Machine Learning, 44, 211–243, 2001

On Exact Learning of Unordered Tree Patterns

THOMAS R. AMOTH amotht@cs.orst.edu
PAUL CULL pc@cs.orst.edu
PRASAD TADEPALLI tadepall@cs.orst.edu
Department of Computer Science, Oregon State University, Corvallis, OR 97331, USA

Editors: Peter Flach and Sašo Džeroski

Abstract. Tree patterns are natural candidates for representing rules and hypotheses in many tasks such as information extraction and symbolic mathematics. A tree pattern is a tree with labeled nodes where some of the leaves may be labeled with variables, whereas a tree instance has no variables. A tree pattern matches an instance if there is a consistent substitution for the variables that allows a mapping of subtrees to matching subtrees of the instance. A finite union of tree patterns is called a forest. In this paper, we study the learnability of tree patterns from queries when the subtrees are unordered. The learnability is determined by the semantics of matching as defined by the types of mappings from the pattern subtrees to the instance subtrees. We first show that unordered tree patterns and forests are not exactly learnable from equivalence and subset queries when the mapping between subtrees is one-to-one onto, regardless of the computational power of the learner. Tree and forest patterns are learnable from equivalence and membership queries for the one-to-one into mapping. Finally, we connect the problem of learning tree patterns to inductive logic programming by describing a class of tree patterns called Clausal trees that includes non-recursive single-predicate Horn clauses and show that this class is learnable from equivalence and membership queries.

Keywords: ILP, tree patterns, exact learning, learning from queries

1. Introduction

In many domains such as natural language processing, information extraction, and symbolic mathematics, examples are naturally described as trees—parse trees or expression trees. It is natural to abstract from trees and to represent sets of trees by "tree patterns." In Amoth, Cull, and Tadepalli (1998), we studied the learnability of ordered trees, that is, trees in which the order of the subtrees of a tree is fixed. In this paper we investigate the learnability of tree patterns where the subtrees are unordered.

We consider Angluin's exact learning framework (Angluin, 1988). In this framework, the teacher chooses an arbitrary target from the target class, which the learner seeks to exactly identify. The learner might have access to a number of oracles including an Equivalence Query (EQ), Subset Query (SQ), Membership Query (MQ) and Superset Query (SupQ). These oracles respectively answer if the argument to the query is equivalent to, is a subset of, is a member of, or is a superset of the target. The task of the learner is to identify the target concept in time and number of oracle calls polynomial in the size of the target and the outputs of the oracles.

A tree instance is a tree whose nodes are labeled by constant symbols. A tree pattern is a tree with labeled nodes, where the leaf nodes may be labeled with constants or variables.

A constant label only matches itself, but a variable label matches any tree, including a constant. To match a tree pattern with an instance its variables should be replaced by subtree instances. For the substitution to be consistent, all occurrences of the same variable in the tree pattern should be substituted with the same subtree. There are different ways of matching a tree pattern with an instance after a substitution is done. The learnability of tree patterns depends on the semantics, i.e., the type of mapping from the subtrees of the tree pattern to those of the instance. The two main types of mappings we consider are one-to-one onto and one-to-one into. Both of these mappings are one-to-one in that each pattern subtree maps to exactly one instance subtree and each instance subtree is mapped to by at most one pattern subtree. In a one-to-one onto mapping all subtrees of the instance tree must be mapped to by some pattern subtree.

Tree patterns are known to be learnable from equivalence queries when the subtrees of the tree pattern are ordered and are mapped to subtrees of the instance in the same order by a one-to-one onto mapping (Ko, Marron, & Tzeng, 1990; Goldman & Kwek, 1999). Finite unions or "forests" of ordered tree patterns are learnable from equivalence and membership queries (Page & Frisch, 1992; Page, 1993; Arimura, Ishizaka, & Shinohara, 1995; Amoth, Cull, & Tadepalli, 1998). The learning problem for the one-to-one onto mapping is considerably more difficult when the subtrees are not ordered. We first show that unordered tree patterns are not learnable from equivalence and subset queries when the mapping between subtrees is one-to-one onto regardless of the computational power of the learner. Because subset queries are strictly more powerful than membership queries, this nonlearnability result also holds for equivalence and membership queries. One way to get around the difficulty of this learning problem is to use a more powerful oracle. Indeed, we show that unordered tree patterns and their finite unions are learnable from equivalence and superset queries. Changing the mapping type from one-to-one onto to one-to-one into also eases the difficulty of learning. We show that tree and forest patterns are learnable from equivalence and membership queries for the one-to-one into mapping.

We relate logical subsumption to a third kind of matching semantics between subtrees. Subtrees correspond to terms and atoms in logic. A set of atoms, A, θ-subsumes another set B, if the variables in A can be substituted with terms yielding a subset of B. Hence θ-subsumption between sets of atoms naturally corresponds to many-to-one into mapping from pattern subtrees to instance subtrees. In other words, more than one pattern subtree can be mapped to the same instance subtree, and some instance subtrees may be left out. Thus a disjunction of atoms can be represented with a tree whose subtrees are matched by a many-to-one mapping to instance subtrees. Since the arguments of predicates and functions should be matched in the same order, they correspond to ordered subtrees. Hence, to capture the predicate clauses, we introduce a class of tree patterns called Clausal trees with two different types of nodes: the root node has unordered subtrees which match according to many-to-one-into semantics and all the other nodes (below the root) have ordered subtrees that correspond to atoms and terms and match according to ordered one-to-one onto semantics. This makes it possible to transfer some results from Inductive Logic Programming (ILP) to learning tree patterns. In particular, the algorithm of Reddy & Tadepalli (1999) to learn single-predicate non-recursive Horn clauses can be used to learn the above Clausal trees and forest patterns.

The rest of the paper is organized as follows: Section 2 presents the formal definitions of various classes of tree patterns and outlines our learning framework. Section 3 shows that unordered tree patterns are not learnable from equivalence and subset queries for one-to-one onto semantics. Section 4 shows that unordered trees and forests are learnable from equivalence and superset queries for one-to-one onto semantics. Section 5 shows that unordered trees and forests are learnable from equivalence and membership queries for one-to-one into semantics. Section 6 shows the relationship of tree patterns to predicate clauses. Section 7 concludes the paper by summarizing the results, discussing their practical importance, and suggesting future work.

2. Formal preliminaries

A *tree* has a root node and zero or more subtrees, each of which is a tree. All trees are *finite*. The root nodes of the immediate subtrees of any tree are the children of the root of the that tree. A node with no children is a *leaf*. A node with children is an *internal node* of the tree. A tree t with a root label L_0 and subtrees $t_1, \ldots, t_k$ may be written as $L_0(t_1, \ldots, t_k)$. If $k = 0$, it is simply written as L_0.

The internal nodes of all trees are constant labels chosen from a *infinite label alphabet*. A *tree instance* is a tree without any variables so all nodes including the leaves are constant labels from the label alphabet. A *tree pattern* is a tree that may have variables or constant labels at its leaf nodes.

A *substitution* σ is a set of pairs v_i/t_i, where v_i is a variable, t_i is a tree, and each variable v_i appears at most once in σ. In this case, we write $t_i = v_i\sigma$. If t is a tree pattern and σ is a substitution, the substituted tree pattern $t\sigma$ is the tree pattern (or instance) obtained by replacing each occurrence of each variable v_i in t with $v_i\sigma$. Note that the above definition implies that a substitution always replaces the same variable with the same tree pattern.

The rules for how/when a tree pattern is matched to an instance are determined by the *match semantics*. All semantics allow constant labels to match themselves. But nodes with variable labels can match any tree instance. By convention, we use small letters at the end of the alphabet such as x, y, and z to stand for variables, and uppercase letters at the beginning of the alphabet such as A, B, and C to stand for constant labels. We also introduce a special symbol $\perp$ called "*bottom*" denoting a pattern that matches no tree instances, and hence represents an empty set. A tree pattern consisting of just a single variable matches the entire instance space. A tree pattern represents the set of all tree instances that match it according to a given matching semantics.

Every subtree in the tree pattern must correspond to some subtree in the tree instance. The different matching semantics are distinguished by how the subtrees of the tree pattern are allowed to correspond to the subtrees of a tree instance. Matching can be *ordered*, meaning the corresponding subtrees are in the same order in the pattern and instance or *unordered*, meaning the corresponding subtrees may be in any order. The target can be required to be a single tree or be allowed to be a union of trees—a *forest*. These specifications define four classes: ordered tree, ordered forest, unordered tree, and unordered forest.

The semantics are also classified according to the kind of mapping allowed between subtrees of corresponding trees. In a *many-to-one* mapping each pattern subtree maps to

exactly one instance subtree. This is true in *one-to-one* mappings as well, but it is also required that each instance subtree is mapped to by at most one pattern subtree. *One-to-one onto* or *bijective* mappings are further restricted by having every instance subtree mapped to by some pattern subtree. Neither of the into mappings has this restriction; *one-to-one into* semantics is also an *injective mapping*, and *many-to-one into* is an *unrestricted mapping*.

For ordered trees, we only consider one-to-one onto maps between subtrees, and define match as follows.

Definition 1. A tree pattern γ matches a tree instance t (by ordered one-to-one onto semantics) iff there is a substitution σ for variables in γ such that $\gamma\sigma$ is identical to t.

For unordered trees, the definitions are more complicated and depend on how we define the mapping between the subtrees. We first define when a tree pattern maps to another (or a tree instance maps to another instance) without variable substitution.

Definition 2. A tree pattern or instance $r = r_0(r_1 \ldots r_k)$ maps to a tree pattern or instance $s = s_0(s_1 \ldots s_l)$ according to semantics Ψ (one-to-one onto, one-to-one into, or many-to-one into) iff the following conditions hold: (1) $r_0 = s_0$, (2) there is a corresponding (one-to-one onto or one-to-one into or many-to-one into respectively) mapping μ from $\{r_1, \ldots, r_k\}$ to $\{s_1, \ldots, s_l\}$ such that (3) the child subtree r_i recursively maps to some child subtree s_j according to semantics Ψ. (Where $1 \leq i \leq k$ and $1 \leq j \leq l$.)

In this case, we write $r_i\mu = s_j$ and $r\mu = s$, and call μ a Ψ-consistent mapping.

Note that μ can be viewed as a tree transformation or homomorphism. In the case of one-to-one onto maps, μ simply permutes the subtrees at all levels. In the case of one-to-one into maps, it permutes the subtrees as well as adds new subtrees at any level to get the instance tree. In the case of many-to-one into maps, μ can identify several subtrees into one, permute the subtrees, and add new ones. The homomorphism μ preserves edges (for each node r_f in r, $r_f\mu$ = some s_g in s and $\text{Parent}(r_f)\mu = \text{Parent}(s_g)$, unless r_f is the root of r) and node labels (constant or variable; $\text{label}(r_f) = \text{label}(s_g)$).

Definition 3. An unordered tree pattern Γ matches a tree instance (or pattern) t according to a given semantics Ψ, denoted by $\Gamma \succeq_\Psi t$, iff there is a substitution σ for its variables so that $\Gamma\sigma$ maps to t under Ψ. We omit the subscript Ψ when it is irrelevant or is clear from the context.

In figure 1, the tree pattern (a) matches instance (b) according to ordered one-to-one onto and all unordered semantics because the structure is basically the same and the identical subtrees $D(AC)$ are matched by the identical y variables. For ordered matching, E and F must be substituted for x and w, respectively. Pattern (a) matches instance (c) according to any unordered semantics (one-to-one onto, one-to-one into, or many-to-one into) by swapping the two subtrees and permuting the subtree with 3 children. Again, $D(AC)$ is substituted for y but in this match, E and F can be substituted for x and w in either order. The pattern (a) matches instance (d) using into (many- or one-) semantics. But this instance is not matched using one-to-one onto semantics because of the extra leaf C under the left

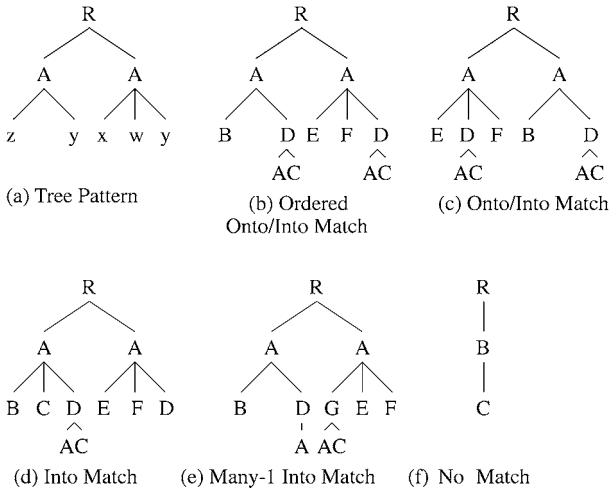

Figure 1. Match semantics example.

subtree as well as the two nodes labeled D having a different number of children. For either into semantics, an appropriate substitution is $\{z/B, y/D, x/E, w/F\}$ (note that the y's can match both D and $D(AC)$). Tree pattern (a) does not match instance (e) by one-to-one onto/into semantics because the number of children differ and there is no pair of identical subtree heads for the identical variables (y) to match. This match does work for many-to-one into semantics by making both pattern subtrees match the same instance subtree (which could be either the right or left subtree) by making two variables match the same instance child. One of numerous possible substitutions is $\{y/B, z/B, x/D, w/B\}$. With many-to-one into semantics, the tree pattern would match any instance having root R with at least one subtree headed by A which in turn has at least one child, i.e., $R(A(\text{ any label } \ldots) \ldots)$. Tree (f) is not matched for any semantics including many-to-one into because there is no A below the root to match. Any instance with only the root with one level of children would also not be matched.[1]

A tree pattern *represents* (according to a semantics Ψ) the set of tree instances that it matches (with Ψ). Hence we say that these instances are *in* the pattern. The instances that are in a given pattern are called its *positive examples*, and the instances that are not in a given pattern are its *negative examples*.

Definition 4. $L_\Psi(P)$ is the set of instances matched by tree pattern P with semantics Ψ (i.e., the language represented by P with Ψ).

The notation $L(P)$ without the Ψ will be used when it is clear from the context what semantics is being used.

We treat substitutions and mappings as left associative.

Definition 5. The composition of two substitutions, $\sigma_1 = \{x_1/s_1, \ldots, x_n/s_n\}$ and $\sigma_2 = \{y_1/t_1, \ldots, y_m/t_m\}$ is $\sigma_1 \circ \sigma_2 = \{x_1/s_1\sigma_2, \ldots, x_n/s_n\sigma_2\} \bigcup \{y_i/t_i |\ y_i \notin \{x_1, \ldots, x_n\}\}$.

The proofs of the following lemmas and other missing proofs are in the appendix.

Lemma 1. *For any tree pattern p,* $(p\sigma_1)\sigma_2 = p(\sigma_1 \circ \sigma_2)$.

Lemma 2. *The composition of two* Ψ*-consistent mappings is a* Ψ*-consistent mapping.*

Lemma 3. *For every pattern P, substitution* σ*, and* Ψ*-consistent mapping* μ *there exists a* Ψ*-consistent mapping* μ' *such that* $P\mu\sigma = P\sigma\mu'$.

Lemma 4. *If* $P \succeq Q$ *and* $Q \succeq R$*, then* $P \succeq R$.

Theorem 5. *Let P and Q be two tree patterns. Then* $L(P) \supseteq L(Q)$ *with semantics* Ψ *iff P matches* $Q(P \succeq_\Psi Q)$ *with* Ψ.

Proof : if: For any tree instance $I \in L(Q)$, $Q \succeq I$. This fact and $P \succeq Q$ implies $P \succeq I$ by Lemma 4, and $L \in L(P)$. Hence $L(P) \subseteq L(Q)$.

only if: Given $L(P) \subseteq L(Q)$. Since the alphabet of constant lables is infinite, we can choose an instance $I \in L(Q)$ with each variable in Q replaced with a constant label not appearing in either P or Q. Then for some σ, $Q\sigma = I$ with σ being a very simple substitution which replaces each variable with a constant. Since these constants appear nowhere else in these trees, there is an inverse substitution σ^{-1} which substitutes variables for constants (in a one-to-one onto fashion) such that $I\sigma^{-1} = Q$. Similarly $I \in L(P)$ implies $P\sigma'\mu' = I$ for some σ' and μ'. Therefore $P\sigma'\mu'\sigma^{-1} = Q$. By Lemma 3, there is a mapping μ'' so $P\sigma'\sigma^{-1}\mu'' = P(\sigma' \circ \sigma^{-1})\mu'' = Q$. Hence, $P \succeq Q$. □

We now formally define the exact learning framework of Angluin that we will be using in this paper (Angluin, 1988). We consider a universe of instances $\mathcal{I}$ that we call the *instance space*.

- An equivalence query $EQ(h)$ asks if a hypothesis h chosen from the given hypothesis space $\mathcal{H}$ is equivalent to the target, i.e., represents the same set of instances as the target. The query is answered 'yes' if they are equivalent and answered with a counterexample otherwise. The counterexample may be in h and not in the target or vice versa.
- A subset query $SQ(h)$ asks if the hypothesis $h \in \mathcal{H}$ represents a subset of the instances in the target. The query is answered 'yes' if it represents a subset and 'no' otherwise.
- A superset query $SupQ(h)$ asks if the hypothesis $h \in \mathcal{H}$ represents a superset of the instances in the target. The query is answered 'yes' if it represents a superset and 'no' otherwise.
- A superset query with a counterexample, $SupQc(h)$, behaves like $SupQ(h)$ but also returns a counterexample when the answer is 'no'.
- A membership query $MQ(x)$ asks if the input instance $x \in \mathcal{I}$ is a member of the target set of instances. The query is answered 'yes' if it is a member and 'no' otherwise.

We are now ready to define the exact learning framework of Angluin that uses a set of query oracles $\mathcal{Q}$ (Angluin, 1988).

Definition 6. A concept class $\mathcal{H}$ is exactly learnable with the help of a set of queries $\mathcal{Q}$ if there is a learning algorithm $\mathcal{A}$ and a polynomial p which meet the following conditions. Given any possible (target) hypothesis T in $\mathcal{H}$, $\mathcal{A}$ always finds a hypothesis which represents exactly the same set of instances in a number of queries and time bounded by $O(p(f, size(T)))$ using (only) the queries in $\mathcal{Q}$, where f is the maximum size of any counterexample returned by the query oracles.

The exact learning framework is stronger (more restrictive) than the Probably Approximately Correct (PAC) model of Valiant (1984) that requires that the examples are chosen using a fixed but unknown distribution. In the PAC learning model, the learner merely needs to learn a concept with approximates the target. In exact learning, the learner is required to learn the concept exactly, even though the teacher may choose the examples arbitrarily, It is known that if a concept class is exactly learnable in Angluin's framework from the EQ oracle, then it is PAC-learnable from examples (Angluin, 1988). Similarly, any positive results on learning with EQ and MQ transfer to PAC learning with MQ, and positive results on exact learning with EQ and SupQ transfer to PAC learning with SupQ.

To prove that various learning algorithms work on polynomial time, there is a need to show that not too many potential tree patterns are created compared to size of the target. Often this is easy to show if the concept class is *compact* in the sense that if a union of concepts covers a tree pattern, then one of the concepts in the union covers it.

Definition 7. A concept class $\mathcal{C}$ is compact iff for any $\mathcal{Z}$ and $\mathcal{V}_1, \ldots, \mathcal{V}_n \in \mathcal{C}$, $\bigcup_{i=1}^{n} L(\mathcal{V}_i) \supseteq L(\mathcal{Z}) \Rightarrow \exists i\, L(\mathcal{V}_i) \supseteq L(\mathcal{Z})$.

Trees are compact for all matching semantics:

Lemma 6. *The classes of* (*Ordered or*) *Unordered tree patterns* (*UT*) *with one-to-one onto*, *one-to-one into*, *and many-to-one into semantics are compact.*

Unfortunately, the matching problem is computationally hard for either onto or into semantics.

Lemma 7. *The problem of deciding whether an unordered tree pattern with repeated variables matches a tree instance with either onto or into semantics is NP-Complete* (*by reduction from CLIQUE*).

3. Nonlearnability of unordered onto-semantics

Ordered trees and forests have been shown to be learnable from equivalence and membership queries. In this section, we show that the unordered trees and unordered forests (with one-to-one onto semantics) are not learnable with these queries. The proof is based on a combinatorial argument that does not depend on any cryptography or complexity theoretic assumptions.

3.1. Unordered onto-trees are not EQ and SQ learnable

In this section, we show that unordered tree patterns for one-to-one onto matching semantics are not learnable with polynomially many queries regardless of the computational power of the learner. For this, we introduce a matrix notation for compactly representing trees.

A 3-matrix is a $3 \times n$ array of natural numbers. Two 3-matrices are *equivalent* if one can be changed into the other by permutations of the rows and columns. Two non-equivalent 3-matrices are called *distinct*.

A 2-matrix is a $2 \times n$ array of natural numbers. A 3-matrix is *consistent* with a 2-matrix if there are permutations of the columns of the 3-matrix so that one row of the 3-matrix is identical to one row of the 2-matrix and the sum of the other two rows of the 3-matrix is identical to the other row of the 2-matrix.

For example, the 3-matrix (a) in figure 2 is consistent with the 2-matrix (b). This is because permuting the last two columns and adding the bottom two rows of the 3-matrix gives the rows 5 4 2 and 0 1 3 which are the rows of the 2-matrix. One necessary (but not sufficient) condition for consistency is that the row totals of one matrix can be produced by adding a subset of the row totals for a matrix it is consistent with. Matrix (a) is not consistent with (c) because its row totals of 4, 10, and 1 can't be combined to produce 3 and 12 for (c).

For our purposes, we want to show that there can be a large number of distinct 3-matrices which are consistent with a given pair of 2-matrices.

Lemma 8. *There are $n!$ distinct 3-matrices which are consistent with the pair of 2-matrices:*

$$\begin{array}{cccccc} 0 & n & 2n & \dots & (n-1)n \\ c & c-n & c-2n & \dots & c-(n-1)n \end{array}$$

and

$$\begin{array}{cccccc} c & c-1 & c-2 & \dots & c-(n-1) \\ 0 & 1 & 2 & \dots & n-1 \end{array}$$

where $c \geq n^2 - 1$.

We now use this lemma to show that unordered trees are not polynomial time learnable from equivalence and subset queries. The idea is that a matrix corresponds to a tree pattern with 3 levels and consistency of matrices corresponds to matching of trees. The top of the

0 3 1	5 4 2	0 1 2
5 2 3	0 1 3	5 4 3
0 0 1		
(a)	(b)	(c)
Pattern	Match	No Match

Figure 2. Matrix representation of tree matching.

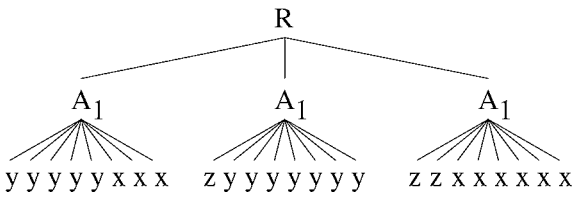

Figure 3. Sample UT target with $n = 3$ subtrees.

tree is a root with label, say R, and the second level has n nodes each with the same label, say A_1. A 3-matrix corresponds to a tree pattern with 3 distinct variables and each column of the matrix corresponds to a subtree. Each subtree rooted at one of these nodes labeled A_1 has some number of each of 3 variables, say x and y and z. The top element in each column is the number of x's in that subtree, the second element in each column is the number of y's in that subtree, and the bottom element is the number of z's. See figure 3 for a tree pattern with $n = 3$ subtrees, which corresponds to the matrix, $\begin{smallmatrix}3&0&6\\5&7&0\\0&1&2\end{smallmatrix}$. For a specified value of n, let $\mathcal{T}$, be the set of potential 3-variable targets formed by converting the $n!$ 3-matrices of Lemma 8 to tree patterns. A 2-matrix corresponds to an unordered tree with 2 distinct constant leaves and can be produced from a tree pattern by substituting one variable with a constant such as B and the other two variables with a constant C.

We can now show the unlearnability theorem.

Theorem 9. *Unordered trees (UT) under one-to-one onto semantics are not polynomial-time learnable from equivalence and subset queries.*

Proof: Let $\mathcal{H}$ (the set of targets consistent with the oracle answers and examples received so far) be initialized to $\mathcal{T}$. The argument of EQ must be a single tree (pattern), so the most useful examples that the learner can force EQ to yield are trees corresponding to two 2-matrices. If the learner tries to obtain any more examples by converting constant leaves in one of these examples to variables and calling EQ, the oracle returns the other example. Since all the targets in $\mathcal{T}$ are consistent with/match these two examples, they do not eliminate any hypothesis.

If the learner guesses any consistent 3-variable tree pattern and calls EQ, the oracle responds *NO* and returns the guessed 3-variable tree pattern with each of the 3 variables assigned a different constant as a negative counterexample (and eliminate that pattern from $\mathcal{H}$). (None of the types of guesses below eliminates any targets from $\mathcal{H}$.) Guesses inconsistent with the examples are handled by answering *NO* and returning one of the examples. Guesses with more than 3 variables are responded to in a similar fashion. For EQ queries with 1- or 2-variable guesses, answer *NO* and give one of the two examples. The SQ oracle behaves similarly. Each query eliminates at most one target from $\mathcal{H}$, so in the worst case $n! - 1$ guesses are needed. □

This result can also be proven using Angluin's "sunflower" lemma—Lemma 2 of Angluin (1988)—although it does not simplify the proof.

3.2. *Unordered onto-forests are not EQ and SQ learnable*

The proof for unordered tree nonlearnability does not directly apply to forests, when the target space is $\mathcal{T}$ in the previous section. Since the hypothesis can then consist of forests, the learner could call EQ with a forest of two trees and thereby obtain a third example. giving away the target. In fact, any fixed number of target variables is insufficient to prove UF is not learnable because the learner could force EQ to effectively yield all possible (exponentially many) ways of partitioning the set of target variables. These approaches to learning the target must be blocked by the availability of exponentially many examples.

The following subclass of UF will make it possible to have an exponentially large set of examples. Define $\mathcal{T}$, to be the set of 3-level tree patterns (not forests) each containing several (independent) sets of subtrees we call "blocks". Each block is of the form used in the UT nonlearnability proof with 3 variables, and n subtrees, each with $n^2 - 1$ children (as in figure 3 for $n = 3$). All blocks/subtrees share a common root. Make the number of blocks in each potential target tree also be n, for a total of n^2 subtrees, and $3n$ variables. Figure 4 shows a sample UF target for $n = 3$. The subtrees in the first block are identical to those used in a target for the UT proof (figure 3), and the other blocks are the same except for using different sets of 3 variables. As in the UT version, each block is derived by combining one example with a permuted version of a second example. In each UF target, the same permutation of the subtrees in the second example is used to generate all blocks. Each block will be consistent with the subtrees of 2 *2-constant examples* (examples using 2 distinct constants in the leaves of each block) having the same form as in the UT proof and will be independent of all other blocks in the same tree since each block was a distinct set of variables. Every target in $\mathcal{T}$ will therefore match every example in a set of 2^n 2-constant examples.

These blocks will be distinguished by using a different root label for the subtrees in each block, so the first block uses label A_1, and the second block could use label A_2, etc. Matching ambiguity between these blocks will then be eliminated, but the learning problem is still difficult due to the choice of matching of subtrees within each block. Any permutation within each of these groups of subtrees is still allowed for matching.

As in the UT proof, each block of the target is one of $n!$ possible choices which are consistent with the two example blocks. All blocks in each target are the same (employ the same permutation) except for variable renaming. Hence $\mathcal{T}$ has $n!$ targets by the same argument as for the UT proof. The strategy of the UF nonlearnability proof will be to use $\mathcal{T}$

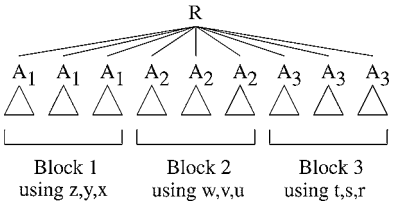

Figure 4. Sample UF target for $n = 3$.

as the adversary's set of potential targets, so up to 2^n counterexamples would be returned by EQ while SQ could eliminate at most one of $n!$ targets in $\mathcal{T}$.

Suppose a query with a 3-variable block is made. The query will be a subset of the target iff that block of 3 variables in the query is a subset of the corresponding block in the target. Each query with a 3-variable block will therefore be a subset of at most one target in the chosen set of targets.

Theorem 10. *UF with one-to-one onto semantics is not learnable with equivalence and subset queries.*

Proof: Let $\mathcal{H}$ be initialized to $\mathcal{T}$. The following invariants are preserved: (1) $\mathcal{H}$ represents the set of targets in $\mathcal{T}$ consistent with all the queries answered so far, i.e., the *version space*, (2) $\mathcal{H}$ is nonempty for any polynomial number of arbitrary queries.

The adversary will respond to a subset (or equivalence) query with an argument having at least one 3-variable block with the answer *no*. (A counterexample will be returned by choosing a simple 3-variable block and turning each variable into a constant.) At most one target in $\mathcal{H}$ will be eliminated because that 3-variable block can be a subset of only one target by the argument in the UT-nonlearnability proof.

Query arguments without any 3-variable blocks but which match the 2-constant examples give the following results. Each block in the argument will be matched by all $n!$ target blocks by the argument in the UT proof. The argument tree is matched by all targets in $\mathcal{H}$ and therefore does not eliminate any of the targets in $\mathcal{H}$. SQ will return *yes*. EQ with a forest hypothesis still cannot cover all 2^n 2-constant examples without having an argument of exponential size, so a new 2-constant example can always be returned as a positive counterexample.

Since the original size of $\mathcal{H}$ is $n!$ and at most one target is eliminated from $\mathcal{H}$ by each query, $\mathcal{H}$ is not exhausted by any polynomial number of queries, and UF is not (polynomial query) learnable. □

Corollary 11. *UT and UF with one-to-one onto semantics are not learnable with equivalence and membership queries* (*SQ trivially simulates MQ*).

The nonlearnability proofs of the UT and UF classes can also be viewed as showing that these classes do not have *polynomial certificates* (see Definition 4.1 of Hellerstein et al. (1996)) and are therefore not learnable by the contrapositive of Hellerstein, et al.'s Theorem 4.1.2. Polynomial certificates are a general, abstract criteria characterizing those concept classes which can be exactly learned with a polynomial number of queries.

When there are no repeated variables, unordered forests are easy to learn with with eqivalnce and either subset or membership queries by independently pruning different parts of the tree pattern. We call this class μ-UF in analogy to μ-formulas of propositional logic.

Theorem 12 (Amoth, Cull, & Tadepalli, 1998)**.** *μ-UF* (*UF without repeated variables*) *is learnable from EQ and SQ.*

4. Learning onto trees with superset queries

One way to circumvent the nonlearnability of unordered trees is to use more powerful queries. This section describes algorithms for learning unordered trees and forests (with one-to-one onto semantics) using superset queries (SupQ, or SupQc with counterexamples).

4.1. Unordered trees

The unordered tree learning algorithm (figure 5) uses a single training example as a template for forming the hypothesis tree pattern starting from the most general hypothesis (a single variable). The hypothesis tree pattern is refined in two stages. In the first stage, the **grow-tree** routine incrementally refines the hypothesis tree by adding subtree branches to the hypothesis as dictated by the example tree instance, while the result is accepted by SupQ. All the variables in the hypothesis are distinct during this stage. The routine recurses down the example tree by calling itself with each pair of corresponding children in the example and the pattern being formed. For example, in Figure 6, the training example (b) is generated for the target tree (a). The algorithm then forms a series of hypothesis trees (c)

```
% grow-tree grows the hypothesis tree until it is identical
% to the target except for variable names.
% s is the hypothesis subtree.
% p is the example subtree, initialized to an example tree instance.

% h is the hypothesis (global), initialized to the
% universal hypothesis (a single variable).
procedure grow-tree (s, p):
If p has at least one child
 then Expand s to include the top level of
    tree p with a new variable for each child.
    if SupQ(h)
        then for i = 1 to number of children of p
                Call grow-tree(subtree s_i of s,
                    subtree p_i of p)
        else restore s to a single variable
 else store constant label from p in s
    if not SupQ(h)
        then restore s to a single variable

procedure fuse-vars (h):  % finds variables identical
for each pair of variables in h   % in the target
    make the second one identical to the first
    if SupQ(h)
        then make the change permanent
        else undo the change
```

Figure 5. SupQ-based algorithm for UT.

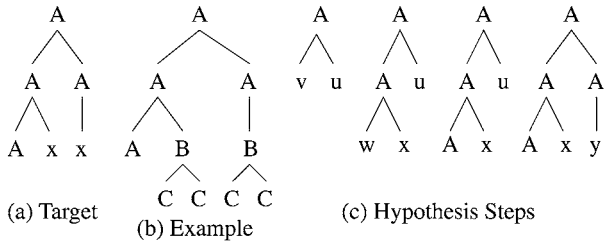

Figure 6. UT learning example.

by successively specializing its current hypothesis. A tree is first formed by copying the top level of the example and making each child be a distinct variable. Then subtrees are formed and specialized, and those specializations which are accepted by SupQ are kept. The result is the last tree in (c).

The following key lemma licenses specializing each part of such hypothesis tree pattern independently of other parts by **grow-tree**.

Lemma 13. *Let $h = h_0(h_1, \ldots, h_n)$ be a hypothesis tree pattern with no repeated variables. Let $t = t_0(t_1, \ldots, t_n)$ be a target tree pattern. Then $L(h) \supseteq L(t)$ iff $h_0 = t_0$ and there is a one-to-one onto mapping μ^* from $\{h_1, \ldots, h_n\}$ to $\{t_1, \ldots, t_n\}$, such that $L(h_i) \supseteq L(h_i\mu^*)$.*

When the **grow-tree** routine terminates, it will have found the most specific tree pattern that covers the target and has all variables different. The second stage, **fuse-vars**, determines which variables should be identical. In the example, the algorithm makes one variable to be the same as the other, the result is accepted by SupQ, and is equivalent to the target. In general, the algorithm works by fusing each pair of variables and asking a SupQ on the result. If the result is accepted by SupQ, the change is retained, otherwise it is undone (see figure 5). Any such identification of variables, once accepted by SupQ, need not be undone, as shown by the following theorem.

Theorem 14. *An unordered tree with repeated variables is learnable using SupQ and one-to-one onto semantics from a single arbitrary positive example.*

4.2. Learning unordered forests

In this section, we briefly sketch the algorithm that learns unordered forests using equivalence and superset queries (both with counterexamples). As in the UT learning algorithm, the hypothesis is repeatedly specialized until it matches the target.

The biggest problem facing the algorithm is that matching a counterexample to the hypothesis trees is too hard (Theorem 7). The algorithm therefore goes to extraordinary lengths to avoid the need for matching. A second problem is that the SupQc oracle will not give useful guidance when a tree is specialized (or "split" into several more specific trees) so the hypothesis is not close to covering the target, until additional specialized trees are

added. Further, the learner doesn't know beforehand how many trees are in the target, so there is no way for it to know how many specialized trees would be needed for some tree split to be successful.

The algorithm therefore faces a double difficulty. It doesn't know which tree (or even node) is too general when a negative counterexample is returned. Even if it could guess which tree/node to specialize, several new trees could be needed before that fact could be verified. All choices for specializing the trees by just one level must be tried in a breadth-first fashion until a specialization which works is found.

The first stage of the algorithm determines which 1-level trees are needed to cover the target. The algorithm starts with the universal hypothesis—a single variable. This hypothesis tree is then "split" into several more specialized 1-level trees (with variables as first-level children) using counterexamples supplied by SupQc until the target is covered.

For example, let the target concept be as shown in figure 7(a). Then SupQc is called to obtain examples to create the most general cover of the target other than just a single variable (b). To obtain this result, each example is effectively truncated by replacing the first-level children with variables. There is one tree pattern for each root-label/number-of-children combination which occurs in the target (notice that the target has 3 trees but (b) has only two). This result therefore completes the learning of the top level of the target. The resulting hypothesis trees are then specialized as much as possible while making the hypothesis still represent a superset of the target (c). These steps are relatively simple because it is always clear which hypothesis tree covers any given example.

Next, the main specialization loop is entered to specialize the trees to the second level. In these later stages which learn deeper levels of the tree, a breadth-first approach must be used to try specialization on all trees in the hypothesis forest. Breadth-first specialization must also be attempted on all parts of each tree pattern in the hypothesis. The example and hypothesis trees are unordered which creates a problem in deciding which parts of

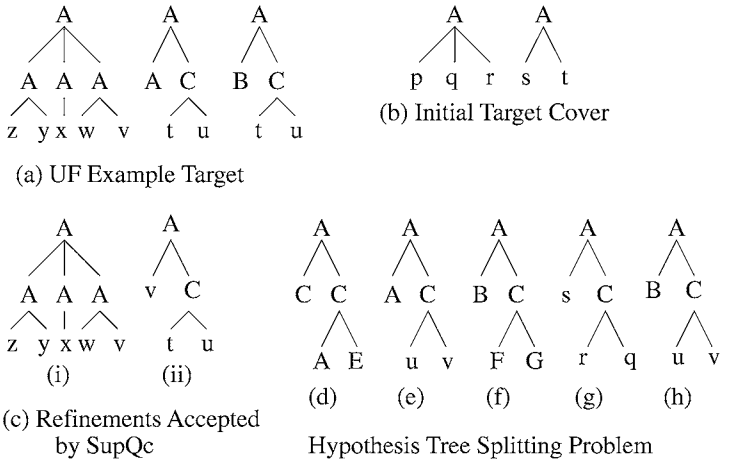

Figure 7. EQ + SupQc UF example.

the hypothesis trees correspond to the parts of the counterexample returned. This difficulty creates an obstacle for the entire specialization process. The following sequence is used for each attempt to specialize a particular tree pattern. The algorithm tries one of the possible ways to specialize that pattern p, then calls EQ to get a positive counterexample. That example is generalized until the entire hypothesis including the generalized example, but excluding the tree p again covers the target. This approach determines how to specialize p without knowing how its subtrees correspond to those of the example. The algorithm then backs up one step to make the new example more specialized than the pattern p.

Figure 7 also shows a target (a) in which two of the tree patterns are identical except for the constant in the left child. A single hypothesis tree of this same form with just a single constant in that leaf would be too specific and a variable in that position would be too general; the learner must discover that two (or more) separate tree patterns are needed to cover the target without being overgeneral. We call this difficulty the hypothesis tree splitting problem. The problem also occurs when a hypothesis tree has to be split into two or more trees with differing numbers of children in the affected subtree.

Tree (d) is a negative counterexample for the 2-level hypothesis (c) in that figure. The difficulty is that (d) has the same structure as two of the hypothesis trees in (a)—which are identical except for the constant leaves A and B. The algorithm has to specialize that child, but a single hypothesis tree is insufficient because two values are allowed for that constant. All specializations of the tree patterns in (c)—as guided by example from which these trees were originally derived—are tried breadth-first. Then the specialization which eventually leads to success is of the left subtree of tree (ii) in (c). Assume the example had an A as the left child. Specialization therefore produces (e). EQ is then called with the hypothesis forest containing only (e) and the first tree in (c). The positive counterexample (f) is returned by EQ.

The example will now be put through a process of first generalization and then specialization to make it into a new hypothesis tree (for the above choice which eventually succeeds in refining the hypothesis). That example is then made sufficiently general in (g) so that it together with the left tree in (c) (but without (e)) covers the target. The process is needed to make (f) as general as (ii) in (c) without having to perform tree matching (which is NP-Complete). (f) is therefore generalized until SupQ (call SupQc but ignore the counterexample) is satisfied and then specialized using the example, i.e., (f) itself, as a guide; the result is (g) which is equivalent to (ii).

Now the algorithm needs to specialize (g) to make it cover just one more constant in the leaf being specialized (left subtree of (ii)) without using a variable which would cover all possible constants and fail to achieve a split of that hypothesis tree. But there is a problem; the trees are unordered, so it is likely to be difficult for the learner to decide which leaf of (g) should be so specialized. The solution is to try all possible specializations and add them to this test hypothesis. Only the specializations involving one change will be used. Tree (h) is the specialization which eventually helps to refine the hypothesis (the other attempts are not shown).

New trees are added to each test hypothesis in a breadth-first search to find a hypothesis that again covers the target according to SupQ. That hypothesis is then used as the new hypothesis, unneeded trees (as judged by SupQ) are eliminated, and the remaining trees are

again specialized as much as possible (using the training example used to form each tree as a guide). This main specialization loop is repeated until EQ says the result is equivalent to the target.

Theorem 15 (Amoth, Cull, & Tadeppali, 1998)**.** *UF with one-to-one onto semantics is EQ + SupQc learnable.*

5. Learning under one-to-one into semantics

The difficulty of learning unordered tree patterns can also be circumvented by adopting one-to-one into semantics. In this section, we describe a bottom-up algorithm for unordered forests with one-to-one into semantics and give an analysis of the algorithm.

Recall that with one-to-one into semantics, each subtree of the tree pattern maps to a subtree of the instance, but some instance subtrees may not be mapped to by any pattern subtree. In other words, any node in the instance may have more children than the corresponding node in the tree pattern that matches it. In particular, a constant node with no children matches only itself under one-to-one onto semantics, but matches any tree with the same root label under one-to-one into semantics, i.e., A matches any $A(\ldots)$. A target with multiple copies of the same variable, e.g., $A(x\ x)$, would therefore match any tree instance having the same label at the root of the subtrees corresponding to those variables—for example $A(B\ B(C\ D))$. This is so because substituting $\{x/B\}$ in the tree pattern yields $A(B\ B)$ which maps to the example. Note that this semantics is slightly different from that of Amoth, Cull, and Tadepalli (1999), which required that both variables should map to exactly the same unordered subtrees. While the old algorithm is still correct with respect to its semantics, the new semantics is cleaner. For example, $\succeq$ is not transitive according to the old semantics, because $A(x\ x) \succeq A(B\ B)$ and $A(B\ B) \succeq A(\ B\ B(C\ D))$, and yet $A(x\ x) \not\succeq A(\ B\ B(C\ D))$. The new semantics obeys transitivity and is more natural than the old semantics. It also simplifies the algorithm of Amoth, Cull, and Tadepalli (1999) as will be shown below.

5.1. *One-to-one into algorithm description*

The main part of the learning algorithm for into-semantics unordered forests using EQ and SQ (equivalence and subset queries) is based on a bottom-up-from-single-example generalization approach and shown in figure 8. All generalizations are tested with SQ (subset query) and undone if not accepted. The algorithm gets a new example tree from EQ, and then applies two generalization routines: **prune** and **variablize**. **prune** removes the extra nodes and edges in the tree instance while making sure that it still remains a positive example of the target pattern. **variablize** turns the constants into variables while testing with SQ that the result is a subset of the target. There still remains the problem of finding a target with multiple variables when the example has the same constant label corresponding to all variables. This problem is solved by a third routine **partition** (called by **variablize**) that partitions the constants into groups that are instances of the same variable in the corresponding target tree pattern. Throughout this process, the hypothesis represents

```
function into-main()
initialize: h = {}
while EQ(h) gives
        counterexample t
        %(which is positive)
    h = h ⋃
        variablize(prune(t))
return h
```

```
procedure prune(t):
        % (example) tree t
while haven't tried leaf f
        cut f and its edge
        if not SQ(t)
            undo the change
return t
```

Figure 8. Into-semantics main (left) and pruning routines (right).

a subset of the target, so the validity of each step can be verified. The tree is then added to the hypothesis and the process repeats until the target is covered. We now describe the three subroutines in more detail.

Pruning: The pruning subroutine is shown in figure 8. Pruning operates by trimming each (constant) leaf. After each change, the hypothesis is tested to be sure it is still a subset of the target; and the change is undone if not. Once all the children of a node are pruned, that node becomes a leaf and is a candidate for pruning. The process repeats as long as pruning yields a tree pattern that is accepted by SQ.

Figure 9 illustrates how the simple pruning technique works. A possible target is given in (a) and single training example could be as in (b). The pruning bottom-up algorithm then attempts to generalize one leaf at a time by trimming the leaf and its edge. The result is the simplest such tree which still represents a subset of the target (c).

Variablizing: This routine (figure 10) picks each set of identically labeled constants or variables (including singleton sets), creates a new variable that corresponds to them, and calls **partition**. Partitioning is necessary because the identical constants may have arisen from substituting the same constant for multiple variables in the target pattern. The task of **partition** is to separate these constants into groups so that the constants in different groups are generated from different target variables. If **partition** is successful in partitioning this set into two sets, they become candidates for further partitioning (except for single, unique variables). When a set of two or more constants is converted to the same variable, partitioning is called on that set. This repeats until partitioning or variablization is unsuccessful on each

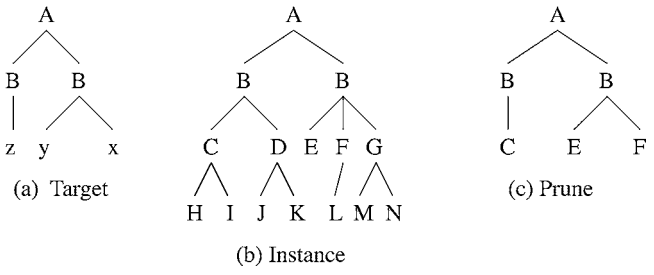

Figure 9. Bottom-up pruning sample execution.

```
procedure variablize(t)
        %generalize leaves to variables in tree t
    while there is a set of one or more leaves in t with identical labels c
            on which partition has not been called
        create a new variable v
        partition(t, c, v) %try partition/variablize c's
    return t

procedure partition(t, o, n):
  %partition old by adding new
designate the copies of o in t as o_1 ... o_k
for i = 1 to k
    add n_i (a copy of n) to the parent of o_i
flag = true   %ensure delete at least one n and one o by:
for i = 1 to k
    if flag then   %try deleting o's first until ...
        delete o_i
        if not SQ(t), then undo that change
            else flag = false   %... succeed—then ...
        delete n_i
        if not SQ(t), undo that change
     else              %... delete n's first
        delete n_i
        if not SQ(t), undo that change
        delete o_i
        if not SQ(t), undo that change
        %(do not eliminate all of one variable first)
return t
```

Figure 10. UF 1-to-1 into repeated variable partition algorithm.

remaining non-singleton set of identically labeled constants or variables. The actual turning of constants to variables occurs in **partition** as a side-effect.

Partitioning: The task of the **partition** routine is to split many copies of the same constant/variable in the hypothesis into two sets such that the two sets correspond to mutually exclusive sets of target variables. With one-to-one onto semantics, this problem is too difficult (Theorem 9). But one-to-one into semantics has an additional flexibility which permits this kind of set to be partitioned in polynomial time.

The approach used by **partition** introduces a new variable, say v, and converts part of the set of c's to this new variable—if possible. One-to-one into semantics permits a matched instance to have extra children not present in a matching pattern; therefore adding extra children (i.e., the v's) to the pattern will cause the latter to match a subset of the set of instances it previously matched. It is therefore possible to add a copy of v wherever c appears, and then eliminate one c or v at a time while doing meaningful subset tests. If all copies of a constant c really should be the same variable, **partition** fails to partition the set of c's, but is designed to favor changing all copies of c to v—if acceptable to SQ. Similarly, if there is only one copy of c, that copy will be converted to a variable, if appropriate. If only one copy of c corresponds to a target variable, then eventually successive applications of **partition** will separate this variable. The above cases cover all tasks, but there is also a

default case. If all copies of a constant c really are constants in the target, then **partition** will fail to do anything because all attempts to delete c's will be rejected by SQ and deletion of all introduced variables will be accepted.

Subroutine **partition** (figure 10) is used by routine **variablize** on one or more copies of the same constant or variable c. This routine uses the following technique to individually test each copy of c. For each copy of c, a copy of a new variable v is hung on the parent of c. For each subtree, the same number of v's is added as there are c's in that subtree. Therefore each subtree gets an equal number of c's and v's, and the total number of v's added in the entire tree is the same as the number of c's. The variables are then eliminated one at a time while checking that the resulting tree t is still accepted by SQ. If all copies of one variable were eliminated first, the result might be unchanged or all copies could be converted to the new variable. Therefore elimination alternates between the two sets of variables (or variable and constant). To ensure the set of c's will be split, if possible, the alternation first works by preferring to delete c before the corresponding v—until one c is successfully deleted. Then v is removed before its corresponding c is for the rest of the tree. Without this switch in the preference, either all the c's or all the v's might be removed even when it is possible to split the set.

Figure 11 shows the execution of a simple repeated-variable example. First the target is shown to be $A(A(xxx)\ A(xyy)\ A(xyz))$. The training example is assumed to be the same but with all variables substituted with the same constant, say C, which are all changed to the same variable (say s).

Further steps in the example will be shown with just the leaves since the upper part of these trees is the same. (The target and training example would be represented as $xxx\ xyy\ xyz$ and $CCC\ CCC\ CCC$ in this notation.)

The variable duplication step would then produce *sssrrr sssrrr sssrrr* (3 subtrees but now with 6 children each). The algorithm would then start on the first subtree and eliminate one variable at a time—first an s, then one of the r's, yielding *ssrr sssrrr sssrrr*. The first hypothesis subtree is no longer matched by the first target subtree, but it can still match the other two target subtrees. Then another s is eliminated, yielding *srr sssrrr sssrrr*. This first subtree can still match either the second or third target subtrees, so these hypotheses are all accepted by SQ. But further pruning will result in rejection—at least 3 children are necessary in all hypothesis subtrees to satisfy SQ.

Similar pruning of the second hypothesis subtree gives *srr srr sssrrr*. Eliminating one of each variable from the third subtree gives *srr srr ssrr*, but no subtree can match the 3 x's.

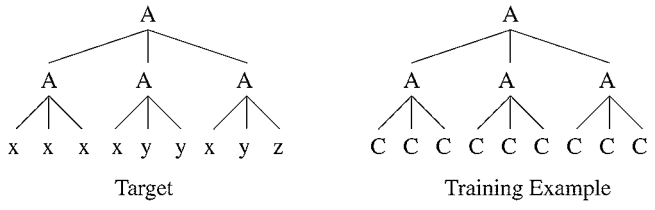

Figure 11. Partition/repeated Variables bottom-up exmple.

Backtracking and eliminating r's gives *srr srr sss*. Any attempt to further partition s only yields an equivalent tree. Partitioning r gives *srrww srrww sss* for the duplication step. Eliminating one of each gives *srw srrww sss* which is accepted showing this partition attempt was successful. Further pruning eliminates one variable (say r) in the second subtree, giving the target. But further partitioning must be attempted on any variable not already tested (just w).

Note that the algorithm has a "flag" which causes it to first eliminate the old/original children. Then once it has succeeded the algorithm switches modes to try eliminating the new child/variable first. This is to avoid eliminating all of one variable even when it is possible to partition the set of identical variables (e.g., for a target of the form *xxx yyy*).

5.2. *Proof of algorithm*

We will now give a correctness proof of our algorithm. The algorithm performs the following three tasks: it prunes all leaf nodes as much as possible, it converts constant leaves to variables, and it partitions sets of identical constants or variables as much as possible.

Lemma 16. *Routine* **prune** (*figure* 8) *will remove all nodes in the example tree not corresponding to nodes in the target tree.*

Lemma 17. *Given a pruned hypothesis tree t, which represents a subset of the target, an "old" leaf o in t and a "new" variable leaf n to replace o, routine* **partition** *will generalize the set of n's if that set is less general than some target tree.*

Lemma 18. *Routine* **variablize** *will learn one of the target trees from a pruned example.*

Theorem 19. *UT with one-to-one into semantics is learnable with equivalence and subset queries.*

Corollary 20. *UF with one-to-one into semantics is learnable with equivalence and subset queries.*

5.3. *One-to-one into semantics with membership queries*

Corollary 21. *UF with one-to-one into semantics is learnable with EQ and MQ.*

6. Relationship to predicate clause learning

In this section, we define a class of tree patterns that makes it possible to relate tree patterns to predicate clauses, the representation of choice for Inductive Logic Programming (ILP).

The primitives of predicate clauses are constants, variables, function symbols, and predicate symbols. Both constants and variables are considered as "terms". If f is a function symbol and $t_1, \ldots, t_k$ are terms, then $f(t_1, \ldots, t_k)$ is also a term. An atom is of the form

$p(t_1, \ldots, t_k)$, where p is a predicate symbol and $t_1, \ldots, t_k$ are terms. A literal is a positive or negated atom. A clause is a set of literals which are disjunctively combined. A definite Horn clause is a clause where exactly one literal is positive. It is also written as $p \rightarrow q$ where p is a set of positive literals and q is a positive literal.

There are many logical settings for predicate Horn clause learning, including Learning from Entailment, Learning from Interpretations, and Inductive Logic Programming (De Raedt, 1997). Of these, Learning from Entailment (LFE) is perhaps the easiest setting to map to learning tree patterns. In LFE, the target concept consists of a set of clauses, P. A positive example e of a target concept P is a clause which logically follows from P. We denote this by $P \models e$, and say that P entails e. Entailment is a semantic relationship. $P \models e$ means that in any world in which P is true, e is also true. A clause is a negative example of P if it is not a positive example.

Since tree patterns do not have a logical semantics, entailment cannot be mapped to trees. However, there is a "match semantics" for predicate clauses called θ-subsumption, which is related to what we called "many-to-one into semantics" for tree patterns.

Definition 8. Let D and E be two clauses viewed as sets of literals. A predicate clause D θ-subsumes a clause E if there exists a substitution θ such that $D\theta \subseteq E$. We denote this as $D \succeq E$, and read it as D subsumes E or as D is more general than E.

Note that θ-subsumption allows more than one literal in D to map to the same literal in E by substitution. Moreover, not all literals in E need to be mapped to by the literals in D. This implies that θ-subsumption at the level of clauses is closest to many-to-one semantics of tree patterns. However, at the lower levels of literals and terms, θ-subsumption requires the substituted terms to match exactly with the terms in the instance in the same order. This means that matching follows the one-to-one onto mapping of ordered trees at the lower levels. These observations motivate the following definition.

Definition 9. The class Clausal-Tree (CT) is the set of tree patterns that employ unordered many-to-one into mapping for the top level of the tree and ordered one-to-one onto semantics for all lower levels.

Definition 10. The least general generalization (lgg) of two clauses C_1 and C_2 is a clause C such that $C \succeq C_i$ for $i = 1, 2$ and for any clause D such that $D \succeq C_i$ for $i = 1, 2$, $D \succeq C$.

The *lgg* of clausal trees is analogously defined. Since multiple literals in the subsuming clause can match a single literal in the subsumed clause, computing the *lgg* of two clauses requires matching each literal in the first clause with each literal in the second and finding their *lgg* (Plotkin, 1970). This is described more fully as the **productlgg** algorithm (figure 12). Since literals are matched in one-to-one onto ordered fashion, the *lgg* of two literals is computed by matching the corresponding terms in the two literals in the same order and finding their least general generalization. Thus, the function **lgg** in figure 12 is the standard ordered-tree lgg algorithm in Amoth, Cull, and Tadepalli (1998) or LGG in the introduction

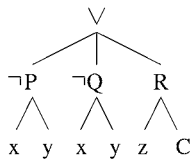

Figure 12. A clausal-tree equivalent to $\neg P(x, y) \vee \neg Q(x, y) \vee R(z, C)$.

of Page (1993), except variable substitutions must be combined so they apply to the Clausal tree as a whole. The following lemma claims the correctness of the **productlgg** algorithm.

Theorem 22 (Plotkin, 1970)**.** *The* **productlgg** *algorithm* (*figure* 12) *returns the least general generalization of two Clausal tree patterns.*

Predicate Horn clauses, e.g., $\neg P(x, y) \vee \neg Q(x, y) \vee R(z, C)$ can be represented by Clausal tree (figure 13), although they do not capture their logical semantics. In this figure, the root is the symbol for disjunction and its immediate subtrees match according to unordered many-to-one semantics. Each subtree represents a predicate. The children of the subtree are the predicate arguments and match according to ordered one-to-one onto semantics (as do all succeeding tree levels which represent functions in the predicate arguments).

For example, suppose we are given two examples of the clause, ¬Father(x, y) ∨ ¬Mother (y, z) ∨ GrandFather(x, z). Let the first example be ¬Father (John, MotherOf(Lisa)) ∨ ¬Mother(MotherOf(Lisa), Lisa) ∨ ¬Father(John, Sam) ∨ GrandFather(John, Lisa). Let the second example of the same clause be ¬Father(Peter, Alice) ∨ ¬Mother(Alice, Mary) ∨ ¬Father(Peter, Mark) ∨ GrandFather(Peter, Mary). Computing the lgg of these clauses using the **productlgg** algorithm yields, ¬Father(x, y) ∨ ¬Father(x, q) ∨ ¬Father(x, o) ∨ ¬Father(x, n) ∨ ¬Mother(y, z) ∨ GrandFather(x, z) with substitutions $\sigma_1 = \{x$/John, y/MotherOf(Lisa), q/John, z/Lisa, o/Sam, n/Sam$\}$ and $\sigma_2 = \{x$/Peter, y/Alice, q/Mark, z/Mary, o/Alice, n/Mark$\}$.

$\sigma_1 = \{\}; \qquad \sigma_2 = \{\};$
productlgg$(t \equiv t_0(t_1, \ldots, t_n), r \equiv r_0(r_1, \ldots, r_m))$:
1) if $t_0 \neq r_0$ return **variablize**$(t_0(t_1, \ldots, t_n), r_0(r_1, \ldots, r_m))$:
2) if $n = 0$, then return t;
else if $m = 0$, return r.
3) return $x = t_0(x_{1,1}, \ldots, x_{1,m}, \ldots, x_{n,1}, \ldots, x_{n,m})$,
where $x_{i,j} =$ **lgg**(t_i, r_j) with σ_1 and σ_2 as global substitutions.

variablize$(a, b) =$
if $\exists V$ s.t. $V\sigma_1 = a$ and $V\sigma_2 = b$,
then V
else $V =$ a new variable, $\sigma_1 = \sigma_1 \circ \{V/a\}$, and
$\sigma_2 = \sigma_2 \circ \{V/b\}$.

Figure 13. Cartesian-product *lgg* algorithm: **productlgg**.

```
main: initialization: h = {}
while EQ(h) gives counterexample x (which is positive):
    If ∃ a tree pattern p in h such that SQ(productlgg(p, x))
        then replace the first such p with reduce(productlgg(p, x))
        else h = h ⋃ x.   % add another tree to hypothesis
return h

reduce(t):
for each subtree s directly under the root of t
    prune s from t
    if not SQ(t), restore s
return t
```

Figure 14. Algorithm for CT forests.

Recall that the match semantics of Clausal trees exactly captures θ-subsumption. Unfortunately, θ-subsumption is strictly stronger than, but not equivalent to entailment. For some subclasses of predicate clauses such as single-predicate, non-recursive, Horn clauses, θ-subsumption and entailment are equivalent (Gottlob, 1987). Hence, a learning algorithm for Clausal trees also works for learning these predicate clauses from entailment. In fact, it turns out that the algorithm of Reddy and Tadepalli (1999) to learn single-predicate, non-recursive Horn programs can be directly adapted to learning CT forests. Figure 14 shows the learning algorithm. The main routine of the algorithm maintains the hypothesis as a set of Clausal trees patterns. It uses the **productlgg** algorithm to combine each new example with a Clausal tree. If the result passes the subset query, then it is used to replace the corresponding CT tree.

Each application of **productlgg** could produce a clause whose size is potentially the product of the sizes of the input clauses. Repeated applications of lgg with new examples could therefore produce an expression of size exponential in the number of examples unless some method is used to eliminate unnecessary predicates. For the same reason, the hypothesis Clausal trees cannot be simply replaced with the result of **productlgg**. Hence, it is pruned by the **reduce** routine, which removes each subtree and tests if the result is still a subset of the target. It is sufficient to go through the top level subtrees in sequence, and prune any subtree as allowed by the SQ oracle. SQ can be replaced by MQ by substituting variables with unique constant trees.

In our example, applying *reduce* to the lgg eliminates unnecessary predicates, giving: $\neg$ Father(x, y) $\vee \neg$Mother(y, z) $\vee$GrandFather(x, z). Note that the functions used in this example are all unary. If functions with more than one argument were used, the corresponding arguments according to their ordering would be matched and generalized by the *lgg* algorithm.

Multiple clause targets are equivalent to forests of Clausal trees. One complication in learning forests is to decide which sets of examples should be combined into a single tree. Following Reddy and Tadepalli (1999), this can be determined by finding the lgg of the new example with each of the hypothesis trees, and asking a subset query on the result. Compactness of the Clausal trees guarantees that if the subset query succeeds, then the resulting lgg clause is subsumed by a single target clause. Otherwise, the next hypothesis

tree is tried. Either the subset query succeeds on the result of lgg with some hypothesis tree, or a new hypothesis tree is initialized to the new example.

In our example, suppose that a new example is given ¬Father(Paul, Bob) ∨ ¬Father(Bob, Ted) ∨ GrandFather(Paul, Ted), which captures a second type of GrandFather relation. The lgg of this example with the current hypothesis clause ¬Father(x, y) ∨ ¬Mother(y, z) ∨ GrandFather(x, z) gives ¬Father(x, y) ∨ ¬Father(w, z) ∨ GrandFather(x, z). However, this new example is not entailed by the target, which requires that the variables y and w be the same. Hence the new example is made into a separate clause and stored as part of the hypothesis.

Theorem 23 (Reddy & Tadepalli, 1999)**.** *Clauses trees and forests are learnable from equivalence and membership queries.*

7. Discussion and future work

In this paper we considered the learnability of unordered tree patterns under various definitions of match semantics and related it to learning first order clauses. The main results of this paper are summarized in Table 1. We began with the negative result that showed that unordered trees under one-to-one onto semantics are not learnable from equivalence and subset queries regardless of the computational power of the learner. This in turn motivated the use of superset queries. We saw that unordered trees and forests are learnable under one-to-one onto semantics from equivalence and superset queries. We then showed that unordered trees and forests are learnable with equivalence and membership queries under one-to-one into semantics. Finally, we considered Clausal trees which are closely related to predicate clauses and have many-to-one into match semantics at the top level and have ordered one-to-one onto semantics at all the lower levels. These trees and forests are learnable from equivalence and membership queries by an algorithm similar to that of Reddy and Tadepalli (1999).

The theorems have shown that it is easy to learn with superset queries (and EQ) or without repeated variables but not with subset queries and repeated variables. While this

Table 1. UT/F learnability summary.

Concept class	Queries	Matching semantics	Learnable?	Justification
UTree	EQ,SQ	1-1 onto	No	Theorem 9
UForest	EQ,SQ	1-1 onto	No	Theorem 10
UTree	EQ,SupQ	1-1 onto	Yes	Theorem 14
UForest	EQ,SupQc	1-1 onto	Yes	Theorem 15
UTree	EQ,MQ	1-1 into	Yes	Theorem 19
UForest	EQ,MQ	1-1 into	Yes	Corollary 21
CT Forests	EQ,MQ	Many-1 into on 1-1 onto ordered	Yes	Theorem 23

might lead one to suspect that superset queries are inherently more powerful than subset queries, such is not the case. Superset queries are more powerful than the subset queries due to the conjunctive nature of our hypothesis space. Indeed, by the duality of Boolean algebra, the hypothesis space of the complement sets of those studied here are learnable by subset (and equivalence) queries and not learnable by superset (and equivalence) queries.

One goal of formal analysis is to pin down the sources of complexity, so that the problem domain can be carefully circumscribed to eliminate the difficult cases and find appropriate remedies. Our analysis points to two sources of complexity in the unordered tree learning: the *onto*-semantics, and the repeated variables. Only when both of these are present, learning is hard. With the *into*-semantics, it is possible to gain useful information through queries by generating instances with more branches than are in the target. If there are no repeated variables, the tree can be learned in divide-and-conquer fashion, since all the subtrees are independent (Theorem 12). If there can be up to k copies of each variable for some fixed k, then the μ-UF algorithm can be adapted to try all $O(2^k)$ ways to split a set of k variables into two partitions.

A possible criticism of the exact learning model is that it is too demanding. Our negative results on unordered onto trees and forests hold regardless of the computational power of the learner, but do not imply that the corresponding classes are not PAC learnable (Valiant, 1984), which only requires approximately correct learning of the target with a high probability. Our negative results also do not rule out the possible existence of a larger hypothesis space that is PAC learnable or even exactly learnable.These are open problems for the future.

All our positive results directly transfer to the PAC learning model by replacing the equivalence query with a random source of examples and using the standard simulation technique described by Angluin (1988). The other queries used in our results, such as subset and superset must be exactly implemented for the theoretical results to hold. While our results are theoretical at the moment, we were motivated by practical applications in symbolic algebra and information extraction. The practicality of an algorithm depends entirely on the practicality of implementing the queries used by that algorithm.

Our results underline the important asymmetry between subset and superset queries for implementation. Even though theoretical results require exact implementations, in many domains it might suffice to approximately implement these queries by sampling, i.e., by membership queries. While the hypothesis argument is available to the teacher in a declarative form and can be used to efficiently generate instances in it, the target may only be available as a black box procedure.[2] This is similar to the situation where a scientist may have an explicit hypothesis about a phenomenon, while the true law itself is inaccessible, and can only be tested in individual cases by conducting experiments. Implementing subset query in this case amounts to generating instances that satisfy the hypotheses and testing them by conducting experiments. It is sometimes possible to do a faithful simulation of a subset query with just one membership query (an experiment), as it is in our tree domains. When a tree domain has an infinite constant alphabet or infinite branching factor, the membership queries are in fact as powerful as subset queries. This is so because for any tree pattern h, it is easy to generate an instance x by instantiating its variables uniquely so that h is a subset of a target if and only if x is an instance of it. Hence subset queries are as easy to implement as the membership queries in our case. The same cannot be said of superset

queries, however. To simulate a superset query, one needs to ensure that all instances in the target are also in the hypothesis. The target is only available as a black box and cannot be used to efficiently generate satisfying instances. Generating instances from a superset of the target (e.g., the entire instance space) and filtering by the target before testing them on the hypothesis would be be too inefficient in any nontrivial domain.

The algorithm for non-recursive Horn clauses (or Clausal trees in Section 6) has been implemented and used to learn goal decomposition rules in planning (Reddy & Tadepalli, 1999). Each decomposition rule specifies how a goal may be decomposed into subgoals when certain conditions are satisfied. A subset query corresponds to testing whether a given rule is valid, i.e., whether it always yields a set of subgoals, when solved, results in achieving the goal. This query was implemented by synthesizing problems that satisfy its precondition and trying to apply the decomposition rule on it. If the rule succeeds on a large enough sample, the subset query is answered 'yes'.[3] The program was able to learn about 12–24 rules with 60–70 examples in two STRIPS-like planning domains with approximately 90% accuracy on an independent test set (Reddy & Tadepalli, 1999). Improving the query-efficiency of the algorithms and making them noise-tolerant would be important steps towards turning our theory into practical implementations.

Tree pattern languages bear some similarities to string pattern languages (Goldman & Kwek, 1999). String patterns are strings of variables and constant symbols. Each variable in the pattern can be substituted with a non-empty string of constant symbols to produce instances that match that pattern (Angluin, 1980). It has been shown that string patterns can be learned with a polynomial-number of disjointness queries in polynomial-time (Lange & Wiehagen, 1991). The disjointness query asks if $A \cap T = \{\}$ (where T is the target) and gives a counterexample if the answer is 'no'. Disjointness queries appear very powerful since the set A need not be in the hypothesis space.

However, string pattern languages are too hard to learn in the representation-independent/predictability model, even with arbitrary queries. This is because string patterns do not have a polynomial size representation with a polynomial-time membership algorithm (Schapire, 1990). It is not known if such a strong negative result also holds for unordered tree patterns.

Learnability of unordered tree patterns with other variations of matching semantics—such as many-to-one into and many-to-one onto at all levels of trees would be interesting. Another extension is tree patterns with different matching semantics at each node. Such trees are quite natural in domains such as symbolic mathematics, where some operators are commutative and associative and others are not.

Appendix

Lemma 1. *For any tree pattern p, $(p\sigma_1)\sigma_2 = p(\sigma_1 \circ \sigma_2)$.*

Proof: If $y_i \notin \{x_1, \ldots, x_n\}$, then any y_i in p is substituted with t_i on both sides of the above equation. If y_i is one of the x_j's, then $\sigma_1 \circ \sigma_2$ correctly eliminates y_i/t_i. Otherwise if y_i is part of s_j, $s_j\sigma_2$ yields the same as first applying σ_1, then σ_2 to x_j. □

Lemma 2. *The composition of two Ψ-consistent mappings is a Ψ-consistent mapping.*

Proof: Let μ_1 and μ_2 be two mappings of the same type Ψ (one-to-one onto, one-to-one into or many-to-one into). Define the composition of μ_1 and μ_2, $\mu_1 \circ \mu_2$, in the natural way so that $x(\mu_1 \circ \mu_2) = (x\mu_1)\mu_2$ (written as $x\mu_1\mu_2$) for any x. It is easy to see that $\mu_1 \circ \mu_2$ is of the type Ψ as well. □

Lemma 3. *For every pattern P, substitution σ, and Ψ-consistent mapping μ there exist a Ψ-consistent mapping μ' such that $P\mu\sigma = P\sigma\mu'$.*

Proof: Let s be a subtree of P. If μ maps s in P to t, let μ' map $s\sigma$ to $t\sigma$. Hence $s\sigma\mu' = t\sigma = s\mu\sigma$. Since this holds consistently for any subtree s of P, μ' is a Ψ-consistent mapping, and $P\sigma\mu' = P\mu\sigma$. □

Lemma 4. *If $P \succeq Q$ and $Q \succeq R$, then $P \succeq R$.*

Proof: Given $P\sigma$ maps to Q for substitution σ and semantics Ψ, there is a mapping μ so that $P\sigma\mu = Q$. Similarly, $Q \succeq R$ implies there exists σ' and μ' so $Q\sigma'\mu' = R$. Therefore $P\sigma\mu\sigma'\mu' = R$, or $P\sigma\sigma'\mu''\mu' = R$ by Lemma 3. Further, $P(\sigma \circ \sigma')(\mu'' \circ \mu') = R$ by Lemmas 2 and 1. Finally, $P \succeq R$ by Definitions 2 and 3. □

Lemma 6. *The classes of (Ordered or) Unordered tree patterns (UT) with one-to-one onto, one-to-one into, and many-to-one into semantics are compact.*

Proof: Let $\mathcal{Z}, \mathcal{V}_1 \ldots \mathcal{V}_n$ be tree patterns such that $\bigcup_{i=1}^{n} L(\mathcal{V}_i) \supseteq L(\mathcal{Z})$. Let $I = \mathcal{Z}\sigma$ where σ replaces each variable in $\mathcal{Z}$ by a constant that does not appear in any of the $\mathcal{V}_i$ or $\mathcal{Z}$. Then the inverse substitution is well defined, giving $I\sigma^{-1} = \mathcal{Z}$. Further, $I \in L(\mathcal{Z})$, so $I \in \bigcup_i L(\mathcal{V}_i)$ and since I is just one tree instance, $I \in L(\mathcal{V}_i)$ for some i. Therefore, for some substitution σ' and mapping μ', $\mathcal{V}_i\sigma'\mu' = I$. Combining with the above gives $\mathcal{V}_i\sigma'\mu'\sigma^{-1} = \mathcal{Z}$. Using Lemma 3, $\mathcal{V}_i\sigma'\sigma^{-1}\mu = \mathcal{Z}$ for another mapping μ. Substitution composition via Lemma 1 gives $\mathcal{V}_i(\sigma' \circ \sigma^{-1})\mu = \mathcal{Z}$ or $\mathcal{V}_i \succeq \mathcal{Z}$ for some i. By Theorem 5, $L(\mathcal{V}_i) \supseteq L(\mathcal{Z})$ for some i. □

Lemma 7. *The problem of deciding whether an unordered tree pattern with repeated variables matches a tree instance with either onto or into semantics is NP-Complete (by reduction from CLIQUE).*

Proof: (CLIQUE $\leq$ match): It is easy to see that the matching problem is in NP. We now reduce the problem of deciding whether a graph has a clique (complete subgraph) of size k to the matching problem of unordered tree patterns. A representation of graphs in terms of unordered trees is chosen, where the graph is represented by a 2-level tree instance and a clique is represented by a 2-level tree pattern, both of which have roots labeled with a special symbol O. The mapping is done as follows:

1. For each edge (P, Q) in the graph, there is a first-level subtree $O(P, Q)$ in the tree instance, where P and Q are constants.

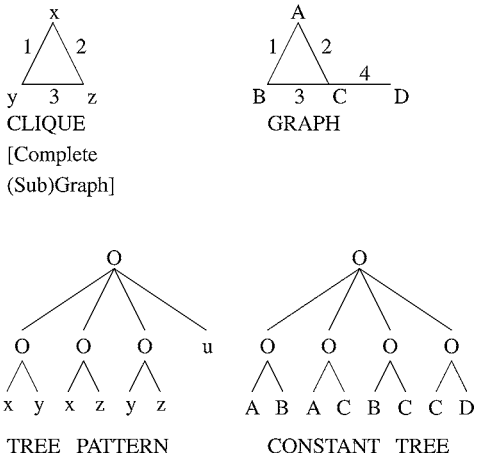

Figure 15. CLIQUE as unordered tree.

2. For each edge (x, y) in a clique of size k, there is a first-level subtree $O(x, y)$ in the tree pattern, where x and y are variables.
3. For onto semantics, the pattern tree has enough extra children in the form of single-variable subtrees so its root has the same number of children as the root of the constant tree.

See figure 15 for an example of a 3-clique and a 4-vertex graph. The problem of testing if a graph has a clique (complete subgraph of a specified number of vertices) reduces to the matching problem for unordered trees by the above mapping. Unordered tree matching is therefore NP-Complete (Garey & Johnson, 1979). □

Note that both repeated variables and unordered matching are necessary to make the tree-matching problem NP-Complete. The matching problem would have been related to "First Order Subsumption" in Garey and Johnson (1979) except that UT trees are unordered whereas the arguments in the expressions or functions in the latter are apparently ordered.

Lemma 8. *There are* $n!$ *distinct 3-matrices which are consistent with the pair of* 2*-matrices*:

$$\begin{array}{cccccc} 0 & n & 2n & \ldots & (n-1)n \\ c & c-n & c-2n & \ldots & c-(n-1)n \end{array}$$

and

$$\begin{array}{cccccc} c & c-1 & c-2 & \ldots & c-(n-1) \\ 0 & 1 & 2 & \ldots & n-1 \end{array}$$

where $c \geq n^2 - 1$.

Proof: We form the $n!$ 3-matrices by taking the $n!$ permutations of the n columns of the second 2-matrix and use the ith column of the first 2-matrix and the ith column of the permuted second matrix to compute the ith column of a 3-matrix. So, for example, from the column $\binom{in}{c-in}$ and $\binom{c-j}{j}$ we form $\binom{in}{j}_{c-j-in}$. No matrix entry is allowed to be less than 0. This condition is satisfied if $c - j - in \geq 0$ in the worst case of $i = j = n - 1$, provided $c \geq n^2 - 1$.

We claim that the 3-matrices so formed are distinct. First, notice that permuting rows cannot change any of these 3-matrices into another of these 3-matrices because the top row will consist of multiples of n, the bottom row will consist of the numbers 0 through $n - 1$, and the middle row will contain a set of numbers which satisfies neither of these criteria. Second, no column permutations can convert one of these 3-matrices to another of these 3-matrices because by our construction technique, the top row of each 3-matrix will consist of the multiples of n in order from $0 \cdot n$ to $(n - 1) \cdot n$, and the bottom row will consist of a permutation of the numbers 0 through $n - 1$. To convert one of these 3-matrices to another, the columns must stay as they are in order to make the top rows agree, but the columns must be permuted to make the bottom rows agree. This contradiction shows that all of these 3-matrices are distinct. □

Lemma 13. *Let $h = h_0(h_1, \ldots, h_n)$ be a hypothesis tree pattern with no repeated variables. Let $t = t_0(t_1, \ldots, t_n)$ be a target tree pattern. Then $L(h) \supseteq L(t)$ iff $h_0 = t_0$ and there is a one-to-one onto mapping μ^* from $\{h_1, \ldots, h_n\}$ to $\{t_1, \ldots, t_n\}$, such that $L(h_i) \supseteq L(h_i\mu^*)$.*

Proof: Only if: Since $L(h) \supseteq L(t)$, by Theorem 5, $h \succeq t$. By Definitions 3 and 2 there is a substitution σ and mapping μ such that $h\sigma\mu = t$. Define the mapping μ^* to give the same result on the subtrees as the combined effect of σ and μ. The mapping μ is one-to-one onto, so μ^* is also. Therefore, for each h_i, there is a t_j so $h_i \succeq t_j$. Finally, $L(h_i) \supseteq L(h_i\mu^*)$ by Theorem 5.

if: Suppose $x \in L(t)$. The root label of x must be $t_0 = h_0$, since otherwise t cannot include x. Moreover, each subtree of x must be in one of the subtrees in t in such a way that all subtrees are covered. Since μ^* is a one-to-one onto map, the ith subtree of x, $x_i \in L(h_i\mu^*)$ for $1 \leq i \leq n$. Since $L(h_i) \supseteq L(h_i\mu^*)$, $x_i \in L(h_i)$ as well. This means $h_i \succeq x_i$, so there exists a substitution σ_i such that there is a on-to-one onto map from $h_i\sigma_i$ to x_i. Consider the substitution $\sigma = \bigcup_i \sigma_i$. Since no variables are shared between different h_i's, σ is a valid substitution, and $h_i\sigma_i$ is the same as $h_i\sigma$. From this it follows that $h \succeq x$, and so $x \in L(h)$. Hence $L(h) \supseteq L(t)$. □

Theorem 14. *An unordered tree with repeated variables is learnable using SupQ and one-to-one onto semantics from a single arbitrary positive example.*

Proof: Note that the hypothesis in the **grow-tree** routine does not have any repeated variables (see figure 5). At any time if the new hypothesis h is a superset of the target as determined by SupQ, then there is a one-to-one onto map μ such that each subtree h_i is a superset of a corresponding target tree by Lemma 13, and so the hypothesis can be refined further. If on the other hand, h is not a superset of the target, there is no such μ by Lemma 13,

and the hypothesis is over-refined. Since the refinement of h occurs on each subtree by one more level at each step, the **grow-tree** routine retracts the last refinement, and tries refining other subtrees. If no subtree can be refined successfully, the hypothesis tree must be identical to the target tree except for grouping of variables by applying appropriate substitution.

After this first phase the hypothesis tree has all leaves labeled with constants or distinct variables. **fuse-vars** tries making each pair of variables identical, tests with a superset query and keeps the change only if it is accepted by SupQ. If the change is accepted by SupQ, that means that $L(h) \supseteq L(t)$, which implies that $h \succeq t$ by Theorem 5. Since h and t share the identical structure, this means that there exists a substitution σ of variables to variables that makes h map to t. If the change is not accepted, $h \not\succeq t$ by Theorem 5, and hence there is no such substitution. Since each change is confined to identifying two variables, if it is not accepted, that change can be retracted with no further repercussions.

Only one positive example is used. The bound for the number of superset queries is $n + l + v^2$, where the number of nodes n and the number of leaves l have to be potentially created and tested to see if they can be constants. All pairs of the v variables have to be tested to see if they can be identical. These measures all refer to the target not the example. The time complexity is bounded by the query complexity times the target size, or $n(n + l + v^2)$. □

Lemma 16. *Routine* **prune** (*figure 8*) *will remove all nodes in the example tree not corresponding to nodes in the target tree.*

Proof: Given a target tree $\mathcal{T}$ which matches a hypothesis tree p, then any leaf l in p must satisfy one of the following conditions. There could be a leaf v in $\mathcal{T}$ which corresponds only to l via the mapping, then l cannot be eliminated or $\mathcal{T}$ would no longer match p. So **prune** will not remove l. If variable v in $\mathcal{T}$ could match a subtree s in p of which l is a leaf, then removing l would still allow $\mathcal{T}$ to match p, so **prune** removes l. Finally, l is a leaf of a subtree s of p matched by a subtree r with depth at least one (at least root and children) of $\mathcal{T}$ in a way that none of the subtrees of r maps to a subtree of s containing l—as allowed by the *into* semantics. Again, **prune** will remove l since the latter is not needed for $\mathcal{T} \succeq p$. Therefore, **prune** removes all the nodes which are not needed to correspond to and be matched by some target tree. □

Lemma 17. *Given a pruned hypothesis tree t, which represents a subset of the target, an "old" leaf o in t and a "new" variable leaf n to replace o, routine* **partition** *will generalize the set of n's if that set is less general than some target tree.*

Proof: By Lemma 6, UT with one-to-one into semantics is compact. Given that t is a pruned tree which represents a subset of the target, there is some tree $\mathcal{T}$ in the target, so $\mathcal{T} \supseteq t$. It follows that there is a correspondence between $\mathcal{T}$ and t represented by the mapping μ of Definition 2, and this correspondence is one-to-one onto (otherwise, the extra subtrees permitted by the into mapping would have been removed by routine **prune**).

If the set of n's in t is less general than the set of corresponding leaves in $\mathcal{T}$, then t will be generalized by the following argument. Designate the o's as o_1 through o_k. Target tree $\mathcal{T}$ can only be more general than the o's in t by having variable(s) in the former correspond

to one or more constant o's in the latter, or by having at least two distinctly labeled leaves in $\mathcal{T}$ correspond to two o's in t.

This algorithm adds n's as extra children (copies of a new variable) to those subtrees of t having o's. One-to-one into semantics permits the original t to match all of the instances matched by the version of t augmented with these n's. So the result will represent a subset of t and therefore of $\mathcal{T}$. Next o's or n's are eliminated one at a time while keeping only those changes which keep this new t a subset of $\mathcal{T}$. This approach permits the set of identical constant or variable o's to be split or partitioned into two sets without having to separately test an exponential number of such splits. Since only changes accepted by SQ are kept, the condition that t represents a subset of the target is maintained.

As one of the cases, suppose $\mathcal{T}$ has variable v in all positions corresponding to the o's in t. Then **partition** will add a number of n's equal to the number of o's. Then this routine will try to remove the o's first and will succeed on the first try. An attempt to then remove the n's first will fail because the target has all the same variable v corresponding to the o's and will not be more general than t with two different variables in those positions. The result will be t with o's replaced with n's which is as general as the target for those positions.

Further split the remaining cases into two possibilities; first assume some of the leaves in $\mathcal{T}$ corresponding to the o's are constants (which must therefore be identical to o) as well as one or more distinct variables—refer to these variables collectively as v. Then **partition** will successfully eliminate n's corresponding to the constants in $\mathcal{T}$ and also eliminate the o's corresponding to v's. Eliminating more n's than the constants in the target or more o's than the variables in v will not pass SQ. Hypothesis tree t will therefore have been successfully partitioned and generalized by changing some of its constants into the new variable n.

If the corresponding leaves in $\mathcal{T}$ have two or more variables, but no constants, arbitrarily split these variables into two sets of variables and designate them as x and y. Then **partition** will, without loss of generality, eliminate o's in positions which can correspond to y's, leaving the n's for those positions. When a position corresponding to an x is encountered, **partition** can eliminate the n in that position. The result will be n's in positions in t corresponding to y's in $\mathcal{T}$ and o's in positions corresponding to x's. Thus, whenever the target has two or more variables corresponding to the o's, **partition** can separate these leaves into two distinct variables, o and n. □

Lemma 18. *Routine* **variablize** *will learn one of the target trees from a pruned example.*

Proof: Given the hypothesis tree p which is already pruned by routine **prune**, since p was accepted by SQ, so $\mathrm{L}(\mathcal{T}) \supseteq \mathrm{L}(p)$ and $\mathcal{T} \succeq p$ by Theorem 5. Hence, there exists a substitution σ and a mapping μ so $\mathcal{T}\sigma\mu = p$. Since $\mathcal{T}$ matches p, and p has been pruned, these trees must be identical in all of the corresponding internal nodes; so only the leaves can differ.

Further, p has the same number of leaves as $\mathcal{T}$ which correspond by μ, so the mapping μ is one-to-one onto for this case. One of the following cases must apply: hypothesis tree p is equivalent to $\mathcal{T}$, in which case **variablize** is done, or $\mathcal{T}$ is strictly more general than t. In the latter case, the difference must be in some leaf of t–or a set of identical leaves. Since **variablize** calls **partition** for all such sets of leaves, t will be generalized by Lemma 17. This process continues until t is eventually equivalent to $\mathcal{T}$. □

Theorem 19. *UT with one-to-one into semantics is learnable with equivalence and subset queries.*

Proof: Given a tree instance $t \in L(\mathcal{T})$ where $\mathcal{T}$ is a target tree, by Definitions 3 and 2 there exists a substitution σ and mapping μ so $\mathcal{T}\sigma\mu = t$. If μ is not one-to-one onto, by Lemma 16, routine **prune** can remove leaves until the mapping satisfies that criteria. Then, by Lemma 17, routine **variablize** will generalize t to be equivalent to $\mathcal{T}$. So this algorithm will learn any UT target with into semantics.

Define a metric for (hypothesis) tree complexity as the total number of tree nodes n, minus the number of distinct variables v, plus the number of edges e, giving $n - v + e$. Let this value for a given example tree be r. Each possible generalization must convert a constant to a variable, partition a variable or eliminate an edge with a distinct variable and thereby decrease this metric. The total number of *generalizations* is therefore bounded by the value r of this metric for the example tree given to the algorithm. The total number of queries is bounded by the number of ways to generalize 1-level trees times the number of nodes to be generalized. Each factor is bounded by r, so the overall bound is $O(r^2) = O(n^2)$. Time complexity is bounded by the example size times the number of queries or $O(n^3)$. □

Corollary 20. *UF with one-to-one into semantics is learnable with equivalence and subset queries.*

Proof: By Lemma 6 this class is compact. Therefore in the algorithm in figure 8 each hypothesis tree that cannot be further generalized will cover a single target tree rather than merely covering parts of multiple target trees. Each target tree is learned from a single example. EQ then supplies another example not already in the hypothesis. This example is then generalized to another hypothesis tree pattern—until the entire target is covered.

The bounds are the sum of the bounds for the individual target trees. □

Corollary 21. *UF with one-to-one into semantics is learnable with EQ and MQ.*

Proof: We can simulate SQ with MQ by using a unique constant in place of each distinct variable. Either the simulation is faithful or a constant conflicts with one in a target tree. In the latter case we get a negative counterexample from EQ and we can restart the algorithm with constants not already tried for this purpose. □

Acknowledgments

This research was partially supported by the NSF under grant number IRI-9520243. We thank Dana Angluin, Lisa Hellerstein, Roni Khardon, Stephen Kwek, David Page, Vijay Raghavan, and Chandra Reddy for interesting discussions on the topic of this paper. We thank the reviewers for many excellent suggestions on the paper.

Notes

1. It could be argued that extra variables in a subtree which do not appear elsewhere in a pattern are useless in many-to-one into semantics–or that those variables do no harm.
2. If the target is also declaratively available to the teacher, he may simply tell it to the learner, thus making the learning problem superfluous.
3. The limited number of constants in planning domains requires trying it on more than one problem.

References

Amoth, T. R., Cull, P., & Tadepalli, P. (1998). Exact learning of tree patterns from queries and counterexamples. In *Proceedings of the Eleventh Annual Conference on Computational Learning Theory*, (pp. 175–186). New York, NY: ACM.

Amoth, T. R., Cull, P., & Tadepalli. P. (1999). Exact learning of unordered tree patterns from queries. In *Proceedings of the Twelth Annual Conference on Computational Learning Theory* (pp. 323–332). New York, NY: ACM.

Angluin, D. (1980). Finding patterns common to a set of strings. *J. of Comp. and Syst. Sciences*, *21*, 46–62.

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

Arimura, H., Ishizaka. H., & Shinohara, T. (1995). Learning unions of tree patterns using queries'. In *Proceedings of the 6th ALT (Algorithmic Learning Theory)* (pp. 66–79). Springer Verlag. Lecture Notes in Artificial Intelligence 997.

De Raedt, L. (1997). Logical settings for concept learning. *Artificial Intelligence, 95:1*, 187–201.

Garey, M., & Johnson, D. (1979). *Computers and Intractability: A Guide to Theory of NP-completeness*. New York, NY: W.H. Freeman.

Goldman, S. A., & Kwek, S. S.(1999). On learning unions of pattern languages and tree patterns. In *Proceedings of the 10th ALT (Algorithmic Learning Theory)* (pp. 347–363). Springer Verlag. Lecture Notes in Artificial Intelligence 1720.

Gottlob, G. (1987). Subsumption and implication. *Information Processing Letters, 24:2*, 109–111.

Hellerstein, L., Pillaipakkamnatt, K., Raghavan, V., & Wilkins, D. (1996). How many queries are needed to learn?. *Journal of the Association for Computing Machinery, 43:4–6*, 840–862.

Ko, K.-I., Marron, A., & Tzeng, W.-G. (1990). Learning string patterns and tree patterns from examples. In B. W. Porter, & R. J. Mooney (Eds.), *Proceedings of the Seventh International Conference on Machine Learning* (pp. 384–391). San Mateo, CA: Morgan Kaufmann.

Lange, S., & Wiehagen, R. (1991). Polynomial-time inference of arbitrary pattern languages. *New Generation Computing, 8*, 361–370.

Page, C. D. (1993). Anti-unification in constraint logics: Foundations and applications to learnability in first-order logic, to speed-up learning, and to deduction. Ph.D. Thesis, University of Illinois, Urbana, IL.

Page, C. D., & Frisch, A. M. (1992). Generalization and learnability: A study of constrained atoms. In S. H. Muggleton (Ed.), *Inductive Logic Programming*. London: Academic Press.

Plotkin, G. (1970). A Note on inductive generalization. In B. Meltzer, & D. Michie (Eds.), *Machine intelligence* (Vol. 5). New York, NY: Elsevier North-Holland.

Reddy, C., & Tadepalli, P. (1999). Learning horn definitions: Theory and an application to planning. *New Generation Computing, 17:1*, 77–98.

Schapire, R. E. (1990). Pattern languages are not learnable. In *Conference on Computational Learning Theory* (pp. 122–129). San Mateo, CA: Morgan Kaufmann.

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

Received May 24, 2000
Revised August 29, 2000
Accepted December 20, 2000
Final Manuscript December 20, 2000