Learning Definable Hypotheses on Trees

Emilie Grienenberger
ENS Paris-Saclay, 61 Avenue du Président Wilson, 94230 Cachan, France
emilie.grienenberger@ens-cachan.fr

Martin Ritzert
RWTH Aachen University, Templergraben 55, 52062 Aachen, Germany
ritzert@informatik.rwth-aachen.de

Abstract

We study the problem of learning properties of nodes in tree structures. Those properties are specified by logical formulas, such as formulas from first-order or monadic second-order logic. We think of the tree as a database encoding a large dataset and therefore aim for learning algorithms which depend at most sublinearly on the size of the tree. We present a learning algorithm for quantifier-free formulas where the running time only depends polynomially on the number of training examples, but not on the size of the background structure. By a previous result on strings we know that for general first-order or monadic second-order (MSO) formulas a sublinear running time cannot be achieved. However, we show that by building an index on the tree in a linear time preprocessing phase, we can achieve a learning algorithm for MSO formulas with a logarithmic learning phase.

2012 ACM Subject Classification Theory of computation → Logic

Keywords and phrases monadic second-order logic, trees, query learning

Digital Object Identifier 10.4230/LIPIcs.ICDT.2019.24

Funding *Martin Ritzert*: This work is supported by the German research council (DFG) Research Training Group 2236 UnRAVeL.

1 Introduction

In this paper we study the algorithmic complexity of learning properties of nodes in directed labeled trees using a declarative framework introduced by Grohe and Turán [18]. Let T be such a tree with nodes $V(T)$. We call T the *background tree* of our learning problem. The tree T encodes the background knowledge of the learning problem and thus provides the information on which the classification of the nodes $u \in V(T)$ can be based. In our setting a (boolean) *classifier* is a function $H\colon V(T) \to \{+,-\}$ that estimates whether a given node admits a certain property. A learning algorithm gets a *training set* $S \subseteq V(T) \times \{+,-\}$, that is a set of pairs (u,c) of positive and negative *examples*, and the background tree T as input and returns a classifier $H_S\colon V(T) \to \{+,-\}$ as its *hypothesis*. We say that learning was successful if the hypothesis H_S is *consistent* with S which means that for every $(u,c) \in S$ we have that $H_S(u) = c$. Achieving consistency with S can be seen as the extreme case of minimizing the *training error*, i.e. the number of $(u,c) \in S$ such that $H_S(u) \neq c$. Minimizing the training error, also called *empirical risk minimization*, results in provably good generalization behavior in the PAC learning model (see [6]). For those generalization results, we use that logical formulas on trees admit bounded VC-dimension (shown by Grohe and Turán [18]) and that any consistent learner can be turned into a PAC learner by an appropriate training set S (see [6]). We give more details on the connection to PAC learning in Section 2.3.

22nd International Conference on Database Theory (ICDT 2019).
Editors: Pablo Barcelo and Marco Calautti; Article No. 24; pp. 24:1–24:18
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

An example of a simple property a node can admit is having an ancestor with label b. This property can be expressed by the logical formula $\varphi(x) = \exists y\,(y < x) \wedge R_b(y)$. We aim to learn properties which can be defined by logical formulas with parameters based on positive and negative examples over tree-structured data such as web pages, XML databases and JSON files.

▶ **Example 1.** Given a large website such as a news portal. The document object model (DOM) of a website is the tree of elements which form the website. Many websites contain a number of ads and some of them are not trivially detectable. A learning algorithm could then estimate the property of a position in the DOM to be part of some ad.

The output of a learning algorithm would then be a formula that distinguishes nodes belonging to the content of the web page from those belonging to ads. Those formulas could then be used as a basis for new simpler or better filter rules.

In Example 1, the user could select parts of a web page, which he sees as advertisement and then let the learning algorithm produce a classifier which is consistent with his choice. In this paper we will not go into detail of how we get our training set but instead only talk about finding consistent hypotheses for a given training set.

We consider learning algorithms that return classifiers based on logical formulas, especially quantifier-free formulas and formulas from monadic second-order logic. In our logical framework, a classifier consists of a formula $\varphi(x\,;\bar{y})$ and an instantiation $\bar{v}$ of the free variables $\bar{y}$. This formula φ has two types of free variables; we refer to x as the *instance variable* and to $\bar{y} = (y_1, \ldots, y_\ell)$ as *parameter variables*, where $\ell \in \mathbb{N}$. The background structure T is the (fixed) background knowledge that encodes the context of a node which is to be classified. The parameters of φ, which can be seen as constants the formula is allowed to use for the classification, are taken from $V(T)$. In Example 1, a classifier would consist of a logical formula and a number of positions in the DOM of the web page. The formula $\varphi(x\,;\bar{y})$ together with an ℓ-tuple $\bar{v} \in V(T)^\ell$ of parameters then defines a binary classifier $[\![\varphi(x\,;\bar{v})]\!]^T : V(T) \to \{+,-\}$ over the tree T as follows. An instance $u \in V(T)$ is classified as positive if $T \models \varphi(u, \bar{v})$ such that we have $[\![\varphi(x\,;\bar{v})]\!]^T(u) = +$. Correspondingly we have $[\![\varphi(x\,;\bar{v})]\!]^T(u') = -$ for any $u' \in V(T)$ with $T \not\models \varphi(u', \bar{v})$. We call such a classifier where φ is an MSO formula an *MSO definable hypothesis*.

We assume the background tree T to be very large, which means large enough that just reading it sequentially takes long, while the logical formula returned by the learning algorithm is assumed to be small for every real-world query. This implies two strategies. First, we use a *data complexity* view for the analysis considering the tree T and the training set S as data and the hypothesis class parameterized by ℓ (and an additional parameter q introduced later) as a constant, such that the complexity results are only given in terms of T and S. Essentially this means that the influence of the formula is considered to be constant. Second, we are interested in finding algorithms which run in sublinear time in the size of T. As such sublinear algorithms are unable to read the whole background tree T, we model the exploration of T using oracles. Those oracles allow the learning algorithm to explore the tree by following edges, starting from the training examples in S. This is formally defined in Section 2.

In general there are no consistent sublinear learning algorithms for first-order and monadic second-order formulas over trees. In [16] the authors have shown that for learning first-order formulas over words, linear time is necessary. The same counterexample can also be used for trees, showing that linear time is again necessary. For structures of bounded degree, there exists a sublinear learning algorithm for first-order formulas (see [17]). This result is not applicable in our setting, as the ancestor relation $\leq$ in the signature of our trees induces unbounded degree (the degree of the root is $|V(T)| - 1$ as it is an ancestor of every other node).

1.1 Our Results

In our formal setting, we consider learnability on trees for monadic second-order logic and the quantifier-free fragment of first-order logic. We show that in contrast to the corresponding case on strings (see [16]), even the relatively simple task of learning quantifier-free formulas on trees needs at least linear time. This is due to the need to synthesize appropriate parameters, a task which can involve searching for the largest common ancestor of two nodes. We show that this is really the core of the problem by giving a sublinear learning algorithm for quantifier-free formulas in Section 3 where we exploit an additional oracle providing access to the largest common ancestor of two nodes.

As we know that there is no sublinear learning algorithm for MSO formulas on trees, we investigate whether the necessary linear computation depends on the training examples. This hardness still holds if the learning algorithm is allowed to return more parameters, as long as the complete training set can not be encoded in those parameters. It turns out that it is possible to build an auxiliary structure in linear time which can then be used for sublinear learning. We present an algorithm which builds such an index structure without knowledge of the training set in linear time and then uses logarithmic time to output a consistent hypothesis. The algorithm builds on the results on strings (see [16]) as well as known techniques for evaluating MSO formulas under updates (see [5]), combining them in a non-trivial way to a learning algorithm for MSO formulas. The linear indexing phase in the learning algorithm cannot be avoided since already for first-order formulas on words it is necessary to invest at least linear time in $\mathcal{O}(|T|)$ to find a consistent hypothesis. The following theorem is the main result of this paper.

▶ **Theorem 2.** *There is a consistent MSO learning algorithm on trees which uses linear indexing time $\mathcal{O}(|T|)$ and logarithmic learning time $\mathcal{O}(|S| \log |T|)$.*

As an application of our indexing algorithm we describe an online learning algorithm that computes an index in $\mathcal{O}(|T|)$ and then, for a sequence of examples, updates its MSO-definable hypothesis in time $\mathcal{O}(\log^2(T))$ per example. That is, in this setting the examples arrive one-by-one and we are able to maintain a consistent hypothesis in polylogarithmic time in the background structure.

1.2 Related work

The field of inductive logic programming (see for example [10, 22, 24, 25, 26]) is very close to our framework. In both cases the aim is to infer logical formulas from positive and negative examples such that the logical formula is consistent with the training examples. The main difference to our setting is that in the ILP framework the background knowledge is also encoded in logic (a so called background theory), while we use a structure to encode background knowledge. In our setting, facts such as gender and age of a person or the issuing institute of a credit card are represented using nodes for person and credit card, as well as unary relations to describe their attributes. Naturally facts which involve multiple entities can be represented by edges. Our framework is able to represent such facts as long as the union of all binary relations still describes a tree or forest while in the ILP setting there is no such restriction. The other important difference is that ILP focuses on first-order logic (and there especially Horn-formulas), while in this paper we work with monadic second-order logic (MSO) which is strictly more expressive than first-order logic. There is a number of other logical frameworks for machine learning, mainly originating from the field of formal verification and databases. Examples are given by [1, 8, 23, 14, 20, 36].

Another related field, which is based on the query by example strategy, is to learn XPATH queries as in [32]. There, unary relations defined by an XPATH expression are learned for arbitrary training sets. The main difference to our setting is that we use MSO formulas, which are in general more expressive than XPATH statements and then restrict the maximal complexity of our formulas.

The field of automata learning and learning of regular languages is also to some degree similar to our setting, especially since we are also applying automata based techniques. There are numerous negative results such as [2, 15, 28, 21, 4]. Of the positive results in that area [3, 30, 27, 13], most of them use an active framework where a teacher iteratively gives counterexamples until the hypothesis is correct. In our framework a consistent hypothesis for a training set is sought and that training set is known from the beginning. Even though our classification problem can be encoded as a learning problem for regular tree languages, their results seem technically unrelated to ours.

2 Preliminaries

In this paper we work with logical formulas from the quantifier-free fragment of first-order logic and formulas from monadic second-order logic. Quantifier-free formulas only consist of boolean combinations of atomic properties, while monadic second-order logic (MSO) extends first-order logic (FO) by quantification over sets of nodes. As an example, take the MSO formula $\exists X \forall z\, Xz$ which is always satisfied as there is always a set X containing all elements of the structure. It is known that a set of trees can be recognized by a deterministic bottom-up tree automaton (DTA) $\mathcal{A}$ if and only if it can be characterized by an MSO sentence Φ and both $\mathcal{A}$ and Φ can be computed from each other. For a more detailed description of MSO and tree automata we refer to [33].

In this paper we consider labeled trees as background structures. The most prominent examples for trees in a database context are the tree-structured data exchange formats XML and JSON. Formally a labeled tree $T = (V(T), E_1, E_2, R_1, \dots, R_r, \leq)$ is a structure with vertex set $V(T)$ and binary edge relations E_1 and E_2 encoding the first and second child of a node in a binary tree, or in case of unranked trees, the first child and the next sibling of each node. The unary relations $R_1, \dots R_r$ define the label of each node and for every $a, b \in V(T)$ we have $a \leq b$ if a is an ancestor of b. In this paper we use formulas over the alphabet $\sigma = \{E_1, E_2, \leq, R_1, \dots, R_r\}$ which means that in a formula we can access the tree structure using E_1 and E_2 as well as the labels of each node using $R_1, \dots R_r$. A fragment of an XML document (focusing on persons) could look like the following.

```
<person name="A. Turing" birthday="1912-06-23">
  <interest>computer science</interest>
  <interest>marathon running</interest>
  ...
</person>
```

A formula has access to all tags occurring in the XML document. In the above XML fragment, the labels are given by the unary relations `‘person’`,`‘name’`,`‘birthday’`,`‘interest’`. Adding additional content-based labels such as `‘computer scientist’` or `‘runner’` to the set of unary relations allows to write formulas that depend on the content of nodes. Without such content-based labels a formula could only use structural properties and define sets such as all persons with at least three interests and two friends.

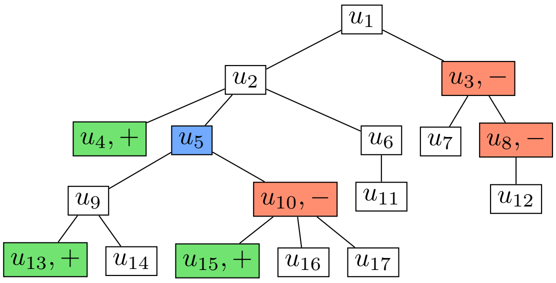

Figure 1 A tree with positive (green) and negative (red) example nodes.

2.1 Learning Model

We consider the model of *supervised learning* where the input to a *learning algorithm* is a *training set* (or training sequence) $S \subseteq V(T) \times \{+,-\}$. Each *example* $(u,c) \in S$ consists of a node u and its classification c. We assume that S is non-contradicting, that is if $(u,c) \in S$, then $(u,\neg c) \notin S$. For the tree given in Figure 1 we define the training set:

$$S = \{(u_3,-),(u_4,+),(u_8,-),(u_{10},-),(u_{13},+),(u_{15},+)\}$$

As already defined in the introduction, a definable hypothesis $[\![\varphi(x\,;\bar{v})]\!]^T$ assigns $+$ to every position $u \in V(T)$ with $T \models \varphi(u,\bar{v})$ and $-$ otherwise. Let $\varphi(x\,;y) = \exists z(E(x,z) \wedge E(z,y)) \wedge x \neq y$ accepting all positions with a distance of 2 from the position of y (T contains no self-loops). In the example from Figure 1 we have $[\![\varphi(x\,;u_5)]\!]^T(u) = +$ if and only if $u \in \{u_1,u_4,u_6,u_{13},u_{14},u_{15},u_{16},u_{17}\}$. This hypothesis is *consistent* with S as it accepts all positive and none of the negative examples from S. The formula $\psi(x) = \exists\, z E_1(z,x)$ defines another consistent hypothesis with $[\![\psi(x)]\!]^T(u) = +$ if and only if $u \in \{u_2,u_4,u_7,u_9,u_{11},u_{12},u_{13},u_{15}\}$.

The quantifier rank $\mathrm{qr}(\varphi)$ of a formula φ is the maximal nesting depth of quantifiers in φ. As every set $S^+ \subseteq V(T)$ is definable by a (long enough) MSO formula φ, we restrict our study to sets which are definable by formulas $\varphi(x\,;y_1,\dots y_\ell)$ with $\mathrm{qr}(\varphi) \leq q$ where q and ℓ are considered to be part of the problem. Restricting q and ℓ reduces the risk of overfitting as such restricted formulas can only memorize a bounded number of positions and thus, on larger training sets, have to exploit more general patterns in the data.

Our framework naturally admits two different learning problems. In *model learning* we assume that there is a consistent classifier $[\![\varphi(x\,;\bar{v})]\!]^T$ with $\bar{v} \in V(T)^\ell$ and $\mathrm{qr}(\varphi) \leq q$, but only q and ℓ are given to the learning algorithm. This reflects the assumption that there is a simple, as expressed by the choice of q and ℓ, but unknown pattern behind the classification of the training examples from S. In *parameter learning* the formula φ of a consistent classifier $[\![\varphi(x\,;\bar{v})]\!]^T$ is fixed. The learning algorithm is not allowed to modify φ and has to find a consistent parameter setting $\bar{v} \in V(T)^\ell$. This variant reflects the case where we have a general idea about how the solution looks like, but are missing the details. Counterintuitively, parameter learning is the harder problem: the restriction to a specific formula φ might impose (unnecessary) restrictions on the parameters. An edifying example is the parameter learning problem on a background structure T with a singleton unary relation R, using the formula $\vartheta(x\,;y) = R(y)$ and a training set $S = \{(u,+)\}$ for an arbitrary $u \in V(T)$. In this example, for any fixed traversal strategy of the learning algorithm on $V(T)$, the single possible parameter can be placed in the position which is evaluated last. For the associated model learning problem, a possible solution would be to return the formula $\psi(x) = \texttt{true}$ without parameters, which defines a hypothesis consistent with S. In the example given

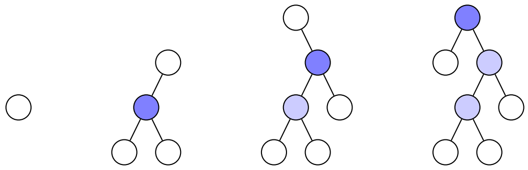

Figure 2 From left to right: Exploration of a tree using neighborhood queries starting from an example node. The dark blue node is the one on which the neighborhood query has been performed last.

in Figure 1, a learning algorithm for the model learning problem would be free to choose between any consistent hypothesis, such as the example formulas $\varphi(x\,;u_5)$ and $\psi(x)$ given above. In the parameter learning problem with the formula $\varphi(x\,;y)$, the (only) consistent output is the assignment $y = u_5$.

In practice, whenever we want to evaluate a definable hypothesis $[\![\varphi(x\,;\bar{v})]\!]^T$, we have to solve an instance of the model checking problem for the formula φ over the structure T. For the case of a fixed MSO formula $\varphi(x\,;\bar{y})$ on a tree T, there is an evaluation strategy in $\mathcal{O}(|T|)$ using tree automata, see for example [33], while a quantifier-free formula ϑ can be evaluated in time $\mathcal{O}(|\vartheta|)$.

2.2 Access Model

A sublinear learning algorithm is unable to read the whole background structure during its computation. We therefore model the access to the background structure by oracles.

Relation Oracles. For a k-ary relation R, the corresponding oracle returns on input of $\bar{u} \in V(T)^k$ whether $\bar{u} \in R$ holds in T.

Neighborhood Oracle. On input of a node $u \in V(T)$, returns the 1-neighborhood of u in T.

In a binary tree, the 1-neighborhood N of a node u consists of u, its parent and its child nodes. In the unranked case $N(u)$ consists of u as well as its left and right sibling, first child and parent. This access model is called *local access* as the tree can only be explored by following edges. Directly jumping to the closest node that is in a relation R or to the last child of an unranked node is not possible in this access model. In practice, those oracles can be implemented using random access on the background tree T. A learning algorithm using local access starts with all nodes occurring in the training set S and then explores the background tree T using the neighborhood and relation oracles. Figure 2 illustrates how the learning algorithm can explore the background tree using subsequent neighborhood queries on the topmost node. The neighborhood queries return the vertices, while the relation queries clarify directions and labels.

2.3 PAC learning

It is known that under certain simplicity restrictions, a consistent learner generalizes well to new and potentially unseen examples. For a *probably approximately correct (PAC)* learning algorithm we have that for every ε and δ there is a size $s \in \mathbb{N}$ of the training set S such that the error of the hypothesis under new examples is bounded by ε with a confidence level of $1-\delta$. This is made formal in Equation (1). Let $c^*\colon V(T) \to \{+,-\}$ be the function that assigns the correct classification to every node $u \in V(T)$. Let the training set $S \in 2^{V(T)\times\{+,-\}}$ be a set of t examples $(u, c^*(u))$ chosen independently and identically distributed (i.i.d.) according to

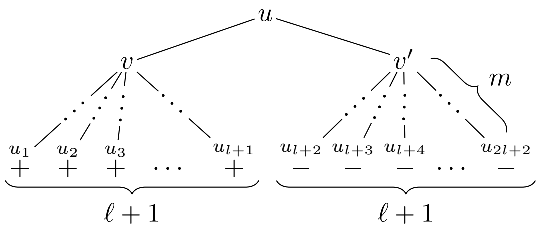

Figure 3 Sketch of the tree T_m parameterized by m and ℓ used in Lemma 4. Substituting the nodes v and v' by balanced binary trees results in the actual tree T_m.

a fixed distribution D. Let $(u, c^*(u)) \sim D$ and $S \sim D$ denote the random choices according to D. Let $H_S \colon V(T) \to \{+, -\}$ be the hypothesis returned by the learning algorithm on input of the training set S. Then the PAC criterion is given by

$$\Pr_{S\sim D}\left(\Pr_{(u,c)\sim D}(H_S(u) \neq c^*(u)) \leq \varepsilon\right) \geq 1-\delta \tag{1}$$

where the outer probability (the *confidence*) $\Pr_{S\sim D}$ is taken over the training set S for which the learning algorithm produces a hypothesis H_S. The inner probability is the expected error of the hypothesis H_S for an example chosen according to the distribution D. For more details on the PAC learning model, we refer to [34].

There is a very general result from learning theory that shows that for any hypothesis class with bounded Vapnik-Chervonenkis (VC) dimension a consistent learner can be turned into a PAC learning algorithm by providing a large enough training set (see [35] or [6]). For the case of MSO definable hypotheses on trees, the VC dimension is bounded (see [18]). Hence, there is a sufficient size s of S, depending polynomially on $\frac{1}{\varepsilon}$ and $\frac{1}{\delta}$, to satisfy the PAC criterion. Using the algorithms from our Theorems 5 and 6, each providing a consistent learning algorithm for learning formulas on trees, we have the following corollary.

▶ **Corollary 3.** *There are (efficient) PAC learning algorithms for of learning quantifier-free formulas and monadic second-order formulas over trees.*

The running time of those PAC learning algorithms follows directly from the corresponding theorems. A PAC learning algorithm is called *efficient* if the dependence of the running time on the number of training examples is polynomial which is the case in Theorem 5 and 6.

3 Quantifier-free formulas

The class of quantifier-free formulas $\Phi_\sigma[n]$ with n free variables over the signature σ is defined as the fragment of first-order logic without quantifiers. This means that every formula $\varphi(\bar{x}) \in \Phi_\sigma[n]$ can only compare the free variables using a boolean combination of atomic properties from σ. We exploit this limitation to obtain a learning algorithm for quantifier-free formulas which runs in constant time with respect to T under a slightly relaxed notion of local access. We start by showing that there is no sublinear learning algorithm using the notion of local access as defined in Section 2.

▶ **Lemma 4.** *For every $\ell \in \mathbb{N}$, there is no consistent learning algorithm using local access that, given a binary tree T and a training set, returns a consistent hypothesis $[\![\varphi(x\,;\bar{v})]\!]^T$ with $\varphi \in \Phi_\sigma[\ell+1]$ in time $o(|T|)$.*

Proof sketch. Let L be such a sublinear learning algorithm. Consider the family of trees $(T_m)_{m\in\mathbb{N}}$ shown in Figure 3 with ℓ fixed to the number of parameters used in the formulas returned by L. We define the training set $S = \{(u_1,+),\dots(u_{\ell+1},+),(u_{\ell+2},-),\dots,(u_{2\ell+2},-)\}$ containing $2\ell+2$ examples. In Figure 3, the positions of the nodes u_i are indicated by their classifications + or −. The size of T_m is linear in m for any fixed ℓ and therefore we can choose m such that the running time of L on T_m is smaller than m. This is possible since L runs in time $o(|T_m|)$.

The formula $\varphi(x;y) = y < x$ with parameter v as indicated in Figure 3 is consistent with S. Let $\psi(x\,;\bar{y}) \in \Phi_\sigma[\ell+1]$ be the formula and $v_1,\dots,v_\ell \in V(T_m)$ the parameters returned by L on input T_m and S. Each parameter v_i is an ancestor to exactly one example from S as the algorithm may only return nodes as parameters it has seen during the computation and every leaf is an example node. Every quantifier-free formula can only compare the free variables (x and $\bar{y}$) directly using relations from σ. Therefore, independent of ψ, every parameter has the same effect on every example but (possibly) the one below the parameter. Since there are $\ell+1$ positive and negative examples, at least one of them will be misclassified which proves the lemma. Intuitively this holds as all paths locally look identical and the algorithm is unable to detect on which side of the tree an example lies. ◀

In the following we extend the notion of local access to cope with the globality of the ancestor relation.

3.1 Extended local access

Our next result states that synthesizing the parameters for quantifier-free formulas is really the core of the linear complexity. *Extended local access* extends local access by a *common ancestor oracle*:

Common Ancestor Oracle. On input of two nodes u and v, it returns the lowest node w such that $w \le u$ and $w \le v$ (their lowest common ancestor)

The consistent parameter v for the counterexample from Theorem 4 can easily be found using extended local access. This is generalized in the following theorem showing that for quantifier-free formulas the gap between linear time and sublinear learnability is due to the computation of common ancestors. Let $\ell \in \mathbb{N}$ be an integer.

▶ **Theorem 5.** *There is a learning algorithm that, on input of a tree T and a training set S, outputs a consistent hypothesis $[\![\varphi(x\,;\bar{v})]\!]^T$, where $\varphi \in \Phi_\sigma[1+\ell]$ and $\bar{v} \in V(T)^\ell$ for binary trees and $\varphi \in \Phi_\sigma[1+2\ell]$ ($\bar{v} \in V(T)^{2\ell}$) for unranked trees. This algorithm uses extended local access and runs in time $\mathcal{O}\left(|S|^2 + |S|^\ell |S|\right)$.*

Note that for unranked trees we consider the alphabet $\sigma = \{E_1, E_2, \le, \preceq\}$. The additional relation $\preceq$ is a partial order, defined as the reflexive transitive closure of E_2. It strictly increases the expressive power of quantifier-free formulas and accounts for need of additional parameters.

Proof sketch. We only sketch the proof for binary trees here. The extension to unranked trees uses the same general idea, but the candidate set for the parameters is slightly different.

The proof uses the concepts of *sufficient sets* and *relative positions*. Given a signature σ. Then v_1 and v_2 share the same relative position if for every binary $R \in \sigma$ and every $x \in S$ we have that $R(x,v_1) \equiv R(x,v_2)$ and $R(v_1,x) \equiv R(v_2,x)$. A *sufficient set* $W \subseteq V(T)$ for a training set S and a set of formulas Φ (such as all quantifier-free formulas) is a set of nodes such that whenever there is a consistent hypothesis $[\![\varphi(x\,;\bar{v})]\!]^T$ with $\varphi \in \Phi$ and

$\bar{v} \in V(T)^\ell$, then there exists another consistent hypothesis $[\![\psi(x\,;\bar{v}')]\!]^T$ with $\psi \in \Phi$ and $\bar{v}' \in W^\ell$. This means that we can restrict ourself to search for consistent parameters from such a sufficient set. For the case of quantifier-free formulas, a set W is sufficient for S if it contains a representative for every possible relative position. Let W be defined by first closing S under lowest common ancestors and then taking the 2-neighborhood of that set. Then W is linear in $|S|$ since we consider binary trees and W is sufficient which can be shown by a simple case-distinction.

The learning algorithm then uses a brute-force process and tests every possible quantifier-free formula for every parameter setting $\bar{v} \in W^\ell$ for consistency. It is bound to find a consistent hypothesis since W is a sufficient set. The set W can be computed in time $\mathcal{O}(|S|^2)$, and each evaluation of a quantifier-free formula uses constant time, resulting in the total runtime of $\mathcal{O}\left(|S|^2 + |S|^\ell |S|\right)$. ◀

4 Monadic Second-Order Logic

In this section we consider learning of unary MSO formulas on trees. The ordering relation $\leq$ used in the previous chapter can be expressed in MSO, thus we do not include it in the signature here. That is we have a binary tree T and want to learn a unary relation $R \subseteq V(T)$ using an MSO formula $\varphi(x\,; y_1, \dots, y_\ell)$ and ℓ parameters $v_1, \dots, v_\ell \in V(T)$. We aim for simple concepts and therefore restrict the number of parameters ℓ and the quantifier rank q of φ. Let $\mathrm{MSO}[q, \ell+1]$ be the class of MSO formulas with $\ell+1$ free variables and quantifier rank up to q. Note that the class $\mathrm{MSO}[q, \ell+1]$ is finite up to equivalence for every fixed q and ℓ, such that we can iterate over all formulas from this set to find a consistent one.

We present an algorithm that separates the task of analyzing the background structure and building an *index* from the task of finding a consistent hypothesis based on that index. Formally, we have an indexing phase in which the algorithm has local access to the tree T, but not to the training set S, and produces an index $I(T)$. In the learning phase, the algorithm gets S and has local access to T and $I(T)$. Based on that input, it produces a formula $\varphi(x\,;\bar{y})$ and a parameter configuration $\bar{v}$ such that $[\![\varphi(x\,;\bar{v})]\!]^T$ is consistent with S. We call such an algorithm an *indexing algorithm*, the time needed to build the index *indexing time* and the time for the actual search of a consistent hypothesis based on that index *search time*. Note that such an algorithm is especially useful when performing multiple similar learning tasks on the same data as the index could be reused in that case. We show the following theorem which implies Theorem 2 stated in the introduction.

▶ **Theorem 6.** *Let $q, \ell \in \mathbb{N}$ be fixed. Given a tree T and a training set S that is consistent with an MSO formula $\vartheta(x\,;\bar{y}) \in \mathrm{MSO}[q, \ell+1]$, it is possible to find a consistent formula $\varphi(x\,;\bar{y}) \in \mathrm{MSO}[q, \ell+1]$ that is consistent with S in indexing time $\mathcal{O}(|T|)$ (without access to S) and search time $\mathcal{O}(|S| \log |T|)$.*

Technically, we only consider binary trees, however, the results also extend to unranked trees as there exist MSO interpretations between those classes increasing the quantifier-rank of the resulting formula by 1.

4.1 Decomposing trees into strings

In order to apply techniques based on monoid structures we introduce the notion of path decompositions. Given a tree $T = (V(T), E(T), \leq)$ and a partition $P_1, \dots, P_p$ of $V(T)$. Let $T[P_i]$ be the induced substructure of P_i on T defined by restricting $V(T), E(T)$ and $\leq$ to

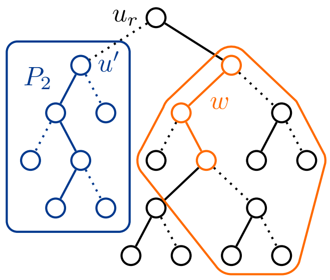

Figure 4 Heavy path decomposition of a tree, solid lines indicate the different heavy paths, the cut-off subtree of u_r is shown in blue and the dependent subtree of w is shown by the orange shape.

the elements from P_i. We consider the case that each induced substructure $T[P_i]$ is a string which implies that $\leq$ restricted to P_i is a (non-reflexive) total order. Let furthermore $T[u]_{\leq}$ be defined as $T[\{w \mid u \leq w\}]$, that is we restrict T to the induced subtree rooted at u.

One special case of decompositions into strings are *heavy path decompositions* introduced by Harel et al. [19]. We say that a node $u \in V(T)$ is at least as *heavy* as $v \in V(T)$ if $|\{w \mid u \leq w\}| \geq |\{w \mid v \leq w\}|$. That means that the heavier element is the one which is the root of the larger induced subtree. Using this definition, we can define the heavy path decomposition as follows.

The *heavy path* of T at $u \in V(T)$ is a path $(u_0, u_1, \ldots, u_\ell)$ such that $u_0 = u$, the node u_ℓ is a leaf and u_i is the leftmost child of u_{i-1} that is at least as heavy as all other children of u_{i-1} for every $i \leq \ell$. Let T be a binary tree and $P = (u_0, \ldots, u_\ell)$ a heavy path. Then the *cut-off subtree* at a node $u_i \in P$ with a child $u' \notin P$ is $T[u']_{\leq}$. The *dependent subtree* of a substring $w \subseteq P$ consists of w and the union of the cut-off subtrees for every $u \in w$. Note that the notions of dependent subtrees and cut-off subtrees refer to binary trees only. The heavy path decomposition $\mathrm{hp}(T) = \{P_1, \ldots, P_p\}$ of T is constructed by first computing P_1 as the heavy path of T at its root and then recursively computing the heavy paths of the cut-off subtrees for each $u \in P_1$. In Figure 4 the heavy path decomposition of a tree is shown. In this figure the blue box includes the cut-off subtree or the root u_r, the node u' is the cut-off child of u_r and the orange shape contains the dependent subtree of the string w. We use the following property of this decomposition in our algorithm.

▶ **Lemma 7** ([5]). *Every path from the root of T to a node $u \in V(T)$ touches at most $\log |T|$ different heavy paths of the heavy path decomposition $hp(T)$.*

4.2 DFAs simulating DTAs

We now describe how we can use path decompositions such as $\mathrm{hp}(T)$ and deterministic finite automata (DFAs) to simulate a tree automaton $\mathfrak{A}$ on T as shown in [5].

Let T be a tree, $\mathrm{hp}(T) = \{P_1, \ldots, P_p\}$ its heavy path decomposition and $\mathfrak{A}$ a tree automaton with states Q. Let ρ be the run of $\mathfrak{A}$ on T and $Q' = Q \mathbin{\dot\cup} \{q_0\}$. We define a DFA $\mathcal{A}$ and a function $f_\rho \colon V(T) \to \Sigma \times Q' \times \{\mathrm{L}, \mathrm{R}\}$ extending the labels of $V(T)$ by annotations about the state and direction of the cut-off subtree. Those extended labels from f_ρ allow us to run a DFA $\mathcal{A}$ on $\mathrm{hp}(T)$ which simulates the tree automaton $\mathfrak{A}$ on T.

▶ **Lemma 8.** *Let ρ be the run of the DTA $\mathfrak{A}$ on T and T' the tree we get from applying f_ρ to every node of T. Then we have for every $u \in P_i$ that $\mathcal{A}$ is in state q after reading $T'[P_i]$ up to position u, if and only if $\rho(u) = q$.*

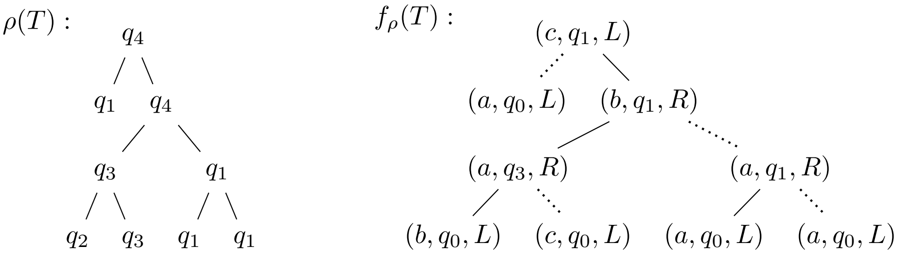

Figure 5 Applying the label transformation f_ρ to $\mathrm{hp}(T)$ for some tree T.

To match the evaluation order of bottom-up tree automata, the constructed DFA reads the paths $T'[P_i]$ in reversed order, that is starting from the leaves and running towards the root of T. The function f_ρ makes use of the run ρ of $\mathfrak{A}$ on T and is defined as follows. For a leaf u with label a in T we have $f_\rho(u) = (a, q_0, \mathrm{L})$. For an inner node $u \in P_i$ with label a and children $u' \in P_i$ and $u'' \notin P_i$ where u'' is a right child of u we have $f_\rho(u) = (a, \rho(u''), \mathrm{R})$ and correspondingly $(a, \rho(u''), \mathrm{L})$ if u'' was a left child of u. Intuitively f_ρ stores the state of the DTA $\mathfrak{A}$ on the cut-off subtree at u in the label of u. Figure 5 shows an example for f_ρ given a tree T and a run ρ of a DTA on T.

Let $\mathfrak{A} = (Q, \Sigma, \delta, F)$ be a deterministic bottom-up tree automaton and $\mathrm{hp}(T) = \{P_1, \ldots, P_p\}$ the heavy path decomposition of T. We assume that the transition relation δ of $\mathfrak{A}$ is split into $\delta_0 \colon \Sigma \to Q$ for leaf nodes and $\delta_2 \colon \Sigma \times Q \times Q \to Q$ for inner nodes. We now define the DFA $\mathcal{A} = (Q', \Sigma \times Q' \times \{\mathrm{L}, \mathrm{R}\}, \delta', q_0, F)$ where $Q' = Q \,\dot\cup\, \{q_0\}$ with Q, Σ and F taken from $\mathfrak{A}$. The transition relation δ' of the DFA $\mathcal{A}$ is defined as follows:

$$\begin{aligned}
\delta'(q_0, (a, q_0, d)) &= \delta_0(a) && \text{for all } a \in \Sigma, d \in \{\mathrm{L}, \mathrm{R}\} \\
\delta'(p, (a, q, \mathrm{L})) &= \delta_2(a, q, p) && \text{for all } a \in \Sigma,\ p, q \in Q \\
\delta'(p, (a, q, \mathrm{R})) &= \delta_2(a, p, q) && \text{for all } a \in \Sigma,\ p, q \in Q
\end{aligned}$$

Lemma 8 holds as at every position $u \in P_i$ the DFA $\mathcal{A}$ has access to the same information as $\mathfrak{A}$ through the extended alphabet.

Let $<_{\mathrm{hp}}$ be the partial order showing dependence between heavy paths of $\mathrm{hp}(T)$. That is, we have $P_j <_{\mathrm{hp}} P_i$ if P_j is part of the dependent subtree of P_i. For every node $u \in P_i$ with cut-off child $u' \in P_j$, the state $\rho(u')$ only depends on P_j with $P_j <_{\mathrm{hp}} P_i$ such that we can recursively compute ρ using $\mathcal{A}$ on $\mathrm{hp}(T)$ together with Lemma 8. We say that $\mathcal{A}$ accepts a tree T if it accepts the string $T[P_1]$ where $P_1 \in \mathrm{hp}(T)$ contains the root of T.

4.3 Monoids and factorization trees

A monoid $\mathcal{M} = \{M, \cdot_M, 1_M\}$ consists of a set of elements M and an associative multiplication operator $\cdot_M$ with neutral element 1_M. We usually refer to the monoid $\mathcal{M}$ by its set of elements M and write $m_1 m_2$ or $m_1 \cdot m_2$ for $m_1 \cdot_M m_2$. A monoid morphism $h \colon M \to M'$ is a function that translates a monoid in a consistent way, meaning that we have $h(m_1 \cdot m_2) = h(m_1) \cdot h(m_2)$ and $h(1_M) = 1_{M'}$.

The free monoid $M_{\mathrm{free}} = \{\Sigma^*, \cdot_{\mathrm{free}}, \varepsilon\}$ consists of all finite strings over Σ with concatenation as multiplication and the empty word ε as neutral element. A language $\mathcal{L}$ over Σ is *finitely monoid recognizable* if there is a finite monoid M, a monoid morphism $h \colon M_{\mathrm{free}} \to M$ and a subset $F \subseteq M$ such that $w \in \mathcal{L}$ if and only if $h(w) \in F$. A language $\mathcal{L}$ is finitely monoid recognizable, if and only if it is regular [29]. The *transition monoid* M of a DFA $\mathcal{A}$ with state space Q is defined as functions $m\colon Q \to Q$ with composition as multiplication, and the identity as neutral element. The corresponding monoid morphism h_M maps a string $w \in \Sigma^*$ to the function $m\colon Q \to Q$ modelling the effect of reading w on the states of $\mathcal{A}$. A monoid element m is *productive* if there are m', m'' with $m' \cdot m \cdot m'' \in F$. For further details on the equivalence between finite monoids and DFAs see for example [11, 7].

Every path $P_i \in \mathrm{hp}(T)$ induces a string $T[P_i]$ over Σ which we interpret as an element of the free monoid over the tree alphabet Σ. In the following we will not explicitly distinguish between the path P_i, the corresponding string $T[P_i]$ and the monoid element.

A *factorization tree* of a sequence of monoid elements $s = m_1, m_2, \ldots, m_n$ from a monoid M is a tree F_s where each node $u \in V(F_s)$ is labeled by an element m_u from M and the sequence of leaf labels is s. For each inner node $u \in V(F_s)$ labeled by m_u with children $u_1, \ldots, u_i$ which are labeled by $m_{u_1}, \ldots, m_{u_i}$ we have $m_u = m_{u_1} m_{u_2} \cdots m_{u_i}$. By choosing a balanced binary tree for F_s we get a factorization tree of height $\lceil \log |s| \rceil$. An alternative are *Simon factorization trees*, where each node u is either a leaf, a binary node or satisfies the property that all child nodes $u_1, \ldots, u_n$ of u share the same idempotent label m_u with $m_u = m_u \cdot m_u$.

▶ **Theorem 9** (Simon [31]). *For every sequence $s = [m_1, m_2, \ldots, m_n]$ of monoid elements from M, there is a Simon factorization tree of height at most $3|M|$. This factorization tree can be computed in time $n \cdot \mathit{poly}(|M|)$.*

In a set of *updates* $U = \{(i_1, m_1), \ldots, (i_s, m_s)\}$ with $i_j \in \mathbb{N}$ and $m_j \in M$ each tuple consists of an index and the new monoid element for this index. Applying U to a sequence of monoid elements s results a sequence s^U with $s^U[i_j] = m_j$ for $j \in \{1, \ldots, s\}$ and $s^U[j] = s[j]$ for all $j \notin \{i_1, \ldots, i_s\}$ where $s[i]$ denotes the ith element of the sequence s. We use the following lemma from [16] to apply such updates in Simon factorization trees.

▶ **Lemma 10** ([16]). *Given a Simon factorization tree F of height h over a sequence s of elements from M and a set U of updates. There is an algorithm returning a Simon factorization tree of height $2h + 3|M|$ for the updated sequence s^U in time $\mathcal{O}(|U|h + |U||M|)$.*

4.4 Constructing the necessary monoids and monoid morphisms

In the following we construct and analyze the monoid structure used in the learning algorithm in Section 4.5. Let let T be a tree over Σ and T_1 the same tree over $\Sigma_1 = \Sigma \times 2^{\{y_1,\ldots,y_\ell\}} \times \{?, N, P\}$ including information about the free variables of a formula $\psi(P, N, \bar{y})$ in the labels. Let $\Sigma_2 = \Sigma_1 \times Q' \times \{\mathrm{L}, \mathrm{R}\}$ be the alphabet from Section 4.2 which extends Σ_1 in order to work with string automata instead of tree automata. Let $f_\rho : V(T) \to \Sigma_2$ be the labeling function used in Lemma 8 and T_2 the tree that results from T by relabeling its nodes using f_ρ. Note that the concrete states Q', and therefore the whole construction, depend on the formula ψ that performs the consistency check.

▶ **Lemma 11.** *Let $\varphi(x\,;\bar{y})$ be an MSO formula and $\mathfrak{A}$ a tree automaton for φ with states Q. Let T be a tree over Σ with $hp(T) = \{P_1, \ldots, P_n\}$ and S a training set. Then there is*

- *an alphabet Σ_3,*
- *a construction transforming T to a tree T_3 over Σ_3 depending on $\mathfrak{A}$*
- *a monoid $\hat{\mathcal{M}}$ with a set $F \subseteq \hat{\mathcal{M}}$ of final elements*
- *and a monoid morphism $\hat{h} : \Sigma_3^* \to \hat{\mathcal{M}}$*

such that: $\hat{h}(T_3[P_1]) \in F$ if and only if there exists $\bar{v} \in V(T)^\ell$ such that $[\![\varphi(x\,;\bar{v})]\!]^T$ is consistent with S.

Proof. We start by defining the formula $\psi(P, N, \bar{y})$ which checks for a given training set S, split into positive (P) and negative examples (N), whether for some parameter vector $\bar{v} \in V(T)^\ell$ the hypothesis $[\![\varphi(x\,;\bar{v})]\!]^T$ is consistent with S:

$$\psi(P, N, \bar{y}) = \forall x(P(x) \to \varphi(x, \bar{y})) \wedge (N(x) \to \neg\varphi(x, \bar{y})).$$

The computation of the final monoid $\hat{\mathcal{M}}$ uses the following intermediate constructions.

1. DTA $\mathfrak{A}$ on $\Sigma_1 = \Sigma \times 2^{\{y_1,\dots,y_\ell\}} \times \{?, N, P\}$
2. DFA $\mathcal{A}$ on $\Sigma_2 = \Sigma \times 2^{\{y_1,\dots,y_\ell\}} \times \{?, N, P\} \times Q \times \{\mathrm{L}, \mathrm{R}\}$
3. Transition monoid $\mathcal{M}$ for $\mathcal{A}$ with final elements $F \subseteq \mathcal{M}$ and monoid morphism $h\colon \Sigma_2 \to \mathcal{M}$
4. Monoid morphism $h'\colon \Sigma_2' \to \mathcal{M}$ with $\Sigma_2' = \Sigma \times 2^{\{y_1,\dots,y_\ell\}} \times \{?, N, P\} \times \mathcal{M} \times \{L, R\}$
5. Monoid $\hat{\mathcal{M}} = 2^{\mathcal{M}}$ with elements $\hat{m} \subseteq \mathcal{M}$ for every $\hat{m} \in \hat{\mathcal{M}}$ and monoid morphism $\hat{h}\colon \Sigma_3 = \Sigma \times \{?, N, P\} \times \hat{\mathcal{M}} \times \{\mathrm{L}, \mathrm{R}\} \to \hat{M}$

The first step from ψ to the tree automaton $\mathfrak{A}$ is a standard construction (see e.g. [9]) that involves extending the tree alphabet Σ by unary relations for the free variables of ψ resulting in the alphabet $\Sigma_1 = \Sigma \times \{P, N, ?\} \times 2^{\{y_1,\dots,y_\ell\}}$. The second step in the construction consists of a translation from a DTA with states Q to a DFA with states $Q' = Q \,\dot{\cup}\, \{q_0\}$ which uses the construction from Lemma 8. The constructed DFA $\mathcal{A}$ is able to simulate a run of $\mathfrak{A}$ on the tree T_2 over the alphabet $\Sigma_2 = \Sigma_1 \times Q' \times \{\mathrm{L}, \mathrm{R}\}$ where $T_2 = T$ but the labels are given by f_ρ from Lemma 8. The monoid $\mathcal{M}$ computed in the third step is the transition monoid of $\mathcal{A}$ (for the construction see e.g. [33]), $h\colon \Sigma_2 \to \mathcal{M}$ is the corresponding monoid morphism and $F \subseteq \mathcal{M}$ is the set of accepting monoid elements. In the fourth step we construct the monoid morphism h' that works on $\Sigma_2' = \Sigma \times 2^{\{y_1,\dots,y_\ell\}} \times \{?, N, P\} \times \mathcal{M} \times \{\mathrm{L}, \mathrm{R}\}$; that is, we substitute the component containing for every node u the state q_u of $\mathcal{A}$ on the cut-off subtree by a monoid element $m_u \in \mathcal{M}$. Essentially this step works as we can extract the state q_u on the cut-off subtree at u from the monoid element m_u and then call the morphism h. With $\mathcal{M}$ and h' we can check whether the hypothesis $[\![\varphi(x\,;\bar{v})]\!]^T$ for some $\bar{v} \in V(T)^\ell$ is consistent with the training set S.

In the fifth and last step we construct the monoid $\hat{\mathcal{M}}$ and the monoid morphism $\hat{h}$ based on $\mathcal{M}$ and h' such that $\hat{h}$ can be used to check whether there is a consistent set $\bar{v} \in V(T)^\ell$ of parameters. We then use $\hat{h}$ and $\hat{\mathcal{M}}$ in the actual learning algorithm. Let $\hat{\mathcal{M}} = (2^{\mathcal{M}}, \cdot_{\hat{\mathcal{M}}}, \{1_{\mathcal{M}}\})$ be a structure with multiplication $\hat{m}_1 \cdot_{\hat{\mathcal{M}}} \hat{m}_2 = \{m_1 \cdot_{\mathcal{M}} m_2 \mid m_1 \in \hat{m}_1, m_2 \in \hat{m}_2\}$. $\hat{\mathcal{M}}$ is a monoid as it is closed under multiplication and $\{1_{\mathcal{M}}\} \in \hat{\mathcal{M}}$ is neutral for $\cdot_{\hat{\mathcal{M}}}$ because $1_{\mathcal{M}} \in \mathcal{M}$ is the neutral element of $\mathcal{M}$. Let $\Sigma_3 := \Sigma \times \{P, N, ?\} \times 2^{\mathcal{M}} \times \{L, R\}$. The final monoid morphism $\hat{h}\colon \Sigma_3^* \to \hat{\mathcal{M}}$ is given using the morphism h'. For $(a, \hat{m}, d) \in \Sigma_3$ with $a \in \Sigma \times \{P, N, ?\}$ and $d \in \{\mathrm{L}, \mathrm{R}\}$ we let

$$\hat{h}((a, \hat{m}, d)) = \left\{ h'((a, \bar{y}, m, d) \;\middle|\; \bar{y} \in 2^{\{y_1,\dots,y_\ell\}}, m \in \hat{m} \right\}.$$

For a given position $u \in V(T)$ labeled $(a, \hat{m}, d) \in \Sigma_3$, the definition of $\hat{h}$ can be read as calling h' for every possible distribution of parameters in the cut-off subtree indicated by $m \in \hat{m}$ and every choice of parameters $\bar{y}$ for that position. For a word $w \in \Sigma_3^*$ we split the word into its positions $w_1, \dots, w_n$ and compute the product $\hat{m} = \prod_{i=1}^n \hat{h}(w_i)$ of the monoid elements of those. This is possible since $\hat{h}$ is a monoid morphism from the free monoid to $\hat{\mathcal{M}}$. Note that the interpretation with the cut-off subtrees only makes sense for strings $w = T_3[P_i]$ defined by heavy paths or substrings of such a w.

Let w be the string from the heavy path P_1 containing the root of T. Since $\hat{h}$ tests all possible distributions of parameters in the dependent subtree, we know that if $\hat{h}(w) \cap F \neq \emptyset$ then there is a consistent parameter setting. Correspondingly there is no consistent parameter setting if $\hat{m}_r \cap F = \emptyset$. ◀

We now categorize the elements from $\mathcal{M}$ constructed in the third step of the proof of Lemma 11. Since $\mathcal{A}$ only accepts trees in which every parameter, indicated by a unary relation, is assigned exactly once we get that every productive monoid element $m \in \mathcal{M}$ contains the information which parameters have been read. This holds as otherwise there was an accepted tree where a single parameter has been assigned multiple times or not at all. Thus we can assign a set of parameters to every productive monoid element. We now formalize this idea.

▶ **Lemma 12.** *There is a function $p : \mathcal{M} \to 2^{\{y_1,\dots,y_\ell\}} \cup \{\bot\}$ that assigns a set $\bar{y} \in 2^{\{y_1,\dots,y_\ell\}}$ of parameters to each productive monoid element from $\mathcal{M}$ in a consistent way. That is, for a tree T and a substring w of a path in $hp(T)$ with $h'(w) = m$, exactly the parameters $p(m)$ occur in the dependent subtree of w.*

4.5 Algorithms

Using the monoids $\mathcal{M}$ and $\hat{\mathcal{M}}$ as well as the monoid morphism $\hat{h}$ from Lemma 11, we can compute a consistent parameter setting $\bar{v}$ for a given formula $\varphi(x\,;\bar{y})$ and a training set S. This proves Theorem 6. The presented algorithm is split in three parts, where the first part is done independent of S in the indexing phase of the algorithm. The remaining two parts, updating according to S and tracing the parameters, together make up the search phase of the indexing algorithm. The indexing part is linear in $|T|$, while the search phase takes time $\mathcal{O}((|S| + \ell)\log|T|)$ where the summands are for the updating and the tracing algorithm respectively. We assume that the underlying formula φ is fixed and therefore ignore the factors depending only on φ but have argued that those can be non-elementary due to exploding statespaces of the constructed automata.

The indexing algorithm

The indexing algorithm starts by computing the monoid $\hat{\mathcal{M}}$ from φ as described in Lemma 11 as well as the heavy path decomposition $\mathrm{hp}(T)$. It outputs the set of Simon factorization trees over $\hat{\mathcal{M}}$ for each P_i from $\mathrm{hp}(T)$ computed by the algorithm from [31]. Technically, the order of the computation of the factorization trees has to be consistent with the dependence relation $<_{\mathrm{hp}}$ as a label $a_u \in \Sigma_3$ of a node $u \in V(T)$ contains the monoid element $\hat{m} \in \hat{\mathcal{M}}$ of the cut-off subtree at u. As the computation of $\hat{\mathcal{M}}$ only depends on φ, the overall computation is dominated by the computation of the factorization trees in $|T| \cdot \mathrm{poly}(|\hat{\mathcal{M}}|)$.

The update algorithm

In the update part of the learning algorithm we add the information from S to the set of Simon factorization trees computed in the indexing part of the algorithm. The updates are performed bottom-up, that is we change the labels in the leafs each factorization tree belonging to positions from S and then propagate this information towards the root of T. The presented algorithm works in two stages: an outer stage that collects all updates for each heavy path and an inner stage that actually performs the update.

The outer stage orchestrates the update process by computing the set of updates U_{P_i} for every $P_i \in \mathrm{hp}(T)$. The set U_{P_i} consists of updates for every $u \in P_i$ occurring in S as well as (possible) updates from paths P_j with $P_j <_{\mathrm{hp}} P_i$ due to previous updates. Updates originating from S change the component $\{P, N, ?\}$ of Σ_3 while updates induced from changes in the cut-off subtree of $u \in P_i$ change the monoid element $\hat{m}$ in the label of u. By updating the factorization trees in an order consistent with $<_{\mathrm{hp}}$ every heavy path needs up be updated at most once. The inner algorithm which actually performs the update is taken from Lemma 10. As every path from a node to the root touches at most logarithmically many heavy paths as stated in Lemma 7, the total runtime of the update part is in $\mathcal{O}(|S| \log |T|)$.

The tracing algorithm

The tracing algorithm gets the updated set of factorization trees $\mathcal{F} = F_{P_1}, \dots, F_{P_n}$ and computes a set of parameters $\bar{v}$ such that $[\![\varphi(x\,;\bar{y})]\!]^T$ is consistent with S. Let P_1 contain the root of T. The algorithm works in a top down manner, that is it starts in the root of F_{P_1} and traces the parameters downwards using Lemma 12. We again divide the algorithm into an inner and an outer algorithm where the inner algorithm traces the parameters similar to [16] in a single factorization tree and the outer algorithm orchestrates the search within $\mathcal{F}$.

Let $\hat{m}_r \in \hat{\mathcal{M}}$ be the monoid element reached in the root of F_{P_1}. The tracing starts by choosing the (initial) local target $m_r \in \hat{m}_r \cap F$ for the root of F_{P_1}. Within each factorization tree F_{P_i} at a position u with local target m_u we select monoid elements m_1, m_2 from the children u_1, u_2 of u such that $m_1 m_2 = m_u$ using a brute-force test. The tracing continues for those u_i where $p(m_i) \neq \emptyset$ with m_i as local target at u_i until the leaves of F_{P_i} are reached and outputs pairs $(u, m_u) \in V(T) \times \mathcal{M}$ with $p(m_u) \neq \emptyset$. Then the outer part of the algorithm decides which of the parameters $p(m_u)$ are placed in P_i and which are further traced in cut-off subtrees with $P_j <_{\text{hp}} P_i$. The algorithm then selects a target monoid element $m \in \mathcal{M}$ for the root of F_{P_j} and calls the inner tracing algorithm until every parameter is placed in some heavy path (which happens eventually as T is finite). The choice of the target monoid element m for the root of F_{P_j} is done by testing at the node $u \in P_i$ every possibility of reachable monoid m in the cut-off subtree while fixing the remaining parameters $K = p(m_u) \setminus p(m)$ at u. This check is possible since the label of u contains a monoid element $\hat{m}_u$ that consists of all reachable monoid elements in the cut-off subtree of u.

The parameter configuration $\bar{v}$ found this way is consistent with S as it is computed in a way that $h'(T_2[P_1]) = m_r \in F$ which is accepted. For each parameter y_i, the inner algorithm uses constant time within each factorization tree and is called at most $\log |S|$ times by Lemma 7 stating that every path from a node u to the root u_r touches at most $\log |S|$ heavy paths. Together with the indexing and update algorithm this is means that we can find a consistent parameter setting $\bar{v}$ for φ in indexing time $\mathcal{O}(|T|)$ and search time $\mathcal{O}(\log |T| \cdot |S|)$.

5 Online learning of MSO formulas

For Theorem 6 we assumed that after an indexing phase the complete training set S is known and the learning task is to find a hypothesis consistent with S. We now lift this result to an online setting where we again have a linear indexing phase and then on input of new batch of examples S_i the algorithm updates its hypothesis H such that H_i is consistent with all examples $S = \bigcup_i S_i$ it has seen so far. This online setting allows examples to arrive over time which is natural for many tasks with human interaction. Note that the algorithm can also handle label updates of the nodes as both types of updates induce a label change in the tree T. Label changes are a common type of update in a database setting since updating the attributes of an already present entity can be modeled by an update of the entity's label.

An *online learning algorithm* is an algorithm that takes as input a background structure T, an index $I(T)$ and a sequence $S = (u_1, c_1), (u_2, c_2), \dots$ of training examples and outputs for every $i \leq |S|$ a hypothesis $H_i = [\![\varphi(x\,;\bar{v})]\!]^T$ consistent with $S_i = \{(u_1, c_1), \dots, (x_i, c_i)\}$.

▶ **Theorem 13.** *Let $q, \ell \in \mathbb{N}$. There is an indexing algorithm A that, given a tree T computes an index $A(T)$ and an online learning algorithm B that, given $T, I(T)$, and an $MSO[q, \ell+1]$-realizable sequence $S = (u_1, c_1), \dots$ for T, maintains a consistent hypothesis $H_i = [\![\varphi(x\,;\bar{v})]\!]^T$ for every $i \in \mathbb{N}$ such that A runs in time $\mathcal{O}(|T|)$ and B runs in time $\mathcal{O}(\log^2(|T|))$ per update.*

This can be achieved by substituting Simon factorization trees by the conceptually simpler binary factorization trees in the construction. This implies two main differences. First, we can update labels arbitrarily often without changing the structure of the factorization tree (which does not hold for Simon factorization trees due to the idempotent elements). Second, the height of each factorization tree is logarithmic in its length, i.e. at most logarithmic in T. Therefore it takes logarithmic time to update each path and since a single update may involve updating logarithmically many paths this results in a runtime of $\mathcal{O}(|S_i| \log^2(|T|))$ per update.

6 Conclusion

We considered the setting of learning quantifier-free and MSO formulas on trees. All learning algorithms provided in this paper search for consistent hypotheses, thus they can be turned into PAC learning algorithms by providing a large enough training set.

We assumed the background structures to be huge and therefore have been researching sublinear algorithms which access to the background structures through the local access oracles. The first result is that even for quantifier free formulas there is no sublinear learning algorithm. However, there is a sublinear learning algorithm when given access to the largest common ancestor of two nodes. Our main result is a learning algorithm for unary MSO formulas which uses a linear indexing phase to build up an auxiliary structure (the index) and admits a logarithmic learning time with local access to that index.

Further research questions might include lifting the result to higher dimensions where examples consist of pairs or tuples of nodes instead of single positions in the tree. Another direction of research could be to extend our results for tree-like structures. For structures of bounded tree-width the approach could use a similar structure as the one from [12]. A slightly different research question would be to look for approximate solutions where only a certain (relative) amount of examples needs to be consistent. Such approaches could also deal with faulty examples, which occur quite regularly in practice.

References

1 A. Abouzied, D. Angluin, C.H. Papadimitriou, J.M. Hellerstein, and A. Silberschatz. Learning and verifying quantified boolean queries by example. In R. Hull and W. Fan, editors, *Proceedings of the 32nd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 49–60, 2013.

2 D. Angluin. On the Complexity of Minimum Inference of Regular Sets. *Information and Control*, 39(3):337–350, 1978.

3 D. Angluin. Learning Regular Sets from Queries and Counterexamples. *Information and Computation*, 75(2):87–106, 1987.

4 D. Angluin. Negative Results for Equivalence Queries. *Machine Learning*, 5:121–150, 1990.

5 A. Balmin, Y. Papakonstantinou, and V. Vianu. Incremental validation of XML documents. *ACM Trans. Database Syst.*, 29(4):710–751, 2004.

6 A. Blumer, A. Ehrenfeucht, D. Haussler, and M.K. Warmuth. Learnability and the Vapnik-Chervonenkis dimension. *Journal of the ACM (JACM)*, 36:929–965, 1989.

7 M. Bojańczyk. Algorithms for regular languages that use algebra. *SIGMOD Record*, 41(2):5–14, 2012.

8 A. Bonifati, R. Ciucanu, and S. Staworko. Learning Join Queries from User Examples. *ACM Trans. Database Syst.*, 40(4):24:1–24:38, 2016.

9 J Richard Büchi. Weak second-order arithmetic and finite automata. *Mathematical Logic Quarterly*, 6(1-6):66–92, 1960.

10 W.W. Cohen and C.D. Page. Polynomial Learnability and Inductive Logic Programming: Methods and Results. *New generation Computing*, 13:369–404, 1995.

11 T. Colcombet. Green's Relations and Their Use in Automata Theory. In *Language and Automata Theory and Applications - 5th International Conference, LATA 2011, Tarragona, Spain, May 26-31, 2011. Proceedings*, volume 6638 of *Lecture Notes in Computer Science*, pages 1–21. Springer, 2011.

12 B. Courcelle. The monadic second-order logic of graphs. I. Recognizable sets of finite graphs. *Information and computation*, 85(1):12–75, 1990.

13 F. Drewes and J. Högberg. Learning a regular tree language from a teacher. In *Developments in Language Theory*, pages 279–291. Springer, 2003.

14 P. Garg, D. Neider, P. Madhusudan, and D. Roth. Learning invariants using decision trees and implication counterexamples. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 499–512, 2016.

15 E.M. Gold. Complexity of Automaton Identification from Given Data. *Information and Control*, 37(3):302–320, 1978.

16 M. Grohe, C. Löding, and M. Ritzert. Learning MSO-definable hypotheses on strings. In *International Conference on Algorithmic Learning Theory, ALT 2017, 15-17 October 2017, Kyoto University, Kyoto, Japan*, pages 434–451, 2017.

17 M. Grohe and M. Ritzert. Learning first-order definable concepts over structures of small degree. In *Proceedings of the 32nd ACM-IEEE Symposium on Logic in Computer Science*, 2017.

18 M. Grohe and G. Turán. Learnability and definability in trees and similar structures. *Theory of Computing Systems*, 37(1):193–220, 2004.

19 Dov Harel and Robert Endre Tarjan. Fast algorithms for finding nearest common ancestors. *siam Journal on Computing*, 13(2):338–355, 1984.

20 C. Jordan and L. Kaiser. Machine Learning with Guarantees using Descriptive Complexity and SMT Solvers. *ArXiv (CoRR)*, 1609.02664 [cs.LG], 2016. arXiv:1609.02664.

21 M.J. Kearns and L.G. Valiant. Cryptographic Limitations on Learning Boolean Formulae and Finite Automata. *Journal of the ACM*, 41(1):67–95, 1994.

22 J.-U. Kietz and S. Dzeroski. Inductive Logic Programming and Learnability. *SIGART Bulletin*, 5(1):22–32, 1994.

23 C. Löding, P. Madhusudan, and D. Neider. Abstract Learning Frameworks for Synthesis. In M. Chechik and J.-F. Raskin, editors, *Proceedings of the 22nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 9636 of *Lecture Notes in Computer Science*, pages 167–185. Springer Verlag, 2016.

24 S. Muggleton. Inductive logic programming. *New Generation Computing*, 8(4):295–318, 1991.

25 S.H. Muggleton, editor. *Inductive Logic Programming*. Academic Press, 1992.

26 S.H. Muggleton and L. De Raedt. Inductive Logic Programming: Theory and methods. *The Journal of Logic Programming*, 19-20:629–679, 1994.

27 J. Oncina and P. García. Identifying regular languages in polynomial time. In *Proceedings of the International Workshop on Structural and Syntactic Pattern Recognition*, volume 5 of *Machine Perception and Artificial Intelligence*, pages 99—-108. World Scientific, 1992.

28 L. Pitt and M.K. Warmuth. The Minimum Consistent DFA Problem Cannot be Approximated within any Polynomial. *Journal of the ACM*, 40(1):95–142, 1993.

29 M.O. Rabin and D.Scott. Finite Automata and Their Decision Problems. *IBM Journal of Research and Development*, 3:114–125, 1959.

30 R.L. Rivest and R.E. Schapire. Inference of Finite Automata Using Homing Sequences. In *Machine Learning: From Theory to Applications*, volume 661 of *Lecture Notes in Computer Science*, pages 51–73. Springer, 1993.

31 I. Simon. Factorization Forests of Finite Height. *Theoretical Computer Science*, 72(1):65–94, 1990.

32 Sławek Staworko and Piotr Wieczorek. Learning twig and path queries. In *Proceedings of the 15th International Conference on Database Theory*, pages 140–154. ACM, 2012.

33 W. Thomas. Languages, Automata, and Logic. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages*, volume 3, pages 389–456. Springer-Verlag, 1997.

34 L.G. Valiant. A theory of the learnable. *Communications of the ACM*, 27(11):1134–1142, 1984.

35 V. Vapnik and A. Chervonenkis. On the uniform convergence of relative frequencies of events to their probabilities. *Theory of Probability and its Applications*, 16:264–280, 1971.

36 Y. Weiss and S. Cohen. Reverse Engineering SPJ-Queries from Examples. In *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 151–166. ACM, 2017.