Exact Learning of Unordered Tree Patterns From Queries

Thomas R. Amoth Paul Cull Prasad Tadepalli
amotht@cs.orst.edu, pc@cs.orst.edu, tadepall@cs.orst.edu
Department of Computer Science
Oregon State University
Corvallis, OR 97331
May 14, 1999

Abstract

We consider learning tree patterns from queries extending our preceding work [Amoth, Cull, & Tadepalli, 1998]. The instances in this paper are unordered trees with nodes labeled by constant identifiers. The concepts are tree patterns and unions of tree patterns (unordered forests) with leaves labeled with constants or variables. A tree pattern matches any tree with its variables replaced with constant subtrees. A negative result for learning with equivalence and membership/subset queries is shown for unordered trees where a successful match requires the number of children in the pattern and instance to be the same. Unordered trees and forests are shown to be learnable with an alternative matching semantics that allows an instance to have extra children at each node.

1 INTRODUCTION

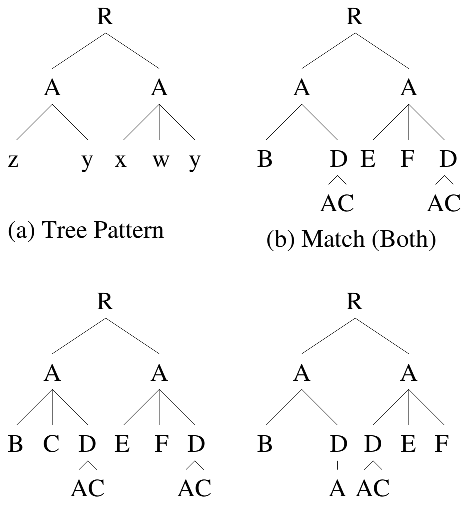
Many applications in mathematics and language processing represent data more naturally as trees (or as unions of these) than as vectors of features. Tree patterns also provide more information than simple string patterns. Some mathematical operations require the parts of the structures to be in a particular order, so the trees are *ordered*. Other applications such as mathematical functions which are commutative (add, multiply) allow their arguments in any order, making their trees *unordered*.

We use the exact learning framework of Angluin with a variety of queries [Angluin, 1988]. In this framework, the teacher can pick any target concept in the concept class. The learner is allowed to ask queries about the target. The number of queries asked by the learner must be bounded by a polynomial function in the size of the target concept. The learner has to exactly identify the target concept, i.e., find a hypothesis which matches exactly the set of instances denoted by the target, in time polynomial in the size of the target and the size of the input to the learner (in the form of various responses to its queries). Unlike the PAC-predictability model, we do not require that the hypothesis output by the learner has a polynomial-time membership algorithm. In all of our algorithms, the queried hypothesis is in the target class.

Query oracles as introduced by Angluin [Angluin, 1988] are used. An *Equivalence Query* (EQ) is given a hypothesis as an argument and returns *true* if that hypothesis is covers the same set of instances as the target but otherwise returns *false* and returns a counterexample. A *Membership Query* (MQ) is given a single instance and returns *true* iff the instance is a member of the target. A *Subset Query* (SQ) is given a possible hypothesis and returns *true* iff that hypothesis is a subset of the target.

Ordered tree patterns with repeated variables have been shown to be learnable with equivalence (EQ) and membership (MQ) queries and either a bound on the number of trees [Arimura, Ishizaka, & Shinohara, 1995] or an infinite alphabet [Amoth, Cull, & Tadepalli, 1998]. In the previous work we showed that unordered trees without repeated variables are learnable from equivalence and membership queries and that superset queries and equivalence queries are sufficient to learn unordered trees with repeated variables. Here we show that unordered trees with repeated variables are not learnable with equivalence and subset queries (SQ).

Since learning unordered trees is hard without queries more powerful than subset queries, an alternative model is studied. The alternative "into" semantics allows extra children in the instance at each matched node as long as each subtree of the pattern matches a distinct child subtree. With this semantics, unordered forests (UF) are shown to be learnable from EQ and SQ. Since SQ can be simulated by MQ in this class (when EQ is available), they are also learnable from EQ and MQ.



(a) Tree Pattern (b) Match (Both) (c) Into Matching (d) Not Match

Figure 1: Match Semantics Example

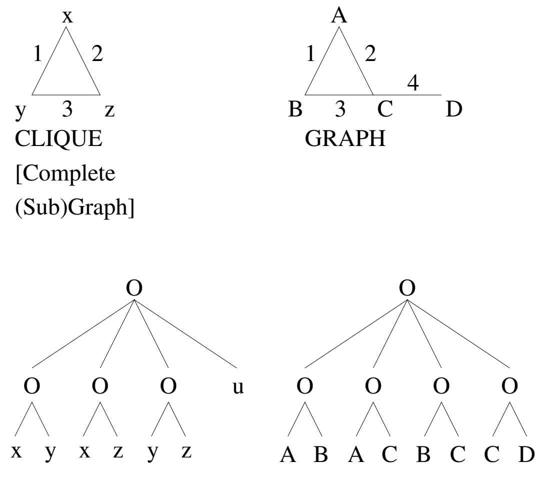


Figure 2:

2 DEFINITIONS

The *tree learning problem* is to learn a tree pattern from example tree instances. A *tree instance* has a label (from the *infinite* constant alphabet) at each node. A *tree pattern* has constants at its internal nodes, but its leaves may be *variables* which will match any constant subtree.

Matching: The trees in Figure 1 could also be represented in a parent(child . . . child) style notation as $R(\ A(zy)\ A(xwy)$), etc. A tree pattern is a tree with the leaves labeled with constants or variables. A constant matches the same constant but a variable matches any constant subtree. Two *unordered* tree *instances* $f(r_1 \ldots r_k)$ and $g(s_1 \ldots s_k)$, match according to *onto* semantics if $f = g$ and there is a one-to-one *onto* mapping or permutation π of $1 \ldots k$ such that the child subtree r_i matches $s_{\pi(i)}$, $1 \leq i \leq k$. A constant node with no children matches only itself.

The difficulty of learning with onto semantics suggests exploration of an alternative semantics which only require the mapping of the children of each node from pattern to instance to be one-to-one *into*. The instance is allowed to have extra, unmatched children; this semantics is closer but not equivalent to that used for Horn-clause learning. Using the above notation, tree pattern $f(r_1 \ldots r_n)$ would match tree instance $g(s_1\ \ldots\ s_m)$ if $f = g$ and $0 \leq n \leq m$, and there exists a one-to-one *into* mapping p such that for all $i \leq n\ \ r_i$ matches $s_{p(i)}$.

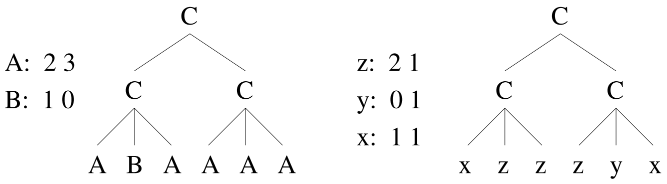
For each variable v in a tree pattern p, a *substitution* replaces all copies of v with the same constant subtree. A tree pattern *matches* a tree instance if there exists a substitution which makes the tree pattern match the instance (with the corresponding semantics). The tree pattern (a) in Figure 1 matches instance (b) with both semantics, and instance (c) only with "into", but does not match (d) with either semantics.

The decision problem for matching is hard (and independent of the learning problem):

Theorem 1 *The problem of deciding whether a tree pattern matches a tree instance with either* **onto** *or* **into** *semantics is NP-Complete.*

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 with the following properties:

1. the entire graph is represented by a constant tree
2. the clique is represented by a tree pattern
3. both the tree example and the tree pattern have two levels
4. the constant tree has a first-level subtree for each edge of the graph
5. the pattern tree has a first-level subtree for each edge in the clique
6. each such subtree has two children–the two vertices joined by the edge
7. each variable name in the tree pattern corresponds to a vertex in the clique
8. each constant label name in the example tree corresponds to a vertex in the graph
9. 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.



(a) Instance (b) Pattern

Figure 3: 2-Level Tree Instance, Pattern, Corresponding Matrix Notations

Under these conditions, the problem of testing if a graph has a CLIQUE (complete subgraph of a specified number of vertices) can be recast as matching an unordered tree instance by an unordered tree pattern (See Figure 2). UT matching is therefore NP-Complete. □

Simpler trees can be matched in poly time as follows: OT (ordered trees with repeated variables) can be matched while caching the variable-value correspondences. μ-UT (unordered trees without repeated variables) can be matched using recursion on the tree depth and a 2-D matching algorithm at each tree level in each node to determine if all the children at that node in a tree instance can match all the children of the node in the tree pattern. (The latter operation is done in such a way that each constant subtree and pattern subtree are matched at most once.) Forest matching takes only polynomially longer than tree matching. Matching of two subtrees of the *same* tree pattern (or two constant trees) is polynomial because the decision of whether a leaf in one tree matches a leaf in the other tree is local. (The problem of determining the correspondence between variables and constants makes the pattern to instance matching problem NP-Complete.)

A tree pattern *represents* the set of tree instances that match it. Hence we say that these instances are *in* the pattern. We consider learning tree patterns as well as finite unions of tree patterns, which we call forests. Two major learning problems based on the following two classes are considered: unordered trees (UT), and unordered forests (UF).

We consider three different queries introduced by Angluin [Angluin, 1988]. An *Equivalence Query* (EQ) oracle is given a hypothesis (tree or forest pattern), and returns *true* if and only if the set of instances in the hypothesis are identical to those in the target. It also returns a counterexample if the answer is *false*. A *Membership Query* (MQ) oracle is given a single instance (a tree instance), and returns *true* iff that instance is in the target and otherwise *false*. A *Subset Query* (SQ) oracle is given a hypothesis (a tree or forest pattern), and returns true iff the hypothesis represents a subset of the target.

3 NEGATIVE RESULT FOR "ONTO" UT WITH EQ AND SQ QUERIES

UF or UT is not learnable from EQ alone since the match problem is NP-hard. (Any hard-to-compute class is hard to PAC-predict by Theorem 7 of [Schapire, 1990].) In this section we now prove that the concept class of unordered trees with repeated variables (UT) is not learnable with both EQ and SQ (equivalence and subset queries) Using a combinatorial argument. We use the proper exact learning framework where the hypothesis given to EQ is required to be in the target class.

3.1 EXAMPLE ILLUSTRATING THE PROOF FOR 3 SUBTREES

In this section, we introduce a subclass of UT called T2 that is hard to learn. T2 consists of 2-level tree patterns with s subtrees, each having c children, and at most 3 variables. We introduce a special notation to make the arguments concise.

The notation represents just the counts of each variable in each subtree since the children of each such subtree can be permuted. A matrix of numbers will be used with each column corresponding to one subtree and each row corresponding to one variable in the target or one constant in an instance. Figure 3 shows a sample tree instance and the corresponding matrix to its left (with the rows labeled to help show this correspondence). The left column containing 2 and 1 are the number of leaves labeled by A's and B's in the left subtree. Permuting the columns of a matrix permutes the subtrees of a tree and produces an isomorphic tree. Permuting the rows reassigns variable names or constants but otherwise creates an identical tree. Changing a pair of numbers that were originally in the same column/row to no longer be in the same column/row produce a non-equivalent tree.

Tree matching requires each instance row to be a sum of rows of the pattern matrix with the columns permuted uniformly for all rows: in Figure 3, y matches B and x and z both match A. With this assignment, swapping the two subtrees gives the instance in (a).

Consider the following example: let one possible target have c=8 children per subtree and s = 3 subtrees/columns, and v = 3 target variables (rows) (upper right 3×3 matrix of Figure 4). Exact learning requires an algorithm to work even in the worst case and has the effect of making EQ behave in an adversarial manner. Suppose EQ then gives a learner an example with all the same constant (8 8 8 in matrix notation). Then the learner could change the constant to a variable and give the tree to EQ. EQ would then have to give an instance with (at least) 2 variables (form the example by adding the bottom two rows of the target). The learner could repeat the process and EQ could return another two-variable instance (by adding the top two rows of the target as shown as the top two 2×3 matrices of the left column in Figure 4). If the learner repeats the process with the last instance, EQ could return the first two-constant instance. The process could indefinitely cycle between these two instances.

$$\begin{array}{ccc} 0 & 3 & 6 \\ 8 & 5 & 2 \end{array}$$

$$\begin{array}{ccc} 8 & 7 & 6 \\ 0 & 1 & 2 \end{array} \qquad \begin{array}{ccc} 0 & 3 & 6 \\ 8 & 4 & 0 \\ 0 & 1 & 2 \end{array}$$

$$\begin{array}{ccc} 7 & 6 & 8 \\ 1 & 2 & 0 \end{array} \qquad \begin{array}{ccc} 0 & 3 & 6 \\ 7 & 3 & 2 \\ 1 & 2 & 0 \end{array}$$

$$\begin{array}{ccc} 6 & 8 & 7 \\ 2 & 0 & 1 \end{array} \qquad \begin{array}{ccc} 0 & 3 & 6 \\ 6 & 5 & 1 \\ 2 & 0 & 1 \end{array}$$

$$\begin{array}{ccc} 7 & 8 & 6 \\ 1 & 0 & 2 \end{array} \qquad \begin{array}{ccc} 0 & 3 & 6 \\ 7 & 5 & 0 \\ 1 & 0 & 2 \end{array}$$

$$\begin{array}{ccc} 6 & 7 & 8 \\ 2 & 1 & 0 \end{array} \qquad \begin{array}{ccc} 0 & 3 & 6 \\ 6 & 4 & 2 \\ 2 & 1 & 0 \end{array}$$

$$\begin{array}{ccc} 8 & 6 & 7 \\ 0 & 2 & 1 \end{array} \qquad \begin{array}{ccc} 0 & 3 & 6 \\ 8 & 3 & 1 \\ 0 & 2 & 1 \end{array}$$

Figure 4: First Example (top left), Permutations of 2nd Example (left except top), Combine for Possible Targets (right)

It will be shown later that even with a polynomial number of subset queries or equivalence queries with other patterns, the target could still be missed.

The learner effectively has only two examples to work with. These examples could have their columns permuted. The seven 2×3 matrices on the left represent the first 2-variable example followed by permuted versions of the second 2-variable example. Combining the two instances by subtracting rows produces the potential targets shown on the right. The middle rows of the results is of the most interest and is obtained by subtracting the top row of the first example from the top row of the permuted second example. Note that the actual set of numbers changes, not just their permutation. Each such result is one possible target. The learner has no way to find the correct target other than by trying all $6 = 3!$ of these ways of combining the two examples.

3.2 UT NON-LEARNABILITY

Theorem 2 *UT with onto semantics is not learnable with EQ and SQ queries.*

Proof: We generalize the above example which has 3 subtrees and therefore 3 columns in the matrix notation. Use s subtrees (columns) with c children each and exactly 3 variables (rows) in the target. As noted above, exact learning includes the worst case and therefore has the effect of making the queries behave in an adversarial manner and give the most uninformative answers allowed. As explained above, the learner can at most force this adversary to return two 2-constant examples. Let EQ return these two examples (called E_1 and E_2):

$$\begin{array}{ccccc} 0 & s & 2s & \ldots & (s-1)s \\ c & c-s & c-2s & \ldots & c-(s-1)s \\ \hline c & c-1 & c-2 & \ldots & c-(s-1) \\ 0 & 1 & 2 & \ldots & s-1 \end{array}$$

The learner can try combining these two examples without permuting their columns and generate the following result (which is one possible target). Combine these two examples to produce a possible target consistent with both by subtracting the top row of E_1 from the top row of E_2 (possibly after permuting the columns of the latter). With $c \geq s^2 - 1$, this process will always produce a non-negative result (for the middle row). If E_2 is not permuted, the result is:

$$\begin{array}{ccccc} 0 & s & 2s & \ldots & s(s-1) \\ c & c-s-1 & c-2s-2 & \ldots & c-s^2+1 \\ 0 & 1 & 2 & \ldots & s-1 \end{array}$$

But the adversary could say this is not the target. If the two examples were combined with an arbitrary permutation, a different result would be produced for each distinct (relative) permutation. To see this fact, first note that the resulting middle row is c minus the sum of the numbers directly above and below it. If the number above was taken from the i'th column of E_1 (counting the columns from 0 to $s-1$), then its value would be si. Similarly, the number below would be j if it came from the j'th column of the bottom row of E_2. The result in the middle row would be $c - si - j$ which will have a different value for each distinct i or j. The middle row therefore uniquely identifies what permutation was used implying each permutation produces a distinct target.

There are therefore exponentially many 3-variable targets consistent with the two 2-constant examples, E_1 and E_2. Consider any EQ with 3 variables. EQ could play the role of adversary, say that is not the target, and return as the negative counterexample the queried pattern with each variable replaced with a distinct constant. (Of course, if the 3-variable pattern given to EQ was inconsistent with either E_1 and/or E_2, give E_1 or E_2 as a positive counter example.) For an SQ with $\leq$2 variables, return no if the pattern is inconsistent with either E_1 or E_2. (Return yes if consistent with both.) If a *strong* version of SQ is used, meaning it returns a negative counterexample if the pattern is not a subset of the target, then return the same counterexamples as for EQ above. If EQ (or SQ) is given a pattern with more than 3 variables, return no with a counterexample formed by converting the pattern variables to constants. EQ could not be given a 2-variable pattern consistent with both E_1 and E_2 because these examples have different sets of numbers (0 through $s-1$ vs. multiples of s, etc.). All counts in an example must be present in the pattern of the same number of rows to allow matching–which is not possible for both examples.

Queries with 2-variable patterns therefore give no more information while queries with 3 or more variables reduce the number of potential targets consistent with the examples by at most one. The learner can not force an adversary controlling the queries to return a positive 3-constant example in poly many attempts. The adversary could therefore keep track of all potential targets that have not yet been tested by the learner and assume one of these could still be the true target . For any given learning algorithm, there is always some target that cannot be learned in poly time.

There is still the need to show the rows can not be confused; otherwise some combinations are duplicates of others and there are less than $s!$ total targets. The middle row of the result is produced by expressions of the form $c - si - j$ and will not have values all lying within a range (maximum minus minimum over the row) of $s - 1$. Therefore it is not interchangeable with the bottom row which ranges from 0 to $s - 1$. The elements of the middle row will all have different values modulo s and therefore can't be confused with the top row, the differences between whose elements are all multiples of s.

Every possible way of combining the two examples will therefore produce a distinct result for the combined 3-variable result. Each call to SQ/MQ or EQ with a 3-variable argument could only eliminate one of these possible targets regardless of whether SQ was called before or after the last EQ. Calls with 2-variable hypotheses will either have the same problem or just tell the learner what it already knows. Since there are $s!$ possible targets, they cannot all be tried with polynomially many queries. □

```
function into-main() initialize: h = {}
while EQ(h) gives counterexample t
            %(which is positive)
    repeat
        t = prune(t)   %local trimming
        t = partition(t)
            %partition repeated variable sets
        t = simult-prune(t) %simultaneously
            %prune identical subtrees
    until no generalization change
    h = h ⋃ t.
return h
```

Figure 5: Into-Semantics Main Bottom-Up Algorithm

This proof can also be expressed in terms of the following:

Lemma 3 (Angluin, 1988) *Suppose the hypothesis space contains a class of distinct sets $L_1 \ldots L_N$, and there exists a set $L_\cap$ which is not a hypothesis, such that for any pair of distinct indices i and j, $L_i \cap L_j = L_\cap$*

Then any algorithm that exactly identifies each of the hypotheses L_i using equivalence, membership, and subset queries must make at least $N - 1$ queries in the worst case.

Let $L_\cap$ be the set consisting of the union of the two 2-constant examples (with the constants converted to variables). This set is not in the hypothesis space because the latter allows only single trees. The L_i are the $s!$ possible targets consistent with those examples. To show the intersection of distinct L_i and L_j is $L_\cap$, note that any example matched by L_i (which has 3 variables) can be generated by mapping the variables onto constants. Since more than one variable might map to the same constant, this operation has the effect of partitioning the 3-element set of variables. Then for each partition, substitute a (distinct) constant subtree for all variables in that partition. The result will be either a tree matched by a variablized (constants to variables) version of one of the two examples or will isolate the variable corresponding to the middle row. In the former case, the generated example is in $L_\cap$. In the latter case, the example will not be covered by any other L_j by the same argument as above showing the middle row is different for each of the $s!$ possible targets. Then Angluin's lemma shows $s! - 1$ queries could be required to find the target.

```
procedure prune(t):
% perform local pruning of (example) tree t
for each leaf f of t
    if f is a constant
        change f to variable
        if not SQ(t) undo the change
    if f is a variable
        cut f and its edge from the tree
        if not SQ(t) undo the change
return t
```

Figure 6: Into-Semantics Pruning Routine

4 BOTTOM-UP ALGORITHM FOR "INTO" UF

This section describes the bottom-up algorithm for UF with into semantics and gives a detailed analysis of the algorithm.

4.1 ALGORITHM DESCRIPTION

The main part of the learning algorithm for into-semantics UF using EQ and SQ (equivalence and subset queries) is based on a bottom-up-from-single-example generalization approach and shown in Figure 5. All generalizations are tested with SQ (subset query) and undone if not accepted. The algorithm gets a new example tree from EQ, then tries three different generalization techniques. These techniques could potentially need to be used in any order; so the algorithm will simply keep trying all three until no further generalization occurs on that tree. The tree is then added to the hypothesis and the process repeats until the target is covered.

Pruning Algorithm: The pruning subroutine is shown in Figure 6. Pruning operates by changing each constant to a variable, and trimming each variable leaf. 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 generalization is possible.

A possible target is $A(\ B(z)\ B(yx)\ )$ and single training example could be $A(\ B(CD)\ B(EFG)\ )$. The pruning bottom-up algorithm then attempts to generalize one leaf at a time by either changing a constant into a variable or trimming the leaf and its edge. First the algorithm tries to change the leaf constants to (new) variables: $A(\ B(wv)\ B(uts)\ )$. Since the target leaves are all distinct variables, all of these changes are accepted. Then each leaf and its corresponding edge is pruned. Pruning w still gives a subset of the target. But cutting leaf v gives a tree pattern which matches some tree instances not matched by the target (i.e., those with no second-level children on one of its two subtrees) and must be undone. A similar process determines that two children are needed on the other subtree; at that point, the result is equivalent to the target.

```
procedure partition(t)
  %generalize repeated variables in tree t
    for each distinct repeated constant in t
        turn all instances of that constant into
            a new identical variable
        if not SQ(t) undo that change
    for each repeated variable (or constant) c in t
        create a new variable v
        partition-prune(t, c, v) %try partition set of c's

 % used by partition on scalars and
 % simult-prune on 1-level trees:
procedure partition-prune(t, o, n):
 %partition old by add new
designate the copies of o as o_1 ... o_k
for i = 1 to k
    add n_i to the parent of o_i
flag = true   %ensure delete at least one n and one o
for i = 1 to k
    if flag then   %try deleting o's 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 7: Repeated Variable Partition Algorithm for Into Semantics

Variable Partitioning Algorithm: The learner determines if a hypothesis with repeated variables (or constants) is not general enough by attempting to partition them into two distinct variables. Subroutine *partition-prune* (Figure 7) is used by routine *partition* on individual constants and variables as well as by *simult-prune* (Figure 8) on identical 1-level trees to perform the partitioning as follows. Each repeated variable (or constant) is tested separately by creating another (new) variable and putting the identical number of copies of that new variable in each subtree as there are of the original variable. (Fewer copies won't always work.) The variables are eliminated one by one while checking that the result is still accepted by SQ. If all copies of one variable were eliminated first, the result might be unchanged. Therefore elimination *alternates* between the two sets of variables (or variable and constant).

Let the target be $A(\ A(xxx)\ A(xyy)\ A(xyz)\ )$. The training example is assumed to be the same but with all 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 9 second-level children 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. Backtracking and eliminating r's gives $srr\ srr\ sss$. An attempt to further partition s gives 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 code has a "flag" which causes it to first eliminate the old/original children. Then once it has succeeded the code switches modes to try eliminating the new child/variable first. This code is designed 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 $xxx\ yyy$). (See the proof for further explanation.)

```
procedure simult-prune(t):
for each set of multiple identical 1-level subtrees s in t
        %(i. e., leaves plus immediate parents identical)
    for g = all possible minimal/1-step generalizations of s
            %(created by trim 1 leaf or
            %convert 1 constant to a variable)
        partition-prune(t, s, g) %try partition set of s's
    %(if successful, then while loop tries generalize
    %both s and g)
return t
```

Figure 8: Bottom-Up Algorithm for Identical Subtrees

Simultaneous Pruning Algorithm: The above generalization techniques are not sufficient to form a complete bottom-up learning algorithm. A training example could have several identical subtrees, and the target might require some identical parts but not require the subtrees to be as deep as in the example (e. g., identical variables matching the identical subtrees). The pruning algorithm would be unable to make any change because it makes only local changes and once the subtrees are no longer identical, the result will no longer match the target. The variable-partition algorithm only changes the leaves and will not trim identical subtrees. An additional routine is necessary to generalize such subtrees.

For each set of identical subtrees and possible way to generalize them, *partition-prune* is called with both the subtree and that generalization. Whenever a partition succeeds, then the resulting sets of subtrees are simultaneously generalized as much as possible. Then the other two generalization routines are given a chance.

Note that the problem of locating identical subtrees within the same tree is easy because the decision of whether two variables or constants are the same is always local to that part of the tree–unlike the matching problem in Theorem 1.

Let the target have 5 leaves with 2 distinct variables: $R(\ A(zzy)\ A(yy)\ )$. The single training example to be used has the same subtree ($B(CC)$) in place of all 5 variables: $R(\ A(B(CC)B(CC)B(CC))\ A(B(CC)B(CC))\ )$. The bottom up algorithm therefore requires generalization of these subtrees–previous techniques are inadequate.

The algorithm will then create generalized duplicates of the subtrees–say by eliminating one of the C's: $R(\ A(B(C)\ B\ (CC)\ B(C)\ B(CC)\ B(C)\ B(CC))\ A(\ B(\ CC\ )\ B(C)\ B(C\ C)\ B(C))\ )$. Using subroutine *partition-prune* on the double-C and single-C versions of the bottom subtree gives $R(\ A(B(C)\ B(C)B(CC)\ A(B(CC)B(CC)))$. The single-C subtrees can then be generalized simultaneously to turn them into a variable which corresponds to z in the target. The double-C subtrees are simultaneously generalized, yielding the target.

4.2 ANALYSIS OF ALGORITHM

We will now give a correctness proof of our algorithm. The algorithm repeatedly tries several generalization techniques while the hypothesis tree is a subset of a target tree until no further progress can be made. The proof must therefore show that all possible generalizations which keep a hypothesis tree a subset of some target tree will be tried. The generalizations of a hypothesis tree that are possible while keeping that tree a subset of the target will first be divided into two major classes. Those generalizations which can be performed locally on the hypothesis tree without changing any other part of the tree will be called *independent*. Those requiring simultaneous changes in multiple parts of the tree to keep that hypothesis tree a subset of the target will be called *dependent*.

Lemma 4 *Routine* **prune** *will perform all independent generalizations of leaves.*

Proof: Let s be a subtree of the hypothesis tree t that is not equilvalent to any tree in the target. For s to be generalizable without changing the rest of t, it must be either a constant that can be changed to a variable or an extra child or a subtree that can be trimmed. For the first possibility, *prune* will change the constant to a variable and still preserve $t \subset \mathcal{T}$ because the other children at all levels are still subsets of corresponding children in $\mathcal{T}$ (or else extra). For the other possibility, *prune* will eliminate a leaf which is not needed to maintain $t \subset \mathcal{T}$. $\Box$

Lemma 5 *Routine* **partition** *will perform all dependent generalizations of a leaf that matches a target variable.*

Proof: If a leaf cannot be "locally" generalized (by changing it alone) but to a variable in the target $\mathcal{T}$, than that target variable must be repeated. All copies of that repeated variable must match identical leaves. If this leaf (say, s) in the hypothesis tree t can be generalized, then it is not a member of a set of repeated variables in t that corresponds to a set of repeated variables in the target of the same size. There are therefore the following two possibilities: First, s is one of a set of identical constants all of which correspond to the same variable in $\mathcal{T}$. This generalization will be performed by the first for loop in *partition*. Second, more than one target variable matches the set of identical leaves of which s is a member. Routine *partition* will create a new variable v and pass it to subroutine *partition-prune*. That routine will perform a generalization (if possible) by splitting the set of repeated variables containing s into two sets as shown by Lemma 6. $\Box$

Lemma 6 *Given a hypothesis tree t, an "old" leaf or subtree o in t and a "new" leaf or subtree n to replace o, routine* **partition-prune** *will partition or split the set of occurrences of o's into two sets with o and n, while keeping t a subset of some target tree $\mathcal{T}$, if it ispossible to do so. Otherwise t remains unchanged.*

Proof: Designate the o's as o_1 through o_k. The following major steps are needed: First use the flexibility of into semantics to add n_i to the parent of o_i for each i. Second, prune one o or n at a time while guaranteeing a mix of o's and n's will be left–if partitioning is possible. Third, guarantee all the o's and n's that could be eliminated will be.

First prove if o does correspond to more than one target variable, then routine *partition-prune* will eliminate at least one copy of o and at least one copy of n. After adding the n's, the resulting tree t is accepted by SQ since into semantics allows extra children. Assume t is a subset of some target tree with at least 2 variables (say, x and y) matching o and the match between these has the following correspondence: Without loss of generality, use o with x and n with y. Then there is at least some ordering (permutation of children at each node) which produces a correspondence between t and the target, and *partition-prune* will perform a left-to-right scan of t and try to eliminate o's and n's one at a time. An o can be successfully eliminated if the corresponding point in the target has a y and an n will be successfully eliminated if the corresponding point in the target has an x. Since "flag" is set true initially, *partition-prune* will first try to eliminate o_i before it tries to eliminate n_i. Assuming there is at least one y in the target, it will succeed in eliminating o from t in the corresponding position. Either an n is eliminaed before this point or since the flag is reset, the routine tries to eliminate n_i before o_i from now on. Again assuming that there is at least one x in the target, it will succeed in eliminating n from t in the corresponding position.

The result is guaranteed to eliminate at least one o and one n (the procedure avoids the possibility of eliminating all of one variable when partitioning is possible). But that if o's cannot be partitioned at all then all occurrences of o will be either unchanged or replaced with identical copies of n.

Any children that can eventually be eliminated will be when it is first tested because SQ will indicate if there is any possible correspondence between t and the target for which the result a subset of the target.

A single (left to right) pruning pass through t is sufficient by the following argument. Suppose not, i. e., a node s can be eliminated from the hypothesis tree t aftter the first pass, but not during the first pass. Let the hypothesis tree be t' when this node is first considered for removal. Since $t - s$ (t with node s removed) is a subset of the target under some permutation of nodes, $t' - s$ also should be a subset of the target under some permutation of ndoes and removal of extra subtrees (in t' but not in t). Hence s should have been removed form t' when it was considered during the first pass.

The partition algorithm as a whole will therefore locate duplicate constant/variable leaves and partition them as needed into sets corresponding to separate target variables (or even target constants). □

Lemma 7 *Routine* **simult-prune** *will generalize a subtree corresponding to a target variable when that generalization is dependent on other parts of the tree.*

Proof: If a leaf cannot be generalized without also changing other parts of the hypothesis tree t and a target variable corresponds not to that leaf alone but to a subtree containing it, then that target variable must be repeated and corresponds to identical subtrees in t. Routine *simult-prune* generalizes the subtrees simultaneously and will therefore preserve the match between t and the target. This routine tries one level of generalization of these subtrees at a time. Each possible way of generalizing the identical, bottom 1-level subtrees is tried (i. e., removal of a node or changing a constant to a variable), and routine *partition-prune* is called with t, the old 1-level subtree and this new generalization of that subtree. This approach takes care of the case where identical subtrees are matched to more than one target variable–possibly at different levels (i.e., one variable matches a 2-level tree containing a 1-level tree identical to one matched by another target variable). □

Theorem 8 *UT with into semantics is learnable with equivalence and subset queries.*

Proof: The strategy is to first guarantee the algorithm can generalize all possible trees. Given a partially generalized hypothesis tree $\mathcal{Z} \subseteq$ the target $\mathcal{T}$, there exists some permuted version $\mathcal{P}$ of $\mathcal{Z}$ so each subtree of $\mathcal{P}$ is either a subset of the corresponding subtree of $\mathcal{T}$ or is extra (as allowed by the "into" mapping). This observation applies recursively. If $\mathcal{P}$ and $\mathcal{T}$ differ, there must be a node in $\mathcal{P}$ different from $\mathcal{T}$ and one of the following cases applies: If a difference between these two trees is not identical to or otherwise dependent on other parts of the trees, then Lemma 4 shows that routine *prune* will generalize it. If $\mathcal{T}$ has identical leaf constants or variables for which generalization of separate copies is dependent, then lemma 5 shows that these nodes can be generalized by subroutine *partition*. If a variable in $\mathcal{T}$ corresponds to identical, dependent subtrees in $\mathcal{P}$, then Lemma 7 shows routine *simult-prune* will generalize $\mathcal{P}$. Any part of $\mathcal{P}$ must be either independent of other parts of the tree or dependent and either a single leaf or an identical subtree, and there are only a finite number of ways $\mathcal{P}$ (as explained in the bounds argument below). The main program (*into-main*) repeatedly tries all three types of generalization until the tree canot be generalized further. So this algorithm will learn any UT target with into semantics.

Bounds: define a metric for (hypothesis) tree complexity as the total number of tree nodes n minus the number of distinct variables v plus e, the number of edges, giving $n - v + e$. 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 t of this metric for the example tree given to the algorithm. The bound on the number of queries is dominated by subroutine *simult-prune* because of the extra work needed to find each generalization. The total number of queries would be bounded by the number of ways to generalize 1-level trees times the number of nodes to be generalized. Each factor is bounded by t, so the overall bound is $O(t^2)$. □

Definition 1 *A concept class* $\mathcal{C}$ *is* **compact** *iff for any* $\mathcal{Z}$ *and* $\mathcal{V}_1, \ldots, \mathcal{V}_n \in \mathcal{C},\ \mathcal{Z} \subseteq \bigcup_{i=1}^{n} \mathcal{V}_i \Rightarrow \exists i$ *such that* $\mathcal{Z} \subseteq \mathcal{V}_i$

Lemma 9 *The class of into unordered tree patterns with an infinite label alphabet is compact.*

Proof: From the tree pattern $\mathcal{Z}$, form the tree instance e by leaving each constant in $\mathcal{Z}$ alone and substituting a distinct constant which does not appear in any of the $\mathcal{V}_i$ for each distinct variable in $\mathcal{Z}$. Since the label alphabet is infinite, there will always be sufficient constants. So $e \in \mathcal{Z} \subseteq \bigcup_{i=1}^{n} \mathcal{V}_i$ and therefore some $\mathcal{V}_i$ matches e. The only parts of $\mathcal{V}_i$ which match the new constants introduced in e are variables. Those constants in e can therefore be replaced by any subtree and still be matched by $\mathcal{V}_i$. This observation implies that any tree instance that matches $\mathcal{Z}$ also matches $\mathcal{V}_i$, so $\mathcal{Z} \subseteq \mathcal{V}_i$. □

Corollary 10 *UF with into semantics is learnable with equivalence and subset queries.*

Proof: By Lemma 9 this class is compact. Therefore in the algorithm in Figure 5 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 (and derived from the corresponding example trees). □

Corollary 11 *UF with 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 later case we get a negative counterexample from EQ and we can restart the algorithm with constants not already tried for this purpose [Amoth, Cull, & Tadepalli, 1998]. □

This strategy of learning in a bottom-up fashion from one example for each target tree is applicable under the following condition. The number of possible minimal or 1-step generalizations (those having no intermediate generalization) from

Concept Class	Queries	Matching Semantics	learn ?	justification
UT	EQ,SQ	1-1 onto	N	Theorem 2
UF	EQ,SQ	1-1 onto	N	to appear
UT	EQ,MQ	1-1 **into**	Y	Theorem 8
UF	EQ,MQ	1-1 **into**	Y	Corollary 11
UF	EQ,MQ	many-1 **into**	Y	see text

Figure 9: Unordered Tree Pattern Learnability Summary

any pattern must be polynomial. Equivalently, if the partial ordering representing all generalizations is viewed as a directed acyclic graph (dag) and undergoes a transitive reduction, then the out degree counting only the edges pointing toward more general hypotheses must be polynomial. This condition is not required to apply to the number of edges pointing to more specific hypotheses (and indeed it does not for UT since the label alphabet is infinite and the number of children is unbounded). The depth of the dag will correspond to the complexity of the training example used. The learning time will therefore be polynomial in both this depth and degree of the dag nodes. A bottom-up algorithm of this general class was also used for μ-UT (unordered tree patterns using onto semantics but without repeated variables) in [Amoth, Cull, & Tadepalli, 1998].

5 DISCUSSION/SUMMARY

Figure 9 summarizes the learnability results of the unordered tree classes using EQ and MQ or SQ. The queries and semantics for matching a tree to the instance tree are shown.

This paper proved the UT onto and UF into entries. But the UT proof depends on the hypothesis set being limited and therefore does not carry over to UF.

UF with many-to-one matching semantics which allows many children variables in the pattern to match the same instance child behaves like Horn-clause learning or learning description logics. This class is therefore learnable from EQ and MQ with the same algorithm using direct-product or cartesian-product-style least-general generalization (lgg) followed by pruning [Reddy & Tadepalli, 1999, Frazier & Pitt, 1996]. This type of algorithm will apparently not work for one-to-one **into** semantics because of difficulties with producing a true lgg.

These results show how the subtle changes in the matching semantics for trees have an effect on learnability. In particular, tree patterns with one-to-one and onto semantics are hard to learn from equivalence and membership queries. But with one-to-one and into semantics, they are easy to learn. Our negative result assumes that the learner is only allowed to use equivalence queries with hypothesis in the target class.

Future Work: A nonlearnability proof for UF with one-to-one onto semantics is under construction and is to be included in the Ph.D. dissertation of T. Amoth. This result also shows that UT is not learnable even when the learner is allowed to use equivalence queries on hypotheses in the form of forests.

Acknowledgments

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

References

[Amoth, Cull, & Tadepalli, 1998] Amoth, T. R.; Cull, P.; and Tadepalli, P. 1998. Exact learning of tree patterns from queries and counterexamples. In *Proceedings of the Eleventh Annual Conference on Computational Learning Theory*, 175–186. ACM.

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

[Arimura, Ishizaka, & Shinohara, 1995] Arimura, H.; Ishizaka, H.; and Shinohara, T. 1995. Learning unions of tree patterns using queries. In *Proceedings of the 6th ALT (Algorithmic Learning Theory)*, 66–79. Springer Verlag. Lecture Notes in Artificial Intelligence 997.

[Frazier & Pitt, 1996] Frazier, M., and Pitt, L. 1996. Classic learning. *Machine Learning* 25:151–193.

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

[Schapire, 1990] Schapire, R. E. 1990. The strength of weak learnability. *Machine Learning* 5:197–227.