Querying Data Graphs with Arithmetical Regular Expressions

Maciej Graboń and Jakub Michaliszyn and Jan Otop and Piotr Wieczorek
Institute of Computer Science, University of Wrocław

Abstract

We propose a query language LARE for graphs whose edges are labelled by a finite alphabet and nodes store unbounded data values. LARE is based on a variant of regular expressions with memory. Queries of LARE can compare quantities of memorised graph nodes and their neighbourhoods. These features allow us to express a number of natural properties, while keeping the data complexity (with a query fixed) in non-deterministic logarithmic space. We establish an algorithm that works efficiently not only with LARE, but also with a wider language defined using effective relational conditions, another formalism we propose.

1 Introduction

Recently, there has been a growing interest in graphs as means of representing data (see surveys [Barceló, 2013; Angles and Gutierrez, 2008]), where beside querying the stored data, one can reason about the links among the data.

In these applications, the graph databases tend to be too big to fit in the modern computers’ memory. Therefore, a typical criterion of feasibility of a query language is having data complexity in NL, the class of problems solvable in non-deterministic logarithmic space [Calvanese et al., 2006; Artale et al., 2007; Barceló et al., 2012]. Checking whether there is a path between two given nodes is already NL-complete, so NL is the best complexity we can obtain for any reasonable language.

Our contribution. We propose a query language, called LARE, which subsumes several previous formalisms, allows to express new interesting properties and at the same time keeps data complexity of query answering in NL.

LARE allows for writing queries against both nodes and paths given as input. A query can existentially quantify nodes and paths and check relationship between many paths using relational conditions defined by arithmetic regular expressions (ARE), which are regular expressions with registers that allow for various arithmetic comparisons between registers as well as for nesting. The main innovation of the LARE queries lays in the ability to express various arithmetic and aggregative properties of nodes and paths, hence we assume natural numbers as the data values. Nevertheless, our approach can be adjusted to different data domains.

LARE is powerful enough to find, for example, nodes \( s, t \) such that there is a path \( p \) from \( s \) to \( t \) such that \( p \) visits a node with the maximal value in the graph and the total sum of all the elements of \( p \) is in a given interval. LARE allows for using nested queries and their negation. This facilitates formulation of properties such as there is a one-way path from \( s \) to \( t \), i.e. a path in which for any two consecutive nodes \( v, v' \), there is no path from \( v' \) to \( v \). Properties like these occur naturally in reasoning about multi-agent systems. Agents are often represented by a Kripke structure, which is basically a labelled graph with distinguished initial states. The considered Kripke structures are typically very large due to the state explosion problem. The query answering algorithm we propose works in (non-deterministic) logarithmic space in size of the generated Kripke structure, but can be easily adjusted to reason in polynomial space in size of a succinct representation of a multiagent system, i.e., without generating the exponentially large Kripke structure. Further examples are in Section 5.

We associate with each graph a separate finite automaton, which recognizes the paths of this graph satisfying the query. Based on this idea, we introduce effective relational conditions, a succinct formalism for representing a family of such automata, containing one automata for each graph. The query answering problem for effective relational conditions is NL-complete. We provide a translation from LARE queries into effective relational conditions, which proves that the data complexity of the query answering problem for LARE queries is in NL. The combined complexity is PSPACE-complete.

Query languages for graphs and LARE. Many of the query languages for graphs are extensions of Regular Path Queries (RPQ) [Cruz et al., 1987]. RPQs can be written in the form \( x \rightarrow^* y \land \pi \in L(e) \) where \( e \) is a (standard) regular expression. Such queries return pairs of nodes \( (v, v') \) connected by a path \( \pi \) such that the labelling of \( \pi \) form a word in \( L(e) \). Conjunctive Regular Path Queries (CRPQs) (see \( Q_{\text{CRPQ}} \) on Fig. 1) are closure of RPQs under conjunction and existential quantification [Consens and Mendelzon, 1990; Mendelzon and Wood, 1995], CRPQs with inverse [Calvanese et al., 2000] allow traversing graph edges back.

Barcelo et al., [2012] considered extended CRPQs (ECRPQs) that have the ability to output and compare not only tuples of nodes, but also tuples of paths (see \( r_{\text{ECRPQ}} \) on Fig. 1).
Tuples of paths can be compared by regular relations [Elgot and Mezei, 1965; Frougny and Sakarovitch, 1993]. Examples of such relations are path equality, length comparisons, prefix (i.e., a path is a prefix of another path), fixed edit distance etc. Regular relations on n-tuples of paths can be defined by the standard regular expressions over alphabet of n-tuples of edge symbols. Such regular expressions are the basic building block of LARE, in particular LARE queries without register assignment and constraints can be seen as unions of ECRPQs. Query answering for ECRPQs is computationally feasible: its combined complexity is PSPACE-complete and data complexity is NL.

The formalisms mentioned above assume that graph edges are labelled by a finite alphabet. In practice, graph nodes often store data values from infinite alphabet and there is a strong need for query formalisms that can combine graph topology and data values tests. In such graphs, paths are interleave sequences of data values and edge labels. This is closely related to data words [Neven et al., 2004; Demri et al., 2007; Segoufin, 2006; Bojańczyk et al., 2011]. Data complexity of query answering for most of the formalisms for data words is NP-hard [Libkin et al., 2016].

Regular Queries with Memory (RQMs) [Libkin and Vrgoč, 2012; Libkin et al., 2016] are again of the form $x \rightarrow^{e} y \land y \in L(e)$, however $e$ is now Regular Expression with Memory (REM). Such queries return pairs of nodes connected by a path in $L(e)$. REMs resemble standard regular expressions but they can store a register the data value at the current position and test its equality with other values stored already in registers (see $\mathcal{Q}_{\text{RQM}}$ on Fig. 1). Data complexity of RQMs is NL. Arithmetical regular expressions of LARE, called ARE, have been inspired by REM—essentially ARE over single path and with constraints having only (dis)equality tests on data values are equivalent to REMs. However, in contrast to REM, ARE work over tuples of paths, which can be compared by regular relations (as in ECRPQs), registers of ARE store nodes of the graph rather than data values and ARE incorporate arithmetical functions and arbitrary comparisons in register constraints. REMs together with their weaker versions (REWBB and REWE) have been further studied in [Libkin et al., 2015].

Walk Logic [Hellings et al., 2013] is a powerful extension of FO with path quantification, and tests of equality of data values on paths. Query answering for WL is decidable but its data complexity is not elementary [Barceló et al., 2015].

Finally, Register Logic [Barceló et al., 2015], is essentially the language of REMs closed under Boolean combinations, node, path and register assignment quantification. Interestingly, it allows for comparing data values in different paths. Its query answering is costly but for the positive fragment RL+ data complexity drops to NL. A particularly interesting fragment is nested RL+ where nested REMs (NREMs) can be used instead of REMs. Nested REMs extend REMs with a branching operator that can filter those nodes in a graph that are the starting point of a path that can be parsed by a nested query which can be NREM again. Data complexity for NRL+ stays in NL. Constraints of ARE allow for nested queries which capture this kind of branching.

Our language LARE can express all the ECRPQ and NRL+ properties and more, as LARE can test various arithmetical conditions rather than just the equality and refer to nodes’ neighbourhood. We provide some concrete examples in the paper, e.g. in Example 1 we show a query $\alpha_{\text{fad}}$ that looks for paths that always choose the next node with the most occurring value in a set of candidates. This query is not expressible in NRL+ or ECRPQ.

Figueira and Libkin [2015] proposed a language to express properties of edge labels of paths, such as the number of a-edges following b-edges is the same as the number of b-edges following a-edges. Data complexity of its query answering problem is in NL, and the proof relies on a tailored version of Parikh automata. We note that it is possible to combine the result with ours, i.e., to have a query language able to express both kind of properties with NL complexity.

2 Graphs and queries

Graphs. We fix a finite set of edge labels $\Sigma$. A $\Sigma,N$-labelled graph, or simply a graph, is a tuple $G = (V,E,\lambda)$ where $V$ is a finite set of nodes, $E \subseteq V \times \Sigma \times V$ is a set of edges labelled by $\Sigma$, and $\lambda : V \rightarrow \mathbb{N}$ is a labelling of nodes.

The size of a graph $G$ is defined as $|G| = |V| + |E| + \sum_{v \in V} \lambda(v)$, i.e., the labels of nodes are represented in the unary notation, which means that their binary representation is logarithmic in the size of the graph. This allows us to compute arithmetical relations on these labels in logarithmic space. The question, which arithmetical relations can be computed in logarithmic space w.r.t. input number given in binary, is related to long-standing open problems, e.g., whether linear programming admits strongly-polynomial algorithm.

A word is an element of the language defined by $V(\Sigma V)^*$. A word $v'$ is an $e$-successor of $v$ if $E(v,e,v')$. A path is a word $p = v_0 e_1 v_1 \ldots v_k$ such that $v_i$ is an $e_i$-successor of $v_{i-1}$ for every $i \in \{1, \ldots, k\}$.

An n-ary relational condition is a graph indexed family of relations $R = \{R^G : G$ is a graph$\}$ such that each $R^G$ is an n-ary relation on the paths from $G$. This may be seen as an n-ary relation on paths, which depends on a graph.

Syntax of queries. Queries $Q$ are defined by the BNF expression $Q ::= Q \lor Q_1 \land Q \mid x_i \rightarrow^{e_i} x_j \mid R(x_1, \ldots, x_n)$, where $n, i, j, k, i_1, \ldots, i_n \in \mathbb{N} \setminus \{0\}$, $R$ ranges over n-ary relational conditions and $x_1, x_2, \ldots$ are variables.

Variables are intended to range over paths; nodes are considered as special cases of paths. Free variables of a query can be distinguished by listing them after the query name, e.g.,
Q(x₁, ..., xₖ) denotes a query whose variables x₁, ..., xₖ are free; the remaining variables are existentially quantified.

**Semantics of queries.** The satisfaction relation ⊨G, which takes a vector of paths ⃗p = (p₁, p₂, ..., pₙ), and a query Q, is recursively defined as follows. We assume that for each i, the path pᵢ is the value of the query variable xᵢ.

- ⃗p ⊨G Q₁ ∨ Q₂ if ⃗p ⊨G Q₁ or ⃗p ⊨G Q₂.
- ⃗p ⊨G Q₁ ∧ Q₂ if ⃗p ⊨G Q₁ and ⃗p ⊨G Q₂.
- ⃗p ⊨G xᵢ → xⱼ when pᵢ, pⱼ are single-node paths and pᵢ is a path from pᵢ to pⱼ.
- ⃗p ⊨G R(x₁, ..., xᵢₙ) if R(G(p₁, ..., pᵢₙ)).

A query Q(x₁, ..., xₙ) holds for paths p₁, ..., pₙ of a graph G, denoted as Q(G(p₁, ..., pₙ)), if there are paths pᵢ₁, pᵢ₂, ..., such that (p₁, p₂, ..., pₙ) ⊨Q. For example, Q(x₁) = x₁ → x₂ ∧ x₄ → x₅, x₁ ∧ R(x₂, x₅) states that there are two paths, one starting and one ending in x₁ satisfying R.

The query-answering problem is formalized as follows: given a graph G, a query Q(x₁, ..., xₙ) and (input) paths p₁, ..., pₙ, does Q(G(p₁, ..., pₙ)) hold? We are interested in the data complexity of the problem, where the size of a query and its input paths is treated as constant, and combined complexity, where there is no such restriction.

## 3 Arithmetical regular expressions

Different relational conditions lead to different expressive power and complexity of queries. As we are interested in queries regarding large systems, representing, for example, Web topology, social networks or Kripke structures, we restrict our attention to relational conditions defining queries whose data complexity is in NL. In this section, we propose n-ary arithmetical regular expressions (ARE), which define relational conditions satisfying this complexity requirement. ARE are regular expressions with arithmetical functions and memory. The memory is formalized as an infinite set of registers R = {rᵢ | i ∈ N}, storing nodes.

It is convenient to reason about paths of the same length. To cope with paths of different lengths, we use □ as a special blank (padding) symbol for nodes and edges and assume E(v, □, □) for any v ∈ V ⊆ and λ(□) = □. One may think that G has an additional dummy node □, which is a 0-successor of all nodes. By V=! we denote the set X extended with □.

We assume a finite set F of functions f : (N₀ ∪ {∞})ⁿ → N ∪ {∞} computable by a non-deterministic Turing machine whose size of working tape and output tape while computing f(x₁, ..., xₙ) is O(log k + max(|x₁|, ..., |xₙ|)), assuming binary representation and 0 = |□| = 1. These conditions are satisfied by aggregative functions such as summation, maximum, minimum (∞ is included for empty set), counting and conditional functions like if x₁ is odd then x₂ else x₃.

**Syntax of ARE.** We define register constraints C and n-ary arithmetical regular expressions α as follows.

\[
C ::= C \lor \neg C \mid \exists r.C \mid f(P₁, ..., Pₖ) \leq f(P₁, ..., Pₖ) \mid \{r = r\} \mid \{[r \mapsto j]\} \mid \{C\} \mid [r \mapsto j] \mid \varepsilon \mid \alpha + \alpha \mid \alpha \cdot \alpha \mid \alpha^+
\]

where \( r \) ranges over \( \mathbb{R} \), \( f \) ranges over \( \mathcal{F} \), \( P \) ranges over expressions of the form \( \ell(x) \) or \( f[r : C] \), \( \ell \in \Sigma \), \( \ell \in \Sigma^n \), \( j \in \{1, ..., n, \square\} \) and \( Q \) is a (nested) query.

By \( \text{ARE}_n \) we denote the class of all n-ary arithmetical regular expressions. We allow standard logical abbreviations, such as \( \land, \lor, \exists \), defined as usual. We put \( E(r, a, r') = V_{\text{max}} E(r, a, r') \) to express there is a (\( \Sigma \)-labelled) edge between \( r \) and \( r' \). Notice that our constraints are standalone entities rather than evaluations applied to subexpressions.

To simplify presentation we require that the register constraints and arithmetical functions only express properties of nodes stored in registers. Hence, all the nodes of the paths have to be stored in registers using the \([r \mapsto i]\) syntax prior to their access. However, it is easy to circumvent this by assuming \( n \) distinguished registers storing the values of the current nodes.

Register constraints allow for Boolean operators (including negation), comparing arithmetical formulas, quantification, checking equality and connectedness of nodes stored in registers and checking nested queries. Note that we allow for negation in front of nested queries, and that the parameters of nested queries are only (nodes stored in) registers. The arithmetical formulas are the way of expressing properties of values of the nodes stored in registers. The construction \( f[r : C] \), that can be read as the function \( f \) applied to the values of all \( r \) satisfying a given condition. For example, \( \max\{r₁ : E(r₂, e, r₁)\} \) stands for the maximum value of \( e \)-successors of the node stored in \( r₂ \).

A valuation \( \sigma : \mathbb{R} \rightarrow V_{\square} \) is a function that assigns nodes to the registers. For a valuation \( \sigma \), we define \( \sigma[r \mapsto v] \) as the valuation such that \( \sigma[r \mapsto v](r) = v₁ \) and \( \sigma[r \mapsto v] \) coincides with \( \sigma \) on all inputs except \( r \).

Let \( f(X) \) denote the result of function \( f \in \mathcal{F} \) whose arguments are elements of \( X \) given in a non-decreasing order.

**Semantics of register constraints.** A graph \( G = (V, E, \lambda) \) and a valuation \( \sigma \) satisfy a constraint \( C \), denoted by \( G, \sigma \models C \), if one of the following holds:

- \( C \equiv C_1 \lor C_2 \) and \( G, \sigma \models C_1 \) or \( G, \sigma \models C_2 \).
- \( C \equiv \neg C' \) and \( G, \sigma \models C \).
- \( C \equiv \exists r.C' \) and there is \( v \in V \) s.t. \( G, \sigma[r \mapsto v] \models C' \).
- \( C \equiv f(P₁, ..., Pₖ) \leq g(P₁, ..., Pₖ) \) and \( f(v₁, ..., vₖ) \leq g(v₁, ..., vₖ) \), where \( vᵢ = \lambda(\sigma(r)) \) if \( Pᵢ \equiv \lambda(r) \) and \( vᵢ \equiv f\{\lambda(\nu)|G, \sigma[r \mapsto v] \models C\} \) if \( Pᵢ \equiv f[r : C] \).
- \( G \models r \neq r' \) and \( \sigma(r) = \sigma(r') \).
- \( C \equiv E(r, e, r') \) and \( E(\sigma, r), e, \sigma(r') \).
- \( C \equiv \{Q\}(r₁, ..., rₙ) \) and \( Q^G(\sigma(r₁), ..., \sigma(rₙ)) \).

The convolution of sequences \( s₁, ..., sₙ \), denoted by \( s₁ \otimes ... \otimes sₙ \), is the sequence \( s \) of the length \( k \) of the longest sequence among \( s₁, ..., sₙ \), such that for every \( i \in \{1, ..., k\} \), the \( i \)-th element of \( s \) is the vector \( (a₁, ..., aₙ) \), where \( a_i \) is the \( i \)-th element of \( s_j \) if it exists and □ otherwise. In plain English, convolution joins \( n \) sequences into one sequence of the length of the longest sequence and fills the missing places with □.

An \( n \)-path of \( G \) is the convolution of \( n \) paths of \( G \). We define a concatenation of two \( n \)-paths \( p₁ = v₀e₁v₁...v₀ = (E, v₀e₁v₁...vₙ, vₙ) \).
\(v'_{k}e'_1 \ldots v'_{k} e'_k\), such that \(v_k = v'_k\) as \(p_1, p_2 = v_0e_1 \ldots v_ke'_1 \ldots v'_k\) (i.e., the common node is not repeated). A splitting of an \(n\)-path \(p\) is a sequence of \(n\)-paths \(p_1,\ldots, p_t\) such that \(p = p_1p_2 \ldots p_t\).

**Language of ARE.** We define the relation \(\vdash\) with the following meaning: \((\alpha, p, \sigma) \vdash^G \sigma'\) if evaluating an expression \(\alpha\) over an \(n\)-path \(p\) of a graph \(G\) with a valuation \(\sigma\) results in valuation \(\sigma'\), i.e., if one of the following holds.

- \(\alpha \equiv \epsilon, p\) is a single element and \(\sigma = \sigma'\).
- \(\alpha \equiv \emptyset C)\), \(p\) is a single element, \(G, \sigma = C\) and \(\sigma = \sigma'\).
- \(\alpha \equiv [r \rightarrow j]\), \(p\) is a single element \((a_1, \ldots, a_n)\) and \(\sigma' = \sigma[r \leftarrow a_j]\) if \(j \neq \emptyset\) and \(\emptyset\) otherwise.
- \(\alpha \equiv \overline{e}\), \(p = v_1\overline{e}v_2\) where \(v_1, v_2\) are single elements and \(\sigma = \sigma'\).
- \(\alpha \equiv \alpha_1 \land \alpha_2\) and there is a splitting \(p = p_1, p_2\) and a valuation \(\sigma''\) s.t. \((\alpha_1, p_1, \sigma'') \vdash^G \sigma''\) and \((\alpha_2, p_2, \sigma'') \vdash^G \sigma'\).
- \(\alpha \equiv \alpha_1 \lor \alpha_2\) and \((\alpha_1, p, \sigma) \vdash^G \sigma'\) or \((\alpha_2, p, \sigma) \vdash^G \sigma'\).
- \(\alpha \equiv \alpha_i\) and there is a splitting \(p = p_1, \ldots p_k\) and valuations \(\sigma = \sigma_0, \sigma_1, \ldots, \sigma_k = \sigma'\) such that for each \(i \in \{1, \ldots, k\}\) we have \((\alpha, p_i, \sigma) \vdash^G \sigma_i\).

Then, the language of an ARE \(\alpha\) is defined as \(L^G(\alpha) = \{p | \exists \sigma, \sigma' \ldots (\alpha, p, \sigma) \vdash^G \sigma'\}\). We define \(\alpha^G(\cdot, \ldots, \cdot)\) iff \(\bar{p} \subseteq \cdot \ldots \cdot \bar{p} \in L^G(\alpha)\); therefore, each \(\alpha^G\) can be treated as a relational condition.

**Example 1.** Consider the following ARE.

\[
\begin{align*}
\alpha_{grd} &= ([r_1 \leftarrow 1] \Sigma [r_2 \leftarrow 1] C_{grd})^*, \\
C_{grd} &= \forall v_3 (E(r_1, v_3) \Rightarrow \lambda(r_3) \leq \lambda(r_2)), \\
\alpha_{fad} &= ([r_1 \leftarrow 1] [r_2 \leftarrow 1] \emptyset v_3 (P(r_3) \leq P(r_2))^*, \\
\alpha_{com} &= [r_1 \leftarrow 1] \ldots [r_n \leftarrow n] \alpha_{gas} (\Sigma [r_1 \leftarrow 1] + \ldots + [r_n \leftarrow n]) + \Sigma^P
\end{align*}
\]

The language of \(\alpha_{grd}\) consists of all greedy paths, i.e., paths in which, at every position, the following node on the path has maximal value among all successors of the current node. To see that, observe that while traversing path \(\alpha_{grd}\) keeps the next-to-latest (on the path) node in register \(r_1\) and the latest node in \(r_2\). After each edge move, denoted by \(\Sigma\), a greedy condition \(C_{grd}\) is checked, whether it holds that \(r_2\) is a maximal neighbour of \(r_1\). The language of \(\alpha_{fad}\) consists of paths in which, at every position, the following node on the path has the value that has the most number of occurrences among the successors of the current node. To express that, \(\alpha_{fad}\) uses arithmetic formula \(P(z)\) counting number of occurrences of value \(\lambda(z)\) in the neighbourhood of \(r_1\), \(\Sigma^P\) is a function from \(\mathcal{F}\) that returns the number of elements given on input. The last example demonstrates processing multiple paths. We have \(\alpha_{com}(x_1, \ldots, x_n)\) iff paths \(x_1, \ldots, x_n\) have a common node. The idea is to guess \(\alpha_{gas}\) a common node for each path separately, and verify the equality at the end.

**Theorem 1.** The query answering problem for LARE queries with bounded nesting depth is in PSPACE and its data complexity is NL.

**4 Effective relational conditions**

We introduce the notion of effective relational conditions, which facilitates the proof of Theorem 1, and show that ARE are effective relational conditions (Theorem 4). The name effective is justified by the fact that answering queries with effective relational conditions can be done within the desired complexity bounds (Theorem 3). We assume that the reader is familiar with finite automata and Turing machines (see Hopcroft and Ullman, 1979).

An Automata Giving Turing Machine (AGTM) is a nondeterministic Turing Machine which works in logarithmic space and only accepts inputs of the form \(i?w\), where \(i \in \{0, 1\}^*, \in \{\epsilon, \ldots, ?_t, ?_f, ?_a\}\), and \(w \in \{0, 1, \ldots\}\) is such that \(|w| = O(\log(|i|))\). An AGTM \(M\) gives (on-the-fly) a set of (nondeterministic) automata \(\{A_i\}_{i \in \{0, 1\}}\) such that each \(A_i\) is of the form \((\Gamma, S_0, I, \delta_i, F)\), where

- \(\Gamma \subseteq \{0, 1\}^*\) consists of labels \(\epsilon\) s.t. \(M\) accepts on \(i?\epsilon\).
- \(S_0 \subseteq \{0, 1\}^*\) consists of states \(q\) s.t. \(M\) accepts on \(i?q\).
- \(I\) consists of initial states \(q \in S\) s.t. \(M\) accepts on \(i?q\).
- \(F\) consists of final states \(q \in S\) s.t. \(M\) accepts on \(i?q\).
- \(\delta_i\) consists of transitions \((q, e, q')\) \(\in S \times \Gamma \times S\) s.t. \(M\) accepts on \(i?\delta_i q\).

An AGTM \(M\) represents a \(n\)-ary relational condition \(R\) if \(M\) gives a set of automata \(\{A_i\}_{i \in \{0, 1\}}\) s. t. for all graphs \(G\) and words \(w_1, \ldots, w_n\), the automaton \(A_G\) accepts \(w_1 \otimes \ldots \otimes w_n\) iff \((w_1, \ldots, w_n) \in R^G\) (we assume an encoding of graphs as binary sequences). A relational condition \(R\) is effective if there is an AGTM \(M\) that represents \(R\). A query is effective if its all relational conditions are given as AGTMs.

**Lemma 2.** Let \(R\) be a relational condition such that \(R^G(x, x, x')\) holds iff \(x \rightarrow x'\). Then \(R\) is effective.

**Proof.** We define an AGTM \(M\) that gives automata \(\{A_i\}_{i \in \{0, 1\}}\) such that for a given graph \(G\), the automaton \(A_G\) recognizes words over the alphabet \(\Sigma^G \cup \Sigma^G\), which encode convolutions \(w_1 \otimes w_2 \otimes w_3\) such that \(w_1, w_2\) are single-node paths and \(w_3\) is a path in \(G\) from the node \(w_1\) to \(w_3\). \(A_G\) reads letters corresponding to nodes \(v\) and labels \(e\) and stores last read letters in its state. Next, while it reads a letter corresponding to a node \(v'\), it checks whether \(G\) has an edge from \(v\) to \(v'\) labeled with \(e\). Moreover, the automaton \(A_G\) stores in its state the node \(v_f\), which is supposed to be the last node and accepts only if the last read letter is \(v_f\).

The machine \(M\) answers questions \(i?x?x\) regarding components (the alphabet, the set of states, etc.) of \(A_G\), where \(G\) is the graph encoded by \(i\). All components are sets of tuples of nodes and labels of \(G\) extended with the padding symbol \(\emptyset\). All these sets of tuples are defined using the edge relation in \(G\), equality among components of the tuples and equality to \(\emptyset\). In any reasonable representation of \(G\); e.g., a list of nodes followed by an adjacency list, \(|G|\) is proportional to \(i\), length of description of labels and nodes of \(G\) is logarithmic in \(i\), and the machine \(M\) can compute in \(\log i\) space (hence in \(\log |G|\) space) whether a given word encodes a label (resp., a node or an edge) of \(G\). Since the length of elements of tuples, i.e., labels and nodes of \(G\) is logarithmic in \(i\), the equality among components of tuples (and equality to \(\emptyset\) can be checked in
log \log i \text{ space. Therefore, the machine } \mathcal{M} \text{ on inputs } i?xk \text{ requires } \log(i) \text{ space, i.e., } \mathcal{M} \text{ is an AGTM.}

**Theorem 3.** Answering effective queries with \( m \) conditions is in \( \text{NSPACE}(m + c \log(n)) \), where \( c \log(n) \) is the space bound on AGTMs representing the conditions.

**Proof sketch.** Consider a query \( Q(x_1, \ldots, x_k) \) whose relational conditions are effective. For any paths \( p_1, \ldots, p_k \), we define in the straightforward way an effective relational condition \( R_{p_1, \ldots, p_k} \) such that \( R_{p_1, \ldots, p_k}(\pi_1, \ldots, \pi_k) \) iff \( p_1 = \pi_1, \ldots, p_k = \pi_k \). This and Lemma 2 imply that we can assume that our query has no free variables and is built of conjunction, disjunction and effective relational conditions. One can observe that effective relational conditions are closed under subfiling, joining and adding spurious arguments, and therefore we can assume that all of them have the same arguments. By employing an argument similar to the standard powerset construction for finite automata, we can reduce in polynomial time a query to a single effective relational condition \( R_{Q,p_1,\ldots,p_k} \).

Given a graph \( G \), answering \( Q^G(p_1, \ldots, p_k) \) amounts to checking emptiness of \( R_{Q,p_1,\ldots,p_k}^G \), which amounts to checking whether a corresponding automaton \( A_G \) is empty. This can be done by employing the standard graph reachability algorithm, which checks reachability of an accepting state from initial states by guessing the consecutive states. Since states are logarithmic in \( \iota \), the problem is in \( \text{NL} \).

The AGTM for the relational condition \( R_Q \) works in space proportional to the number of atomic formulas of the query \((m)\) plus the maximum of the space usage of AGTMs representing relational conditions in \( Q(c \log(n)) \). Therefore, the combined complexity is as required.

Let \( 0\text{-ARE} \) be the empty set and \((d + 1)\text{-ARE} \) be the subset of \( \text{ARE} \) containing expressions whose all nested queries are built of relational conditions from \( d\text{-ARE} \).

**Theorem 4.** Every ARE relational condition \( R \) is effective and if \( R \) is \( d\text{-ARE} \), then an AGTM working in space \(|R|\log(n)\) representing \( R \) is computable in polynomial time.

**Proof sketch.** We prove Theorem 4 by induction w.r.t. \( d \). The basis of induction is trivial as the set \( 0\text{-ARE} \) is empty. Assume that for any \( R \in d\text{-ARE} \) one can compute in polynomial time in \(|R|\) an AGTM computing \( R \) that works in space \(|R|\log(n)\).

**Evaluating register constraints.** We first show that for a given register constraint \( C \) whose nested queries are built of \( d\text{-ARE} \), one can compute in polynomial time in \(|C|\) a non-deterministic Turing machine \( \mathcal{M}_C \) that works in space \(|C|\log |G|\) and decides, for a graph \( G \) and a registers valuation \( \sigma \), whether \( G, \sigma \models C \). The construction of \( \mathcal{M}_C \) is a top-down recursion on \( C \) and can be done in polynomial time. The space used by \( \mathcal{M}_C \) is bounded by \( O(|C|\log |G|) \), which, using tape compression [Hopcroft and Ullman, 1979], can be replaced by \( |C|\log |G| \). Below we discuss the most interesting recursive cases.

To compute \( f[r : C] \), observe that using additional space \( \log |G| \) we can implement a virtual tape storing \( \lambda(v_1) \leq \ldots \leq \lambda(v_i) \), where \( v_1, \ldots, v_i \) are all nodes of \( G \) satisfying \( C \), i.e., we can implement a subroutine that for a given \( i \) returns \( \lambda(v_i) \). Such a subroutine iterates over all nodes \( v \) of \( G \). For each \( v \), it computes \( i^*_v \) (resp., \( i^*_v \)) defined as the number of nodes \( u \), which satisfy \( C \) and \( \lambda(u) \leq \lambda(v) \) (resp., \( \lambda(u) \leq \lambda(v) \)), and checks whether \( i^*_v < i \leq i^*_v \). In such a case we know that all elements between \( i^*_v + 1 \) and \( i^*_v \) have the same value \( \lambda(v) \). Next, with such a virtual input tape we execute a non-deterministic Turing machine \( \mathcal{M}_f \) computing \( f \in F \). All such Turing machines work in space \( O(\log |G|) \).

Consider a nested query \([Q]|_{v_1, \ldots, v_i} \). All the relational conditions of \( Q \) are \( d\text{-ARE} \), therefore by Theorem 3 and the inductive assumption, we can compute (in polynomial time) a non-deterministic machine \( \mathcal{M}_Q \) that computes \([Q]|_{v_1, \ldots, v_i} \) in space \(|Q|\log |G|\). We also compute a non-deterministic machine \( \mathcal{M}_\triangleleft \) that in space \( c_{IS}(|Q|\log |G|) \) computes \( \triangleleft([Q]|_{v_1, \ldots, v_i}) \), where \( c_{IS} \) is the constant from Immerman-Szelepcsenyi theorem [Immerman, 1988] showing \( \text{NL = coNL} \). We execute \( \mathcal{M}_Q \) and \( \mathcal{M}_\triangleleft \) in parallel.

**Evaluating \((d + 1)\text{-ARE} \).** Now we prove the inductive thesis. Let \( R \) be a \((d + 1)\text{-ARE} \) relational condition defined by \((d + 1)\text{-ARE} \). For a graph \( G \), we define three automata \( A_G^G, A_G^C \) and \( A_G^Q \) over an alphabet \( \Gamma \) consisting of tuples of edge labels \( \Sigma^n \), nodes \( V^n \), alphabet symbols \( \{aC : C \text{ from } R\} \) referring to register constraints and \( \{b_{r,j} : [r \leftarrow j] \text{ from } R\} \) referring to register assignments. We substitute in \( \alpha \) subexpressions \( \langle C \rangle \) (resp., \( \langle r \leftarrow j \rangle \)) by letter \( aC \) (resp., \( b_{r,j} \)). The resulting \( \alpha^{reg} \) is a regular expression and \( A_G^{\alpha^{reg}} \) recognizes the language of \( \alpha^{reg} \), i.e., \( A_G^{\alpha^{reg}} \) accepts words over \( \Gamma \) that satisfy \( R \) while neglecting the graph structure \( G \) and the register constraints checks in \( R \). For the register constraints checks, this automaton verifies only that the letters \( aC \) (resp., \( b_{r,j} \)) occur at positions that correspond to register constraints checks \( \langle C \rangle \) (resp., assignments \( [r \leftarrow j] \)) in \( R \). The automaton \( A_G^{\alpha^{reg}} \) checks consistency of register constraints checks marked by \( aC \) provided that register assignments are executed done according to markings \( b_{r,j} \), employing the machine \( \mathcal{M}_C \) defined above. The automaton \( A_G^{\alpha^{reg}} \) recognizes words that, with letters \( aC \) and \( b_{r,j} \) deleted, are \( n\)-paths of \( G \). Finally, we construct \( A_G^Q \), which is the product of automata \( A_G^G, A_G^C \) and \( A_G^{\alpha^{reg}} \) projected on the alphabet \( \Sigma^n \cup V^n \). Observe that \( A_G^Q \) recognizes the languages of all \( n\)-paths satisfying \( R \). It can be shown that the construction of automata \( A_G^G, A_G^C \) and \( A_G^{\alpha^{reg}} \) can be performed in a way that ensures that the sets containing all such automata can be given on-the-fly.

Finally, we show how Theorems 3 and 4 imply Theorem 1. Let \( d > 0 \) and consider LARE queries of nesting depth bounded by \( d \), i.e., queries whose relational conditions are \((d + 1)\text{-ARE} \). For every such query \( Q \), Theorem 4 states that for every relational condition \( R \) from \( Q \), one can construct in polynomial time an AGTM working in \(|R|\log(n)\) space, which represents \( R \). Next, observe that the sum of the number and the sizes of relational conditions from \( Q \) is bounded by \( |Q| \). Therefore, by Theorem 3, space required to answer \( Q \) is bounded by \(|Q|\log(n)\). Consequently, the query answering problem is in \( \text{PSPACE} \) and its data complexity is in \( \text{NL} \).

**Remark.** We have shown that for a given query \( Q \) and a graph \( G \) we can obtain a single relational condition \( R \).
and then an automaton $A^G_R$ accepting all $n$-paths $w$ over $V^N((X^n)\times V^*)$ such that $w$ is a convolution of paths that satisfy $Q$. Note that we can modify $A^G_R$ to be able to process paths that remember also the values of the nodes, i.e., the paths over $(V \times N)^n((X^n)\times V^*)^*$. Therefore, we can state our result in the following alternative version:

**Corollary 5.** For each $\Sigma, N$-labelled graph $G$ and a LARE query $Q$ with $k$ variables, there is a $2^X, N^k$-labelled graph $G'$ of size polynomial in $|G|$ and sets of nodes $I, F$ such that a word $w$ is a path from $I$ to $F$ in $G'$ iff $w$ is a convolution of paths that satisfy $Q$.

5 Examples

We show how ARE power the LARE queries, and combine nicely with language features like nested queries, condition negation, or aggregative functions. For convenience, we define a macro \([n] = [r_1 \leftarrow 1] \ldots [r_n \leftarrow n]\) that stores nodes from first $n$ paths into registers $1, \ldots, n$. We allow to use different variables than $x_i$ and associate variables $x, y, x_i, y_i$ to be nodes, this can be enforced by appending the atom $i \neq j$ to query body; we treat it as implicitly written.

**Basic techniques.** Consider the queries:

- $Q_4(x, \pi, \pi_2) = \alpha_i(x, \pi_1) \land \alpha_2(x, \pi_2)$, where
  - $\alpha_1 = \lceil \pi \rceil = \lceil \pi \rceil = \lceil \pi \rceil = \lceil \pi \rceil$,
  - $\alpha_2 = \lceil \pi \rceil \land \alpha(x, \pi_2)$,
- $Q_2(\pi) = \lceil \pi \rceil = \lceil \pi \rceil = \lceil \pi \rceil$,
- $Q_3(x, y) = x \leftarrow \pi \land \alpha(x, y), \alpha = \lceil \pi \rceil = \lceil \pi \rceil = \lceil \pi \rceil$.

The query $Q_4$ checks whether a node $x$ appears in path $\pi$ but not in $\pi_2$. The query $Q_2$ checks whether a given path has unique nodes with respect to strongly connected components. The query $Q_3$ that check whether a given graph is a directed acyclic graph. The query has no parameters, so it is considered a representation of a local state of an agent. The technique presented in this paper can be straightforwardly generalised to graphs with more than one node-labelling function in order to deal with more agents.

Our query language then is powerful enough to formulate indistinguishability in the epistemic interval-temporal logic EHS [Lomuscio and Michaliszyn, 2013], where two paths are indistinguishable for an agent $i$ if they are of the same length and the corresponding states of both paths are indistinguishable/have the same label for $i$. In particular, $Q_{K1}$ (below) expresses that there is a path indistinguishable but different from $\pi$. $Q_{K2}$ expresses that there is a path from $x$ to $y$ indistinguishable from $\pi$, that visits $z$ while $\pi_i$ does not. In other words, an agent, who can observe only their local state (node labelling), does not know whether a given path avoids $z$.

- $Q_{K1}(\pi) = \alpha_i(\pi_1, \pi_2)$,
- $\alpha_i = \lceil \pi \rceil \land \alpha_2(\pi_1, \pi_2), \alpha = \lceil \pi \rceil \land \alpha_2(\pi_1, \pi_2)$.

6 Conclusion and future work

We introduced the query language LARE for data graphs, suited for expressing arithmetical properties of nodes, paths between nodes and their neighbourhoods. We showed that the query answering problem for LARE can be solved in logarithmic space in size of the graph by employing a new formalism called effective relational conditions.

LARE expresses some aggregative properties of paths, as exemplified in Section 5. In our future work we plan to incorporate more properties of this type. In particular, are interested in formalisms that can compare sums of values on different paths while keeping the complexity low.
Acknowledgement  We would like to thank Leonid Libkin who has introduced the subject to us. This paper has been supported by Polish National Science Center grant UMO-2014/15/D/ST6/00719.

References


