Learning Tree Patterns from Example Graphs

Sara Cohen and Yaacov Y. Weiss

Rachel and Selim Benin School of Computer Science and Engineering
Hebrew University of Jerusalem
Jerusalem, Israel
{sara,yyweiss}@cs.huji.ac.il

Abstract

This paper investigates the problem of learning tree patterns that return nodes with a given set of labels, from example graphs provided by the user. Example graphs are annotated by the user as being either *positive* or *negative*. The goal is then to determine whether there exists a tree pattern returning tuples of nodes with the given labels in each of the positive examples, but in none of the negative examples, and, furthermore, to find one such pattern if it exists. These are called the *satisfiability* and *learning* problems, respectively.

This paper thoroughly investigates the satisfiability and learning problems in a variety of settings. In particular, we consider example sets that (1) may contain only positive examples, or both positive and negative examples, (2) may contain directed or undirected graphs, and (3) may have multiple occurrences of labels or be uniquely labeled (to some degree). In addition, we consider tree patterns of different types that can allow, or prohibit, wildcard labeled nodes and descendant edges. We also consider two different semantics for mapping tree patterns to graphs. The complexity of satisfiability is determined for the different combinations of settings. For cases in which satisfiability is polynomial, it is also shown that learning is polynomial. (This is non-trivial as satisfying patterns may be exponential in size.) Finally, the *minimal learning* problem, i.e., that of finding a minimal-sized satisfying pattern, is studied for cases in which satisfiability is polynomial.

1998 ACM Subject Classification H.2.3 Query languages, I.2.6 Learning

Keywords and phrases tree patterns, learning, examples

Digital Object Identifier 10.4230/LIPIcs.ICDT.2015.127

1 Introduction

Node-labeled graphs (or simply *graphs*, for short) are a popular data model and are useful in describing information about entities and their relationships. Traditional query languages for graphs are often based on *pattern matching*, i.e., the query is a pattern with a tuple of annotated nodes, called *output nodes*. Query results are simply tuples of nodes that are in the image of the output nodes with respect to some mapping from the query to the graph. The precise properties required of such a mapping depend on the setting (they may be required to be injective, they may allow edges to be mapped to paths, etc.).

Formulating a query over a set of graphs that returns specific tuples of nodes can be a difficult task. The graphs may be intricate, amalgamating many different relationships among nodes into a single structure. Finding a precise query that returns exactly the tuples of nodes of interest to the user may require extensive investigation into the precise structure of the graphs, and thus, may be infeasible for the user.

In this paper we study the problem of learning tree patterns from a given set of graphs with annotated tuples of interest. If tree patterns can be learned, then query formulation for

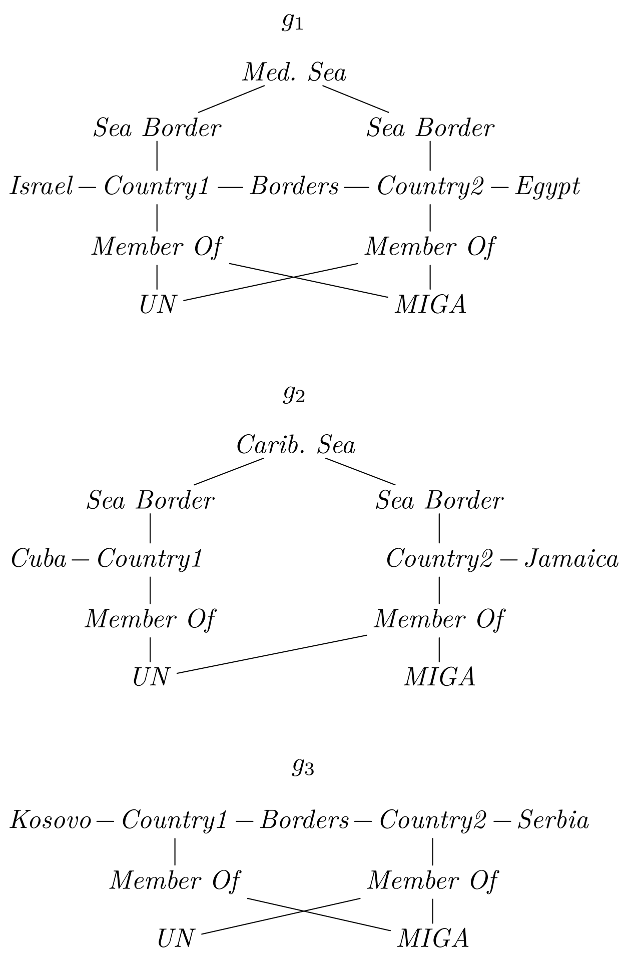

Figure 1 Motivating example with data from the CIA fact-book.

the user can be simplified, by having the user simply provide examples, while the system infers a satisfying pattern.

The ability to learn satisfying patterns is also of interest in other scenarios. For example, it allows certain and possible answers to be identified and returned [7]. Intuitively, using the terminology of [7], a certain (resp. possible) answer is one that is returned by all (resp. at least one) patterns satisfying the given examples. It has been shown in [7] that the ability to determine satisfiability of a set of examples immediately implies the ability to determine whether an answer is certain or possible. Thus, even without learning the user's precise query, if satisfiability can be determined, then the system can provide the user with answers that will (certainly or possibly) be of interest.

▶ **Example 1.** Consider graphs g_1, g_2, g_3 in Figure 1. These graphs depict a small portion of the information from the CIA fact-book, a labeled graph data source containing information about countries around the world, and their relationships. Thus, for example, g_1 describes the relationships between Israel and Egypt. These countries border each other, and both border the Mediterranean Sea. In addition, both Israel and Egypt are members of the UN and of MIGA. Similarly, g_2 contains information about the relationships of Cuba and

Jamaica. Given both g_1 and g_2 as positive examples, one possible pattern that can be inferred as holding between Country1 and Country2 is p_2, i.e., that they both border on some common sea. Another possible pattern that can be inferred as holding between Country1 and Country2 is that derived by replacing "UN" in p_3 (of Figure 1) by a wildcard. We call this pattern p_4.

The goals of this paper are to *(1)* determine whether there exists a satisfying pattern for a set of examples (e.g., "yes" for g_1, g_2), and to *(2)* develop methods to automatically find a (minimal) satisfying pattern (e.g., p_2, p_4). As explained above, a by-product of these results is the ability to determine whether a new (unlabeled) example is certainly, or possibly, of interest to the user (e.g., g_3 is possibly of interest, as the user may desire countries that are members of the same organization, and this holds for each of g_1, g_2, g_3).

The setting considered in this paper is as follows. We are given a set of graphs G, containing both *positive examples* g^+ and *negative examples* g^-. In addition, we are given a tuple of *distinguished labels* $l_1, \ldots, l_n$. Depending on the setting, these labels may uniquely identify a tuple of nodes in each graph of G, or may correspond to many different tuples of nodes. In the former case, we say that G is *output identifying*, and in the latter, G is *arbitrary*. Additional features of interest of the dataset $\Delta = (G, (l_1, \ldots, l_n))$ include *(1)* whether the graphs are *uniquely labeled*, in which case no labels appear more than once in a graph, *(2)* whether the graphs are directed or undirected, and *(3)* whether there are only positive examples, or both positive and negative examples.

We consider tree patterns that may contain (or prohibit) nodes with wildcards and/or descendant edges. In addition, a tree pattern contains a designated tuple of *output nodes*. Different semantics are defined for mapping tree patterns to graphs, depending on whether we require such mappings to be injective or not. Intuitively, a given tree pattern satisfies a positive (resp. negative) example, if there exists (resp. does not exist) a mapping from the tree pattern to the graph that maps the output nodes to nodes with the distinguished labels.

Three problems of interest are defined and studied. First, the *satisfiability problem* is to determine whether there exists a pattern satisfying a given dataset Δ. Second, the *learning problem* is to find a pattern satisfying Δ, if one exists. Note that even if satisfiability is in polynomial time, learning may still be difficult as satisfying patterns may be exponential in size. Third, the *minimal learning problem* is to find a pattern satisfying Δ, that is of minimal size. The complexity of these problems varies greatly depending on the given setting, i.e., characteristics of the dataset, the type of patterns allowed, and the type of mappings required from the patterns to the dataset. In some cases these problems are quite easy (i.e., in polynomial time), while in other cases even satisfiability is PSPACE-complete.

The main contributions of this paper are a thorough study of the three problems of interest, in different combinations of settings. Our results are quite extensive and are summarized in Table 1 for datasets with only positive examples, in Table 2 for datasets with both positive and negative examples, and in Table 3 for the minimal learning problem. Thus, our work clearly delineates in which cases learning is efficient, and thereby, provides the groundwork for a system which induces tree patterns from user examples.

2 Related Work

While the problem at hand has not been studied in the past, there is significant previous related work, falling into three general categories: learning queries from examples, graph mining and explaining query results. We discuss relevant work in each of these areas.

Learning Queries from Examples. There is a wealth of work on using machine learning techniques to learn XPath queries or tree patterns from positive and negative examples. Most often, these works are in the context of wrapper induction.

The goal of [5, 2, 4, 1, 17, 22, 21, 23] is to learn a tree-like query, usually using equivalence or membership questions. Intuitively, equivalence questions provide the user with a query (or examples of results of a query) and ask the user if the computer-provided query is equivalent to that of the user. Membership questions provide the user with an example, and ask the user whether this example should be marked positively. Many negative results have been shown, most notably, that tree automata cannot be learned in polynomial time with equivalence questions [2], as well as a similar negative result for learning even simple fragments of XPath [5]. Several of these works [17, 22] have been experimentally tested against wrapper induction benchmarks, and have been shown to work extremely well. Among these works, only [21] considers learning trees from graphs (but their setting differs significantly from that considered here, e.g., graphs contain edge variables that can be replaced by additional graphs, and the focus is on equivalence or subset queries). The above results are completely different from (and complementary to) those of ours. Most significantly, we do not ask the user questions of any type. The goal in this paper is quite simple: determine whether there exists any satisfying pattern, and find one if possible.

Somewhat slightly related to our work is the definability problem for graph query languages, studied in [3]. The goal in [3] is to determine whether a given relation over nodes in a graph can be defined by a query in a given language. The language considered differs significantly from ours, as they allow regular expressions and non-tree patterns. We note that they also have only positive examples (while all other examples are implicitly negative, and thus, there are no "non-labeled" examples), and they do not consider the problem of learning minimal sized satisfying patterns.

In [7] the problem of learning tree patterns from tree examples was studied. It was also shown that the ability to efficiently determine satisfiability is sufficient in order to efficiently determine whether a new unlabeled example can possibly be positive (i.e., there exists a satisfying pattern returning this answer), or is certainly positive (i.e., all satisfying patterns return this answer). While [7] showed this result in the context of tree data, it carries over to our framework.

Our paper differs very significantly from [7] – in its setting, scope and techniques. In [7], only tree data was considered (i.e., all examples are trees), trees are always directed and arbitrarily labeled, the width of the labels is exactly two (and always includes the root of each example) and the number of examples is constant. In the current paper, we consider a large number of different scenarios, all of which involve graph examples. Graph examples are a much richer formalism than tree examples. This is true even though our patterns are trees, as patterns must now choose between exponentially many alternative paths in a graph, and can traverse a cycle arbitrarily many times. The current paper presents comprehensive results on the many different possible settings (whereas [7] mostly focused on a single polynomial case). Thus, the current paper is significantly larger in scope than [7]. We note also that the techniques used in [7] are not applicable for the problems in the current paper as they were heavily based on the fact that the data in all examples are trees (and on the additional restrictions considered there), and do not carry over to a graph setting.

Graph Mining. Given a set of n graphs, the graph mining problem is to find subgraphs that are common to at least k of these graphs, where k is a user provided parameter. One commonly considered flavor of this problem is when the frequent subgraphs are trees. See [14] for a survey on frequent subgraph mining and [15] for recent theoretical results on the complexity of frequent subgraph mining.

The results of this paper are closely related to the frequent subtree mining problem, for the special case where $k = n$, i.e., when the subtree must appear in all graphs. In particular, when there are only positive examples, and patterns cannot contain descendant edges, then learning tree patterns is quite similar in spirit to subtree mining. However, in our setting, the user also provides a tuple of distinguished labels that must be "covered" by the satisfying patterns. This significantly differs from subtree mining, in which no constraints require that specific subportions of the graph appear in any frequent subtree. It also makes determining whether there exists a satisfying pattern significantly harder. For example, it is always easy to find a single frequent subtree [15]; however, finding a satisfying pattern is often intractable.

Explaining Query Answers. Recently, there has been considerable interest in explaining query results to a user, i.e., in justifying why certain tuples do, or do not, appear in a query result. Most of this work, e.g., [6, 13, 19, 11], assumes that the query is given. Answers take the form of provenance of tuples, or explanations as to which query clause caused tuples to be eliminated. This is very different from our framework in which only the answers are given, and not the queries.

On the other hand [26, 25, 8] explain query answers, by synthesizing a query from the result. In [26], the focus is on generating a query that almost returns a given output (with a precise definition for "almost"). In [25] missing tuples in a query result are explained by generating a query that returns the query result, as well as the missing tuples. Finally, in [8] the problem of synthesizing view definitions from data is studied. These works consider relational databases and conjunctive queries, and the results are not immediately applicable to other settings. However, the results in this paper can immediately be applied to the problem of explaining query answers, for tree patterns over graph datasets. In particular, we show in this paper how to synthesize (i.e., learn) a tree pattern from a set of examples.

3 Definitions

In this section, we present basic terminology used throughout the paper and formally define the problems if interest.

Graphs and Examples. In this paper we consider labeled graphs that may be directed or undirected. We use Σ to denote an infinite set of labels. Given a graph g, we use V_g to denote the nodes of g, E_g to denote the edges of g and $l_g : V_g \to \Sigma$ to denote the *labeling function* that associates each node in g with a label from Σ.

We say that l is a *unique label* in g if there is precisely one node v for which $l_g(v) = l$. We say that a graph g is *uniquely labeled* if, for all $u \neq v \in V_g$ it holds that $l_g(u) \neq l_g(v)$.

A graph g, annotated with the superscript "+" or "−" is called an *example*. We say that g^+ is a *positive example* and g^- is a *negative example*. We use Δ to denote a pair $(G, \bar{l})$ where G is a set of examples, and $\bar{l} = (l_1, \ldots, l_n)$ is a tuple of distinct labels. We call Δ a *dataset* and $\bar{l}$ the *distinguished labels*.

We distinguish several special types of datasets. In particular, Δ is *positive* if G contains only positive examples, Δ is *uniquely labeled* if all graphs in G are uniquely labeled, and Δ is *output identifying* if, for all $i \leq n$, we have that l_i is a unique label in all graphs in G. Note that if Δ is uniquely labeled, then it is also output identifying, but the opposite does not hold in general. We require Δ to contain only directed or only undirected graphs (and not a mix of the two). In the former case, we say that Δ is *directed* and in the latter, we say that Δ is *undirected.*

▶ Remark. Recall that the goal of this paper is to learn patterns connecting a series of labels in a graph. The special case of output identifying datasets is of particular interest, as it is equivalent to the following problem: We are given a set of graphs, each of which has a tuple of distinguished nodes. The goal is to find a pattern that returns the given tuples of nodes from the positive examples, and does not return the tuples of nodes from the negative examples. In other words, learning patterns from output-identifying datasets is the problem of learning patterns connecting a given series of nodes from each example graph.

▶ **Example 2.** Consider the graphs g_1, g_2, g_3 of Figure 1. The dataset Δ, defined as $(\{g_1^+, g_2^+, g_3^-\}, (\text{Country1}, \text{Country2}))$ contains undirected graphs, and both positive (g_1^+, g_2^+) and negative (g_3^-) examples. It is output identifying, as there is a single node in each graph with label *Country1* and a single node in each graph with label *Country2.* Note that the graphs are not uniquely labeled. The dataset $\Delta' = (\{g_1^+, g_2^+, g_3^-\}, (\text{Sea Border}, \text{Member Of}))$, which looks for patterns connecting the labels *Sea Border* and *Member Of* is not output identifying.

Tree Patterns. *Tree patterns* (or simply *patterns*, for short) are used to extract tuples of nodes from graphs. To be precise, a *tree pattern* is a pair $p = (t, \bar{o})$, where t is a labeled tree, and $\bar{o}$ is a tuple of nodes from t, called the *output nodes*, such that

- The labels of t are drawn from the set $\Sigma \cup \{*\}$ where $*$ is a special label, called the *wildcard.*
- The set of edges E_t of t is the union of disjoint sets $E_t^{/}$ and $E_t^{//}$, representing *child* and *descendant* edges, respectively.
- All leaf nodes in t are also output nodes (but not necessarily vice-versa).

As before, we use V_t and l_t to denote the nodes and labeling function of t, respectively. We will consider both undirected trees, and directed, rooted trees. When t is a directed rooted tree, we use $r_t \in V_t$ to denote the root of t.

▶ Remark. In this paper we focus on learning tree patterns from example graphs. Tree patterns in graphs have been of interest in the past, e.g., in the context of keyword proximity search, as they represent tight relationships between nodes [9, 16, 12]. While we do not study the problem of learning graph patterns directly, we note that this is somewhat less interesting than tree patterns. In particular, if there are only positive examples, then whenever there is a connected graph pattern satisfying the examples, there is also a connected tree pattern (derived by removing edges). When negative examples are allowed, then the setting in which there is a single positive and a single negative example already reduces to the graph homomorphism problem, and thus, will not be tractable [10]. Thus, the additional expressive power provided by graphs is either unnecessary (in the positive case) or quickly yields both NP and Co-NP hardness (when negative examples are allowed).

Embeddings. An *embedding* from a tree pattern $p = (t, \bar{o})$ to a graph g, is a function $\mu : V_t \to V_g$ such that

- μ maps (directed) edges in $E_t^{/}$ to (directed) edges in E_g, i.e., for all $(u, v) \in E_t^{/}$, $(\mu(u), \mu(v)) \in E_g$;
- μ maps (directed) edges in $E_t^{//}$ to (directed) paths in E_g, i.e., for all $(u, v) \in E_t^{//}$, there is a (directed) path from $\mu(u)$ to $\mu(v)$ in E_g of length at least 1;

- μ maps nodes v in V_t with a non-wildcard label to nodes in V_g with the same label, i.e., for all $v \in V_t$, either $l_t(v) = *$ or $l_t(v) = l_g(\mu(v))$.

Next we define when a tree pattern p satisfies a dataset Δ, using the notion of an embedding. We note that in the following definition, and throughout this paper, we assume that either *(1)* Δ contains undirected graphs and p is an undirected tree or *(2)* Δ contains directed graphs and p is a directed rooted tree. Moreover, we assume that the tuple of distinguished labels in Δ has the same cardinality as the output nodes tuple in p.

Given a tree pattern $p = (t, \bar{o})$ and a dataset $\Delta = (G, \bar{l})$, we will say that p *satisfies* Δ if

- for all $g^+ \in G$, there exists an embedding μ from p to g such that $l_g(\mu(\bar{o})) = \bar{l}$;
- for all $g^- \in G$, there does not exist an embedding μ from p to g for which $l_g(\mu(\bar{o})) = \bar{l}$.

Similarly, we say that p *injectively satisfies* Δ, if for all positive examples g^+, there exists an injective embedding that maps $\bar{o}$ to nodes with the labels $\bar{l}$ in g^+, and for all negative examples g^-, there does not exist an injective embedding that maps $\bar{o}$ to nodes with the labels $\bar{l}$ in g^-.

▶ **Example 3.** Figure 1 contains three tree patterns. The output nodes in these patterns are denoted with a rectangular box. Consider the datasets

$$\Delta_1 = (\{g_1^+, g_2^+, g_3^-\}, (\text{Country1},\ \text{Country2}))$$
$$\Delta_2 = (\{g_1^+, g_2^-, g_3^+\}, (\text{Country1},\ \text{Country2}))$$
$$\Delta_3 = (\{g_1^+, g_2^+\}, (\text{Country1}, \text{Country2}))\,.$$

The patterns p_2 and p_3 satisfy the datasets Δ_1 and Δ_3, but not Δ_2. Note that every pattern satisfying Δ_1 will satisfy Δ_3, but not vice-versa. Pattern p_1 satisfies Δ_2 (but not any of the others). We note that the pattern derived by replacing "UN" in p_3 with "$*$", satisfies Δ_3, as well as the dataset in which all three examples are positive.

Problems of Interest. Given a dataset, we will be interested in determining whether there exists a satisfying pattern (or an injectively satisfying pattern), as well as in finding such a pattern if one exists. Thus, our two problems of interest can be defined as follows.

▶ **Problem 1** ((Injective) Satisfiability)**.** Given a dataset Δ, determine whether there exists a pattern p that (injectively) satisfies Δ.

▶ **Problem 2** ((Injective) Learning)**.** Given a dataset Δ, find a pattern p that (injectively) satisfies Δ.

The complexity of these problems depends highly upon the types of patterns, embeddings and datasets allowed. Thus, we will study these problems for a variety of settings, considering: injective and arbitrary embeddings, patterns allowing/prohibiting wildcards and descendant edges, directed and undirected datasets, and uniquely labeled, output distinguishing and arbitrary datasets. We will also consider bounding various aspects of the problem (e.g., the number of examples, the number of distinguished labels or the pattern size), and see how this affects the complexity.

We note that the learning problem is at least as hard as the satisfiability problem. Indeed, in some cases, satisfiability may be in polynomial time, but no polynomial size satisfying pattern may exist. For the most part in this paper, we focus on satisfiability, as this problem is usually already hard (i.e., not in polynomial time, unless P = NP). However, for cases in which satisfiability is polynomial, we also consider the learning problem, and show how to find a satisfying pattern (or compact representation of a satisfying pattern) in polynomial time. Later, in Section 6, we consider a third problem of interest – that of finding a minimal-sized satisfying pattern.

Table 1 Complexity of satisfiability for positive datasets.

	Pattern Features	Embedding Type	Graph Type	Data Set	Additional Conditions	Complexity
1.1	$\{//\}$, $\{*,//\}$	–	–	–	–	PTIME (Thm. 5)
1.2	–	–	–	–	BoundP[1]	PTIME (Thm. 5)
1.3	$\emptyset$	–	–	unq	–	PTIME (Thm. 5)
1.4	–	1:m	–	–	BoundE[2]	PTIME (Thm. 7)
1.5	$\{*\}$	1:m	udt	oident, unq	–	PTIME (Thm. 7)
1.6	$\{*\}$	1:m	udt	any	ConD[3]	PTIME (Thm. 7)
1.7	$\{*\}$	1:m	udt	any	BoundLD[4]	PTIME (Thm. 7)
1.8	$\{*\}$	1:1	–	–	–	NPC (Thm. 8)
1.9	$\emptyset$	1:1	–	oident, any	–	NPC (Thm. 8)
1.10	$\{*\}$	1:m	drt	–	–	NPC (Thm. 9)
1.11	$\{*\}$	1:m	udt	any	–	NPC (Thm. 9)
1.12	$\emptyset$	1:m	udt	oident, any	–	NPH (Thm. 10)
1.13	$\emptyset$	1:m	drt	oident, any	–	PSPACE (Thm. 10)

[1] *bounded number of nodes in patterns*
[2] *bounded number of examples*
[3] *connected dataset*
[4] *bounded number of distinct labels in dataset*

4 Positive Datasets

In this section we consider the satisfiability and learning problems for positive datasets. Table 1 summarizes the complexity of satisfiability for a variety of settings. Note that:

- The first column is simply a running number, that will be convenient for reference later in Table 3.
- The second column specifies features allowed in the pattern. All patterns allow labeled nodes and child edges. We note the set of additional features (wildcards and/or descendant edges) allowed in each row. For example, the first row considers the complexity of satisfiability when patterns can contain descendant edges (and possibly wildcards) and the third row considers the complexity of satisfiability when no special features (neither wildcards nor descendant edges) are allowed.
- The third column specifies the type of satisfaction (i.e., embedding) allowed. We use 1:1 to denote injective embeddings and 1:m to denote arbitrary embeddings.
- The fourth column indicates whether the graphs (in the dataset and pattern) are undirected (udt) or directed (drt).
- Finally, the fifth column indicates whether the dataset is uniquely labeled (unq), is output identifying (oident) or can be arbitrary (any). Note that every dataset that is uniquely labeled is also output identifying.

For table cells in which no value is specified (i.e., "–" appears), the complexity result holds regardless of the value of the cell. Due to lack of space, all proofs are deferred to the appendix. However, next to each theorem, we state the intuition behind the proof.

In the remainder of this section, we discuss the complexity of satisfiability and learning, with an emphasis on when and why the problem becomes tractable or intractable. The following example demonstrates one of the difficulties, namely that it is sometimes the case that all satisfying patterns will be exponential in the size of the input.

▶ **Example 4.** Consider the dataset in Figure 2, defined as $\Delta = (\{g_1^+, g_2^+, g_3^+\}, (S, F))$. It is not difficult to see that the smallest tree pattern which satisfies all examples has

$$\begin{tikzcd} g_1 & S \arrow[r] & A \arrow[r] & A \arrow[l, bend right] \arrow[r] & F \end{tikzcd}$$

$$\begin{tikzcd} g_2 & S \arrow[r] & A \arrow[r] & A \arrow[r] & A \arrow[ll, bend right] \arrow[r] & F \end{tikzcd}$$

$$\begin{tikzcd} g_3 & S \arrow[r] & A \arrow[r] & A \arrow[r] & A \arrow[r] & A \arrow[r] & A \arrow[llll, bend right] \arrow[r] & F \end{tikzcd}$$

Figure 2 Dataset $\Delta = (\{g_1^+, g_2^+, g_3^+\}, (S, F))$, for which satisfying patterns are large.

$1 + 2 * 3 * 5 + 1 = 32$ nodes – it is simply a path starting with a node labeled S, and then containing 30 consecutive nodes labeled A, and finally ending with a node labeled F (with the first and last nodes in the path being the output nodes).

Example 4 demonstrates that we cannot always expect the problem of learning from positive examples to be polynomial, as the pattern that must be learned may itself be *at least* exponential in size. The precise complexity of satisfiability and learning, given various restrictions of the settings, is discussed next.

Polynomial Cases. We start by considering cases where both satfiability/injective satisfiability and learning/injective learning, are tractable. Theorem 5 presents three such cases. Intuitively, they all follow from the fact that the given restrictions imply that (1) there are polynomially many (small) patterns that must be considered, in order to find a satisfying pattern, should one exists *and* (2) it is possible to check in polynomial time whether such a pattern satisfies the dataset.

▶ **Theorem 5.** *The (injective) satisfiability and (injective) learning problems are in polynomial time for positive datasets, if one of the following conditions holds*

1. *patterns may contain descendant edges;*
2. *patterns are bounded in size;*
3. *patterns can contain neither wildcards nor descendant edges,* and *datasets are uniquely labeled.*

Intuitively, (1) holds since when patterns can contain descendant edges, (injective) satisfiability and (injective) learning reduce to the problem of graph reachability. Case (2) is immediate, since it implies that there are a bounded number of patterns, each of bounded size, that must be considered. We note that this is useful in practice, as the user may sometimes only be interested in viewing small patterns, and may disregard large patterns as appearing by chance.

To show Case (3), we need the following definition. Let $g_1, \dots, g_m$ be graphs. We denote by $g_1 \otimes \cdots \otimes g_m$ the *multiplication graph* g defined as follows:

- $V_g \subseteq V_{g_1} \times \cdots \times V_{g_m}$ is defined as the set $\{(v_1, \dots, v_m) \mid l_{g_1}(v_1) = \cdots = l_{g_m}(v_m)\}$;
- $l_g((v_1, \dots, v_m)) = l_{g_1}(v_1)$;
- $((v_1, \dots, v_m), (u_1, \dots, u_m)) \in E_g$ if and only if $(v_i, u_i) \in E_{g_i}$ for all $i \leq m$;

The following proposition is easy to show.

▶ **Proposition 6.** *Let $\Delta = (\{g_1^+, \dots, g_m^+\}, (l_1, \dots, l_n))$ be a dataset. Let $g = g_1 \otimes \cdots \otimes g_m$. There exists a pattern p containing no descendant edges and no wildcards that satisfies Δ if and only if there are sets $V \subseteq V_g$ and $E \subseteq E_g$, and a tuple of nodes $(o_1, \dots, o_n)$ from V such that all three of the following conditions hold:* (1) *(V, E) is a tree,* (2) *$l_g(o_i) = l_i$ for all $i \leq n$ and* (3) *every leaf node in (V, E) is in $(o_1, \dots, o_n)$.*

We note that Proposition 6 deals with satisfaction, and does not specifically consider injective satisfaction. Notwithstanding, Case (3) of Theorem 5 holds for both satisfiability and injective satisfiability. Intuitively, the proof follows from the fact that the multiplication graph is polynomial in size when the dataset is uniquely labeled, and from the fact that the existence of an embedding implies the existence of an injective embedding in this case.[1]

We now consider cases in which satisfiability and learning is tractable, but the injective versions are not. (We show intractability of the injection versions of these problems in the next section.)

▶ **Theorem 7.** *The satisfiability problem for positive datasets is in polynomial time, if one of the following conditions holds:*

1. *the number of examples is bounded by a constant;*
2. *patterns can contain wildcards, the dataset is undirected and also*
   (a) either *is output identifying,*
   (b) or *is connected (i.e., all examples are connected graphs),*
   (c) or *contains a bounded number of distinguished labels.*

Moreover, a satisfying pattern can be found and compactly represented in polynomial time and space.

To prove Case (1) of Theorem 7, we show that if Δ is a satisfiable positive dataset containing m examples, each of which contains a graph with at most k nodes, then there exists a pattern satisfying Δ of size at most k^m. Thus, if the number of examples is bounded, then there exists a satisfying pattern of size polynomial in the input. Note that such patterns can be enumerated and their satisfaction verified. (Injective satisfaction cannot, however, be efficiently verified, and thus, we will see later that two examples are sufficient for intractability of injective satisfiability.)

We now consider Case (2) of Theorem 7. This is proven by showing that if there exists a satisfying pattern, then there is one of a special format, called an *output-rooted pattern*. Such patterns $(t, (o_1, \ldots, o_k))$ have only wildcard labeled nodes, and consist of a path from o_1 to each o_i for $1 < i \leq k$. This format allows compact representation (as we only have to store the path lengths, and not the actual paths). Moreover, checking for the existence of such a satisfying output-rooted pattern can be reduced to the problem of determining the existence of paths of odd or even lengths between pairs of nodes, which is solved by 2-coloring the graph. This holds as whenever there is a path of odd (resp. even) distance between two nodes, it can be extended to arbitrarily large paths of odd (resp. even) distance, by repeatedly traversing back and forth over some edge in the path.

Intractable Cases. We now consider cases in which satisfiability, or injective satisfiability, is intractable (unless P = NP). We start by considering injective satisfiability for positive datasets, and show that in every case not covered by the previous section, the problem is NP-complete. We note that membership in NP is immediate, since only injective embeddings are allowed, and hence, one must only consider tree patterns which are at most the size of the smallest graph in any example. NP-hardness is shown by a reduction to the Hamiltonian path problem.

▶ **Theorem 8.** *Injective satisfiability is NP-complete for positive datasets if patterns may contain wildcards, or if datasets are not uniquely labeled. This holds even if there are only two examples in the dataset, and even if there are only two distinguished labels in the dataset.*

[1] Note that the tuple of distinguished labels of a dataset cannot contain repeated labels, by definition.

Next, we consider the satisfiability problem. Theorem 7 showed cases in which the presence of wildcards makes satisfiability polynomial. In Theorem 9 we show that in all remaining cases (either directed datasets, or undirected datasets of arbitrary type), if wildcards are allowed, satisfiability becomes NP-complete.

▶ **Theorem 9.** *When patterns can contain wildcards, satisfiability is NP-complete if one of the following conditions holds:*

1. *the dataset is directed;*
2. *the dataset is undirected, is not output identifying, has an unbounded number of distinguished labels* and *examples may be unconnected graphs.*

For Case (1) of Theorem 9, both NP-hardness and membership in NP are not easy to show. For NP-hardness, our proof is by a reduction from 3-SAT. However, since patterns can contain wildcards, the reduction is intricate, as labels on nodes can no longer be used to differentiate between truth assignments. Instead, in our proof we create examples containing cycles of different prime lengths, and employ the Chinese Remainder Theorem to show that the dataset constructed is satisfiable if and only if the given 3-SAT formula is satisfiable.

The principle difficulty in showing membership in NP is that satisfying patterns may be exponential in size. Thus, we must show that if there is a satisfying pattern, we can find one representable in polynomial size, and also verify the correctness of such a pattern in polynomial time. The crux of the proof lies in two claims that we show. First, if there exists a satisfying pattern, then there is one that has a very simple form (a tree which branches only at the root), with paths that are not more than exponentially long, and hence, can be compactly written. Second, it is possible to determine in polynomial time whether there is a (not necessarily simple) path of a given length l from a given node u to another node v.

For Case (2) of Theorem 9, Membership in NP follows from the fact that if a satisfying pattern exists, then it can be polynomially represented using an output-rooted pattern. NP-hardness is shown by a reduction from 3-SAT.

The remaining case to be considered is when both wildcards and descendent edges are prohibited, and the dataset is not uniquely labeled. For undirected graphs, the precise complexity of satisfiability is unknown; however satisfiability is at least NP-hard. For directed graphs, satisfiability is PSPACE complete.

▶ **Theorem 10.** *When patterns cannot contain wildcards, satisfiability of positive datasets is NP-hard for graphs that are undirected and PSPACE-complete for graphs that are directed. This holds even if there are only two distinguished labels in the dataset, and even if the dataset is output identifying.*

NP-hardness for undirected graphs is shown by a reduction from 3-SAT. PSPACE-completeness for directed graphs is shown by a reduction from the problem of determining emptiness of the intersection of deterministic finite automata [18].

5 Datasets with Positive and Negative Examples

In this section, we explore (injective) satisfiability and (injective) learning for datasets that may contain both positive and negative examples. Unsurprisingly, these problems are usually significantly harder when negative examples are allowed. We present comprehensive results on the complexity of satisfiability and learning. For the injective versions of these problems we also present complexity results; however, there are some remaining open problems. Note that, as before, we primarily consider the (injective) satisfiability problem, but in cases where it is shown to be polynomial, we also consider (injective) learning. See Table 2 for a summary of the satisfiability results of this section.

Table 2 Complexity of satisfiability for arbitrary datasets (which may contain both positive and negative examples).

	Pattern Features	Embedding Type	Graph Type	Data Set	Additional Conditions	Complexity
2.1	–	–	–	–	BoundP[1]	PTIME (Thm. 11)
2.2	$\emptyset$, $\{//\}$	1:m	udt	unq	–	PTIME (Thm. 12)
2.3	$\{*\}$, $\{*, //\}$	1:m	udt	unq	BoundLD[2]	PTIME (Thm. 12)
2.4	$\emptyset$, $\{//\}$	1:1	–	unq	–	NPC (Thm. 13)
2.5	–	1:m	drt	unq	–	NPC (Thm. 14)
2.6	$\{*\}$, $\{*, //\}$	1:m	udt	unq	–	NPC (Thm. 14)
2.7	–	1:m	–	oident, any	–	PSPACE (Thm. 15)
2.8	–	1:1	–	–	–	NPH, Co-NPH, in Σ_2^P (Thm. 16)

[1] *bounded number of nodes in patterns* [2] *bounded number of distinct labels in dataset*

Polynomial Cases. When datasets can contain negative examples, there are only few cases in which satisfiability is tractable. In particular, for the special case where the patterns are of bounded size, the problem remains polynomial. As before, this holds as there are a polynomial number of different bounded patterns that must be considered, and all embeddings can be polynomially enumerated and verified.

▶ **Theorem 11.** *(Injective) Satisfiability and (injective) learning of datasets is in polynomial time, regardless of the type of dataset and pattern, if the tree patterns are bounded in size.*

Theorem 12 presents two additional cases in which satisfiability and learning are in polynomial time. The first case, in which patterns cannot contain wildcards, is easily verified using the multiplication graph of the positive examples. The second case, which allows for wildcards, is shown by proving that the number of different patterns that must be considered is polynomial in the size of the input. Once again, this requires the ability to bound lengths of pattern paths over undirected graphs.

▶ **Theorem 12.** *Satisfiability and learning of uniquely-labeled undirected datasets is in polynomial time, if one of the following conditions hold:*
1. *patterns cannot contain wildcards;*
2. *patterns can contain wildcards, but only have a bounded number of distinguished labels.*

Intractable Cases. Theorem 12 showed cases in which the fact that the dataset is uniquely labeled yields tractability. Unfortunately, the following two theorems shows that this is not always the case. First, we show that Case (1) of Theorem 12 no longer holds if *injective* satisfiability is desired.

▶ **Theorem 13.** *The injective satisfiability problem is NP-complete if the dataset is uniquely labeled and the patterns cannot contain wildcards.*

For the non-injective case, the following theorem shows that every case of uniquely labeled datasets, other than those considered in Theorem 12, is NP-complete.

▶ **Theorem 14.** *Satisfiability of uniquely-labeled datasets is NP-complete, if one of the following conditions holds*
1. *the dataset is directed;*
2. *the patterns can contain wildcards, and there is an unbounded number of distinguished labels.*

For Case (1), NP-hardness is shown by a reduction from 3-SAT. Membership in NP is shown by proving that if a satisfying pattern exists, then there is one that has a representation that is polynomial in the size of the input. We note that proving the latter (i.e., membership) is quite intricate, and is based extensively on the fact that the datasets are uniquely-labeled. Proving this result requires, among other claims, the ability to bound the length of paths in a satisfying pattern. Thus, for example, we can show that in a directed path with k nodes, whenever there is a path from a node u to a node v of length $c > 2k^2 \cdot k!$, there is also a path from node u to node v of length $c - k!$.

For Case (2), membership is shown using the techniques from Theorem 12, while hardness is shown by a reduction from 3-SAT.

The previous theorems fully cover the cases in which the dataset is uniquely labeled. For datasets that are not uniquely labeled, satisfiability is PSPACE complete, regardless of whether the dataset is directed or undirected, and regardless of the allowed features in the patterns. In fact, whereas bounding the number of examples was sufficient to achieve polynomial time for determining satisfiability of positive datasets, this is no longer the case when negative examples are allowed. Satisfiability is PSPACE complete, for datasets that are output identifying or arbitrary, even when there is only a single positive example and a constant number of negative examples.

▶ **Theorem 15.** *Satisfiability of datasets that are output identifying or arbitrary is PSPACE-complete. This holds even if there are only two distinguished labels, a single positive example and a constant number of negative examples.*

We show the above result by a reduction to the problem of equivalence of NFAs [24]. This is achieved by creating examples in which satisfying patterns corresponds to words in an NFA. We note that even when wildcards and descendant edges are technically allowed within patterns, these features can be effectively ruled out using appropriately defined negative examples. We note also that it is possible to use an undirected graph to simulate an automaton (which is, in essence, a directed graph).

We now consider one remaining case of injective satisfiability. For arbitrary graphs, injective satisfiability is NP-hard and Co-NP-hard, and is in Σ_2^P. NP-hardness follows from Theorem 8, Co-NP-hardness can be shown by a reduction to the Hamiltonian path problem, and containment in Σ_2^P is immediate.

▶ **Theorem 16.** *The injective satisfiability problem is NP-hard, Co-NP-hard and in Σ_2^P if the dataset can contain both positive and negative examples, and either the dataset is not uniquely labeled, or patterns can contain wildcards.*

6 Learning Minimal Tree Patterns

The previous sections dealt with the (injective) satisfiability and (injective) learning problems, i.e., determining whether there exists a satisfying pattern for a dataset, and finding an arbitrary such pattern. In this section we focus on finding a *minimal* satisfying pattern, i.e., a satisfying pattern of minimal size.

Formally, given a dataset Δ, we say that a pattern $p = (t, \bar{o})$, satisfying Δ, is *minimal* if there does not exist a pattern $p' = (t', \bar{o}')$ such that *(1)* p' satisfies Δ and *(2)* t' has less nodes than t. This section studies the following problem:

▶ **Problem 3** (Minimal (Injective) Learning). Given a dataset Δ, find a minimal pattern p that (injectively) satisfies Δ.

Table 3 Complexity of minimal learning problem.

Case	Num of Dist. Labels	Pos/ Neg	Pattern Features	Embedding Type	Graph Type	Data Set	Additional Conditions	Complexity
1.1	–	+	$\{//\}$, $\{*, //\}$	–	–	–	–	PTIME (Thm. 17)
1.2, 2.1	–	–	–	–	–	–	BoundP[1]	PTIME (Thm. 17)
1.3–1.7	var	+	$\emptyset$, $\{*\}$	–	–	–	–	NPC (Thm. 19)
1.3	const	+	$\emptyset$	–	–	unq	–	PTIME (Thm. 20)
1.4	const	+	–	1:m	–	–	BoundE[2]	PTIME (Thm. 20)
1.5–1.7	const	+	$\{*\}$	1:m	udt	any	–	PTIME (Thm. 18)
2.2, 2.3	–	$\pm$	–	1:m	udt	unq	–	NPC (Thm. 21)

[1] *bounded number of nodes in patterns* [2] *bounded number of examples*

Obviously, the problem of finding a minimal satisfying pattern is no easier than determining satisfiability or finding an arbitrary satisfying pattern. Under most conditions, both of the latter problems are intractable. However, in the previous sections we identified several cases in which satisfiability and learning are polynomial-time problems. For these cases, we study the complexity of the problem of finding a minimal satisfying pattern. Finding such patterns is clearly a problem of practical interest.

The complexity results are summarized in Table 3. *We emphasize that the only cases considered here are those for which satisfiability and learning have previously been shown to be in polynomial time. Thus, cases not considered in this table are certainly intractable.* All columns other that the first three, appeared in previous tables. In addition:

- The first column indicates the number in Table 1 or 2 of the case considered.
- The second column indicates whether the number of distinguished labels in the dataset is assumed to be constant (const) or whether this number is a variable of the input (var). We note that it is natural to assume that the number of distinguished labels is constant, as it is likely to a small number (whereas the examples may be large). As indicated in the table, often the assumption that the number of distinguished labels is constant is sufficient to achieve polynomial time.
- The third column indicates whether the dataset contains only positive examples (+) or both positive and negative ($\pm$). In previous sections this column was not needed as these two different cases were presented in different tables.[2]

In the remainder of this section, we prove the complexity results summarized in Table 3. We divide up the discussion by the type of proof technique used to derive the results.

Polynomial Number of Candidates. In some cases, given Δ, it is possible to define a polynomial size set of patterns P, for which it is guaranteed that if Δ is satisfiable, then P must contain a minimal-size satisfying pattern. We will say that P is the set of minimal pattern *candidates*. If P is polynomial in size, and can be efficiently found, then minimal learning is clearly in polynomial time. The next theorem deals with two cases in which such a polynomial size set of candidates can be efficiently found, and thus, minimal satisfying and injectively satisfying patterns can be found in polynomial time. Both of these cases are quite straightforward – descendent edges significantly simplify the candidates that must be considered (in the first case), and the second case explicitly bounds the size of the candidates considered.

[2] As before, columns with "–" allow for any value, without affecting problem complexity.

▶ **Theorem 17.** *Let Δ be a satisfiable dataset. Then, it is possible to find a minimal (injectively) satisfying pattern in polynomial time if any of the following conditions hold:*
1. *Δ is positive, and patterns can contain descendant edges;*
2. *there is a constant k, such that only patterns containing at most k nodes are to be returned;*

The following theorem presents an additional case where the set of candidates for satisfaction is polynomial. Note that bounding the number of distinguished labels infers a bound on the number of leaf nodes in the patterns, which can be used to significantly reduce the number of patterns considered.

▶ **Theorem 18.** *Let Δ be a satisfiable dataset with a constant number of distinguished labels. Then, it is possible to find a minimal satisfying pattern in polynomial time if Δ is positive and undirected, and patterns can contain wildcards.*

Reductions to the Steiner Tree Problem. We consider cases in which the problem of finding a minimal satisfying pattern is similar to the problem of finding a Steiner tree in a graph. The Steiner tree decision problem is: given a graph g, a set of nodes V, and a number k, find a subtree of g of size at most k, containing V. It is well known that the Steiner tree problem is NP-complete. However, it is solvable in polynomial time if the number of nodes in V is constant, both if the graph is undirected [9], and if the graph is directed [20] (and a rooted Steiner tree is desired).

▶ **Theorem 19.** *Let Δ be a positive dataset with an unbounded number of distinguished labels and let k be a positive integer. Then, the problem of determining whether there exists a pattern p that (injectively) satisfies Δ and contains at most k nodes is NP-complete if no descendant edges are allowed in the pattern.*

Membership is easy, as given a pattern with k nodes, we can guess and verify embeddings in polynomial time. Hardness is shown by a reduction from the Steiner tree problem.

When considering datasets with a constant number of labels, the minimal learning problem sometimes becomes tractable. This is the case, in particular, in Theorem 20. The proof leverages the aforementioned polynomial case for Steiner trees. In particular, in both cases below, the multiplication graph of the examples is guaranteed to be polynomial in size, and a Steiner tree in this graph can be used to generate a minimal satisfying pattern.

▶ **Theorem 20.** *Let Δ be a positive dataset with a constant number of distinguished labels. Then, it is possible to find a minimal satisfying pattern in polynomial time if one of the following conditions hold:*
1. *Δ is uniquely labeled, and no wildcards are allowed in the patterns; or*
2. *Δ contains a constant number of examples.*

Datasets With Negative Examples. The final case that must be considered is one in which the dataset can contain both positive and negative examples. Bounding the number of distinguished labels no longer leads to tractability in this case.

▶ **Theorem 21.** *Let Δ be a dataset with both positive and negative examples, and with two distinguished labels. Let k be a positive integer. Then, the problem of determining whether there exists a pattern p that satisfies Δ and contains at most k nodes is NP-complete if Δ is uniquely labeled and undirected.*

Once again, membership in NP is immediate. We show NP-hardness by using the same reduction from 3-SAT as in the proof of Theorem 13.

7 Conclusion

In this paper we extensively studied three problems – satisfiability, learning and minimal learning, for datasets containing positive and negative example graphs, and a tuple of distinguished labels. We have shown that the complexity of these problems is highly dependent on the precise setting, i.e., that the presence or absence of specific features can make these problems significantly easier or harder.

Many interesting problems have been left open for future work. First, from a practical standpoint, we intend to investigate how these results can be integrated into a system, in order to allow users to provide examples, instead of explicit queries. A second problem of interest is to develop an algorithm that can find top-k satisfying patterns (preferably of diverse structure), for some appropriately defined ranking problem,. Third, we intend to study the problem of maximally satisfying the given examples, when there is no pattern that can satisfy all examples. Finally, we intend to study classic machine learning questions arising from this setting, such as investigating in which cases any specific hypothesis can be guaranteed to be learned, given that the system can provide examples to the user, and specifically ask whether these examples are positive. In machine learning terminology, this is the problem of learning patterns from (equivalence/membership/subset etc.) queries.

Acknowledgments. The authors were supported by the Israel Science Foundation (Grant 1467/13) and the Ministry of Science and Technology (Grant 3-9617).

References

1 Thomas Amoth, Paul Cull, and Prasad Tadepalli. On exact learning of unordered tree patterns. *Machine Learning*, 44:211–243, 2001.

2 Dana Angluin. Negative results for equivalence queries. *Machine Learning*, 5(2):121–150, July 1990.

3 Timos Antonopoulos, Frank Neven, and Frédéric Servais. Definability problems for graph query languages. In *Proceedings of the 16th International Conference on Database Theory*, pages 141–152, New York, NY, USA, 2013. ACM.

4 Hiroki Arimura, Hiroki Ishizaka, and Takeshi Shinohara. Learning unions of tree patterns using queries. *Theor. Comput. Sci.*, 185(1):47–62, 1997.

5 Julien Carme, Michal Ceresna, and Max Goebel. Query-based learning of XPath expressions. In *ICGI*, 2006.

6 Adriane Chapman and H. V. Jagadish. Why not? In *SIGMOD*. ACM, 2009.

7 Sara Cohen and Yaacov Y. Weiss. Certain and possible XPath answers. In *ICDT*, 2013.

8 Anish Das Sarma, Aditya Parameswaran, Hector Garcia-Molina, and Jennifer Widom. Synthesizing view definitions from data. In *ICDT*, 2010.

9 S. E. Dreyfus and R. A. Wagner. The steiner problem in graphs. *Networks*, 1(3):195–207, 1971.

10 Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W.H. Freeman, 1979.

11 Melanie Herschel, Mauricio A. Hernández, and Wang-Chiew Tan. Artemis: a system for analyzing missing answers. *Proc. VLDB Endow.*, 2:1550–1553, August 2009.

12 Vagelis Hristidis, Yannis Papakonstantinou, and Andrey Balmin. Keyword proximity search on XML graphs. In *ICDE*, 2003.

13 Jiansheng Huang, Ting Chen, AnHai Doan, and Jeffrey F. Naughton. On the provenance of non-answers to queries over extracted data. *PVLDB*, 1(1):736–747, 2008.

14 Chuntao Jiang, Frans Coenen, and Michele Zito. A survey of frequent subgraph mining algorithms. *Knowledge Eng. Review*, 28(1):75–105, 2013.

15 Benny Kimelfeld and Phokion G. Kolaitis. The complexity of mining maximal frequent subgraphs. In *PODS*, 2013.

16 Benny Kimelfeld and Yehoshua Sagiv. Finding and approximating top-k answers in keyword proximity search. In *PODS*, 2006.

17 Raymond Kosala, Maurice Bruynooghe, Jan Van Den Bussche, and Hendrik Blocked. Information extraction from web documents based on local unranked tree automaton inference. In *IJCAI*, 2003.

18 D. Kozen. Lower bounds for natural proof systems. In *FOCS*, 1977.

19 Alexandra Meliou, Wolfgang Gatterbauer, Katherine F. Moore, and Dan Suciu. WHY SO? or WHY NO? Functional Causality for Explaining Query Answers. In *Management of Uncertain Data*, 2010.

20 Neeldhara Misra, Geevarghese Philip, Venkatesh Raman, Saket Saurabh, and Somnath Sikdar. FPT algorithms for connected feedback vertex set. *J. Comb. Optim.*, 24(2):131–146, 2012.

21 Rika Okada, Satoshi Matsumoto, Tomoyuki Uchida, Yusuke Suzuki, and Takayoshi Shoudai. Exact learning of finite unions of graph patterns from queries. In *Algorithmic Learning Theory*, LNCS, pages 298–312. Springer Berlin Heidelberg, 2007.

22 Stefan Raeymaekers, Maurice Bruynooghe, and Jan Bussche. Learning (k,l)-contextual tree languages for information extraction from web pages. *Machine Learning*, 71(2-3):155–183, June 2008.

23 Slawek Staworko and Piotr Wieczorek. Learning twig and path queries. In *ICDT*, 2012.

24 L. J. Stockmeyer and A. R. Meyer. Word problems requiring exponential time. In *STOC*, 1973.

25 Quoc Trung Tran and Chee-Yong Chan. How to conquer why-not questions. In *SIGMOD*, 2010.

26 Quoc Trung Tran, Chee-Yong Chan, and Srinivasan Parthasarathy. Query by output. In *SIGMOD*. ACM, 2009.