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 (N₁ to N₆), along with cinemas (C₁ and C₂), and restaurants (R₁ and R₂) 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 N₂ and N₃, that in the neighborhood N₄ exists a cinema C₁, and so on.

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:

\[ q = (\text{tram} + \text{bus})^* \cdot \text{cinema} \]

The query q selects the nodes N₁, N₂, N₄, and N₆ as they are entailed by the following paths in the graph:

\[ N₁ \xrightarrow{\text{tram}} N₄ \xrightarrow{\text{cinema}} C₁, \]
\[ N₂ \xrightarrow{\text{bus}} N₄ \xrightarrow{\text{tram}} N₄ \xrightarrow{\text{cinema}} C₁, \]
\[ N₄ \xrightarrow{\text{cinema}} C₁, \]
\[ N₆ \xrightarrow{\text{cinema}} C₂. \]

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-
hhibit 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:

$$N_2 \xrightarrow{bus} N_3 \xrightarrow{trans} N_4 \xrightarrow{cinema} C_1,$$

$$N_6 \xrightarrow{cinema} C_2.$$

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:

$$\text{ProteinPurification} \cdot \text{ProteinSeparation}^* \cdot \text{MassSpectrometry}.$$  

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 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. 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 $\Sigma$ is a finite, ordered set of symbols. A word over $\Sigma$ is a sequence $a_1 \ldots a_n$ of symbols from $\Sigma$. By $|w|$ we denote the length of a word $w$. The concatenation of two words $w_1 = a_1 \ldots a_n$ and $w_2 = a_{n+1} \ldots a_m$ is $w_1 w_2 = a_1 \ldots a_m$. 

1. http://www.w3.org/TR/sparql11-query/
Graph databases. A graph database is a finite, directed, edge-labeled graph [7, 41]. Formally, a graph (database) $G$ over an alphabet $\Sigma$ 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 $(\nu_0, a, \nu_1) \in V \times \Sigma \times V$, where $\nu_0$ is the origin of the edge, $\nu_1$ is the end of the edge, and $a$ is the label of the edge. We often abuse notation and write $\nu \in G$ and $(\nu_0, a, \nu_1) \in G$ instead of $\nu \in V$ and $(\nu_0, a, \nu_1) \in E$, 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 $G_0$.](image)

Paths. A word $w = a_1 \ldots a_n$ matches a sequence of nodes $\nu_0 \nu_1 \ldots \nu_n$ if, for each $1 \leq i \leq n$, the triple $(\nu_{i-1}, a_i, \nu_i)$ is an edge in $G$. For example, for the graph $G_0$ in Figure 3 we have paths$_G(\nu_1) = \{\epsilon, a, b, c\}$. Note that $\epsilon \in$ paths$_G(\nu)$ 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 $\nu$. In the sequel, we restrict ourselves to such words as paths, and moreover, we say that a path $w$ is covered by a node $\nu$ if $w \in$ paths$_G(\nu)$. Paths are ordered using the canonical order $\leq$. For example, for the graph $G_0$ in Figure 3 we have paths$_{G_0}(\nu_1) = \{\epsilon, a, b, c\}$. Note that $\epsilon \in$ paths$_{G_0}(\nu)$ for every $\nu \in G$. Moreover, note that paths$_{G_0}(\nu)$ is finite if there is no cycle reachable from $\nu$. 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$_G(X) = \bigcup_{x \in X}$ paths$_G(x)$.

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

$q := \epsilon \mid a \mid (a \in \Sigma) \mid q_1 + q_2 \mid q_1 \cdot q_2 \mid q^*.$

where $\cdot$ denotes 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 $c, abc, ababc, \ldots$. 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 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 $(a \cdot b)^*c$.](image)

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 $G$:

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

For example, given the graph $G_0$ in Figure 3, the query $a$ selects all nodes except $\nu_4$, the query $(a \cdot b)^* \cdot c$ selects the nodes $\nu_1$ and $\nu_3$, 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 $P_Q$ 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 $ab \ldots b$ 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 $q \cdot (q' + \epsilon)$, where $q'$ can be every $P_Q$. 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 unreasability 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 $ν ∈ V$ and $α ∈ \{+, −\}$. We say that an example of the form $(ν, +)$ is a positive example while an example of the form $(ν, −)$ is a negative example. A sample $S$ is a set of examples i.e., a subset of $V × \{+, −\}$. Given a sample $S$, we denote the set of positive examples $\{ν ∈ V \mid (ν, +) ∈ S\}$ by $S_P$ and the set of negative examples $\{ν ∈ V \mid (ν, −) ∈ S\}$ by $S_N$. 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. $S_P ⊆ q(G)$ and $S_N ∩ q(G) = ∅$. 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. $S_1 = \{ν_1, ν_3\}$ and $S_2 = \{ν_3, ν_2\}$; $S$ is consistent because there exist queries like $(a · b)^* · c$ or $c + (a · b · c)$ 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 in 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 $ν ∈ S_P$, it holds that $\text{paths}_C(ν) \nsubseteq \text{paths}_S(S_N)$.

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(1)” in [4].

Lemma 3.3 Given a graph $G$ and a sample $S$, deciding whether there exists a query of the form $a_1 · · · a_n$ (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 $|Σ|$) 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 Lemmas 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 $Q$ 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 $Q$ that is consistent with $S$, or null if no such query exists or it cannot be constructed efficiently.

2. Complete. For every query $q ∈ Q$, 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., $CS ⊇ S$ 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 RPNJ [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 $\leq$) 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. $\leq$) consistent paths for each node. As an example, for the graph $G_0$ in Figure 3 and a sample $s.t. S_+ = \{v_1, v_3\}$ and $S_- = \{v_2, v_7\}$, we obtain the SCPs $a b c$ for $v_1$ and $c$ for $v_3$, 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 $pq$, 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 $\nu \in S_+$, $\exists p \in S^\leq p \in \text{paths}_{G}(\nu)$ \text{paths}_{G}(S_-) do
2: $P := P \cup (\min_{\nu}(\text{paths}_{G}(\nu) \cap \text{paths}_{G}(S_-)))$
3: let $A$ be the prefix tree acceptor for $P$
4: while $\exists s, s' \in A. L(A_{s \rightarrow s'}) \cap \text{paths}_{G}(S_-) = \emptyset$ do
5: $A := A_{s \rightarrow s}$
6: if $\forall \nu \in S_+. L(A) \cap \text{paths}_{G}(\nu) \neq \emptyset$ then
7: 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 $s.t. S_+ = \{v_1, v_3\}$ and $S_- = \{v_2, v_7\}$. 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 $\Sigma^k$ 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 $A_{s \rightarrow s'}$, 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 $c$ and $a$ cannot be merged (because the obtained DFA would select the path $bc$ that is covered by the negative $v_2$), the states $\varepsilon$ and $c$ cannot be merged (because the path $\varepsilon$ is covered by both negatives), while the states $\varepsilon$ and $ab$ can be merged without covering any negative example. On our example, we obtain the query 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 $\{ab\} \ast \cdot c$ selects all positive nodes hence is returned.

![Figure 5: A graph with an inconsistent sample.](image-url)
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 \( v \in CS_+ \) s.t. \( p = \min_\leq \min_\leq (L(q) \cap \text{paths}_{G}(v)) \), (ii) for each \( p \in P_- \), a node \( v \in CS_- \) s.t. \( p \in \text{paths}_{G}(v) \), and (iii) for each \( p' \) that is smaller (w.r.t. the canonical order \( \leq \)) than a word \( p \in P_+ \), and is not prefixed by any word in \( L(q) \), a node \( v \in CS_- \) s.t. \( p' \in \text{paths}_{G}(v) \). For our query \((a \cdot b)^*c\), (i) implies two positive nodes: a node \( v \) s.t. \( c \in \text{paths}_{G}(v) \) and another node \( v' \) s.t. \( \varepsilon \in \text{paths}_{G}(v') \) and \( c \notin \text{paths}_{G}(v') \), while (ii) and (iii) imply a node \( v'' \) s.t. \( v'' \notin q(G) \) and \( \{\varepsilon, a, b, ac, bc\} \subseteq \text{paths}_{G}(v'') \) (cf. ii) and \( \{\varepsilon, a, b, ba, ab, ac, bb, bc, aac, aab, abc, abb\} \subseteq \text{paths}_{G}(v'') \) (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 \( |q| \), 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 \text{learner} (assuming the presence of a characteristic sample), we need the parameter \( k \) of \text{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. \( \square \)

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. \( S_+ = \{v_1, v_2\} \) and \( S_- = \{v_3, v_4\} \) on the graph in Figure 3) is characteristic for \((a \cdot b)^*c\) and all above mentioned negatives paths are covered by two negative nodes.
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 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 $\Theta$ 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. The user visualizes the neighborhood of a given node $\nu$ and labels $\nu$ 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 $G$:

$$C(G, S) = \{ q \in PQ \mid S_+ \subseteq q(G) \land S_- \cap q(G) = \emptyset \}.$$ 

Assuming that the user labels the nodes consistently with some goal query $q$, the set $C(G, S)$ always contains $q$. Initially, $S = \emptyset$ and $C(G, S) = PQ$. Therefore, an ideal strategy of presenting nodes to the user is to get us quickly from $S = \emptyset$ to a sample $S$ s.t. $C(G, S) = \{ q \}$. 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 $\nu$ is certain (w.r.t. $S$) if it belongs to one of the following sets:

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

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

In other words, a node is certain with a label $\alpha$ if labeling it explicitly with $\alpha$ does not eliminate any query from $C(G, S)$. For instance, take the graph in Figure 10 with a positive, a negative, and an unlabeled node $\nu$ in $G$, we say that $\nu$ is certain (w.r.t. $S$) if it belongs to one of the following sets:

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

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

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 $\nu$ from $G$:

1. $\nu \in Cert_+(G, S)$ iff there exists $\nu' \in S_+$ s.t. $\text{paths}_G(\nu') \subseteq \text{paths}_G(S_+) \cup \text{paths}_G(\nu),$

2. $\nu \in Cert_-(G, S)$ iff $\text{paths}_G(\nu) \subseteq \text{paths}_G(S_-)$.

Additionally, given a graph $G$, a sample $S$, and a node $\nu$, we say that $\nu$ is informative (w.r.t. $S$) if it 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 $\nu$ 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 $K$ consists of taking randomly a $k$-informative node while strategy $K_S$ 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 x 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,
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 $bio_2$, 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 $syn_3$ and $syn_4$), 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 ($bio_2$), 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.

<table>
<thead>
<tr>
<th>Query</th>
<th>Selectivity</th>
</tr>
</thead>
<tbody>
<tr>
<td>$bio_1$</td>
<td>$b \cdot A \cdot A^*$</td>
</tr>
<tr>
<td>$bio_2$</td>
<td>$C \cdot C^* \cdot a \cdot A \cdot A^*$</td>
</tr>
<tr>
<td>$bio_3$</td>
<td>$C \cdot E$</td>
</tr>
<tr>
<td>$bio_4$</td>
<td>$I \cdot I^* \cdot I^*$</td>
</tr>
<tr>
<td>$bio_5$</td>
<td>$A \cdot A \cdot A^* \cdot I \cdot I \cdot I^*$</td>
</tr>
<tr>
<td>$bio_6$</td>
<td>$A \cdot A \cdot A^*$</td>
</tr>
</tbody>
</table>

**Table 1: Biological queries.**
Figure 11: Summary of static experiments – F1 score.

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

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Bio query</th>
<th>Labels needed for F1 score = 1 without interactions</th>
<th>Interactive strategy</th>
<th>Labels needed for F1 score = 1 with interactions</th>
<th>Time between interactions (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Biological queries</td>
<td>bio1</td>
<td>7%</td>
<td>kR</td>
<td>0.06%</td>
<td>0.19</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>kS</td>
<td>0.06%</td>
<td>0.19</td>
</tr>
<tr>
<td></td>
<td>bio2</td>
<td>7%</td>
<td>kR</td>
<td>1.78%</td>
<td>0.26</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>kS</td>
<td>3.13%</td>
<td>0.48</td>
</tr>
<tr>
<td></td>
<td>bio3</td>
<td>66%</td>
<td>kR</td>
<td>1.24%</td>
<td>0.34</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>kS</td>
<td>1.49%</td>
<td>0.45</td>
</tr>
<tr>
<td></td>
<td>bio4</td>
<td>12%</td>
<td>kR</td>
<td>1.32%</td>
<td>0.23</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>kS</td>
<td>0.22%</td>
<td>0.53</td>
</tr>
<tr>
<td></td>
<td>bio5</td>
<td>87%</td>
<td>kR</td>
<td>7.7%</td>
<td>3.45</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>kS</td>
<td>7.39%</td>
<td>3.79</td>
</tr>
<tr>
<td></td>
<td>bio6</td>
<td>12%</td>
<td>kR</td>
<td>1.18%</td>
<td>0.24</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>kS</td>
<td>0.35%</td>
<td>0.3</td>
</tr>
<tr>
<td>Synthetic query syn1</td>
<td>10000</td>
<td>51%</td>
<td>kR</td>
<td>0.15%</td>
<td>1.33</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>kS</td>
<td>0.17%</td>
<td>1.35</td>
</tr>
<tr>
<td></td>
<td>20000</td>
<td>26%</td>
<td>kR</td>
<td>0.07%</td>
<td>5.83</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>kS</td>
<td>0.06%</td>
<td>5.92</td>
</tr>
<tr>
<td></td>
<td>30000</td>
<td>22%</td>
<td>kR</td>
<td>0.04%</td>
<td>13.5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>kS</td>
<td>0.04%</td>
<td>13.95</td>
</tr>
<tr>
<td>Synthetic query syn2</td>
<td>10000</td>
<td>20%</td>
<td>kR</td>
<td>0.38%</td>
<td>1.57</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>kS</td>
<td>0.36%</td>
<td>1.58</td>
</tr>
<tr>
<td></td>
<td>20000</td>
<td>11%</td>
<td>kR</td>
<td>0.23%</td>
<td>6.63</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>kS</td>
<td>0.22%</td>
<td>6.78</td>
</tr>
<tr>
<td></td>
<td>30000</td>
<td>8%</td>
<td>kR</td>
<td>0.17%</td>
<td>15.24</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>kS</td>
<td>0.16%</td>
<td>15.38</td>
</tr>
<tr>
<td>Synthetic query syn3</td>
<td>10000</td>
<td>5%</td>
<td>kR</td>
<td>0.1%</td>
<td>1.32</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>kS</td>
<td>0.1%</td>
<td>1.32</td>
</tr>
<tr>
<td></td>
<td>20000</td>
<td>3%</td>
<td>kR</td>
<td>0.05%</td>
<td>5.66</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>kS</td>
<td>0.05%</td>
<td>5.68</td>
</tr>
<tr>
<td></td>
<td>30000</td>
<td>2%</td>
<td>kR</td>
<td>0.04%</td>
<td>13.15</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>kS</td>
<td>0.04%</td>
<td>13.41</td>
</tr>
</tbody>
</table>

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