

MODULAR PATH QUERIES WITH ARITHMETIC

JAKUB MICHALISZYN, JAN OTOP, AND PIOTR WIECZOREK

University of Wrocław, Poland
e-mail address: {jmi,jotop,piotrek}@cs.uni.wroc.pl

ABSTRACT. We propose a new approach to querying graph databases. Our approach balances competing goals of expressive power, language clarity and computational complexity. A distinctive feature of our approach is the ability to express properties of minimal (e.g. shortest) and maximal (e.g. most valuable) paths satisfying given criteria. To express complex properties in a modular way, we introduce labelling-generating ontologies. The resulting formalism is computationally attractive — queries can be answered in non-deterministic logarithmic space in the size of the database.

1. INTRODUCTION

Graphs are one of the most natural representations of data in a number of important applications such as modelling transport networks, social networks, technological networks (see the surveys [AAB⁺17, Woo12, Bar13]). The main strength of graph representations is the possibility to naturally represent the connections among data. Effective search and analysis of graphs is an important factor in reasoning performed in various tasks. This motivates the study of query formalisms, which are capable of expressing properties of paths.

Nevertheless, most real-world data still resides in relational databases and relational engines are still the most popular database management systems [dbe18]. Hence, it would be desirable to consider a query formalism that directly generalizes the relational approach, offers a natural representation of data values and, at the same time, enables convenient querying of the graph structure.

Modern day databases are often too large to be stored in computers' memory. To make a query formalism computationally feasible, its query evaluation problem should be, preferably, in logarithmic space w.r.t. the size of the database (data complexity) [CDGL⁺06, ACKZ07, BLLW12]. Checking existence of a path between two given nodes is already NL-complete and hence NL is the best possible lower bound for an expressive language for graph querying. It is worth to mention that every problem in NL can be solved deterministically in $O(\log^2(n))$ space, which is a reasonable bound even for huge databases.

Key words and phrases: relational databases, graph databases, queries, aggregation.

* The work was supported by Polish National Science Centre grant 2014/15/D/ST6/00719.

Our contribution. We propose a new approach for querying graph data, in which labelling-generating ontologies are first-class citizens. It can be integrated with many existing query formalisms, both relational and graphical. To make the presentation clear we introduce the concept by defining a new language OPRA. OPRA features NL-data complexity, good expressive power and a modular structure. The expressive power of OPRA strictly subsumes the expressive power of popular existing formalisms with the same complexity (see Fig. 1). Distinctive properties expressible in OPRA are based on aggregation of data values along paths and computation of extremal values among aggregated data. One example of an OPRA-expressible property is “ p is a path from s to t that has both the minimal weight and the minimal length among all paths from s to t ”. Within our model, our OPRA queries can also use labelling functions defined as classic relational views by SQL queries.

Our data model is fairly general. The database consists of a finite number of nodes and a number of labelling functions assigning integers to fixed-size vectors of nodes. We assume for simplicity that the data values are integers; if needed, the labelling functions can be generalized to other datatypes. Relational tables (of any arity) can be viewed as labelling functions assigning the value 1 to the vectors in the given relation and 0 otherwise. In particular, classic edges can be represented implicitly by a binary labelling function returning 1 for all pairs of nodes connected by an edge and 0 otherwise, and weighted edges can be represented by allowing other values. Nodes labelled by integers can be represented using unary labellings.

We define OPRA in Section 3 in stages starting with two fragments of OPRA, which we also employ later on in the discussion on the expressive power. The first fragment is PR, whose main components are the two types of constraints: *Path* and *Regular*. We use path constraints to specify endpoints of graph paths; the other constraints only specify properties of paths. Regular constraints specify paths using regular expressions, adapted to deal with multiple paths and infinite alphabets.

The second fragment is PRA, that extends PR with the *Arithmetical* constraints. The arithmetical constraints compare linear combinations of aggregated values, i.e., values of *labels* accumulated along whole paths. The language PRA can only aggregate and compare the values of labelling functions already defined in the graph. The properties we are interested in often require performing some arithmetical operations on the labellings, either simple (taking a linear combination) or complicated (taking minimum, maximum, or even computing some subquery). Such operations are often nested inside regular expressions (as in [GMOW16]) making queries unnecessarily complicated. Instead, similarly as in [AGP14], we specify such operations in a modular way as *ontologies*. This leads to the language OPRA, which comprises *Ontologies* and PRA. In our approach all knowledge on graph nodes, including all data values, is encoded by labellings. Our ontologies are also defined as *auxiliary* labellings. For example, having a labelling $\text{child}(x, y)$ stating that x is a child of y , we can define a labelling $\text{descendant}(x, y)$ stating that x is a descendant of y . Such labellings can be computed on-the-fly during the query evaluation.

Then, in Section 4, we present a number of examples intended as an illustration of the versatility and usefulness of the language. We also discuss the closure properties of OPRA and possible applications of OPRA to the verification of properties of Kripke structures.

In Section 7, we compare the expressive power of our formalisms and the classic query languages *extended Conjunctive Regular Path Queries* (ECRPQ) and ECRPQ with *linear constraints* (ECRPQ+LC) [BLLW12]. We prove the main results depicted in Figure 1,

primarily that PR subsumes ECRPQ and PRA subsumes ECRPQ+LC. We also illustrate there the additional expressive power of OPRA over PRA.

Finally, in Section 8, we study the complexity of the query evaluation problem for OPRA. Namely, when the depth of nesting of auxiliary labellings is bounded, the problem is PSPACE-complete and its data complexity is NL-complete.

This work is based on two conference papers [GMOW16] and [MOW17]. The goal of [GMOW16] is to design a query language for graphs that is able to express various arithmetic and aggregative properties of nodes and paths, assumed that nodes are labelled with natural numbers. The resulting language can express properties like “there is a path from u to v with the sum of nodes’ values less than c ”; however, it only works because the labels are non-negative. The language also cannot compare averages or even compare values of different paths. The paper [GMOW16] introduces one of the most important technical tools we use here: Query Applying Machines, which are Turing machines designed to provide a succinct representation of graphs that allow us to operate on them in logarithmic space. The second paper [MOW17] successfully removed many of these limitations, while keeping the good (NL) data complexity. In [MOW17], we introduced the language OPRA, which we also use in this paper. The language OPRA operates on integer labels and can compare values of different paths, which allows us to study properties of best/longest/shortest paths. In this paper, we extend the contribution of [GMOW16] and [MOW17] in a few ways. First of all, the paper [MOW17] was written for graph databases, where we deal with a database that consists of a single graph. Here, we adjust the narrative to present OPRA as graph language that can be used with graph databases but also with relational databases, as by design it can easily deal with relations of arity greater than 2. We made the presentation of the language easier and provided some additional examples. We also added a study on the closure properties of OPRA and a careful study on the expressive power on the language (there was a short discussion on the expressive power of OPRA in [MOW17], but it focused mostly on the fact that OPRA has access to more properties of the graph; e.g., OPRA is more expressive than ECRPQ+LC because ECRPQ+LC cannot access nodes’ labels; here we provide an analysis that does not depend on such technicalities). Finally, we sharpen the complexity a bit: in [MOW17], we proved that for a bounded number of auxiliary labellings the data complexity is NL-complete and the combined complexity is PSPACE-complete, whereas here we show that the data complexity is always NL-complete, and the combined complexity is PSPACE-complete for queries with bounded nesting depth. We also provide here the full proofs of the complexity bounds.

Design choices. We briefly discuss the design choices we made during the development of OPRA. Our main goal was to extend the expressive power of graph queries to express properties exemplified in Section 4. Once we overcame the main technical difficulties, the natural question arose: should we incorporate this expressive power into some existing language, either practical or academic, or create a new language? Our initial decision was to do the latter, and so we designed the language presented in the paper [GMOW16], which can be seen as a non-conservative extension of ECRPQ (i.e., an extension increasing the expressive power). This language, however, gained some negative feedback because of the complexity of the language, that obscured the main features we presented. So here, having the lesson learned, we designed a new, modular language that, while is based on some ideas from ECRPQ, is much simpler and therefore provides much easier insight into the expressive power we provide. Due to the modular composition of the language, it is possible to integrate

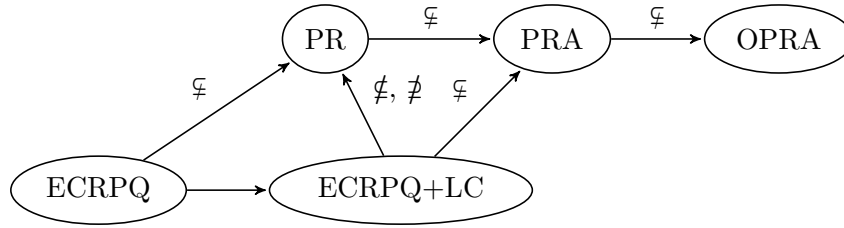


FIGURE 1. Comparison between different query languages. ECRPQ is strictly subsumed by PR by Theorem 7.1, ECRPQ+LC is strictly subsumed by PRA by Theorem 7.2, PR and ECRPQ+LC are incomparable by Theorem 7.2, OPRA strictly subsumes PRA by Remark 7.3, and PRA strictly subsumes PR because it contains ECRPQ+LC whilst PR does not.

its features with other languages; for example, one can use OPRA on top of a relational database that normally uses SQL. It is also possible to integrate the distinctive features of OPRA directly into the other languages, if needed.

Another choice we made in this paper was to abstract from the technical details of the implementation of OPRA and to focus on the computational complexity aspect. There is an ongoing work on an implementation of OPRA where these details are to be addressed.

Related work. *Regular Path Queries (RPQs)* [CMW87, CDLV00] are usually used as a basic construction for graph querying. RPQs are of the form $x \rightarrow^\pi y \wedge \pi \in L(e)$ where e is a (standard) regular expression. Such queries return pairs of nodes (v, v') connected by a path π such that the labelling of π forms a word from $L(e)$. *Conjunctive Regular Path Queries (CRPQs)* are the closure of RPQs under conjunction and existential quantification [CM90, MW95]. Barcelo et al., [BLLW12] introduced *extended CRPQs (ECRPQs)* that can compare tuples of paths by *regular relations* [EM65, FS93]. Examples of such relations are path equality, length comparisons, prefix (i.e., a path is a prefix of another path) and fixed edit distance. Regular relations on tuples of paths can be defined by the standard regular expressions over the alphabet of tuples of edge symbols.

Graph nodes often store *data values* from an infinite alphabet. In such graphs, paths are interleaved sequences of data values and edge labels. This is closely related to *data words* studied in XML context [NSV04, DLN07, Seg06, BDM⁺11]. Data complexity of query evaluation for most of the formalisms for data words is NP-hard [LMV16]. This is not the case for *register automata* [KF94], which inspired Libkin and Vrgoč to define *Regular Queries with Memory (RQMs)* [LMV16]. RQMs are again of the form $x \rightarrow^\pi y \wedge \pi \in L(e)$, where e is a *Regular Expression with Memory (REM)*. REMs can store in a register the data value at the current position and test its equality with other values already stored in registers. Register Logic [BFL15] is, essentially, the language of REMs closed under Boolean combinations, node, path and register-assignment quantification. It allows for comparing data values in different paths. The positive fragment of Register Logic, RL^+ , has data complexity in NL, even when REMs can be nested using a branching operator.

Another related formalism is Walk Logic [HKdBZ13] (WL), which extends FO with path quantification and equality tests of data values on paths. The main disadvantage of the Walk Logic is high complexity: query evaluation for WL is decidable but its data complexity is not elementary [BFL15].

Aggregation. Ability to use aggregate functions such as sum, average or count is a fundamental mechanism in database systems. Klug [Klu82] extended the relational algebra and calculus with aggregate functions and proved their equivalence. Early graph query languages G^+ [CMW88] or GraphLog [CM90, CM93] can aggregate data values. Consens and Mendelzon [CM93] studied *path summarization*, i.e., summarizing information along paths in graphs. They assumed natural numbers in their data model and allowed to aggregate summarization results. In order to achieve good complexity (in the class NC) they allowed aggregate and summing operators that form a closed semiring. Other examples of aggregation can be found in [Woo12].

Summing vectors of numbers along graph paths have been already studied in the context of various formalisms based on automata or regular expressions and lead to a number of proposals that have combined complexity in PSPACE and data complexity in NL. Kopczyński and To [KT10] have shown that *Parikh images* (i.e., vectors of letter counts) for the usual finite automata can be expressed using unions of linear sets that are polynomial in the size of the automaton and exponential in the alphabet size (the alphabet size, in our context, corresponds to the dimension of vectors). Barcelo et al. [BLLW12] extended ECRPQs with linear constraints on the numbers of edge labels counts along paths. They expressed the constraints using reversal-bounded counter machines, translated further to Presburger arithmetic formulas of a polynomial size and evaluate them using techniques from [KT10, Sca84].

Figueira and Libkin [FL15a] studied *Parikh automata* introduced in [KR03]. These are finite automata that additionally store a vector of *counters* in \mathbb{N}^k . Each transition also specifies a vector of natural numbers. While moving along graph paths according to a transition the automaton adds this transition's vector to the vector of counters. The automaton accepts if the computed vector of counters is in a given semilinear set in \mathbb{N}^k . Also, a variant of regular expressions capturing the power of these automata is shown. This model has been used to define a family of variants of CRPQs that can compare tuples of paths using *synchronization languages* [FL15b]. This is a relaxation of regularity condition for relations on paths of ECRPQs and leads to more expressive formalisms with data complexity still in NL. These formalisms are incomparable to ours since they can express non-regular relations on paths like suffix but cannot express properties of data values, nodes' degrees or extrema.

Cypher [The18] is a practical query language first implemented in the graph database engine Neo4j. It uses *property graphs* as its data model. These are graphs with labelled nodes and edges, but edges and nodes can also store attribute values for a set of *properties*. **MATCH** clause of Cypher queries allows for specifying graph patterns that depend on nodes' and edges' labels as well as on their properties values. OpenCypher, an initiative to standardize the language, has produced Cypher Query Language Reference (Version 9) [ope17]. More on Cypher can be found in the surveys [AAB⁺17] and [FGG⁺18].

G-Core [AAB⁺18] is a joint effort of industrial and academic partners to define a language that is composable (i.e. graphs are inputs and outputs of a query), treats paths as first-class citizens and integrates the most important features of existing graph query languages. The G-Core data model extends property graphs with paths. Namely, in a graph, there is also a (possibly empty) collection of paths. The paths have their identity and can have their own labels and (property, value) pairs. G-Core includes also features like aggregation and (basic) arithmetic along paths that is closely related with our proposal. G-Core allows for defining costs of paths either by the hop-count (length) or by the positive weights (which

may be computed by functional expressions). The full cost of a path is the sum of the weights and G-Core is able to look for paths that minimize it. In contrast, our data model allows negative weights.

In Section 5 we compare the language constructions of G-Core, Cypher and OPRA in more detail.

Another proposal of a graph query language of commercial strength is PGQL [vRHK⁺16]. PGQL closely follows syntactic structures of SQL and defines powerful regular expressions that allow for filtering nodes and edges along paths as well as computing shortest paths.

Our data model allows to operate on property graphs [AAB⁺17], where many edges between a pair of nodes are allowed. To do so, for each of the edges we introduce a single, unique, additional node. Then, an edge can be represented by a binary labelling returning 1 for the pair of the source node and the additional node, and for the pair of the additional node and the target node for the edge. Naturally, the property values for the edge are assigned by unary labellings (named after the property keys) of the additional node. We present an example of how to encode a property graph in Section 2. Alternatively, an edge can be represented by a ternary labelling function returning 1 for all triples of the source, the target node and the corresponding additional node.

RDF [CLW14] is a W3C standard that allows encoding of the content on the Web in a form of a set of *triples* representing an edge-labelled graph. Each triple consists of the subject s , the predicate p , and the object o that are resource identifiers (URI's), and represents an edge from s to o labelled by p . Interestingly, the middle element, p , may play the role of the first or the third element of another triple. Our formalism OPRA allows operating directly on RDF without any complex graph encoding, by using a ternary labelling representing RDF triples. This allows for convenient navigation by regular expressions, in which also the middle element of a triple can serve as the source or the target of a single navigation step (cf. [LRSV18]). The standard query formalism for RDF is SPARQL [PS08, HS13]. It implements *property paths*, which are RPQs extended with inverses and limited form of negation (see the survey [AAB⁺17]).

Another idea in processing graphs, fundamentally different than our approach is based on Pregel model [MAB⁺10] where the computation is organized in rounds. Basically, each node has a state and in each round is able to send messages to its neighbouring nodes and change its state according to the messages sent by its neighbours.

2. PRELIMINARIES

Various kinds of data representations for graphs are possible and presented in the literature. The differences typically include the way the elements of graphs are labelled — both nodes and edges may be labelled by finite or infinite alphabets, which may have some inner structure. Here, we choose a general approach in which a *labelled graph*, or simply a graph, is a tuple consisting of a finite number of *nodes* V and a number of labelling functions $\lambda : V^l \rightarrow \mathbb{Z} \cup \{-\infty, \infty\}$ assigning integers to vectors of nodes of some fixed size (we allow labellings without parameters, i.e., $l = 0$, for reasons that will be apparent later).

While edges are not explicitly mentioned, if needed, one can consider an *edge* labelling E such that $E(v, v')$ is 1 if there is an edge from v to v' and it is 0 otherwise.

A special case of a labelled graph is a *relational database*. In this case, the range of all the labellings is $\{0, 1\}$ (and hence the labellings can be called “relations”), and every node

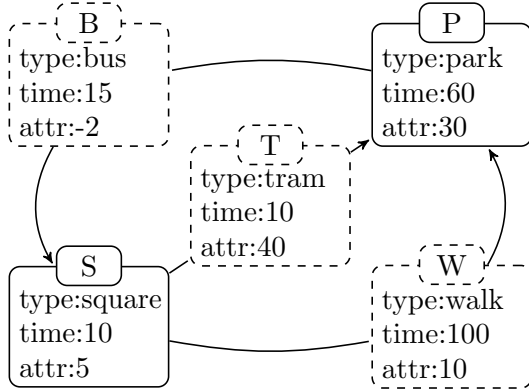


FIGURE 2. A property graph G : the properties of the edges are in dashed boxes.

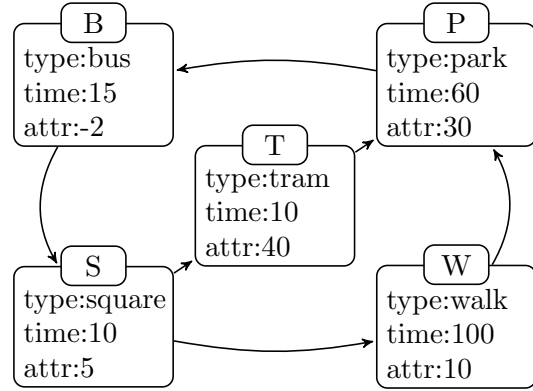


FIGURE 3. An equivalent graph G' : its nodes represent the nodes and the edges of G .

has to be in at least one relation. Clearly, each of the labellings defines a relation of the same arity as the labelling.

For convenience, we assume that the set of nodes contains a distinguished node \square — this is an artificial node we use to avoid problems with paths of different lengths. Note that all labellings must define their values also for tuples with \square and we do not restrict these values in principle.

A *path* is a sequence of nodes. For a path $p = v_1 \dots v_k$, by $|p|$ we denote the length of p and by $p[i]$ we denote its i -th element, v_i , if $i \leq k$, and \square otherwise.

To compare our language with other formalisms, we will also consider a special subclass of labelled graphs, called *standard graphs*. These are graphs with a single unary labelling function $\lambda : V \rightarrow \{0, \dots, k\}$ for some k and a single binary labelling function $E : V^2 \rightarrow \{0, 1\}$. We show how to represent *property graphs* [AAB⁺17] in our model. In this model nodes and edges of a graph can be annotated with properties of the form of key-value pairs. We present an example of a property graph G in Figure 2. The graph G represents a fragment of a map. The nodes represent *places* and the edges represent *links* between the places. Both the nodes and the edges of G contain labels such as S or T and a number of properties, e.g., a type which identifies what kind of a place, or a link, it is. As depicted in Figure 3, and already discussed in Section 1, we represent both nodes and edges of property graphs as nodes in our model. This is hardly a surprise since we adopt a lightweight concept of edges. The binary labelling E we use, yields a value 0 or 1 for a pair of nodes specifying the existence of an edge and does not store any additional information. Note that the key-value pairs are also represented with labellings, in particular each labelling is named after the key.

3. LANGUAGE OPRA

To define OPRA, we first introduce its two fragments, PR and PRA, which can be seen as syntactic restrictions of OPRA. The first fragment, PR, includes *Path constraints* and *Regular constraints*, and the second one, PRA, includes also *Arithmetical constraints*.

The language OPRA will extend PRA with *Ontologies*. For simplicity, we assume that the queries return only node identifiers. For practical purposes this can be straightforwardly extended to returning labels.

We now introduce basic notational conventions we use. By x we denote a node variable, by π a path variable, by \vec{x} and $\vec{\pi}$ tuples of node and path variables respectively, by c an integer value. By λ we denote a labelling; for binary labellings (i.e. with two arguments), we often use symbol E instead of λ . Our conventions are summarised in the following listing.

$x, x_1, \dots \in \text{NodeVariables}$	node variables
$\pi, \pi_1, \dots \in \text{PathVariables}$	path variables
$c, c_1, \dots \in \text{Integers}$	integer values
$E, E_1, \dots \in \text{BinaryLabellings}$	binary labellings
$\lambda, \lambda_1, \dots \in \text{Labellings}$	labellings (of any arity)
$v, t, \dots \in V$	nodes
$p, p_1, \dots \in V^*$	paths
$f, g, f_1, \dots \in \mathcal{F}$	functions

OPRA queries are of the form (red font distinguishes the terminal symbols):

```

Q( $\vec{x}, \vec{\pi}$ ) ::= LET Ontologies IN
SELECT NODES  $\vec{x}$ , PATHS  $\vec{\pi}$ 
SUCH THAT PathConstraints
WHERE RegularConstraints
HAVING ArithmeticalConstraints

```

$\left. \begin{array}{l} \text{PR} \\ \text{PRA} \end{array} \right\} \text{OPRA}$

where \vec{x} are free node variables, $\vec{\pi}$ are free path variables, **PathConstraints** is a conjunction of *path constraints*, **RegularConstraints** is a conjunction of *regular constraints*, **ArithmeticalConstraints** is a conjunction of *arithmetical constraints*, and **Ontologies** is a sequence of *auxiliary labellings definitions* defined in Section 3.3. Either of the tuples $\vec{x}, \vec{\pi}$ can be empty. The conjuncts in these constraints are connected with the keyword **AND**:

```

PathConstraints      ::= PathConstraint [ AND PathConstraints ]
RegularConstraints   ::= RegularConstraint [ AND RegularConstraints ]
ArithmeticalConstraints ::= ArithmeticalConstraint
                        [ AND ArithmeticalConstraints ]

```

The constraints may contain variables not listed in the **SELECT** clause (which are then existentially quantified). Unnecessary components may be omitted (e.g., the keyword **WHERE** if no regular constraints are needed). Each node variable may be also treated as a path variable, representing a single-node path.

3.1. Path and regular constraints. The language PR is a syntactic fragment of OPRA obtained by disallowing the keywords **LET ... IN** and **HAVING**. In other words, PR queries consist of *path constraints*, which involve node and path variables, and *regular constraints*, which involve path variables only. We complete its definition by defining the path constraints and regular constraints.

Syntax. A path constraint is an expressions of the form $x_s \rightarrow_E^\pi x_t$, where x_s, x_t are node variables, π is a path variable, and E is a binary labelling, i.e.,

```
PathConstraint ::=  $x_s \rightarrow_E^\pi x_t$ 
```

where $x_s, x_t \in \text{NodeVariables}$, $\pi \in \text{PathVariables}$, $E \in \text{BinaryLabellings}$.

Formal syntax of regular constraints is defined in the following way.

RegularConstraint	::= ϵ NodeConstraintConj RegularConstraint*
	RegularConstraint + RegularConstraint
	RegularConstraint · RegularConstraint
NodeConstraintConj	::= NodeConstraint [\wedge NodeConstraintConj]
NodeConstraint	::= $\langle \top \rangle$ $\langle \text{NCValue NCOperator NCValue} \rangle$
NCValue	::= c $\lambda(\text{PathPositionVariable}, \dots, \text{PathPositionVariable})$
PathPositionVariable	::= Prev (π) π Next (π)
NCOperator	::= \leq \lt \lt \geq \gt \neq

where $c \in \text{Integers}$, $\pi \in \text{PathVariables}$, $\lambda \in \text{Labelligs}$.

Each node constraint has a natural number parameter k . Such k -node constraint is either $\langle \top \rangle$, denoting a dummy constraint that always holds or an expression of the form $\langle X \Delta X' \rangle$, where $\Delta \in \{\leq, <, =, >, \geq, \neq\}$ and each of X, X' is an integer constant or a labelling function λ applied to some of the variables in Π .

A regular constraint $R(\pi_1, \dots, \pi_k)$ is syntactically a regular expression over an infinite alphabet consisting of conjunctions of all k -node constraints. We write regular expressions with ϵ denoting the empty word, \cdot denoting *concatenation*, $+$ denoting *alternation*, and $*$ denoting *Kleene star*.

Regular constraints are evaluated over paths from SELECT part and paths quantified existentially. In order to allow accessing nodes on a path π , we introduce fresh, free node variables **Prev**(π), π , and **Next**(π). Naturally, **Prev**(π), π and **Next**(π) represent the nodes at the previous, the current and the next position of π accordingly. By Π we denote the set of the variables **Prev**(π), π , **Next**(π) for all paths π of a given query.

Intuitions for Regular Constraints. Before we define the semantics of regular constraints we present an intuition first. An important part of a query language for graphs is the way to specify graph patterns. In PR, we define them using conjunctions of regular constraints. Regular constraints extend the formalism of Regular Path Queries (RPQs) to deal with the values of labelling functions. The idea is different than the one involving Regular Expressions with Memory (REMs) [LMV16]. REMs can store the values at the current position in registers and then test their equality with other values already in registers. Here, we limit PR expressions to access only the nodes at the current, the next and the previous position on each of the paths. Later on, in the last example of Section 4, we show how OPRA enables to compare values of nodes at any distant positions.

Path Constraints Semantics. Given a set of node and path variables \mathcal{V} and a graph G with the nodes V we define a variable instantiation η^G as a function from \mathcal{V} to the nodes and paths of G . We sometimes omit the superscript when the graph is clear from the context and we do not distinguish an instantiation and its canonical extension to tuples of node and path variables.

Given a labelling λ in a graph G and a tuple \vec{v} of nodes of G (of appropriate arity) we say that $G \models \lambda(\vec{v})$ iff $\lambda(\vec{v}) \neq 0$. A path constraint $x_s \rightarrow_{\text{E}}^{\pi} x_t$ holds in a graph G under an instantiation η^G , which we denote by $G, \eta^G \models x_s \rightarrow_{\text{E}}^{\pi} x_t$, iff $\eta^G(\pi)$ is a sequence of nodes $v_0 v_1 \dots v_n$ of G starting from $v_0 = \eta^G(x_s)$, ending in $v_n = \eta^G(x_t)$ and such that for all $i \in \{0, \dots, n-1\}$, $G \models \text{E}(\eta^G(x_i), \eta^G(x_{i+1}))$.

Regular Constraints Semantics. Given paths p_1, \dots, p_k we define $W(p_1, \dots, p_k) \in (V^{3k})^*$ to be the word of the length of the longest path among p_1, \dots, p_k such that for each j we have $W(p_1, \dots, p_k)[j] = (p_1[j-1], p_1[j], p_1[j+1], \dots, p_k[j-1], p_k[j], p_k[j+1])$ (i.e., it is a vector

consisting of three consecutive nodes of each path, substituted by \square if not present). For example, if $p_1 = v_1v_2v_3$ and $p_2 = v_4v_5$ then $W(p_1, p_2) = ((\square, v_1, v_2, \square, v_4, v_5), (v_1, v_2, v_3, v_4, v_5, \square), (v_2, v_3, \square, v_5, \square, \square))$.

Intuitively, the semantics is given by applying the specified labelling functions to the nodes in the given vector and comparing the values according to the Δ symbol. A node constraint may be seen as a function that takes a vector of $3k$ nodes (i.e., $W(p_1, \dots, p_k)[j]$, if j is the current position on the paths), represented by the variables in Π , and returns a Boolean value.

For an example consider a conjunction c of two node constraints

$$\langle \lambda(\pi_1) > \lambda(\mathbf{Next}(\pi_1)) \wedge \lambda(\mathbf{Prev}(\pi_1)) = \lambda(\mathbf{Next}(\pi_2)) \rangle,$$

let i be the current position and for some η let

$$W(\eta(\pi_1), \eta(\pi_2))[i] = ((v_{i-1}^1, v_i^1, v_{i+1}^1), (v_{i-1}^2, v_i^2, v_{i+1}^2)).$$

Then c requires that $v_i^1 > v_{i+1}^1$ and $v_{i-1}^1 = v_{i+1}^2$.

Given a graph G over vertices V , the language of R , denoted as $L^G(R)$, is a subset of $(V^{3k})^*$ defined using the usual rules: $L^G(\epsilon)$ is the empty word, for a conjunction of k -node constraints r , $L^G(r)$ is the set of all vectors of length $3k$ for which r returns true, and $L^G(R \cdot R')$, $L^G(R + R')$ and $L^G(R^*)$ are defined inductively as the concatenation, the union and Kleene star closure of appropriate languages.

Given a graph G and an instantiation η^G we say that $R(\pi_1, \dots, \pi_k)$ holds in G under η^G , $G, \eta^G \models R(\pi_1, \dots, \pi_k)$, if and only if $W(\eta^G(\pi_1), \dots, \eta^G(\pi_k)) \in L^G(R)$.

Example. Consider the query:

```
SELECT NODES  $x, y$ 
SUCH THAT  $x \rightarrow_{\mathbf{E}}^{\pi} y$ 
WHERE  $\langle \mathbf{E}(\mathbf{Next}(\pi), \pi) \rangle^* \langle \top \rangle$  AND  $\langle \lambda(\pi) > 0 \rangle^*$ 
```

Let G be a graph and let η^G be an instantiation. Let $v = \eta^G(x)$ and $w = \eta^G(y)$. The query holds in G iff there is a bidirectional path between v and w whose each node is labelled by λ with a positive number. Notice that $\langle \top \rangle$ is required as $\mathbf{Next}(\pi)$ is \square for the last node. If G has an additional binary labelling \mathbf{E}^{-1} such that for all nodes v, w we have $\mathbf{E}^{-1}(v, w) = \mathbf{E}(w, v)$, then the same query can be stated as follows:

```
SELECT NODES  $x, y$ 
SUCH THAT  $x \rightarrow_{\mathbf{E}}^{\pi} y$  AND  $x \rightarrow_{\mathbf{E}^{-1}}^{\pi} y$ 
WHERE  $\langle \lambda(\pi) > 0 \rangle^*$ 
```

Note that the existentially quantified path π is mentioned in both of the path constraints above. We discuss in Section 3.3 how to define \mathbf{E}^{-1} as an auxiliary labelling based on \mathbf{E} .

Query semantics. Let $Q(\vec{x}, \vec{\pi})$ be a PR query. Given a graph G and tuples of nodes \vec{v} and paths \vec{p} of G we say that $Q(\vec{v}, \vec{p})$ holds in G , denoted by $G \models Q(\vec{v}, \vec{p})$, if and only if there exists an instantiation η^G of all free and existential node and path variables of Q such that $\eta^G(\vec{x}) = \vec{v}$, $\eta^G(\vec{\pi}) = \vec{p}$ and all path and regular constraints of Q hold for G under η^G .

3.2. Arithmetical constraints. The syntax for arithmetical constraints is defined as follows.

```

ArithmeticalConstraint ::= ArithmeticalSum ≤ d
ArithmeticalSum       ::= c ArithmeticalAtom [ + ArithmeticalSum ]
ArithmeticalAtom      ::= λ [ π1, ..., πn ]

```

where $c \in \text{Integers}$, $d \in \text{Integers}$ or d is a parameterless labelling, $\pi_1, \dots, \pi_n \in \text{PathVariables}$ and $\lambda \in \text{Labellings}$.

The language PRA is a syntactic fragment of OPRA obtained by disallowing the keyword `LET ... IN`, i.e., it extends PR with arithmetical constraints, which we define below.

An *arithmetical atom* is of the form $\lambda[\pi_1, \dots, \pi_k]$, where λ is a labelling. An *arithmetical constraint* is an inequality $c_1\Lambda_1 + \dots + c_j\Lambda_j \leq d$, where c_1, \dots, c_j are integer constants, d is either an integer constant or a parameterless labelling, and each Λ_ℓ is an arithmetical atom.

Semantics. Let Λ be an arithmetical atom $\lambda[\pi_1, \dots, \pi_k]$ and let G be a graph. Consider an instantiation η^G of path variables π_1, \dots, π_k with paths p_1, \dots, p_k . The value of Λ under η^G , denoted by $\text{val}_{\eta^G}(\Lambda)$, is defined as

$$\sum_{i=1}^s \lambda(p_1[i], \dots, p_k[i])$$

where $s = \max(|p_1|, \dots, |p_k|)$.

The arithmetical constraint $c_1\Lambda_1 + \dots + c_j\Lambda_j \leq d$ holds in G under η^G iff the value $c_1\text{val}_{\eta^G}(\Lambda_1) + \dots + c_j\text{val}_{\eta^G}(\Lambda_j)$ is less than or equal to d .

Example. Consider a class of graphs with an edge labelling E , a unary labelling One , which returns 1 for all nodes, and a unary labelling λ . The following query holds in a graph G under an instantiation η^G when the nodes $\eta^G(x), \eta^G(y)$ are connected by a path $p = \eta^G(\pi)$ such that each node of p is labelled by λ with a positive number and the average value of λ over all nodes is at most 5.

```

SELECT NODES x, y
SUCH THAT x →Eπ y
WHERE ⟨λ(π) > 0⟩*
HAVING λ[π] - 5 · One[π] ≤ 0

```

Query semantics. The query semantics is defined in a virtually the same way as for PR queries; we additionally require the instantiation to satisfy all arithmetical constraints of the query.

3.3. Auxiliary labellings. The language OPRA extends PRA with the constructions that define auxiliary labellings of graphs, which are defined based on existing graph labellings and its structure.

```

Ontologies           ::= LabellingDef [ , Ontologies ]
LabellingDef         ::= λ(x̄) := LabellingDefTerm(x̄)
LabellingDefTerm(x̄) ::= c | λ(ȳ) | x1 = x2 | [Q(x̄)] |
                    f(LabellingDefTerm(ȳ1), ..., LabellingDefTerm(ȳk)) |
                    minλ, π Q(x̄, π) | maxλ, π Q(x̄, π) |
                    f({ LabellingDefTerm(x) : LabellingDefTerm(x, ȳ)})

```

where Q is an OPRA query, c is an integer, $-\infty$ or $+\infty$, $\pi \in \text{PathVariables}$, x is a fresh

node variable (not in \vec{x}), \vec{x} is a tuple of node variables, x_1, x_2 are variables from \vec{x} and tuples $\vec{y}, \vec{y}_1, \dots, \vec{y}_k$ consist of the variables from \vec{x} , $\lambda \in \mathbf{Labellings}$.

The ability to define auxiliary labellings significantly extends the expressive power of the language. The essential property of auxiliary labellings is that their values do not need to be stored in the database, which would require polynomial memory, but can be computed *on demand*. An auxiliary labelling may be seen as an *ontology* or a *view*.

Ontologies is of the form $\lambda_1(\vec{x}_1):=t_1, \dots, \lambda_n(\vec{x}_n):=t_n$. Such a sequence defines auxiliary labellings $\lambda_1, \dots, \lambda_n$ that can be used in the PRA query and also in the following labellings, i.e., λ_i can be used in λ_j if $i < j$. The labellings are defined by means of *terms* t_1, \dots, t_n , which are expressions with free variables $\vec{x}_1, \dots, \vec{x}_n$.

To define terms, we assume a set $\mathcal{F} = \mathcal{F}_A \cup \mathcal{F}_b$ of *fundamental functions*

$$f : (\mathbb{Z} \cup \{-\infty, \infty\})^* \rightarrow \mathbb{Z} \cup \{-\infty, \infty\},$$

where \mathcal{F}_A is a set of *aggregate functions* including maximum MAX, minimum MIN, counting COUNT, summation SUM, and \mathcal{F}_b is a set of *binary functions* including $+$, $-$, \cdot and \leq . The set \mathcal{F} can be extended, if needed, by any functions computable by a non-deterministic Turing machine (see Remark 8.3) whose size of all tapes while computing $f(\vec{x})$ is logarithmic in length of \vec{x} and values in \vec{x} , assuming binary representation, provided that additional aggregate functions in \mathcal{F} are invariant under permutation of arguments.

We distinguish four types of terms.

Basic terms. There are three basic types of terms: a constant c , a labelling value $\lambda(\vec{y})$, and a node identity test $y = y'$, where y, y' are node variables, \vec{y} is a vector of node variables, c is a constant, λ is a labelling.

Function application. A term can be a function application to another terms: $f(t_1, \dots, t_n)$, where $f \in \mathcal{F}$ and t_1, \dots, t_n are terms.

Subqueries. There are two essential constructs involving subqueries in terms. First, we can evaluate the truth value of a subquery, i.e., the expression $[Q(\vec{y})]$, where Q is an OPRA query with \vec{y} being free node variables. Note that \vec{y} occur in the main query and hence they become instantiated in the subquery. Therefore, $[Q(\vec{y})]$ returns the value 1 if under the current instantiation of \vec{y} , query $Q(\vec{y})$ holds, and 0 otherwise. Second, we minimise (resp., maximise) the value of a parameter satisfying a subquery, i.e., the value of the expression $\min_{\lambda, \pi} Q(\vec{y}, \pi)$ (resp. $\max_{\lambda, \pi} Q(\vec{y}, \pi)$), where Q is an OPRA query with free node variables \vec{y} and a single free path variable π , to obtain the minimum (resp. maximum) of values $\lambda[\pi]$ over paths satisfying Q (as usual, the minimum of the empty set is $+\infty$ and maximum of the empty set is $-\infty$). The expression $\lambda[\pi]$ denotes as in the arithmetical constraints the value of the labelling λ applied to the path π (i.e., the sum of the values of λ along π).

Aggregative properties. The last type of term allows us to apply a function to a set of labels of all nodes *satisfying* some label. The syntax is $f(\{t(x):t'(x, \vec{y})\})$, where $f \in \mathcal{F}_A$ is an aggregate function. The value can be defined as follows: first, we compute the set $X = \{x_1, \dots, x_s\}$ of nodes such that for every $x \in X$ we have $t'(x, \vec{y}) = 1$. Then, the value of the term is the value of $f(t(x_1), t(x_2), \dots, t(x_s))$. Notice that since f is an aggregate function, the value does not depend on the order among x_1, \dots, x_s .

Formal semantics. The semantics of terms is as follows. Let G be a graph and let η^G be an instantiation. In Table 1 we inductively extend instantiations to terms. If G is clear from the context, we write $t(\vec{v})$ as a shorthand of $\eta^G(t(\vec{x}))$, where $\eta^G(\vec{x}[i]) = \vec{v}[i]$ for all i .

Term t	The value of $\eta^G(t)$
c	c , where c is a constant
$\lambda(\vec{y})$	$\lambda(\eta^G(\vec{y}))$, where λ is a labelling of G
$[Q(\vec{y})]$	1 if $Q(\eta^G(\vec{y}))$ holds in G and 0 otherwise,
$\min_{\lambda, \pi} Q(\vec{y}, \pi)$	the minimum of values of $\lambda[p]$, defined as in the arithmetical constraints, over all paths p such that $Q(\eta^G(\vec{y}), p)$ holds in G ,
$\max_{\lambda, \pi} Q(\vec{y}, \pi)$	the maximum of values of $\lambda[p]$, defined as in the arithmetical constraints, over all paths p such that $Q(\eta^G(\vec{y}), p)$ holds in G ,
$y = y'$	1 if $\eta^G(y) = \eta^G(y')$ and 0 otherwise (a node identity check),
$f(t_1(\vec{y}_1), \dots, t_k(\vec{y}_k))$	the value of $f(\eta^G(t_1(\vec{y}_1)), \dots, \eta^G(t_k(\vec{y}_k)))$,
$f(\{t(x):t'(x, \vec{y})\})$	the value of $f(t(v_1), \dots, t(v_n))$, where v_1, \dots, v_n are all nodes v of G satisfying $t'(v, \eta^G(\vec{y})) = 1$.

TABLE 1. Value of terms.

Auxiliary labellings. Consider a term $t(\vec{x})$ of an arity k and a graph G , which does not have a labelling λ . We define the graph $G[\lambda:=t]$ as the graph G extended with the labelling λ of an arity k such that $\lambda(\vec{v}) = t(\vec{v})$ for all $\vec{v} \in V^k$. We call λ an *auxiliary labelling* of G . We write $G[\lambda_1:=t_1, \dots, \lambda_n:=t_n]$ to denote the results of successively adding labellings $\lambda_1, \dots, \lambda_n$ to the graph G , i.e., $((G[\lambda_1:=t_1])[\lambda_2:=t_2]) \dots [\lambda_n:=t_n]$.

Size of an OPRA query. The size of a query Q of the form `LET 0 IN Q'` is the sum of binary representations of terms t_1, \dots, t_n in `0` and the size of the query Q' .

Example. We can define labellings `E-1` and `One` presented in the above examples with terms `E-1(x, y) := E(y, x)` and `One(x) := 1`. The example from Section 3.2 can be stated for graphs without `One` labelling as follows:

```
LET One(x) := 1 IN
SELECT NODES x, y
SUCH THAT x →Eπ y
WHERE ⟨λ(π) > 0⟩
HAVING λ[π] - 5 · One[π] ≤ 0
```

Semantics of OPRA. Let $Q(\vec{v}, \vec{p})$ be of the form `LET 0 IN Q'`. Given a graph G and tuples of nodes \vec{v} and paths \vec{p} of G we say that $Q(\vec{v}, \vec{p})$ holds in G , denoted by $G \models Q(\vec{v}, \vec{p})$ if and only if $G[O] \models Q'(\vec{v}, \vec{p})$. Note that $Q'(\vec{x}, \vec{\pi})$ is a PRA query, which can refer to auxiliary labellings $\lambda_1, \dots, \lambda_n$.

4. EXAMPLES

We focus on the following scenario: a graph database that corresponds to a map of some area. Each graph's node represents either a *place* or a *link* from one place to another. The graph has four unary labellings and one binary labelling. The labelling type represents the type of a place for places (e.g., square, park, pharmacy) or the mode of transport for links (e.g., walk, tram, train); we assume each type is represented by a constant, e.g., c_{square} , c_{park} . The labelling attr represents attractiveness (which may be negative, e.g., in unsafe areas), and time represents time. The binary labelling E represents edges: for nodes v_1, v_2 , the value $E(v_1, v_2)$ is 1 if there is an edge from v_1 to v_2 and 0 otherwise. For example, the graph on Fig. 3 represents a map with two places: S is a square and P is a park. There are three nodes representing links: node W represents moving from S to P by walking, T moving from S to P by tram and B moving from P to S by bus.

4.1. Language PRA. Sums. The language PRA can express properties of paths' sums. Consider the query $Q_1(x, y)$ below. For nodes s, t , the query $Q_1(s, t)$ holds iff there is a route from s to t that takes at most 6 hours and its attractiveness is over 100. $Q_1(x, y) = \text{SELECT NODES } (x, y) \text{ SUCH THAT } x \rightarrow_{\text{E}}^{\pi} y \text{ HAVING } \text{time}[\pi] \leq 360 \wedge \text{attr}[\pi] > 100$

Furthermore, we can compute averages, to some extent. For example, the following arithmetical constraint says that for some path π the average attractiveness of π is at least 4 attractiveness points per minute: $\text{attr}[\pi] \geq 4\text{time}[\pi]$.

Multiple paths. We define a query $Q_2(x, y)$ that asks whether there is a route from x to y , such that from every place we can take a tram (e.g., if it starts to rain). We express that by stipulating a route π from x to y and a sequence ρ of tram links, such that every node of π representing a place is connected with the corresponding tram link in ρ . In a way, ρ works as an existential quantifier for nodes of π .

$Q_2(x, y) = \text{SELECT NODES } (x, y) \text{ SUCH THAT } x \rightarrow_{\text{E}}^{\pi} y \text{ WHERE } \langle \text{type}(\rho) = c_{\text{tram}} \rangle^* \wedge \text{link}(\pi, \rho)$

where the parameterized macro $\text{link}(\pi, \rho)$ is defined as

$$(\langle \text{type}(\pi) = c_{\text{bus}} \rangle + \langle \text{type}(\pi) = c_{\text{walk}} \rangle + \langle \text{type}(\pi) = c_{\text{tram}} \rangle + \langle E(\pi, \rho) = 1 \rangle)^*$$

states that every node of the first path either is not a place, i.e, it represents any of possible links (by a bus, a walk or a tram), or is connected with the corresponding node of the second path.

4.2. Language OPRA. We show how to employ auxiliary labellings in our queries. For readability, we introduce some syntactic sugar — constructions which do not change the expressive power of OPRA, but allow queries to be expressed more clearly. We use the function symbols $=, \neq$ and Boolean connectives, which can be derived from \leq and arithmetical operations. Also, we use terms $t(x, y)$ in arithmetical constraints, which can be expressed by first defining the labelling $\lambda(x, y) := t(x, y)$, defining additional paths $\rho_1 = x, \rho_2 = y$ of length 1, and using $\lambda[\rho_1, \rho_2]$.

Processed labellings. Online route planners often allow looking for routes which do not require much walking. The query $Q_3(x, y)$ asks whether there exists a route from x to y such that the total walking time is at most 10 minutes. To express it, we define a labelling $t_{\text{walk}}(x)$, which is the time of x , if it is a walking link, and 0 otherwise.

```

 $Q_3(x, y) = \text{LET } t\_walk(x) := (\text{type}(x) = c_{walk}) \cdot \text{time}(x) \text{ IN}$ 
 $\text{SELECT NODES } (x, y) \text{ SUCH THAT } x \rightarrow_E^\pi y$ 
 $\text{HAVING } t\_walk[\pi] \leq 10$ 

```

Nested queries. It is often advisable to avoid crowded places, which are usually the most attractive places. We write a query that holds for routes that are always at least 10 minutes away from any node with attractiveness greater than 100. We define a labelling `crowded(x)` as

```

 $Q_4(x) = [\text{SELECT NODES } (x) \text{ SUCH THAT } x \rightarrow_E^\pi y$ 
 $\text{WHERE } \langle \tau \rangle^* \langle \text{attr}(\pi) > 100 \rangle \text{ HAVING } \text{time}[\pi] \leq 10]$ 

```

Notice that the variables π and y are existentially quantified. We check whether the value of `crowded` is 0 for each node of the path π .

```

 $Q_5(\pi) = \text{SELECT PATHS } (\pi) \text{ SUCH THAT } x \rightarrow_E^\pi y \text{ WHERE } \langle \text{crowded}(\pi) = 0 \rangle^*$ 

```

Nodes' neighbourhood. “Just follow the tourists” is an advice given quite often. With OPRA, we can verify whether it is a good advice in a given scenario. A route is called *greedy* if at every position, the following node on the path is the most attractive successor. We define a labelling `MAS(x, y)` that is 1 if y is the most attractive successor of x , and 0 otherwise, and use it to express that there is a greedy route from a node x to a node y .

```

 $Q_6(x, y) = \text{LET } \text{MAS}(x, y) := (\text{COUNT}(\{\text{attr}(z): E(x, z) \wedge \text{attr}(z) \geq \text{attr}(y)\}) = 1) \text{ IN}$ 
 $\text{SELECT NODES } (x, y) \text{ SUCH THAT } x \rightarrow_E^\pi y$ 
 $\text{WHERE } \langle \text{MAS}(\pi, \text{Next}(\pi)) = 1 \rangle^* \langle \tau \rangle$ 

```

The above query considers routes that in each place select the most attractive link, and in each link select the most attractive place. What if we are interested in the attractiveness of the places only? If we assume that x and y are places and all edges are between places and links only, then this can be achieved by replacing the `WHERE` clause of the above query by:

```

 $\text{WHERE } (\langle \tau \rangle \langle \text{MAS}'(\text{Prev}(\pi), \text{Next}(\pi)) = 1 \rangle)^* \langle \tau \rangle$ 

```

where `MAS'` is a counterpart of `MAS` obtained by replacing $E(x, z)$ by $E(x, x') \wedge E(x', z)$. Such a query checks at every link that the following place is the most attractive place that can directly reached from the previous place.

Properties of paths' lengths. In route planning, we can optimize different aspects such as time, necessary budget or attractiveness. We can express the conjunction of such objectives, i.e., specify routes that are optimal for several objectives. The following query asks whether is it possible to get from s to t with a route that takes the shortest time among all routes, and at the same time it maximises the attractiveness among all routes.

```

 $Q_8(x, y) = \text{SELECT NODES } (x, y) \text{ SUCH THAT } x \rightarrow_E^\pi y$ 
 $\text{HAVING } (\text{attr}[\pi] = \max_{\text{attr}, \rho} Q_{\text{route}}(x, y, \rho)) \wedge (\text{time}[\pi] = \min_{\text{time}, \rho} Q_{\text{route}}(x, y, \rho))$ 

```

where $Q_{\text{route}}(x, y, \rho)$ stands for `SELECT NODES (x, y), PATHS ρ SUCH THAT $x \rightarrow_E^\rho y$` .

Registers. Registers are an important concept in graph query languages. For instance, to express that two paths have a non-empty intersection, we load a (non-deterministically picked) node to a register and check whether it occurs in both paths. This can be achieved by the following query.

```

 $Q_9(\pi_1, \pi_2) = \text{LET } \text{loop}(x, y) := x = y \text{ IN}$ 
SELECT PATHS  $\pi_1, \pi_2$ 
SUCH THAT  $z \xrightarrow{\pi}_{\text{loop}} z$ 
WHERE  $\text{same}(\pi_1, \pi) \wedge \text{same}(\pi_2, \pi)$ 

```

where $\text{same}(\rho_1, \rho_2) = \langle \top \rangle^* \langle \rho_1 = \rho_2 \wedge \rho_1 \neq \square \rangle \langle \top \rangle^*$. This query uses a constant register π .

The following query asks whether there exists a route from a club s to a club t on which the attractiveness of visited clubs never decreases. In the register-based approach, we achieve this by storing the most recently visited club in a separate register. Here, we express this register using an additional path ρ , storing the values of the register, a labelling $r(x', y, y')$ which states that $y' = x'$ if x' is a club, and $y' = y$ otherwise, and an auxiliary labelling \oplus which is true for all the pairs of nodes.

```

 $Q_{10}(x, y) = \text{LET } \oplus(x, y) := 1,$ 
 $r(x', y, y') := (\text{type}(x') = c_{\text{club}} \Rightarrow y' = x') \wedge (\text{type}(x') \neq c_{\text{club}} \Rightarrow y = y') \text{ IN}$ 
SELECT NODES  $(x, y)$  SUCH THAT  $x \xrightarrow{\pi_E} y \wedge x \xrightarrow{\oplus} y$ 
WHERE  $\text{ends}(\pi) \wedge \text{regs}(\pi, \rho) \wedge \text{inc}(\rho)$ 

```

where the macro $\text{ends}(\pi) = \langle \text{type}(\pi) = c_{\text{club}} \rangle \langle \top \rangle^* \langle \text{type}(\pi) = c_{\text{club}} \rangle$ states that both ends of a path are clubs, $\text{regs}(\pi, \rho) = \langle r(\text{Next}(\pi), \rho, \text{Next}(\rho)) = 1 \rangle^* \langle \top \rangle$ ensures that at each position the second path contains the most recently visited club along the first path, and the part $\text{inc}(\rho) = \langle \text{attr}(\rho) \leq \text{attr}(\text{Next}(\rho)) \rangle^* \langle \top \rangle$ checks that the attractiveness never decreases.

5. G-CORE, CYPHER AND OPRA

In this section we compare constructs of G-Core [AAB⁺18], Cypher [The18] and OPRA.

All three formalisms have SQL-like clauses and rely crucially on matching of graph patterns. Such patterns are specified in G-Core and Cypher using ASCII-art syntax, e.g., the pattern

```
(n)-[:connection]->(m)->n
```

binds the variables n, m to the pairs of nodes (a, b) such that there is a directed edge labelled **connection** from a to b and there is also some edge going back. Patterns can be fixed structures (*rigid* patterns), consisting of the exact graph that should match the input graph but also may be specified by navigational paths using regular expressions. In OPRA we provide two options: path constraints $x_s \xrightarrow{\pi_E} x_t$ that essentially define reachability over a binary labelling that encodes edges, as well as powerful regular constraints.

The process of matching patterns generates tuples of matched node, edge and path values that can be filtered by WHERE conditions. Technically, a graph-pattern match corresponds to a homomorphism from a query Q to the input graph G . All three formalisms allow for an unconstrained semantics where the multiple variables may match the same value (a node or an edge). The default matching semantics in CYPHER is, however, *no-repeated-edge semantics* [AAB⁺17] where variables corresponding to edges have to map one-to-one and the other variables need not to be mapped injectively. Each of the languages may produce an enumeration of all matched tuples as well as of some of the projections on subsets of variables. G-Core and CYPHER support shortest paths (hop count) matching, G-Core allows also for weighted shortest paths where the costs may be specified using positive weights.

In CYPHER and G-Core the stream of tuples generated by pattern matching can be aggregated and then returned to be processed in the following parts of a query. For example, consider the following CYPHER query [AAB⁺17] to find the longest movies in a collection.

```
MATCH (m:Movie) WITH MAX(m.runtime) AS maxTime
MATCH (m:Movie) WHERE m.runtime = maxTime
RETURN m
```

The first pattern `MATCH (m:Movie)` matches all movies, aggregates their lengths and return the maximal length using `WITH` clause. Then, the second `MATCH (m:Movie)` again matches all movies but this time, however, it filters out the ones with runtime not equal to `maxTime`.

Although we do not allow in OPRA for an aggregation at this stage we can reformulate such queries as follows.

```
LET maxTime() = MAX(runtime(x) | type(x) = cmovie) IN
SELECT NODES (x)
WHERE (maxTime() = runtime(x))
```

Recall that here we use the fact that each node variable may be also treated as a path variable that represents a single-node path.

The specific thing for OPRA is the ability to compare paths in terms of regular relations [BLLW12]. Regular relations that can be specified in regular constrains include path equality, prefix (i.e., is a path a prefix of another?), length comparisons, fixed-edit distance, synchronous transformation.

G-Core and OPRA have tractable data complexity, on the other hand there are at least two reasons for which data complexity of CYPHER is NP-hard. We have already mentioned that CYPHER has no-repeated-edge semantics for graph pattern matching. This makes the evaluation intractable [MW95]. The second reason is its ability to unwind paths, that is to return path elements (e.g. nodes) and then to process them. This feature may be used [AAB⁺17] to write a fixed query that returns two different disjoint routes between given two nodes which is also an NP-hard problem. Note that this is precisely the reason for which we do not allow in OPRA nested queries with free *path* variables (only node variables are allowed). We discuss this topic in Section 6.

G-Core and CYPHER have a number of features that are not present in OPRA. In particular, in OPRA there are no features allowing for any modification of graphs nor their data values. We can only define new labellings and then use them in the following part of queries. It is also not possible to construct and return graphs (and thus queries are not composable).

6. CLOSURE PROPERTIES

In this section we discuss closure properties of OPRA under standard set-theoretic operations. We define these operations formally as follows. First, for a query $Q(\vec{x}, \vec{\pi})$ and a graph G we define the *result* of Q on G , denoted by $Q[G]$, as

$$Q[G] = \{(\vec{v}, \vec{p}) \mid \vec{v} \in (V \setminus \{\square\})^{|\vec{x}|}, \vec{p} \in ((V \setminus \{\square\})^*)^{|\vec{\pi}|}, G \models Q(\vec{v}, \vec{p})\}$$

To avoid problems with the artificial node \square , which is used to align paths of different lengths, we ignore it in $Q[G]$.

- a query $Q_{\exists}(x_{i_1}, \dots, x_{i_k}, \pi_{j_1}, \dots, \pi_{j_l})$ is a *projection* of $Q(\vec{x}, \vec{\pi})$ if and only if i_1, \dots, i_k are distinct indices from $\{1, \dots, |\vec{x}|\}$, j_1, \dots, j_l are distinct indices from $\{1, \dots, |\vec{\pi}|\}$, and for every graph G we have $Q_{\exists}[G] = \{(v_{i_1}, \dots, v_{i_k}, p_{j_1}, \dots, p_{j_l}) \mid (\vec{v}, \vec{p}) \in Q[G]\}$,
- a query $Q_{\cap}(\vec{x}, \vec{\pi})$ is an *intersection* of $Q_1(\vec{x}, \vec{\pi})$ and $Q_2(\vec{x}, \vec{\pi})$ if and only if for every graph G we have $Q_{\cap}[G] = Q_1[G] \cap Q_2[G]$,
- a query $Q_{\cup}(\vec{x}, \vec{\pi})$ is a *union* of $Q_1(\vec{x}, \vec{\pi})$ and $Q_2(\vec{x}, \vec{\pi})$ if and only if for every graph G we have $Q_{\cup}[G] = Q_1[G] \cup Q_2[G]$,
- a query $Q_{\times}(\vec{x}, \vec{x}', \vec{p}, \vec{p}')$ is a *Cartesian product* of $Q_1(\vec{x}, \vec{\pi})$, $Q_2(\vec{x}, \vec{\pi})$ if and only if \vec{x} and \vec{x}' are disjoint, $\vec{\pi}$ and $\vec{\pi}'$ are disjoint, and for every graph G we have

$$Q_{\times}[G] = \{(\vec{v}, \vec{v}', \vec{p}, \vec{p}') \mid (\vec{v}, \vec{p}) \in Q_1[G], (\vec{v}', \vec{p}') \in Q_2[G]\}$$

- a query $Q_C(\vec{x}, \vec{\pi})$ is a *complement* of Q if and only if

$$Q_C[G] = (V \setminus \{\square\})^{|\vec{x}|} \times ((V \setminus \{\square\})^*)^{|\vec{\pi}|} \setminus Q[G].$$

Theorem 6.1. *Given OPRA queries Q_1, Q_2 , we can compute in polynomial time every projection of Q_1 , the intersection, the union, and the Cartesian product of Q_1 and Q_2 . If Q_1 has no free path variables, then we can compute in polynomial time the complement of Q_1 .*

Proof of Theorem 6.1. The projection case is straightforward — a projection of a query can be obtained by simply not listing the unwanted variables in the `SELECT` statement.

To define the complement we simply use Q_1 as a subquery (we can do that only for queries without free path variables).

```
LET C( $\vec{x}$ ) := [Q1( $\vec{x}$ )] IN SELECT NODES  $\vec{x}$  HAVING C[ $\vec{x}$ ]=0
```

Having queries Q_1, Q_2 with only node variables, we can give similar constructions for the cases of a Cartesian product, an intersection and a union. For queries with free path variables, the constructions are more complex.

Assume that, for $i = 1, 2$ the query Q_i is of the form

```
LET  $O_i$  IN
SELECT NODES  $\vec{x}_i$ , PATHS  $\vec{\pi}_i$ 
SUCH THAT  $P_i$ 
WHERE  $R_i$ 
HAVING  $A_i$ 
```

Without loss of generality, we assume that quantified variables in Q_1 and Q_2 are disjoint (if not, we simply rename the conflicting entities).

For the Cartesian-product case, we assume that nodes variables \vec{x}_1 and \vec{x}_2 are disjoint, and path variables $\vec{\pi}_1$ and $\vec{\pi}_2$ are disjoint as well. Then, the following query expresses Q_{\times} :

```
LET  $O_1, O_2$  IN
SELECT NODES  $\vec{x}_1, \vec{x}_2$ , PATHS  $\vec{\pi}_1, \vec{\pi}_2$ 
SUCH THAT  $P_1$  AND  $P_2$ 
WHERE  $R_1$  AND  $R_2$ 
HAVING  $A_1$  AND  $A_2$ 
```

For the intersection case, we assume that $\vec{x}_1 = \vec{x}_2$ and $\vec{\pi}_1 = \vec{\pi}_2$. We construct the query Q_{\cap} as follows:

```

LET  $O_1, O_2$  IN
SELECT NODES  $\vec{x}_1$ , PATHS  $\vec{\pi}_1$ 
SUCH THAT  $P_1$  AND  $P_2$ 
WHERE  $R_1$  AND  $R_2$ 
HAVING  $A_1$  AND  $A_2$ 

```

Finally, the union case is the most difficult one. Roughly speaking, we want to do the Cartesian product of two queries and define the result as the union of the projections of this product. Assuming, without the loss of generality, that the variables in the queries are disjoint, this can be achieved in the following way.

```

LET  $O_1, O_2$  IN
SELECT NODES  $\vec{x}$ , PATHS  $\vec{\pi}$ 
SUCH THAT  $P_1$  AND  $P_2$ 
WHERE  $R_1$  AND  $R_2$  AND EQ
HAVING  $A_1$  AND  $A_2$ 

```

where $\vec{x}, \vec{\pi}$ are fresh variables. The regular constraint EQ guarantees that either ($\vec{x} = \vec{x}_1$ and $\vec{\pi} = \vec{\pi}_1$), or ($\vec{x} = \vec{x}_2$ and $\vec{\pi} = \vec{\pi}_2$). It can be defined in an OPRA query in a straightforward way using a regular constraint with alternation and a new auxiliary labelling defined with a node identity check; notice that the definition will depend on the arity of the vectors.

Notice however, if one of the queries is empty, the Cartesian product is empty, and this naive approach fails.

To avoid this problem, we first define, for each $i \in \{1, 2\}$, an additional labelling $\lambda_{Q_i} = [Q_i]$. By definition, λ_{Q_i} is 0 if Q_i returns the empty result and 1 otherwise. Then, for each i , we define R'_i based on R_i in the following way: for each conjunct r of R_i , R'_i contains $r + \langle \lambda_{Q_i} = 0 \rangle^*$. Finally, for each i , we define A'_i based on A_i in the following way: for each conjunct $\sum_i s_i < d$ of A_i , A'_i contains the conjunct $\sum_i s_i < \lambda_d$, where λ_d is an auxiliary labelling equal to d if $\lambda_{Q_i} = 1$ and ∞ otherwise. This can be defined as $\min(d, (2\lambda_{Q_i} - 1) \cdot \infty)$.

The above definition means that R'_i and A'_i are trivially satisfied if $\lambda_{Q_i} = 0$. Let O be the definitions of the auxiliary labellings described in the paragraphs above. Putting it all together, we obtain:

```

LET  $O, O_1, O_2$  IN
SELECT NODES  $\vec{x}$ , PATHS  $\vec{\pi}$ 
SUCH THAT  $P_1$  AND  $P_2$ 
WHERE  $R'_1$  AND  $R'_2$  AND EQ
HAVING  $A'_1$  AND  $A'_2$ 

```

□

In Theorem 6.1 the construction for the complement is given only for OPRA queries without free path variables. We show that assuming that $NL \neq NP$, OPRA queries are not closed under the complement. Indeed, we show in the following Lemma 6.2 that if all OPRA queries are closed under the complement, then there exists a boolean query Q_{ham} which holds if there is a Hamiltonian cycle in a graph. However, we show in Theorem 8.1 that for a fixed query, the query evaluation problem is NL-complete. Therefore, having the query Q_{ham} , we can decide the existence of a Hamiltonian cycle in a given G in NL and hence $NL = NP$.

Lemma 6.2. *Assume that all OPRA queries are closed under the complement. Then, there exists a boolean OPRA query Q_{ham} such that for every graph G , the query Q_{ham} holds in G if and only if G has a Hamiltonian cycle.*

Proof. The query Q_{ham} is the conjunction of the following queries with a free path variable π that becomes existentially quantified in Q_{ham} :

- $Q_{\text{len}}(\pi)$ which holds for the cycles connected by E with the length equal to the number of all nodes in a graph, and
- $Q_{\text{unique}}(\pi)$ which holds for the paths in which all nodes are different.

Note that such paths are Hamiltonian cycles.

The query $Q_{\text{len}}(\pi)$ is as follows

```
LET One(x) := 1, Nodes(x) := sum({One(y) :  $\tau$ }) IN
SELECT PATHS  $\pi$ 
SUCH THAT  $x \rightarrow_{\pi}^E x$ 
HAVING One[ $\pi$ ] - Nodes[ $y$ ] = 0
```

Note the use of an existantial variable y and how we count the number of all nodes in a graph with the Nodes labelling.

Now, we construct the query $Q_{\text{unique}}(\pi)$. First, we express $Q_{\text{repeats}}(\pi)$ that holds for paths with some node occuring at least twice.

```
LET loop(x, y) :=  $x = y$  IN
SELECT PATHS  $\pi$ 
SUCH THAT  $z \rightarrow_{\text{loop}}^{\pi'} z$ 
HAVING loop[ $\pi, \pi'$ ]  $\geq 2$ 
```

It states that π' consists of the same node u repeated multiple times and we require that there are at least two positions i in π such that $u = \pi'[i] = \pi[i]$.

Finally, as we assume that all OPRA queries are closed under the complement, there exists a query $Q_{\text{unique}}(\pi)$ in OPRA that is the complement of $Q_{\text{repeats}}(\pi)$. \square

Here are some examples of employing the closure properties. To check whether a given graph is a *directed acyclic graph*, we have to check that the graph has no cycle. Instead, we can check whether the graph has a cycle using the following query and then complement this query:

```
SELECT () SUCH THAT  $x \rightarrow_{\pi}^E x$  WHERE  $\langle \tau \rangle \langle \tau \rangle \langle \tau \rangle^*(\pi)$ 
```

The above query is Boolean, i.e., it has no free variables, so it is considered over an empty tuple, $()$. This query checks whether there exists a path with the same initial and final nodes of length at least 2, i.e., a cycle.

Finally, we can write a query that asks whether there is a unique path between x and y . First, the following query asks for nodes x, y connected with at least two different paths.

```
LET loop(x, y) := AND( $x = y, x \neq \square$ ) IN
SELECT NODES x, y
SUCH THAT  $x \rightarrow_{\pi}^E y$  AND  $x \rightarrow_{\pi'}^E y$ 
HAVING loop[ $\pi, \pi'$ ]  $\geq 1$ 
```

Next, we take the complement of the above query and intersect that complement with the following query stating that there is at least one path from x to y :

```
SELECT NODES x, y
SUCH THAT  $x \rightarrow_{\pi}^E y$ 
```

7. EXPRESSIVE POWER

To understand the expressive power of OPRA, we compare it with other languages. Let us first mention that OPRA expresses all SQL queries over relational databases (subject to technical details arising from types and the fact that SQL can return an ordered list with repetitions). Most of the main ingredients of the proof of this claim are presented in Theorem 6.1, where we have shown the closure properties of OPRA. We skip the proof because it provides little insight into what we are really interested in — graph-oriented properties. Instead, we compare OPRA with a well-known graph query language ECRPQ and its extension with linear constraints (ECRPQ+LC) [BLLW12]. We prove the results depicted in Figure 1: that PR subsumes ECRPQ and PRA subsumes ECRPQ+LC. The strength of ECRPQ comes from the possibility of comparing properties of paths that are expressible by synchronized regular automata. Nevertheless, ECRPQ cannot deal with data values. Therefore, in the final part we show that OPRA subsumes Regular Queries with Memory (RQM)[LMV16] over graphs with integer data values. We conclude with a short discussion on additional expressive power of OPRA over PRA.

An *ECRPQ graph* [BLLW12] is a tuple $\langle V, E \rangle$, where V is a finite set of nodes, and $E \subseteq V \times \Sigma \times V$ is a set of edges labelled by a finite alphabet Σ . A path in an ECRPQ graph G is a sequence of interleaved nodes and edge labels $v_0 e_0 v_1 \dots v_k$ such that for every $i < k$ we have $E(v_i, e_i, v_{i+1})$. The difference between ECRPQ graphs and our graphs is mostly syntactical, yet obscures the close relationship between ECRPQ and PR. To overcome this problem, we define the *standard embedding*, which is a natural transformation of ECRPQ graphs to graphs. The main idea is to replace paths of the form $v_0 e_0 v_1 e_1 \dots v_n$ with paths of the form $(v_0, e_0)(v_1, e_1) \dots (v_{n-1}, e_{n-1})(v_n, \square)$.

The standard embedding of an ECRPQ graph $G = \langle V, E \rangle$ over $\Sigma = \{b_1, \dots, b_k\}$ is a graph G^{se} whose set of nodes is $V^{\text{se}} = V \times \Sigma_{\square}$, where $\Sigma_{\square} = \Sigma \cup \{\square\}$. The graph is equipped with a binary Boolean labelling E^{se} encoding the edge relation: $E^{\text{se}}((v, a), (v', a')) = 1$ if and only if $E(v, a, v')$, and $|\Sigma_{\square}|$ unary Boolean labellings λ_b encoding the edge labels: $\lambda_b(v_1, a) = 1$ if and only if $a = b$. To deal with variables that occur multiple times in path constraints (e.g. x in $x \rightarrow^{\pi} x$), we need an additional Boolean binary labelling \sim that ties nodes representing the same node in G : $\sim((v, a), (v', a')) = 1$ if and only if $v = v'$, for every $(v, a), (v', a') \in V^{\text{se}}$. We say that a node v *corresponds* to the node $v^{\text{se}} = (v, \square)$, and that a path $p = v_1 e_1 v_2 \dots v_n$ *corresponds* to the path $p^{\text{se}} = (v_1, e_1) \dots (v_{n-1}, e_{n-1})(v_n, \square)$.

7.1. Extended conjunctive regular path queries (ECRPQs). An ECRPQ $Q(\vec{x}, \vec{\pi})$ over Σ is a conjunction of *path constraints* of the form $x_i \rightarrow^{\pi_k} x_j$ and *regular-relation constraints* of the form $\mathbf{R}(\pi_{i_1}, \dots, \pi_{i_n})$, where x_i, x_j are node variables, $\pi_k, \pi_{i_1}, \dots, \pi_{i_n}$ are path variables, and \mathbf{R} is a regular expression defining an n -ary regular relation over Σ . An ECRPQ $Q(\vec{x}, \vec{\pi})$ can contain other node and path variables beside those listed among \vec{x} or $\vec{\pi}$; the remaining nodes and path variables are existentially quantified.

The language of ECRPQs is based on the notion of *regular relations*. An n -ary relation R on Σ^* is regular if there is a regular expression \mathbf{R} over the alphabet $(\Sigma \cup \{\square\})^n$ such that for all words $w_1, \dots, w_n \in \Sigma^*$ we have $(w_1, \dots, w_n) \in R$ if and only if $W(w_1, \dots, w_n) \in L(\mathbf{R})$ (the notion $W(p_1, \dots, p_k)$ has been introduced in Section 3 to define the semantics of regular constraints). Note that we use the symbol \square to deal with the differences of paths' lengths, and hence, we need regular expressions over the alphabet $(\Sigma \cup \square)^n$ to define n -ary relations over Σ .

The semantics of ECRPQs is defined with respect to an ECRPQ graph G and an instantiation of all node and path variables ν , i.e., for a nodes \bar{v} of G and paths \bar{p} in G , we have that $Q(\bar{v}, \bar{p})$ holds in G if and only if there is an instantiation ν of nodes and path variables, which is consistent with \bar{v} and \bar{p} on free nodes and respective path variables and such that all constraints of $Q(\bar{x}, \bar{\pi})$ are satisfied. A constraint $x_i \rightarrow^{\pi_k} x_j$ is satisfied by ν if $\nu(\pi_k)$ is a path from $\nu(x_i)$ to $\nu(x_j)$; the semantics is the same as that of $x_i \xrightarrow{E}^{\pi_k} x_j$ in PR. The ECRPQ graph G and ν satisfy $\mathbf{R}(\pi_{i_1}, \dots, \pi_{i_n})$ if and only if the sequences of labels of paths $\nu(\pi_{i_1}), \dots, \nu(\pi_{i_n})$ belong to the relation defined by \mathbf{R} .

ECRPQs are defined in a similar way to PR queries. However, regular-relation constraints in ECRPQs and regular constraints are different. In the case of a single path, regular-relation constraints specify regular languages of labels, while regular constraints specify regular languages of node constraints, which are supposed to match the path. Node constraints can express that a given node has a specific label and hence regular constraints (over a single path) can specify that a path has the sequence of labels from a given regular language. The same reasoning works for multiple paths and it shows that regular constraints from PR polynomially subsume regular-relation constraints from ECRPQs.

A query Q_1 on ECRPQ graphs is *se-equivalent* to a query Q_2 on graphs if for all ECRPQ graphs G , nodes \bar{v} and paths \bar{p} , we have $Q_1(\bar{v}, \bar{p})$ holds in G if and only if $Q_2(\bar{v}^{\text{se}}, \bar{p}^{\text{se}})$ holds in G^{se} . A query language \mathcal{L} on graphs *subsumes* a query language \mathcal{L}' on ECRPQ graphs if for every query in \mathcal{L}' there exists a se-equivalent \mathcal{L} query. Moreover, \mathcal{L} *polynomially subsumes* \mathcal{L}' if every query in \mathcal{L} can be transformed to a query in \mathcal{L}' and the underlying transformation of queries is effective and takes polynomial time.

Theorem 7.1. (1) PR polynomially subsumes ECRPQ. (2) There is a PR query Q with no ECRPQ query Q' se-equivalent to Q .

Proof. (1): ECRPQs consist of two types of constraints. Path constraints $x_i \rightarrow^{\pi_k} x_j$ of ECRPQ have similar semantics to path constraints $x_i \xrightarrow{E}^{\pi_k} x_j$ in PR, but there is a subtle difference arising from different path representation. For example, if we take an ECRPQ graph with an edge (v, a, v) , then $x \rightarrow^{\pi} x$ should be satisfied by $\pi = vav$. However, then π^{se} becomes $(v, a)(v, \square)$ that has different endpoints. Therefore we do as follows. For each $x_i \rightarrow^{\pi_k} x_j$ we use a fresh variable x'_i . The translation now consists of a path constraint $x'_i \xrightarrow{E^{\text{se}}}^{\pi_k} x_j$ and three regular constraints: $\langle \lambda_{\square}(x_i) = 1 \rangle$, $\langle \lambda_{\square}(x_j) = 1 \rangle$ and $\langle \sim(x_i, x'_i) = 1 \rangle$. Note that in the translation of $x \rightarrow^{\pi} x$ the last of the regular constraints has the form $\langle \sim(x, x') = 1 \rangle$.

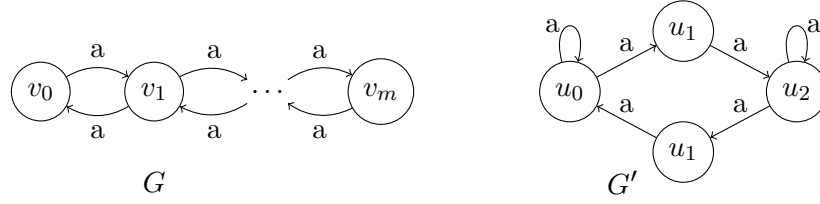
The regular-relation constraints of ECRPQs are basically regular expressions over the alphabet $(\Sigma \cup \{\square\})^n$. In PR, any letter $(a_1, \dots, a_n) \in (\Sigma \cup \{\square\})^n$ can be expressed as the node constraint $\langle \lambda_{a_1}(\pi_1) = 1 \wedge \dots \wedge \lambda_{a_n}(\pi_n) = 1 \rangle$ referring to the current positions on the respective path variables of ECRPQ. This can be extended to a translation of all regular-relation constraints in a straightforward way. Hence PR polynomially subsumes ECRPQ.

(2): Consider the following PR query Q_b :

```
SELECT NODES  $x, y$  SUCH THAT  $x \xrightarrow{E}^{\pi} y$  WHERE  $\langle E(\text{Next}(\pi), \pi) \rangle^* \langle \top \rangle$ 
```

which holds on nodes x, y such that there is a bidirectional path between x and y . We claim that this query is not expressible in ECRPQ.

Suppose that there is an ECRPQ $Q'(x, y)$ that holds if and only if there is a bidirectional path between x and y . Let m be the number of all node variables in Q' . Consider the graphs G, G' depicted in Figure 4. We show that if $Q'(v_0, v_m)$ holds in G , then $Q'(u_0, u_2)$ holds in

FIGURE 4. The graphs G and G' .

G' . Indeed, consider a node variable instantiation ν of $Q'(v_0, v_m)$ in G . There is some node v_j of G , which is not referred by any node variable. We define a node variable instantiation ν' of $Q'(u_0, u_2)$ such that $\nu'(x) = u_0$ if $\nu(x) = v_i$ and $i < j$, and $\nu'(x) = u_2$ otherwise.

Observe that if there is a path in G of length l between two nodes $\nu(x)$ and $\nu(y)$, then there is a path of the same length between $\nu'(x)$ and $\nu'(y)$ in G' . Similarly, if there is a path in G of length l from (resp., to) $\nu(x)$, then there is a path of the same length in G' from (resp., to) $\nu'(x)$. It follows that we can extend the instantiation ν' to path variables such that paths in ν and ν' have the same endpoints among instances of node variables and the same lengths. Therefore, all constraints from $Q'(u_0, u_2)$ of the form $x_i \rightarrow^{\pi_k} x_j$ are satisfied in G' under ν' . Finally, since for every path variable $\nu(\pi)$ and $\nu'(\pi)$ have the same length and a is the only label, all regular constraints of $Q'(u_0, u_2)$ are satisfied as well. Hence, we have that $Q'(u_0, u_2)$ holds in G' , but there is no bidirectional path between u_0 and u_2 . \square

7.2. ECRPQs with linear constraints. ECRPQ+LC [BLLW12] is an extension of ECRPQ with *linear constraints*, expressing that a given vector of paths $\vec{\pi}$ satisfying a given ECRPQ query satisfies linear inequalities, which specify the multiplicity of edge labels in various components of $\vec{\pi}$. Formally, a *linear constraint* is given by $h > 0$, a $h \times (|\Sigma| \cdot n)$ matrix A with integer coefficient and a vector $\vec{b} \in \mathbb{Z}^h$. An instantiation ν of $\vec{\pi}$ (of length n) satisfies this constraint if $A\vec{l} \leq \vec{b}$ holds for the vector $\vec{l} = (l_{1,1}, \dots, l_{|\Sigma|,1}, l_{1,2}, \dots, l_{|\Sigma|,n})$, where $l_{j,i}$ is the number of occurrences of the j -th edge label in $\nu(\vec{p}[i])$. In ECRPQ+LC, we require a tuple of paths to satisfy both the ECRPQ part and the linear constraints.

Linear constraints can be expressed by arithmetical constraints of PRA. This and Theorem 7.1 imply that PRA polynomially subsumes ECRPQ+LC. Still, linear constraints do not help with expressing structural graph properties. In particular, ECRPQ+LC does not express the query “ x and y are connected with a bidirectional path”, which is expressible in PR. Nevertheless, there are ECRPQ+LC queries not expressible in PR. In consequence, ECRPQ+LC and PR are incomparable and we have the following.

Theorem 7.2. (1) PRA polynomially subsumes ECRPQ+LC. (2) There is a PR query Q with no ECRPQ+LC query Q' se-equivalent to Q . (3) There is an ECRPQ+LC query Q with no PR query Q' se-equivalent to Q .

Proof. (1): Language PRA can express all ECRPQs with linear constraints. Consider a query Q of ECRPQ+LC. First, due to Theorem 7.1, PR polynomially subsumes ECRPQs, and hence there is a PR query φ_E corresponding to the ECRPQ part of Q . Second, we can express each $l_{j,i}$ by the arithmetical atom $\Lambda_{j,i} = \lambda_{b_j}[\pi_i]$. Then, for $k = 1, \dots, h$, the arithmetical constraint φ_A^k corresponding to the k -th row of A can be constructed as the product of the row of A and all the atoms $\Lambda_{j,i}$ compared to the k -th element of the vector b .

Thus, φ_A , defined as the conjunction of all φ_A^k , corresponds to the linear constraints $A\vec{l} \leq \vec{b}$ of Q . Finally, we define the PRA query se-equivalent to Q as the conjunction of φ_E and φ_A .

(2): Consider the PR query Q_b presented in the proof of (2) in Theorem 7.1. The argument from (2) in Theorem 7.1 straightforwardly extends to ECRPQ+LC. We assume towards a contradiction that there is an ECRPQ+LC $Q'(x, y)$ that holds if there is a bidirectional path between x and y . Let k be the number of all node variables in Q' . Then, we proceed as in the proof of (2) in Theorem 7.1 to show that if $Q'(v_0, v_k)$ is satisfied under some instantiation ν in G (depicted in Figure 4), then we can define the corresponding instantiation ν' in G' (depicted in Figure 4) such that paths in ν and ν' have the same endpoints among instances of node variables and the same lengths. Therefore, the ECRPQ part of $Q'(u_0, u_2)$ holds in G' under ν' . Observe that the value of the vector \vec{l} in the linear constraints of $Q'(v_0, v_k)$ under ν in G is the same as the value of \vec{l} under ν' in G' . Thus, the linear constraints of Q' are satisfied in G' under ν' . Hence, we have that $Q'(u_0, u_2)$ holds in G' , but there is no bidirectional path between u_0 and u_2 .

(3): Consider the query $Q_{a=b}(\pi)$: “a given path π has the same number of edges labelled a as edges labelled b ”. We claim that $Q_{a=b}(\pi)$ is expressible in ECRPQ+LC, whereas it is not expressible in PR.

To see this, let us fix a graph G consisting of a single state and two self-loops labelled by a and b respectively. Consider the set of labellings of all paths satisfying $Q_{a=b}$. This set can be regarded as the language of words over $\{a, b\}$ with the same number of a 's and b 's. This language is not regular whereas for any PR query Q , the language of labellings of paths from G satisfying Q is regular.

Indeed, suppose that Q has k free path variables and no other path variables. Observe that the language L_Q of labellings of paths satisfying Q is a regular language over $\{a, b, \square\}^k$, i.e., k element tuples over $\{a, b, \square\}$. Now, if Q' is obtained from Q by making some free path variables existentially quantified, then $L_{Q'}$ is obtained from L_Q by projecting out the components $\{a, b, \square\}^k$ that correspond to existentially quantified variables. Such an operation preserves regularity of the language; the language $L_{Q'}$ is still regular. \square

Now we discuss why OPRA is provably stronger than PRA.

Remark 7.3 (OPRA is stronger than PRA). The language OPRA syntactically contains PRA and it strictly subsumes PRA. Consider the property: “an input graph is a directed acyclic graph (dag)”. PRA queries are monotonic and, in consequence, no PRA query expresses the property. On the other hand it is expressible in OPRA. We can write a boolean PR query $Q()$ that holds on input graphs with a cycle. From Theorem 6.1 the complement of $Q()$ can be expressed in OPRA.

7.3. Regular queries with memory (RQMs). Regular Query with Memory (RQM) [LMV16, LTV15] is of the form $x \rightarrow^\pi y \wedge \pi \in L(e)$, where e is a regular expression with memory (REM) [LMV16, LTV15]. We refrain from presenting a formal definition of REMs as we do not use it below. Intuitively, REMs can store in a variable the data value at the current position and test its (dis)equality with other values already stored. RQMs are evaluated over *data graphs* [LMV16]. A data graph G over a finite alphabet Σ and countable infinite set \mathcal{D} is a triple (V, E, ρ) , where V is a finite set of nodes, $E \subseteq V \times \Sigma \times V$ is a set of labelled edges; and $\rho: V \rightarrow \mathcal{D}$ is a function that assigns a data value to each node in V . In this paper we assume that \mathcal{D} is a set of integer numbers. A path in a data graph G is a sequence of interleaved

nodes and edge labels $v_0 e_0 v_1 \dots v_k$ such that for every $i < k$ we have $E(v_i, e_i, v_{i+1})$. REMs are evaluated over *data paths*. Given a path $p = v_0 e_0 v_1 \dots v_k$, a data path corresponding to p is $\rho(v_0) e_0 \rho(v_1) \dots \rho(v_k)$, i.e., a sequence of alternating data values and labels that starts and ends with data values.

Given a data graph G , the result of the RQM $x \rightarrow^\pi y \wedge \pi \in L(e)$ on G consists of pairs of nodes (v, v') such that there is a data path w from v to v' that belongs to $L(e)$.

In order to relate PRA to RQM we apply a natural transformation of data graphs to graphs. We discussed it in Section 2, see the examples 2 and 3. The standard embedding of a data graph $G_{\text{data}} = (V, E, \rho)$ is a graph G^{sed} whose set of nodes is $V^{\text{sed}} = V \cup E$. G^{sed} is equipped with the following labellings:

- a binary Boolean labelling E^{sed} encoding the edge relation: for each $v \in V$ and $e \in E$ we set $E^{\text{sed}}(v, e) = 1$ if and only if there is v' such that $e = (v, a, v') \in E$ and we set $E^{\text{sed}}(e, v) = 1$ if and only if there is v' such that $e = (v', e, v) \in E$;
- a unary labelling λ encoding data labelling of G_{data} defined as $\lambda(v) = \rho(v)$ if $v \in V$ and $\lambda(v) = 0$ otherwise.
- for each $b \in \Sigma$, a unary Boolean labelling λ_b encoding the edge labels: for each $e = (v, a, v') \in E$ we set $\lambda_b(e) = 1$ if $a = b$ and $\lambda_b(e) = 0$ otherwise.

We say that a query Q_1 on data graphs is *se-equivalent* to a query Q_2 on graphs if for all data graphs G and nodes v, v' the query $Q_1(v, v')$ holds in G if and only if $Q_2(v, v')$ holds in G^{sed} .

Similarly as before, a query language \mathcal{L} on graphs *subsumes* a query language \mathcal{L}' on data graphs if for every query in \mathcal{L}' there exists a se-equivalent \mathcal{L} query. Also, \mathcal{L} *polynomially subsumes* \mathcal{L}' if every query in \mathcal{L}' can be transformed to a query in \mathcal{L} and the transformation is effective and can be computed in polynomial time.

Now, we would like to prove that PRA subsumes RQM. To make the proof easier we introduce an intermediate step through *regular data path queries (RDPQ)* [LMV16]. RDPQs are automata-based formalism and, like RQMs, define pairs of nodes of data graphs. In the proof, given a RQM Q we express it by RDPQ Q_1 and then we construct a se-equivalent PRA query.

Now we formally define RDPQs [LMV16]. An RDPQ is of the form $x \rightarrow^\pi y \wedge \pi \in L(\mathcal{A})$, where \mathcal{A} is a *Register Data Path Automaton (RDPA)*. RDPAs, similarly as REMs, are evaluated over data paths. In order to compare the data values, RDPA use Boolean combinations of the conditions of the form $x_i^=$, x_i^\neq , $z^=$ and z^\neq , where each variable x_i refers to the i -th register and z is a data value from \mathcal{D} (a constant). Let \mathcal{C}_k be the set of all such conditions over k registers and their Boolean combinations. Each of the registers store either a data value or a special value \perp that means that the register has not been assigned yet. Semantics of the conditions is defined with respect to a (current) data value d and a valuation of registers $\tau = (d_1, \dots, d_k) \in (\mathcal{D} \cup \perp)^k$ in a natural way: for each $\otimes \in \{=, \neq\}$ and each $i \in \{1, \dots, k\}$ we define $d, \tau \models x_i^\otimes$ iff $d \otimes d_i$, and $d, \tau \models z_i^\otimes$ iff $d \otimes z$. In the sequel, given a register valuation $\tau = (d_1, \dots, d_k)$ we will write $\tau(i)$ for d_i , the i th element of the tuple τ .

A k -register RDPA [LMV16] consists of a finite set of *word states* Q_w , a finite set of *data states* Q_d , an initial state $q_0 \in Q_d$, a set of final states $F \subseteq Q_w$ and two transition relations δ_w, δ_d such that:

- $\delta_w \subseteq Q_w \times \Sigma \times Q_d$ is the word transition relation;
- $\delta_d \subseteq Q_d \times \mathcal{C}_k \times 2^{\{1, \dots, k\}} \times Q_w$ is the data transition relation.

Given a data path $w = d_0 e_1 d_2 \dots d_l$, where each $d_i \in \mathcal{D}$ and each $e_i \in \Sigma$, a *computation* of \mathcal{A} on w is a sequence of tuples $(0, q^0, \tau^0), \dots, (l+1, q^{l+1}, \tau^{l+1})$, where $q^0 = q_0$, $\tau^0 = \perp^k$ and:

- for each even j there is a transition $(q^j, c, I, q^{j+1}) \in \delta_d$ such that $d_j, \tau^j \models c$ and for each $i \in \{1, \dots, k\}$ the value $\tau^{j+1}(i)$ is equal to d_j if $i \in I$ and to $\tau^j(i)$ otherwise.
- for each odd j , there is a transition $(q^j, e_j, q^{j+1}) \in \delta_w$ and $\tau^{j+1} = \tau^j$.

A data path w is *accepted* by \mathcal{A} if \mathcal{A} has a computation on w that ends in a configuration containing a final state. Given a data graph G , the result of the RDPQ $x \rightarrow^\pi y \wedge \pi \in L(\mathcal{A})$ on G consists of pairs of nodes (v, v') such that there is a data path w from v to v' that is accepted by \mathcal{A} .

Each RQM can be expressed with a RDPQ. This is because for each REM with k variables one can construct in PTIME a RDPA with k registers that accepts the same language of data paths [LMV16, Proposition 3.13] and [LTV15, Theorem 4.4].

We show that PRA subsume RQM. The transformation from an RQM to a PRA involves a single exponential blow-up.

Theorem 7.4. (1) PRA subsumes RQM. (2) There is an OPRA query Q with no RQM query Q' equivalent to Q .

Proof. Let Q be an RDPQ of the form $x \rightarrow^\pi y \wedge \pi \in L(\mathcal{A})$, where \mathcal{A} is a RDPA. We will construct an se-equivalent PRA query as follows. First, assume that \mathcal{A} contains exactly one final state. We explain later what to do when this is not true. We construct an *intermediate Path Automaton (iPA)* \mathcal{A}' that is equivalent to \mathcal{A} in the following sense: given a data graph G , nodes v, v' of G the RDPA \mathcal{A} accepts a data path from v to v' if and only if \mathcal{A}' accepts a path from v to v' in G^{sed} . The iPA is an automaton where transitions are labelled by regular constraints from PRA.

iPA process tuples of paths of a graph (like PRA) rather than data paths of a data graph (as in the case of RDPA). A tuple of paths includes additional paths that store values during computation just like registers do in the case of RDPA. Therefore, iPA do not have registers explicitly and whenever we mention *registers* in the context of iPA we mean the mechanism to store values in additional paths. Moreover, the stored values are nodes of a graph as in PRA queries and not data values as in RDPQ.

Formally, a k -register iPA consists of a set of states Q , a single initial state q_I , a single final state q_F , and a transition relation $\delta \subseteq Q \times \text{Regular_constraints} \times Q$. The regular constraints in a k -register iPA use a path variable π_0 corresponding to an input path and k existentially quantified, additional path variables π_1, \dots, π_k to store the values of the k registers.

In what follows, for a path p , by $p[i, j]$ we denote the fragment of p that starts at position i and ends at the position j . e.g., for $p = v_1 e_2 v_3 e_4 v_5$, the expression $p[3, 5]$ denotes $v_3 e_4 v_5$.

We say that for a given graph G , an iPA \mathcal{A}' accepts a path p_0 if there are paths p_1, \dots, p_k , each of the same length as p_0 , there are: a number $n \in \mathbb{N}$, positions $0 = i_0 < i_1 < i_2 < \dots < i_{n-1} \leq i_n = |p_0|$ and a sequence of states q_0, \dots, q_n such that

- q_0 is the initial state, q_n is the final state, and
- for each $j \in \{0, \dots, n-1\}$ there is a transition $(q_j, R(\pi_0, \dots, \pi_k), q_{j+1}) \in \delta$ such that
 - for $j < n-1$, the paths $p_0[i_j, i_{j+1}], \dots, p_k[i_j, i_{j+1}]$ satisfy the regular constraint $R \cdot \langle \top \rangle$, i.e., $R \cdot \langle \top \rangle (p_0[i_j, i_{j+1}], \dots, p_k[i_j, i_{j+1}])$ holds.
 - for $j = n-1$, the paths $p_0[i_j, i_{j+1}], \dots, p_k[i_j, i_{j+1}]$ satisfy the regular constraint R

Note that for $j \in \{1, \dots, n-1\}$ the positions i_j on the paths p_0, \dots, p_k are processed twice, first by the always satisfied node constraint $\langle \top \rangle$ and only then by the actual transition. We apply this trick to be able to refer to the next position on the paths even if the current position is the last one in the current section. In other words, for $0 \leq j \leq n-2$ we include the position i_{j+1} in the fragments $p_i[i_j, i_{j+1}]$ only to be able to refer to the next position while being at the position $i_{j+1} - 1$.

Now, given a k -register RDPA $\mathcal{A} = (Q_w, Q_d, q_0, F, \delta_w, \delta_d)$ we define an equivalent k -register iPA $\mathcal{A}' = (Q, q_I, q_F, \delta)$. Let $Q = Q_w \cup Q_d$. We define q_I to be q_0 and q_F to be the only state in F .

We now define a transition relation δ .

For each transition $(q, a, q') \in \delta_w$ we define a new transition $(q, R, q') \in \delta$, where the regular constraint $R = \langle \lambda_a(\pi_0) = 1 \wedge E^{\text{sed}}(\pi_0, \mathbf{Next}(\pi_0)) = 1 \wedge \bigwedge_{i \in \{1, \dots, k\}} \lambda(\mathbf{Next}(\pi_i)) = \lambda(\pi_i) \rangle$ ensures that

- the current node on π_0 corresponds to an edge in the graph that is labelled with a ;
- the edge goes to the next node on π_0 ;
- for each path π_1, \dots, π_k the next value is the same as the current value (i.e., the registers values do not change).

Note that we can safely refer to the next position on paths π_i because q' is not the final state as $q_F \in Q_w$ and $\delta_w \subseteq Q_w \times \Sigma \times Q_d$.

For each transition $(q, c, I, q') \in \delta_d$, with a condition c and an update set I , we put (q, R, q') in δ , where the regular constraint R ensures that

- (1) there is an edge between the current and the next node on π_0 in G ;
- (2) the condition c is satisfied assuming for each $i \in \{1, \dots, k\}$ the value of the register i is the current value of the path π_i ;
- (3) unless q' is the final state, for each $i \in I$ the next value of the path π_i is set to the current value of the input path π_0 ;
- (4) unless q' is the final state, for each $i \in \{1, \dots, k\} \setminus I$ the next value of the path π_i is the same as the current value of π_i .

We express the condition $c \in \mathcal{C}_k$ as the Boolean combination of node constraints, denoted by $N(c)$, by replacing each x_i^\otimes by $\langle \lambda(\pi_i) \otimes \lambda(\pi_0) \rangle$ and z^\otimes by $\langle z \otimes \lambda(\pi_0) \rangle$, for $\otimes \in \{=, \neq\}$. Recall that λ is the labelling encoding the data values of the nodes.

We cannot include $N(c)$ directly in a regular constraint because regular constraints are conjunctions of node constraints. Hence, we transform $N(c)$ to DNF. Then we remove all negations swapping $=$ and \neq in x_i^\otimes and z^\otimes as required. Denote the resulting expression by $N = \bigvee_l N_l$.

If q' is not the final state then for each of the conjunctions N_l we define a regular constraint R_l as

$$\langle E^{\text{sed}}(\pi_0, \mathbf{Next}(\pi_0)) = 1 \wedge N_l \wedge \bigwedge_{i \in I} \lambda(\mathbf{Next}(\pi_i)) = \lambda(\pi_0) \wedge \bigwedge_{i \in \{1, \dots, k\} \setminus I} \lambda(\mathbf{Next}(\pi_i)) = \lambda(\pi_i) \rangle$$

Otherwise R_l is defined as $\langle N_l \rangle$. Finally, we define R as $R_1 + \dots + R_s$, where $N = N_1 \vee \dots \vee N_s$. This finishes the construction of the automaton \mathcal{A}' . The automaton \mathcal{A}' has the same number of states as \mathcal{A} , but its size may be exponential in $|\mathcal{A}|$ due to transformation from an arbitrary Boolean combination of node constraints to a DNF. This blow-up can be avoided if we allow OPRA queries, in which we can define a new labelling corresponding to any Boolean function, and then use this labelling to express a Boolean combination of node constraints.

To end the proof of (1) we use the standard state removal method of converting an NFA to a regular expression. This removes all the states of \mathcal{A}' except q_I and q_F . Then we use the same techniques as when removing states to define from the remaining transitions (i.e., the loops on q_I , on q_F , and the transition from q_I to q_F and back) the single transition $t = (q_I, R, q_F)$ with the regular constraint R such that R describes equivalently all possible paths from q_I and q_F . We use R to define PRA query `SELECT NODES x, y WHERE R` that is se-equivalent to the original RDPQ query Q . The transformation from a \mathcal{A}' to R is exponential in the number of states of \mathcal{A}' , which is the same as for \mathcal{A} , and hence the whole construction results in a single exponential blow-up.

If the RDPA \mathcal{A} has more than one final state in F we repeat the above construction $|F|$ times. Namely, for each $q \in F$ we construct \mathcal{A}_q which is identical as \mathcal{A} but with q as its only final state. This way for each $q \in F$ we obtain a regular constraint R_q and then we set R as the alternation (+) of the regular constraints R_q for all $q \in F$.

In order to prove (2) it is enough to note that OPRA can express the query “an input graph is a dag” as we discuss in Remark 7.3. Clearly, RQMs are monotonic (i.e., if an RQM query holds in a graph G then it holds in any G' containing G) and cannot express this query. \square

8. THE QUERY EVALUATION PROBLEM

The query evaluation problem asks, given an OPRA query $Q(\vec{x}, \vec{\pi})$, a graph G , nodes \vec{v} and paths \vec{p} of G , whether $Q(\vec{v}, \vec{p})$ holds in G . We are interested in the *combined complexity* of the query evaluation problem, where the size of the input is the size of G , paths \vec{p} and an input OPRA query, and in the *data complexity*, where a query is fixed and only G and \vec{p} are the input.

Unary encoding of graph labels. To obtain the desired complexity results, we assume that the absolute values of the graph labels are polynomially bounded in the size of a graph, or equivalently that graph labels are encoded in unary. This allows us to compute arithmetical relations on these labels in logarithmic space. Without such a restriction, the data complexity of the query evaluation problem we study is NP-hard by a straightforward reduction from the knapsack problem.

For the combined complexity, we assume that auxiliary labellings have a *bounded depth* defined as follows. For auxiliary labellings $O := \lambda_1 := t_1, \dots, \lambda_n := t_n$, we say that λ_i depends on λ_j if t_i refers to λ_j . The relation *depends on* defines a directed acyclic graph on $\lambda_1, \dots, \lambda_n$ and we define the *depth* of O as the maximal length of a path in this acyclic graph. An OPRA query $Q := \text{LET } O \text{ IN } Q'$ is an *OPRA query of depth (at most) s*, denoted by $\text{OPRA}[s]$, if $s = 0$, and Q is a PRA query, i.e., O is empty, or $s > 0$, O has depth at most s and all subqueries of Q' are $\text{OPRA}[s-1]$ queries.

Theorem 8.1. *The following holds:*

- (1) *The data complexity of the query evaluation problem for OPRA queries is NL-complete.*
- (2) *The combined complexity of the query evaluation problem for OPRA queries with a bounded depth of auxiliary labellings is PSPACE-complete.*

The lower bounds in Theorem 8.1 holds even for PR. Indeed, the NL-hardness result in (1) of Theorem 8.1 follows from NL-hardness of the reachability problem, which can be expressed

even in PR. The PSPACE-hardness result in (2) Theorem 8.1 follows from PSPACE-hardness of the query evaluation problem for ECRPQs [BLLW12] and polynomial-time translation from ECRPQs to OPRA (Theorem 7.1). The upper bound in Theorem 8.1 follows directly from the following Lemma 8.2.

Lemma 8.2. *For every fixed $s \geq 0$, we have:*

- (1) *Given a graph G , nodes \bar{v} and paths \bar{p} of G , and an OPRA[s] query $Q(\bar{x}, \bar{\pi})$, we can decide whether $Q(\bar{v}, \bar{p})$ holds in G in non-deterministic polynomial space in $|Q|$ and non-deterministic logarithmic space in $|G|$.*
- (2) *Given a graph G , nodes \bar{v} of G , and an OPRA[s] query $Q(\bar{x}, \pi)$, we can compute $\min_{\lambda, \pi} Q(\bar{v}, \pi)$ (resp., $\max_{\lambda, \pi} Q(\bar{v}, \pi)$) in non-deterministic polynomial space in $|Q|$ and non-deterministic logarithmic space in $|G|$. The computed value is either polynomial in $|G|$ and exponential in $|Q|$, or $-\infty$ (resp., ∞).*

Remark 8.3 (Functions computed non-deterministically). In Lemma 8.2 values $\min_{\lambda, \pi} Q(\bar{v}, \pi)$, $\max_{\lambda, \pi} Q(\bar{v}, \pi)$ are computed on a non-deterministic machine, which is a non-standard notion. However, some subcomputations of our non-deterministic decision procedure return values. Therefore, we adopt non-deterministic framework to functional problems as follows. We say that a non-deterministic machine M computes a function f , if for every input w , (1) M can have multiple computations on w each *accepting* or *rejection*, (2) for all accepting computations of M on w , it returns the correct value $f(w)$, (3) M has at least one accepting computation on w .

Note that when considering data complexity the query is fixed and hence its depth is bounded.

We first prove the upper bounds for PRA (i.e., for $s = 0$), and then extend the results to OPRA. The general idea of the proof is in a similar vein as the proof of the upper bound of ECRPQ. However, since our language is much more complex, we use more sophisticated, well-tailored tools.

8.1. Language PRA.

Assume a PRA query

```
Q = SELECT NODES  $\bar{x}$ , PATHS  $\bar{\pi}$  SUCH THAT  $P$  WHERE  $R$  HAVING  $A$ 
```

We prove the results in two steps. First, we construct a Turing machine of a special kind (later on called QAM) that represents graphs, called *answer graphs*. Given a query Q and a graph G , the answer graph is a graph with distinguished initial and final nodes such that every path from an initial node to a final node in this graph is an encoding of a vector of paths that satisfy constraints P and R of Q in graph G (for some instantiation of variables \bar{x}). The answer graph is augmented with the computed values of the expressions that appear in the arithmetical constraints A . Thus, the query evaluation problem reduces to the existence of a path in the answer graph satisfying A . The instantiation of \bar{x} can be inferred from the path.

Second, we prove that checking whether there is a path from an initial node to a final node in the answer graph that encodes a path in G satisfying A can be done within desired complexity bounds. However, the answer graph for Q and G has a polynomial size in G and hence it cannot be explicitly constructed in logarithmic space. We represent these graphs *on-the-fly* using Query Applying Machines (QAM). Such a representation allows us to construct the answer graph and check the existence of a path satisfying A in non-deterministic logarithmic space in G .

The first step is an adaptation of the technique commonly used in the field, e.g., in [BLLW12, GMOW16]. We encode vectors (p_1, \dots, p_n) of paths of nodes from some V as a single path $p_1 \otimes \dots \otimes p_n$ over the product alphabet V^n (shorter paths are padded with \square).

Answer graphs. Consider a graph G with nodes V , its paths \bar{p} and an OPRA query

$Q = \text{SELECT NODES } \bar{x}, \text{ PATHS } \bar{\pi} \text{ SUCH THAT } P \text{ WHERE } R_1, \dots, R_p \text{ HAVING } \bigwedge_{i=1}^m A_i \leq c_i$

with $|\bar{p}| = |\bar{\pi}|$ and existentially quantified path variables $\bar{\pi}'$. Let $k = |\bar{\pi}| + |\bar{\pi}'|$ be the number of path variables. For every regular constraint R_i in Q , we build an NFA \mathcal{A}_i with the set of states C_i recognizing the language of R_i . We extend each \mathcal{A}_i with self-loops on all final states labelled by \perp — an auxiliary node constraint satisfied if all the input nodes are \square . We define $C_Q^{\bar{p}}$ as $C_1 \times \dots \times C_p \times \{1, \dots, N, \infty\}$, where N is the maximal length of paths in \bar{p} .

The *answer graph* for Q on G , \bar{p} is a triple (G', S, T) , where $S, T \subseteq C_Q^{\bar{p}} \times V^k$;

- G' is a graph with nodes $C_Q^{\bar{p}} \times V^k$, labelled by $\lambda_1, \dots, \lambda_m, \lambda_E$;
- for each $i \leq m$ and a node $(s, v_1, \dots, v_k) \in C_Q^{\bar{p}} \times V^k$, the labelling $\lambda_i(s, v_1, \dots, v_k)$ is defined as the value of the arithmetical constraint A_i over a vector of single-node paths v_1, \dots, v_k ; and
- paths q_1, \dots, q_k s.t. $(q_1, \dots, q_{|\pi|}) = \bar{p}$ satisfy $P \wedge R$ if and only if there is c such that the path $q = c \otimes q_1 \otimes \dots \otimes q_k$ from (a node in) S to (a node in) T is such that $\lambda_E(v, v') = 1$ for all consecutive v, v' of q .

Intuitively, the labelling λ_E can be defined in such a way that along paths of G' the V^k -components of the nodes correctly encode paths of Q satisfying the path constraints P and the $C_Q^{\bar{p}}$ components store valid runs of automata corresponding to the regular constraints R .

Query Applying Machine (QAMs). Answer graphs are typically too big to be stored in an explicit way, but can be represented (on-the-fly) in logarithmic space. To do so, we introduce a Query Applying Machine (QAM). QAM is a non-deterministic Turing Machine, which works in logarithmic space and only accepts inputs encoding tuples of the form (G, t, w) , where G is a graph, t is a *query type* consisting of symbols $\underline{V}, \underline{\lambda}, \underline{S}, \underline{T}$, and w is the object being queried.

For a graph G and $k \geq 0$, a QAM M gives a graph $G_k^M = (V, \lambda_1, \dots, \lambda_k, \lambda_E)$ and sets of nodes S_G^M, T_G^M such that:

- V consists of all the *nodes* v such that M accepts on (G, \underline{V}, v) ,
- λ_i is such that $\lambda_i(\vec{v}) = n$ if and only if M accepts on $(G, \underline{\lambda}, (i, \vec{v}, n))$
- λ_E is such that $\lambda_E(\vec{v}) = n$ if and only if M accepts on $(G, \underline{\lambda}, (E, \vec{v}, n))$
- S_G^M (resp., T_G^M) consists of $v \in V$ such that M accepts on (G, \underline{S}, v) (resp., (G, \underline{T}, v)).

For soundness, we require that for each G, i and \vec{v} there is exactly one n such that M accepts on $(G, \underline{\lambda}, (i, \vec{v}, n))$ and, analogously, for each G, \vec{v} there is exactly one n such that M accepts on $(G, \underline{\lambda}, (E, \vec{v}, n))$.

Given a PRA query and its parameters, we can construct a QAM that gives on-the-fly answer graphs for this query:

Lemma 8.4. *For a given query Q and paths \bar{p} , we can construct in polynomial time a QAM M^Q such that for every graph G , the machine M^Q gives an answer graph for Q on G, \bar{p} .*

Proof. Consider a vector \bar{p} of paths of G and an OPRA query

$Q = \text{SELECT NODES } \vec{x}, \text{ PATHS } \vec{\pi} \text{ SUCH THAT } P_1, \dots, P_l \text{ WHERE } R_1, \dots, R_p \text{ HAVING } \bigwedge_{i=1}^m A_i \leq c_i$
 with $|\vec{p}| = |\vec{\pi}|$ and existentially quantified path variables $\vec{\pi}'$. Let $k_1 = |\vec{\pi}|$, $k_2 = |\vec{\pi}'|$ and $k = k_1 + k_2$. We discuss how each of the components of an answer graph for Q on an input graph G with nodes V and \vec{p} can be constructed.

The nodes. The set of nodes is $C_Q^{\vec{p}} \times V^k$, which can be recognized in logarithmic space (note that $C_Q^{\vec{p}}$ and k do not depend on the input of the QAM).

The labelling λ_E . Let \vec{u}_1, \vec{u}_2 be nodes of G' . We put $\lambda_E(\vec{u}_1, \vec{u}_2) = 1$ if nodes \vec{u}_1, \vec{u}_2 are *path consistent* and *state consistent*, defined as follows, $\lambda_E(\vec{u}_1, \vec{u}_2) = 0$ otherwise.

Path consistency. Let $\vec{u}_1 = (c, v_1, \dots, v_k)$ with $c = (s_1, \dots, s_p, j)$, and let $\vec{u}_2 = (c', v'_1, \dots, v'_k)$ with $c' = (s'_1, \dots, s'_p, j')$. Path consistency ensures that the first k_1 components of V^k encode the input paths \vec{p} , paths that end satisfy path constraints, and paths that have terminated do not restart, i.e.,

- $j' = j + 1$ if $j < N$,
- $j' = \infty$ if $j \geq N$,
- for each $i \in \{1, \dots, k_1\}$ we have $v_i = p_i[j]$ or $v_i = \square$ and $j > |p_i|$ (in particular if $j = \infty$),
- for each $i \in \{k_1 + 1, \dots, k\}$, if $v_i \neq \square$ and $v'_i = \square$ (v_i is the last node of π_i), then for every path constraint $x_s \rightarrow^{\pi_i} x_t$, we require $v_i = x_t$,
- for each $i \in \{k_1 + 1, \dots, k\}$ if $v_i = \square$, then $v'_i = \square$.

State consistency. The state consistency ensures that the component $C_Q^{\vec{p}}$ stores valid runs of automata for the regular constraints, i.e., that for each $i \in \{1, \dots, p\}$, the automaton \mathcal{A}_i has a transition (s_i, a_i, s'_i) , where a_i is a node constraint, and a_i is satisfied over the nodes of $v_1, v'_1, \dots, v_k, v'_k$ (we assume that a_i selects from the list of all paths only the relevant paths listed in the regular constraint).

It is easy to check that given a graph G and its two nodes \vec{v}_1, \vec{v}_2 , we can decide in logarithmic space in G whether $\lambda_E(\vec{v}_1, \vec{v}_2)$ is 0 or 1.

The labellings $\lambda_1, \dots, \lambda_m$. For $i \in \{1, \dots, m\}$ we define $\lambda_i(c, v_1, \dots, v_k)$ as the value of A_i computed on the subset of v_1, \dots, v_k selected by A_i . Since A_i is a linear combination and each labelling of G is given in unary, all labellings of G' can be computed in logarithmic space in G .

The initial and final sets S and T . The set S consists of the nodes $(c, v_1, \dots, v_k, 1)$ such that (1) $c = (s_1, \dots, s_p)$ and s_i is an initial state of \mathcal{A}_i for $i \in \{1, \dots, p\}$, (2) for every path constraint $x_s \rightarrow^{\pi} x_t$, we require $v_i = x_s$, and (3) for every $i \in \{1, \dots, k_1\}$ we have $v_i = p_i[1]$. The set T consists of the nodes (c, v_1, \dots, v_k) such that (1) $c = (s_1, \dots, s_p, \infty)$ and s_i is a final state of \mathcal{A}_i for $i \in \{1, \dots, p\}$, (2) for every $i \in \{1, \dots, k\}$ we have $v_i = \square$, i.e., all paths have terminated. \square

We have shown that a QAM representing answer graphs for a given query can be efficiently constructed. The query holds if and only if its answer graph has a path satisfying arithmetical constraints. Now, we show that the existence of a path satisfying given arithmetical constraints can be efficiently decided on graphs represented by QAMs. We additionally prove that we can effectively compute the minimum and the maximum over labellings of paths satisfying given arithmetical constraints. The second result will be used in Section 8.2.

Lemma 8.5. *For a graph G and a QAM M^Q , let Π be the set of paths from $S_G^{M^Q}$ to $T_G^{M^Q}$ satisfying $\bigwedge_{i=1}^m \lambda_i[\pi] \leq c_i$ in $G_m^{M^Q}$. (1) Checking emptiness of Π can be done non-deterministically in polynomial space in Q and logarithmic space in G . (2) Computing*

the minimum (resp., maximum) of the value $\lambda_j[\pi]$ over all paths in Π can be done non-deterministically in polynomial space in Q and logarithmic space in G . The computed extremal value is $-\infty$, ∞ , or polynomial in G and exponential in Q .

Proof. A vector addition system with states (VASS) is a \mathbb{Z}^d -labelled graph G , i.e., $G = (V, E)$, where V is a finite set and E is a finite subset of $V \times \mathbb{Z}^d \times V$. Depending on the representation of labels \mathbb{Z}^d , we distinguish unary and binary VASS. The \mathbb{Z} -reachability problem for VASS, asks, given a VASS G and its two configurations $(s, \vec{u}_1), (t, \vec{u}_2) \in V \times \mathbb{Z}^d$, whether there exists a path $\pi = (s, \vec{x}_0, q_1), (q_1, \vec{x}_1, q_2) \dots (q_k, \vec{x}_k, t)$ in G from s to t such that $\vec{u}_1 + \sum_{i=0}^k \vec{x}_i = \vec{u}_2$, i.e., \vec{u}_1 plus the sum of labels along π equals \vec{u}_2 . In contrast to reachability in VASS, in \mathbb{Z} -reachability we allow configurations with negative components.

We discuss how to reduce the problem of non-emptiness of Π to the \mathbb{Z} -reachability problem for VASS. We transform $G_m^{M^Q}$ into a VASS $G' = (V, E)$ over the set of nodes of $G_m^{M^Q}$ with two additional nodes s, t . We put an edge between two nodes connected node $q_1, q_2 \in V$ labelled by the label of the source node \vec{v} , i.e., for all $q_1, q_2 \in V$ we have (q_1, \vec{v}, q_2) if $\lambda_E(q_1, q_2) = 1$ and $\vec{v} = (\lambda_1(q), \dots, \lambda_m(q))$. Moreover, we define s as the source and t and the sink, i.e., (1) for every $q \in s_G^{M^Q}$ we put an edge (s, \vec{v}, q) , where $\vec{v} = (c_1, \dots, c_m)$ (constants from the definition of Π), and (2) for every $q \in T_G^{M^Q}$ we put an edge (q, \vec{v}, t) , where $\vec{v} = (\lambda_1(q), \dots, \lambda_m(q))$. Finally, we allow the labels to be increased in t , i.e., for every $i \in \{1, \dots, m\}$, we put $(t, 1_i, t)$, where $1_i \in \mathbb{Z}^d$ has 1 at the component i , and 0 at all other components.

Observe that paths from Π correspond to paths in VASS G' from $(s, \vec{0})$ to $(t, \vec{0})$. The \mathbb{Z} -reachability problem for unary VASS of the fixed dimension (which is m in the reduction) is in NL [BFG⁺15, Theorem 19]. In the proof, it has been shown that if weights are given in unary and there exists a path between given two configurations, then there also exists a path, which is bounded by $p(|G'|)^{p'(d)}$, where p, p' are polynomial functions, $|G'|$ is the size of the VASS and d is the dimension. This means that such a path is: (a) polynomially bounded in the VASS, provided that the dimension is fixed, and (b) bounded exponentially otherwise.

To show (1), observe that all the labels are given in unary, and that m is fixed since Q is fixed. Therefore, if Π is non-empty, then it contains a path of a polynomial size in $|G|$. The existence of such a path can be verified in non-deterministic logarithmic space using the QAM M^Q as an oracle to query for the nodes, the edges and the labelling of $G_m^{M^Q}$. The QAM M^Q requires logarithmic space in $|G|$. Therefore checking whether Π is empty can be done non-deterministically in logarithmic space in $|G|$.

If Π is non-empty, then it contains a path of the size $p(|G'|)^{p'(d)}$, where $|G'| = |G_m^{M^Q}|$ is exponential in $|Q|$, but the dimension d is the number of labellings in the answer graph $G_m^{M^Q}$, and it is bounded by the number of arithmetical constraints in Q . Therefore, d is bounded by $|Q|$ and hence $p(|G'|)^{p'(d)}$ is exponential in $|Q|$. Finally, if Π is non-empty, then it contains a path of the exponential size in $|Q|$. The existence of such a path can be verified in non-deterministic polynomial space using the QAM M^Q as an oracle to query for the nodes, the edges and the labelling of $G_m^{M^Q}$. Therefore checking the emptiness of Π can be done non-deterministically in a polynomial space in Q .

To show (2), we need to analyse the proof of [BFG⁺15, Theorem 19]. The set of paths in a VASS from a configuration (s, \vec{v}_1) to (t, \vec{v}_2) can be infinite, but it can be finitely presented with *path schemes*. A *path scheme* is a regular expression ρ over transitions of a VASS such

that every sequence of transitions matching ρ is a path in the VASS. It has been shown that there exists a finite set S of path schemes of the form $\alpha_0\beta_1^*\dots\beta_k^*\alpha_k$, where $\alpha_0, \dots, \alpha_k$ are paths and β_0, \dots, β_k are cycles, such that (i) each path scheme in S has a polynomially bounded length in the size of VASS, (ii) for all configurations $(s, \bar{v}_1), (t, \bar{v}_2)$, if there is a path from (s, \bar{v}_1) to (t, \bar{v}_2) , then there is a path that matches some path scheme from S . Next, it has been shown that for every path scheme $\alpha_0\beta_1^*\dots\beta_k^*\alpha_k = \rho \in S$, there is a system of linear Diophantine equations \mathcal{E}_ρ such that \mathcal{E}_ρ has a solution x_1, \dots, x_k if and only if $\alpha_0\beta_1^{x_1}\dots\beta_k^{x_k}\alpha_k$ is a path from (s, \bar{v}_1) to (t, \bar{v}_2) . For each such a system of linear Diophantine equations \mathcal{E}_ρ , the set of all its solutions has a special form. Let $\text{cone}(P_\rho)$ be the set of linear combinations of vectors from P with non-negative integer coefficients. Then, the set of solution of \mathcal{E}_ρ is of the form $B_\rho + \text{cone}(P_\rho)$, where B_ρ, P_ρ are sets of vectors whose coefficients are (a) exponentially bounded in the dimension, (b) polynomially bounded in the size of VASS (with the dimension fixed). It follows that if Π is non-empty, one of the following holds:

- (1) For some path scheme $\rho = \alpha_0\beta_1^*\dots\beta_k^*\alpha_k$, sets B_ρ, P_ρ are non-empty (\mathcal{E}_ρ has infinitely many solutions), and for some $\bar{u} = (u_1, \dots, u_k)$, the sum of the value $\lambda_j[\pi]$ over paths $\beta_1^{u_1}, \dots, \beta_k^{u_k}$ is negative, and hence the minimum is $-\infty$,
- (2) Otherwise, the minimum exists and it is realized by some path π matching some path scheme $\rho = \alpha_0\beta_1^*\dots\beta_k^*\alpha_k$ such that $\pi = \alpha_0\beta_1^{x_1}\dots\beta_k^{x_k}\alpha_k$, where $(x_1, \dots, x_k) \in B_\rho$. Observe that it does not pay off to incorporate vectors from P_ρ as they cannot decrease the value of $\lambda_j[\pi]$. Finally, observe that the size of such a path π is polynomial in the size of VASS if the dimension of the VASS is fixed and it is exponential in the dimension.

From (1) and (2), we derive bounds $b_1(G, Q) < b_2(G, Q)$, which are polynomial in G and exponential in Q such that if the minimum of $\lambda_j[\pi]$ over path in Π exists, then it is realized by some path of length bounded by $b_1(G, Q)$. However, if there is a path $\pi \in \Pi$ of length between $b_1(G, Q)$ and $b_2(G, Q)$, with $\lambda_j[\pi]$ lower than the value of any path shorter than $b_1(G, Q)$, then the minimum is infinite. Since NL and PSPACE are closed under the complement and bounded alternation (only two conditions need to be checked), both conditions can be checked in non-deterministically in polynomial space in Q and logarithmic space in G . The case of the maximum is symmetric. \square

8.2. Language OPRA. Assume $O = \lambda_1:=t_1, \dots, \lambda_n:=t_n$ has depth s . We show by induction on s that the values of the labellings of a graph $G[O]$ can be non-deterministically computed in space polynomial in O .

Lemma 8.6. *Let s be fixed. For a graph G and O of depth s the value of each labelling of $G[O]$ can be non-deterministically computed in polynomial space in O and logarithmic space in G .*

The proof studies all the possible constructors of terms. In the case of subqueries, we apply the inductive assumption, i.e., Theorem 8.1 for a query with fewer auxiliary labellings. The case of the minimum follows from (2) of Lemma 8.5, i.e., $\min_{p \in \Pi} \lambda_i[p]$ is either $-\infty, +\infty$ or exponential in Q and polynomial in G . Therefore, it can be computed using Lemma 8.5 and the bisection method. The case for the maximum is symmetric. Finally, the application of a function symbol to terms or ranges can be implemented in the expected complexity.

Proof. The proof is by induction on s . The basis of induction, $s = 0$, is trivial. Assume that for s the lemma statement and Lemma 8.2 hold. We show that it holds for $s + 1$. Consider a graph G , ontologies O , and $O' = O$, $\lambda_1 := t_1, \dots, \lambda_s := t_k$ such that $\lambda_1, \dots, \lambda_k$ are independent, i.e., t_1, \dots, t_k do not refer to labellings $\lambda_1, \dots, \lambda_k$.

We show that the value of each t_i can be non-deterministically computed in polynomial space in O and logarithmic space in G . We start the computation from the bottom, the leaves, and show that the values of leaves can be non-deterministically computed in polynomial space in O and logarithmic space in G . Indeed, leaves are of one of the following forms: $c \mid \lambda(\vec{y}) \mid [Q(\vec{y})] \mid \min_{\lambda, \pi} Q(\vec{y}, \pi) \mid \max_{\lambda, \pi} Q(\vec{y}, \pi) \mid y = y$. The cases of c and $y = y$ are trivial. For leaves of the form $\lambda(\vec{y})$, we know that λ is either a graph labelling or it is from O and hence we use the inductive assumption of this lemma. Finally, for leaves of the form $[Q(\vec{y})]$, $\min_{\lambda, \pi} Q(\vec{y}, \pi)$ and $\max_{\lambda, \pi} Q(\vec{y}, \pi)$, it follows from Lemma 8.2 applied inductively for s .

The internal nodes of t_i are of the form $f(t_1(\vec{y}), \dots, t_k(\vec{y}))$ or $f'(\{t(x):t(x, \vec{y})\})$. Having the values of subterms $t_1(\vec{y}), \dots, t_k(\vec{y})$, the value $f(t_1(\vec{y}), \dots, t_k(\vec{y}))$ can be computed in logarithmic in length of \vec{x} and values in \vec{x} , i.e., we require space $\max(\log(|t_1(\vec{y})|), \dots, \log(|t_k(\vec{y})|)) + \log(k) + C$, where C is a constant. Similarly, to compute $f'(\{t(x):t(x, \vec{y})\})$, we require space $\max_x(\log(|t_1(x, \vec{y})|)) + \log(|G|) + C$, where C is a constant. It follows that to compute the value of t_i , we require space $|t_i|(\log |G| + C) \cdot M$, where C is the maximal constant taken over all $f \in \mathcal{F}$ (which is fixed), and M is the maximum over space requirements of the leaves, which is logarithmic in G and polynomial in t_i .

Since all labellings $\lambda_1, \dots, \lambda_k$ are independent, the result follows. \square

Finally, we sketch the proof of Lemma 8.2.

Proof. (of Lemma 8.2) The proof is by induction on s . Lemma 8.4 and Lemma 8.5 imply the basis of induction. Next, assume that this lemma holds for s . Consider an query `LET O IN Q'` , with $\text{DEPTH}(O) = s + 1$ and a graph G . We first build a QAM $M^{Q'}$ as in Lemma 8.4. Since $M^{Q'}$ may refer to labellings from O , not defined in G , we change it so that whenever it wants to access a value of one of the labellings defined in O , it instead runs a procedure guaranteed by Lemma 8.6. Lemma 8.6 holds because this lemma holds for s . Finally, we use Lemma 8.5 to determine the result. \square

9. CONCLUSIONS

We defined a new graphical query language for databases, OPRA. Among its advantages are good expressive power, modularity and reasonable complexity.

We demonstrated the expressive power of OPRA in two ways. We presented examples of natural properties and OPRA queries expressing them in an organised, modular way. We also showed that OPRA strictly subsumes ECRPQ+LC. Despite the additional expressive power, the complexity of the query evaluation problem for OPRA matches the complexity for ECRPQ.

REFERENCES

- [AAB⁺17] Renzo Angles, Marcelo Arenas, Pablo Barceló, Aidan Hogan, Juan Reutter, and Domagoj Vrgoč. Foundations of modern query languages for graph databases. *ACM Comput. Surv.*, 50(5):68:1–68:40, 2017.

- [AAB⁺18] Renzo Angles, Marcelo Arenas, Pablo Barceló, Peter A. Boncz, George H. L. Fletcher, Claudio Gutierrez, Tobias Lindaaker, Marcus Paradies, Stefan Plantikow, Juan F. Sequeda, Oskar van Rest, and Hannes Voigt. G-CORE: A core for future graph query languages. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 1421–1432. ACM, 2018.
- [ACKZ07] Alessandro Artale, Diego Calvanese, Roman Kontchakov, and Michael Zakharyashev. DL-lite in the light of first-order logic. In *Proceedings of the national conference on artificial intelligence*, volume 22, page 361. AAAI Press; MIT Press, 2007.
- [AGP14] Marcelo Arenas, Georg Gottlob, and Andreas Pieris. Expressive languages for querying the semantic web. In *PODS 2014*, pages 14–26. ACM, 2014.
- [Bar13] Pablo Barceló. Querying graph databases. In *Proceedings of the 32nd Symposium on Principles of Database Systems (PODS13)*, pages 175–188, 2013.
- [BDM⁺11] Mikołaj Bojańczyk, Claire David, Anca Muscholl, Thomas Schwentick, and Luc Segoufin. Two-variable logic on data words. *ACM Trans. Comput. Log.*, 12(4):27, 2011.
- [BFG⁺15] Michael Blondin, Alain Finkel, Stefan Göller, Christoph Haase, and Pierre McKenzie. Reachability in two-dimensional vector addition systems with states is pspace-complete. In *30th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS15)*, pages 32–43, 2015.
- [BFL15] Pablo Barceló, Gaëlle Fontaine, and Anthony W. Lin. Expressive path queries on graph with data. *Logical Methods in Comp. Sci.*, 11(4), 2015.
- [BLLW12] Pablo Barceló, Leonid Libkin, Anthony W. Lin, and Peter T. Wood. Expressive languages for path queries over graph-structured data. *ACM Transactions on Database Systems*, 37:31:1–31:46, 2012.
- [CDGL⁺06] Diego Calvanese, Giuseppe De Giacomo, Domenico Lembo, Maurizio Lenzerini, and Riccardo Rosati. Data complexity of query answering in description logics. In *Proceedings 10th International Conference on Principles of Knowledge Representation and Reasoning (KR06)*, volume 6, pages 260–270, 2006.
- [CDLV00] Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Moshe Y. Vardi. Containment of conjunctive regular path queries with inverse. In *Proceedings of the 7th International Conference Principles of Knowledge Representation and Reasoning (KR00)*, pages 176–185, 2000.
- [CLW14] Richard Cyganiak, Markus Lanthaler, and David Wood. RDF 1.1 concepts and abstract syntax. W3C Recommendation, 2014.
- [CM90] Mariano Consens and Alberto Mendelzon. GraphLog: a visual formalism for real life recursion. In *Proceedings of the 9th Symp. on Principles of Database Systems (PODS90)*, pages 404–416, 1990.
- [CM93] Mariano Consens and Alberto Mendelzon. Low complexity aggregation in GraphLog and Datalog. *Theor. Comp. Sci.*, 116(1):95–116, 1993.
- [CMW87] Isabel F. Cruz, Alberto O. Mendelzon, and Peter T. Wood. A graphical query language supporting recursion. In *Proc. of the ACM Special Interest Group on Management of Data (SIGMOD87)*, pages 323–330, 1987.
- [CMW88] Isabel F. Cruz, Alberto O. Mendelzon, and Peter T. Wood. G+: recursive queries without recursion. In *Expert Database Conf.*, pages 645–666, 1988.
- [dbe18] DB-Engines Ranking, 2018.
- [DLN07] Stéphane Demri, Ranko Lazic, and David Nowak. On the freeze quantifier in Constraint LTL: Decidability and complexity. *Inf. Comput.*, 205(1):2–24, 2007.
- [EM65] Calvin C. Elgot and Jorge E. Mezei. On relations defined by generalized finite automata. *IBM J. Res. Dev.*, 9(1):47–68, January 1965.
- [FGG⁺18] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. Cypher: An evolving query language for property graphs. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD 2018*, pages 1433–1445, 2018.
- [FL15a] Diego Figueira and Leonid Libkin. Path logics for querying graphs: Combining expressiveness and efficiency. In *Proceedings of the 30th Annual Symposium on Logic in Computer Science (LICS15)*, pages 329–340, 2015.
- [FL15b] Diego Figueira and Leonid Libkin. Synchronizing relations on words. *Theory Comput. Syst.*, 57(2):287–318, 2015.

- [FS93] Christiane Frougny and Jacques Sakarovitch. Synchronized Rational Relations of Finite and Infinite Words. *Theor. Comp. Sci.*, 108:45–82, 1993.
- [GMOW16] Maciej Graboń, Jakub Michaliszyn, Jan Otop, and Piotr Wieczorek. Querying data graphs with arithmetical regular expressions. In *Proceedings of the 25rd International Joint Conference on Artificial Intelligence (IJCAI16)*. AAAI Press, 2016.
- [HKdBZ13] Jelle Hellings, Bart Kuijpers, Jan Van den Bussche, and Xiaowang Zhang. Walk logic as a framework for path query languages on graph databases. In *Proceedings of the 16th International Conference on Database Theory (ICDT13)*, pages 117–128, 2013.
- [HS13] Steve Harris and Andy Seaborne. SPARQL 1.1 Query Language. W3C Recommendation, 2013.
- [KF94] Michael Kaminski and Nissim Francez. Finite-memory automata. *Theor. Comput. Sci.*, 134(2):329–363, 1994.
- [Klu82] Anthony C. Klug. Equivalence of relational algebra and relational calculus query languages having aggregate functions. *J. ACM*, 29(3):699–717, 1982.
- [KR03] Felix Klaedtke and Harald Rueß. Monadic second-order logics with cardinalities. In *Automata, Languages and Programming, 30th International Colloquium, ICALP 2003*, pages 681–696, 2003.
- [KT10] Eryk Kopczyński and Anthony Widjaja To. Parikh images of grammars: Complexity and applications. In *Proceedings of the 25th Symposium on Logic in Comp. Sci. (LICS10)*, pages 80–89, 2010.
- [LMV16] Leonid Libkin, Wim Martens, and Domagoj Vrgoč. Querying graphs with data. *J. ACM*, 63(2):14:1–14:53, March 2016.
- [LRSV18] Leonid Libkin, Juan L. Reutter, Adrián Soto, and Domagoj Vrgoč. Trial: A navigational algebra for RDF triplestores. *ACM Trans. Database Syst.*, 43(1):5:1–5:46, 2018.
- [LTV15] Leonid Libkin, Tony Tan, and Domagoj Vrgoč. Regular expressions for data words. *Journal of Computer and System Sciences*, 81(7):1278 – 1297, 2015.
- [MAB⁺10] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data, SIGMOD '10*, pages 135–146, 2010.
- [MOW17] Jakub Michaliszyn, Jan Otop, and Piotr Wieczorek. Querying best paths in graph databases. In *37th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2017*, pages 43:1–43:15, 2017.
- [MW95] Alberto Mendelzon and Peter Wood. Finding regular simple paths in graph databases. *SIAM Journal on Computation*, 24(6):1235–1258, 1995.
- [NSV04] Frank Neven, Thomas Schwentick, and Victor Vianu. Finite state machines for strings over infinite alphabets. *ACM Trans. Comput. Log.*, 5(3):403–435, 2004.
- [ope17] Cypher Query Language Reference. Version 9, 2017.
- [PS08] Eric Prud'hommeaux and Andy Seaborne. SPARQL Query Language for RDF. W3C Recommendation, 2008.
- [Sca84] Bruno Scarpellini. Complexity of subcases of Presburger arithmetic. *Transactions of the American Mathematical Society*, 284(1):203–218, 1984.
- [Seg06] Luc Segoufin. Automata and logics for words and trees over an infinite alphabet. In *Proceedings of the 20th Int. Workshop on Computer Science Logic (CSL06)*, pages 41–57, 2006.
- [The18] The Neo4j Team. The Neo4j Manual v3.4., 2018.
- [vRHK⁺16] Oskar van Rest, Sungpack Hong, Jinha Kim, Xuming Meng, and Hassan Chafi. Pqql: A property graph query language. In *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems, GRADES '16*. Association for Computing Machinery, 2016.
- [Woo12] Peter T. Wood. Query languages for graph databases. *SIGMOD Record*, 41(1):50–60, 2012.