Learning Path Queries on Graph Databases

Angela Bonifati, Radu Ciucanu, Aurélien Lemay

Learning Path Queries on Graph Databases

Angela Bonifati Radu Ciucanu Aurélien Lemay
University of Lille & INRIA, France
{angela.bonifati, radu.ciucanu, aurelien.lemay}@inria.fr

ABSTRACT

We investigate the problem of learning graph queries by exploiting user examples. The input consists of a graph database in which the user has labeled a few nodes as *positive* or *negative examples*, depending on whether or not she would like the nodes as part of the query result. Our goal is to handle such examples to find a query whose output is what the user expects. This kind of scenario is pivotal in several application settings where unfamiliar users need to be assisted to specify their queries. In this paper, we focus on *path queries* defined by *regular expressions*, we identify fundamental difficulties of our problem setting, we formalize what it means to be *learnable*, and we prove that the class of queries under study enjoys this property. We additionally investigate an interactive scenario where we start with an empty set of examples and we identify the *informative nodes* i.e., those that contribute to the learning process. Then, we ask the user to label these nodes and iterate the learning process until she is satisfied with the learned query. Finally, we present an experimental study on both real and synthetic datasets devoted to gauging the effectiveness of our learning algorithm and the improvement of the interactive approach.

1. INTRODUCTION

Graph databases [41] are becoming pervasive in several application scenarios such as the Semantic Web [5], social [37] and biological [36] networks, and geographical databases [2], to name a few. A graph database is essentially a directed, edge-labeled graph. As an example, consider in Figure 1 a graph representing a geographical database having as nodes the neighborhoods of a city area (N_1 to N_6), along with cinemas (C_1 and C_2), and restaurants (R_1 and R_2) in such neighborhoods. The edges represent public transportation facilities from a neighborhood to another (using labels *tram* and *bus*), along with other kind of facilities (using labels *cinema* and *restaurant*). For instance, the graph indicates that one can travel by bus between the neighborhoods N_2 and N_3, that in the neighborhood N_4 exists a cinema C_1, and so on.

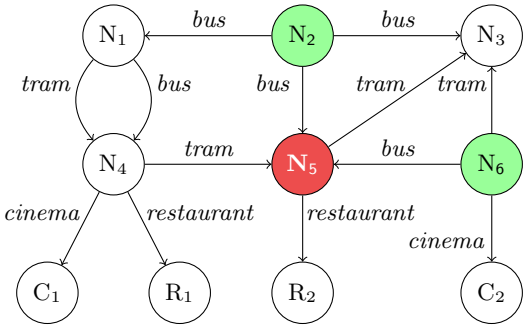


Figure 1: A geographical graph database.

Many mechanisms have been proposed to query a graph database, the majority of them being based on *regular expressions* [7, 41]. By continuing on our running example, imagine that a user wants to know from which neighborhoods in the city represented in Figure 1 she can reach cinemas via public transportation. These neighborhoods can be retrieved using a *path query* defined by the following *regular expression*:

$$q = (tram + bus)^* \cdot cinema$$

The query q selects the nodes N_1, N_2, N_4, and N_6 as they are entailed by the following *paths* in the graph:

$N_1 \xrightarrow{tram} N_4 \xrightarrow{cinema} C_1$,
$N_2 \xrightarrow{bus} N_1 \xrightarrow{tram} N_4 \xrightarrow{cinema} C_1$,
$N_4 \xrightarrow{cinema} C_1$,
$N_6 \xrightarrow{cinema} C_2$.

Although very expressive, graph query languages are difficult to understand by non-expert users who are unable to specify their queries with a formal syntax. The problem of *assisting non-expert users to specify their queries* has been recently raised by Jagadish et al. [24, 34]. More concretely, they have observed that "constructing a database query is often challenging for the user, commonly takes longer than the execution of the query itself, and does not use any insights from the database". While they have mentioned these problems in the context of relational databases, we argue that they become even more difficult to tackle for graph databases. Indeed, graph databases usually do not carry proper metadata as they lack schemas and/or do not ex-

hibit a clear distinction between instances and schemas. The absence of metadata along with the difficulty of visualizing possibly large graphs make unfeasible traditional query specification paradigms for non-expert users, such as query by example [43]. Our work follows the recent trend of specifying graph queries by example [33, 25]. Precisely, we focus on graph queries using regular expressions, which are fundamental building blocks of graph query languages [7, 41], while both [33, 25] consider simple graph patterns.

While the problem of executing path queries defined by regular expressions on graphs has been extensively studied recently [6, 32, 27], no research has been done on how to actually specify such queries. Our work focuses on the problem of assisting non-expert users to specify such path queries, by exploiting elementary user input.

By continuing on our running example, we assume that the user is not familiar with any formal syntax of query languages, while she still wants to specify the above query q on the graph database in Figure 1 by providing examples of the query result. In particular, she would positively or negatively label some graph nodes according to whether or not they would be selected by the targeted query. Thus, let us imagine that the user labels the nodes N_2 and N_6 as *positive examples* because she wants these nodes as part of the result. Indeed, one can reach cinemas from N_2 and N_6, respectively, through the following paths:

$$N_2 \xrightarrow{bus} N_1 \xrightarrow{tram} N_4 \xrightarrow{cinema} C_1,$$
$$N_6 \xrightarrow{cinema} C_2.$$

Similarly, the user labels the node N_5 as a *negative example* since she would not like it as part of the query result. Indeed, there is no path starting in N_5 through which the user can reach a cinema. We also observe that the query q above is *consistent* with the user's examples because q selects all positive examples and none of the negative ones. Unfortunately, there may exist an infinite number of queries consistent with the given examples. Therefore, we are interested to find either the "exact" query that the user has in mind or, alternatively, an equivalent query, which is close enough to the user's expectations.

Apart from assisting unfamiliar users to specify queries, our research has other crucial applications, such as mining *scientific workflows*. Regular expressions have already been used in the literature as a well-suited mechanism for inter-workflow coordination [21]. The path queries on graph databases that we study in this paper can be applied to assist scientists in identifying interrelated workflows that are of interest for them. For instance, assume that a biologist is interested in retrieving all interrelated workflows having a pattern that starts with protein purification, continues with an arbitrary number of protein separation steps, and ends with mass spectrometry. This corresponds to the following regular expression:

$$ProteinPurification \cdot ProteinSeparation^* \cdot MassSpectrometry.$$

Instead of specifying such a pattern in a formal language, the biologist may be willing to label some sequences of modules from a set of available workflows as positive or negative examples, as illustrated in Figure 2. Our algorithms can be thus applied to infer the workflow pattern that the biologist has in mind. Typically in graphs representing workflows the labels are attached to the nodes (e.g., as in Figure 2) instead of the edges. In this paper, we have opted for

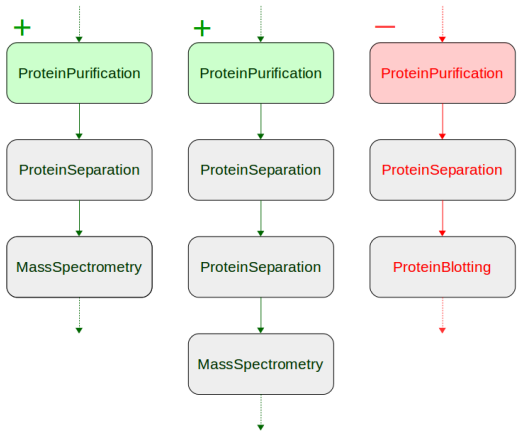


Figure 2: A set of scientific workflows examples.

edge-labeled graphs rather than node-labeled graphs, but our algorithms and learning techniques for the considered class of queries are applicable to the latter class of graphs in a seamless fashion. The problem of mining scientific workflows has been considered in recent work [8], which leverages data instances as representatives of the input and output of a workflow module. In our approach, we rely on simpler user feedback, namely Boolean labeling of sequences of modules across interrelated workflows.

Since our goal is to infer the user queries while minimizing the amount of user feedback, our research is also applicable to *crowdsourcing* scenarios [19], in which such minimization typically entails lower financial costs. Indeed, we can imagine that crowdworkers provide the set of positive and negative examples mentioned above for path query learning. Moreover, our work can be used in assisting non-expert users in other fairly complex tasks, such as specifying *schema mappings* [42] i.e., logical assertions between two path queries, one on a source schema and another on a target schema.

To the best of our knowledge, our work is the first to study the problem of learning path queries defined by regular expressions on graphs via user examples. More precisely, we make the following main contributions:

- We investigate a learning framework inspired by computational learning theory [26], in particular by grammatical inference [18] and we identify fundamental difficulties of such a framework. We consequently propose a definition of learnability adapted to our setting.
- We propose a learning algorithm and we precisely characterize the conditions that a graph must satisfy to guarantee that every user's goal query can be learned. Essentially, the main theoretical result of the paper states that for every query q there exists a polynomial set of examples that given as input to our learning algorithm guarantees the learnability of q. Additionally, our learning algorithm is guaranteed to run in polynomial time, whether or not the aforementioned set of examples is given as input.
- We investigate an interactive scenario, which bootstraps with an empty set of examples and builds it along the way. Indeed, the learning algorithm finely

interacts with the user by proposing nodes that can be labeled and repeats the interactions until the goal query is learned. More precisely, we analyze what it means for a node to be informative for the learning process, we show the intractability of deciding whether a node is informative or not, and we propose efficient strategies to present examples to the user.

- To evaluate our approach, we have run experiments on both real-world and synthetic datasets. Our study shows the effectiveness of the learning algorithm and the advantage of using an interactive strategy, which significantly reduces the number of examples needed to learn the goal query.

Finally, we would like to spend a few words on the class of queries that we investigate in this paper. As already mentioned, we focus on regular expressions, which are fundamental for graph query languages [7, 41] and lately used in the definition of SPARQL property paths[1]. Graph queries defined by regular expressions have been known as *regular path queries*. Intuitively, such queries retrieve pairs of nodes in the graph s.t. one can navigate between them with a path in the language of a given regular expression [7, 41]. Although the usual semantics of regular path queries is *binary* (i.e., selects pairs of nodes), in this paper we consider a generalization of this semantics that we call *monadic*, as it outputs only the originated nodes of the paths. The motivation behind using a monadic semantics is essentially threefold. First, it entails a larger space of potential solutions than a binary semantics. Indeed, with the latter semantics the end node of a path is fixed, which basically corresponds to have a smaller number of candidate paths that start at the originated node and that can be possibly labeled by the user. Second, in our learning framework, the amount of user effort should be kept as minimal as possible (which led to design an interactive scenario) and thus we let the user focus solely on the originated nodes of the paths rather than on pairs of nodes. Third, the development of the learning algorithm for monadic queries is extensible to binary queries and n-ary queries in a straightforward fashion, as shown in the paper.

Organization. In Section 2, we introduce some basic notions. In Section 3, we define our framework for learning from a set of examples, we present our learning algorithm, and we prove our learnability results. In Section 4, we propose an interactive algorithm and characterize the quantity of information of a node. In Section 5, we experimentally evaluate the performance of our algorithms. Finally, we conclude our paper and outline future directions in Section 6. Due to space restrictions, in this paper we omit the proofs of several results and we refer to the appendix of our technical report [11] for detailed proofs.

Related work

Learning queries from examples is a popular and interesting topic in databases. Very recently, algorithms for learning relational queries (e.g., quantifiers [1], joins [14, 15]) or XML queries (e.g., tree patterns [38]) have been proposed. Besides learning queries, researchers have investigated the learnability of relational schema mappings [40], as well as schemas [9] and transformations [30] for XML. A fairly close problem to learning is definability [4]. In this paragraph, we discuss the positioning of our own work w.r.t. these and other papers.

A wealth of research on using computational learning theory [26] has been recently conducted in databases [1, 9, 14, 30, 38, 40]. In this paper, we use *grammatical inference* [18] i.e., the branch of machine learning that aims at constructing a formal grammar by generalizing a set of examples. In particular, all the above papers on learning tree patterns [38], schemas [9, 16], and transformations [30] are based on it.

Our definition of learnability is inspired by the well-known framework of *language identification in the limit* [20], which requires a learning algorithm to be polynomial in the size of the input, *sound* (i.e., always return a concept consistent with the examples given by the user or a special *null* value if such concept does not exist) and *complete* (i.e., able to produce every concept with a sufficiently rich set of examples). In our case, we show that checking the consistency of a given set of examples is intractable, which implies that there is no algorithm able to answer *null* in polynomial time when the sample is inconsistent. This leads us to the conclusion that path queries are not learnable in the classical framework. Consequently, we slightly modify the framework and require the algorithm to learn the goal query in polynomial time if a polynomially-sized characteristic set of examples is provided. This learning framework has been recently employed for learning XML transformations [30] and is referred to as learning with *abstain* since the algorithm can abstain from answering when the characteristic set of examples is not provided.

The classical algorithm for learning a regular language from positive and negative word examples is RPNI [35], which basically works as follows: (i) construct a DFA (usually called prefix tree acceptor or PTA [18]) that selects all positive examples; (ii) generalize it by state merges while no negative example is covered. Unfortunately, RPNI cannot be directly adapted to our setting since the input positive and negative examples are not words in our case. Instead, we have a set of graph nodes starting from which we have to select (from a potentially infinite set) the paths that led the user to label them as examples. After selecting such paths, we generalize by state merges, similarly to RPNI.

A problem closely related to learning is *definability*, recently studied for graph databases [4]. Learning and definability have in common the fact that they look for a query consistent with a set of examples. The difference is that learning allows the query to select or not the nodes that are not explicitly labeled as positive examples while definability requires the query to select nothing else than the set of positive examples (i.e., all other nodes are implicitly negative). Nonetheless, some of the intractability proofs for definability can be adapted to our learning framework to show the intractability of consistency checking (cf. Section 3). To date, no polynomial algorithms have been yet proposed to construct path queries from a consistent set of examples.

2. GRAPH DATABASES AND QUERIES

In this section we define the concepts that we manipulate throughout the paper.

Alphabet and words. An *alphabet* Σ is a finite, ordered set of symbols. A *word* over Σ is a sequence $a_1 \dots a_n$ of symbols from Σ. By $|w|$ we denote the *length* of a word w. The *concatenation* of two words $w_1 = a_1 \dots a_n$ and $w_2 =$

[1] `http://www.w3.org/TR/sparql11-query/`

$b_1 \dots b_m$, denoted $w_1 \cdot w_2$, is the word $a_1 \dots a_n b_1 \dots b_m$. By ε we denote the *empty word*. A *language* is a set of words. By Σ^* we denote the language of all words over Σ. We extend the order on Σ to the standard lexicographical order $\leqslant_{lex}$ on words over Σ and define a well-founded *canonical order* $\leqslant$ on words: $w \leqslant u$ iff $|w| < |u|$ or $|w| = |u|$ and $w \leqslant_{lex} u$.

Graph databases. A graph database is a finite, directed, edge-labeled graph [7, 41]. Formally, a *graph* (*database*) G over an alphabet Σ is a pair (V, E), where V is a set of *nodes* and $E \subseteq V \times \Sigma \times V$ is a set of *edges*. Each edge in G is a triple $(\nu_o, a, \nu_e) \in V \times \Sigma \times V$, where ν_o is the *origin* of the edge, ν_e is the *end* of the edge, and a is the *label* of the edge. We often abuse notation and write $\nu \in G$ and $(\nu_o, a, \nu_e) \in G$ instead of $\nu \in V$ and $(\nu_o, a, \nu_e) \in E$, respectively. For example, take in Figure 3 the graph G_0 containing 7 nodes and 15 edges over the alphabet $\{a, b, c\}$.

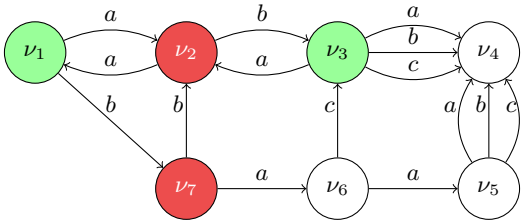


Figure 3: A graph database G_0.

Paths. A word $w = a_1 \dots a_n$ *matches* a sequence of nodes $\nu_0 \nu_1 \dots \nu_n$ if, for each $1 \leqslant i \leqslant n$, the triple (ν_{i-1}, a_i, ν_i) is an edge in G. For example, for the graph G_0 in Figure 3, the word aba matches the sequences of nodes $\nu_1\nu_2\nu_3\nu_4$ and $\nu_3\nu_2\nu_3\nu_4$, respectively, but does not match the sequence $\nu_1\nu_2\nu_7\nu_2$. Note that the empty word ε matches the sequence $\nu\nu$ for every $\nu \in G$. Given a node $\nu \in G$, by $paths_G(\nu)$ we denote the language of all words that match a sequence of nodes from G that starts by ν. In the sequel, we refer to such words as *paths*, and moreover, we say that a path w is *covered* by a node ν if $w \in paths_G(\nu)$. Paths are ordered using the canonical order $\leqslant$. For example, for the graph G_0 in Figure 3 we have $paths_{G_0}(\nu_5) = \{\varepsilon, a, b, c\}$. Note that $\varepsilon \in paths_G(\nu)$ for every $\nu \in G$. Moreover, note that $paths_G(\nu)$ is finite iff there is no cycle reachable from ν in G. For example, for the graph G_0 in Figure 3, $paths_{G_0}(\nu_1)$ is infinite. We naturally extend the notion of paths to a set of nodes i.e., given a set of nodes X from a graph G, by $paths_G(X) = \bigcup_{\nu \in X} paths_G(\nu)$.

Regular expressions and automata. A *regular language* is a language defined by a *regular expression* i.e., an expression of the following grammar:

$$q := \varepsilon \mid a \quad (a \in \Sigma) \mid q_1 + q_2 \mid q_1 \cdot q_2 \mid q^*,$$

where by "$\cdot$" we denote the *concatenation*, by "$+$" we denote the *disjunction*, and by "$*$" we denote the *Kleene star*. By $L(q)$ we denote the *language* of q, defined in the natural way [22]. For instance, the language of $(a \cdot b)^* \cdot c$ contains words like $c, abc, ababc$, etc. Regular languages can alternatively be represented by automata and we also refer to [22] for standard definitions of *nondeterministic finite word automaton* (*NFA*) and *deterministic finite word automaton* (*DFA*). In particular, we represent every regular language by its *canonical DFA* that is the unique smallest DFA that describe the language. For example, we present in Figure 4 the canonical DFA for $(a \cdot b)^* c$.

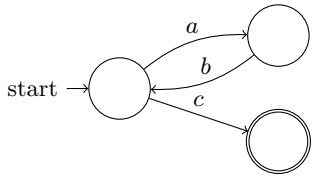


Figure 4: Canonical DFA for $(a \cdot b)^* c$.

Path queries. We focus on the class of path queries defined by regular expressions i.e., that select nodes having at least one *path* in the language of a given regular expression. Formally, given a graph G and a query q, we define the set of nodes *selected* by q on G:

$$q(G) = \{\nu \in G \mid L(q) \cap paths_G(\nu) \neq \emptyset\}.$$

For example, given the graph G_0 in Figure 3, the query a selects all nodes except ν_4, the query $(a \cdot b)^* \cdot c$ selects the nodes ν_1 and ν_3, and the query $b \cdot b \cdot c \cdot c$ selects no node. In the rest of the paper, we denote the set of all path queries by PQ and we refer to them simply as *queries*. We represent a query by its canonical DFA, hence the *size* of a query is the number of states in the canonical DFA of the corresponding regular language. For example, the size of the query $(a \cdot b)^* \cdot c$ is 3 (cf. Figure 4).

Equivalent queries. Two queries q and q' are *equivalent* if for every graph G they select exactly the same set of nodes i.e., $q(G) = q'(G)$. For example, the queries a and $a \cdot b^*$ are equivalent since each node having a path $ab \dots b$ has also a path a. This example can be easily generalized and yields to defining the class of prefix-free queries. Formally, we say that a query q is *prefix-free* if for every word from $L(q)$, none of its prefixes belongs to $L(q)$. Given a query q, there exists a unique prefix-free query equivalent to q, which, moreover, can be constructed by simply removing all outgoing transitions of every final state in the canonical DFA of q. Our interest in prefix-free queries is that they can be seen as *minimal representatives* of *equivalence classes* of queries and as such they are desirable queries for learning. Indeed, every prefix-free query q is in fact equivalent to an infinite number of queries $q \cdot (q' + \varepsilon)$, where q' can be every PQ. In the remainder, we assume w.l.o.g. that all queries that we manipulate are prefix-free.

3. LEARNING FROM EXAMPLES

The input of a learning algorithm consists of a graph on which the user has annotated a few nodes as *positive* or *negative examples*, depending on whether or not she would like the nodes as part of the query result. Our goal is to exploit such examples to find a query that satisfies the user. In this paper, we explore two learning protocols: (i) the user provides a *sample* (i.e., a set of examples) that remains *fixed* during the learning process, and (ii) the learning algorithm

interactively asks the user to label more examples until the learned query behaves exactly as the user wants.

First, we concentrate on the case of a fixed set of examples. We identify the challenges of such an approach, we show the unfeasability of the standard framework of *language identification in the limit* [20] and slightly modify it to propose a learning framework with *abstain* (Section 3.1). Next, we present a learning algorithm for the class of PQ (Section 3.2) and we identify the conditions that a graph and a sample must satisfy to allow polynomial learning of the user's goal query (Section 3.3). We study the case of query learning from user interactions in Section 4.

3.1 Learning framework

Given a graph $G = (V, E)$, an *example* is a pair (ν, α), where $\nu \in V$ and $\alpha \in \{+, -\}$. We say that an example of the form $(\nu, +)$ is a *positive example* while an example of the form $(\nu, -)$ is a *negative example*. A *sample* S is a set of examples i.e., a subset of $V \times \{+, -\}$. Given a sample S, we denote the set of positive examples $\{\nu \in V \mid (\nu, +) \in S\}$ by S_+ and the set of negative examples $\{\nu \in V \mid (\nu, -) \in S\}$ by S_-. A sample is *consistent* (with the class of PQ) if there exists a (PQ) query that selects all positive examples and none of the negative ones. Formally, given a graph G and a sample S, we say that S is *consistent* if there exists a query q s.t. $S_+ \subseteq q(G)$ and $S_- \cap q(G) = \varnothing$. In this case we say that q is *consistent with* S. For instance, take the graph G_0 in Figure 3 and the sample S s.t. $S_+ = \{\nu_1, \nu_3\}$ and $S_- = \{\nu_2, \nu_7\}$; S is consistent because there exist queries like $(a \cdot b)^* \cdot c$ or $c + (a \cdot b \cdot c)$ that are consistent with S.

Next, we want to formalize what means for a class of queries to be *learnable*, as it is usually done in the context of *grammatical inference* [18]. The standard learning framework is *language identification the limit* (*in polynomial time and data*) [20], which requires a learning algorithm to operate in time *polynomial* in the size of its input, to be *sound* (i.e., always return a query consistent with the examples given by the user or a special *null* value if no such query exists), and *complete* (i.e., able to produce every query with a sufficiently rich set of examples).

Since we aim at a polynomial time algorithm that returns a query consistent with a sample, we must first investigate the *consistency checking* problem i.e., deciding whether such a query exists. To this purpose, we first identify a necessary and sufficient condition for a sample to be consistent.

Lemma 3.1 *Given a graph G and a sample S, S is consistent iff for every $\nu \in S_+$ it holds that $paths_G(\nu) \not\subseteq paths_G(S_-)$.*

From this characterization we can derive that the fundamental problem of consistency checking is PSPACE-complete.

Lemma 3.2 *Given a graph G and a sample S, deciding whether S is consistent is PSPACE-complete.*

PROOF SKETCH. The membership to PSPACE follows from Lemma 3.1 and the known result that deciding the inclusion of NFAs is PSPACE-complete [39]. The PSPACE-hardness follows by reduction from the universality of the union problem for DFAs, known as being PSPACE-complete [28]. ▫

This implies that an algorithm able to always answer *null* in polynomial time when the sample is inconsistent does not exist, hence our class of queries is not learnable in the classical framework. One solution could be to study less expressive classes of queries. However, as shown by the following Lemma, consistency checking remains intractable even for a very restricted class of queries, referred as "SORE(·)" in [4].

Lemma 3.3 *Given a graph G and a sample S, deciding whether there exists a query of the form $a_1 \cdot \ldots \cdot a_n$ (pairwise distinct symbols) consistent with S is NP-complete.*

PROOF SKETCH. For the membership of the problem to NP, we point out that a non-deterministic Turing machine guesses a query q that is a concatenation of pairwise distinct symbols (hence of length bounded by $|\Sigma|$) and then checks whether q is consistent with S. The NP-hardness follows by reduction from 3SAT, well-known as being NP-complete. ▫

The proofs of Lemma 3.2 and 3.3 rely on techniques inspired by the definability problem for graph query languages [4]. We also point out that the same intractability results for consistency checking hold for binary semantics.

Another way to overcome the intractability of our class of queries is to relax the soundness condition and adopt a learning framework with *abstain*, similarly to what has been recently done for learning XML transformations [30]. More precisely, we allow the learning algorithm to answer a special value *null* whenever it cannot efficiently construct a consistent query. In practice, the *null* value is interpreted as "not enough examples have been provided". However, the learning algorithm should always return in polynomial time either a consistent query or *null*. As an additional clause, we require a learning algorithm to be *complete* i.e., when the input sample contains a *polynomially-sized characteristic sample* [18, 20], the algorithm must return the goal query. More formally, we have the following.

Definition 3.4 *A class of queries $\mathcal{Q}$ is* learnable with abstain in polynomial time and data *if there exists a polynomial learning algorithm learner that is:*

1. **Sound with abstain.** *For every graph G and sample S over G, the algorithm learner(G, S) returns either a query in $\mathcal{Q}$ that is consistent with S, or null if no such query exists or it cannot be constructed efficiently.*
2. **Complete.** *For every query $q \in \mathcal{Q}$, there exists a graph G and a polynomially-sized characteristic sample CS on G s.t. for every sample S extending CS consistently with q (i.e., $CS \subseteq S$ and q is consistent with S), the algorithm learner(G, S) returns q.*

Note that the polynomiality depends on the choice of a representation for queries and recall that we represent each PQ with its canonical DFA. Next, we present a polynomial learning algorithm fulfilling the two aforementioned conditions and we point out the construction of a polynomial characteristic sample to show the learnability of PQ.

3.2 Learning algorithm

In a nutshell, the idea behind our learning algorithm is the following: for each positive node, we seek the path that the user followed to label such a node, then we construct the disjunction of the paths obtained in the previous step, and we end by generalizing this disjunction while remaining consistent with both positive and negative examples.

More formally, the algorithm consists of two complementary steps that we describe next: *selecting the smallest consistent paths* and *generalizing* them.

Selecting the smallest consistent paths (SCPs). Since the labeled nodes in a graph may be the origin of multiple paths, classical algorithms for learning regular expressions from words, such as RPNI [35], are not directly applicable to our setting. Indeed, we have a set of nodes in the graph from which we have to first select (from a potentially infinite set) the paths responsible for their selection. Therefore, the first challenge of our algorithm is to select for each positive node a path that is not covered by any negative. We call such a path a *consistent path*. One can select consistent paths by simply enumerating (according to the canonical order $\leqslant$) the paths of each node labeled as positive and stopping when a consistent path for each node is found. We refer to the obtained set of paths as the set of *smallest consistent paths* (SCPs) because they are the smallest (w.r.t. $\leqslant$) consistent paths for each node. As an example, for the graph G_0 in Figure 3 and a sample s.t. $S_+ = \{\nu_1, \nu_3\}$ and $S_- = \{\nu_2, \nu_7\}$, we obtain the SCPs abc and c for ν_1 and ν_3, respectively. Notice that in this case the disjunction of the SCPs (i.e., the query $c + (a \cdot b \cdot c)$) is consistent with the input sample and one may think that a learning algorithm should return such a query. The shortcoming of such an approach is that the learned query would be always very simple in the sense that it uses only concatenation and disjunction. Since we want a learning algorithm that covers all the expressibility of PQ (in particular including the Kleene star), we need to extend the algorithm with a further step, namely the generalization. We detail such a step at the end of this section.

Another problem is that the user may provide an inconsistent sample, by labeling a positive node having no consistent path (cf. Lemma 3.1). To clarify when such a situation occurs, consider a simple graph such as the one in Figure 5 having one positive node (labeled with $+$) and two negative ones (labeled with $-$). We observe that the positive

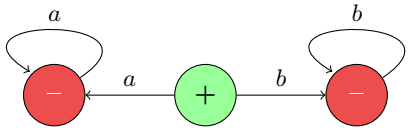


Figure 5: A graph with an inconsistent sample.

node has an infinite number of paths. However, all of them are covered by the two negative nodes, thus yielding an inconsistent sample. This example also shows that the simple procedure described above (i.e., enumerate the paths of the positive nodes and stop when finding consistent ones) fails because it leads to enumerating an infinite number on paths and never halting. On the other hand, we cannot perform consistency checking before selecting the SCPs since this fundamental problem is intractable (cf. Lemma 3.2 and 3.3). As a result, to avoid this possibly infinite enumeration, we choose to fix the maximal length of a SCP to a bound k. This bound engenders a new issue: for a fixed k it may not be possible to detect SCPs for all positive nodes. For instance, assume a graph that the user labels consistently with the query $(a \cdot b)^* \cdot c$, in particular she labels three positive nodes for which the SCPs are c, abc, and $ababc$. For a fixed $k = 3$, the SCP $ababc$ is not detected and the disjunction of the first two SCPs (i.e., $c + (a \cdot b \cdot c)$) is not a query consistent with the sample.

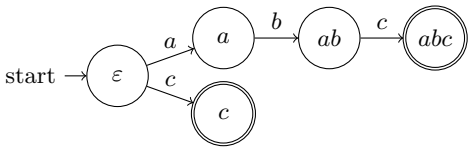
For all these reasons, we introduce a generalization phase in the algorithm, which permits to solve the two mentioned shortcomings i.e., (i) to learn a query that covers all the expressibility of PQ, and (ii) to select all positive examples even though not all of them have SCPs shorter than k.

Generalizing SCPs. We have seen how to select, whenever possible, a SCP of length bounded by k for each positive example. Next, we show how we can employ these SCPs to construct a more general query. The *learning algorithm* (Algorithm 1) takes as input a graph G and a sample S, and outputs a query q consistent with S whenever such query exists and can be built using SCPs of length bounded by k; otherwise, the algorithm outputs a special value *null*.

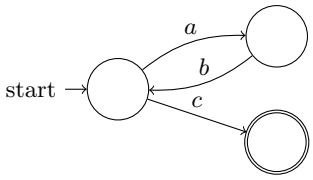
Algorithm 1 Learning algorithm – $learner(G, S)$.

Input: graph G, sample S
Output: query q consistent with S or *null*
Parameter: fixed $k \in \mathbb{N}$ //maximal length of a SCP
1: **for** $\nu \in S_+$. $\exists p \in \Sigma^{\leqslant k}$. $p \in paths_G(\nu) \backslash paths_G(S_-)$ **do**
2: $\quad P := P \cup \{\min_{\leqslant}(paths_G(\nu) \backslash paths_G(S_-))\}$
3: **let** A be the prefix tree acceptor for P
4: **while** $\exists s, s' \in A$. $L(A_{s' \to s}) \cap paths_G(S_-) = \varnothing$ **do**
5: $\quad A := A_{s' \to s}$
6: **if** $\forall \nu \in S_+$. $L(A) \cap paths_G(\nu) \neq \varnothing$ **then**
7: $\quad$ **return** query q represented by the DFA A
8: **return** *null*

We illustrate the algorithm on the graph G_0 in Figure 3 with a sample S s.t. $S_+ = \{\nu_1, \nu_3\}$ and $S_- = \{\nu_2, \nu_7\}$. For ease of exposition, we assume a fixed $k = 3$ (we explain how to obtain the value of k theoretically in Section 3.3 and empirically in Section 5). At first (lines 1-2), the algorithm constructs the set P of SCPs bounded by k for the positive nodes from which these paths in P can be constructed. Note that by $\Sigma^{\leqslant k}$ we denote the set of all paths of length at most k. For instance, on our example in Figure 3, we obtain $P = \{abc, c\}$. Then (line 3), we construct the *PTA* (*prefix tree acceptor*) [18] of P, which is basically a tree-like DFA accepting only the paths in P and having as states all their prefixes. Figure 6(a) illustrates the obtained PTA A for our example. Then (lines 4-5), we *generalize* A by merging two of its states if the obtained DFA selects no negative node. Note that by $A_{s' \to s}$ we denote the DFA obtained from A by modifying each occurrence of the state s' in s. Recall that on our example we have $P = \{abc, c\}$ and the PTA A in Figure 6(a). Next, we try to merge states of A: the states ε and a cannot be merged (because the obtained DFA would select the path bc that is covered by the negative ν_2), the states ε and c cannot be merged (because the path ε is covered by both negatives), while the states ε and ab can be merged without covering any negative example. On our example, we obtain the DFA in Figure 6(b), where no further states can be merged. Finally, the algorithm checks whether the query represented by A selects all the positive examples (not only those from whose SCPs we have constructed A), and if this is the case, it outputs the query (lines 6-7). In our case, the obtained $(a \cdot b)^* \cdot c$ selects all positive nodes hence is returned.



(a) Prefix tree acceptor.



(b) Result of generalization.

Figure 6: DFAs considered by *learner* for Figure 3.

3.3 Learnability results

Recall that in Section 3.1 we have formally defined what means for a class of queries to be learnable while in Section 3.2 we have proposed a learning algorithm for PQ. In this section, we prove that the proposed algorithm satisfies the conditions of Definition 3.4, thus showing the main theoretical result of the paper. We conclude the section with practical observations related to our learnability result.

By $\text{PQ}^{\leqslant n}$ we denote the PQ of size at most n.

Theorem 3.5 *The query class* $\text{PQ}^{\leqslant n}$ *is learnable with abstain in polynomial time and data, using the algorithm learner with the parameter* k *set to* $2 \times n + 1$.

Proof. First, we point out that *learner* works in *polynomial time* in the size of the input. Indeed, the number of paths to explore for each positive node (lines 1-2) is polynomial since the length of paths is bounded by a fixed k. Then, both testing the consistency of queries considered during the generalization (lines 3-5) and testing whether the computed query selects all positive nodes (lines 6-7) reduce to deciding the emptiness of the intersection of two NFAs, known as being in PTIME [29]. Moreover, *learner* is *sound with abstain* since it returns either a query consistent with the input sample if it is possible to construct it using SCPs of length bounded by k, or *null* otherwise.

As for the *completeness* of *learner*, we show, for every $q \in \text{PQ}$, the construction of a graph G and of a characteristic sample CS on G with the properties from Definition 3.4 i.e., for every sample S that extends CS consistently with q (i.e., $CS \subseteq S$ and q is consistent with S), the algorithm $learner(G, S)$ returns q. The idea behind the construction is the following: we know that RPNI [35] is an algorithm that takes as input positive and negative word examples and outputs a DFA describing a consistent regular language, and moreover, the core of RPNI is based on DFA generalization via state merges similarly to *learner*; hence, for a query q, we want a graph and a sample on it s.t. when *learner* selects the SCPs, it should get exactly the words that RPNI needs for learning q if we see it as a regular language. Then, since RPNI is guaranteed to learn the goal regular language from these words, we infer that if *learner* selects and generalizes the SCPs corresponding to them, then *learner* is also guaranteed to learn the goal PQ.

We exemplify the construction on the query $q = (a \cdot b)^* \cdot c$. First, given q, we construct two sets of words P_+ and P_- that correspond to a characteristic sample used by RPNI to infer the regular language of q. In our case, we obtain $P_+ = \{c, abc\}$ and $P_- = \{\varepsilon, a, ab, ac, bc\}$. Then, the characteristic graph for learning the graph query q needs (i) for each $p \in P_+$, a node $\nu \in CS_+$ s.t. $p = \min_{\leqslant}(L(q) \cap paths_G(\nu))$, (ii) for each $p \in P_-$, a node $\nu \in CS_-$ s.t. $p \in paths_G(\nu)$, and (iii) for each p' that is smaller (w.r.t. the canonical order $\leqslant$) than a word $p \in P_+$ and is not prefixed by any word in $L(q)$, a node $\nu \in CS_-$ s.t. $p' \in paths_G(\nu)$. For our query $(a \cdot b)^* c$, (i) implies two positive nodes: a node ν s.t. $c \in paths_G(\nu)$ and another node ν' s.t. $abc \in paths_G(\nu')$ and $c \notin paths_G(\nu')$, while (ii) and (iii) imply a node ν'' s.t. $\nu'' \notin q(G)$ and $\{\varepsilon, a, ab, ac, bc\} \subseteq paths_G(\nu'')$ (cf. ii) and $\{\varepsilon, a, b, aa, ab, ac, ba, bb, bc, aaa, aab, aac, aba, abb\} \subseteq paths_G(\nu'')$ (cf. iii). In Figure 7 we illustrate such a graph and we highlight the two positive and one negative node examples.

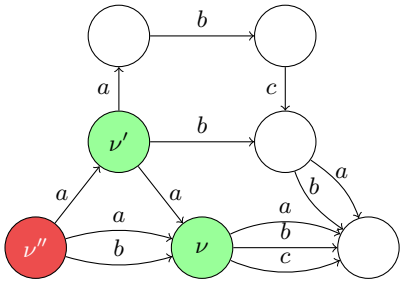


Figure 7: Graph from the proof of Theorem 3.5.

Recall that the size of a query is the number of states of its canonical DFA. According to the above construction, we need $|CS_+| = |P_+|$ and $|CS_-| = 1$. Since $|P_+|$ is polynomial in the size of the query [35], we infer that $|CS|$ is also polynomial in the size of the query. Moreover, to learn the regular language of a query of size n, the longest path in P_+ is of size $2 \times n + 1$ [35]. Hence, to be able to select this path with *learner* (assuming the presence of a characteristic sample), we need the parameter k of *learner* to be at least $2 \times n + 1$. Thus, for each possible size n of the goal query there exists a polynomial learning algorithm satisfying the conditions of Definition 3.4, which concludes the proof. ▫

We end this section with some practical observations related to our learnability result.

First, we point out that although Theorem 3.5 requires a certain theoretical value for k to guarantee learnability of queries of a certain size, our experiments indicate that small values of k (between 2 and 4) are enough to cover most practical cases. Then, even though the definition of learnability requires that one characteristic sample exists, in practice there may be many of such samples and there exists an infinite number of graphs on which we can build them. In fact, a graph that contains a subgraph with a characteristic sample is also characteristic.

Second, we also point out that a practical sample may be characteristic without having all negative paths on the same node as required by the aforementioned construction. For instance, the sample that we have used to illustrate the learning algorithm (i.e., the sample s.t. $S_+ = \{\nu_1, \nu_3\}$ and $S_- = \{\nu_2, \nu_7\}$ on the graph in Figure 3) is characteristic for $(a \cdot b)^* \cdot c$ and all above mentioned negatives paths are covered by two negative nodes.

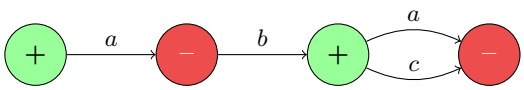


Figure 8: A graph and a sample for $(a \cdot b)^* \cdot c$.

Third, if a graph does not own a characteristic sample, the user's goal query on that graph cannot be exactly identified. In such a case, the learning algorithm returns a query equivalent to the goal query on the graph and hence *indistinguishable* by the user (i.e., they select exactly the same set of nodes). For instance, take the graph in Figure 8 and assume a user labeling the nodes w.r.t. the goal $(a \cdot b)^* \cdot c$. The learning algorithm returns the query a that selects exactly the same set of nodes on this graph.

Fourth, the presented learning techniques are directly applicable to different query semantics such as binary and n-ary queries. We give here only the intuition behind these applications and we point out that more details about the corresponding algorithms can be found in the appendix of our technical report [11]. To learn a binary query, the only change to Algorithm 1 is that each positive example implies a smaller number of candidate paths from which we have to choose a consistent one (since the destination node is also known). Then, for the n-ary case (i.e., when an example is a tuple of nodes labeled with + or −), we have simply to apply the previous algorithm to learn a query for each position in the tuple and then to combine those.

4. LEARNING FROM INTERACTIONS

In this section, we investigate the problem of query learning from a different perspective. In Section 3 we have studied the setting of a fixed set of examples provided by the user and no interaction with her during the learning process. In this section, we consider an *interactive scenario* where the learning algorithm starts with an empty sample and continuously interacts with the user and asks her to label additional nodes until she is satisfied with the output of the learned query. To this purpose, we first propose the *interactive scenario* (Section 4.1). Then, we discuss different parameters for it, in particular what means for a node to be *informative* and what is a *practical strategy* of proposing nodes to the user (Section 4.2). We also point out that we have employed the aforementioned interactive scenario as the core of a system for interactive path query specification on graphs [12].

4.1 Interactive scenario

We consider the following *interactive scenario*. The user is presented with a node of the graph and indicates whether the node is selected or not by the query that she has in mind. We repeat this process until a sufficient knowledge of the goal query has been accumulated (i.e., there exists at most one query consistent with the user's labels). This scenario is inspired by the well-known framework of *learning with membership queries* [3] and we have recently formalized it as a general paradigm for learning queries on big data [13]. In Figure 9, we describe the current instantiation for path queries on graphs and we detail next its different steps.

(1) (2) We consider as input a graph database G. Initially, we assume an empty sample that we enrich via simple interactions with the user. The interactions continue until

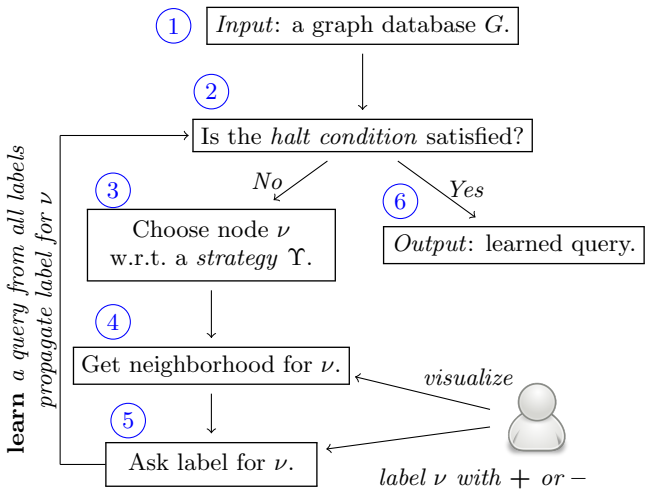


Figure 9: Interactive scenario.

a *halt condition* is satisfied. A natural halt condition is to stop the interactions when there is exactly one consistent query with the current sample. In practice, we can imagine weaker conditions e.g., the user may stop the process earlier if she is satisfied by some candidate query proposed at some intermediary stage during the interactions.

(3) We propose nodes of the graph to the user according to a *strategy* Υ i.e., a function that takes as input a graph G and a sample S, and returns a node from G. Since our goal is to minimize the amount of effort needed to learn the user's goal query, a smart strategy should avoid proposing to the user nodes that do not bring any information to the learning process. The study of such strategies yields to defining the notion of *informative nodes* that we formalize in the next section.

(4) A node by itself does not carry enough information to allow the user to understand whether it is part of the query result or not. Therefore, we have to enhance the information of a node by zooming out on its *neighborhood* before actually showing it to the user. This step has the goal of producing a small, easy to visualize fragment of the initial graph, which permits the user to label the proposed node as a positive or a negative example. More concretely, in a practical scenario, all nodes situated at a distance k (as the parameter of Algorithm 1 explained in Section 3.2) should be sufficient for the user to decide whether she wants or not the proposed node. In any case, the user has neither to visualize all the graph that can be potentially large, nor to look by herself for interesting nodes because our interactive scenario proposes such nodes to the user.

(5) (6) The user visualizes the neighborhood of a given node ν and labels ν w.r.t. the goal query that she has in mind. Then, we propagate the given label in the rest of the graph and prune the nodes that become uninformative. Moreover, we run the learning algorithm *learner* (i.e., Algorithm 1 from Section 3.2), which outputs in polynomial time either a query consistent with all labels provided by the user, or *null* if such a query does not exist or cannot be constructed efficiently. When the halt condition is satisfied, we return the latest output of *learner* to the user. In particular, the halt condition may take into account such an

intermediary learned query q e.g., when the user is satisfied by the output of q on the instance and wants to stop the interactions.

In the next section, we precisely describe what means for a node to be informative for the learning process and what is a practical strategy of proposing nodes to the user.

4.2 Informative nodes and practical strategies

Before explaining the informative nodes, we first define the set of all queries consistent with a sample S over a graph G:

$$\mathcal{C}(G, S) = \{q \in \text{PQ} \mid S_+ \subseteq q(G) \wedge S_- \cap q(G) = \varnothing\}.$$

Assuming that the user labels the nodes consistently with some goal query q, the set $\mathcal{C}(G, S)$ always contains q. Initially, $S = \varnothing$ and $\mathcal{C}(G, S) = \text{PQ}$. Therefore, an ideal strategy of presenting nodes to the user is able to get us quickly from $S = \varnothing$ to a sample S s.t. $\mathcal{C}(G, S) = \{q\}$. In particular, a good strategy should not propose to the user the *certain nodes* i.e., nodes not yielding new information when labeled by the user. Formally, given a graph G, a sample S, and an unlabeled node $\nu \in G$, we say that ν is *certain* (w.r.t. S) if it belongs to one of the following sets:

$$Cert_+(G, S) = \{\nu \in G \mid \forall q \in \mathcal{C}(G, S).\ \nu \in q(G)\},$$
$$Cert_-(G, S) = \{\nu \in G \mid \forall q \in \mathcal{C}(G, S).\ \nu \notin q(G)\}.$$

In other words, a node is certain with a label α if labeling it explicitly with α does not eliminate any query from $\mathcal{C}(G, S)$. For instance, take the graph in Figure 10 with a positive, a negative, and an unlabeled node, which belongs to $Cert_+$ because it is selected by the unique (prefix-free) query in $\mathcal{C}(G, S)$ (i.e., b). Additionally, we observe that labeling it otherwise (i.e., with a $-$) leads to an inconsistent sample. The notion of certain nodes is inspired by possible world semantics and certain answers [23], and already employed for XML querying for non-expert users [17] and for the inference of relational joins [14].

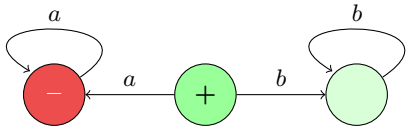


Figure 10: Two labeled nodes and a certain node.

Next, we give necessary and sufficient conditions for a node to be certain, for both positive and negative labels:

Lemma 4.1 *Given a sample S and a node ν from G:*

1. $\nu \in Cert_+(G, S)$ *iff there exists* $\nu' \in S_+$ *s.t.* $paths_G(\nu') \subseteq paths_G(S_-) \cup paths_G(\nu)$,
2. $\nu \in Cert_-(G, S)$ *iff* $paths_G(\nu) \subseteq paths_G(S_-)$.

Additionally, given a graph G, a sample S, and a node ν, we say that ν is *informative* (w.r.t. S) if ν has not been labeled by the user nor it is certain. Unfortunately, by using the characterization from Lemma 4.1, we derive the following.

Lemma 4.2 *Given a graph G and a sample S, deciding whether a node ν is informative is PSPACE-complete.*

An intelligent strategy should propose to the user only informative nodes. Since deciding the informativeness of a node is intractable (cf. Lemma 4.2), we need to explore *practical strategies* that efficiently compute the next node to label. Consequently, we propose two simple but effective strategies that we detail next and that we have evaluated experimentally. The basic idea behind them is to avoid the intractability of deciding informativeness of a node by looking only at a small number of paths of that node. More precisely, we say that a node is *k-informative* if it has at least one path of length at most k that is not covered by a negative example. If a node is k-informative, then it is also informative, otherwise we are not able to establish its informativeness w.r.t. the current k. Then, strategy kR consists of taking *randomly* a k-informative node while strategy kS consists of taking the k-informative node having the *smallest* number of non-covered k-paths, thus favoring the nodes for which computing the SCPs is easier. In the next section, we discuss the performance of these strategies as well as how we set the k in practice.

5. EXPERIMENTS

In this section, we present an experimental study devoted to gauge the performance of our learning algorithms. In Section 5.1, we introduce the used datasets: the *AliBaba biological graph* and randomly generated *synthetic graphs*. In Section 5.2 and Section 5.3, we present the results for the two settings under study: *static* and *interactive*, respectively. Our algorithms have been implemented in C and our experiments have been run on an Intel Core i7 with 4×2.9 GHz CPU and 8 GB RAM.

5.1 Datasets

Despite the increasing popularity of graph databases, benchmarks allowing to assess the performance of emerging graph database applications are still lacking or under construction [10]. In particular, there is no established benchmark devoted to graph queries defined by regular expressions. Due to this lack, we have adopted a real dataset recently used by [27] to evaluate the performance of optimization algorithms for regular path queries. This dataset, called *AliBaba* [36], represents a real graph from research on biology, extracted by text mining on PubMed. The dataset has a semantic part consisting of a network of protein-protein interactions and a textual part, reporting text co-occurrence for words. The first part was more appropriate to apply our learning algorithms than the second. Therefore, we have extracted the semantic part from the original graph, thus obtaining a subgraph of about 3k nodes and 8k edges. Similarly, from the set of real-life queries reported in [27], we have retained those that select at least one node on the graph to obtain at least one positive example for learning. Thus, we have used 6 biological queries (denoted by $bio_1, \ldots, bio_6$), which are structurally complex and have selectivities varying from 1 to a total of 711 nodes i.e., from 0.03% to 22% of the nodes of the graph. We summarize these queries in Table 1. By small letters a, b we denote symbols from the alphabet while by capital letters A, C, E, I we denote disjunctions of symbols from the alphabet i.e., expression of the form $a_1 + \ldots + a_n$. These disjunctions contain up to 10 symbols, with possibly overlapping ones among them.

Additionally, we have implemented a synthetic data generator, which yields graphs of varying size and similar to real-world graphs. The latest feature let us generate *scale-free* graphs with a *Zipfian* edge label distribution [27]. We report here the results for generated graphs of size 10k, 20k,

	Query	*Selectivity*
bio_1	$b \cdot A \cdot A^*$	0.03%
bio_2	$C \cdot C^* \cdot a \cdot A \cdot A^*$	0.2%
bio_3	$C \cdot E$	3%
bio_4	$I \cdot I \cdot I^*$	11%
bio_5	$A \cdot A \cdot A^* \cdot I \cdot I \cdot I^*$	12%
bio_6	$A \cdot A \cdot A^*$	22%

Table 1: Biological queries.

and 30k nodes, and with a number of edges three times larger. Moreover, we focus on synthetic queries that are similar in structure to the aforementioned real-life biological queries. In particular, the three queries that we report here (denoted by syn_1, syn_2, and syn_3) have the structure $A \cdot B^* \cdot C$, where A, B, and C are disjunctions of up to 10 symbols, with overlapping ones among them. The difference between these three queries is w.r.t. their selectivity: regardless the actual size of the graph, syn_1, syn_2, and syn_3 select 1%, 15%, and 40% of the graph nodes, respectively.

Before presenting the experimental results, we say a few words about how we set empirically the parameter k from *learner*. Since in our experiments we assume that the user labels the nodes of the graph consistently with some goal query, the input sample is always consistent. Hence, there exists a consistent path for each positive node and we dynamically discover the length of the SCPs. In particular, we start with $k = 2$; if for a given k, the query learned using SCPs shorter that k does not select all positive nodes, we increment k and iterate. For the interactive case, the aforementioned procedure becomes: start with $k = 2$; seek k-informative nodes (cf. Section 4.2) and increase k when the current k does not yield any k-informative node. In practice, in the majority of cases $k = 2$ is sufficient and it may reach values up to 4 in some isolated cases.

5.2 Static experiments

The setup of static experiments is as follows. Given a graph and a goal query, we take as positive examples some random nodes of the graph that are selected by the query and as negative examples some random nodes that are not selected by it. All these examples are given as input to *learner*, which returns a consistent query. We consider the learned query as a binary classifier and we measure the F1 score w.r.t. the goal query. Thus, for different percentages of labeled nodes in the graph, we measure the F1 score of the learned query along with the learning time. We present the summary of results in Figure 11 and 12, which show the F1 score and the learning time for the biological (a) and synthetic queries (b, c, d), respectively. Since the positive effect of the generalization in addition to the selection of SCPs is generally of 1% in F1 score, we do not highlight the two steps of the algorithm for the sake of figure readability.

We can observe that, not surprisingly, by increasing the percentage of labeled nodes of the graph, the F1 score also increases (Figure 11). Overall, the F1 score is 1 or sufficiently close to 1 for all queries, except a few cases that we discuss below. The worst behavior is clearly exhibited by query bio_5, when we can observe that the F1 score converges to 1 less faster than the others. For this query, we can also observe that the learning time is higher (Figure 12(a)). This is due to the fact that the graph is not characteristic for it (cf. Section 3.3), hence the selection of SCPs yields paths that are not relevant for the target query.

For what concerns the learning time, this remains reasonable (of the order of seconds) for all the queries and for both datasets. The most selective queries (bio_4, bio_5, bio_6) are more problematic since they entail a larger number of positive nodes in the step of selection of the SCPs. As a conclusion, notice that even when the F1 score is very high, these results on the static scenario are not fully beneficial since we need to label at least 7% of the graph nodes to have an F1 score equal to 1. As we later show with the interactive experiments, we can significantly reduce the number of labels needed to reach an F1 score equal to 1.

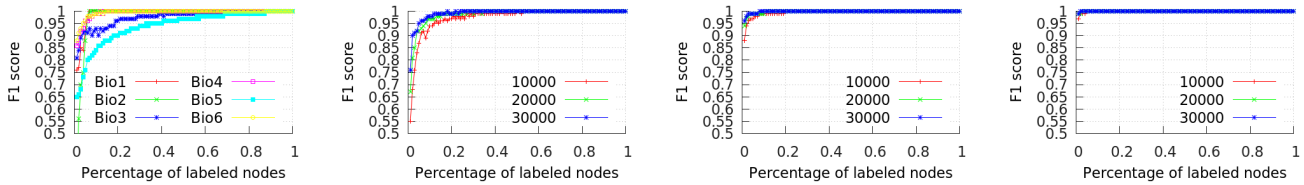
Finally, the synthetic experiments on various graph sizes and query selectivities confirm the aforementioned observations. In particular, when the queries are more selective (as syn_2 and syn_3), increasing the number of examples implies more visible changes in the learning time (cf. Figure 12). Additionally, we observe the goal queries with higher selectivity converge faster to a F1 score equal to 1 (cf. Figure 11). Intuitively, this is due to the fact that such cases imply a bigger number of positive examples from which the learning algorithm can benefit to generalize faster the goal query.

5.3 Interactive experiments

The setup of interactive experiments is as follows. Given a graph and a goal query, we start with an empty sample and we continuously select a node that we ask the user to label, until the learned query selects exactly the same set of nodes as the goal query or, in other words, until the goal query and the learned query are indistinguishable by the user (cf. Section 3.3). This corresponds to obtaining an F1 score of 1. In this setting, we measure the percentage of the labeled nodes of the graph and the learning time i.e., the time needed to compute the next node to label. In particular, the number of interactions corresponds to the total number of examples, the latter being the sum of the number of positive examples and the number of negative examples. The summary of interactive experiments is presented in Table 2.

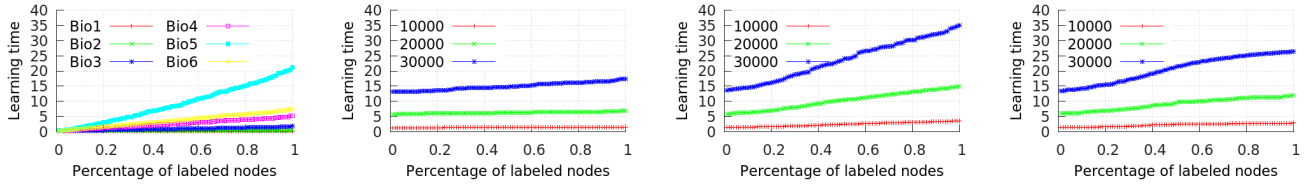
We can observe that, differently from the static scenario, labeling around 1% of the nodes of the graph suffices to learn a query with F1 score equal to 1. Even for the most difficult one (bio_5), we get a rather significant improvement, as the percentage of interactions is drastically reduced to 7.7% of the nodes in the graph (while in the static case it was 87% of the nodes in the graph). Overall, these numbers prove that the interactive scenario considerably reduces the number of examples needed to infer a query of F1 score equal to 1. The synthetic experiments confirm this behavior for various graph sizes and query selectivities. While learning a query with F1 score equal to 1 corresponds to the strongest halt condition of our interactive scenario (cf. Figure 9), we believe that in practice the user may choose to stop the interactions earlier (and hence label less nodes) if she is satisfied by an intermediate query proposed by the learning algorithm. We consider such halt conditions in our system demo [12].

Moreover, these experiments also show that the two strategies (kS and kR, cf. Section 4.2) have a similar behavior, even though kS is slightly better (w.r.t. minimizing the number of interactions) for the most selective queries. Intuitively, this happens because such a strategy favors the nodes for which computing the SCPs has a smaller space of solutions (cf. Section 4.2). Finally, we can observe that the two strategies are also efficient, as they lead to a learning time of the order of seconds in all cases.



(a) Biological queries. (b) Synthetic query syn_1. (c) Synthetic query syn_2. (d) Synthetic query syn_3.

Figure 11: Summary of static experiments – F1 score.



(a) Biological queries. (b) Synthetic query syn_1. (c) Synthetic query syn_2. (d) Synthetic query syn_3.

Figure 12: Summary of static experiments – Learning time (seconds).

Dataset	*Bio query / Graph size*	*Labels needed for F1 score = 1* **without interactions**	*Interactive strategy*	*Labels needed for F1 score = 1* **with interactions**	*Time between interactions (seconds)*
Biological queries	bio_1	7%	kR	0.06%	0.19
			kS	0.06%	0.33
	bio_2	7%	kR	1.78%	0.26
			kS	3.13%	0.48
	bio_3	66%	kR	1.24%	0.34
			kS	1.49%	0.45
	bio_4	12%	kR	1.32%	0.23
			kS	0.22%	0.53
	bio_5	87%	kR	7.7%	3.45
			kS	7.39%	3.79
	bio_6	12%	kR	1.18%	0.24
			kS	0.35%	0.3
Synthetic query syn_1	10000	51%	kR	0.15%	1.33
			kS	0.17%	1.35
	20000	26%	kR	0.07%	5.83
			kS	0.06%	5.92
	30000	22%	kR	0.04%	13.5
			kS	0.04%	13.95
Synthetic query syn_2	10000	20%	kR	0.38%	1.57
			kS	0.36%	1.58
	20000	11%	kR	0.23%	6.63
			kS	0.22%	6.78
	30000	8%	kR	0.17%	15.24
			kS	0.16%	15.38
Synthetic query syn_3	10000	5%	kR	0.1%	1.32
			kS	0.1%	1.32
	20000	3%	kR	0.05%	5.66
			kS	0.05%	5.68
	30000	2%	kR	0.04%	13.15
			kS	0.04%	13.41

Table 2: Summary of interactive experiments.

6. CONCLUSIONS AND FUTURE WORK

We have studied the problem of learning path queries defined by regular expressions from user examples. We have identified fundamental difficulties of the problem setting, formalized what means for a class of queries to be learnable, and shown that the above class enjoys learnability. Additionally, we have investigated an interactive scenario, analyzed what means for a node to be informative, and proposed practical strategies of presenting examples to the user. Finally, we have shown the effectiveness of the algorithms and the improvements of using an interactive approach through an experimental study on both real and synthetic datasets.

We envision several directions of our work, one of which being to sample a graph and finding informative nodes on representative samples, in the spirit of [31]. Moreover, motivated by the absence of benchmarks devoted to queries defined by regular expressions, we want to develop such a benchmark. This would permit to better analyze the performance of algorithms involving regular expressions on graphs, including the learning algorithms proposed in this paper.

Acknowledgements. We would like to thank the authors of [27] for sharing the AliBaba dataset and to Sarah Cohen-Boulakia for her comments on a draft of the paper.

7. REFERENCES

[1] A. Abouzied, D. Angluin, C. H. Papadimitriou, J. M. Hellerstein, and A. Silberschatz. Learning and verifying quantified boolean queries by example. In *PODS*, pages 49–60, 2013.

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

[3] D. Angluin. Queries and concept learning. *Machine Learning*, 2(4):319–342, 1988.

[4] T. Antonopoulos, F. Neven, and F. Servais. Definability problems for graph query languages. In *ICDT*, pages 141–152, 2013.

[5] M. Arenas and J. Pérez. Querying semantic web data with SPARQL. In *PODS*, pages 305–316, 2011.

[6] G. Bagan, A. Bonifati, and B. Groz. A trichotomy for regular simple path queries on graphs. In *PODS*, pages 261–272, 2013.

[7] P. Barceló. Querying graph databases. In *PODS*, pages 175–188, 2013.

[8] K. Belhajjame. Annotating the behavior of scientific modules using data examples: A practical approach. In *EDBT*, pages 726–737, 2014.

[9] G. J. Bex, W. Gelade, F. Neven, and S. Vansummeren. Learning deterministic regular expressions for the inference of schemas from XML data. *TWEB*, 4(4), 2010.

[10] P. A. Boncz, I. Fundulaki, A. Gubichev, J.-L. Larriba-Pey, and T. Neumann. The linked data benchmark council project. *Datenbank-Spektrum*, 13(2):121–129, 2013.

[11] A. Bonifati, R. Ciucanu, and A. Lemay. Learning path queries on graph databases, 2014. TR at `http://hal.inria.fr/hal-01068055`.

[12] A. Bonifati, R. Ciucanu, and A. Lemay. Interactive path query specification on graph databases. In *EDBT*, 2015.

[13] A. Bonifati, R. Ciucanu, A. Lemay, and S. Staworko. A paradigm for learning queries on big data. In *Data4U*, pages 7–12, 2014.

[14] A. Bonifati, R. Ciucanu, and S. Staworko. Interactive inference of join queries. In *EDBT*, pages 451–462, 2014.

[15] A. Bonifati, R. Ciucanu, and S. Staworko. Interactive join query inference with JIM. *PVLDB*, 7(13):1541–1544, 2014.

[16] R. Ciucanu and S. Staworko. Learning schemas for unordered XML. In *DBPL*, pages 31–40, 2013.

[17] S. Cohen and Y. Weiss. Certain and possible XPath answers. In *ICDT*, pages 237–248, 2013.

[18] C. de la Higuera. *Grammatical Inference: Learning Automata and Grammars*. Cambridge University Press, 2010.

[19] M. J. Franklin, D. Kossmann, T. Kraska, S. Ramesh, and R. Xin. CrowdDB: answering queries with crowdsourcing. In *SIGMOD Conference*, pages 61–72, 2011.

[20] E. M. Gold. Complexity of automaton identification from given data. *Information and Control*, 37(3):302–320, 1978.

[21] C. Heinlein. Workflow and process synchronization with interaction expressions and graphs. In *ICDE*, pages 243–252, 2001.

[22] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, 1979.

[23] T. Imielinski and W. Lipski Jr. Incomplete information in relational databases. *J. ACM*, 31(4):761–791, 1984.

[24] H. V. Jagadish, A. Chapman, A. Elkiss, M. Jayapandian, Y. Li, A. Nandi, and C. Yu. Making database systems usable. In *SIGMOD Conference*, pages 13–24, 2007.

[25] N. Jayaram, A. Khan, C. Li, X. Yan, and R. Elmasri. Towards a query-by-example system for knowledge graphs. In *GRADES*, 2014.

[26] M. J. Kearns and U. V. Vazirani. *An introduction to computational learning theory.* MIT Press, 1994.

[27] A. Koschmieder and U. Leser. Regular path queries on large graphs. In *SSDBM*, pages 177–194, 2012.

[28] D. Kozen. Lower bounds for natural proof systems. In *FOCS*, pages 254–266, 1977.

[29] K.-J. Lange and P. Rossmanith. The emptiness problem for intersections of regular languages. In *MFCS*, pages 346–354, 1992.

[30] G. Laurence, A. Lemay, J. Niehren, S. Staworko, and M. Tommasi. Learning sequential tree-to-word transducers. In *LATA*, pages 490–502, 2014.

[31] J. Leskovec and C. Faloutsos. Sampling from large graphs. In *KDD*, pages 631–636, 2006.

[32] K. Losemann and W. Martens. The complexity of regular expressions and property paths in SPARQL. *ACM Trans. Database Syst.*, 38(4):24, 2013.

[33] D. Mottin, M. Lissandrini, Y. Velegrakis, and T. Palpanas. Exemplar queries: Give me an example of what you need. *PVLDB*, 7(5):365–376, 2014.

[34] A. Nandi and H. V. Jagadish. Guided interaction: Rethinking the query-result paradigm. *PVLDB*, 4(12):1466–1469, 2011.

[35] J. Oncina and P. García. Inferring regular languages in polynomial update time. *Pattern Recognition and Image Analysis*, pages 49–61, 1992.

[36] P. Palaga, L. Nguyen, U. Leser, and J. Hakenberg. High-performance information extraction with AliBaba. In *EDBT*, pages 1140–1143, 2009.

[37] R. Ronen and O. Shmueli. SoQL: A language for querying and creating data in social networks. In *ICDE*, pages 1595–1602, 2009.

[38] S. Staworko and P. Wieczorek. Learning twig and path queries. In *ICDT*, pages 140–154, 2012.

[39] L. J. Stockmeyer and A. R. Meyer. Word problems requiring exponential time: Preliminary report. In *STOC*, pages 1–9, 1973.

[40] B. ten Cate, V. Dalmau, and P. G. Kolaitis. Learning schema mappings. *ACM Trans. Database Syst.*, 38(4):28, 2013.

[41] P. T. Wood. Query languages for graph databases. *SIGMOD Record*, 41(1):50–60, 2012.

[42] L.-L. Yan, R. J. Miller, L. M. Haas, and R. Fagin. Data-driven understanding and refinement of schema mappings. In *SIGMOD Conference*, pages 485–496, 2001.

[43] M. M. Zloof. Query by example. In *AFIPS National Computer Conference*, pages 431–438, 1975.

APPENDIX

A. OMITTED PROOFS

Section 3

Before presenting the proofs, we first formally define automata. A *nondeterministic finite word automaton* (*NFA*) A is a tuple $(Q, \Sigma, \delta, I, F)$, where Q is a finite set of states, $I \subseteq Q$ is the set of initial states, $F \subseteq Q$ is the set of final states, and $\delta : Q \times \Sigma \to 2^Q$ is the transition function. A *run* of A on a word $w = a_1 \dots a_n$ is a sequence of states $q_0 q_1 \dots q_n$ s.t. $q_i \in \delta(q_{i-1}, a_i)$ for $1 \leqslant i \leqslant n$ and $q_0 \in I$. A run on w is *accepting* if $q_n \in F$. A word w is *accepted* by A if there is an accepting run of A on w. By $L(A)$ we denote the set of words accepted by A. A *deterministic finite word automaton* (*DFA*) A is a NFA $(Q, \Sigma, \delta, I, F)$ s.t. I is a singleton, and $\delta(q, a)$ is either a singleton or the empty set (for each $q \in Q$ and $a \in \Sigma$).

Lemma 3.1 *Given a graph G and a sample S, S is consistent iff for every $\nu \in S_+$ it holds that $paths_G(\nu) \not\subseteq paths_G(S_-)$.*

Proof. For the *if* part, for $1 \leqslant i \leqslant |S_+|$, for ν_i in S_+, let p_i be the path witnessing $paths_G(\nu_i) \not\subseteq paths_G(S_-)$. Then, the query $p_1 + \ldots + p_{|S_+|}$ is consistent with S.

The *only if* part follows directly from the semantics of queries and the definition of consistency. ▫

Lemma 3.2 *Given a graph G and a sample S, deciding whether S is consistent is PSPACE-complete.*

Proof. The membership of the problem to PSPACE follows from Lemma 3.1 and the known result that deciding the inclusion of NFAs is PSPACE-complete [39].

Next, we show the PSPACE-hardness by reduction from the universality of the union problem for DFAs, known as PSPACE-complete [28], and by using a proof technique inspired by [4]. The reduction works as follows. Take n DFAs $D_1, \dots, D_n$ over Σ and two fresh symbols s_1, s_2 which are not in Σ. We construct a graph G as the disjoint union of $n+2$ graphs over $\Sigma \cup \{s_1, s_2\}$:

- For each DFA $D_i = (Q_i, \Sigma, \delta_i, I_i, F_i)$ (with $1 \leqslant i \leqslant n$), let $G_i = (V_i, E_i)$ s.t. $V_i = Q_i \cup \{\nu_i, \nu_i'\}$ and
$$E_i = \{(\nu, a, \nu') \mid \delta(\nu, a) = \nu'\} \cup \{(\nu_i, s_1, q) \mid q \in I_i\} \cup \{(q, s_2, \nu_i') \mid q \in F_i\},$$
- Let $G_{n+1} = (V_{n+1}, E_{n+1})$ s.t. $V_{n+1} = \{\nu_{n+1}, u_1\}$ and $E_{n+1} = \{(\nu_{n+1}, s_1, u_1)\} \cup \{(u_1, a, u_1) \mid a \in \Sigma\}$.
- Let $G_{n+2} = (V_{n+2}, E_{n+2})$ s.t. $V_{n+2} = \{\nu_{n+2}, u_2, \nu_{n+2}'\}$ and $E_{n+2} = \{(\nu_{n+2}, s_1, u_2), (u_2, s_2, \nu_{n+2}')\} \cup \{(u_2, a, u_2) \mid a \in \Sigma\}$.

Then, take the sample S s.t. $S_+ = \{\nu_{n+2}\}$ and $S_- = \{\nu_1, \dots, \nu_n, \nu_{n+1}\}$. We present in Figure 13 the graph and the sample that we construct by the described reduction.

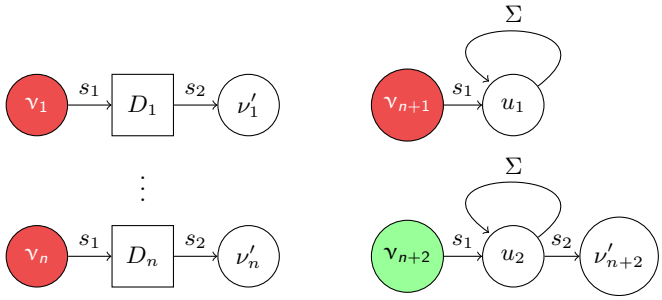


Figure 13: Constructed graph and sample.

We claim that S is consistent iff the union of the DFAs $D_1, \dots, D_n$ is not universal i.e., $\bigcup_{i=1}^{n} L(D_i) \neq \Sigma^*$.

For the *if* part, let $w \in \Sigma^*$ be the word witnessing the non-universality of the DFAs $D_1, \dots, D_n$. Then, take the query $q = s_1 \cdot w \cdot s_2$ and notice that $\nu_{n+2} \in q(G)$. Since $w \notin L(D_i)$, we infer that $\nu_i \notin q(G)$ (for $1 \leqslant i \leqslant n$). Moreover, since w ends with an s_2 and $s_2 \notin \Sigma$, we infer that $\nu_{n+1} \notin q(G)$. We conclude that q is a witness of the consistency of S.

For the *only if* part, the consistency of S implies by Lemma 3.1 that there exists a path $p \in paths_G(\nu_{n+2})$ s.t. $p \notin paths_G(S_-)$. Since $p \in paths_G(\nu_{n+2})$ and $p \notin paths_G(\nu_{n+1})$, we infer that p is of the form $s_1 \cdot w \cdot s_2$, with $w \in \Sigma^*$. Since $paths_G(\{\nu_1, \dots, \nu_n\})$ is the set of prefixes of all paths of the form $s_1 \cdot w' \cdot s_2$, where $w' \in L(D_1) \cup \ldots \cup L(D_n)$, we infer that $w \notin L(D_1) \cup \ldots \cup L(D_n)$, hence w is a witness of the non-universality of the DFAs $D_1, \dots, D_n$.

Note that the described reduction works in polynomial time. ▫

Lemma 3.3 *Given a graph G and a sample S, deciding whether there exists a query of the form $a_1 \cdot \ldots \cdot a_n$ (pairwise distinct symbols) consistent with S is NP-complete.*

Proof. To show the membership of the problem to NP, we point out that a non-deterministic Turing machine guesses a query q of the form $a_1 \cdot \ldots \cdot a_n$ with pairwise distinct symbols (hence of length bounded by $|\Sigma|$) and then checks whether q is consistent with S.

Next, we show the NP-hardness by reduction from 3SAT, known as NP-complete and by using a proof technique inspired by [4]. The reduction works as follows. Take a 3CNF formula $\varphi = C_1 \wedge \ldots \wedge C_k$ over the variables $x_1, \dots, x_n$. Take the alphabet $\Sigma = \{s_1, s_2, a_{11}, a_{12}, a_{13}, \dots, a_{k1}, a_{k2}, a_{k3}, \}$. Note that for $1 \leqslant i \leqslant k$, the labels a_{i1}, a_{i2}, a_{i3} *correspond* to the literals from the clause C_i on the position 1, 2, and 3, respectively. Next, construct the graph G as the disjoint union of at most $n+2$ graphs over Σ:

- Let $G_\varphi^+ = (V_\varphi^+, E_\varphi^+)$ be the graph that intuitively encodes the formula φ. Formally, $V_\varphi^+ = \{\nu_\varphi^+, u_1^+, \dots, u_{k+1}^+, \nu_\varphi'^+\}$ and $E_\varphi^+ = \{(\nu_\varphi^+, s_1, u_1^+), (u_{k+1}^+, s_2, \nu_\varphi'^+)\} \cup \{(u_i^+, a_{ij}, u_{i+1}^+) \mid 1 \leqslant i \leqslant k, 1 \leqslant j \leqslant 3\}$,

- Let $G_\varphi^- = (V_\varphi^-, E_\varphi^-)$ be the graph that intuitively forces any possible consistent query to end with an s_2. Formally, $V_\varphi^- = \{\nu_\varphi^-, u_1^-, \ldots, u_{k+1}^-\}$ and $E_\varphi^- = \{(\nu_\varphi^-, s_1, u_1^-)\} \cup \{(u_i^-, a_{ij}, u_{i+1}^-) \mid 1 \leqslant i \leqslant k, 1 \leqslant j \leqslant 3\}$,
- For $1 \leqslant i \leqslant n$, if the variable x_i appears in both positive and negative literals in clauses of φ, construct G_i as the graph that intuitively encodes the fact that a variable x_i cannot be assigned both the value true and false. If a variable x_i appears only in positive or only in negative literals, then such graphs are not of interest. Before defining G_i formally, let $T_i = \{a_{jl} \mid$ the clause C_j has on position l the positive literal $x_i\}$ and $F_i = \{a_{jl} \mid$ the clause C_j has on position l the negative literal $\neg x_i\}$. Next, let $G_i = \{V_i, E_i\}$ s.t. $V_i = \{\nu_{i1}, \ldots, \nu_{i5}\}$ and

$$\begin{aligned} E_i = \ & \{(\nu_{i1}, s_1, \nu_{i2})\} \cup \{(\nu_{i2}, a, \nu_{i2}) \mid a \in \Sigma \backslash \{s_2\} \backslash T_i \backslash F_i\} \cup \{(\nu_{i5}, a, \nu_{i5}) \mid a \in \Sigma\} \\ & \cup \{(\nu_{i2}, a, \nu_{i3}) \mid a \in F_i\} \cup \{(\nu_{i3}, a, \nu_{i3}) \mid a \in \Sigma \backslash \{s_2\} \backslash T_i\} \cup \{(\nu_{i3}, a, \nu_{i5}) \mid a \in T_i\} \\ & \cup \{(\nu_{i2}, a, \nu_{i4}) \mid a \in T_i\} \cup \{(\nu_{i4}, a, \nu_{i4}) \mid a \in \Sigma \backslash \{s_2\} \backslash F_i\} \cup \{(\nu_{i4}, a, \nu_{i5}) \mid a \in F_i\}. \end{aligned}$$

Finally, take the sample S_φ s.t. $S_{\varphi_+} = \{\nu_\varphi^+\}$ and $S_{\varphi_-} = \{\nu_\varphi^-\} \cup \{\nu_{i1} \mid 1 \leqslant i \leqslant n$ and x_i appears in both positive and negative literals in $\varphi\}$.

For example, we present in Figure 14 the graph and the sample constructed for the formula $\varphi_0 = (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_3 \vee \neg x_4)$. Notice that x_1 is the only variable that appears in both positive and negative literals in φ_0 hence we have constructed its corresponding subgraph and labeled the node ν_{11} as a negative example.

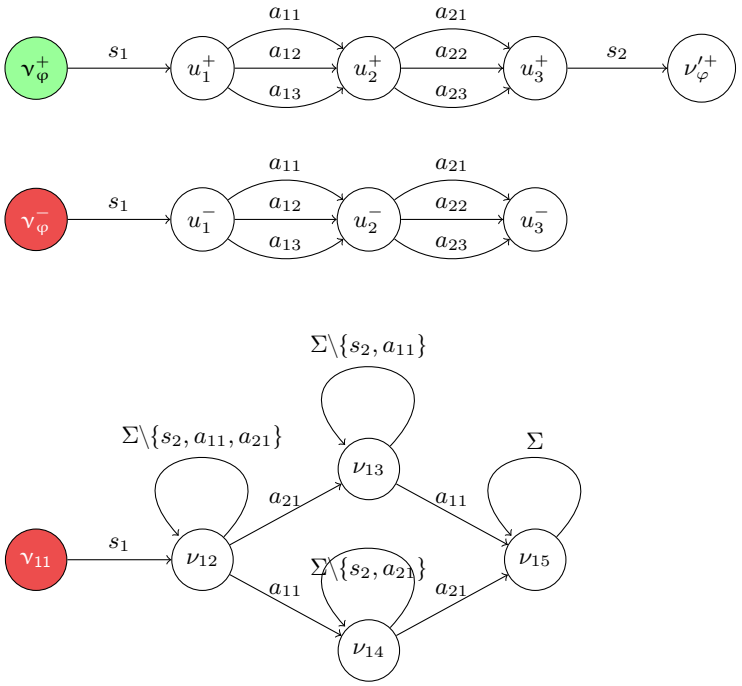


Figure 14: Graph and sample constructed for the formula $\varphi_0 = (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_3 \vee \neg x_4)$.

We claim that there exists a query of the form $a_1 \cdot \ldots \cdot a_n$ (with pairwise distinct symbols) consistent with S iff $\varphi \in$ 3SAT.

For the *if* part, take the valuation $v : \{x_1, \ldots, x_n\} \to \{true, false\}$ that makes φ satisfiable. Then, for each clause C_i (for $1 \leqslant i \leqslant k$) let a_{il_i} (with $1 \leqslant l_i \leqslant 3$) be the label corresponding to a literal satisfied by the valuation v. Take the query $q = s_1 \cdot a_{1l_1} \cdot \ldots \cdot a_{kl_k} \cdot s_2$. Clearly $\nu_\varphi^+ \in q(G)$. Since q encodes a valuation, none of the existing ν_{i1} (with $1 \leqslant i \leqslant n$) is in $q(G)$. Moreover, since q ends with s_2, we infer that $\nu_\varphi^- \notin q(G)$. We conclude that q is a query consistent with S, and indeed, it has the form of a path with pairwise distinct symbols.

For the *only if* part, take the query q consistent with S. Since $\nu_\varphi^+ \in q(G)$ and $\nu_\varphi^- \notin q(G)$, we infer that q has the form $s_1 \cdot a_{1l_1} \cdot \ldots \cdot a_{kl_k} \cdot s_2$, where each a_{il_i} (for $1 \leqslant i \leqslant k$) corresponds to a literal from the clause C_i. Moreover, since none of the existing ν_{i1} (with $1 \leqslant i \leqslant n$) is in $q(G)$, we infer that the path w encodes a valuation $v : \{x_1, \ldots, x_n\} \to \{true, false\}$. Since $\nu_\varphi^+ \in q(G)$, we conclude that the valuation v makes φ satisfiable.

Note that the described reduction works in polynomial time. □

□

Section 4

Given a graph G and a sample S, recall that by $\mathcal{C}(G, S)$ we denote the set of all queries consistent with S. Also recall that $\mathcal{C}(G, S)$ is not empty if the user has labeled the examples in S consistently with some goal query that she has in mind. Moreover, notice that for a consistent sample S and an unlabeled node ν from G, the two possible labels of ν split $\mathcal{C}(G, S)$ in two disjoint sets. Formally, we have the following.

Lemma A.1 *Given a graph G, a consistent sample S over G, and an unlabeled node ν from G, it holds that*

$$\mathcal{C}(G, S) = \mathcal{C}(G, S \cup \{(\nu, +)\}) \cup \mathcal{C}(G, S \cup \{(\nu, -)\}) \text{ and } \mathcal{C}(G, S \cup \{(\nu, +)\}) \cap \mathcal{C}(G, S \cup \{(\nu, -)\}) = \varnothing.$$

PROOF. The subset $\mathcal{C}(G, S \cup \{(\nu, +)\}) \subseteq \mathcal{C}(G, S)$ is the set of queries in $\mathcal{C}(G, S)$ that select ν while the subset $\mathcal{C}(G, S \cup \{(\nu, -)\}) \subseteq \mathcal{C}(G, S)$ is the set of queries in $\mathcal{C}(G, S)$ that do not select ν. Notice that the intersection of the two subsets is empty and their union is $\mathcal{C}(G, S)$. □

Lemma 4.1 *Given a sample S and a node ν from G:*

1. $\nu \in Cert_+(G, S)$ *iff there exists* $\nu' \in S_+$ *s.t.* $paths_G(\nu') \subseteq paths_G(S_-) \cup paths_G(\nu)$,
2. $\nu \in Cert_-(G, S)$ *iff* $paths_G(\nu) \subseteq paths_G(S_-)$.

PROOF. From Lemma A.1, we can derive that given a graph G, a consistent sample S over G, and an unlabeled node ν from G, ν is informative iff both $S \cup \{(\nu, +)\}$ and $S \cup \{(\nu, -)\}$ are consistent. Then, we can rewrite this equivalence and obtain a necessary and sufficient condition for a node to be certain i.e., ν is certain iff (i) $S \cup \{(\nu, +)\}$ is not consistent or (ii) $S \cup \{(\nu, -)\}$ is not consistent. From this observation and the characterization of a sample to be consistent (Lemma 3.1) we infer that the two parts of the Lemma are true. □

Lemma 4.2 *Given a graph G and a sample S, deciding whether a node ν is informative is PSPACE-complete.*

PROOF. The membership of the problem to PSPACE follows from Lemma 4.1 and the known result that deciding the inclusion of NFAs is PSPACE-complete [39].

For the PSPACE-hardness, take the same reduction from the proof of Lemma 3.2, the only difference being that the node ν_{n+2} is unlabeled (i.e., it is no more a positive example). Hence, we have the same graph, the same set of negative examples $S_- = \{\nu_1, \ldots, \nu_n, \nu_{n+1}\}$ and an empty set of positive examples $S_+ = \varnothing$. We claim that the union of the DFAs $D_1, \ldots, D_n$ is not universal iff ν_{n+2} is informative. From Lemma A.1, we can derive that this is equivalent to saying that the union of the DFAs $D_1, \ldots, D_n$ is not universal iff both $S \cup \{(\nu_{n+2}, +)\}$ and $S \cup \{(\nu_{n+2}, -)\}$ are consistent. Notice that $S \cup \{(\nu_{n+2}, -)\}$ is clearly consistent since any query a (where $a \in \Sigma$) is consistent with it. Thus, we have to prove that $S \cup \{(\nu_{n+2}, +)\}$ is consistent iff the union of the DFAs $D_1, \ldots, D_n$ is not universal, which follows exactly as in the proof of Lemma 3.2. □

B. LEARNING ALGORITHMS FOR BINARY AND n-ARY SEMANTICS

In Section 2, we have defined the class of queries that select the nodes having at least one path in the language of a given regular expression. This kind of semantics is usually called *monadic* since it selects single nodes of the graph. The results shown in the main body of the paper for monadic semantics are directly applicable to the case of binary and n-ary semantics. Here, we present the learning algorithms for binary and n-ary semantics (Algorithm 2 and Algorithm 3, respectively). First, let us introduce some auxiliary notions.

Given a graph $G = (V, E)$ and two nodes ν, ν' from V, by $paths^2_G(\nu, \nu')$ we denote the *set of all paths between the nodes ν and ν'*. Formally, a word $w = a_1 \ldots a_n$ belongs to $paths^2_G(\nu, \nu')$ if there exists a sequence of nodes $\nu\nu_1 \ldots \nu_{n-1}\nu'$ such that the triples (ν, a_1, ν_1), (ν_{i-1}, a_i, ν_i) (with $2 \leqslant i \leqslant n-1$), and (ν_{n-1}, a_n, ν') belong to E. We naturally extend the notion of paths between two nodes to a set of pairs of nodes $X \subseteq V \times V$ i.e., $paths^2_G(X) = \bigcup_{(\nu,\nu') \in X} paths^2_G(\nu, \nu')$.

An *n-ary path query* Q is a sequence of regular expressions $(q_1, \ldots, q_{n-1})$. Given a graph $G = (V, E)$ and an n-ary path query $Q = (q_1, \ldots, q_{n-1})$, the set of *tuples of nodes of G selected by Q*, denoted $Q(G)$ is as follows.

$$Q(G) = \{(\nu_1, \ldots, \nu_n) \mid \forall 1 \leqslant i \leqslant n-1.\ paths^2_G(\nu_i, \nu_{i+1}) \cap L(q_i) \neq \varnothing\}.$$

By NPQ we denote the set of all n-ary path queries. We may refer to them simply as n-ary queries.

When $n = 2$, we have the particular case of binary semantics. In such a case, a query Q consists of a single regular expression q and selects the pairs of nodes of the graph that are linked via a path in the language of q. An example for learning is now a pair of nodes (ν, ν') labeled with $+$ or $-$. To learn a binary query, the only change to Algorithm 1 (cf. Section 3.2) is that each positive example implies a smaller set of candidate paths from which we have to choose a consistent one (since the destination node is also known). Algorithm 2 illustrates this idea.

Algorithm 2 Learning algorithm for binary semantics – $learner^2(G, S)$.

Input: graph G, sample S
Output: query q consistent with S or *null*
Parameter: fixed $k \in \mathbb{N}$ //maximal length of a SCP

```
for (ν, ν') ∈ S+. ∃p ∈ Σ^{⩽k}. p ∈ paths^2_G(ν, ν')\paths^2_G(S−) do
    P := P ∪ {min_⩽(paths^2_G(ν, ν')\paths^2_G(S−))}
let A be the prefix tree acceptor for P
while ∃s, s' ∈ A. L(A_{s'→s}) ∩ paths^2_G(S−) = ∅ do
    A := A_{s'→s}
if ∀(ν, ν') ∈ S+. L(A) ∩ paths^2_G(ν, ν') ≠ ∅ then
    return query Q = (q) where q is represented by the DFA A
return null
```

For the n-ary case, an example is $((\nu_1, \ldots, \nu_n), \alpha)$ where $\alpha \in \{+, -\}$ and $(\nu_1, \ldots, \nu_n)$ is a tuple of nodes of the graph. We have simply to apply the previous algorithm to learn a query for each position in the tuple and then to combine those. Algorithm 3 illustrates this idea.

Let $\text{NPQ}^{\leqslant s}$ be the subset of NPQ consisting of the n-ary path queries Q s.t. the maximal size of a regular expression in Q is s (recall that the size is the number of states in its canonical DFA). Finally, using Algorithm 3 and the same proof technique as for the learnability of (monadic) path queries (i.e., Theorem 3.5), we can state the following result.

Corollary B.1 *The query class $\text{NPQ}^{\leqslant s}$ is learnable with abstain in polynomial time and data, using the algorithm $learner^n$ with the parameter k set to $2 \times s + 1$.*

Algorithm 3 Learning algorithm for n-ary semantics – $learner^n(G, S)$.

Input: graph G, sample S
Output: query q consistent with S or *null*
Parameter: fixed $k \in \mathbb{N}$ //maximal length of a SCP

1: **for** $i \in \{1, \ldots, n-1\}$ **do**
2: **let** $S^i_+ = \{(\nu_i, \nu_{i+1}) \mid \forall(\nu_1, \ldots, \nu_n) \in S_+)\}$
3: **let** $S^i_- = \{(\nu_i, \nu_{i+1}) \mid \forall(\nu_1, \ldots, \nu_n) \in S_-)\}$
4: **let** $q_i = learner^2(G, S^i)$
5: **if** q_i is *null* **then**
6: **return** *null*
7: **return** query $Q = (q_1, \ldots, q_{n-1})$