalism in terms of expressiveness. Little is known about the cost of query evaluation for this language, save for the decidability of the evaluation problem and NP-hardness of its data complexity. Our first main contribution is to pinpoint the exact complexity of the evaluation problem for WL (and thus answering an open problem from [7]): we prove that it is non-elementary, and that this holds even in data complexity, which rules out the practicality of the language.

[4] Recall that data complexity is a reasonable measure of complexity in the database scenario [15], since queries are often much smaller than the underlying data.

We thus move to the class of REMs, which suffers from the opposite drawback: Although the complexity of evaluation for queries in this class is reasonable, the expressiveness of the language is too rudimentary for expressing some important path properties due to its inability to (i) compare data values in *different* paths and (ii) express branching properties of the graph database. An example of an interesting query that is not expressible as an REM is the following: (Q) *Find pairs of nodes x and y, such that there is a node z and a path π from x to y in which each node is connected to z.* Notice that this is the query that lies at the basis of the queries (Q1) and (Q2) we presented before.

Our second contribution then is to identify a natural extension of this language, called *register logic* (RL), that closes REMs under Boolean combinations and existential quantification over nodes, paths and register assignments. The latter allows the logic to express comparisons of data values appearing in different paths, as well as branching properties of the data. This logic is incomparable in expressive power to WL. Besides, many natural queries relating data and topology in data graphs can be expressed in RL including: the query (Q), hamiltonicity, the existence of an Eulerian trail, bipartiteness, and complete graphs with an even number of nodes. We then study the complexity of the problem of query evaluation for RL, and show that it can be solved in elementary time (in particular, that it is Expspace-complete). This is in contrast to WL, for which even the data complexity is non-elementary. With respect to data complexity, we prove that RL is Pspace-complete. We then identify a slight extension of its existential-positive fragment, which is tractable (NLogspace) in data complexity and can express many queries of interest (including the query (Q)). The idea behind this extension is that atomic REMs can be enriched with an existential branching operator – in the style of the class of *nested regular expressions* [3] – that increases expressiveness without affecting the cost of evaluation.

Organization of the paper. Section 2 defines our data model. In Section 3, we briefly recall the definition of walk logic and some basic results from [7]. In Section 4, we prove that the data complexity of WL is nonelementary. Section 5 contains our results concerning register logic. We conclude in Section 6 with future work.

2 The Data Model

We start with a definition of our data model: data graphs.

Definition 1 (Data graph). *Let Σ be a finite alphabet. A data graph G over Σ is a tuple (V, E, κ), where V is the finite set of nodes, $E \subseteq V \times \Sigma \times V$ is the set of directed edges labeled in Σ (that is, each triple $(v, a, v') \in E$ is to be understood as an edge from v to v' in G labeled a), and $\kappa : V \to \mathcal{D}$ is a function that assigns a data value in $\mathcal{D}$ to each node in V.*

This is the data model adopted by Libkin and Vrgoč [9] in their definition of REMs. In the case of WL [7], the authors adopted *graph databases* as their data model, i.e., data graphs $G = (V, E, \kappa)$ such that κ is injective (i.e. each node

carries a different data value). We shall adopt the general model of [9] since none of our complexity results are affected by the data model: upper bounds hold for data graphs, while all lower bounds are proved in the more restrictive setting of graph databases.

There is also the issue of node-labeled vs edge-labeled data graphs. Our data model is edge-labeled, but the original one for WL is node-labeled [7]. We have chosen to use the former because it is the standard in the literature [2]. Again, this choice is inessential, since all the complexity results we present in the paper continue being true if the logics are interpreted over node-labeled graph databases or data graphs (applying the expected modifications to the syntax).

Finally, in several of our examples we use logical formulas to express properties of undirected graphs. In each such case we assume that an undirected graph H is represented as a graph database $G = (V, E, \kappa)$ over unary alphabet $\Sigma = \{a\}$, where V is the set of nodes of H and E is a symmetric relation (i.e. $(v, a, v') \in E$ iff $(v', a, v) \in E$).

3 Walk Logic

WL is an elegant and powerful formalism for defining properties of paths in graph databases, that was originally proposed in [7] as a yardstick for measuring the expressiveness of different path logics.

The syntax of WL is defined with respect to countably infinite sets Π of *path variables* (that we denote $\pi, \pi_1, \pi_2, \dots$) and $\mathcal{T}(\pi)$, for each $\pi \in \Pi$, of *position variables* of sort π. We assume that position variables of different sort are different. We denote position variables by $t, t_1, t_2, \dots$, and we write t^π when we need to reinforce that position variable t is of sort π.

Definition 2 (Walk logic (WL)). *The set of formulas of WL over finite alphabet Σ is defined by the following grammar, where (i) $a \in \Sigma$, (ii) t, t_1, t_2 are position variables of any sort, (iii) π is a path variable, and (iv) t_1^π, t_2^π are position variables of the same sort π:*

$$\phi, \phi' := E_a(t_1^\pi, t_2^\pi) \mid t_1^\pi < t_2^\pi \mid t_1 \sim t_2 \mid \neg\phi \mid \phi \vee \phi' \mid \exists t\phi \mid \exists \pi\phi$$

As usual, WL formulas without free variables are called Boolean.

To define the semantics of WL we need to introduce some terminology. A *path* (a.k.a. *walk* in [7]) in the data graph $G = (V, E, \kappa)$ is a finite, nonempty sequence $\rho = v_1 a_1 v_2 \cdots v_{n-1} a_{n-1} v_n$, such that $(v_i, a_i, v_{i+1}) \in E$ for each $1 \leq i < n$. The set of *positions* of ρ is $\{1, \dots, n\}$, and v_i is the node in position i of ρ, for $1 \leq i \leq n$. The intuition behind the semantics of WL formulas is as follows. Each path variable π is interpreted as a path $\rho = v_1 a_1 v_2 \cdots v_{n-1} a_{n-1} v_n$ in the data graph G, while each position variable t of sort π is interpreted as a position $1 \leq i \leq n$ in ρ (that is, position variables of sort π are interpreted as positions in the path that interprets π). The atomic formula $E_a(t_1^\pi, t_2^\pi)$ is true iff π is interpreted as path $\rho = v_1 a_1 v_2 \cdots v_{n-1} a_{n-1} v_n$, the position p_2 that interprets t_2

in ρ is the successor of the position p_1 that interprets t_1 (i.e. $p_2 = p_1 + 1$), and node in position p_1 is linked in ρ by an a-labeled edge to node in position p_2 (that is, $a_{p_1} = a$). In the same way, $t_1^\pi < t_2^\pi$ holds iff in the path ρ that interprets π the position that interprets t_1 is smaller than the one that interprets t_2. Furthermore, $t_1 \sim t_2$ is the case iff the data value carried by the node in the position assigned to t_1 is the same than the data value carried by the node in the position assigned to t_2 (possibly in different paths). We formalize the semantics of WL below.

Let $G = (V, E, \kappa)$ be a data graph and ϕ a WL formula. Assume that $\mathcal{S}_\phi$ is the set that consists of (i) all position variables t^π and path variables π such that t^π is a free variable of ϕ, and (ii) all path variables π such that π is a free variable of ϕ. Intuitively, $\mathcal{S}_\phi$ defines the set of (both path and position) variables that are relevant to define the semantics of ϕ over G. An *assignment* α for ϕ over G is a mapping that associates a path $\rho = v_1 a_1 v_2 \cdots v_{n-1} a_{n-1} v_n$ in G with each path variable $\pi \in \mathcal{S}_\phi$, and a position $1 \leq i \leq n$ with each position variable of the form t^π in $\mathcal{S}_\phi$ (notice that this is well-defined since $\pi \in S_\phi$ every time a position variable of the form t^π is in S_ϕ). As usual, we denote by $\alpha[t \to i]$ and $\alpha[\pi \to \rho]$ the assignments that are equal to α except that t is now assigned position i and π the path ρ, respectively.

We say that G *satisfies* ϕ *under* α, denoted $(G, \alpha) \models \phi$, if one of the following holds (we omit Boolean combinations which are standard):

- $\phi = E_a(t_1^\pi, t_2^\pi)$, the path $\alpha(\pi)$ is $v_1 a_1 v_2 \cdots v_{n-1} a_{n-1} v_n$, and it is the case that $\alpha(t_2^\pi) = \alpha(t_1^\pi) + 1$ and $a = a_{\alpha(t_1^\pi)}$.
- $\phi = t_1^\pi < t_2^\pi$ and $\alpha(t_1^\pi) < \alpha(t_2^\pi)$.
- $\phi = (t_1 \sim t_2)$, t_1 is of sort π_1, t_2 is of sort π_2, and $\kappa(v_1) = \kappa(v_2)$, where v_i is the node in position $\alpha(t_i)$ of $\alpha(\pi_i)$, for $i = 1, 2$.
- $\phi = \exists t^\pi \psi$ and there is a position i in $\alpha(\pi)$ such that $(G, \alpha[t^\pi \to i]) \models \psi$.
- $\phi = \exists \pi \psi$ and there is a path ρ in G such that $(G, \alpha[\pi \to \rho]) \models \psi$.

Example 1. A simple example from [7] that shows that WL expresses NP-complete properties is the following query that checks if a graph has a Hamiltonian path:

$$\exists \pi \left(\forall t_1^\pi \forall t_2^\pi \, (t_1^\pi \neq t_2^\pi \to t_1^\pi \not\sim t_2^\pi) \wedge \forall \pi' \forall t_1^{\pi'} \exists t_2^\pi (t_1^{\pi'} \sim t_2^\pi) \right).$$

In fact, this query expresses that there is a path π that does not repeat nodes (because π satisfies $\forall t_1^\pi \forall t_2^\pi (t_1^\pi \neq t_2^\pi \to t_1^\pi \not\sim t_2^\pi)$), and every node belongs to such path (because π satisfies $\forall \pi' \forall t_1^{\pi'} \exists t_2^\pi (t_1^{\pi'} \sim t_2^\pi)$, and, thus, every node that occurs in some path π' in the graph database also occurs in π). □

4 WL Evaluation is Non-elementary in Data Complexity

In this section we pinpoint the precise complexity of query evaluation for WL. It was proven in [7] that this problem is decidable. Although the precise complexity of this problem was left open in [7], one can prove that this is, in fact, a non-elementary problem by an easy translation from the satisfiability problem for FO formulas – which is known to be non-elementary [13, 14]. In databases,

however, one is often interested in a different measure of complexity – called *data complexity* [15] – that assumes the formula ϕ to be fixed. This is a reasonable assumption since databases are usually much bigger than formulas. Often in the setting of data complexity the cost of evaluating queries is much smaller than in the general setting in which formulas are part of the input. The main result of this section is that the data complexity of evaluating WL formulas is nonelementary even *over graph databases*, which rules out its practicality.

Theorem 1. *The evaluation problem for WL is non-elementary in data complexity. In particular, for each $k \in \mathbb{Z}_{>0}$, there is a finite alphabet Σ and a Boolean formula ϕ over Σ, such that the problem* Eval*(WL,ϕ) of evaluating the WL formula ϕ is k-*ExpSpace*-hard. In addition, the latter holds even if the input is restricted to the class of graph databases.*

Proof (Sketch): We start by sketching the case $k = 1$ here, which provides insightful technical details about the nature of the proof. There is a Turing machine M such that the following problem is ExpSpace-hard: given a word w of size n, is there an accepting run of M over w using at most 2^{cn} cells? We prove that there is a formula $\phi \in$ WL of size polynomial in the size of M such that for all words w of size n, we can compute a graph G_w such that

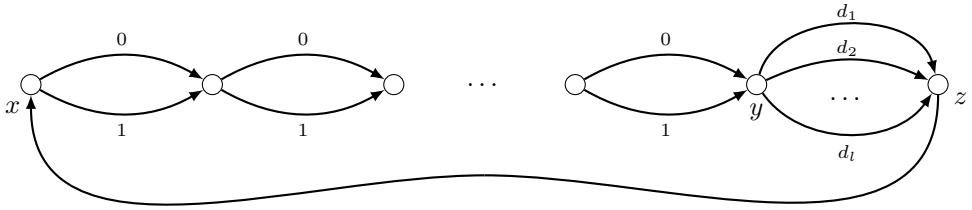
$$G_w \vDash \phi \quad \text{iff} \quad \text{there is an accepting run of } M \text{ over } w \text{ using} \leq 2^{cn} \text{ cells.} \qquad (1)$$

The formula ϕ is of the form $\exists\pi\psi(\pi)$, where ψ is a formula that does not contain any quantification over path variables. Given a word w of size n, the label of the path π in the graph G_w will encode an accepting run of M over the word w in the following way. Suppose that in a configuration C, the content of the tape is the word $a_1 \dots a_{2^{cn}}$, the head is scanning cell number j_0 and the state is q_0. The configuration C is encoded by the word e_C defined by

$$c(0)(\$, a_0) \dots c(j_0 - 1)(\$, a_{j_0-1})c(j_0)(q_0, a_{j_0})c(j_0 + 1)(\$, a_{j_0+1}) \dots c(2^n)(\$, a_{2^{cn}}),$$

where $c(j)$ is the binary representation of the number j. The pair $c(j)(q_j, a_j)$ (where $q_j = q_0$ if $j = j_0$ and $q_j = \$$ otherwise) is the *description* of cell number j in C. A run $C_0C_1 \dots$ is encoded as the word $e_{C_0}e_{C_1} \dots$.

We think of a path π encoding a run as consisting of two parts: the first part contains the encoding e_{C_0} of the initial configuration and is a path through a subgraph I_w of G_w, while the second part contains the encoding $e_{C_1}e_{C_2} \dots$ and is a path through the subgraph H_w of G_w. If Q is the set of states of M and Γ is the alphabet, we define H_w as the following graph



where $\{d_j : 1 \leq j \leq l\} = (Q \cup \{\$\}) \times \Gamma$ and the number of nodes with outgoing edges with labels 0 and 1 is equal to cn. The label of a path π' from the "left-most" node x to the "right-most" node z with only once occurrence of x is exactly the description of a cell in a configuration: it is the binary encoding of a number $\leq 2^{cn}$ followed by a pair of the form (q', a). We can define a formula $\phi_C \in \mathrm{WL}$ such that for all paths π starting in x and ending in z,

$$H_w \vDash \phi_C(\pi) \quad \text{iff} \quad \text{the label of } \pi \text{ is the encoding of a configuration.}$$

We do not give details; ϕ_C has to express that the first number encoded in binary is 0, that the last number is 2^{cn} and that the encoding of the description of cell number j is followed by the description of cell number $j+1$. Using the formula ϕ_C, we can define a formula ϕ_1 such that for all paths π,

$$H_w \vDash \phi_1(\pi) \quad \text{iff} \quad \text{the label of } \pi \text{ is the encoding of an accepting run.}$$

The formula ϕ_1 has to ensure that if $e_C e_{C'}$ occurs in the label of π, then C and C' are consecutive configurations according to M. Moreover, ϕ_1 has to express that eventually we reach the final state. In order to express ϕ_C and ϕ_1, we use the ability of WL to check whether two positions correspond to the same node. For example, in order to define ϕ_1, since we need to compare consecutive configurations e_C and $e_{C'}$, we need to be able to compare the content of a cell in configuration C and the content of that same cell in C'. In particular, we want to be able to express whether two subpaths π'_0 and π'_1 of π starting in x and ending in y correspond to the binary encoding of the same number. Since the length of such subpaths depends on n, we cannot check node by node whether the two subpaths are equal. However, it is sufficient to check that if $t_0^{\pi'_0}$ and $t_1^{\pi'_1}$ corresponds to the same node ($t_0^{\pi'_0} \sim t_1^{\pi'_1}$), then their successors also correpond to the same node ($t_0^{\pi'_0} + 1 \sim t_1^{\pi'_1} + 1$). Similarly, in the formula ϕ_C, we use the operator $\sim$ in order to express that two subpaths correspond to the binary encodings of numbers that are successors of each other.

Similarly to the way we define the graph H_w, we can introduce a graph I_w and a formula $\phi_0(\pi)$ such that

$$I_w \vDash \phi_0(\pi) \quad \text{iff} \quad \text{the label of } \pi \text{ is the encoding } e_{C_0},$$

where C_0 is the initial configuration of the run of M over w. By adding an "adequate edge" from I_w to H_w, we construct a graph G_w such that for all paths π, $G_w \vDash \phi_0(\pi) \wedge \phi_1(\pi)$ iff the label of π is the encoding of an accepting run over w. Hence, the formula $\phi := \exists\pi(\phi_0(\pi) \wedge \phi_1(\pi))$ satisfies (1).

For the case where $k > 1$, the problem to adapt the above proof is that we have to consider runs using a number of cells that is bound by a tower of exponentials of height k. If $k > 1$, the binary representation of such a bound is not polynomial. The trick is to represent such exponential towers by k-counters. A 1-counter is the binary representation of a number. If $k > 1$, a k-counter c is a word $\sigma_0 l_0 \ldots \sigma_{j_0} l_{j_0}$, where l_j is a $(k-1)$-counter and $\sigma_j \in \{0, 1\}$. The

counter c represents the number $r(c) = \sum_{j=0}^{j_0} \sigma_j r(\sigma_j)$. In particular, a tower of exponentials of height k is represented by a k-counter of polynomial size.

We can show that there are a graph F_k and a formula $\chi_k(\pi)$ such that the label of π is a k-counter iff $F_k \vDash \chi_k(\pi)$. Using F_k and χ_k, we can then adapt the above proof to the cases where $k > 1$. □

As a corollary to the proof of Theorem 1, we obtain that data complexity is non-elementary even for simple WL formulas that talk about a single path in a graph database.

Corollary 1. *The evaluation problem for WL over graph databases is non-elementary in data complexity, even if restricted to Boolean WL formulas of the form $\exists\pi\psi$, where ψ uses no path quantification and contains no position variable of sort different that π.*

5 Register Logic

We saw in the previous section that WL is impractical due to its very high data complexity. In this section, we start by recalling the notion of regular expressions with memory (REM) and their basic results from [9]. The problem with this logic though is its limitation in expressive power. For instance, the query (Q) from the introduction cannot be expressed in REM. We then introduce an extension of REM, called regular logic (RL), that remedies this limitation in expressive power (in fact, it can express many natural examples of queries expressible in WL, e.g., those given in [7]) while retaining elementary complexity of query evaluation. Finally, we study which fragments of RL are well-behaved for database applications.

5.1 Regular expressions with memory

REMs define pairs of nodes in data graphs that are linked by a path that satisfies a constraint in the way in which the topology interacts with the underlying data. REMs allow to specify when data values are remembered and used. Data values are stored in k registers $r_1, \ldots, r_k$. At any point we can compare a data value with one previously stored in the registers. As an example, consider the REM $\downarrow r.a^+[r^=]$. It can be read as follows: Store the current data value in register r, and then check that after reading a word in a^+ we see the same data value again (condition $[r^=]$). We formally define REM next.

Let $r_1, \ldots, r_k$ be registers. The set of *conditions* c over $\{r_1, \ldots, r_k\}$ is recursively defined as: $c := r_i^= \mid c \wedge c \mid \neg c$, for $1 \leq i \leq k$. Assume that $\mathcal{D}_\perp$ is the extension of the set $\mathcal{D}$ of data values with a new symbol $\perp$. Satisfaction of conditions is defined with respect to a value $d \in \mathcal{D}$ (the data value that is currently being scanned) and a tuple $\tau = (d_1, \ldots, d_k) \in \mathcal{D}_\perp^k$ (the data values stored in the registers, assuming that $d_i = \perp$ represents the fact that register r_i has no value assigned) as follows (Boolean combinations omitted): $(d, \tau) \models r_i^=$ iff $d = d_i$.

Definition 3 (REMs). *The class of REMs over Σ and $\{r_1, \ldots, r_k\}$ is defined by the grammar:*

$$e \;:=\; \varepsilon \mid a \mid e \cup e \mid e \cdot e \mid e^+ \mid e[c] \mid \downarrow \bar{r}.e$$

where a ranges over symbols in Σ, c over conditions over $\{r_1, \ldots, r_k\}$, and $\bar{r}$ over tuples of elements in $\{r_1, \ldots, r_k\}$.

That is, REM extends the class of regular expressions e – which is a popular mechanism for specifying topological properties of paths in graph databases (see, e.g., [16, 2]) – with expressions of the form $e[c]$, for c a condition, and $\downarrow \bar{r}.e$, for $\bar{r}$ a tuple of registers – that define how such topology interacts with the data.

Semantics: To define the evaluation $e(G)$ of an REM e over a data graph $G = (V, E, \kappa)$, we use a relation $[\![e]\!]_G$ that consists of tuples of the form $(u, \lambda, \rho, v, \lambda')$, for u, v nodes in V, ρ a path in G from u to v, and λ, λ' two k-tuples over $\mathcal{D}_\perp$. The intuition is the following: the tuple $(u, \lambda, \rho, v, \lambda')$ belongs to $[\![e]\!]_G$ if and only if the data and topology of ρ can be parsed according to e, with λ being the initial assignment of the registers, in such a way that the final assignment is λ'. We then define $e(G)$ as the pairs (u, v) of nodes in G such that $(u, \perp^k, \rho, v, \lambda) \in [\![e]\!]_G$, for some path ρ in G from u to v and k-tuple λ over $\mathcal{D}_\perp$.

We inductively define relation $[\![e]\!]_G$ below. We assume that $\lambda_{\bar{r}=d}$, for $d \in \mathcal{D}$, is the tuple obtained from λ by setting all registers in $\bar{r}$ to be d. Also, if $\rho_1 = v_1 a_1 v_2 \cdots v_{k-1} a_{k-1} v_k$ and $\rho_2 = v_k a_k v_{k+1} \cdots v_{n-1} a_{n-1} v_n$ are paths, then $\rho_1 \rho_2$ is the path $v_1 a_1 v_2 \cdots v_{k-1} a_{k-1} v_k a_k v_{k+1} \cdots v_{n-1} a_{n-1} v_n$. Then:

- $[\![\varepsilon]\!]_G = \{(u, \lambda, \rho, u, \lambda) \mid u \in V,\ \rho = u,\ \lambda \in \mathcal{D}_\perp^k\}$.
- $[\![a]\!]_G = \{(u, \lambda, \rho, v, \lambda) \mid \rho = uav,\ \lambda \in \mathcal{D}_\perp^k\}$.
- $[\![e_1 \cup e_2]\!]_G = [\![e_1]\!]_G \cup [\![e_2]\!]_G$.
- $[\![e_1 \cdot e_2]\!]_G = [\![e_1]\!]_G \circ [\![e_2]\!]_G$, where $[\![e_1]\!]_G \circ [\![e_2]\!]_G$ is the set of tuples $(u, \lambda, \rho, v, \lambda')$ such that $(u, \lambda, \rho_1, w, \lambda'') \in [\![e_1]\!]_G$ and $(w, \lambda'', \rho_2, v, \lambda') \in [\![e_2]\!]_G$, for some $w \in V$, k-tuple λ'' over $\mathcal{D}_\perp$, and paths ρ_1, ρ_2 such that $\rho = \rho_1 \rho_2$.
- $[\![e^+]\!]_G = [\![e]\!]_G \cup ([\![e]\!]_G \circ [\![e]\!]_G) \cup ([\![e]\!]_G \circ [\![e]\!]_G \circ [\![e]\!]_G) \ldots$
- $[\![e[c]]\!]_G = \{(u, \lambda, \rho, v, \lambda') \in [\![e]\!]_G \mid (\kappa(v), \lambda') \models c\}$.
- $[\![\downarrow \bar{r}.e]\!]_G = \{(u, \lambda, \rho, v, \lambda') \mid (u, \lambda_{\bar{r}=\kappa(u)}, \rho, v, \lambda') \in [\![e]\!]_G\}$.

For each REM e, we will use the shorthand notation e^* to denote $\varepsilon \cup e^+$.

Example 2. The REM $\Sigma^* \cdot (\downarrow r.\Sigma^+[r^=]) \cdot \Sigma^*$ defines the pairs of nodes that are linked by a path in which two nodes have the same data value. The REM $\downarrow r.(a[\neg r^=])^+$ defines the pairs of nodes that are linked by a path ρ with label in a^+, such that the data value of the first node in the path is different from the data value of all other nodes in ρ. □

The problem Eval(REM) is, given a data graph $G = (V, E, \kappa)$, a pair (v_1, v_2) of nodes in V, and an REM e, is $(v_1, v_2) \in e(G)$? The data complexity of the problem refers again to the case when ϕ is considered to be fixed. REMs are tractable in data complexity and have no worst combined complexity than FO over relational databases:

Proposition 1 ([9]). EVAL(REM) *is* PSPACE*-complete, and in* NLOGSPACE *in data complexity.*

5.2 Register logic

REM is well-behaved in terms of the complexity of evaluation, but its expressive power is rather rudimentary for expressing several data/topology properties of interest in data graphs. As an example, the query (Q) from the introduction – which can be easily expressed in WL – cannot be expressed as an REM (we actually prove a stronger result later). The main shortcomings of REM in terms of its expressive power are its inability to (i) compare data values in different paths and (ii) express branching properties of the data.

In this section, we propose register logic (RL) as a natural extension of REM that makes up for this lack of expressiveness. We borrow ideas from the logic CRPQ$^{\neg}$, presented in [4], that closes the class of *regular path queries* [6] under Boolean combinations and existential *node* and *path* quantification. In the case of RL we start with REMs and close them not only under Boolean combinations and node and path quantification – which allow to express arbitrary patterns over the data – but also under *register assignment* quantification – which permits to compare data values in different paths. We also prove that the complexity of the evaluation problem for RL is elementary (EXPSPACE), and, thus, that in this regard RL is in stark contrast with WL.

To define RL we assume the existence of countably infinite sets of *node*, *path* and *register assignment variables*. Node variables are denoted $x, y, z, \ldots$, path variables are denoted $\pi, \pi', \pi_1, \pi_2, \ldots$, and register assignment variables are denoted $\nu, \nu_1, \nu_2, \ldots$

Definition 4 (Register logic (RL)). *We define the class of RL formulas ϕ over alphabet Σ and $\{r_1, \ldots, r_k\}$ using the following grammar:*

$$\begin{aligned} \texttt{atom} &:= x = y \mid \pi = \pi' \mid \nu = \nu' \mid \nu = \bar{\perp} \mid (x, \pi, y) \mid e(\pi, \nu_1, \nu_2) \\ \phi &:= \texttt{atom} \mid \neg\phi \mid \phi \vee \phi \mid \exists x\phi \mid \exists \pi\phi \mid \exists \nu\phi \end{aligned}$$

Here x, y are node variables, π, π' are path variables, ν, ν' are register assignment variables, and e is an REM over Σ and $\{r_1, \ldots, r_k\}$.

Intuitively, $\nu = \bar{\perp}$ holds iff ν is the empty register assignment, (x, π, y) checks that π is a path from x to y, and $e(\pi, \nu, \nu')$ checks that π can be parsed according to e starting from register assignment ν and finishing in register assignment ν'. The quantifier $\exists\nu$ is to be read "there exists an assignment of data values in the data graph to the registers".

Let $G = (V, E, \kappa)$ be a data graph over Σ and ϕ a RL formula over Σ and $\{r_1 \ldots, r_k\}$. Assume that D is the set of data values that are mentioned in G, i.e., $D = \{\kappa(v) \mid v \in V\}$. An *assignment* α for ϕ over G is a mapping that assigns (i) a node in V to each free node variable x in ϕ, (ii) a path ρ in G to each free path variable π in ϕ, and (iii) a tuple λ in $(D \cup \{\perp\})^k$ to each register variable ν

that appears free in ϕ. That is, for safety reasons we assume that $\alpha(\nu)$ can only contain data values that appear in the underlying data graph. This represents no restriction for the expressiveness of the logic.

We inductively define $(G, \alpha) \models \phi$, for G a data graph, ϕ a RL formula, and α an assignment for ϕ over G, as follows (we omit equality atoms and Boolean combinations since they are standard):

- $(G, \alpha) \models \nu = \bar{\perp}$ iff $\alpha(\nu) = \perp^k$.
- $(G, \alpha) \models (x, \pi, y)$ iff $\alpha(\pi)$ is a path from $\alpha(x)$ to $\alpha(y)$ in G.
- $(G, \alpha) \models e(\pi, \nu, \nu')$ iff $(u, \alpha(\nu), \alpha(\pi), v, \alpha(\nu')) \in [\![e]\!]_G$, assuming $\alpha(\pi)$ goes from node u to v.
- $(G, \alpha) \models \exists x \phi$ iff there is node $v \in V$ such that $(G, \alpha[x \to v]) \models \phi$.
- $(G, \alpha) \models \exists \pi \phi$ iff there is path ρ in G such that $(G, \alpha[\pi \to \rho]) \models \phi$.
- $(G, \alpha) \models \exists \nu \phi$ iff there is tuple λ in $(D \cup \{\perp\})^k$ such that $(G, \alpha[\nu \to \lambda]) \models \phi$.

Thus, each REM e is expressible in RL using the formula:

$$\exists \pi \exists \nu \exists \nu' \, (\, \nu = \bar{\perp} \, \wedge \, e(\pi, \nu, \nu') \,).$$

Example 3. Recall query (Q) from the introduction: *Find pairs of nodes x and y in a graph database, such that there is a node z and a path π from x to y in which each node is connected to z.* This query can be expressed in RL over $\Sigma = \{a\}$ and a single register r as follows:

$$\exists \pi \, \big(\, (x, \pi, y) \wedge \exists z \forall \nu (e_1(\pi, \nu, \nu) \to \exists z' \exists \pi' ((z', \pi', z) \wedge e_2(\pi', \nu, \nu))) \, \big),$$

where $e_1 := a^*[r^=] \cdot a^*$ is the REM that checks whether the node (i.e. data) stored in register r appears in a path, and $e_2 := \varepsilon[r^=] \cdot a^*$ is the REM that checks if the first node of a path is the one that is stored in register r.

In fact, this formula defines the pairs of nodes x and y such that there exists a path π that goes from x to y and a node z for which the following holds: for every register value ν (i.e., for every node ν) such that $e_1(\pi, \nu, \nu)$ (i.e. node ν is in π), it is the case that there is a path π' from some node z' to z such that $e_2(\pi', \nu, \nu)$ (i.e., $z' = \nu$ and π' connects ν to z). Notice that this uses the fact that the underlying data model is that of graph databases, in which each node is uniquely identified by its data value. □

Complexity of evaluation for RL: The evaluation problem for RL, denoted Eval(RL), is as follows: Given a data graph G, a RL formula ϕ, and an assignment α for ϕ over G, is it the case that $(G, \alpha) \models \phi$? As before, we denote by Eval(RL,ϕ) the evaluation problem for the fixed RL formula ϕ.

We show next that, unlike WL, register logic RL can be evaluated in elementary time, and, actually, with only one exponential jump over the complexity of evaluation of REMs:

Theorem 2. Eval(RL) *is* Expspace*-complete. The lower bound holds even if the input is restricted to graph databases.*

Proof (Idea): For the upper bound, we adapt for RL the proof that CRPQ$^\neg$ formulas can be evaluated in PSPACE [4]. This requires some care in the way in which register values and atomic REM formulas are handled. The extra exponential blow up is produced by the fact that checking whether a path ρ in a data graph G does not satisfy an REM e (i.e. whether it is not the case that $(u, \bar{\perp}, \rho, v, \lambda) \in [\![e]\!]_G$, for some register assignment λ, assuming that ρ goes from u to v) requires exponential space. The lower bound is obtained by a reduction from the acceptance problem for a Turing machine that works in EXPSPACE. □

The increase in expressiveness of RL over REM has an important cost in data complexity, which becomes intractable:

Theorem 3. EVAL(RL) *is in* PSPACE *in data complexity. Furthermore, there is a finite alphabet Σ and a RL formula ϕ over Σ and a single register r, such that* EVAL(RL,ϕ) *is* PSPACE*-hard. In addition, the latter holds even if the input is restricted to graph databases.*

In the next section we introduce an interesting language, based on a restriction of RL, that is tractable in data complexity, and thus better suited for database applications. This language is a proper extension of REM. But before, we make some important remarks about the expresive power of RL.

Expressive power of RL: We now look at the expressive power of the logic RL. It was proven in [7] that CRPQ is not subsumed in WL. Since RL subsumes CRPQ$^\neg$, it follows that RL is not subsumed in WL. On the other hand, WL is also not subsumed in RL due to Theorem 1, Theorem 2, and the standard time/space hierarchy theorem from complexity theory. Therefore, we have the following proposition:

Proposition 2. *The expressive powers of WL and RL are incomparable.*

On the other hand, we shall argue now that many natural queries about the interaction between data and topology are also expressible in RL. The aforementioned query (Q) is one such example. We shall now mention other examples: hamiltonicity (H), the existence of Eulerian trail (E), bipartiteness (B), and complete graphs with even number of nodes (C2). The first two are expressible in WL, while (B) and (C2) are not known to be expressible in WL. We conjecture that they are not.

We now show how to express in RL the existence of a hamiltonian path in a graph; the query (E) can be expressed in the same way but with two registers (to remember edges, i.e., consisting of two nodes). This is done with the following formula over $\Sigma = \{a\}$ and a single register r:

$$\exists\pi \left(\forall\lambda\forall\lambda' \neg e_1(\pi, \lambda, \lambda') \wedge \forall\lambda(\lambda \neq \perp \rightarrow e_2(\pi, \lambda, \lambda)) \right),$$

where $e_1 := a^* \cdot (\downarrow r.a^+[r^=]) \cdot a^*$ is the REM that checks whether in a path some node is repeated (i.e., that it is not a simple path), and $e_2 := a^*[r^=]a^*$ is the REM that checks that the node stored in register r appears in a path. In fact,

this query expresses that there is a path π that it is simple (as expressed by the formula $\forall\lambda\forall\lambda'\neg e_1(\pi,\lambda,\lambda')$), and every node of the graph database is mentioned in π (as expressed by the formula $\forall\lambda(\lambda \neq \perp \rightarrow e_2(\pi,\lambda,\lambda))$).

We now show how to express in RL the property bipartiteness from graph theory. An undirected $G = (V, E)$ is *bipartite* if its set of nodes can be partitioned into two sets S_1 and S_2 such that, for each edge $(v, w) \in E$, either (i) $v \in S_1$ and $w \in S_2$, or (ii) $v \in S_2$ and $w \in S_2$. It is well-known that a graph database is bipartite iff it does not have cycles of odd length. The latter is expressible in RL since the existence of an odd-length cycle can be expressed as $\exists\pi\exists\lambda\exists\lambda' e(\pi,\lambda,\lambda')$, where $e = \downarrow r.a(aa)^*[r^=]$.

We now show how to express in RL that a graph database is a complete graph with an even number of nodes. To this end, it is sufficient and necessary to express the existence of a hamiltonian path π with an odd number of edges in the graph. But this is a simple modification of our formula for expressing hamiltonicity: we add the check that π has an odd number of edges by adding the conjunct $e(\pi,\nu,\nu')$, where $e = a \cup a(aa)^+$, and close the entire formula under existential quantification of ν and ν'.

5.3 Tractability in data complexity

Let RL^+ be the positive fragment of RL (i.e. the logic obtained from RL by forbidding negation and adding conjunctions). It is easy to prove that the data complexity of the evaluation problem for RL^+ is tractable (NLOGSPACE). This fragment contains the class of *conjunctive* REMs, that has been previously identified as tractable in data complexity [9]. However, the expressive power of RL^+ is limited as the following proposition shows.

Proposition 3. *The query* (Q) *from the introduction is not expressible in* RL^+.

On the other hand, increasing the expressive power of RL^+ with some simple forms of negation leads to intractability of query evaluation in data complexity:

Proposition 4. *There is a finite alphabet* Σ *and REMs* e_1, e_2, e_3, e_4 *over* Σ *and a single register* r*, such that* EVAL(RL,ϕ) *is* PSPACE*-complete, where* ϕ *is either* $\exists\pi\exists\lambda\neg(e_1(\pi,\perp,\lambda) \vee e_2(\pi,\perp,\perp))$ *or* $\exists\pi\forall\lambda\neg(e_3(\pi,\perp,\lambda) \vee e_4(\pi,\perp,\perp))$.

In the case of basic navigational languages for graph databases, it is possible to increase the expressive power – without affecting the cost of evaluation – by extending formulas with a branching operator (in the style of the class of *nested regular expressions* [3]). The same idea can be applied in our scenario, by extending atomic REM formulas in RL^+ with such branching operator. The resulting language is more expressive than RL^+ (in particular, this extension can express query (Q)), yet remains tractable in data complexity. We formalize this idea below.

The class of *nested* REMs (NREM) extends REM with a nesting operator $\langle\cdot\rangle$ defined as follows: If e is an NREM then $\langle e\rangle$ is also an NREM. Intuitively, the formula $\langle e\rangle$ filters those nodes in a data graph that are the origin of a path that

can be parsed according to e. Formally, if e is an NREM over k registers and G is a data graph, then $[\![\langle e \rangle]\!]_G$ consists of all tuples of the form $(u, \lambda, \rho = u, u, \lambda)$ such that $(u, \lambda, \rho', v, \lambda') \in [\![e]\!]_G$, for some node v in G, path ρ' in G, and k-tuple λ' over $\mathcal{D}_\perp$.

Let NRL$^+$ be the logic that is obtained from RL$^+$ by allowing atomic formulas of the form $e(\pi, \nu, \nu')$, for e an NREM. Given a data graph G and an assignment α for π, ν and ν' over G, we write as before $(G, \alpha) \models e(\pi, \nu, \nu')$ if and only if $\alpha(\pi)$ goes from u to v and $(u, \alpha(\nu), \alpha(\pi), v, \alpha(\nu')) \in [\![e]\!]_G$. The semantics of NRL$^+$ is thus obtained from the semantics of these atomic formulas in the expected way. The following example shows that query (Q) is expressible in NRL$^+$, and, therefore, that NRL$^+$ increases the expressiveness of RL$^+$.

Example 4. Over graph databases, the query (Q) from the introduction is expressible in NRL$^+$ using the following formula over $\Sigma = \{a\}$ and register r:

$$\phi \;=\; \exists\pi\exists\nu \big((x, \pi, y) \wedge e(\pi, \nu, \nu) \big),$$

where $e := (\langle e_1 \rangle \cdot a)^* \langle e_1 \rangle$, for $e_1 = a^*[r^=]$. Intuitively, e_1 checks in a path whether its last node is precisely the node stored in register r, and thus e checks whether every node in a path can reach the node stored in register r. Therefore, the formula ϕ defines the set of pairs (x, y) of nodes, such that there is a path π that goes from x to y and a register value ν (i.e., a node ν) that satisfy that every node in π is connected to ν. □

The extra expressive power of NRL$^+$ over RL$^+$ does not affect the data complexity of query evaluation:

Theorem 4. *Evaluation of NRL$^+$ formulas can be solved in* NLogspace *in data complexity.*

From the proof of Theorem 4 it also follows that NRL$^+$ formulas can be evaluated in Pspace in combined complexity.

6 Conclusions and Future Work

We have proven that the data complexity of walk logic is nonelementary, which rules out the practicality of the logic. We have proposed register logic, which is an extension of regular expressions with memory. Our results in this paper suggest that register logic is capable of expressing natural queries about interactions between data and topology in data graphs, while still preserving the elementary data complexity of query evaluation (Pspace). Finally, we showed how to make register logic more tractable in data complexity (NLogspace) through the logic NRL$^+$, while at the same time preserving some level of expressiveness of RL.

We leave open several problems for future work. One interesting question is to study the expressive power of extensions of walk logic, in comparison to RL and ECRPQ$^\neg$ from [4]. For example, we can consider extensions with regularity tests (i.e. an atomic formula testing whether a path belongs to a regular language).

Even in this simple case, the expressive power of the resulting logic, compared to RL and ECRPQ$^{\neg}$, is already not obvious. Secondly, we do not know whether NRL^{+} is strictly more expressive than RL. Finally, we will also mention that expressibility of bipartiteness in WL is still open (an open question from [7]). We also leave open whether the query that a graph database is a complete graph with an even number of nodes is expressible in WL.

Acknowledgements: We thank the reviewers for the useful comments. Barceló is funded by Fondecyt grant 1130104, Fontaine by Fondecyt postdoctoral grant 3130491, and Lin by EPSRC (EP/H026878/1). Part of this work was done when Lin visited Barceló funded by Fondecyt grant 1130104.

References

1. R. Angles, C. Gutiérrez. Survey of graph database models. *ACM Comput. Surv.* 40(1), 2008.
2. P. Barceló. Querying graph databases. In *PODS* 2013, pages 175-188.
3. P. Barceló, J. Pérez, J. L. Reutter. Relative expressiveness of nested regular expressions. In *AMW* 2012, pages 180-195.
4. P. Barceló, L. Libkin, A. W. Lin, P. T. Wood. Expressive languages for path queries over graph-structured data. *ACM Trans. Database Syst.* 37(4): 31, 2012.
5. E. M. Clarke, O. Grumberg, D.A. Peled. *Model checking*. MIT Press, 2000.
6. I. Cruz, A. O. Mendelzon, P. T. Wood. A graphical query language supporting recursion. In *SIGMOD* 1987, 323-330.
7. J. Hellings, B. Kuijpers, J. van den Bussche, X. Zhang. Walk logic as a framework for path query languages on graph databases. In *ICDT* 2013, pages 117-128.
8. M. Kaminski, N. Francez. Finite memory automata. *TCS*, 134(2), pages 329-363, 1994.
9. L. Libkin, D. Vrgoč. Regular path queries on graphs with data. In *ICDT* 2012, pages 74-85.
10. L. Libkin, D. Vrgoč. Regular expressions for data words. In *LPAR* 2012, pages 274-288.
11. L. Libkin, J. L. Reutter, D. Vrgoc. Trial for RDF: adapting graph query languages for RDF data. In *PODS* 2013, pages 201-212.
12. A. O. Mendelzon, P. T. Wood. Finding regular simple paths in graph databases. *SIAM J. Comput.*, 24(6):1235–1258, 1995.
13. L.E. Robertson. Structure of complexity in the weak monadic second-order theories of the natural numbers. In *STOC* 1974, pages 161-171.
14. L. Stockmeyer. The complexity of decision problems in automata theory and logic. Ph.D. thesis, MIT, 1974.
15. M. Y. Vardi. The complexity of relational query languages. In *STOC* 1982, pages 137-146.
16. P. T. Wood. Query languages for graph databases. *SIGMOD Record* 41(1), pages 50-60, 2012.