Querying Graph Patterns

Pablo Barcelo
Department of Computer Science, Universidad de Chile
pbarcelo@dcc.uchile.cl

Leonid Libkin
School of Informatics, University of Edinburgh
libkin@inf.ed.ac.uk

Juan Reutter
School of Informatics, University of Edinburgh
juan.reutter@ed.ac.uk

ABSTRACT

Graph data appears in a variety of application domains, and many uses of it, such as querying, matching, and transforming data, naturally result in incompletely specified graph data, i.e., graph patterns. While queries need to be posed against such data, techniques for querying patterns are generally lacking, and properties of such queries are not well understood.

Our goal is to study the basics of querying graph patterns. We first identify key features of patterns, such as node and label variables and edges specified by regular expressions, and define a classification of patterns based on them. We then study standard graph queries on graph patterns, and give precise characterizations of both data and combined complexity for each class of patterns. If complexity is high, we do further analysis of features that lead to intractability, as well as lower-complexity restrictions. We introduce a new automata model for query answering with two modes of acceptance: one captures queries returning nodes, and the other queries returning paths. We study properties of such automata, and the key computational tasks associated with them. Finally, we provide additional restrictions for tractability, and show that some intractable cases can be naturally cast as instances of constraint satisfaction problem.

Categories and Subject Descriptors. H.2.1 [**Logical Design**]: *Data Models*; F.1.1 [**Models of Computation**]: *Automata*

General Terms. Theory, Languages, Algorithms

Keywords. Graph databases, graph patterns, query languages, complexity, automata, constraint satisfaction.

PODS'11, June 13–15, 2011, Athens, Greece.

1. INTRODUCTION

Querying and mining graph-structured data has received much attention lately, due to numerous applications in areas such as biological networks [32, 33, 35], social networks [37, 38], and the semantic Web [25, 36]. In such applications, the underlying data is naturally modeled as graphs, in which nodes are objects, and edge labels define relationships between those objects [2].

A standard way of querying graph data is to look for reachability patterns. Such patterns specify that paths satisfying certain conditions should exist between nodes. Initially proposed in a simple form in [15, 14], pattern languages have been developed over time and used in a variety of applications, such as biology, studying network traffic, crime detection, modeling object-oriented data, querying and searching RDF data, etc. [20, 21, 25, 26, 32, 33, 34, 36, 37, 38, 39, 40]; see also the survey [2]. In their simplest form, patterns are just graphs, whose occurrences in large graphs are of interest. Already in this simple form, they are very important in biological applications, where search for network motifs [33] is a common task. But for applications such as, for example, crime detection or RDF data, more complex patterns are needed, as one can look for connections between elements in a network that involve complex paths via some intermediaries.

The notions of finding matches for complex patterns also evolved with time, from traditional NP-complete subgraph isomorphism (used, nonetheless, in practical applications, e.g., in [12, 39]) to notions based on graph homeomorphisms (i.e., mapping edges to paths) and simulation relations between patterns and graphs [20, 21]. Outputs of matching queries are patterns themselves: their nodes are those that are involved in the simulation relation, and relationships between them are those specified in the pattern. For example, in a crime detection scenario, a query may output a set of individuals who might be involved in a crime network, together with descriptions of paths specifying their relationships. Similar scenarios arise in querying semistructured data as well, where it is sometimes natural to output incomplete query results [29]. When such matching and query results require extracting additional information

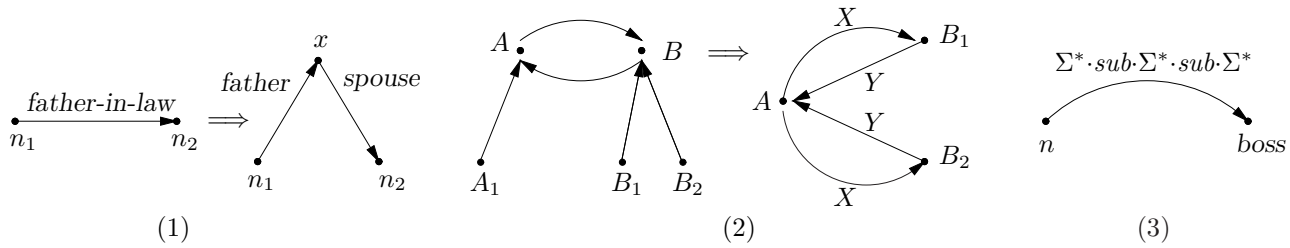


Figure 1: Examples of (1) node variables, (2) label variables, and (3) regular expressions

from them, one ends up querying patterns rather than graphs.

There are other scenarios where the need for querying patterns naturally arises. A pattern represents partial information about graph-structured data. Querying partial information is commonly present in integrating and exchanging (or translating) data [3, 19, 31]. In such applications, one queries the result of applying some schema mapping rules to source data, which yields a partially specified database. Partial databases – whether relational or XML – are typically viewed as patterns [27, 4, 6]. For graph data, the study of schema mappings and transformations for data exchange an integration has started recently [11, 38], but techniques for querying resulting partially specified graphs are currently lacking.

Motivated by these considerations, we would like to study querying partially defined graph data, i.e., graph patterns. As for other data models [3, 4, 19, 27, 31], one is looking for answers that are independent of the way in which the missing parts of patterns are interpreted, i.e., certain answers.

Based on the examples arising in querying and transforming graph data, we now analyze types of features that need to be addressed in the study of querying graph patterns. Recall that in the relational case, one deals with variables in place of missing data values [27]. In the case of XML, one may also have missing structural information [4]. For graph databases, partiality of specifications mainly arises in the following three ways.

Node variables Similarly to values missing in relational or XML data, identities of some nodes can be missing in graph data. For example, in transforming a social network that has different types of relationship edges, we can split an edge *(Name1, father-in-law, Name2)* into two edges *(Name1, father, x)* and *(x, spouse, Name2)*, with an unknown identity x. Variables can also be used to model blank nodes in RDF [36].

Label variables We may also miss the precise relationships between nodes. But even if we do not know them, we may still know that some of the relationships are the same. Taking an example from social networks, consider transforming a network where we have two 'celebrities' A and B who have 'followers' $A_1, \ldots, A_n$ and $B_1, \ldots, B_m$ (like on the Twitter network). Suppose we know the relationship between A and B (e.g., they like, or dislike each other). We may wish to record this as a relationship between their followers: for instance, if A hates B and A_i follows A, we may deduce something about how A_i relates to B. At the time of transforming a network we may not know the exact nature of such a relationship, but we know there exists one, and it should be the same for all the followers of A. Likewise, all the followers of B will be in some relationship with A (but not necessarily the same as the followers of A with B). So we add edges

$$(A_1, X, B), \ldots, (A_n, X, B),\ (B_1, Y, A), \ldots, (B_m, Y, A)$$

where X and Y are edge labels: we do not yet know what the relationship will be, but want to record that it is the same among all the followers.

Regular expressions Returning to the example with crime detection in a network of people, the result of a matching may contain facts like "there is a path between x and the boss that goes via at least two intermediaries", which will be expressed by a regular expression $\Sigma^* \cdot sub \cdot \Sigma^* \cdot sub \cdot \Sigma^*$, where *sub* indicates subordinacy in the hierarchy, and Σ is the set of all labels. In general, the situation where only regular paths between nodes can be deduced from a matching is very common [22]. Thus, when we do not have an exact path between two nodes, we attempt to replace it by an edge (A, e, B), where e is a regular expression.

These three examples are illustrated in Fig. 1. Once we have these features added to patterns, we need to define a query language for them. Most commonly used query languages for graph databases specify the existence of paths between nodes, with the restriction that the labels of such path belong to regular languages [1, 15, 14, 26, 10]. The simplest such queries are known as *regular path queries*, or RPQs [15]; those select nodes connected by a path that belongs to a regular language. *Conjunctive* RPQs, or CRPQs, extend them by allowing intermediate nodes in paths. Dealing with incomplete data, we often have *duality* between data and queries. For example, relational naive tables are tableaux of conjunctive queries, and in XML, typical query languages are based on tree patterns, i.e., incomplete descriptions of documents. We shall see that queries such as RPQs

and CRPQs arise as special cases of graph patterns, continuing the analogy with the well studied cases.

To sum up, our *main goal* is to define classes of graph patterns, study their properties, and query answering over them. Our main contributions are as follows.

1. We define classes of graph patterns that have the key features listed above – node variables, label variables, and edges labeled with regular expressions – and provide a complete classification of their expressiveness.
2. We study the complexity of query answering (i.e., the problem of finding certain answers to queries over graph patterns). We fully analyze it for CRPQs, both for data complexity (which ranges from NLOGSPACE to coNP) and for combined complexity (which ranges from NP to EXPSPACE). For classes of high complexity, we do an in-depth analysis, showing which features lead to intractability. We also show that upper bounds for CRPQs extend to more expressive queries.
3. We provide an automaton model for query answering. Specifically, we define a class of automata, called *incomplete automata*, that naturally give rise to two acceptance notions that precisely capture certain answers: one of them corresponds to queries that return nodes, and the other to queries that return paths. In the latter case, answers to queries are represented by NFAs. We analyze the complexity of incomplete automata, and prove lower bounds on the sizes of NFAs representing query answers.
4. Returning to the intractable cases for query answering, we look at two ways of reducing complexity: by imposing structural restrictions, and by reducing to problems for which many efficient heuristics are known. Along these lines, we prove that for several classes of graph patterns, the bounded treewidth restriction guarantees tractability. We also show how to cast finding certain answers as a constraint satisfaction problem, which allows us to use algorithmic techniques from that field.

Organization In Section 2 we define graph databases and queries over them. In Section 3 we define graph patterns and in Section 4 we study their classifications and structural properties. In Section 5 we analyze both data and combined complexity of query answering. In Section 6 we deal with incomplete automata, and relate them to answering queries over graph patterns. In Section 7 we look at tractability restrictions and reduction to constraint satisfaction.

2. GRAPH DATABASES, RPQS AND CRPQS

A *graph database* [2, 10, 15] is just a finite edge-labeled graph. Let Σ be a finite alphabet, and $\mathcal{N}$ a countably infinite set of node ids. Then a graph database over Σ is a pair $G = (N, E)$, where N is the set of nodes (a finite subset of $\mathcal{N}$), and E is the set of edges, i.e., $E \subseteq N \times \Sigma \times N$. That is, we view each edge as a triple (n, a, n'), whose interpretation, of course, is an a-labeled edge from n to n'. When Σ is clear from the context, we shall simply speak of a graph database.

A *path* ρ from n_0 to n_m in G is a sequence (n_0, a_0, n_1), $(n_1, a_1, n_2), \cdots, (n_{m-1}, a_{m-1}, n_m)$, for some $m \geq 0$, where each (n_i, a_i, n_{i+1}), for $i < m$, is an edge in E. In particular, all the n_i's are nodes in N and all the a_j's are letters in Σ. The *label* of ρ, denoted by $\lambda(\rho)$, is the word $a_0 \cdots a_{m-1} \in \Sigma^*$. We also define the empty path as (n, ϵ, n) for each $n \in N$; the label of such path is the empty word ϵ.

The basic querying mechanism for graph databases is provided by means of *regular path queries*, or *RPQs* [1, 15, 10]. They retrieve pairs of nodes in a graph database connected by a path whose label belongs to a given regular language. Formally, an RPQ Q is an expression of the form (x, L, y) where $L \subseteq \Sigma^*$ is a regular language. We shall assume that syntactically L is given as a regular expression. Given a graph database $G = (N, E)$ and an RPQ Q, both over Σ, the answer $Q(G)$, is the set of all pairs $(n, n') \in N$ such that there is path ρ between them whose label $\lambda(\rho)$ is in L.

It has been argued (see, e.g., [1, 15, 14, 7]) that analogs of conjunctive queries whose atoms are RPQs are much more useful in practice than simple RPQs. In such queries, multiple RPQs can be combined, and some variables can be existentially quantified. Formally, a *conjunctive regular path query*, or *CRPQ* Q over a finite alphabet Σ is an expression of the form:

$$Ans(\bar{z}) \leftarrow \bigwedge_{1 \leq i \leq m} (x_i, L_i, y_i), \tag{1}$$

such that $m > 0$, each (x_i, L_i, y_i) is an RPQ, and $\bar{z}$ is a tuple of variables among $\bar{x}$ and $\bar{y}$. A query with the head $Ans()$ (i.e., no variables in the output) is called a *Boolean* query.

Intuitively, such a query Q selects tuples $\bar{z}$ for which there exist values of the remaining node variables from $\bar{x}$ and $\bar{y}$ such that each RPQ in the body is satisfied. Formally, given Q of the form (1) and a graph $G = (N, E)$, a valuation is a map $\sigma : \bigcup_{1 \leq i \leq m} \{x_i, y_i\} \to N$. We write $(G, \sigma) \models Q$ if $(\sigma(x_i), \sigma(y_i))$ is in the answer to RPQ $(x_i.L_i, y_i)$ in G, i.e., if there is a path ρ_i in G from $\sigma(x_i)$ to $\sigma(y_i)$ with $\lambda(\rho_i) \in L_i$. Then $Q(G)$ is the set of all tuples $\sigma(\bar{z})$ such that $(G, \sigma) \models Q$. If Q is Boolean, we let $Q(G)$ be `true` if $(G, \sigma) \models Q$ for some σ (that is, as usual, the singleton set with the empty tuple models `true`, and the empty set models `false`).

We also allow existentially quantified variables in path queries. That is, RPQs will be of the form $Ans(\bar{z}) \leftarrow (x, L, y)$, where $\bar{z}$ contains variables from $\{x, y\}$. For example, $Ans() \leftarrow (x, L, y)$ is a Boolean RPQ checking whether there is a path whose label is in L.

3. GRAPH PATTERNS

As explained in the introduction, the key new features of graph patterns are the ability to use the following (in addition to nodes and edge labels of graph databases):

- node variables, i.e., marked nulls for graph nodes;
- label variables, i.e., marked nulls for edge labels;
- regular expressions as labels for edges.

Thus, we shall define graph patterns as graph databases over constant nodes and node variables, whose edges will be labeled with regular expressions that may use label variables. To do this, we shall use the following (countably infinite) sets:

- $\mathcal{V}_{\text{node}}$ of *node variables* (normally denoted by lower-case letters), and
- $\mathcal{V}_{\text{lab}}$ of *label variables* (normally denoted by upper-case letters).

If Γ is an arbitrary (finite or infinite) set of symbols, we write $\text{REG}(\Gamma)$ to denote the set of nonempty regular languages over Γ (if Γ is infinite, then each $L \in \text{REG}(\Gamma)$ only uses finitely many symbols from Γ). Recall that a graph database over a labeling alphabet Σ was defined as a labeled graph, (N, E), where $N \subseteq \mathcal{N}$ is the set of nodes and $E \subseteq N \times \Sigma \times N$ is the set of labeled edges. We are now in a position to define graph patterns formally.

Definition 1. *A* graph pattern *over finite alphabet Σ is a pair $\pi = (N, E)$ where*

- $N \subseteq \mathcal{N} \cup \mathcal{V}_{\text{node}}$ *is the finite set of nodes, and*
- $E \subseteq N \times \text{REG}(\Sigma \cup \mathcal{V}_{\text{lab}}) \times N$ *is the set of edges.*

Semantics In complete analogy with relational naive tables or incomplete XML documents, the semantics is defined via homomorphisms. To define those, we need extensions of partial functions $f : \Gamma \to \Gamma$ to languages $L \in \text{REG}(\Gamma)$ defined as $f(L) = \{f(w) \mid w \in L\}$, where $f(w)$ is obtained by replacing each symbol a of a word w on which f is defined by $f(a)$, and leaving symbols b on which f is not defined intact.

Since variables can occur at the level of both nodes and edge labels, homomorphisms will be in fact *pairs* of mappings. Given a graph database $G = (N, E)$ and a pattern $\pi = (N', E')$, a *homomorphism* $h : \pi \to G$ is a pair $h = (h_1, h_2)$ of mappings $h_1 : N' \to N$ and h_2 that maps label variables used in π to labels used in G such that:

1. $h_1(n) = n$ for every node id $n \in \mathcal{N}$; and
2. for every edge $(p, L, p') \in E'$, there is path between $h_1(p)$ and $h_1(p')$ in G whose label is in $h_2(L)$.

We now write $G \models \pi$ if there is a homomorphism $h : \pi \to G$. The semantics is defined with respect to a labeling alphabet Σ:

$$[\![\pi]\!]_\Sigma \quad = \quad \{G \text{ over } \Sigma \mid G \models \pi\}.$$

Most often Σ is clear from the context and we write simply $[\![\pi]\!]$ then.

Example 1. An illustration is given in Fig. 2: a homomorphism is defined by letting label variable X be b, and by mapping both node variables x and y into n_3. The edge $(n_1, (a|b)(a|b), x)$ is then mapped into the *path* $(n_1, a, n_4), (n_4, b, n_3)$ with label ab. The edge $(n_1, (ab)^*, y)$ is mapped into the same path, since ab belongs to regular languages denoted by both $(a|b)(a|b)$ and $(ab)^*$. The edge (y, a^*X, n_2) is mapped into (n_3, b, n_2), since b is in the language denoted by a^*b.

Certain answers Consider queries Q that take graph databases as input and return sets of tuples of their nodes. For example, RPQs and CRPQs are such queries. For them, we can define their certain answers on graph patterns in the standard way:

$$\text{certain}_\Sigma(Q, \pi) \ = \ \bigcap\{Q(G) \mid G \in [\![\pi]\!]_\Sigma\}.$$

Again, if Σ is clear from the context, we write simply $\text{certain}(Q, \pi)$.

Example 2. The labeling alphabet can make a difference in finding certain answers. Consider a pattern with edges $(n_1, a, n_2), (n_2, X, n_3), (n_3, b, n_4)$, where X is a label variable. Let Q be the Boolean RPQ $Ans() \leftarrow (x, ab, y)$. Then $\text{certain}_{\{a,b\}}(Q, \pi) =$ true: whether X is a or b, there is a path labeled ab. However, $\text{certain}_{\{a,b,c\}}(Q, \pi) =$ false (by setting $X = c$). □

Graph patterns as queries Graph patterns can naturally be viewed as queries – again in complete analogy with relational databases (where naive tables are a natural representation of conjunctive queries, i.e., tableaux) and XML documents (where tree patterns form the basis of tree conjunctive queries [6, 24]). This view has also been explored in [13].

We adopt the convention that patterns used in queries are denoted by ξ, and patterns used as data are denoted by π. A *graph query* is a pair $Q = (\xi, \bar{x})$, where $\xi = (N, E)$ is a graph pattern, and $\bar{x}$ is a tuple of elements from N. For example, a CRPQ $Ans(\bar{z}) \leftarrow \bigwedge_{i \le m}(x_i, L_i, y_i)$, can be viewed as a graph query $(\xi, \bar{z})$, where ξ simply contains the edges (x_i, L_i, y_i) for $i \le m$.

We now define the semantics of a graph query on graph databases (later, we shall extend it to graph patterns). Given a graph database $G = (N, E)$ with $N \subset \mathcal{N}$, and a graph query $Q = (\xi, \bar{x})$ with $|\bar{x}| = k$, the answer to Q on G is $Q(G) = \{\bar{v} \in N^k \mid G \models \xi[\bar{v}/\bar{x}]\}$. Here $\xi[\bar{v}/\bar{x}]$ is the result of substituting $\bar{v}$ for $\bar{x}$ in the pattern ξ.

It is easy to see that when Q is a CRPQ viewed as a graph query, the result $Q(G)$ coincides with the standard semantics of CRPQs.

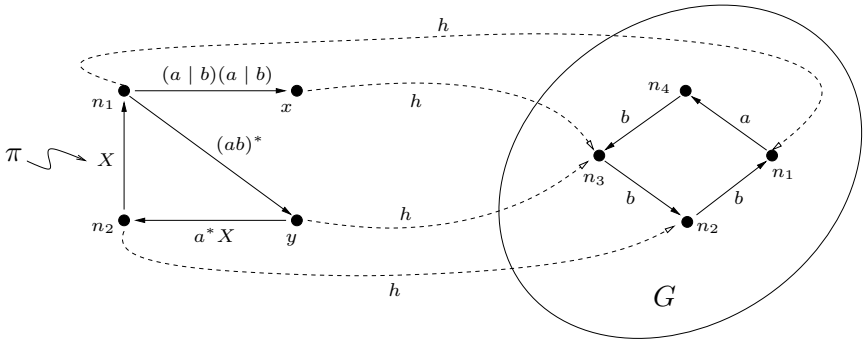


Figure 2: A homomorphism $h : \pi \to G$

Example 3. Consider again the example in Fig. 2 and the homomorphism described in Example 1. Let ξ be the pattern obtained from π by changing X to b, and replacing n_1 and n_2 with variables z_1 and z_2. The resulting pattern can be viewed as a CRPQ (ξ, x, y):

$$Ans(x,y) \leftarrow (z_1,(a|b)(a|b),x),\ (z_1,(ab)^*,y),\ (y,a^*b,z_2),\ (z_2,b,z_1)$$

If it is evaluated in graph G shown in Fig. 2, one tuple in the output will be (n_3, n_3), since $G \models \xi[n_3/x, n_3/y]$, as witnessed by homomorphism h. □

4. CLASSIFICATION AND BASIC PROPERTIES

The three key features of graph patterns – node variables, label variables, and regular expressions – provide a natural classification of patterns. We shall refer to classes of patterns as $\boldsymbol{\mathcal{P}}^{\sigma}$, where σ enumerates the present features. We use 'nv' for node variables, 'lv' for label variables, and 're' for regular expressions. This gives us 8 classes, from $\boldsymbol{\mathcal{P}}$ (none of the features is present) to $\boldsymbol{\mathcal{P}}^{\mathrm{nv,lv,re}}$ (all are present).

Of course $\boldsymbol{\mathcal{P}}$ is the class of graph databases (N, E) with $N \subseteq \mathcal{N}$ and $E \subseteq N \times \Sigma \times N$, and $\boldsymbol{\mathcal{P}}^{\mathrm{nv,lv,re}}$ is the class of all graph patterns as in Definition 1 with $N \subseteq \mathcal{N} \cup \mathcal{V}_{\mathrm{node}}$ and $E \subseteq N \times \mathrm{REG}(\Sigma \cup \mathcal{V}_{\mathrm{lab}}) \times N$. We now examine some others.

$\boldsymbol{\mathcal{P}}^{\mathrm{nv}}$ is the class of graphs where nodes could be either constants, or node variables; all edges are labeled with alphabet letters, i.e. $N \subseteq \mathcal{N} \cup \mathcal{V}_{\mathrm{node}}$ and $E \subseteq N \times \Sigma \times N$. These patterns can be represented by relational naive tables.

$\boldsymbol{\mathcal{P}}^{\mathrm{nv,re}}$ is the class of patterns where nodes could be either constants or node variables, and edges are labeled with regular expressions over Σ. That is, $N \subseteq \mathcal{N} \cup \mathcal{V}_{\mathrm{node}}$ and $E \subseteq N \times \mathrm{REG}(\Sigma) \times N$.

These are essentially CRPQs, which are graph queries $(\xi, \bar{x})$ where ξ is from $\boldsymbol{\mathcal{P}}^{\mathrm{nv,re}}$ and uses only node variables (without this restriction we have the class of CRPQs that can mention constants).

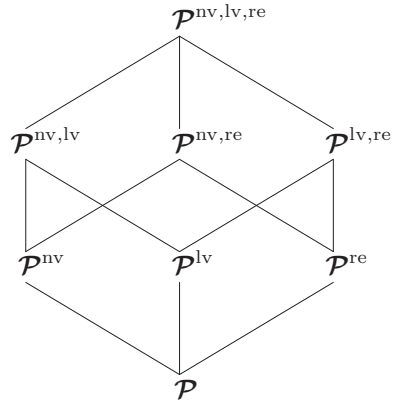
$\boldsymbol{\mathcal{P}}^{\mathrm{nv,lv}}$ is the class of patterns where nodes could be either constants or node variables, and edges are labeled with letters or variables. That is, $N \subseteq \mathcal{N} \cup \mathcal{V}_{\mathrm{node}}$ and $E \subseteq N \times (\Sigma \cup \mathcal{V}_{\mathrm{lab}}) \times N$. The class $\boldsymbol{\mathcal{P}}^{\mathrm{lv}}$ is its restriction when $N \subseteq \mathcal{N}$.

Given multiple features of graph patterns, it is natural to ask whether all are necessary, or some are expressible with others. We now show that all three are essential.

Classes of patterns are of the form $\boldsymbol{\mathcal{P}}^{\sigma}$, where σ is a subset of $\{\mathrm{nv}, \mathrm{lv}, \mathrm{re}\}$. We write $\boldsymbol{\mathcal{P}}^{\sigma} \preceq \boldsymbol{\mathcal{P}}^{\sigma'}$ if $\boldsymbol{\mathcal{P}}^{\sigma'}$ is at least as expressive as $\boldsymbol{\mathcal{P}}^{\sigma}$: for every pattern $\pi \in \boldsymbol{\mathcal{P}}^{\sigma}$, there is a pattern $\pi' \in \boldsymbol{\mathcal{P}}^{\sigma'}$ so that $[\![\pi]\!] = [\![\pi']\!]$ (i.e., $[\![\pi]\!]_{\Sigma} = [\![\pi']\!]_{\Sigma}$ for each Σ containing the labels used in π). We write $\boldsymbol{\mathcal{P}}^{\sigma} \sim \boldsymbol{\mathcal{P}}^{\sigma'}$ if $\boldsymbol{\mathcal{P}}^{\sigma}$ and $\boldsymbol{\mathcal{P}}^{\sigma'}$ are equally expressive (i.e., $\boldsymbol{\mathcal{P}}^{\sigma} \preceq \boldsymbol{\mathcal{P}}^{\sigma'}$ and $\boldsymbol{\mathcal{P}}^{\sigma'} \preceq \boldsymbol{\mathcal{P}}^{\sigma}$). Finally, $\boldsymbol{\mathcal{P}}^{\sigma} \prec \boldsymbol{\mathcal{P}}^{\sigma'}$ means that $\boldsymbol{\mathcal{P}}^{\sigma'}$ is strictly more expressive than $\boldsymbol{\mathcal{P}}^{\sigma}$: that is, $\boldsymbol{\mathcal{P}}^{\sigma} \preceq \boldsymbol{\mathcal{P}}^{\sigma'}$, but they are not equally expressive.

Theorem 1. *Adding each new feature to graph patterns strictly increases their expressiveness: in other words,* $\boldsymbol{\mathcal{P}}^{\sigma} \prec \boldsymbol{\mathcal{P}}^{\sigma'}$ *iff* $\sigma \subsetneq \sigma'$, *and* $\boldsymbol{\mathcal{P}}^{\sigma} \sim \boldsymbol{\mathcal{P}}^{\sigma'}$ *iff* $\sigma = \sigma'$.

These relationships are visualized in the figure below.



In both relational and XML patterns it is common to consider a restriction in which variables cannot be repeated. In relations, these are Codd tables [27] that

model SQL's nulls. We say that a graph pattern is a *Codd pattern* if every variable – node or label – occurs at most once in it. In other words, Codd patterns do not allow us to express equality between unknown entities.

If σ contains nv or lv, we shall write $\mathcal{P}^{\sigma}_{\text{Codd}}$ for the Codd patterns in $\mathcal{P}^{\sigma}$. We next show that Codd patterns are strictly weaker than the usual ones, and describe classes of patterns for which adding variables under Codd interpretation increases expressiveness.

Proposition 1.
- *Codd patterns are strictly less expressive: $\mathcal{P}^{\sigma}_{\text{Codd}} \prec \mathcal{P}^{\sigma}$ when σ contains* nv *or* lv.
- *Adding variables under Codd interpretation makes patterns more expressive except adding label variables to regular expressions. That is, if $\sigma' \subsetneq \sigma$ and $\sigma - \sigma'$ contains either* nv *or* lv*, then $\mathcal{P}^{\sigma'} \prec \mathcal{P}^{\sigma}_{\text{Codd}}$ except one case: $\mathcal{P}^{\text{re}} \sim \mathcal{P}^{\text{lv,re}}_{\text{Codd}}$.*

5. QUERY ANSWERING

The goal of this section is to study the complexity – both data and combined – of query answering over graph patterns. Recall that for queries Q returning tuples of nodes, we want to find certain answers defined as $\text{certain}(Q,\pi) = \bigcap\{Q(G) \mid G \in [\![\pi]\!]\}$. More precisely, one needs to find $\text{certain}_{\Sigma}(Q,\pi)$, with G ranging over graph databases with edges labeled in Σ; it will be clear from the proofs, however, that the complexity of query answering does not depend on the labeling alphabet.

Since each class of patterns gives rise to a class of graph queries $Q = (\xi, \bar{x})$, one could potentially ask for the exact bounds on combined and data complexity for all these classes of queries on all the classes of patterns. Of course we are not going to consider all the resulting 128 cases. Instead, we do the following.

As our benchmark language we use CRPQs, and provide exact complexity bounds for CRPQs over all classes of patterns. Recall that CRPQs can be viewed as graph queries $(\xi, \bar{x})$ with $\xi \in \mathcal{P}^{\text{nv,re}}$. We then show that the upper bounds for CRPQs extend to the most expressive patterns from $\mathcal{P}^{\text{nv,lv,re}}$. After that, we delve further into intractable cases, and analyze what really causes intractability. In such cases, we consider restricted classes of queries based on simpler graph patterns. We shall discuss lower bounds for more expressive patterns in the full version of the paper.

Certain answers as pattern implication It is a standard and yet useful observation that the problem of computing certain answers can be cast as the problem of *implication of patterns*. Recall that pattern implication is defined as follows: if π_1 and π_2 are two patterns, then we say that π_1 *implies* π_2, and write $\pi_1 \models \pi_2$ if $[\![\pi_1]\!] \subseteq [\![\pi_2]\!]$. In other words, $\pi_1 \models \pi_2$ if $G \models \pi$ entails $G \models \pi_2$ for every graph database G. The following is now immediate from the definitions.

Lemma 1. *Given a graph pattern $\pi = (N, E)$ and a graph query $Q = (\xi, \bar{x})$ with $|\bar{x}| = k$,*

$$\text{certain}(Q,\pi) \;=\; \{\bar{v} \in N^k \mid \pi \models \xi[\bar{v}/\bar{x}]\}.$$

For Boolean graph queries $Q = (\xi, ())$ with the empty tuple of output variables (i.e., true/false queries), Lemma 1 states that $\text{certain}(Q,\pi) = \text{true}$ iff $\pi \models \xi$. This simple connection with the implication problem will let us use known results on containment of CRPQs [7] to obtain some of the bounds for the combined complexity of query answering.

Remark: using naive evaluation Some classes of patterns can be represented as naive tables, perhaps with constraints. For example, patterns from $\mathcal{P}^{\text{nv}}$ can be stored as naive tables, and patterns without regular expressions (from $\mathcal{P}^{\text{nv,lv}}$) are represented as relational naive tables with an additional constraint that the interpretation for label variables must come from the labeling alphabet Σ. This can easily be coded as an inclusion constraint.

Since CRPQs can be expressed in datalog, such a representation gives us good tractable bounds for data complexity for $\mathcal{P}^{\text{nv}}$ patterns. But for combined complexity, and for data complexity for other classes, we cannot use known results to get tight bounds. For example, even evaluating conjunctive queries over naive tables with inclusion constraints is known to be PSPACE-hard [28], and we shall see better bounds obtained for CRPQs over $\mathcal{P}^{\text{nv,lv}}$ patterns.

5.1 Combined complexity

The problem we are dealing with is as follows:

INPUT:	a pattern $\pi = (N, E)$, a graph query $Q = (\xi, \bar{x})$ with $\lvert\bar{x}\rvert = k$, a tuple $\bar{v} \in N^k$.
QUESTION:	Is $\bar{v} \in \text{certain}(Q,\pi)$?

Checking $\bar{v} \in \text{certain}(Q,\pi)$ amounts to checking $\pi \models \xi[\bar{v}/\bar{x}]$, and the problem is known to be EXPSPACE-complete when both π and ξ are in $\mathcal{P}^{\text{nv,re}}$ [7]. We now provide a complete analysis of the complexity.

Theorem 2. *The combined complexity of answering CRPQs over classes of graph patterns is as shown in Figure 3.*

The abbreviation '-c.' in the figure means, of course, complete for the class. The combined complexity of CRPQs on usual graph databases is the same as the combined complexity of conjunctive queries over the usual relational database, i.e., NP-complete. Thus, adding node variables comes with no cost, while adding both node and label variables carries a small cost in terms of combined complexity (jumping up one level in the polynomial hierarchy). Adding regular expressions comes at a significant cost (jumping up an exponential).

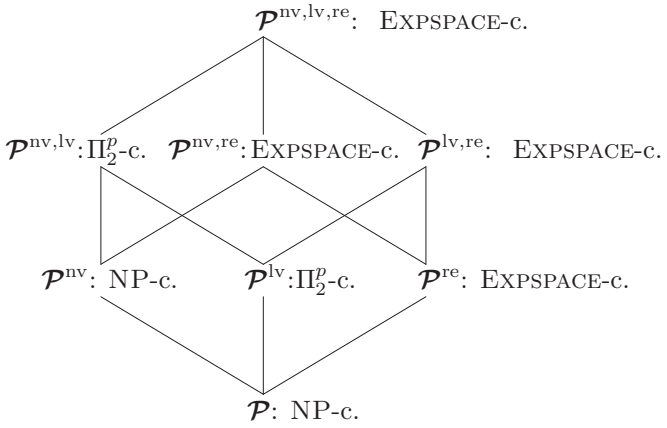


Figure 3: Combined complexity for CRPQs over graph patterns

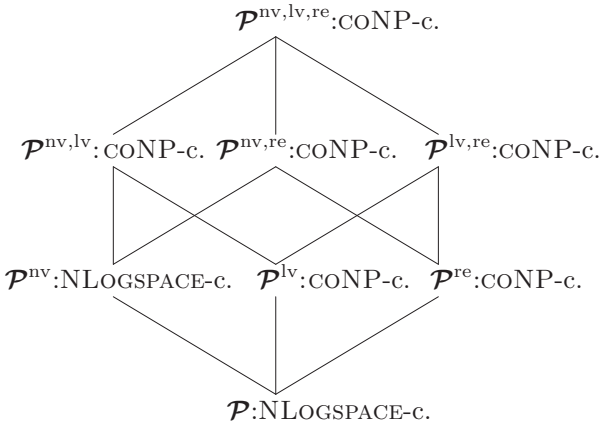


Figure 4: Data complexity for CRPQs over graph patterns

Using essentially the same techniques as in [7], we can prove that the previous upper bound extends beyond CRPQs.

Proposition 2. *The combined complexity of arbitrary graph queries on arbitrary patterns is in* EXPSPACE.

The next question is whether we can lower the EXPSPACE bound for patterns in $\mathcal{P}^{\text{re}}$. There are two natural ways of looking for better behaved subclasses: by restricting queries, or restricting patterns. Restrictions on queries by means of simplifying regular languages were studied in [17]. For example, it showed that for regular languages built with concatenation and the Kleene star, the combined complexity drops to Π_2^p-complete. Another possibility is to restrict to RPQs; then, using techniques similar to [7], we can prove a PSPACE bound, matching the combined complexity of relational calculus. It also follows from [7] that restricting the class of patterns does not help lower the combined complexity.

Proposition 3.
- *The combined complexity of answering CRPQs on patterns $\pi \in \mathcal{P}^{\text{re}}$ is* EXPSPACE*-hard even for patterns π that contain a single edge.*
- *The combined complexity of answering RPQs on graph patterns from $\mathcal{P}^{\text{nv,lv,re}}$ is* PSPACE*-complete. The problem remains* PSPACE*-hard even for answering RPQs on patterns $\pi \in \mathcal{P}^{\text{re}}$ that contain a single edge.*

5.2 Data complexity

We now turn to data complexity, i.e. the complexity of query answering when the query is fixed. In what follows, Q refers to a graph query $(\xi, \bar{x})$ with $|\bar{x}| = k$.

PROBLEM:	DATA COMPLEXITY(Q)
INPUT:	a pattern $\pi = (N, E)$, a tuple $\bar{v} \in N^k$.
QUESTION:	Is $\bar{v} \in \text{certain}(Q, \pi)$?

This too can be viewed as a pattern-implication problem $\pi \models \xi[\bar{v}/\bar{x}]$ but for a *fixed* pattern ξ.

As already mentioned, some cases are simple: for example, patterns in $\mathcal{P}$ are graphs, and thus due to the monotonicity of CRPQs, computing certain answers is the same as evaluating CRPQs on graphs, i.e., NLOGSPACE-complete. Similarly, since $\mathcal{P}^{\text{nv}}$ patterns can be represented as a naive table, and since CRPQ queries can be translated into datalog, we retain an NLOGSPACE bound. For other cases, as it turns out, the complexity is intractable.

Theorem 3. *The data complexity of answering CRPQs over classes of graph patterns is as shown in Figure 4.*

Proof sketch. The coNP upper bound follows from a more general result presented in Proposition 4.

We have already explained how to obtain the NLOGSPACE upper bounds, thus we only need to show coNP-hardness for $\mathcal{P}^{\text{re}}$ and $\mathcal{P}^{\text{lv}}$. We now present a simple hardness proof for $\mathcal{P}^{\text{re}}$. It will be tightened significantly (and extended to $\mathcal{P}^{\text{lv}}$) in the remainder of the section.

For $\mathcal{P}^{\text{re}}$, we use reduction to non-3-colorability. Assume we have an arbitrary undirected graph G; we represent it as a labeled graph where between two nodes n_1 and n_2 connected by an edge we have two edges labeled a, i.e., (n_1, a, n_2) and (n_2, a, n_1). Now we turn it into a $\mathcal{P}^{\text{re}}$ pattern π_G over the alphabet $\{a, r, g, b\}$ by adding edges $(n, rr|gg|bb, n)$ for each node n. That is, in every graph represented by this pattern, associated with each node n there is another node n' and edges $(n, \ell, n'), (n', \ell, n)$ where ℓ is one of r, g, b. It is now easy to see that the certain answer to the Boolean RPQ $Ans() \leftarrow (x, rar|gag|bab, y)$ over π_G is true iff G is not 3-colorable. □

The upper bound again extends to arbitrary queries. In order to prove this, we apply similar techniques to

those used in [8] to show that the data complexity of the problem of answering RPQs using views is in coNP.

Proposition 4. *Data complexity of arbitrary graph queries over arbitrary graph patterns is in* coNP.

Looking at Figure 4, we see that there are two features that cause coNP-hardness: label variables, and regular expressions. We now analyze their role in causing the high complexity of query answering. In both cases, we need to investigate two ways of lowering the complexity: by restricting queries, and by restricting their inputs.

The role of label variables For restrictions on queries, we shall look at simple RPQs. To define restrictions on inputs, we use the notion of the *underlying graph* G_π of a pattern $\pi = (N, E)$: this is simply the graph obtained by erasing labels on edges, i.e. $G_\pi = (N, \{(v_1, v_2) \mid (v_1, L, v_2) \in E)$.

We now show that the coNP-hardness result is very robust. Recall that $\boldsymbol{\mathcal{P}}^{\sigma}_{\text{Codd}}$ stands for class of Codd patterns in $\boldsymbol{\mathcal{P}}^{\sigma}$, i.e., patterns that use each variable once.

Theorem 4.
- *There is a Boolean RPQ Q such that* Data complexity*(Q) is* coNP*-hard even over input patterns in $\boldsymbol{\mathcal{P}}^{\text{lv}}$ whose underlying graph is a path. Moreover, the regular language in Q is built using only concatenation and the Kleene star.*
- *There is a Boolean RPQ Q of the form $Ans() \leftarrow (x, w, y)$, where w is a word in $\{0,1\}^*$, such that* Data complexity*(Q) is* coNP*-hard even over $\boldsymbol{\mathcal{P}}^{\text{lv}}_{\text{Codd}}$ patterns whose underlying graph is a DAG.*

The only possibility for a polynomial-time query answering algorithm left open by this result appears to be Codd patterns in $\boldsymbol{\mathcal{P}}^{\text{lv}}$ with very nice underlying graphs. We shall see in Section 7, when we study tractable restrictions, that there is indeed a tractable class obtained along these lines.

The role of regular expressions In the case of patterns from $\boldsymbol{\mathcal{P}}^{\text{re}}$ we have an additional parameter to vary: the regular expressions used in patterns. Nevertheless, we shall see that coNP-hardness is already witnessed by very simple regular expressions.

Theorem 5.
- *There exists a Boolean RPQ Q of the form $Ans() \leftarrow (x, w, y)$, where w is a single word over $\Sigma = \{0,1\}$, such that* Data complexity*(Q) is* coNP*-hard even over input patterns in $\boldsymbol{\mathcal{P}}^{\text{re}}$ over Σ whose underlying graph is a DAG. It remains* coNP*-hard even if each regular expression used in input patterns is $0|1$.*
- *There exists a Boolean RPQ Q such that* Data complexity*(Q) is* coNP*-hard even over input patterns in $\boldsymbol{\mathcal{P}}^{\text{re}}$ that only use regular expressions of the form a, for $a \in \Sigma$, or $a_1^* \dots a_n^*$, where the a_i's are distinct letters in Σ.*

Like the case of patterns with label variables, this leaves open the possibility that more restrictive underlying graphs may lead to tractability. Indeed, we shall prove such results in Section 7.

6. INCOMPLETE AUTOMATA FOR QUERYING PATTERNS

Notice that graph databases can be viewed as finite automata. Graph patterns in turn can be viewed as *incomplete automata*. We now define those, and show that they naturally generate two notions of acceptance. These notions correspond to certain answers: one for certain answers as we defined them, and the other for certain answers for queries that can output paths.

Extensions of CRPQs outputting paths have been defined in [5]. We shall present this notion for RPQs (for CRPQs, it includes the concept of synchronizing paths, which will complicate the presentation). An *RPQ with a path output* is a query of the form

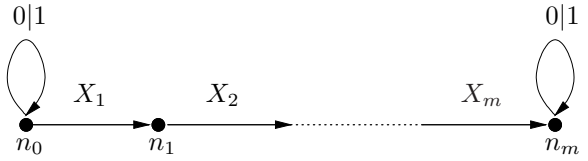
$$Ans(\bar{z}, \rho) \ \leftarrow \ (x, \rho : L, y)$$

where, on top of the usual RPQ $Ans(\bar{z}) \leftarrow (x, L, y)$, one is allowed to name the path ρ witnessing the query, and to output its label. Of course the number of L-paths between two nodes could be infinite, but one easily observes that for every nodes n_1, n_2 in a graph database, the set of labels of L-paths between them is regular, and thus can be represented by a finite automaton.

Assume we have an RPQ Q with a path variable, as above, and a graph pattern π. Let n_1, n_2 be two nodes from $\mathcal{N}$ that occur in π. We say that a word $\rho \in \Sigma^*$ is a *certain path between n_1 and n_2 with respect to Q* if for every $G \in [\![\pi]\!]$, there is an L-path between n_1 and n_2 with label ρ. The set of such certain paths will be denoted by $\text{certain}^{\text{path}}(Q; \pi, n_1, n_2)$. We shall write $\text{certain}^{\text{path}}_{\Sigma}$ when Σ is not clear from the context.

The following example illustrates this concept.

Example 4. For $m > 0$, consider the pattern π_m over $\Sigma = \{0, 1\}$ shown in the figure below.



Notice that each $G \in [\![\pi_m]\!]$ will contain a path from node n_0 to node n_m. In particular, (n_0, n_m) is a certain answer to the RPQ Q given by $(x, \rho : (0|1)^*, y)$.
However, one can see that every word in $\text{certain}^{\text{path}}_{\Sigma}(Q; \pi, n_0, n_m)$ must contain, as subwords, all the 2^m words of length m over $\{0,1\}$ since the X_is can be instantiated arbitrarily. Due to the presence of the loops, the converse also holds, and $\text{certain}^{\text{path}}_{\Sigma}(Q; \pi, n_0, n_m)$ consists precisely of the words that contain all the 2^m subwords of length m. In particular, the smallest certain paths are precisely the non-circular De Bruijn sequences of order m, and thus

have length $2^m + m - 1$. One can also easily show that any NFA accepting $\text{certain}_{\Sigma}^{\text{path}}(Q; \pi, n_0, n_m)$ will have exponentially many states (in m). □

This example suggests that the problem of computing the certain paths is inherently different from the problem of computing certain answers for graph patterns, and thus we need to develop new tools for solving this problem. This is what we do next.

6.1 Incomplete automata and certain answers

For convenience, we shall assume that NFAs can have edges labeled by words. That is, NFAs will be of the form $\mathcal{A} = (Q, \Sigma, q_0, F, \delta)$, where Q is the set of states, Σ is the alphabet, q_0 is the initial state, F is the set of final states, and the transition relation δ is a finite subset of $Q \times \Sigma^* \times Q$. The notion of acceptance extends to such an automaton in the standard way: if there is a transition (q, w, q'), the automaton is in state q, then, if w is a subword that starts in the current position, the automaton skips it and moves to the state q'. When all the w's used in transitions are single letters, this is the standard notion of NFAs; in that case we shall refer to them as *standard NFAs*.

The language accepted by an NFA is denoted by $L(\mathcal{A})$. Note that for each NFA, one can construct, in polynomial time, a standard NFA $\mathcal{A}'$ such that $L(\mathcal{A}) = L(\mathcal{A}')$. This is done by converting each word in a transition into a DFA (in polynomial time) and plugging it in place of the transition. Hence, using extended transitions is indeed just a matter of convenience.

Definition 2. *An incomplete automaton $\mathcal{A}$ is a tuple $\mathcal{A} = (Q, \Sigma, \mathcal{W}, q_0, F, \delta)$, where $\mathcal{W}$ is a finite set of label variables from $\mathcal{V}_{\text{lab}}$, and $\delta \subseteq Q \times \text{REG}(\Sigma \cup \mathcal{W}) \times Q$.*

Thus, an incomplete automaton is really just a graph pattern from $\boldsymbol{\mathcal{P}}^{\text{nv,lv,re}}$ with a distinguished node corresponding to the initial state, and a set of nodes corresponding to the final states.

To define acceptance by these automata, we need the notion of *valuation*. For an incomplete automaton $\mathcal{A} = (Q, \Sigma, \mathcal{W}, q_0, F, \delta)$, a valuation is a pair $\nu = (\eta, \theta)$, where $\eta : \mathcal{W} \to \Sigma$ maps label variables in $\mathcal{W}$ to Σ, and $\theta : (Q \times \text{REG}(\Sigma \cup \mathcal{W}) \times Q) \to (Q \times \Sigma^* \times Q)$ assigns to each transition $(q, L, q') \in \delta$ a transition (q, w, q'), where w is a word that belongs to $\eta(L)$. Thus, a valuation $\nu = (\eta, \theta)$ for an incomplete automaton $\mathcal{A}$ defines an NFA $\nu(\mathcal{A}) = (Q, \Sigma, q_0, F, \theta(\delta))$.

We now consider two notions of acceptance. *Weak acceptance* refers to the consistency of a *language* with the automaton, regardless of the interpretation of variables, and *strong acceptance* refers to the consistency of a *word* and an automaton.

Definition 3.
- *A regular language $L \subseteq \Sigma^*$ is* weakly accepted *by an incomplete automaton $\mathcal{A}$ if $L \cap L(\nu(A)) \neq \emptyset$ for every valuation ν.*
- *A word $w \in \Sigma^*$ is* strongly accepted *by an incomplete automaton $\mathcal{A}$ if $w \in L(\nu(A))$ for every valuation ν.*

We write $\mathcal{L}_w(\mathcal{A})$ for the set of languages weakly accepted by $\mathcal{A}$, and $L_s(\mathcal{A})$ for the set of words strongly accepted by $\mathcal{A}$. Note that $\mathcal{L}_w(\mathcal{A}) \subseteq 2^{\Sigma^*}$ while $L_s(\mathcal{A}) \subseteq \Sigma^*$.

While not immediately obvious from the definition, we can show the following.

Proposition 5. *For an incomplete automaton $\mathcal{A}$, the language $L_s(\mathcal{A})$ of words strongly accepted by $\mathcal{A}$ is regular. An NFA accepting $L_s(\mathcal{A})$ can be constructed in doubly exponential time.*

We shall see later (Theorem 8) that the bound is tight.

Given a graph pattern $\pi = (N, E) \in \boldsymbol{\mathcal{P}}^{\text{nv,lv,re}}$ over Σ that uses label variables $\mathcal{W}$, and two nodes n_1, n_2 from $\mathcal{N} \cap N$ (i.e., nodes which are not variables), we let $\mathcal{A}_\pi(n_1, n_2)$ be the incomplete automaton $(N, \Sigma, \mathcal{W}, n_1, \{n_2\}, E)$.

Theorem 6. *Let $Ans(x, y, \rho) \leftarrow (x, \rho : L, y)$ be an RPQ, $\pi = (N, E)$ a graph pattern, and n_1, n_2 two of its nodes from $\mathcal{N}$. Then*

1. *$(n_1, n_2) \in \text{certain}(Q, \pi)$ iff L is weakly accepted by $\mathcal{A}_\pi(n_1, n_2)$.*
2. *$w \in \text{certain}^{\text{path}}(Q; \pi, n_1, n_2)$ iff $w \in L$ and w is strongly accepted by $\mathcal{A}_\pi(n_1, n_2)$.*

Thus, the query evaluation problem, for both nodes and paths, can be stated in purely automata-theoretic terms. In particular, the set $\text{certain}^{\text{path}}(Q; \pi, n_1, n_2)$ is regular for every RPQ. Thus, our next goal is to study properties of incomplete automata.

6.2 Computational problems for incomplete automata

Theorem 6 suggests studying computational problems for incomplete automata related to query evaluation. Results for weak acceptance have, in essence, been established earlier, so we are interested in strong acceptance, which accounts for having paths in the output.

For weak acceptance, *membership* (i.e., given incomplete automaton $\mathcal{A}$ and a regular language L, presented as a regular expression or as an NFA, does L belong to $\mathcal{L}_w(\mathcal{A})$?) is the problem of finding certain answers to RPQs. Hence, we have

Corollary 1. *The membership problem for incomplete automata under weak acceptance is* PSPACE*-complete, and* coNP*-complete if the language L is fixed.*

It can also be easily seen that the emptiness problem under weak acceptance, i.e., whether $\mathcal{L}_w(\mathcal{A}) \neq \emptyset$, is solvable in polynomial time.

Now we address the case of strong acceptance, which, by Theorem 6, gives us complexity bounds for computing

paths that are returned with certainty. There are three versions of the problem we consider:

- Checking whether the query output is not empty. In automata-theoretic terms, this is the *emptiness problem under strong acceptance*: given an incomplete automaton $\mathcal{A}$, check whether $L_s(\mathcal{A}) \neq \emptyset$.
- Checking whether a specific path belongs to the output, i.e., whether $w \in \text{certain}^{\text{path}}(Q; \pi, n_1, n_2)$. In automata-theoretic terms, we are interested in the *membership problem under strong acceptance*, i.e., given an incomplete automaton $\mathcal{A}$ and a word w, check whether $w \in L_s(\mathcal{A})$.
- Computing $\text{certain}^{\text{path}}(Q; \pi, n_1, n_2)$. As this set is regular, in automata-theoretic terms, we study the following problem: For an incomplete automaton $\mathcal{A}$, construct an NFA $\mathcal{A}'$ so that $L(\mathcal{A}') = L_s(\mathcal{A})$.

As we analyze these problems, we shall see that hardness results will be witnessed by an especially simple kind of incomplete automata: namely, *wildcard automata*, in which all regular languages used in transitions are single letters (alphabet letters or variables). Formally, a wildcard automaton $\mathcal{A}$ is $(Q, \Sigma, \mathcal{W}, q_0, F, \delta)$, where $\delta \subseteq Q \times (\Sigma \cup \mathcal{W}) \times Q$.

We now show that problems related to computing certain paths are computationally hard as long as regular expressions or label variables are present in the edges. The following does not appear to follow from known EXPSPACE-completeness results for graph databases [7, 5], and requires a new and quite involved proof.

Theorem 7. *The emptiness problem under strong acceptance is* EXPSPACE*-complete. It remains* EXPSPACE*-hard for wildcard automata, as well as for incomplete automata that do not use any label variables.*

We now consider problems related to query answering. The first is finding certain paths, or, in automata-theoretic terms, the membership problem under strong acceptance.

Proposition 6. *The membership problem under strong acceptance for incomplete automata is* coNP*-complete. It remains* coNP*-hard for wildcard automata and for incomplete automata that do not use any label variables.*

The next question is about the size of automata defining $L_s(\mathcal{A})$. Normally large size bounds are easy to obtain for deterministic automata, while NFAs could be exponentially smaller. Here we use techniques from [23] to show that even the smallest NFAs capturing certain paths in the answer to an RPQ could be doubly exponential, matching the upper bound of Proposition 5.

Theorem 8. *There exists a polynomial p and a family $\{\mathcal{A}_n\}_{n\in\mathbb{N}}$ of wildcard automata such each $\mathcal{A}_n$ is of size at most $p(n)$ and uses n wildcards, and every NFA $\mathcal{A}'_n$ satisfying $L(\mathcal{A}'_n) = L_s(\mathcal{A}_n)$ has $2^{2^{\Omega(n)}}$ states.*

There also exists a family of incomplete automata without label variables with the same property.

This gives a lower bound on the size of automata for representing certain paths in answers to RPQs.

Corollary 2. *There exists a polynomial p, a family $\{\pi_n\}_{n\in\mathbb{N}}$ of $\boldsymbol{\mathcal{P}}^{\text{lv}}$ graph patterns, each with two distinguished nodes n_1 and n_2, and an RPQ Q such that the size of π_n is at most $p(n)$, and every NFA defining $\text{certain}^{\text{path}}(Q; \pi_n, n_1, n_2)$ has $2^{2^{\Omega(n)}}$ states.*
The same holds for $\boldsymbol{\mathcal{P}}^{\text{re}}$ patterns.

Note there is an exponential gap between the complexity of the membership problem and the size of a representation of all words strongly accepted by an incomplete automaton. There is no contradiction, of course, between Theorem 8 and Proposition 6 as smallest NFAs accepting even finite languages L can be of size exponential in the maximum length of a word in L.

Remark: Theorems 7 and 8, Proposition 6, and Corollary 2 remain true even for the Codd interpretation of patterns and wildcard automata (i.e., each label variable is used in at most one transition).

7. TRACTABILITY RESTRICTIONS AND HEURISTICS

While many results of Sections 5 and 6 point to a rather high complexity of query answering, they still leave a few routes for finding tractable classes, or providing heuristics that – at least based on the experience of other areas – may be useful.

If we look at data complexity, results of Subsection 5.2 show that one possibility of getting tractable cases is to impose further restrictions on underlying graphs of patterns. Being DAGs, as we saw, is not enough, which suggests trees. We shall in fact get a more general result, replacing trees with graphs of bounded treewidth.

Combined complexity results in Subsection 5.1 point to $\boldsymbol{\mathcal{P}}^{\text{nv,lv}}$ as the largest class with acceptable combined complexity (i.e., not exceeding that of FO; in fact staying in the 2nd level of the polynomial hierarchy). The data complexity for the class, although intractable, drops to the 1st level of the polynomial hierarchy. This suggests using techniques from a field that has achieved great success in solving problems of this complexity, namely constraint satisfaction [16, 30]. The field has identified many tractable restrictions and, what is equally important, provided many practical heuristics that help solve intractable problems. The connection between RPQs on graph databases and constraint satisfaction was already established in [9]. As the second contribution of this section, we show how to cast the query answering problem for RPQs over graph patterns as a constraint satisfaction problem, with a particularly simple translation for several classes.

7.1 Tractability restrictions

Recall the standard definition of tree decompositions and treewidth of a graph $G = (N, E)$, with $E \subseteq N \times N$ (see, e.g., [18]). A tree decomposition is a pair (T, f) where T is a tree and $f : T \to 2^N$ assigns to each node t in T a set of nodes $f(t)$ of G such that every edge of G is contained in one of the sets $f(t)$, and each set $\{t \mid n \in f(t)\}$ is a connected subset of T for all $n \in N$. The width of such a decomposition is $\max_t |f(t)| - 1$. The *treewidth* of G is the minimum width of a tree decomposition of G. The treewidth of a connected graph G equals 1 iff G is a tree.

A class of graph patterns is of *bounded treewidth* if there is a fixed $k \in \mathbb{N}$ so that for every pattern π in the class, the treewidth of its underlying graph G_π is at most k.

We saw that label variables and regular expressions lead to intractable data complexity of query answering. We now show that bounded treewidth guarantees tractability for large classes of patterns with these features.

Theorem 9. *The data complexity of finding certain answers to CRPQs over classes of graph patterns of bounded treewidth in* $\boldsymbol{\mathcal{P}}^{\mathrm{nv,re}}$ *and* $\boldsymbol{\mathcal{P}}^{\mathrm{nv,lv}}_{\mathrm{Codd}}$ *is in* PTIME.

The Codd interpretation of label variables is essential, since without it the problem is already coNP-hard for treewidth 1 (see Theorem 4). For $\boldsymbol{\mathcal{P}}^{\mathrm{re}}$ patterns, coNP-hardness results of Theorem 5 used classes of DAGs of unbounded treewidth.

7.2 Certain answers via constraint satisfaction

We now demonstrate the potential of using techniques from constraint satisfaction for answering queries over graph patterns, in the spirit of [9]. We shall consider patterns in $\boldsymbol{\mathcal{P}}^{\mathrm{nv,lv}}$, for which data complexity is in coNP. Of course pure complexity-theoretic argument tells us that (the complement of) query answering can be cast as a constraint satisfaction problem; what we show here is that the translation for RPQs is very transparent, opening up the possibility of bringing the huge arsenal of tools from constraint satisfaction [16].

We adopt the standard view of the constraint satisfaction problem (CSP) as checking for the existence of a homomorphism from a relational structure $\mathcal{M}_1$ to another structure $\mathcal{M}_2$ of the same vocabulary [30], referring to this problem as $\mathrm{CSP}(\mathcal{M}_1, \mathcal{M}_2)$. Often this problem is considered with $\mathcal{M}_2$ fixed, in which case one refers to non-uniform CSP.

Consider a pattern $\pi = (N, E)$ in $\boldsymbol{\mathcal{P}}^{\mathrm{nv,lv}}$, i.e., $E \subseteq N \times (\Sigma \cup \mathcal{W}) \times N$ for a finite set $\mathcal{W}$ of label variables. Let Q be an RPQ given by $Ans(x, y) \leftarrow (x, L, y)$, where $L \subseteq \Sigma^*$ is a regular language. We now define logical structures $\mathcal{M}_\pi(n, n')$ and $\mathcal{M}_Q$ over vocabulary

$$(Nodes, Expr, (Lab_a)_{a\in\Sigma}, Src, Sink, Edge),$$

where $Edge$ is a ternary relation and other relations are unary. Here n and n' are two node ids of π.

Structure $\mathcal{M}_\pi(n, n')$ The domain is the disjoint union of N, Σ, and $\mathcal{W}$, the set of label variables used in π. The interpretation of the predicates is as follows:

$$\begin{array}{rclrcl} Nodes & := & N & Edge & := & E \\ Lab_a & := & \{a\} & Src & := & \{n\} \\ Expr & := & \mathcal{W} & Sink & := & \{n'\} \end{array}$$

Structure $\mathcal{M}_Q$ Assume that L is recognized by an NFA $(S, \Sigma, q_0, F, \delta)$ with $\delta : S \times \Sigma \to 2^S$ (extended, as usual, to a transition function on sets $\delta(S', a) = \bigcup_{s\in S'} \delta(s, a)$). The domain of $\mathcal{M}_Q$ is the disjoint union of 2^S and Σ. The interpretation of the predicates is:

$$\begin{array}{rl} Nodes := 2^S & Edge := \{(S', a, S'') \in 2^S \times \Sigma \times 2^S \mid \\ & \qquad \delta(S', a) \subseteq S''\} \\ Lab_a := \{a\} & Src := \{S' \in 2^S \mid q_0 \in S'\} \\ Expr := \Sigma & Sink := 2^{S-F} \end{array}$$

Theorem 10. *For patterns* $\pi \in \boldsymbol{\mathcal{P}}^{\mathrm{nv,lv}}$*, under the above translations,* $(n, n') \in \mathrm{certain}(Q, \pi)$ *iff there is no solution to* $\mathrm{CSP}(\mathcal{M}_\pi(n, n'), \mathcal{M}_Q)$.

Many algorithmic techniques for constraint satisfaction for $\mathrm{CSP}(\mathcal{M}_1, \mathcal{M}_2)$ are based on exploiting properties of the structure $\mathcal{M}_1$, so the extremely simple construction of $\mathcal{M}_\pi(n, n')$ indeed opens up the possibility of using a large body of heuristics developed in that area.

The case of data complexity corresponds to the non-uniform version of CSP, with $\mathcal{M}_Q$ fixed. In that case one can immediately conclude (using known results on CSP [16, 30]) that if we have a class of patterns $\pi \in \boldsymbol{\mathcal{P}}^{\mathrm{nv,lv}}$ which, when viewed as ternary relations E, has bounded treewidth, then the data complexity of RPQs over such a class is in PTIME (note that this is incompatible with Theorem 9 which gives a PTIME result for a larger class of queries, but under the restriction of the Codd interpretation of label variables).

An analog of Theorem 10 for patterns in $\boldsymbol{\mathcal{P}}^{\mathrm{nv,re}}$ was shown in [9]. Combining both techniques we can extend the result to all patterns in $\boldsymbol{\mathcal{P}}^{\mathrm{nv,lv,re}}$, but at the cost of much more complex definitions of the structures $\mathcal{M}_\pi(n, n')$ and $\mathcal{M}_Q$ compared to those we used here. We shall discuss this in the full version of the paper.

8. CONCLUSIONS

We studied structural properties and querying of graph patters. We looked at three main features of patterns: node variables, label variables, and regular expressions specifying paths. We showed that each of these features strictly increases the expressiveness of patters. We looked at data and combined complexity of answering CRPQs and other queries (both extensions and restrictions of CRPQs). We developed a model of automata

that capture query answering, both for returning nodes and paths, and studied their properties. Finally, we identified tractable restrictions, as well as classes of reasonable combined complexity for which query answering is naturally viewed as a constraint satisfaction property.

The main conclusion is that, without carefully chosen restrictions, querying graph patterns is computationally harder than querying relational or XML patterns. In particular, this has implications for ongoing work on defining schema mappings as well as integration and exchange techniques for graph-structured data. However, we can identify rather robust classes with either tractable query answering, or for which one can hope to find good heuristics by using techniques from other fields. Developing such techniques is a natural continuation of this work. Another line for further work is to study tractable restrictions for integrating and exchanging graph data.

Acknowledgments We thank Wenfei Fan and Peter Wood for their comments. Partial support provided by Fondecyt grant 1110171, EPSRC grant G049165 and FET-Open Project FoX, grant agreement 233599. Part of this work was done when the first author visited Edinburgh, and the second and the third author visited Santiago.

9. References

[1] S. Abiteboul, P. Buneman, D. Suciu. *Data on the Web: From Relations to Semistructured Data and XML*. Morgan Kauffman, 1999.

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

[3] M. Arenas, P. Barceló, L. Libkin, F. Murlak. *Relational and XML Data Exchange*. Morgan & Claypool, 2010.

[4] P. Barceló, L. Libkin, A. Poggi, C. Sirangelo. XML with incomplete information. *J. ACM* 58(1): 1–62 (2010).

[5] P. Barceló, C. Hurtado, L. Libkin, P. Wood. Expressive languages for path queries over graph-structured data. In *PODS*, pages 3-14, 2010.

[6] H. Björklund, W. Martens, and T. Schwentick. Conjunctive query containment over trees. In *DBPL'07*, pages 66–80.

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

[8] D. Calvanese, G. de Giacomo, M. Lenzerini, M. Y. Vardi. Answering regular path queries using views. In *ICDE*, pages 389-398, 2000.

[9] D. Calvanese, G. de Giacomo, M. Lenzerini, M. Y. Vardi. View-based query processing and constraint satisfaction. In *LICS*, pages 361-371, 2000.

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

[11] D. Calvanese, G. de Giacomo, M. Lenzerini, M. Y. Vardi. Simplifying schema mappings. In *ICDT 2011*, to appear.

[12] J. Cheng, J. X. Yu, B. Ding, P. S. Yu, and H. Wang. Fast graph pattern matching. In *ICDE 2008*, pages 913–922.

[13] S. Cohen and Y. Sagiv. An abstract framework for generating maximal answers to queries. In *ICDT 2005*, pages 129–143.

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

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

[16] R. Dechter. *Constraint Processing*. Morgan Kaufmann, 2003.

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

[18] R. Diestel. *Graph Theory*. Springer, 2005.

[19] R. Fagin, Ph. Kolaitis, R. Miller, and L. Popa. Data exchange: semantics and query answering. *TCS*, 336(1):89–124, 2005.

[20] W. Fan, J. Li, S. Ma, H. Wang, Y. Wu. Homomorphism revisited for graph matching. *PVLDB* 3(1): 1161-1172 (2010).

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

[22] W. Fan, J. Li, S. Ma, N. Tang, Y. Wu. Adding regular expressions to graph reachability and pattern queries. In *ICDE 2011*, to appear.

[23] I. Glaister, J. Shallit. A lower bound technique for the size of nondeterministic finite automata. *IPL* 59:75-77, 1996.

[24] G. Gottlob, C. Koch, K. Schulz. Conjunctive queries over trees. *J. ACM* 53(2) (2006), 238-272.

[25] C. Gutierrez, C. Hurtado, A. Mendelzon. Foundations of semantic web databases. In *PODS 2004*, pages 95–106.

[26] M. Gyssens, J. Paredaens, J. Van den Bussche, D. Van Gucht. A graph-oriented object database model. *IEEE TKDE* 6(4) (1994), 572–586.

[27] T. Imielinski, W. Lipski. Incomplete information in relational databases. *J. ACM* 31 (1984), 761–791.

[28] D. Johnson, A. Klug. Testing containment of conjunctive queries under functional and inclusion dependencies. *JCSS*, 28(1) (1984), pages 167-189.

[29] Y. Kanza, W. Nutt, Y. Sagiv. Querying incomplete information in semistructured data. *JCSS* 64 (3) (2002), 655–693.

[30] P. Kolaitis and M. Vardi. A logical approach to constraint satisfaction. In *Finite Model Theory and Its Applications*, Springer 2007, pages 339–370.

[31] M. Lenzerini. Data integration: a theoretical perspective. In *PODS'02*, pages 233–246.

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

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

[34] M. Natarajan. Understanding the structure of a drug trafficking organization: a conversational analysis. *Crime Prevention Studies* 11 (2000), 273–298.

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

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

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

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

[39] H. Tong, C. Faloutsos, B. Gallagher, and T. Eliassi-Rad. Fast best-effort pattern matching in large attributed graphs. In *KDD 2007*.

[40] G. Weikum, G. Kasneci, M. Ramanath, F. Suchanek. Database and information-retrieval methods for knowledge discovery. *CACM* 52(4):56-64 (2009).