AN ABSTRACT OF THE THESIS OF

Thomas R. Amoth for the degree of Doctor of Philosophy in Computer Science presented on June 1, 2001.

Title: Exact Learning of Tree Patterns

Signature redacted for privacy.

Abstract approved: _____

Paul Cull

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 thesis, we study the learnability of tree patterns from queries when the subtrees are ordered or 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. Angluin's exact supervised learning model is used, in which the learner has to exactly identify the target from a polynomial number of queries and in polynomial time.

We first show that ordered tree patterns and forests, with an infinite label alphabet (or equivalent condition), are learnable from equivalence and membership queries. Ordered forests and similar classes with bounded alphabet and branching factor are shown to be as hard to learn as DNF. We next 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. On the other hand, 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.

Exact Learning of Tree Patterns

by

Thomas R. Amoth

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Doctor of Philosophy

Completed June 1, 2001
Commencement June 2002

ACKNOWLEDGMENT

I thank the following people for their support and patience: my parents Victor and Lydia Amoth, and my mentors in the CS department during my graduate years, computer science professors Paul Cull and Prasad Tadepalli. I thank my thesis committee members Tim Budd and Mary Flahive Jonathan King, and former committee members Robert Robson, Lawrence Crowl, and Walter Loveland.

I thank the machine learning reading group members for interesting discussions on a variety of topics: Prof. Tom Dietterich, graduate students Dragos Margineantu, Valentina Bayer, Xin Wang, Dan Forrest, Chandra Reddy, and others.

Special mention goes to numerical analysis professors Bob Higdon, Andre Weideman, and the late Joel Davis for kindling my interest in that subject as well as Bob Higdon again for his introduction to symbolic mathematics systems.

I thank the professors who served as my mentors during my undergraduate years, all of whom encouraged me to skip ahead as much as two years in some cases: John W. Lee and Bob Burton in math, Victor Madson and Karl Kocher in physics, and Paul Cull in computer science (who also introduced me to LaTeX). I thank writing professors Carol Watt and Norma Rudinsky for giving me an initial boost in my writing skill. Special mention goes to English professor Jon Lewis for making liberal arts credit in film courses unusually interesting.

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

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

LIST OF FIGURES

LIST OF FIGURES (Continued)

LIST OF FIGURES (Continued)

LIST OF TABLES

EXACT LEARNING OF TREE PATTERNS

1. INTRODUCTION

Trees are abstract data structures that are used throughout computer science, in particular, in parsing and information retrieval. Trees also serve as a natural representation for mathematical expressions. Manipulating symbolic expressions is crucial in tasks such as computer algebra and natural language processing. Such manipulation is usually performed by a collection of rewrite rules. To decide if a rule applies, the system tries to match a *pattern*, representing the rule precondition, to an *instance* which is the data for the problem being solved. One approach to building high performance systems for these tasks is to hand-design these rules. However, hand-designing rules is time-consuming and involves exhorbitant human effort. This problem is sometimes called the *knowledge acquisition bottleneck*. A solution to this problem is to make the computer *learn* the rules from examples. In this thesis, we investigate several approaches to learning tree patterns that represent sets of symbolic expressions or parse trees from an expert that answers queries.

Several applications, such as information extraction, symbolic mathematics, compilation, and databases deal more directly with trees (and tree patterns) rather than with, for example, string patterns or integer or real-valued vectors (Cardie, 1997; Califf & Mooney, 1997; Aho, Sethi, & Ullman, 1987; Ullman, 1982). It is unnatural, if not impossible, to represent mathematical expressions as fixed-size vectors. Tree patterns provide more information than simple string patterns; the latter treats all of its individual elements as being on the same level–there is no hierarchy. Even when the input is given in some form other than a tree structure, many applications would derive a corresponding tree, e.g., by parsing, which effectively turns a string into a tree. The problem of deciding which trees are of interest from examples would then be a tree pattern learning problem.

Symbolic mathematics is useful because of its ability to derive formulas (rather than just numerical answers) and to aid in understanding a problem or discipline. For example, a user may want a formula for the motion of a body, given a formula for its velocity or acceleration. It would be more general and useful to have a result as a formula rather than just a curve plotted from numerical data. If the problem is complex, it would be useful to have a computer perform the symbolic manipulation rather than do that derivation by hand. Such symbolic mathematical formulas are naturally represented as trees. In mathematical applications, some operations require the parts of the structures to be in a particular order, so trees with these operators are called *ordered.* Other math operations such as functions which are commutative (add and multiply) allow their arguments to be in any order, making their trees *unordered.*

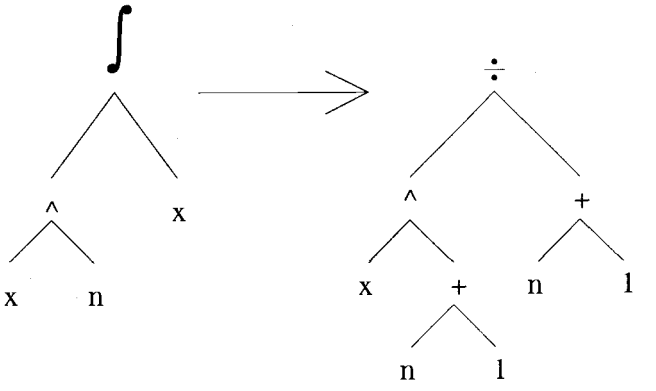


FIGURE 1.1. Simple Integration Rule

A sample integration rule is shown in tree form in Figure 1.1. This figure represents a transformation which takes a tree expression of the form on the left (where the *pattern variables* x and n can be replaced by other tree expressions)

and changes it into the tree on the right. The left tree has integration as its top-level operation; the integration is done with respect to the right child (the lone x). The right tree represents the result of integrating x^n with respect to x, i.e., $x^{n+1}/(n+1)$. It should be noted that leaves which are variables from the viewpoint of the integration operation can be constants from the viewpoint of tree pattern matching–and vice versa–these two notions of "variable" are independent.

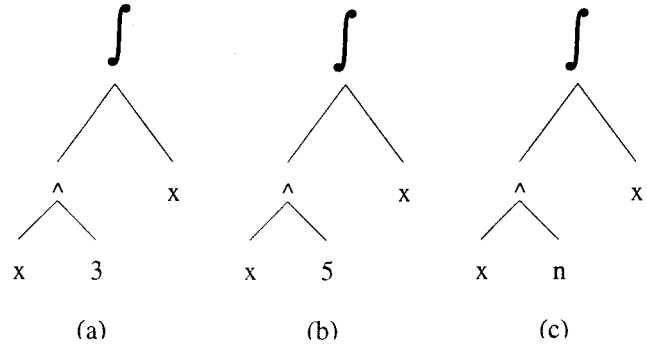


FIGURE 1.2. Simple Learning Illustration (Integration/Ordered Trees)

Programming symbolic mathematical systems by handcoding of rewrite rules is a difficult, laborious process which is prone to error. We ask if such programming could be automated. As a first step we investigate whether tree patterns can be learned from examples.

For example, consider the simplest possible problem for learning a symbolic integration rule. Here we consider learning the precondition (left hand side) of a rule of the form, $\int x^n dx \rightarrow x^{n+1}/(n+1)$. Figure 1.2 shows two training examples (a) and (b). The generalized result, (c), is produced by combining the two examples using a technique called least general generalization (lgg). It

represents the precondition of a general transformation rule which can also be seen as the set of all applications of this rule.

Tree instances represent specific expressions and are effectively the constants in the tree-learning problem because they represent only themselves. Tree patterns contain variables which can match any constant subtree and therefore represent a set of tree instances. The learning problem is to find a predetermined tree pattern called the *target* which is hidden from the learner. The learner is given a collection of examples through access to some *query oracles* and tries to derive the target tree pattern from this information.

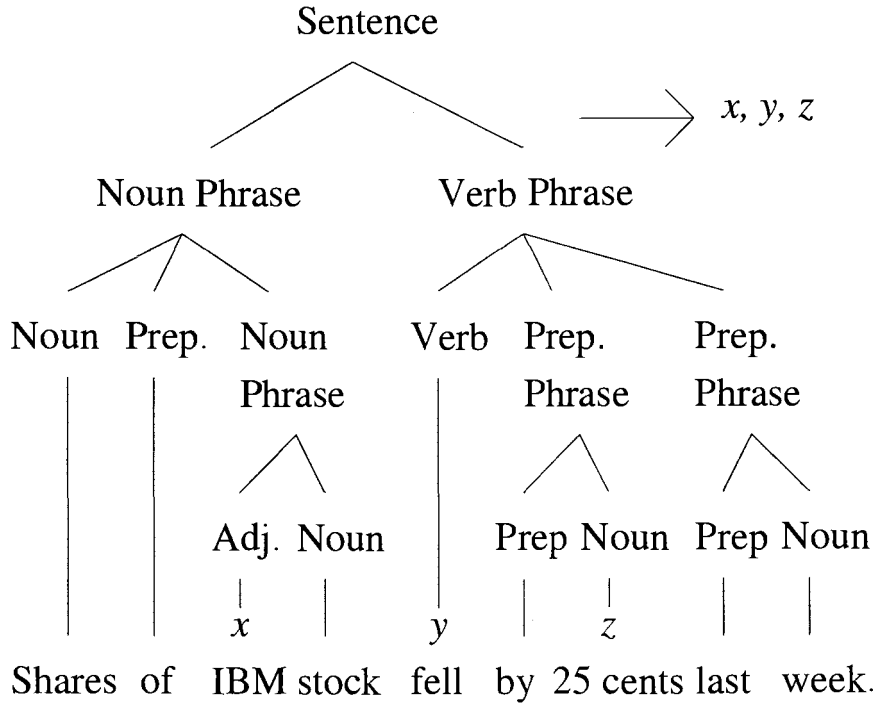


FIGURE 1.3. Information Extraction Example

With more complex integration techniques such as integration-by-parts, a single tree pattern is insufficient to cover all the cases in which a rule would apply. Therefore it becomes necessary to study learning of multiple tree patterns which we call *forests.*

Applications to process natural language text on the internet can first parse the text into tree form and then perform *information extraction* from those trees. An application may need to put the extracted information into a database which can then be queried using a formal language. One way to perform the extraction is to write rules that process the parsed trees. A rule would then be represented as a tree pattern as in Figure 1.3. The tree is a template for recognizing a certain structure so certain fields of interest can be isolated. The example sentence matches this structure, so the fields of interest can be paired with the variables x, y, and z. The values corresponding to these variables (i.e., "IBM", "fell", and "25 cents") can then be put into a database that records stock news.

The learning problems discussed in this thesis have the following form. First, there is a set or *universe* of *instances* called an *instance space.* A *hypothesis* or a *concept* represents a subset of the instance space. A *hypothesis space* is a set of hypotheses for the learner to consider. The *teacher* picks an arbitrary hypothesis from the hypothesis space called the *target* and hides it from the learner. The job of the learning algorithm is to form hypotheses which are better and better approximations to the target until a hypothesis is found that is equivalent to the target. The learner is allowed to use query oracles which answer questions about the relation between a specified example or hypothesis to the target (e.g., is a specified hypothesis a *subset* or *superset* of the target). In the learning framework used in this thesis, one of these query oracles tells when the hypothesis is equivalent to the target.

The following is a simple facial pattern learning problem. The instances ("constants" in the learning problem–which represent only themselves) are 8 possible faces ranging from  to  with 3 binary features. A *face pattern* or hypothesis may have some missing features and represents the set of all instances where the missing features can take any value.

So the pattern  represents  , and  . The more missing features a face pattern has, the larger the set of instances it represents and therefore the more general it is. A learning algorithm for this problem would take some instances as examples (e.g., the two instances shown above) and find the least general pattern which covers/includes those instances (the face pattern above).

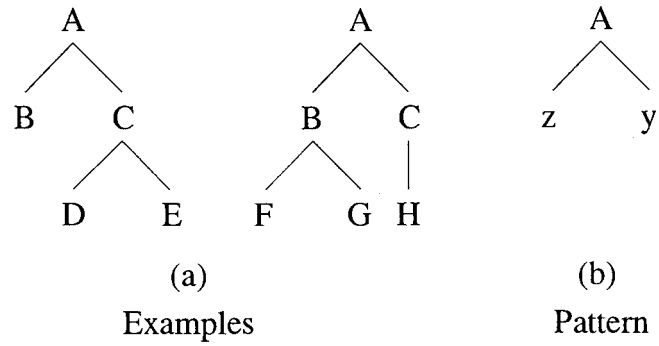


FIGURE 1.4. Learning with Tree Patterns

Figure 1.4 shows a sample learning problem for learning with tree patterns. The *instance space* consists of trees labeled with uppercase letters, which represent constant *labels*, on both leaves and internal nodes. Hypotheses are single tree patterns which have constant labels on internal nodes and constants

or variables on leaves. Lower case letters are used to represent variables and can match any constant subtree. The target tree pattern is hidden from the learner, which must therefore rely on example trees to learn it. Figure (a) shows two examples which are included in the set of instances matched by the target (b). The learner must derive a single tree pattern which matches each example, i.e., a tree pattern which becomes identical to the example when a constant label or subtree is substituted for each variable. First consider finding an appropriate pattern to match the left subtrees of two examples. The examples have B with no child and B with two children, respectively. But no tree pattern with any specific number of children can match a tree instance unless it has the same number of children. Therefore no single tree pattern can match both of these example subtrees unless it matches every possible subtree. The result must therefore be a variable (z in (b)). Now find a subpattern for the right subtrees. The examples have C with two children and C with one child. The same situation applies, so the result must be another variable (y in (b)). Part (b) is therefore the most specific tree pattern which matches both examples. This technique for combining examples is known as *least general generalization* (or *most specific generalization*) and the class of which this learning problem is a member is known as *ordered trees*. With ordered trees, the corresponding children must be in the same order at each corresponding node in the pattern and instance (Chapter 3). With unordered trees, the correspondence can be any permutation (Chapters 5 and 6).

To formally analyze learning, 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 hypothesis space. 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).

When learning tree patterns, the instances are ordered or unordered trees with nodes labeled by constant identifiers. The hypotheses are tree patterns and with leaves labeled with constants or variables. A tree pattern matches any tree instance that can be obtained by replacing the pattern's variables with constant subtrees (possibly with other adjustments as allowed by various matching semantics discussed below).

Many transformations on trees will work for a whole family of expressions which cannot be expressed as a single tree pattern. So we also consider the learning of *ordered forests* (OF) which are finite sets of tree patterns. Certain mathematical operations are very rigid in that each of the subexpressions must appear in a fixed order. But other operations use commutative operators like addition and multiplication and allow subexpressions to occur in any order to be matched by a tree pattern. When an expression is represented as a tree pattern, these properties imply the children of a tree node having one of those operators can be in any order to be matched by a tree pattern. So we consider learning of both *ordered* and *unordered tree patterns*. These tree patterns could be the antecedents and consequents of our mathematical transformations.

Having only examples (from an equivalence oracle) is insufficient for efficient learning, so we consider learning with a variety of oracles. After seeing some examples, the learner can form a hypothesis pattern and ask the teacher a question about this hypothesis pattern and the correct "target" pattern. For example, an *equivalence oracle* (EQ) would be given the hypothesized pattern and if the pattern were equivalent to the target pattern then the oracle would say *yes*. But if the hypothesized pattern were not equivalent to the correct pattern, the oracle would respond *no* and also give a counterexample, which would be either an example included in the hypothesized pattern but not included in the target pattern, or an example not included in the hypothesized pattern but

included in the target pattern. Query oracles allow the learner to ask questions about the relationship of a specific hypothesis pattern or instance to the target pattern. The response is either a *yes* or a *no* with an optional counterexample, depending on the oracle used. We consider a variety of such oracles and attempt to show which sets of oracles are sufficient and which are insufficient for effective learning. For example, a *membership query* (*MQ*) is given a single instance and returns *yes* iff the instance is a member of the target. A *subset query* (*SQ*) is given a possible hypothesis and returns *yes* iff that hypothesis is a subset of the target.

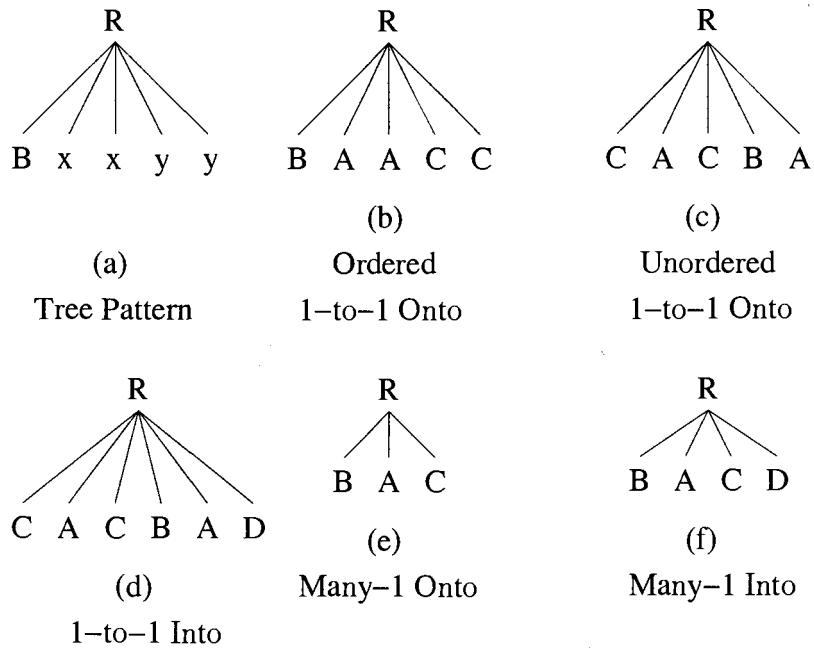


FIGURE 1.5. Mapping Unordered Tree Pattern Children to Instance Children

Learnability is a function not only of what queries are allowed but also of how matching is defined. Matching semantics are classified by the type of mapping allowed from pattern children to instance children of corresponding nodes. Learnability with one-to-one onto, one-to-one into, and many-to-one into matching semantics is shown. In Figure 1.5, tree pattern (a) matches the tree instances (b) through (f) according to various matching semantics. Tree instance (b) is matched by pattern (a) by *ordered one-to-one onto* semantics because substituting A for x and C for y in (a) without reordering the children produces (c). (This same match also works for the other semantics mentioned below). The pattern matches (c) by *unordered one-to-one onto* semantics since substituting A for x and C for y in (a) and then permuting the children produces (c). (Other mappings and permutations are possible.) The various kinds of onto semantics require all instance children to be mapped to by some pattern child, but into semantics allows an instance to have extra children that do not take part in a match. Tree pattern (a) matches instance (d) by *unordered one-to-one into* semantics using the same substitution and permutation as for (c) but with the addition of an extra child D to produce (d). Instance (e) is matched by pattern (a) according to *many-to-one onto semantics* by the same substitution but with both x's mapping onto the same A child and similarly the y's mapping onto the C. Similarly, (f) is mapped to by *many-to-one into* semantics with the same substitution and mapping, but there is an extra child (D) that is not in the range of the mapping.

We show the following results on ordered trees and forests with one-to-one onto semantics in this thesis. Ordered trees with one-to-one onto semantics are learnable from equivalence queries alone. Ordered forests with the same semantics and an infinite constant alphabet (or equivalent) are learnable from equivalence and membership queries. Similar (but not necessarily identical) results are also shown by (Page, 1993; Arimura, Ishizaka, & Shinohara, 1995; Ko, Marron, & Tzeng, 1990).

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). Unorderered trees are learnable from superset query oracles, and unordered forests are learnable with either superset and equivalence or superset and subset oracles.

With *one-to-one into* semantics, unordered trees and forests are learnable from equivalence and membership queries. With many-to-one into semantics, unordered forests are learnable from equivalence and membership queries. We also show some strong negative results by proving that some classes of tree patterns are not learnable. Unordered trees using superset and membership queries are not learnable because the learner has no way to test when it is done. Unordered trees with equivalence and subset/membership queries are not learnable because the oracles give too little information. Many-to-one onto semantics has the same difficulty.

The rest of the thesis is organized as follows: Chapter 2 formally defines the learning framework. Chapter 3 shows that ordered trees and forests are learnable with one-to-one onto semantics. Chapter 4 ties the learnability of various tree classes with bounded alphabet and branching factor to the learnability of DNF. Chapter 5 shows that unordered trees and forests are not learnable with equivalence and subset queries under one-to-one onto semantics. Chapter 6 shows the learnability of unordered trees (and forests) with superset (and equivalence) queries. Chapter 7 gives formal definitions for all matching semantics and shows unordered trees and forests are learnable from equivalence and membership queries using one-to-one into semantics. Chapter 8 ties trees with many-to-one into semantics to Horn clauses. This chapter also shows many-to-one onto semantics is not learnable for trees with equivalence and sub-

set queries. Chapter 9 gives a summary of the results, discussion, conclusions, and future work.

2. PRELIMINARIES

2.1. General Definitions

The following definitions and explanations are applicable to learning in general and not specific to trees. First learning is defined (for the exact framework) in general. Some general properties of the concept classes of particular interest to this work are discussed. Then the query oracles are defined and explained.

There are may variations of machine learning problems. A *supervised learning* problem uses a teacher in the form of query oracles which provide examples and (positive or negative) class labels for instances selected by the teacher–or the learner. It is therefore immediately clear to the learner which class label given instance has. Another mode of learning, called *reinforcement learning*, also uses a teacher but class labels are only provided *indirectly* usually in the form of a delayed reward, so it is not immediately clear which choice is correct. With this type of learning, the learner has the additional burden of determining which choices correspond best to the rewards. The least studied type of learning is called *unsupervised learning* which has neither class labels nor a teacher. This type of learning is also called *clustering* because the task of the learner is to discover patterns or clusters in the data without the guidance of any externally-provided answers.

Supervised learning is used throughout this thesis. The supervised concept learning problem is to find an unknown *concept* which represents a set of *instances*. These instances are the specific values or constants of the learning problem and are members of some universal set of instances called an *instance space*. A concept represents a set of instances which is usually infinite and therefore is implicitly described in some finite form. In each learning session, an unknown concept called a *target* is to be determined by the learner. The learner may receive only examples labeled as being either in or not in the concept

(*positive* and *negative* training examples, respectively) for the chosen learning problem. Sometimes other queries or oracles help the learning process. This unknown concept is often thought of as having been chosen by a teacher which also provides the examples and responds to the queries. The learner then tries to find an equivalent concept called a *hypothesis.*

For example, suppose the universal set of instances is all possible pieces of furniture (including chairs and tables). Each instance is defined by its characteristics (number of legs, whether upolestered, material used, shape of top, presence of arms, etc.). In supervised learning, the teacher chooses the subset of chairs as the target, then gives examples, some of which are specified to be chairs and some are not. The learner then looks for patterns in the positive and negative examples and decides which characteristics correspond to and predict that a given piece of furniture is a chair (e.g., 4–or more legs, top not flat–may have arms, tall back).

A learning problem is a triple $<\mathcal{I}, \mathcal{H}, L>$, where $\mathcal{I}$ is a set of instances called the *instance space*, $\mathcal{H}$ is a set of hypotheses called the *hypothesis space* or *concept class*, and L is a matching function from $\mathcal{H}$ to subsets of $\mathcal{I}$. Each hypothesis $h \in \mathcal{H}$ *represents* a set of instances in $\mathcal{I}$ defined by the matching function $L : \mathcal{H} \rightarrow 2^{\mathcal{I}}$. An instance $x \in \mathcal{I}$ is a *positive example* of $h \in \mathcal{H}$ according to L if $x \in L(h)$ and a *negative example* of h if $x \notin L(h)$. We informally speak of h *containing* or *covering* x if $x \in L(h)$.

Usually hypotheses contain an infinite number of elements and must therefore be specified in some language which allows a specification of finite size to define this infinite set of elements. All the hypothesis specified in a given language constitute a hypothesis space $\mathcal{H}$. It is meaningful to talk about the learnability of a hypothesis space (many possible hypotheses) but not about a single set of instances (i.e., one specific hypothesis) because the latter represents just a single answer and it would be trivial to encode that one answer in the learning algorithm.

For learning to be meaningful, the concept class used must have a property called *bias*. Bias is the learner's prior knowledge (Natarajan, 1991) or any basis for choosing one generalization over another. (Mitchell, 1980) gives the definition, "... bias to refer to any basis for choosing one generalization over another other than strict consistency with the observed training examples." The type of bias from that reference used here is, "the generalization language is not capable of expressing all possible classes of instances" which is sometimes also called *restricted hypothesis space bias*. Otherwise, there would be no way for a learner to predict the class label (positive or negative) of an example it has not yet seen–and therefore effectively no learning at all. The hypotheses in a concept class are therefore described in some language which restricts the possibilities to be considerably less than all possible subsets of the instances. The faces pattern class (Section 2.2) and the class it is isomorphic to (Boolean conjunctions of three variables), restrict hypotheses to those which can be represented by a conjunction. For example, a hypothesis could represent all faces with open eyes and a smile, but no hypothesis represents all faces having exactly one of these features. Therefore, if a (positive) example is seen with open eyes and a frown and another example has closed eyes and a smile (with either nose), then necessarily open eyes and a smile is in the target as is closed eyes and a frown. No allowable hypothesis in the concept class includes those two training examples without also including the examples with all combinations of these features.

Similarly, the tree pattern classes have bias. Those classes which permit only a single tree pattern as a hypothesis require all instances to have the same basic structure; i.e., the upper part of the instance trees must look like the corresponding part of the tree pattern. The *forest* classes which permit more than one tree pattern in a single hypothesis also have a bias even though they are more expressive and hence less restricted than the single-tree classes. First, only a finite number of possible upper-level structures are permitted.

Second, wherever a pattern variable appears, any possible constant subtree can be substituted for it; restriction to some arbitrary set of possible substitutions is not permitted. Thus, the forests classes actually greatly restrict the possible sets of instances represented even if it might not seem to be that way at first.

There is another way to look at the restriction which bias imposes on the set of possible hypotheses. Note that the set of instances for the tree classes is *countably infinite* (the same as the number of integers). The class of possible subsets of these instances is therefore *uncountably infinite* (the same as the number of real numbers). But each allowable hypothesis in any of the classes has a finite representation, so there is some way to encode it as an integer. Therefore the set of possible hypotheses in any of the tree classes is countably infinite–far less than the number of subsets of instances. Without a restriction of this type, learning in these classes would be impossible.

Exactly learning a hypothesis space corresponds to identifying a target concept that the teacher chooses from the hypothesis space (Angluin, 1988). The learner is allowed to ask various queries such as equivalence, membership, superset, and subset. The equivalence query (EQ) oracle allows the learner to determine when it is done learning. EQ is also a source of examples that give the learner a clue on how to adjust its tentative hypothesis to make it converge to the target. The equivalence and other queries are defined more precisely as follows.

- 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 *no* 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 argument 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).

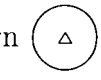
Definition 2.1 *A concept class $\mathcal{H}$ is* exactly learnable *from a set of queries $\mathcal{Q}$ if there is a learning algorithm $\mathcal{A}$ and polynomial p which meet the following condition. Given any possible (target) hypothesis T in $\mathcal{H}$, $\mathcal{A}$ always finds a hypothesis in $\mathcal{H}$ 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 demanding) than the Probably Approximately Correct (PAC) model of Valiant (Valiant, 1984b) that requires that the examples are chosen using a fixed but unknown distribution. This framework is discussed along with PAC-predictability in the introductory section of Chapter 4. In the PAC learning model, the learner merely needs to learn a concept which 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 and membership of examples in a hypothesis can be evaluated in polynomial time, 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.

2.2. Simple Learning Problem: Faces

The learning problem will be introduced through a simple example of learning *facial patterns.* An *instance* (constant or specific example) is a *face,* for example,  . Let us assume that each face has one of two types of eyes, either circles or arcs, e.g., either o o or ⌒ ⌒, one of two types of noses (either o or △), and one of two types of mouths either ⌣ or ⌒. So there are a total of 8 different faces:  . These 8 faces are the set of instances, i.e., the *instance space.*

We want to be able to learn sets of faces. Since there are 8 faces, there are $2^8 = 256$ possible *sets* of faces. Fortunately we will *not* try to represent all of these sets. Instead, we define a *face pattern* which is similar to a face, but some features (eyes, nose, mouth) may be missing. The idea is that a face pattern represents the set of faces that have the features which are explicit in it. If a feature is missing in the face pattern then a face in the represented set may have either value for the missing feature. For example,  is the face pattern which represents the set {  ,  }. Face pattern  represents the set {  ,  ,  ,  }.

The face patterns are the *hypotheses.* The set of all face patterns is the hypothesis space or *concept class.* A face pattern *matches* a face instance when the instance includes all the features present in the pattern.

In our learning task, we want to discover which face pattern (hypothesis) matches the target set of faces. If the target can be any subset of the instance space, this task is impossible, because there are 256 sets of faces, but only 27 face patterns. If we include every possible subset of the instance space in the hypothesis space, it becomes impossible to generalize and find the correct

hypothesis without being given every possible face as a (positive or negative) training example. So instead we assume that the target *has* a face pattern representation, and we seek to discover which face pattern matches the target.

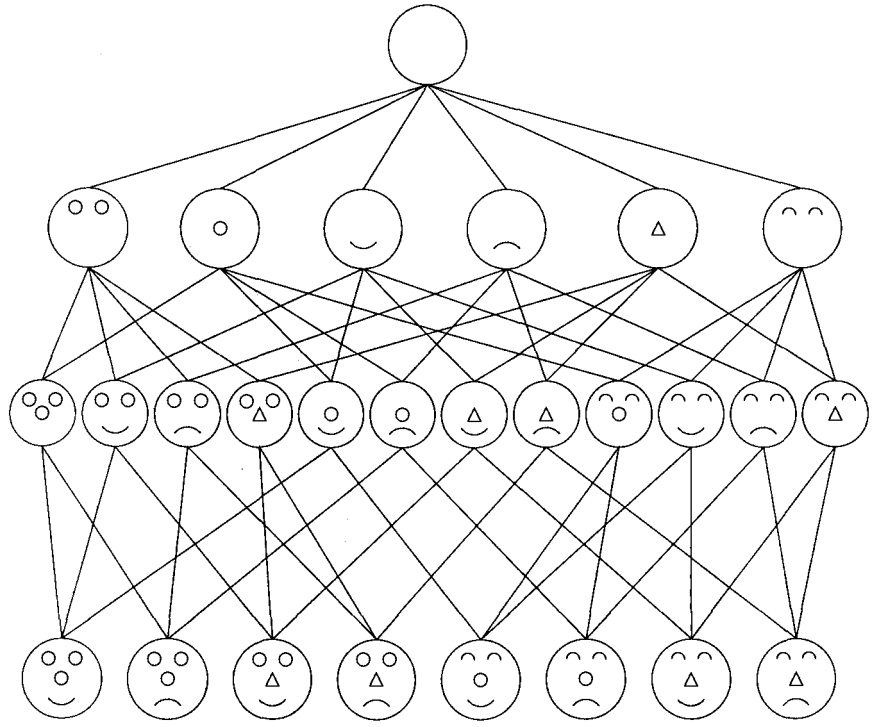


FIGURE 2.1. Faces Generalization Lattice/Hasse Diagram

The face patterns can be arranged into a *Hasse diagram* (or *generalization lattice*) as in Figure 2.1 so that a face pattern is connected upward to the patterns which represent supersets of the sets represented by the lower pattern. Only direct supersets are explicitly represented in that if the set of face instances matched by a face pattern A contains the set matched by a face pattern C (denoted $A \supseteq C$) but there is another pattern B so $A \supseteq B \supseteq C$, then no

edge is drawn directly between A and C since $A \supseteq C$ is implied by the edges between A and B and between B and C (*transitivity* of containment). At the lowest level are face patterns which represent a set with only one face. Each face pattern represents all of the faces at the lowest level which can be reached by following the lines downwards and is a superset of any other face pattern than can be similarly reached. We use the *single-representation trick* which uses the single face at the lowest level as both a face instance and the face pattern representing the singleton set containing that face instance. At the top, is the face pattern  which represents the set containing all 8 faces.

The number of face patterns is 1 for no facial features, 6 for one feature, 12 for two features and 8 for all three features. These numbers correspond to the terms in the binomial expansion of $(1 + 2)^3$.

Learning for this hypothesis space is very easy. If our learner is given a face instance, it responds with the hypothesis consisting of the face pattern which matches only the given instance. Now the learner asks the oracle if this is the target (or is equivalent to the target). If the oracle says *yes*, then the learner has correctly and quickly learned the target. If the oracle answers *no*, and gives a counterexample, i.e., another face that is covered by the target but is not covered by the hypothesis, then the learner finds (computes) the face pattern which is lowest in the diagram and covers both the old hypothesis (the old face pattern) and the new face instance. In this example, it is easy to show that this *least upper bound* of the generalization lattice, also called *least general generalization* exists, is unique, and is easy to compute.

The learning then proceeds in a similar manner. The learner submits a hypothesis to the oracle, and either the learning is finished or a new counterexample is returned, a new hypothesis is computed and the learning continues. In each step, the learning algorithm creates a new pattern by eliminating any features which are not consistent with all of the examples. This process can be done incrementally–i.e., by maintaining a hypothesis which is consistent with

the examples seen so far and adjusting the hypothesis (by generalizing it) for each new example. In this simple learning problem, the learning can take at most 4 rounds since the new hypothesis is always at a higher level in the diagram than the old hypothesis.

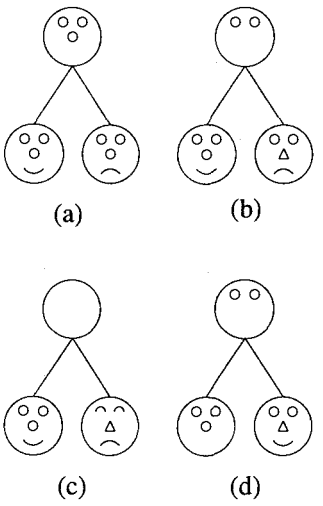


FIGURE 2.2. Computing New Faces Hypotheses From Examples/Old Hypothesis

Figure 2.2 gives some examples of computing the new hypothesis from the old hypothesis and a new instance. Each diagram within that figure shows the two example faces (or face patterns) which are to be combined to form the result above. Lines are drawn to indicate that the result is more general than either of the two faces supplied just as they are in the generalization lattice (Figure 2.1). In diagram (a), the two faces combined are the same except for the mouth, so the result has the mouth omitted. Similarly diagram (b) omits both the mouth and nose. Diagram (c) differs in and omits all three features. Diagram (d) starts with a face pattern with the mouth omitted; it keeps that

omission and also omits the nose which differs between the two faces to be combined.

As a sample learning sequence, suppose the first example is  . The learner can interpret this instance as a tentative hypothesis (which only represents this one instance). Let the second training example be  . The learner would find the face pattern which covers both of these examples by eliminating the feature which differs between them:  . Suppose the third training example is  . The resulting hypothesis face pattern must have the mouth eliminated to cover all of these examples:  . If this pattern was indeed the target, no further steps would be needed, and learning would terminate at this point. But suppose a fourth example,  , was given. Then the only pattern which covers all the examples is the universal pattern,  .

The faces problem above is equivalent to learning the concept class of Boolean conjunctions over a space of 3 boolean variables. Let these 3 variables be e, n, and m representing the eyes, nose, and mouth, respectively. Let *false* for each of the three variables represent closed eyes, triangular nose, and frown, respectively (and *true* be the opposite features).

Then the instances might be represented by boolean conjunctions with all three variables. The hypotheses or face patterns are boolean conjunctions with 0 to 3 variables. Instead of using a distinct notation for instances and for patterns that represent only one instance, we use boolean conjunction notation for both. Therefore the 8 possible instances (corresponding to the 8 faces shown at the beginning of this section) are represented as enm, $en\overline{m}$, $e\overline{n}m$, $e\overline{n}\,\overline{m}$, $\overline{e}nm$, $\overline{e}n\overline{m}$, $\overline{e}\,\overline{n}m$, and $\overline{e}\,\overline{n}\,\overline{m}$, respectively. The equivalent generalization lattice for Figure 2.1 would have these 8 boolean conjunctions of 3 variables on the bottom row, and the 12 boolean conjunctions with 2 of these 3 variables on the row next to the bottom (ranging from en to $\overline{e}\,\overline{m}$). The next row up would have just individual variables e through $\overline{m}$, but the top row would have some symbol

that represents always *true*. For example, the hypothesis em represents the set $\{enm, e\overline{n}m\}$ and $\overline{n}$ represents $\{e\overline{n}m, e\overline{n}\,\overline{m}, \overline{e}\,\overline{n}m, \overline{e}\,\overline{n}\,\overline{m}\}$.

Generalization works like Figure 2.2. In example (a), enm and $en\overline{m}$ combine to yield the pattern en. In example (b) enm and $e\overline{n}\,\overline{m}$ combine to yield e. Example (c) combines enm and $\overline{e}\,\overline{n}\,\overline{m}$ to get *true*. Example (d) combines en (which represents 2 instances) with $e\overline{n}m$ to get e. The general technique is to eliminate those variables which are not the same in all examples or hypotheses combined.

The learning example sequence for faces described earlier is the following. The first two training examples are enm and $e\overline{n}m$, and the resulting pattern is em. The third example is $e\overline{n}\,\overline{m}$ which yields the pattern, e. The fourth example is $\overline{e}n\overline{m}$ and yields the final pattern *true*. Thus, learning in this class of boolean conjunctions over three variables is equivalent to learning with faces in that all operations correspond between the two classes.

This example is easy to understand because the instance space and the hypothesis space are both finite. In more realistic situations as in the case of tree patterns, one would expect both of these spaces to be infinite. Nevertheless, similar algorithms apply with some variations.

3. LEARNABILITY OF ORDERED TREES AND FORESTS

This chapter studies the learnability of *ordered trees* and *ordered forests* (*OT* and *OF*, respectively). The algorithms for both OT and OF use the equivalence query (EQ) to provide examples and the *least general generalization* (*lgg*) algorithm to combine multiple examples/patterns. OT is shown to be learnable with EQ. OF is shown to be learnable with EQ and either subset or membership queries (SQ or MQ) to help decide which hypothesis tree pattern each example should be combined with.

Ordered trees have been shown to be learnable with equivalence queries alone (Ko et al., 1990; Page, 1993; Goldman & Kwek, 1999). Ordered forests with a bound on the number of trees have been shown to be learnable with equivalence queries alone (Goldman & Kwek, 1999). This algorithm gives a time bound which is exponential in the number of tress in the forest. A different method to show learnability of a bounded number of trees using equivalence and membership queries was used by (Arimura et al., 1995). This result uses a number of queries bounded by the number of target trees plus the number of symbols in the constant alphabet to prove that membership queries will perfectly simulate subset queries. Our results assume an infinite label alphabet and use negative counterexamples to detect when the simulation of subset queries by membership queries fails. Ordered forests without bounding the number of trees are learnable from equivalence and membership queries (Page & Frisch, 1992; Page, 1993).

The class of *ordered tree patterns* consists of single tree patterns including ϕ which represents the empty set of tree instances. The *ordered trees* (OT) class therefore only has representations for those sets of tree instances whose members all have the same basic "tree structure" such as the set of all tree instances having A as the root node label and three children. OT is actually

equivalent to the class of *first order terms* defined over a set of function symbols, constants, and variables (Arimura et al., 1995).

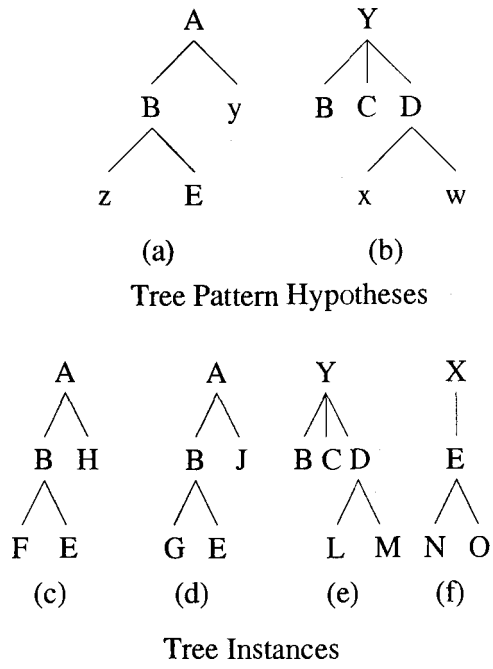


FIGURE 3.1. Ordered Tree and Forest hypotheses example

Figure 3.1 has two tree pattern hypotheses (a) and (b) and instances (c) through (f). Tree (a) is a hypothesis in the OT class and matches (c) and (d) but not (e). There is no way a single OT hypothesis tree pattern could cover (c), (d), and (e) without covering every possible tree instance including (f) because (e) has 3 subtrees at the top level and (c) and (d) have only 2. The *ordered forests* class includes all *finite unions*/collections of ordered tree patterns and is denoted *OF*. Hence, an *ordered forest* hypothesis could include tree patterns

such as (a) and (b) and would then cover all three instances (c), (d), and (e) but not (f).

The learning algorithms for both classes use the *least-general generalization* (*lgg*) algorithm as a tool to combine multiple examples into the best possible hypothesis. The lgg algorithm finds the tree pattern covering the smallest set of instances that includes both of two tree instances or patterns given to it as arguments. Both classes use an equivalence oracle to obtain training examples. The OF algorithm uses a membership or subset query to determine whether two examples should be in the same tree pattern. The OT algorithm needs no oracle other than EQ since the hypothesis space guarantees that there is a single tree pattern consistent with all the examples.

3.1. 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 *label alphabet*. Except where stated otherwise, we only consider *infinite label alphabets* in this thesis. A *tree instance* is a tree without any variables so all nodes including the leaves have 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 ϕ 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.The definition is recursively applied to those children until the leaves of the tree pattern are reached. 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.

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

Definition 3.1 *A tree pattern* γ matches *a tree instance* t *(or another tree pattern* β*) written* $\gamma \succeq t$ *(*$\gamma \succeq \beta$*) 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 *(or* β*).*

Definition 3.2 $L(P)$ *is the set of instances matched by tree pattern* P *(i.e., the language represented by* P*).*

Lemma 3.1 *For ordered one-to-one onto semantics, if* $P \succeq Q$ *and* $Q \succeq R$, *then* $P \succeq R$.

Proof (Sketch): A full proof will be given for Lemma 7.4 in Chapter 7. For ordered trees, the essential point is that the composition of two substitutions is a substitution (Lemma 7.1). □

Theorem 3.2 *Let* P *and* Q *be two ordered tree patterns. Then* $L(P) \supseteq L(Q)$ *iff* $P \succeq Q$.

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 3.1, and $I \in L(P)$. Hence $L(P) \supseteq L(Q)$.
only if: Given $L(P) \supseteq L(Q)$. Since the alphabet of constant labels is infinite, we can choose any 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 $I\sigma^{-1} = Q$. $L(P) \supseteq L(Q)$ also implies $I \in L(P)$ and therefore $P\sigma' = I$ for some σ'. So, $P\sigma'\sigma^{-1} = Q$, $P\sigma'' = Q$ for some σ'' by Lemma 3.1, and $P \succeq Q$ by Definition 3.1. □

3.2. Least-General Generalization (lgg):

The primary operation used by the ordered tree and forest algorithms is the *least general generalization* (*lgg*) (also known as the *most specific generalization*). The purpose of this operation is to form the hypothesis from two arguments (either examples/tree instances or hypothesis tree patterns) which satisfies the following two criteria. First, this resulting combined hypothesis (tree pattern) will cover all the examples covered by each argument given to

lgg. Second, this result will also be the minimal or most specific hypothesis tree pattern which satisfies the first criteria.

The learning algorithm then uses lgg to combine successive examples until the target is found. The examples are therefore combined in a manner that is "minimal" (i.e., covers the smallest set of instances) while still keeping the result a member of the class of tree patterns. A learning algorithm will therefore use lgg to create a hypothesis that is just sufficient to cover the examples it is given. Since the resulting hypothesis pattern is the least general which covers the examples/arguments, it cannot be *overgeneral* (more general than the target).

The tree patterns form a *generalization lattice* (Birkhoff, 1967) with lgg as the *join* operation, and intersection of the corresponding sets of instances can be used as the *meet* operation. The *more general than relation* is the *partial ordering* operation used by the lattice. A tree pattern is more general than another if it represents a set of tree instances which is a superset of the tree instances represented by the other tree pattern. The intersection of two tree patterns is either another tree pattern or the symbol (ϕ) corresponding to the empty set of tree instances.

Figure 3.2 shows how two examples can be combined into a tree pattern using lgg. The common parts are retained, but the parts of the trees which differ between the two examples are replaced by variables, since that is the only result that can cover both arguments. Note that the corresponding subtrees of the two trees are combined in exactly the same way as the original tree patterns. The lgg algorithm therefore recursively descends down the tree. (However, multiple, identical pattern variables make this process more complicated as will be explained below.) More formally, the *lgg* operation is defined as follows.

Definition 3.3 *The* least general generalization *(lgg) of two tree patterns X and Y is the tree pattern U such that $L(U) \supseteq L(X), L(U) \supseteq L(Y)$ and for all V such that $L(V) \supseteq L(X) \cup L(Y)$, $L(V) \supseteq L(U)$.*

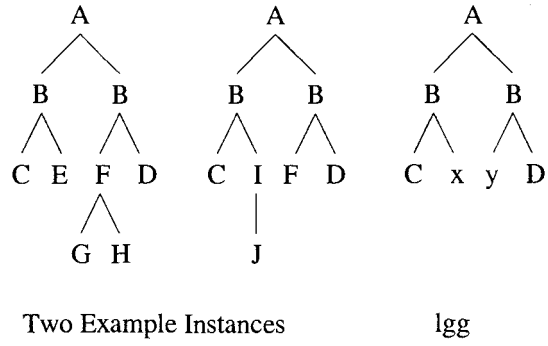


FIGURE 3.2. Example Constant Trees and lgg Pattern Tree

This definition implies that the lgg is matched by every possible V which matches both of the arguments given to the lgg algorithm (which therefore implies that the lgg is the most specific tree pattern satisfying the latter).

An algorithm to compute the lgg of two tree patterns must recursively examine the trees. The corresponding subtrees (i.e., ith subtree from left to right of each tree) are compared. If the corresponding subtrees have the same (internal) node label and number of children, then they are identical as far as this particular level of recursion is concerned. Hence a tree with the same label and number of subtrees is returned where each subtree is the result of the lgg of the corresponding subtrees of the two argument trees. But a pattern variable is returned to represent corresponding subtrees which are different, either in the root label or the number of subtrees.

These variables must be generated–i.e., a new variable name must be created for each such pair of different, corresponding subtrees. Further, whenever

Initialize cache = {} %= empty (before main/non-recursive call to ordlgg)

ordlgg($\mathcal{X}, \mathcal{Y}$) %given two tree patterns/instances
case
  $\mathcal{X}$ or $\mathcal{Y} = \phi$:
    return the other
  $\mathcal{X}$ and $\mathcal{Y}$ are both constant leaves and equal :
    return $\mathcal{X}$
  root node labels of $\mathcal{X}$ and $\mathcal{Y}$ differ,
      at least one of $\mathcal{X}$ and $\mathcal{Y}$ is a variable,
    or number of children differ:
    variablize($\mathcal{X}, \mathcal{Y}$) % returns cached
      %variable for $\mathcal{X}, \mathcal{Y}$, if any.
      % or creates a new variable and caches it.
  default :
    return a tree with $\mathcal{X}$'s root node label at top and
      the n children:
    $\langle$ ordlgg($\mathcal{X}_1, \mathcal{Y}_1$), ... ,
      ordlgg($\mathcal{X}_n, \mathcal{Y}_n$)$\rangle$
    % where $\mathcal{X}_i$ is the ith subtree of $\mathcal{X}$.

variablize(r, s) %Given two (sub)tree patterns/instances
if there is a triple of the form (r,s,var) in cache,
  then return var
  else create a new variable var,
    add (r,s,var) to cache,
    return var

FIGURE 3.3. Algorithm ordlgg for computing the *lgg* of ordered tree patterns

the same, identical pair of subtrees occurs more than once in the tree patterns, the same variable must be returned. This restriction is needed so the resulting output tree pattern will not be more general than necessary. The learning algorithm depends on having the least general hypothesis that will cover the training examples. So the lgg algorithm must not return an overgeneral hypothesis. The algorithm remembers when the pair of subtrees reoccurs by caching the subtree pairs along with the corresponding generated variable. Whenever the same pair of arguments with different root labels or different numbers of children is given to *lgg* more than once in generalizing a pair of trees, the same (repeated) variable is returned.

A proof that the ordlgg algorithm produces a result which satisfies the lgg definition follows. Induction on the minimum depths of the two arguments of ordlgg is used. (Where the root node is taken to be depth 0, and the depth of a tree is defined to be the maximum length of the path from the root to any leaf.)

Lemma 3.3 ordlgg $(\mathcal{X},\mathcal{Y}) \succeq \mathcal{X}$ *and* ordlgg $(\mathcal{X},\mathcal{Y}) \succeq \mathcal{Y}$.

Proof: Let k be the minimum of the depths of the two argument trees and the inductive hypothesis be: given $\mathcal{X}$ and $\mathcal{Y}$ with depth $k \leq n$, ordlgg $(\mathcal{X},\mathcal{Y}) \succeq \mathcal{X}$ and ordlgg $(\mathcal{X},\mathcal{Y}) \succeq \mathcal{Y}$. To prove the lemma's conclusion, substitutions σ_x and σ_y are defined so ordlgg $(\mathcal{X},\mathcal{Y})\,\sigma_x = \mathcal{X}$ and ordlgg $(\mathcal{X},\mathcal{Y})\,\sigma_y = \mathcal{Y}$.

Base Cases: $n = 0$: The following cases apply.

(1) The bottom/empty-set symbol (ϕ) is handled as a separate case. If $\mathcal{X}$ is ϕ, then ordlgg returns $\mathcal{Y}$ which is $\succeq \mathcal{Y}$ as well as $\mathcal{X}$. The same argument applies if $\mathcal{Y} = \phi$.

(2) $\mathcal{X}$= $\mathcal{Y}$= the same constant leaf C. Then ordlgg$(\mathcal{X},\mathcal{Y}) = C$ and $\sigma_x = \sigma_y = \{\}$.

Induction Cases: $n > 0$: (3) either $\mathcal{X}$ or $\mathcal{Y}$ is just a single variable, or $\mathcal{X} = \mathcal{X}_0(\mathcal{X}_1 \ldots \mathcal{X}_m)$ and $\mathcal{Y} = \mathcal{Y}_0(\mathcal{Y}_1 \ldots \mathcal{Y}_l)$ with $\mathcal{X}_0 \neq \mathcal{Y}_0$ or $m \neq l$ and no repeated variables. (if $\mathcal{X}$ is a variable, then $\mathcal{X}_0$ is that variable and $m = 0$; similarly for $\mathcal{Y}$.) ordlgg returns a variable v; $\sigma_x = \{v/\mathcal{X}\}$ and $\sigma_y = \{v/\mathcal{Y}\}$ shows that v

matches both lgg arguments by Definition 3.1.

(4) $\mathcal{X}_0 = \mathcal{Y}_0$ and $m = l$ (otherwise same as (3)–with no repeated variables). ordlgg is called with each pair of corresponding subtrees. By the inductive hypothesis for tree depth $n-1$, ordlgg computes the lgg of each each $\mathcal{X}_i$ and $\mathcal{Y}_i$ pair. By Definition 3.1, there are substitutions σ_{xi} and σ_{yi} so ordlgg$(\mathcal{X}_i, \mathcal{Y}_i)\sigma_{xi}$ $= \mathcal{X}_i$, and similarly for σ_{yi}. Since the σ_{xi} have no variables in common, $\sigma_x = \cup_i$ σ_{xi}.

Repeated Variables: (5) (in cases 3 and 4). Denote all pairs of subtrees combined by cases (3) or (4) as p_i and q_i and the corresponding variables as v_i. Then ordlgg produces $v_i = v_j$ iff $p_i = p_j$ and $q_i = q_j$. Let substitutions σ_x $= \{\ldots v_i/p_i \ldots\}$ and $\sigma_y = \{\ldots v_i/q_i \ldots\}$ (with duplicate variable substitutions eliminated). Then $\forall_i\ v_i\sigma_x = p_i$ and $v_i\sigma_y = q_i$, so ordlgg $(\mathcal{X},\mathcal{Y}) \succeq \mathcal{X}$, $\mathcal{Y}$. □

Lemma 3.4 *If* $\mathcal{Z} \succeq \mathcal{X}$ *and* $\mathcal{Z} \succeq \mathcal{Y}$ *then* $\mathcal{Z} \succeq$ ordlgg$(\mathcal{X}, \mathcal{Y})$.

Proof: Following Lemma 3.3, let k be the minimum depth of the two arguments and the inductive hypothesis be: $\forall\ \mathcal{Z} \succeq \mathcal{X}, \mathcal{Y} \Rightarrow \mathcal{Z} \succeq$ ordlgg $(\mathcal{X},\mathcal{Y})$.

Base Cases: $n = 0$:

(1) $\mathcal{X} = \phi$. Then any $\mathcal{Z} \succeq \mathcal{X}$, $\mathcal{Y}$ also implies $\mathcal{Z} \succeq$ ordlgg $(\mathcal{X},\mathcal{Y})$ since the latter is just $\mathcal{Y}$. The equivalent argument applies if $\mathcal{Y} = \phi$.

(2) $\mathcal{X} = \mathcal{Y}$ = the same constant leaf C. Then $\mathcal{Z} \succeq \mathcal{X},\mathcal{Y}$ implies $\mathcal{Z} \succeq C$, and ordlgg gives C also implies $\mathcal{Z} \succeq$ ordlgg $(\mathcal{X},\mathcal{Y})$.

Induction Cases: $n > 0$: (3) $\mathcal{X}$ or $\mathcal{Y}$ is just a variable s, or $\mathcal{X} = \mathcal{X}_0(\mathcal{X}_1 \ldots \mathcal{X}_m)$ and $\mathcal{Y} = \mathcal{Y}_0(\mathcal{Y}_1 \ldots \mathcal{Y}_l)$ with $\mathcal{X}_0 \neq \mathcal{Y}_0$ or $m \neq l$ and no repeated variables in ordlgg $(\mathcal{X},\mathcal{Y})$ or $\mathcal{Z}$. Then $\mathcal{Z} \succeq \mathcal{X},\mathcal{Y}$ and $\mathcal{Z}$ is a tree rather than a variable implies any substitutions σ_x and σ_y can not change the root label and number of subtrees of $\mathcal{Z}$. But $\mathcal{X}$ and $\mathcal{Y}$ differ in one of these characteristics, yielding a contradiction. $\mathcal{Z}$ can only be a variable so $\mathcal{Z} \succeq$ ordlgg $(\mathcal{X},\mathcal{Y})$.

(4) $\mathcal{X}_0 = \mathcal{Y}_0$ and $m = l$ (and the other conditions same as (3)–including no repeated variables). $\mathcal{Z} \succeq \mathcal{X},\mathcal{Y}$ implies $\forall_i \mathcal{Z}_i \succeq \mathcal{X}_i, \mathcal{Y}_i$. By the induction hypothesis for $n-1$, $\mathcal{Z}_i \succeq$ ordlgg $(\mathcal{X}_i, \mathcal{Y}_i)$. By Definition 3.1, there is a substitution σ_i

corresponding to this match. Given that there are no repeated variables in $\mathcal{Z}$ or ordlgg $(\mathcal{X},\mathcal{Y})$, form a substitution $\sigma = \cup_i \sigma_i$ to show $\mathcal{Z} \succeq$ ordlgg $(\mathcal{X},\mathcal{Y})$.
(5) Repeated variables in $\mathcal{Z}$ or ordlgg $(\mathcal{X},\mathcal{Y})$. Adapt the arguments in cases (3) and (4), but merge the substitutions as follows. Denote the variables in $\mathcal{Z}$ as $z_1, \ldots, z_j, \ldots$. Then by $\mathcal{Z} \succeq \mathcal{X},\mathcal{Y}$ and the corresponding substitutions σ_x and σ_y, there must be subtrees at the positions corresponding to these variables in $\mathcal{X}$ and $\mathcal{Y}$ (i.e., same depth and child of child ...). Denote them as $x_1, \ldots, x_j, \ldots$and $y_1, \ldots , y_j, \ldots$. Let the corresponding subtrees in ordlgg $(\mathcal{X},\mathcal{Y})$ be $o_1, \ldots, o_j, \ldots$(which must exist by the arguments of cases (3) and (4)). Suppose variables z_j and z_l in first-level subtrees $\mathcal{Z}_f$ and $\mathcal{Z}_g$, respectively, of $\mathcal{Z}$ are the same variable–call it z. Let x_j, x_l, y_j, y_l, o_j, and o_l be the subtrees corresponding to z_j and z_l in $\mathcal{X}$, $\mathcal{Y}$, and ordlgg $(\mathcal{X},\mathcal{Y})$. ordlgg therefore forms o_j from x_j and y_j in a deterministic manner except for choosing the name of a new variable (similarly for o_l). But subroutine **variablize** ensures that the same variable name will be returned for the same argument trees; hence $o_j = o_l$. The substitutions for the subtrees are $z_j\sigma_j =$ ordlgg$(\mathcal{X},\mathcal{Y})_j$ and $z_l\sigma_l =$ ordlgg$(\mathcal{X},\mathcal{Y})_l$. The substitutions for first-level subtrees $\mathcal{Z}_f$ and $\mathcal{Z}_g$ must be of the form, $\sigma_f = \{\ldots, z/o_j, \ldots\}$ and $\sigma_g = \{\ldots, z/o_l, \ldots\}$. But the substitutions for z are really the same, so σ_f and σ_g can be merged by taking the union of the substitutions and eliminating duplicates. This approach applies to all of $\mathcal{Z}$, so $\mathcal{Z} \succeq$ ordlgg $(\mathcal{X},\mathcal{Y})$. □

Theorem 3.5 *The algorithm* ordlgg *(Figure 3.3) computes the lgg of two ordered trees (Definition 3.3) in polynomial time.*

Proof: By Lemma 3.3, ordlgg $(\mathcal{X},\mathcal{Y}) \succeq \mathcal{X}, \mathcal{Y}$. By Lemma 3.4, $\forall\ \mathcal{Z} \succeq \mathcal{X}, \mathcal{Y}, \mathcal{Z} \succeq$ ordlgg $(\mathcal{X},\mathcal{Y})$. Hence, by Definition 3.3, ordlgg $(\mathcal{X},\mathcal{Y}) =$ lgg$(\mathcal{X},\mathcal{Y})$.

The time complexity of the lgg algorithm is bounded by the need to examine each node of the input trees. This time is therefore O(total input tree nodes). □

3.3. Ordered Tree Algorithm and Its Justification

The learning algorithm for ordered trees uses the equivalence query (EQ) to obtain training examples (tree instances). This algorithm maintains a hypothesis tree representing the examples it has seen so far and successively combines this hypothesis with new examples using *lgg* until EQ indicates that the target has been found. The properties of the ordered-tree class and lgg guarantee that this algorithm will successfully terminate after a polynomial number of examples (i.e., the chain of generalization steps can't be exponentially long).

The learning algorithm works as follows. The initialization sets the tentative hypothesis h to ϕ which represents the empty set of tree instances. This value ensures that the hypothesis is a subset of any possible target. If the target was the empty set, then learning would be done and the while loop terminates (since EQ immediately returns true), returning ϕ as the correct target.

The learner then executes the loop. EQ is used at the beginning of the loop to serve two purposes: to supply examples and to test when the loop is done. The loop test determines if the learner's hypothesis h is equivalent to the target and if so terminates the loop. The way the algorithm is structured guarantees both that h is always a subset of the target and that this terminating condition will eventually become true.

While the loop is not done, EQ supplies an example tree instance which is covered by the target but not by h. This example therefore suggests how the hypothesis h should be generalized to bring it closer to the target. The lgg algorithm then combines the new example with the previous hypothesis h to form a new hypothesis which covers both the hypothesis and the new example and hence all examples seen so far. This routine retains all those parts of the example and hypothesis which could be part of some target consistent with the examples already obtained.

The new hypothesis formed by lgg is then stored as the current hypothesis h. Because it covers an example not covered by the previous h, this new h

is always guaranteed to be more general. Further, the amount of generalization will always be significant (as measured by a *metric* to be explained later).

This hypothesis will always be a subset of the target. The initial hypothesis (ϕ) represents a subset of all possible targets. The dual invariant of all examples being positive and the hypothesis being a subset of the target is then preserved in all subsequent learning steps (Lemma 3.6). The resulting h will therefore never become more general than the target (i.e., never "overshoot" or *overgeneralize* the target). These facts together guarantee that the target will be found within a limited number of steps.

For example, Figure 3.4 shows several aspects of this learning. The sample target is given in (a), and the first two examples are (b) and (c). Note that learning process after that point alternates between calculating the lgg and getting an additional example–until the target is obtained. The lgg is first calculated from the initial hypothesis (ϕ, not shown) and the first example (b), and the result is the same as (b). Then taking the lgg with (c) yields (d), taking lgg with (e) yields (f), etc.

Note that examples (b) and (c) require generalization in different parts of the tree. When these two examples are combined into a single tree pattern by lgg (d), the result covers more than just the union of the sets of instances covered by (b) and (c). Example (c) is deeper in the first subtree than example (b), but example (b) is more general in the second and third subtrees. A tree pattern which covers both examples must necessarily be at least as general as either in the first child, and the only way to accomplish that is to make the first child a variable (t) in the lgg (d). The second (and third) subtrees have root C and two children with the left child being D so that part is retained in the lgg. But the right child is quite different so it is replaced with a variable (s) in (d). The target necessarily includes all the instances covered by (d) because the target has to cover both lgg arguments–(b) and (c) and there is no more specific tree within the ordered tree class which can cover both of these arguments because

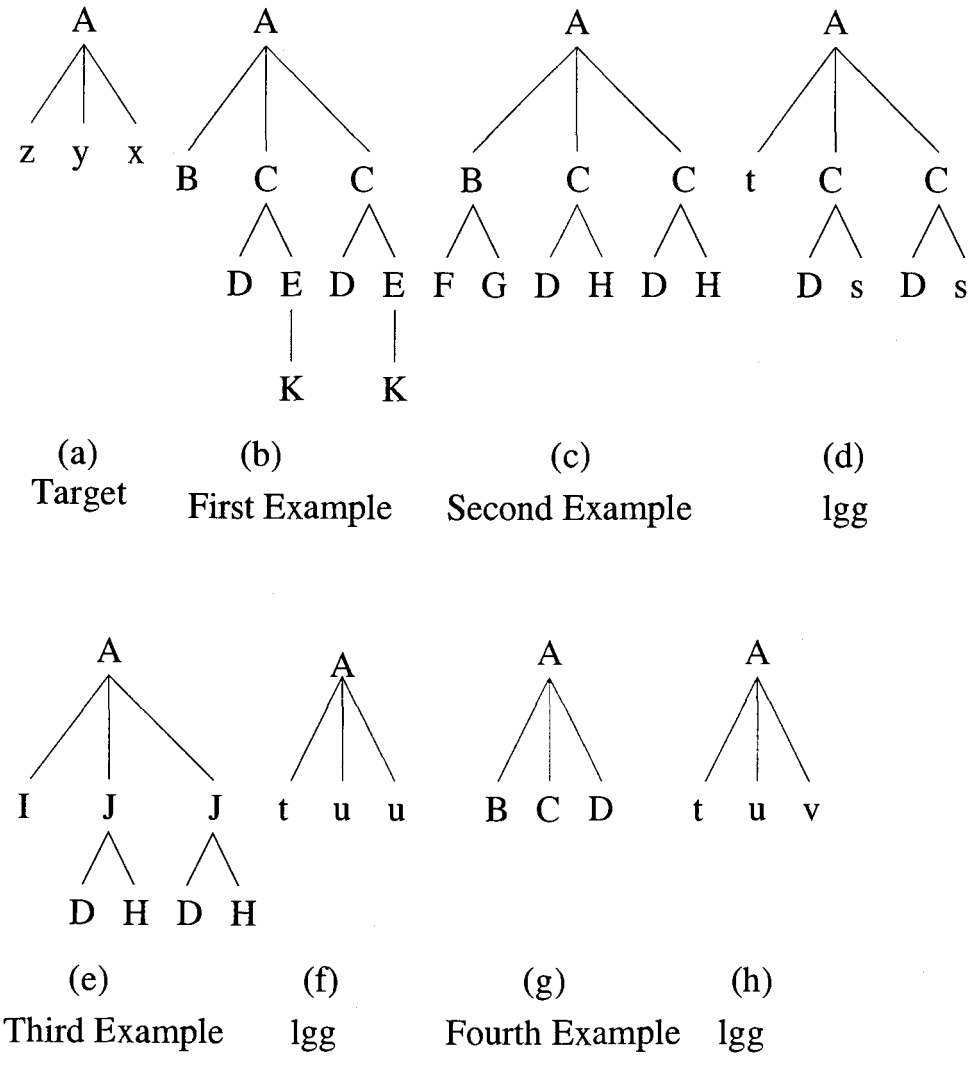


FIGURE 3.4. Ordered Tree Learning Example

any tree pattern that is more specific in one of the child subtrees won't cover both of the corresponding subtrees in examples (b) and (c).

There are some more subtle issues involving the presence of more than one copy of the same variable in a tree pattern. When lgg forms (d), it also detects the fact that the same pair of subsubtrees is combined by lgg twice– $E(K)$ and H–and therefore must be given the same variable name s. Tree (f) is a generalization of (d) which keeps the variables identical but at the next, higher level. Tree pattern (h) separates these repeated variables into two distinct variables. This result is equivalent to the target, so EQ indicates that learning is done.

$h = \phi$ % initialize hypothesis to empty set
while $EQ(h)$ yields a counterexample x %each counterexample is in
% the target set.
$h = lgg(h, x)$
return h

FIGURE 3.5. OT Learning Algorithm Using EQ only

Lemma 3.6 *In the ordered tree algorithm (Figure 3.5), the training examples returned by EQ are always positive and the hypothesis h is always a subset of the target T.*

Proof: The two points to be proven mutually depend on each other, and are therefore proven together. Use mathematical induction on the learning sequence with the inductive hypothesis being $L(h) \subseteq L(T)$.

Base case: h is initialized to ϕ and clearly $\mathrm{L}(\phi) \subseteq T$.

Induction step: Assume $\mathrm{L}(h) \subseteq \mathrm{L}(T)$ when EQ is called. Then any counterexample x supplied by EQ is in the set difference $L(T) - L(h)$ since $L(h) \subseteq L(T) \Rightarrow L(h) - L(T) = \phi$. Therefore both h and the positive counterexample x are contained in T. Let $l = \mathrm{lgg}(h,x)$. By definition, l must be matched by any tree pattern that matches both h and x, so $L(l) \subseteq L(T)$. Since l becomes the new h, the inductive hypothesis is true the next time EQ is called. □

Theorem 3.7 *Ordered trees are exactly learnable from EQ in time polynomial in the size of the initial instance and target.*

Proof: By Lemma 3.6, the hypothesis is always a subset of the target. On each loop iteration, the new example must be outside of the old h but in the new h. This guarantees that h is made more general in each step and makes progress toward the target. The following argument bounds the number of iterations needed to converge to the target.

The lgg must result in a change in the learned hypothesis in one of the following ways:

1. Turning a constant label on a leaf node into a variable
2. Trimming the children/subtrees below a node and turning that node into a variable
3. Making two identical variables distinct (or, more generally, splitting a set of identical variables into two sets with each such set having a distinct variable

Let n be the number of nodes in the hypothesis tree pattern and v be the number of *distinct variables* in that tree. Then it is easy to see that in each of the above cases this tree-size *metric* $n - v$ decreases by at least 1. Since initially

n equals the number of nodes in the first counterexample and v is 0, the total number of queries, including the first EQ query on the null pattern, is bounded by $n_0 + 1$, where n_0 is the maximum number of nodes in any counterexample.

Recall that the *lgg* algorithm runs in time linear in the sizes (number of nodes) of its input trees. Other operations which manipulate the trees as a whole obey the same bound. The time for each loop iteration is therefore bounded by $O(n_0)$. The time complexity for the OF algorithm is therefore $O(n_0^2)$. □

The OT result can be extended to a concept class in which the label alphabet is finite and there is a bound on the number of subtrees allowed under any node in a tree. Using the notation of Section 4.2, this class is $OT_{l,b}$.

Corollary 3.8 *Ordered trees with both label alphabet and the number of subtrees bounded ($OT_{l,b}$ with l, b finite) are exactly learnable.*

Proof: No part of the ordered-tree or lgg algorithms or proofs depends on the ordered-tree class having an infinite label alphabet or lack of a bound on the number of subtrees. □

3.4. Ordered Forests

This section describes the *ordered forests* (OF) concept class and its learnability. OF allows hypotheses containing more than one tree pattern and therefore is more expressive in that it can represent sets of instances which the OT class (ordered trees) can't. As described in the ordered tree introduction, this class can represent sets of tree instances which do not all have the same structure, and hence are not representable by hypotheses in the OT class. This class requires an additional oracle in order to be learnable. We will use the subset query or the membership query. Ordered forests obey a special property called

compactness (Definition 3.5) which makes this class learnable from equivalence and subset queries.

Definition 3.4 *In the concept class of ordered forests (OF), each hypothesis is a finite set of tree patterns, where the subtrees of each tree pattern are ordered from left to right. Further, either the label alphabet is infinite or the number of children of any node is unbounded (or both).*

The OF algorithm (Figure 3.8) maintains a set of hypothesis trees. It obtains examples one at a time from EQ, and uses SQ (or MQ) to determine whether a particular example should be combined with a particular hypothesis tree pattern. Each target tree is therefore successively approximated as additional examples are obtained and combined with the appropriate hypothesis tree as judged by SQ. The learning process therefore works like multiple copies of the OT learning algorithm–in effect there are multiple "slots" where each slot is a hypothesis tree pattern, and SQ helps decide which slot an example should be placed in and combined with the corresponding hypothesis tree pattern. The else part in the code creates a new slot (hypothesis tree) when necessary.

This approach might seem to be sufficient to guarantee that the target can be efficiently found, but there is a problem. The correspondence between hypothesis trees and target trees is not a priori guaranteed and must be justified. A guarantee of this correspondence depends on the hypothesis space having the *compactness* property as defined below.

To prove that various learning algorithms work in polynomial time, there is a need to show that not too many potential tree patterns are created compared to the 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 concept, then one of the concepts in the union covers it by itself.

Definition 3.5 *A concept class $\mathcal{C}$ is* compact *iff for any $\mathcal{Z}$ and $\mathcal{V}_1, \ldots, \mathcal{V}_n \in \mathcal{C}$, $\bigcup_{i=1}^{n} L(\mathcal{V}_i) \supseteq L(\mathcal{Z}) \Rightarrow \exists i\ L(\mathcal{V}_i) \supseteq L(\mathcal{Z})$*

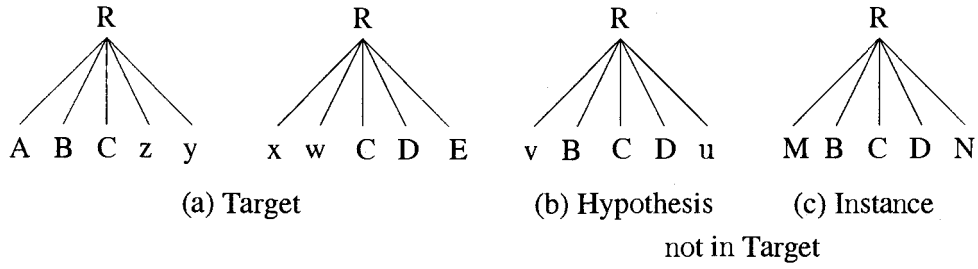


FIGURE 3.6. OF Learning Depends on Compactness

See Figure 3.6 for an example illustrating the compactness property. Let the two pattern trees in (a) be the target and let (b) be a hypothesis tree. Then (b) does not represent a subset of the left tree of (a) because it has a variable in place of A nor a subset of the right target tree because of the variable in place of E. Is it possible for a hypothesis tree such as (b) to cover examples outside of each of the target trees and yet be a subset of their union? At first glance, this might seem possible because (b) has a variable in place of one of the constants of each of the target trees and (b) contains only constant and variable children which also have an equivalent in one of the target trees. But a more careful analysis shows that the instance (c) is matched by (b) but not by either tree in (a), so the assumption is false.

With a more complex example, an OF learning algorithm might create so many hypotheses like (b) that it would have great difficulty finding the correct target trees. Can the OF algorithm fall into this trap? The compactness property for OF says no; i.e., if (b) is a subset of the target, then it must be a subset of an individual target tree in (a). Therefore SQ would reject (b) in the

OF algorithm so that this tree pattern would never be placed in h. Note that the OT class does not have this problem because all examples must be under the same target tree since the class restricts the target to consist of only one tree pattern. We now formally show that this property holds for the OT class.

Lemma 3.9 *The class of ordered tree patterns (OT) is compact.*

Proof: From the tree pattern $\mathcal{Z}$, form the tree instance e by leaving each constant in $\mathcal{Z}$ alone and substituting constant subtrees for variables as follows. Replace each distinct variable in $\mathcal{Z}$ with either a single constant label (or a 1-level constant subtree–root plus children). Use the same constant tree for each copy of a repeated variable. Choose these labels or subtrees so they are not matched by any nontrivial subtrees (other than a single variable) in any of the $\mathcal{V}_i$. Since either the label alphabet size or number of children is unbounded, there are always enough of these distinct subtrees or labels available.

Then for each i, either $e \notin \mathcal{V}_i$ or $\mathcal{V}_i \succeq \mathcal{Z}$ by the following argument. When a particular $\mathcal{V}_i$ matches e, by the above choices any constant subtree c in e is only matched by single variables in $\mathcal{V}_i$. Therefore c can be replaced by any constant subtree in e and $\mathcal{V}_i$ will still match e. This observation implies that any tree instance $\mathcal{Z}$ matches will also be matched by $\mathcal{V}_i$, so some $\mathcal{V}_i \succeq \mathcal{Z}$ or $\cup\mathcal{V}_i \not\succeq \mathcal{Z}$. □

It is also worth noting why the following simpler algorithm does not work for learning OF. The simpler algorithm takes each positive example obtained from the equivalence query and uses a series of subset queries to generalize it as much as possible until it cannot be generalized any more, in which case it is stored away as one of the tree patterns in the target. This version of the algorithm uses just one example per target tree to be learned and depends on the fact that the number of possible ways to generalize a hypothesis is limited to be polynomial. The algorithm is therefore informally described as "bottom-up learning from single examples". This approach works fine as long as there are no

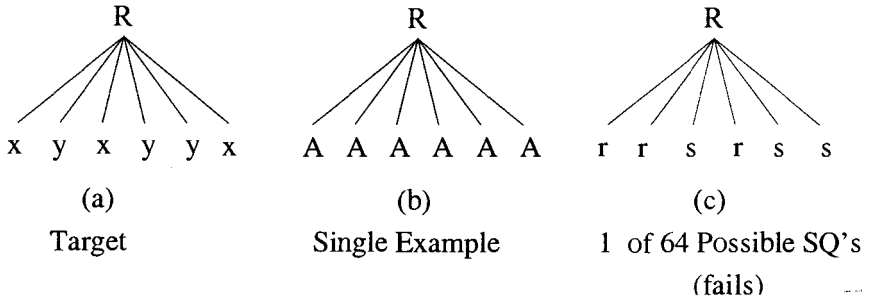


FIGURE 3.7. OF Bottom-Up from Single Example Not Polynomial

repeated variables in the tree and is indeed the algorithm we give for the $\mu-$UF class (Figure 6.1). If there are repeated variables, however, it is not possible to extract a tree pattern from a tree instance with a reasonable number of subset queries. Consider, for example, a tree pattern with $2n$ leaves, half of which are labeled with the variable x and the rest labeled with the variable y (Figure 3.7 (a)). Suppose the counterexample has all the $2n$ variables instantiated to the same constant A (part b of figure). Now, unless the algorithm correctly guesses which half of the $2n$ constants should be generalized to the same variable, any attempt to generalize a subset of the constants will get a negative answer from the subset query–e.g., (c) in the figure. The process of finding the correct subset of the variables would be like guessing a password. Exhaustive search for the correct generalization requires a number of queries exponential in n. (The number of choices to test would be close to 2^{2n} because there might not be the

same number of copies of the two variables and each child could independently be either variable.[1])

Our algorithm (Figure 3.8), instead, uses several examples to learn each tree pattern in the forest, and combines them using the *lgg* algorithm described earlier. It solves the above problem by letting the counterexamples determine which subset of the constants should be generalized to the same variable. The method is similar to the approaches used elsewhere for learning propositional and first order Horn clauses (Angluin, Frazier, & Pitt, 1992; Reddy & Tadepalli, 1998, 1999).

initialization: $h = \{\}$
while EQ(h) gives counterexample x (which is positive):
  If $\exists$ a tree pattern p in h such that SQ(p')
    where $p' = lgg(p, x)$
    then replace the first such p with p'
    else $h = h \bigcup x$. % add another tree to hypothesis
Return(h)

FIGURE 3.8. OF-EQSQ algorithm

[1]The cases with all one variable are not generalizations and swapping the two variables makes no difference, so the actual number is a bit less. Also see the Chapter 5 on the non-learnability of UT for a discussion of additional issues related to this approach.

3.5. Learning Ordered Forests With Subset Queries

In this section, we describe and analyze a learning algorithm for OF for an infinite label alphabet or with an unbounded number of children using equivalence and subset queries (and from EQ and membership queries in the next section).

The algorithm for OF using EQ and SQ is given in Figure 3.8. This algorithm works much like the OT algorithm for each individual hypothesis tree being formed in h. But it uses the subset query to decide which examples should be combined into the same hypothesis tree. The algorithm keeps getting example tree instances and combining them into tree patterns while maintaining the tree patterns it has generated so far in h. The algorithm doesn't know which examples belong in the same target tree pattern, so it tries combining each example with every hypothesis tree in h in sequence using lgg. Each such result is then tested to see if it is a subset of the target and therefore that example really belongs in that tree pattern. If so, then the resulting tree pattern is kept in h. Otherwise that change is rejected and the next hypothesis tree is tried. If the new example can't be successfully combined with any tree pattern in h, it is added to h as a separate, new tree (which therefore matches only itself).

Lemma 3.10 below proves a property of the OF learning algorithm which is needed to support the OF-learnability proof (Theorem 3.11). This lemma applies the compactness property to the OF algorithm. The result shows that each tree pattern in h has to correspond to (is a subset of) a particular target tree and no two tree patterns correspond to the same target tree (i.e., there is a one-to-one into mapping from the tree patterns in h to the target tree patterns). This result is needed to show that the OF algorithm will not produce an overly complex hypothesis any more hypothesis with excess tree patterns and in turn implies that the learning algorithm will work within reasonable time and query complexity bounds.

Lemma 3.10 *Let the target forest for the OF algorithm be* $f = \{t_1 \ldots t_n\}$ *and let the hypothesis at some stage be* $h = \{p_1 \ldots p_m\}$. *Then: (1) for every* $p_j \in h$, *there is a* $t_i \in f$ *such that* $L(p_j) \subseteq L(t_i)$. *(2) No two distinct* p_j*'s in* h *are subsets of the same* t_i.

Proof: (1): Every p_j which is introduced by the algorithm is a subset of the target because the introduced p_j is either a positive counterexample x or was approved as a subset by the SQ call in Figure 3.8. Then by compactness of ordered trees, part (1) follows.
(2): Assume the contrary: $p_j, p_k \subseteq t_i$ for some $i \leq n$ and $j < k$ at some point in the algorithm. Let p'_j and p'_k be the j^{th} and the k^{th} patterns in the hypothesis immediately after the k^{th} pattern was first introduced. Since the OF algorithm monotonically generalizes the tree patterns in the hypothesis, all previous versions of p_j and p_k, including p'_j and p'_k, must be subsets of t_i. When p'_k was first introduced, it was just a new example, and $t_i \succeq lgg(p'_j, p'_k)$, by Definition 3.3 of *lgg*. Hence these two trees would have been combined into one in the hypothesis by the OF-EQSQ learning algorithm, leading to a contradiction. □

The proof of correctness of the algorithm relies on the fact that lgg and SQ work together in the OF algorithm to determine which examples should be combined into a tree pattern.

Theorem 3.11 *OF is learnable from equivalence and subset queries.*

Proof: The algorithm tries to combine each new example with each hypothesis tree accumulated so far using *lgg*, the result of which is verified with SQ. By compactness, the result of each such *lgg* step that passes the subset query must yield a tree pattern which is matched by one particular target tree pattern. Further, each such step must generalize the tree nontrivially, or else the new example would have been matched by the original tree pattern, and would not have been a counterexample.

For the bounds calculation, recall that Theorem 3.7 uses the *metric* $n-v$ to measure progress in generalizing a tree pattern (where n is the number of nodes and v is the number of *distinct* variables.) This same metric is used to calculate a similar bound for OF. But it is not sufficient to use the metric on just one example. Instead, each hypothesis tree which eventually becomes equivalent to one of the target trees is essentially independent of all the other hypothesis trees. Therefore not just the first example for the learning sequence but the first example used in forming *each* hypothesis tree must be used in calculating the bound.

Hence, for each target tree, the number of counterexamples is bounded by the number of nodes in the first example and one more for that initial example. The equivalence queries, or counterexamples for the entire target are therefore bounded by $t(m+1)$ where t is the number of target trees and m is the maximum size of any counterexample.

An attempt is made to generalize each counterexample with each hypothesis tree; by Lemma 3.10, the number of hypothesis trees is never more than the number of target trees. Therefore, the number of subset queries is bounded by t times the number of counterexamples or $t^2(m+1)$ or $O(t^2m)$. □

3.6. Learning Ordered Forests With Membership Queries

A subset query tests whether a given tree pattern matches a subset of the instances matched by the target. A *membership query* (*MQ*) merely tests if one particular instance is matched by the target. It might therefore seem that SQ is more powerful than MQ but that is not true in most classes. An MQ test accurately simulates an SQ test if all variables in the tree pattern given to SQ are replaced with unique constants (or constant subtrees) that do not occur anywhere in the target.

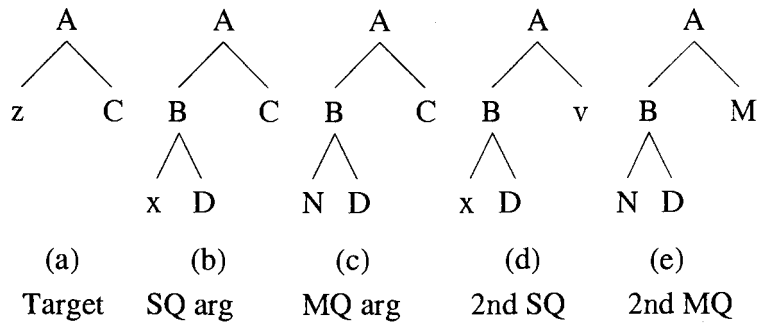


FIGURE 3.9. How MQ can simulate SQ

Figure 3.9 shows an example of membership queries simulating subset queries. Let the target be (a). If a subset query is executed with the tree pattern (b), the answer will be true because z can cover everything matched by the subtree at B. A membership query would simulate this operation by replacing x with a new constant N as in (c) and would get the correct answer.

But there is a problem: suppose it is desired to simulate a subset query that has pattern (d) as an argument–which would give *false* (for the target in (a)). Then substituting constants N and M for the two variables as in (e) would again give the correct answer. But the learner has no way of knowing which constants are in/not in the target and it might also try substituting C for the second variable, yielding the tree (c) and getting an incorrect answer. (Equivalently, the target could have M as the right child, giving the same result.) These examples show both how the simulation can work quite well in most cases but still be imperfect. Some error detection and recovery is therefore necessary to be able to learn with membership queries.

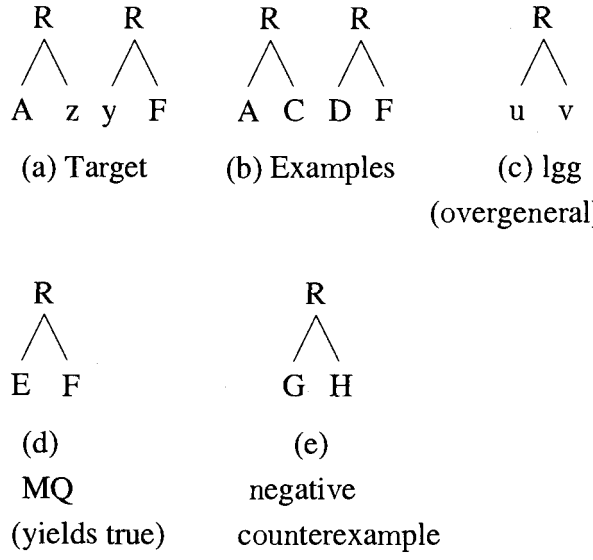


FIGURE 3.10. Learning With MQ

Figure 3.10 shows a possible learning sequence using MQ. The chosen target consists of the two trees in (a); then the examples in (b) are given to the learner. The learner forms the lgg with (c) which is overgeneral. Suppose the learner calls MQ with the constants E and F substituted for the variables in (c) as shown in (d); the MQ returns true because this instance is matched by the right tree pattern in the target and because one of the constants chosen (F) also appeared in the target. The learner therefore incorrectly concludes that (c) is a subset of the target, so the algorithm has overgeneralized. The malfunction is not detected until further learning is attempted by giving tree pattern (c) as hypothesis to EQ and obtaining a negative counterexample (e) (matched by the hypothesis but not by the target). The learner can therefore conclude that

some of the constants it used (either E or F) caused the malfunction but it doesn't know which. To fix the problem, the learner simply does not use any of those constants anymore but uses new ones (say G and H). Step (d) is repeated with G and H as the leaves in (e) and MQ returns the correct value, *false*. The learner must then make the union of the two examples (without any generalization–i.e., no variables) as the new hypothesis because there are no other examples available and therefore no guide on how to further generalize the examples at this point. The rest of the steps will work like the SQ. G and H can continue to be substituted for variables until another overgeneral hypothesis is detected or learning is done.

The MQ algorithm (Figure 3.11) is given with a version that works with either an infinite constant alphabet or an unbounded number of children under any tree node. To simulate a subset query on a tree pattern with v variables, when the label alphabet is infinite, the algorithm tries to find v consecutive unused labels with sequential search. When the number of children of a node is not bounded, the algorithm searches for v distinct 1-level tree instances with a root and large enough numbers of children, $w, w+1, \ldots$, so that they do not occur as subtrees of the target tree patterns. The search is done by doubling w each time an equivalence query gives a negative example suggesting that one of such tree instances occurs in the target.

The theorem below depends on the results above that learning OF with SQ work as well for MQ–by using the error correction process. When the simulation of SQ by MQ works perfectly, the MQ version of the algorithm works exactly like the SQ version–including the bounds on the number of queries. But if an overgeneralization occurs, it will always be detected by EQ returning a negative example within a time bound no greater than that for the SQ algorithm to learn the target. The MQ algorithm then restarts learning with a new sequence of constants. Since only constants in the target can cause such failures, the number of restarts is bounded by the number of constant labels in

```
c = test constant for MQ simulate SQ):
   = first label if node label alphabet is infinite, or
   = 1-level tree instance with number of children w = 1
         if unbounded
repeat   % use new constants until MQ
         % correctly simulates SQ:
   c = next label if infinite labels or
     = tree with number of children doubled if unbounded.
       % The following is essentially the OF-EQSQ algorithm
         % with constants in place of variables:
   initialize hypothesis: h = {}
   while EQ(h) gives a positive counterexample x:
      If ∃ a tree pattern p in h such that
            MQ(lgg(p, x) with variables replaced by
               c and its successors)
            returns "true"
         then replace p with lgg(p, x)
         else h = h ⋃ x.
until no negative counterexample is returned by EQ.
Return(h)
```

FIGURE 3.11. OF algorithm using EQ and MQ

the target; and after all these constants are exhausted, no overgeneralization can occur. This factor multiplied by the bounds for the SQ algorithm gives corresponding bounds for the MQ algorithm.

Theorem 3.12 *OF (with either an infinite label alphabet or an unbounded number of children) is learnable from equivalence and membership queries.*

Proof: The algorithm is given in Figure 3.11. First we consider the case where there is no bound on the maximum number of children of any node in

the tree patterns. The subset query of the OF-EQSQ algorithm is simulated by a membership query where the variables are substituted with distinct tree instances c and its successors, which have a root with a number of children $w, w+1, \ldots$, where the "width" w is initialized to 1. If MQ answers *false*, then SQ would have answered *false* as well and the algorithm works correctly. But if MQ answers *true*, it may be falsely interpreted as a *true* for SQ if one of the tree patterns in the target forest contains c as a subtree. In that case, the algorithm overgeneralizes. This will be detected when there is a negative counterexample to EQ, then the algorithm doubles the width w and starts all over. If the maximum number of children of any node in the target is b, then in $\lceil \log b \rceil$ iterations, it will reach a w which is larger than b and from then on MQ faithfully simulates SQ. Hence all the bounds for the OF-EQSQ will have to be multiplied by $\log b$ to get the bounds for the new algorithm.

When there is no bound on the number of possible node labels, the algorithm substitutes each variable with a distinct new label generated successively starting with c in Figure 3.11. If the number of constant labels in the target forest is l, it is easy to see that there are l consecutive unused labels in the first $l^2 + l$ labels. So, in this case, the multiplication factor for the time and query complexity bounds is $l^2 + l$. □

3.7. Summary and Discussion

Ordered trees were proven to be learnable from from equivalence queries alone. Ordered forests are learnable from equivalence and either subset or membership queries. An algorithm for combining multiple examples (*lgg*) is an important tool for both of these learning algorithms.

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 et al., 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 et al., 1995; Amoth, Cull, & Tadepalli, 1998).

In related work on ordered forests, Arimura considers a slightly different learning setting in which each symbol in the alphabet has an associated arity (the number of children allowed for a node labeled by it) and gives a learning algorithm for OF with EQ and MQ, where the size of the label alphabet is greater than the number of trees in the target forest (Arimura et al., 1995). Their algorithm assumes a bound on the number of trees in the target forest, while our algorithm uses negative examples to determine this number.

Ordered forests with a bound on the number of trees have been shown to be learnable with equivalence queries alone (Goldman & Kwek, 1999); this claim gives a time bound which is exponential in the number of tress in the forest.

4. REDUCTIONS BETWEEN ORDERED FORESTS AND DNF

This chapter ties the learnability of some tree classes to learnability of Boolean formulas in *disjunctive normal form* (*DNF*). DNF has been proven to not be exactly learnable with equivalence queries only (Angluin, 1990). *PAC* is a learning framework in which the learner is allowed to return a hypothesis which approximates the target within some *error rate parameter*, ϵ. But PAC learnability of DNF is an open question. Unlike PAC, the *PAC predictability* framework does not require the learner to return a hypothesis, but merely requires the class of examples the learner has not seen before to be predicted with an error rate no greater than a parameter ϵ. Membership queries have been shown to not be helpful in learning DNF in the PAC predictability framework by (Angluin & Kharitnov, 1995), based on the assumption that cryptographic systems such as the RSA cryptosystem are hard to break. The PAC (or exact) learnability of DNF with (equivalence and) membership queries is still not known.

On the assumption the above classes are hard, this chapter concentrates on negative results. When both the label alphabet and a bound on the number of children of any node are finite, this chapter shows that exactly learning ordered forests without repeated variables (μ-OF) is at least as hard as learning DNF–with both equivalence and membership queries. With equivalence queries alone, several classes are proven to not be learnable without relying on any complexity assumptions. These classes include OF, DNF for any base, and even μ-UT (Section 5.2) (However, the simplest class–ordered trees–is learnable under these conditions–Corollary 3.8.)

Under the *prediction with membership queries* (*pwm*) framework, DNF in base k is shown to be reducible to ordinary base-2 DNF, and DNF (in an appropriate base) is shown to be reducible to OF with repeated variables–even

though these reductions (with membership queries) are open problems in the exact framework.

4.1. PAC Prediction

PAC prediction is a probabilistic learning framework which, unlike *probably approximately correct* (*PAC*), does not require a hypothesis to be returned (Valiant, 1984a; Natarajan, 1991; Kearns & Vazirani, 1994; Pitt & Warmuth, 1990). The prediction framework merely requires the learner to be able to predict the class of test instances. *Prediction-preserving reducibility* (*PPR*) (Pitt & Warmuth, 1990) and *prediction with membership queries* (*pwm*) (Angluin & Kharitnov, 1995) are learning reduction techniques with explicit criteria given in the literature. The *PAC* or *Probably Approximately Correct* learning framework is probabilistic in that it requires a hypothesis with a small error rate to be returned at the end of learning with high probability. The examples during training and testing are to be chosen from the same stationary distribution unknown to the learner. PPR is a reduction technique in the PAC-predictability framework which has no queries in addition to that which provides random examples. But *pwm* is a criterion for reduction with membership queries in the predictability framework.

Each reduction definition requires conversion functions between the two classes for the target and for examples and hypotheses between the two classes as needed by the learning framework and queries used. Consistency requirements between hypotheses and example conversions then ensure that the queries for the two classes in the reduction behave properly. Each concept class is actually a class of representations (hypotheses) $\mathcal{R}$ with a corresponding implicit mapping L from each representation $r \in \mathcal{R}$ to a subset of the instance space $\mathcal{I}$. A reduction consists of mappings of instances and representations with certain properties.

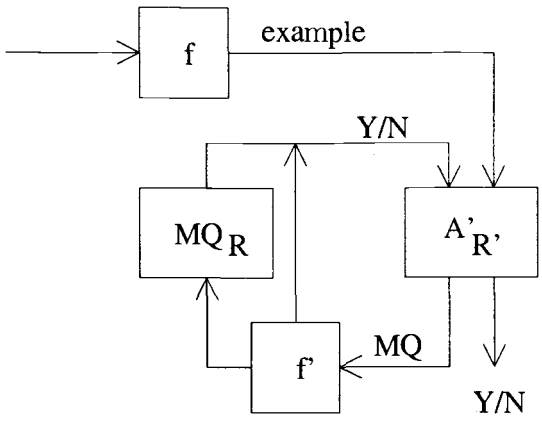


FIGURE 4.1. Simulation for pwm Reduction

Definition 1 of (Angluin & Kharitnov, 1995) is an explicit criterion for reduction between concept classes in the *prediction with membership queries* (*pwm*) framework. This reduction of $\mathcal{R}$ to $\mathcal{R}'$ is denoted $\mathcal{R} \leq_{pwm} \mathcal{R}'$and implies $\mathcal{R}'$ is at least as hard to learn as $\mathcal{R}$ is, or learnability of $\mathcal{R}'$ in the PAC-predictability framework with membership queries implies the same for $\mathcal{R}$. Define $\mathcal{I}$ and $\mathcal{I}'$ to be the corresponding instance spaces.

Figure 4.1 diagrams the simulation used by the reduction. Function f converts and defines the correspondence from examples in $\mathcal{I}$ provided to the learner to examples in $\mathcal{I}'$ which are then given to learning algorithm $\mathcal{A}'$ for $\mathcal{R}'$. Function f' does the reverse (but not necessarily inverse) conversion of example arguments in $\mathcal{I}'$ given to MQ by $\mathcal{A}'$ to examples in $\mathcal{I}$ which can then be given to MQ for $\mathcal{R}$ as shown in the figure. This function f can return $\top$ or $\bot$ which represent "always true" and "always false", respectively. These special values are then translated directly to *yes* or *no* without calling MQ_R.

There is no requirement for an *explicit* conversion of hypotheses in either direction in prediction reductions, so none is shown in Figure 4.1. But a mapping g is implied by the requirement that for each hypothesis $r \in \mathcal{R}$, there exists a corresponding hypothesis $g(r) \in \mathcal{R}'$. All of these functions are required to produce a result with polynomial size and be computable in polynomial time, except the definition does not require g to be computable in polynomial time. PAC-predictability differs from the exact learning model both in that it is probabilistic and it does not require a hypothesis to be returned. Hence there is no need to convert a hypothesis to the class being learned. Following (Angluin & Kharitnov, 1995), here is a version of the *pwm* definition simplified by removing the explicit polynomial-size arguments:

Definition 4.1 *Let $\mathcal{R}$ and $\mathcal{R}'$ be representations of concepts with corresponding instance spaces $\mathcal{I}$ and $\mathcal{I}'$. Let $\top$ and $\bot$ be elements not in $\mathcal{I}$. Then $\mathcal{R}$ is* pwm-reducible *to $\mathcal{R}'$, denoted $\mathcal{R} \leq_{pwm} \mathcal{R}'$, if and only if there exist three mappings $f: \mathcal{I} \to \mathcal{I}'$, $g: \mathcal{R} \to \mathcal{R}'$, and $f': \mathcal{I}' \to \mathcal{I}$ with the following properties:*

1. *for $r \in \mathcal{R}$, $g(r)$ is a hypothesis r' of size polynomial in the size of r.*
2. *$\forall\, r \in \mathcal{R}$, $w \in \mathcal{I}$, $f(w) \in \mathcal{I}'$ and $w \in L(r) \Leftrightarrow f(w) \in L(g(r))$. Moreover f is computable in time polynomial in the sizes of w and r*
3. *For every hypothesis $r \in \mathcal{R}$ and every $w' \in \mathcal{I}'$, $f'(w')$ is either $\top$, $\bot$, or $f'(w') \in L(r) \Leftrightarrow w' \in L(g(r))$. If $f'(w') = \top$, then $w' \in L(g(r))$; if $f'(w') = \bot$, then $w' \notin L(g(r))$. Moreover f' is computable in time polynomial in the sizes of r and w'.*

This definition essentially says g converts hypotheses in a way consistent with f and f' for converting examples in both directions, but a little bit of flexibility is allowed by using $\top$ and $\bot$ which act like special instances even though they are not in $\mathcal{I}$. The definition for *PPR reduction*, denoted here by

$\leq_{PPR}$, (Pitt & Warmuth, 1990) is identical to the above except for the absence of any h function to convert the instance arguments of *membership queries*).

The following concept class is used as an intermediate step between DNF and μ-OF$_{l,b}$:

Definition 4.2 *The concept class k-value DNF consists of disjunctions of conjunctions of statements of the form $v = x$, where v is a variable and $0 \leq x < k$. k can be infinite in which case x can be any natural number.*

Constructs associated with this class will be designated with the k-prefix; e.g., k-target, k-variable, etc.

This class generalizes the usual DNF in which a variable x can be taken to mean $x = 1$ and a negated variable $\neg x$ can be taken to mean $x = 0$. But *explicit negation* (e.g., a literal of the form, $x \neq 2$) is not allowed. In 3- value DNF, $x \neq 2$ can be represented as $x = 0 \vee x = 1$. But the size of such expressions such as $x_1 \neq 2 \wedge x_2 \neq 1 \ldots x_n \neq 1$ will grow exponentially in the number of variables when translated to DNF.

DNF with explicit negation which permits expressions such as $x \neq 2 \wedge x \neq 3$ would allow conjunctions of arbitrary subsets of the set of possible values for each variable–with a polynomial-sized expression. This variation of the DNF class can therefore convert hypotheses between bases while directly representing the corresponding set of instances. However, explicit negation is not easily translated to and from the OF domain. Hence we only consider k- value DNF without negation.

The following definitions for concept classes are used in this chapter.

Definition 4.3 *OF$_{l,b}$ is the concept class of ordered forests with a label alphabet of l symbols and at most ("branching factor") b children at each node, where l and b are assumed to be finite unless otherwise specified.*

OF$_{\infty,\infty}$ is identical to OF in Section 3.5, and OF$_{\infty,b}$ and OF$_{l,\infty}$ have identical learnability. These notations will be used when there is a need to

distinguish classes corresponding to particular values of l and b. For these class to be meaningful, b must be ≥ 2, and generally l needs to be ≥ 2 as well.

By analogy with the notation for read-once Boolean formulas:

Definition 4.4 *The prefix μ- in front of any tree or forest patterns restricts that class to hypotheses* without repeated variables. *So μ-*UT *means unordered trees without repeated variables.*

The following lemma and proof support the claim that, as with DNF in (Angluin & Kharitnov, 1995), MQ does not help learning DNF with k values. This lemma and Angluin's version are essentially reductions between sets of queries rather than between concept classes. In this case the reduction is from pwm (prediction with membership queries) to PAC-predictability (prediction without membership queries).

Lemma 4.1 *Under widely believed cryptographic assumptions, if k-value DNF is learnable with membership queries in the PAC-predictability framework, then it is learnable without membership queries in the same framework.*

Proof (Sketch): (Angluin & Kharitnov, 1995) showed under widely believed cryptographic assumptions, if DNF can be learned (within the *PAC predictability* framework) with membership queries, then it can be learned without membership queries. The proof ties the problem of cracking the cryptography systems mentioned in Angluin's paper to learning. Further, the proof encodes the logic and arithmetic of the cryptography systems in DNF by representing implications between Boolean variables in CNF (*conjunctive normal form*).

The argument of that paper can be adapted to k-value DNF and thereby suggests that the learnability of k- value DNF with membership queries has similar behavior to that of learning DNF without membership queries since the same relationship to learnability without membership queries holds for both classes.

The proof in Angluin's paper can be adapted to k-value DNF with MQ showing if DNF is learnable with MQ then it is learnable without MQ (in the PAC predictability framework). Merely change the arithmetic and logic used in the cryptography system to work in base k. The top right of page 344 in that reference shows how to encode a logic and gate and an inverter in CNF. To show how the encoding could be done for a base $k > 2$, let $k = 3$. Then $x = 1 \Rightarrow y = 2$ would be represented in k-value CNF as: $(x \neq 1 \vee y \neq 1) \wedge (x \neq 1 \vee y \neq 0)$. Note that the individual literals of k-value CNF must be inequalities (not restricted to equalities) or the language would not be very expressive (and would also not be the complement of k-value DNF). For general k, this kind of implication will use $k - 1$ disjunctions. □

Lemma 4.2 *k-value DNF $\leq_{pwm}$ (2-value) DNF with $k > 2$.*

Proof (Sketch): Represent each k-value variable as a tuple of 2-value variables, and define f to convert instances and g to convert hypotheses accordingly. In general, not all codes in a tuple of variables will be used, so define $f'$to map instances in DNF with an unused code to $\top$. Make g consistent with this mapping to $\top$ by augmenting a hypothesis with conjunctions covering all of these unused codes when translating that hypothesis from $\mathcal{R}$ to $\mathcal{R}'$.

It is readily seen that the consistency properties of g with f and $f'$are satisfied as well as the polynomial size/time requirements in Definition 4.1. □

For the reduction proof from $l(b{+}1)$-value DNF to μ-OF$_{l,b}$, represent each of n DNF variables as a particular tree node at depth $d = \lceil \log_b n \rceil$ (tree skeleton on left in Figure 4.2 for $b = 2$, $l = 3$, and $n = 3$). Each conjunction in a DNF expression is represented as a single tree. Each DNF variable v_i corresponds to tree node v_i' for $i = 0$ to $n - 1$. Make each combination of one of the l node labels and 0 to b children under a given node be equivalent to one of the $l(b{+}1)$

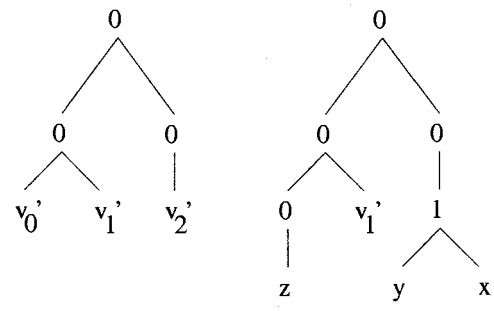


FIGURE 4.2. Tree skeleton (left), pattern produced by g from $v_0 = 3 \wedge v_2 = 7$ with $b = 2$, $l = 3$, $n = 3$ (right).

values for the DNF variable corresponding to that tree node. Without loss of generality, we assume the values for DNF variables are $0 \ldots l(b+1)$-1 and the tree node labels are $0 \ldots l - 1$. The corresponding tree node for a DNF value v_i will then have $\lfloor v_i/l \rfloor$ children and node label $v_i \bmod l$. Conversion of a tree node to a DNF value will be the inverse of this conversion, i.e., a tree node with label i and c children will be mapped to $cl + i$. Figure 4.2 (left) shows a tree skeleton with $n = 3$ nodes representing DNF variables at depth $d = 2$. The tree on the right shows the tree pattern equivalent of the conjunction $v_0 = 3 \wedge v_2 = 7$ for $b = 2$, $l = 3$, $n = 3$. The right (third) subtree has node label 7 mod 3 = 1 and two children with variables that are not used elsewhere, the middle subtree is just a tree pattern variable since v_1 is unconstrained in the conjunction.

Lemma 4.3 *k-value DNF $\leq_{EQMQ}$ $OF_{l,b}$ for $2 \leq k \leq l(b+1)$.*

Proof (Sketch): Represent each DNF variable as (always the same) node in a tree as in Figure 4.2. Each of the k values for DNF is represented as one of the

$l(b+1)$ combinations of the node label and number of children in the corresponding tree node. Functions f and g are then straightforward. Function f'will be the inverse of f by translating each node label/children combination in a tree instance to the corresponding value of a DNF variable, where applicable. Otherwise, f'will translate a tree instance not conforming to this conversion scheme to $\perp$. But a tree instance w' having a combination of node label and number of children not representing a DNF code will be translated to $\top$. Further, g will be defined to append tree patterns covering each of these unused codes.

Then the consistency requirements for g with f and f'in Definition 4.1 hold because tree instances not matched by any tree pattern produces by g are translated to $\perp$, and tree instances corresponding to an unused label-children combination are mapped to $\top$ and matched by the extra patterns added by g. Further, the polynomial size/time requirements are met. □

4.2. Definitions

The following definitions are used to help show that exact learning an ordered forest (or unordered tree) with certain restrictions is as hard as exact learning of DNF.

The reduction definitions for PAC prediction are not directly applicable to exact learning with EQ, so Definition 4.5 below will be used. The symbol $\leq_{EQ}$ below means exact reducibility with only the EQ oracle available. This definition requires the mapping f always produce an element in the instance space of $\mathcal{R}'$.

In the following, a reduction is defined from exact learning of a class (set of representations) $\mathcal{R}$ to exact learning of another class $\mathcal{R}'$ with corresponding instance spaces $\mathcal{I}$ and $\mathcal{I}'$. Functions f and g map instances and hypotheses from $\mathcal{R}$ to $\mathcal{R}'$, and g' maps hypotheses in the reverse direction subject to the

conditions in these definitions. Unlike in the definition of pwm reduction, the hypotheses in $\mathcal{R}'$ are all required to map back to $\mathcal{R}$ via g' in a way consistent with all the instances in $\mathcal{I}$.

Definition 4.5 *$\mathcal{R} \leq_{EQ} \mathcal{R}'$ with corresponding instance spaces $\mathcal{I}$ and $\mathcal{I}'$ iff $\exists$ mappings f: $\mathcal{I} \rightarrow \mathcal{I}'$, g': $\mathcal{R}' \rightarrow \mathcal{R}$, and g: $\mathcal{R} \rightarrow \mathcal{R}'$ such that:*
(1) $\forall\ r \in \mathcal{R}$ and $w \in \mathcal{I}$, $w \in L(r) \Leftrightarrow f(w) \in L(g(r))$
(2) $\forall r' \in \mathcal{R}'$ and $w \in \mathcal{I}$, $w \in L(g'(r')) \Leftrightarrow f(w) \in L(r')$
(3) Function g produces a polynomial-sized result, and the other functions produce a result in polynomial time (and therefore polynomial size).

The following lemma essentially says that as far as the subsets of instances in $\mathcal{I}$ are concerned, functions g and g' behave like inverses:

Lemma 4.4 *$\forall\ r \in \mathcal{R}\ L(r) = L(g'(g(r)))$.*

Proof: $\forall\ w \in \mathcal{I}$, $w \in L(r) \Leftrightarrow f(w) \in L(g(r))$ by property (1) $\Leftrightarrow w \in L(g'(g(r)))$ by property (2) of Definition 4.5. □

(This lemma says nothing about the relationship between g and g' with instances in $\mathcal{I}'$ which are not in the range of f, nor does it imply $L(g(g'(r'))) = L(r')$ for $r' \in \mathcal{R}'$.)

Lemma 4.5 *If $\mathcal{R} \leq_{EQ} \mathcal{R}'$ and $\mathcal{R}'$ is exactly learnable with EQ only, then $\mathcal{R}$ is exactly learnable with EQ only, i.e., $\mathcal{R}$ is no harder to learn than $\mathcal{R}'$ is.*

Proof: Suppose that a learning algorithm $\mathcal{A}'$ is given for learning $\mathcal{R}'$ with only equivalence queries. Construct an algorithm $\mathcal{A}$ for $\mathcal{R}$ using the mappings f and g' as follows. When $\mathcal{A}'$ calls EQ with a hypothesis r', r' gets translated by g' and EQ is called with $g'(r')$ (Figure 4.3). The example returned is translated by f and is provided to $\mathcal{A}'$ as a counterexample to r'. By property (1) of Definition 4.5, for any target $t \in \mathcal{R}$, there is always a corresponding hypothesis $t' \in \mathcal{R}'$ of size polynomial in $|r|$. When $\mathcal{A}'$ calls EQ with a hypothesis r' ,

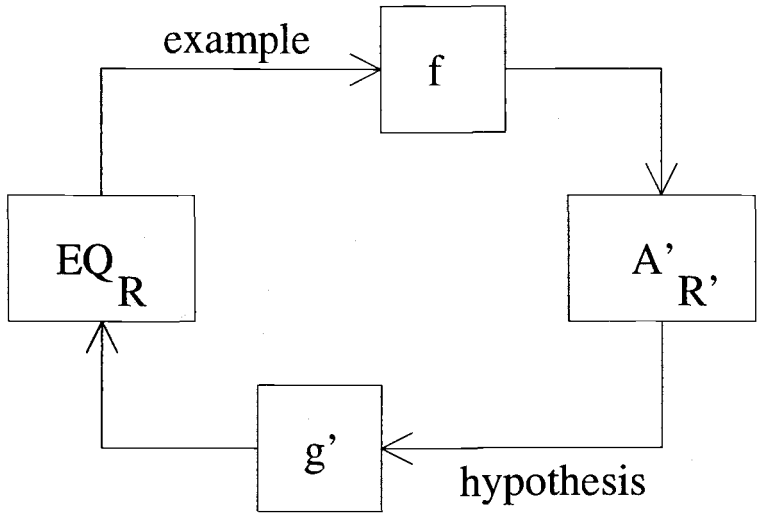


FIGURE 4.3. Reduction Simulation Learning $\mathcal{R}$ using $\mathcal{A}'$ algorithm.

which is converted to $g'(r')$ and a counterexample x is returned, the following is true. Counterexample x is in $L(g'(r')) \oplus L(t)$ (symmetric difference of converted hypothesis and target). Then $f(x) \in L(r') \oplus L(g(t))$ by properties (1) and (2). Similarly, when EQ says yes, then $L(g'(r')) = L(t)$, and the reduced learning algorithm $\mathcal{A}$ will therefore correctly identify the target set t. The learning problem will therefore look like a valid problem to algorithm $\mathcal{A}'$, and $\mathcal{A}'$ can learn any target corresponding to one in $\mathcal{R}$ while getting a normal response from EQ. $\square$

4.3. Reductions for Exact Learning With Equivalence Queries

The following lemma shows a reduction of exact learning of DNF classes from a lower base to a higher base.

Lemma 4.6 *For any infinite or finite $k > 2$, 2-value DNF $\leq_{EQ}$ k-value DNF.*

Proof: Let variables $v_1 \ldots v_n$ in DNF be mapped to variables $v'_1 \ldots v'_n$ in k-value DNF and each conjunction in DNF be mapped to a conjunction in k-value DNF. Define f to map each instance variable-value pair $v_i = j$ within each conjunction to $v'_i = j$.

Let r_i denote the ith variable in r and similarly for w_i and $g(r)_i$. Define g to map constrained variables like f does: $v_i = j$ to $v'_i = j$. Then for all $w \in \mathcal{I}$ and $r \in \mathcal{R}$, a single-conjunction hypothesis r matches an instance w iff for each variable-value pair $r_i = c_i$ in r, there is a pair $w_i = c_i$ in w. Further there will also be the same sets of pairs $g(r)_i = c_i$ and $w'_i = c_i$. If r is a disjunction, then one of its conjunctions matches w iff one conjunction in $g(r)$ matches $f(w)$. So the same subset and matching relationship holds in $\mathcal{R}'$ and property (1) of Definition 4.5.

The g' function works like the inverse of g except conjunctions with values (i.e., the c_j) not equal to 0 or 1 are discarded. Then for any r' and w, the latter is matched by $g'(r')$ iff some conjunction in r' has a subset of the variable-value pairs in $f(w)$, and this conjunction will not be one of the discarded ones. Property (2) is therefore satisfied.

All of these functions compute polynomial-sized results in polynomial time, satisfying property (3) and all of Definition 4.5. □

Theorem 4.7 *k-value DNF with $k \geq 2$ and k either finite or infinite is not learnable with EQ alone.*

Proof: Combine nonlearnability of 2- value DNF (Angluin, 1990) and Lemma 4.6. □

To make a reduction mapping work between trees and DNF, the same tree node must always map to or from the same DNF variable. Otherwise, generalization will be inconsistent between the two classes.

Lemma 4.8 *$l(b{+}1)$-value DNF $\leq_{EQ}$ μ-$OF_{l,b}$ with $l(b{+}1)$ either finite or infinite and $b \geq 2, l \geq 1$.*

Proof: Suppose the DNF expressions to be represented have variables $v_1 \ldots v_n$. Fix a tree with just sufficient children under each node to represent each v_i with a tree node v_i' at depth $d = \lceil \log_b n \rceil$ levels below the root as in Figure 4.2 and Lemma 4.2. (If $b = \infty$, then $d = 1$.) The parts of the tree above depth d are fixed to be the same for all hypotheses and instances mapped from $\mathcal{R}$. The combinations of l available node labels and 0 to b children under each v_i' represent the $l(b{+}1)$ possible DNF values, Without loss of generality, value c_i for variable v_i in DNF could be translated to tree node v_i' with $\lfloor c_i/l \rfloor$ children and label number $c_i \bmod l$. Unconstrained DNF variables are mapped to unconstrained tree pattern variables.

Define $f(w)$ to map variable-value pairs $w_1 = c_1 \ldots w_n = c_n$ in DNF instance w to the corresponding tree variables $f(w)_1 \ldots f(w)_n$ with subtrees according to the above representation. Similarly, define g to use the same correspondence but with each unconstrained DNF variable r_i translated to tree node $g(r)_i$ as a tree variable not appearing elsewhere in the tree. Property (1) of Definition 4.5 is satisfied for the same reasons as in Lemma 4.6.

Define $g'(r')$ to perform the inverse of g for each tree it is given in a forest hypothesis which conforms to the above definitions. Other tree patterns are mapped by g' according to what set of tree instances produced by f they match. If a tree pattern in r' does not match any tree instance of the form $f(w)$, g' ignores it (and g' returns ϕ, the empty DNF hypothesis if all the tree patterns are this way). Otherwise subtree r_i' at depth d will match either just one of the combinations of label and number of children or all $l(b{+}1)$ of them; return a constrained or unconstrained DNF variable number i, respectively.

This definition automatically guarantees the ith subtree at depth d of r' will match the ith subtree of $f(w)$ iff DNF variable i in $g'(r')$ is unconstrained or constrained to have the same value as variable i in w, satisfying property (2).

All functions produce polynomial-sized results in polynomial time, satisfying property (3) of Definition 4.5. □

The following theorem ties together all the classes with a finite number of values (and using only equivalence queries).

Theorem 4.9 *DNF* $\leq_{EQ}$ $l(b{+}1)$*-value DNF* $\leq_{EQ}$ $OF_{l,b}$ *and all of these classes are not learnable with EQ alone.*

Proof: Combine (Angluin, 1990) with Lemmas 4.6, 4.8 (and 4.5). □

4.4. Reductions for Exact Learning with Equivalence and Membership Queries

These sections show reductions between DNF and OF classes when both equivalence and membership queries are available.

Unlike the previous reductions for exact learning with EQ only, the following reductions from class $\mathcal{R}$ to $\mathcal{R}'$ allow membership queries which permit the learner to ask if an example it chooses is a member of the target. A learning algorithm $\mathcal{A}'$ for class $\mathcal{R}'$ will be used to learn $\mathcal{R}$, assuming the EQ and MQ oracles for $\mathcal{R}$ are available. This requirement means conversion of examples from $\mathcal{R}$ to $\mathcal{R}'$ (f below), and the reverse for MQ (f'), and the hypotheses (g'). The requirement that $\mathcal{R}'$ have a target corresponding to each target in $\mathcal{R}$ also implies a correspondence of hypotheses in the forward direction (call this function g). As in the reductions for exact learning with EQ only, there is no requirement for an explicit conversion routine for this case.

The simulation to reduce a class $\mathcal{R}$ to a class $\mathcal{R}'$ for exact learning with membership queries is diagrammed in Figure 4.4. Start by invoking the learn-

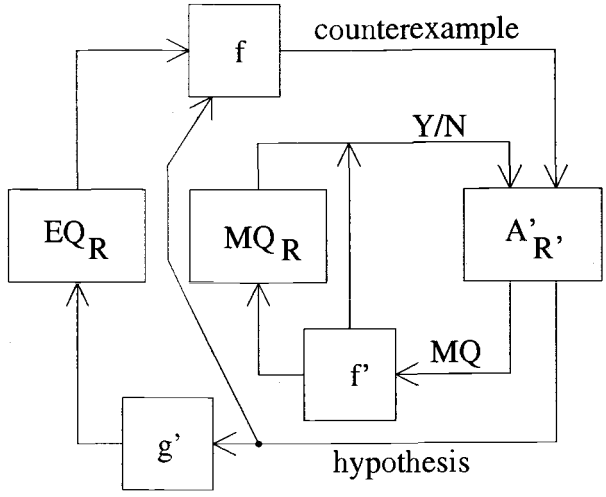


FIGURE 4.4. Simulation for EQ+MQ Reduction

ing algorithm $\mathcal{A}'$ for $\mathcal{R}'$. We assume that for each target $r \in \mathcal{R}$, there is a corresponding target $g(r) \in \mathcal{R}'$. $\mathcal{A}'$ learns a target in $\mathcal{R}'$ by calling EQ with hypotheses in $\mathcal{R}'$ and MQ with instances in $\mathcal{I}'$. But as Figure 4.4 shows, we simulate these oracles with the corresponding oracles for $\mathcal{R}$.

When $\mathcal{A}'$ asks a membership query over an instance $x' \in \mathcal{I}'$, it is then fed to f' which maps this instance to an instance $f'(x') \in \mathcal{I}$. The MQ oracle for $\mathcal{R}$, looks at $f'(x')$ and responds *yes* or *no* depending on whether or not $f'(x')$ is in the target. This answer is then passed on to $\mathcal{A}'$. When $\mathcal{A}'$ makes an equivalence query on a hypothesis, the function g' maps the hypothesis in $\mathcal{R}'$ to a hypothesis in $\mathcal{R}$. This function can be many-to-one.

It is only necessary to find a target giving the correct subset of $\mathcal{I}$, but membership queries with examples not represented in $\mathcal{I}$ must be answered to

make the algorithm $\mathcal{A}'$ work correctly. Following the *pwm* definition in (Angluin & Kharitnov, 1995), the special symbols $\perp$ and $\top$ (called *bottom* and *top*, respectively) represent special instances associated with $\mathcal{I}$ that return *false* and *true* when queried with a membership query. The function f' is allowed to map an instance in $\mathcal{I}'$ to $\top$or $\perp$. Further, $\top$will always be matched by (contained in the set of instances represented by) any pattern $r \in \mathcal{R}$. Similarly, $\perp$ will never be matched by any hypothesis r.

The following definition enforces three consistency relationships between the functions used in the reduction. The purpose of each part of the MQ reduction definition is as follows: Part (1) enforces consistency between f' and g so membership queries will get a response consistent with the target. Part (2) covers consistency between g and f to achieve consistency between counterexamples returned by EQ and the target. Part (3) requires consistency between f and g' to make counterexamples returned from EQ consistent with the hypothesis given as an argument of EQ. Part (4) specifies the time and size requirements of these functions. Unlike Definition 4.1 of pwm reduction, this definition requires a means, i.e., g', to convert the returned hypothesis.

Definition 4.6 *$\mathcal{R} \leq_{EQMQ} \mathcal{R}'$ with corresponding instance spaces $\mathcal{I}$ and $\mathcal{I}'$ if $\exists$ mappings g: $\mathcal{R} \to \mathcal{R}'$, g': $\mathcal{R}' \to \mathcal{R}$, f: $\mathcal{I} \to \mathcal{I}'$, and f': $\mathcal{I}' \to \mathcal{I} \cup \{\perp, \top\}$ such that $\forall\ w \in \mathcal{I}$, $w' \in \mathcal{I}'$, $r \in \mathcal{R}$, $r' \in \mathcal{R}'$:*
(1) (MQ-target) $f'(w') \in L(r) \cup \{\perp\} \Leftrightarrow w' \in L(g(r))$
(2) (EQ counterexample-target) $w \in L(r) \Leftrightarrow f(w) \in L(g(r))$
(3) (EQ hypothesis-counterexample) $w \in L(g'(r')) \Leftrightarrow f(w) \in L(r')$
(4) Function g produces a result of polynomial size, and the other functions run in polynomial time (and therefore produce polynomial-sized results).

Corollary 4.10 *If $\mathcal{R} \leq_{EQMQ} \mathcal{R}'$, then $\mathcal{R} \leq_{pwm} \mathcal{R}'$.*

Proof: The requirements for $\leq_{EQMQ}$ (Definition 4.6) are a superset of those for $\leq_{pwm}$ (Definition 4.1). □

A learning algorithm for the class on the upper side of a reduction (i.e., $\mathcal{R}'$) is supposed to be able to learn any target in the lower class ($\mathcal{R}$). The conversions in the reduction must therefore make learning the lower class look like learning the upper class. The following lemma establishes this claim.

Lemma 4.11 *If $\mathcal{R} \leq_{EQMQ} \mathcal{R}'$ and $\mathcal{R}'$ is (exactly) learnable with equivalence and membership queries, then $\mathcal{R}$ is learnable with equivalence and membership queries.*

Proof: The poly size bound on g guarantees the target in $\mathcal{R}'$ has size polynomial in the target r for $\mathcal{R}$. By property (1) of Definition 4.6, a simulated membership query with argument w' will give an answer *yes* (with respect to the target $g(r)$) iff $\mathrm{MQ}(f'(w'))$ answers *yes* with respect to r. Therefore membership query responses will faithfully test the simulated target. Even if f' returns $\bot$ or $\top$, the property is still required to hold so the membership query simulation gives a faithful response.

Similarly, by property (2) of the definition, a counterexample w returned by EQ will be matched by a target r iff $f(w)$ which is the example translated to $\mathcal{I}'$ is matched by the converted target $g(r)$–for any r. The simulation will therefore properly provide appropriate counterexamples in the class $\mathcal{R}'$ for the learning algorithm $\mathcal{A}'$.

Property (3) implies a counterexample w to a hypothesis $g'(r')$ when converted to $f(w) \in \mathcal{I}'$ will always be a counterexample for hypothesis $r' \in \mathcal{R}'$, even if r' does not represent the same set of instances as any $g(r)$.

Each simulated query for $\mathcal{R}'$ invokes no more than one query in $\mathcal{R}$. So the number of actual queries is bounded by the bound for algorithm $\mathcal{A}'$ for targets of size determined by the size of the target $r \in \mathcal{R}$ and the size polynomial for g and similarly for the example sizes. □

The simulation implied by the reduction (Figure 4.4) assumes there is a learning algorithm $\mathcal{A}'$ for $\mathcal{R}'$. Then an algorithm $\mathcal{A}$ for learning $\mathcal{R}$ is constructed

using conversion routines for the three functions f', f, and g'. The mapping g is required by the existence of a target in $\mathcal{R}'$ for each target in $\mathcal{R}$, although this mapping is not explicitly used in the reduction simulation.

The reduction definition effectively splits $\mathcal{I}'$ into equivalence classes of instances, each of which corresponds to a single instance in $\mathcal{I}$. But some instances are not so mapped, so f' maps them back to $\top$ or $\bot$ (which are always and never matched, respectively). The mapping from $\mathcal{I}$ to the equivalence classes of $\mathcal{I}'$ is therefore *one-to-one into.* All corresponding hypotheses between the two classes have to be consistent with this mapping by either matching or not matching corresponding instances in these classes.

4.5. Reducing DNF to a Higher Base

The following lemma compares DNF classes of different bases by reducing a DNF class with a lower base to a DNF class with a higher base–with membership queries.

Lemma 4.12 *(2-value) DNF* $\leq_{EQMQ}$ *k-value DNF with* $2 \leq k < \infty$.

Proof: Let variables $v'_1 \ldots v'_n$ in k-value DNF correspond to variables $v_1 \ldots v_n$ in (2-value) DNF. Define g to map each of the values in 2-value DNF to first 2 values in k-value DNF and convert a target/hypothesis $r \in \mathcal{R}$ to

$$g(t') = r \vee \bigvee_{i=1}^{n} \bigvee_{j=2}^{k-1} (v'_i = j)$$

Where r' is r with each v_i is replaced with v'_i. This conversion adds a term to r for each value in k-value DNF not appearing in 2-value DNF and for each Boolean variable $v'_1 \ldots v'_n$ used in k-value DNF.

Define function f' to map instances $w' \in \mathcal{I}'$ using only values 0 and 1 back to their corresponding instances in $\mathcal{I}$ (i.e., effectively the inverse of g when applicable to instances in $\mathcal{I}$). Instances using values ≥ 2 would be mapped to $\top$

and always return *true* when used with MQ. Since g has hypothesis terms which cover all of the examples when an instance has values ≥ 2, both sides of equation in property (1) are true. When an instance does not have variables with values ≥ 2, $g(r)$ matches w' implies the set of variable-value pairs in $g(r)$ (not counting the special terms added by g–which do not apply) is a subset of those in w'. The same is true for r and $f'(w')$ so the latter pair also matches. This argument works equally well in both directions, so property (1) of Definition 4.6 is satisfied in all cases.

Define f as follows. An instance w will have all variables constrained. Convert $v_i = j$ in 2-value DNF to $v'_i = j$ in k-value DNF. Then an example w is matched by a conjunct c of a hypothesis $r \in \mathcal{R}$ iff that example contains all the variable=value pairs of c. This relation holds for $f(w)$ and $g(r)$, so property (2) for consistency between f and g is satisfied.

Function g' is defined by discarding hypothesis terms having values 2 through $k-1$. That is, for each conjunctive term c' that contains a constraint of the form $v'_i = j$ with $j \geq 2$, g' will discard c'. Otherwise g' converts the term by converting to corresponding variables (v'_i to v_i) and otherwise leaving c' unchanged.

The justification of property (3) between f and g' is as follows. A positive counterexample will not be matched by any conjunctive term in the hypothesis, whereas a negative counterexample will be matched by (at least) one term. If term c' in r' has values ≥ 2, then it will not match any example converted from $\mathcal{I}$ by f and will be discarded when converted by g'. Otherwise, as above, the term will be converted with the only change being to replace the v'_i with v_i, and corresponding instances will match corresponding hypotheses in both classes. In both cases, property (3) is satisfied. All of these functions/conversions are polynomial time and size. □

The requirement that k be finite is needed to prevent the expression for $g(r)$ from being infinite–and to avoid contradicting Corollary 4.14.

4.6. Reducing DNF to Ordered Forests

Lemma 4.13 *k-value DNF $\leq_{EQMQ}$ μ-$OF_{l,b}$ for $k = l(b+1)$, with $l,b \geq 2$ and either finite or infinite.*

Proof: Define g as follows. Follow the approach of Lemma 4.8 for EQ only and Figure 4.2. Use a tree skeleton U' with n distinct variables at the lowest level (depth $d = \lceil \log_b n \rceil$ or 1 if $b = \infty$) to represent n DNF variables. Include only enough subtrees in the skeleton at each level to have n nodes at depth d. Each DNF conjunction is mapped to a single tree pattern. Within a particular conjunction, each DNF variable is mapped to a particular tree node at depth d. Let the conjunction r be $r_1 = c_1 \wedge r_2 = c_2 \wedge \ldots \wedge r_p = c_p$ with $p \leq n$. Then g converts r_i to subtree r'_i with root label $c_i \bmod l$ and number of children $\lfloor c_i/l \rfloor$ which replaces variable v'_i in the tree skeleton. The DNF variables v_i that do not appear in that term will be represented as nodes labeled by unique variables v'_i in the corresponding tree pattern r'. Function f is similar to g except it maps only specific instances, i.e., all variables are assigned specific values.

Define f' to map instances in the reverse direction as follows. Have f' map any tree instance w' not matched by the skeleton U' to $\perp$. Otherwise each v'_i in U' corresponds to some subtree w'_i in w'. Hence the tree instance w' can be converted back to $\mathcal{I}$ by inspecting each w'_i and translating its root label and number of children back to the corresponding value for v_i. If w'_i has root label m and c children, then it is translated as $v_i = cl + m$.

To satisfy property (1) of Definition 4.6, suppose that w' is a tree instance and $w = f'(w')$ is the corresponding vector of variable-value pairs. Let the target r be a disjunction of terms, where each term is a conjunction of variable=value pairs. Then $r' = g(r)$ is a forest of tree patterns. A term r'_j in r' matches w iff each variable that is constrained in r_j is similarly constrained in w. The same conversion between DNF values and node label/number of children is used for g and f. Because the unconstrained variables in r_j are left as unconstrained

variable nodes in $g(w_j)$ r_j matches w iff the corresponding subtree $g(w_j)$ matches the corresponding tree instance w'. Hence r matches w iff there is a tree in $g(w)$ that matches w'. In other words, $w = f'(w') \in L(r)$ iff $w' \in L(g(r))$.

Define g' as follows. Ignore any tree pattern that is not subsumed by the skeleton U'. All other tree patterns are mapped similarly to f' except leaf nodes which are labeled by variables are unconstrained in the corresponding conjunction. Function g' maps r' to a disjunction of all such conjunctions, or ϕ (the empty hypothesis) if no conjunction remains.

To show that property (2) is satisfied, note that $w \in L(r)$ iff all constraints in some term s in r are satisfied by w. Since f and g work the same way for the constrained variables, all the constrained variables in s and the corresponding variables in w will be mapped to identical subtrees by g and f, respectively. Since unconstrained variables in r map to variable nodes that match any subtree, and $f(w)$ and $g(r)$ share the same skeleton U', $f(w) \in L(g(r))$ iff $w \in L(r)$.

To show that property (3) is satisfied, first note that any tree pattern in r' that does not share the skeleton U' cannot match any $f(w)$. Let r^* be the subset of tree patterns in hypothesis r' that share U'. Since U' also matches $f(w)$, $f(w) \in L(r')$ iff $f(w) \in L(r^*)$. For the tree patterns in r^*, g' and g are exact inverses of each other since they all share U'. This implies that $g'(r^*) = r$ iff $r^* = g(r)$. From property (2), it follows that $w \in L(g'(r^*))$ iff $f(w) \in L(r^*)$. From the relation between r' and r^* above, $w \in L(g'(r^*))$ iff $f(w) \in L(r')$. Since $g'(r^*) = g'(r')$, $w \in L(g'(r'))$ iff $f'(w) \in L(r')$. All functions produce polynomial-sized results in polynomial time. All properties in Definition 4.6 are therefore satisfied. □

Corollary 4.14 *∞-value DNF is learnable with equivalence and membership queries.*

Proof: By Lemma 4.13, ∞-value DNF reduces to μ-OF. The latter is learnable by Theorem 3.12 for OF. The algorithms for each of this class works just as well

for the μ-OF subclass by stripping the parts dealing with multiple copies of the same variable, since this type of hypothesis is not allowed in μ-OF. □

4.7. Discussion and Open Problems

Some potential reductions such as reducing k- value DNF to a lower base such as 2- value DNF will not be able to represent accurately all hypotheses in the former class (lower side of the reduction) as hypotheses in the latter class. The obvious way to reduce 4- value DNF to 2- value DNF would be to represent each variable in the former as a pair of variables in the latter. But if one variable in a pair is constrained and the other is not, there is no compact way to exactly represent such a hypothesis in 4- value DNF (without exponential size if there are many variable pairs of this type). Any attempt to approximately translate the hypothesis would lead to an inconsistency with some instances; other approaches lead to similar problems. We call this problem the *overly expressive hypothesis space problem* or *untranslatable generalization problem*, since the hypothesis space contains some hypotheses that cannot be translated into the space at the lower side of the reduction. The problem of reducing k- value DNF to ordered forests with *repeated variables* has this same difficulty because the hypotheses with repeated variables apparently can not be translated back to DNF in a way consistent with all instances. We now list the open problems from this chapter.

Open Question 1 *k- value DNF $\leq_{EQMQ}$ (2-value) DNF with $k > 2$?*

Open Question 2 *k- value DNF $\leq_{EQMQ}$ $OF_{l,b}$ for $2 \leq k < l(b+1)$?*

These two reductions will work in the *pwm* framework (Lemmas 4.2 and 4.3) because no hypothesis ever needs to be returned, avoiding the above difficulties. But the PAC framework will have similar difficulties with the hypotheses that can't be accurately translated since the final hypothesis must be converted.

Open Question 3 $\mu\text{-}OF_{l,b} \leq_{EQMQ} l(b+1)\text{-}value\ DNF?$

This question asks whether ordered trees *without* repeated variables can be reduced to DNF. Part of the difficulty with this reduction is a size problem. Trees could be very deep and narrow, so a fixed correspondence between tree nodes and DNF variables would require exponentially many DNF variables. An attempt to assign the correspondence dynamically (which probably requires making the learner also learn this correspondence) faces the difficulty of preventing the DNF learner from using up all the allocated DNF variables prematurely–in the worst case, using membership queries. In this sense, tree patterns are a *sparse representation* whereas DNF is a *fixed representation.* These same obstacles would occur when trying to reduce any sparse notation class to a fixed notation class, so this reduction problem could be called the *sparseness mismatch problem.*

Open Question 4 $OF_{l,b} \leq_{EQMQ} l(b+1)\text{-}value\ DNF?$

This question asks whether ordered trees with repeated variables can be reduced to DNF. Any proof of this question also requires a solution to the same problem encountered in reducing trees without repeated variables to DNF–the sparseness mismatch problem or the *sparse reduction difficulty.*

In this chapter, we have shown that DNF learning reduces to learning unions of ordered tree patterns with bounded degree and bounded alphabet size (bounded ordered forests) without repeated variables (Theorem 4.8). The

reduction proofs for the exact learning model (Angluin, 1988) also apply to the PAC-predictability with and without membership queries (Pitt & Warmuth, 1990; Angluin & Kharitnov, 1995). It is simpler to reduce the classes in the prediction frameworks because the hypotheses of the learning algorithm need not be translated back by the reduction algorithm. The argument in (Angluin & Kharitnov, 1995) that membership queries are not helpful for learning DNF in the PAC framework (assuming certain cryptography problems are hard) applies equally well to k-value DNF. This latter result strongly suggests, but does not prove, that the DNF classes with different bases have the same complexity in the exact learning framework.

5. NONLEARNABILITY OF UNORDERED TREES WITH ONTO-SEMANTICS

Ordered trees and forests have been shown to be learnable from equivalence and membership queries in Chapter 3. In this chapter, 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 complexity theoretic assumptions.

5.1. Unordered Tree Matching

Section 3.1 defines *ordered tree matching* (Definition 3.1) of a tree pattern to a tree instance merely consists of substituting subtrees for variables to produce a tree identical to the tree instance–without rearranging the subtrees. But *unordered tree matching* allows the corresponding children to be in any order, so there is a mapping between the subtrees which can be an arbitrary permutation. Whereas ordered tree matching requires the mapping to be one-to-one onto and respect subtree ordering and be the identity permutation. For ease of analysis, the matching process is therefore separated into substitution and mapping/permutation steps (in that order).

Definition 5.1 *A tree pattern or instance* $r = r_0(r_1 \ldots r_n)$ maps to *a tree instance* $s = s_0(s_1 \ldots s_n)$ *according to* unordered one-to-one onto semantics *iff (1)* $r_0 = s_0$, *(2) there is a one-to-one onto permutation* μ *from* $\{r_1, \ldots, r_n\}$ *to* $\{s_1, \ldots s_n\}$ *such that (3) the child subtree* r_i *recursively maps to some child subtree* s_j *according to unordered one-to-one onto semantics. (Where* $1 \leq i \leq k$ *and* $1 \leq j \leq l$.*)*

We write $r_i\mu = s_j$ *and* $r\mu = s$. *Mapping* μ *is then consistent with one-to-one onto semantics.*

Definition 5.2 *A tree pattern t matches an instance w according to unordered one-to-one onto semantics iff there is a permutation μ consistent and a substitution θ consistent with unordered one-to-one onto semantics such that $t\theta\mu = w$.*

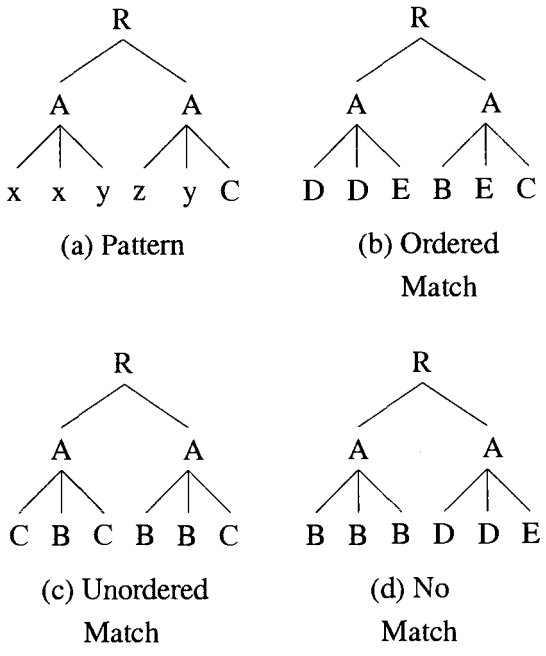


FIGURE 5.1. Unordered Tree Matching

In Figure 5.1, tree pattern (a) matches tree instance (b) according to ordered (or unordered) one-to-one onto semantics. The tree pattern matches instance (c) (only by unordered semantics) with substitutions (of the form, tree pattern variable/instance subtree) $\{x/C, y/B, z/b\}$ by permuting the second and third child of the first subtree.. But instance (c) can also be matched with the substitutions $\{x/B, y/C, z/B\}$ by swapping the two subtrees and the first

and second children of the right pattern subtree. Instance (d) is not matched by tree pattern (a) according to unordered one-to-one onto semantics both because the is no C child to be matched by the C in the pattern and because there is no child label common to both subtrees for the y variable to match.

5.2. Unordered Tree Reductions with Equivalence Queries

This section shows that unordered trees without repeated variables (μ-UT) are not learnable with equivalence queries alone (based on a reduction from DNF). The *monotone DNF* class (*MDNF*) is used as an intermediate step. The notation $\leq_{EQ}$means a reduction in the exact learning framework with only equivalence queries available (as defined in Chapter 4). The functions f, g, and g' map instances and hypotheses between the two classes in the reduction.

Definition 5.3 Monotone DNF (MDNF) *is the subclass of (2-value) DNF expressions which has no negated variables.*

Lemma 5.1 *DNF* $\leq_{EQ}$ *Monotone DNF (MDNF).*

Proof (Sketch): Use the standard reduction of representing each DNF variable v and its negation $\neg v$ as two separate MDNF variables (Kearns & Vazirani, 1994). □

Lemma 5.2 *MDNF* $\leq_{EQ}$ *μ-UT.*

Proof: First assume the number of conjunctions in the MDNF target formula is known, and define g as follows. The disjunction of conjunctions is represented as just a single tree pattern rather than as a forest of trees as in the ordered-tree reductions in Chapter 4. Represent each conjunctive term in an MDNF expression as one subtree of a two-level μ-UT tree pattern with each subtree having as many children as the number of Boolean variables. Let $v_1, \ldots v_n$ be

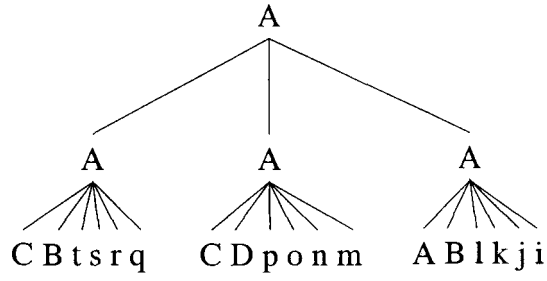


FIGURE 5.2. Monotone DNF hypothesis $xy \vee xw \vee zy$ to μ-UT (g mapping)

these Boolean variables available for MDNF; then associate each variable v_j with a tree node label C_j. Convert each conjunction number i of the form $v_{i1} \wedge v_{i2} \wedge \ldots v_{ik}$ to a subtree with leaves $C_{i1}C_{i2} \ldots C_{ik}\ s_{ik+1} \ldots s_{in}$. Where the s_{ij} are unique variables within the μ-UT pattern. For example, Figure 5.2 shows a MDNF expression and its corresponding tree (where the letters $t \ldots i$ are the unique variables and the letters $A \ldots F$ represent the constants C_j which correspond to variables $z \ldots u$).

Define f to convert each instance with variables $v_1 \ldots v_k$ true to tree having the first subtree with $C_1 \ldots C_k G \ldots G$, where G is a constant not used in any tree pattern returned by g. The other subtrees have the entire set of C's: $C_1 \ldots C_n$ (Figure 5.3). Then an MDNF expression r matches a boolean instance w implies the set of (true) variables in some conjunction (say conjunction i with variables $v_1 \ldots v_k$) in r is a subset of the set of true variables in w (say $v_1 \ldots v_l$, with $l \geq k$). Therefore constants $C_1 \ldots C_k$ will be present in subtree i of $g(r)$ and constants $C_1 \ldots C_l$ in the first subtree of $f(w)$, and the other subtrees of $f(w)$ have all constants $C_1 \ldots C_n$, so $g(r)$ matches $f(w)$ Similarly the same

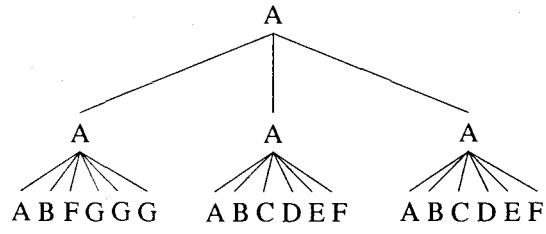


FIGURE 5.3. Monotone DNF instance zyu to μ-UT

subset relationship in these tree patterns ($g(r)$ matches $f(w)$) implies the above subset of variables relationship for r and w, and property (1) of Definition 4.5 is satisfied.

Define $g'(r')$ for any $r' \in \mathcal{R}'$ to produce an MDNF formula matching the set of instances $w \in \mathcal{I}$ such that r' matches $f(w)$ as follows. If r' does not match any tree instances produced by f, i.e., not even a tree in which all subtrees have all C_j constants, then return the empty hypothesis. For each subtree r'_i of r', perform the following. Determine what subset of $\{C_1, \ldots C_n\}$ is required in the first subtree of a tree instance for r'_i (and therefore r') to match. Return an MDNF conjunction for r'_i with the variables corresponding to this subset of the C's. Tree pattern r' then matches $f(w)$ iff some subtree of r' has a subset of the C_j's in the first subtree of $f(w)$. The latter is true iff some conjunction of $g'(r')$ has a set of variables constrained to be true which is a subset of the true variables in w. Property (2) of Definition 4.5 is therefore satisfied for both the empty hypothesis and non empty r' with this translation.

Use the following strategy for an unknown number of conjunctions. Represent the empty MDNF hypothesis with the empty tree hypothesis (ϕ). Run

the learning algorithm for μ-UT with $k = 1$ subtrees. Either the algorithm succeeds or exceeds its polynomial running time. In the latter case, increase k by 1 and repeat the process. This process terminates in polynomial time.

All functions produce polynomial-sized results in polynomial time, satisfying property (3) and therefore Definition 4.5. □

Note that the ordering of different subtrees and the ordering of children within subtrees doesn't matter because they effectively behave like sets.

Theorem 5.3 *μ-UT is not learnable with EQ alone.*

Proof: Combine Lemmas 5.1 and 5.2 with (Angluin, 1990). □

Angluin's result uses a combinatorial argument to show DNF is not learnable with EQ alone, then these lemmas show MDNF and μ-UT are at least as hard to learn with EQ alone–and are therefore not learnable.

5.3. Unordered Onto-Trees are not Learnable from EQ and SQ

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 proof, 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 iff 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 5.4 is consistent with the 2-matrix (b). This is because permuting the last two columns and adding 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 5.4. Matrix Representation of Tree Matching

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 5.4 *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 i^{th} column of the first 2-matrix and the i^{th} column of the permuted second matrix to compute the i^{th} column of a 3-matrix. So, for example, from the columns $\begin{matrix} in \\ c-in \end{matrix}$ and $\begin{matrix} c-j \\ j \end{matrix}$ we form $\begin{matrix} in \\ c-j-in \\ j \end{matrix}$. 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 $i = j = n - 1$, or $c \geq n^2 - 1$.

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 distinct 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. □

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 root and 2 levels and consistency of matrices corresponds to matching of trees. The top of the 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

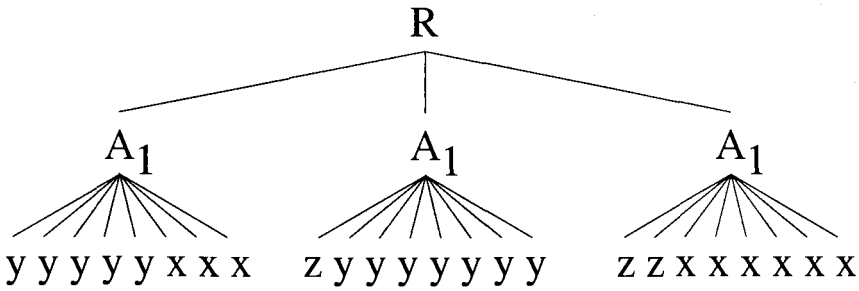


FIGURE 5.5. Sample UT target with $n = 3$ subtrees.

in each column is the number of y's in that subtree, and the bottom element is the number of z's. See Figure 5.5 for a tree pattern with $n = 3$ subtrees, which corresponds to the matrix, $\begin{matrix} 3 & 0 & 6 \\ 5 & 7 & 0 \\ 0 & 1 & 2 \end{matrix}$. 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 5.4 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, say B, and the other two variables with a constant C. We can now show the unlearnability theorem.

Theorem 5.5 *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 eliminates that pattern from $\mathcal{H}$). (None of the types of guesses below eliminates any targets from $\mathcal{H}$.) Guesses inconsistent with the two 2-constant examples are handled by answering *no* and returning one of the two 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.

Lemma 5.6 (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.*

This lemma uses the assumption that the hypothesis space contains a collection of sets L_i any distinct pair of which have a set $L_\cap$ as their intersection–where the latter is not in the hypothesis space.

A diagram could be made with $L_\cap$ in the middle and all the L_i's would include that middle and some petals that collectively surround it. The result would look like a sunflower. The lemma says, given that assumption, the learner

has to do an exhaustive search by testing the L_i's one at a time in order to identify an arbitrary L_i.

Theorem 5.7 *At least n! queries are required to find the target in Theorem 5.5.*

Proof: An alternative proof for Theorem 5.5 can be built using the above lemma as follows. Let E_1 and E_2 be the two example trees obtained from EQ which can be represented as the 2-matrices in Lemma 5.4. Let $L_\cap$ = variablize(E_1) $\cup$ variablize(E_2), where *variablize* converts each distinct constant leaf to a variable (yielding a forest of 2 tree patterns). Let the L_i be the $n!$ possible targets consistent with those two examples; i.e., the 3-matrices of Lemma 5.4. Then this choice satisfies the conditions of Lemma 5.6 because $L_\cap$ has two trees and is therefore not a hypothesis in UT. To show the intersection of distinct L_i and L_j is $L_\cap$, analyze the sets of instances matched by these tree patterns. Note that the matching process is a many-to-one onto mapping from variables in the (potential) target tree to arbitrary constant (subtrees). Multiple variables can map onto the same value. By the properties of matrix matching, an instance with 3 distinct constants matched by a 3-variable target will have the same form and matrix, apart from permutations; so this type of instance cannot be in the intersection of two such L_i. An instance having only one value on all of its second-level children will be matched by all $n!$ 3-variable targets and will be in $L_\cap$. Again, by the properties of matrix matching, a 2-variable instance will be matched by L_i if its matrix is formed by combining two rows in L_i and permuting. An instance matched by both L_i and L_j cannot form one of its rows by adding the top and bottom rows of one of those 3-variable targets, because the other row would then have to come from the middle row of L_i and would have a different set of values than are present in L_j. Therefore the only 2-constant instances matched by both L_i and L_j are those that add the top tow rows (represented by example E_1) and those that add the bottom two rows together (covered by the tree pattern corresponding to E_2). This set of examples is clearly in $L_\cap$.

By Angluin's lemma, at least $n! - 1$ queries are needed, and so UT is not learnable from EQ and SQ. □

5.4. Unordered Onto-Forests are not Learnable from EQ and SQ

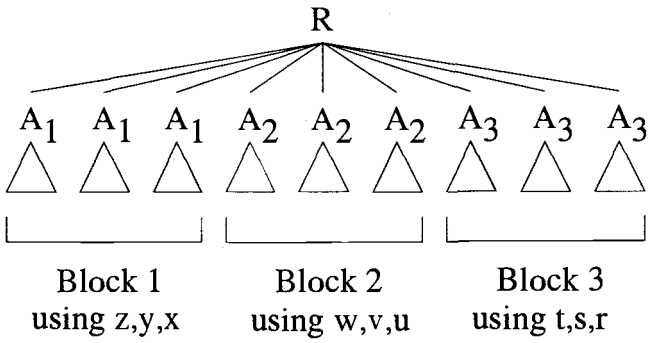


FIGURE 5.6. Sample UF Target for $n = 3$

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, which gives 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 ways of partitioning the set of target variables. These approaches to learning the target must be blocked by the availability of exponentially many partitions and examples.

Even though forest hypotheses are allowed, the following subclass of UT will be sufficient to create a set of targets simultaneously consistent with an exponential number of examples and thereby to construct a proof. Define

$\mathcal{T}$ to be the set of 3-level tree patterns (not forests) each containing several (independent) sets of subtrees and variables 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 5.5 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 5.6 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 5.5), 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 has 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.

Unordered trees potentially have many ways to make variables correspond to constants when matching a pattern to an instance, which would unnecessarily complicate the proof. $\mathcal{T}$ is therefore chosen to enforce a particular correspondence between target variables and constant leaves in examples as follows. Each block of 3 variables in the target will be associated with one of these subtree root labels, and therefore with the corresponding block of con-

stants in an example. The pattern used in the block in the UT proof guarantees which variables within each block must match each constant, so the correspondence between variables in a tree pattern and constant leaves in an instance is completely determined.

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 $\mathcal{T}$ 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 $\mathcal{T}$. The following lemma essentially says only corresponding blocks match:

Lemma 5.8 *A tree pattern $P \in \mathcal{T}$ with blocks (of subtrees) $P_1 \ldots P_n$ matches an instance (or subsumes another pattern) I with blocks $I_1 \ldots I_n$ iff each P_i matches I_i.*

Proof: **only if**: Subtree identification via subtree root labels guarantees that only subtrees in corresponding blocks can match. The definition of matching and the equality of the number of subtrees guarantee that each each P_i must match I_i.

if: Combine the mappings for each pair of corresponding blocks, into a mapping for all subtrees. Each block uses a different set of 3 variables, so there is no conflict between blocks. This mapping and matching root labels guarantee the entire trees match. □

Suppose a query with a 3-variable block is made. Lemma 5.8 guarantees that 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 5.9 *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. By Lemma 5.8, the argument tree is matched by all targets in $\mathcal{H}$ and therefore will 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 5.10 *UT and UF with one-to-one onto semantics are not learnable with equivalence and membership queries.*

Proof: Follows since 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 Defini-

tion 4.1 of (Hellerstein, Pillaipakkamnatt, Raghavan, & Wilkins, 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.

This criteria constrains the number of queries, but proves nothing about polynomial-time learnability. Hellerstein, et.al.'s Definition 4.1 uses examples of size $\leq n$, target size $\leq m$, polynomials p and q as the maximum learned hypothesis size and total size of the examples, and f is an arbitrary concept (subset of the instance space) which is not equivalent to any small (size p) hypothesis. This definition essentially compares the consistency of f with hypotheses of size $\leq p(m,n)$ on all examples of size $\leq n$ versus the consistency of f with targets of size $\leq m$ on a q-sized subset of those examples. The negation of the definition is (with consistency tested only on examples restricted to size $\leq n$): A concept class does not have polynomial certificates iff $\forall$ polynomials p, q $\exists m, n$, and an f not consistent with any concept of size at most $p(m,n)$ and $\forall$ sets Q of examples of total size $\leq q(m,n)$ $\exists$ a concept of size $\leq m$ consistent with f over a set of examples Q.

For UT, the the following values can be used for the parameters in Definition 4.1. The examples used in the proof are all of the same size and therefore equal to the value of n for polynomial certificates (n subtrees of $n^2 - 1$ leaves each using the n defined in the UT nonlearnability proof for Theorem 5.5). The target size m is also the same. Let f be the single concept formed by taking the union of the two examples (with their constant leaves converted to variables). This concept is not expressible as a single tree pattern and therefore satisfies the requirements by not being in the hypothesis space. By the UT proof, any set Q of (polynomially many) examples of size n will be insufficient to distinguish all targets of size m from f because f is consistent with many targets on the 2-constant examples and exponential 3-constant examples would be required to

cover all the targets described to force one of them to be inconsistent with f. Therefore UT does not have polynomial certificates.

For UF, the polynomial-certificate parameters are as follows. Again, the chosen examples and targets are all of the same size (n^2 subtrees with $n^2 - 1$ leaves each using the n defined in the UF nonlearnability proof, giving a total of $1 + n^2 + (n^4 - n^2) = n^4 + 1$ tree nodes). For the hypothesis of exponential size, f, choose the union of variablized versions of all 2^n 2-constant examples. Then any set Q of examples will be inconsistent with at most polynomially many of the targets described and therefore consistent with f on at least one target. UF does not have polynomial certificates showing this class is not learnable.

Hence these proofs are not based on any complexity theoretic assumptions, and are the first of their kind for any "natural" class using polynomial certificates to the best of our knowledge.

5.5. Summary

This chapter showed that the unordered tree and unordered forest concept classes are not learnable with equivalence and subset (or membership queries). These proofs used a combinatorial approach by showing that there are too many choices for how to generalize a hypothesis and no way for a learner to know which one is right without potentially trying them all. The class μ-UT was also shown to not be learnable with EQ alone by reduction from (Angluin, 1990) which uses a combinatorial argument.

6. LEARNING UNORDERED ONTO TREES WITH SUPERSET QUERIES

In Chapter 5 we showed that unordered trees and unordered forests are not learnable with equivalence and subset queries (under one-to-one onto semantics). This chapter first shows there is no difficulty learning unordered trees or forests without repeated variables. Without repeated variables, unordered trees are learnable from subset queries and a single positive example (or equivalence and membership queries) and unordered forests are learnable from equivalence and membership queries.

Like ordered trees, unordered trees are *compact.* Unlike ordered trees, matching is NP-Complete for unordered trees with repeated variables. One way to circumvent the nonlearnability of unordered trees and forests with repeated variables is to use more powerful queries. This chapter describes algorithms for learning unordered trees with superset queries and a single positive training example. Then unordered forests are shown to be learnable from equivalence and superset queries–with both returning counterexamples. The latter algorithm goes to great lengths to avoid the need for matching because of the NP-Completeness of that test.

6.1. Unordered Compactness

Recall that an unordered tree pattern matches an instance if any permutation (one-to-one onto mapping) of the subtrees of the pattern results in a match. We now show that unordered tree patterns are compact.

Lemma 6.1 *The class of Unordered tree patterns (UT) with one-to-one onto semantics is compact.*

Proof: Let $\mathcal{Z}$ be a UT pattern so that $\bigcup \mathcal{V}_i \succeq \mathcal{Z}$. Any permutation $\pi(\mathcal{Z})$ of $\mathcal{Z}$ is contained (according to ordered tree matching) in the union of the permutations of the $\mathcal{V}$'s. Now treat $\pi(\mathcal{Z})$ as an ordered tree, and by Lemma 3.9,

$\pi_j(\mathcal{V}_i) \succeq \pi(\mathcal{Z})$ for some permutation π_j and some $\mathcal{V}_i$. Since the permutations form a group, each permutation of $\mathcal{Z}$ is contained in some permutation of $\mathcal{V}_i$, and hence $\mathcal{V}_i \succeq \mathcal{Z}$ when these are considered as unordered trees. □

6.2. Unordered Forests With No Repeated Variables

When there are no repeated variables, unordered forests are easy to learn with equivalence and either subset or membership queries by independently pruning different parts of the tree pattern. We call this class μ-UF in analogy to read-once μ-formulas of propositional logic. The lack of repeated variables implies the subtrees can be generalized independently. Starting from a single example (for each target tree pattern), each subtree having only leaves as children can be pruned, replaced by a new variable, and the change retained iff the result is a subset of the target. Only polynomially many operations need be performed with this approach to learn the target.

Theorem 6.2 *μ-UF (UF without repeated variables) is learnable from EQ and SQ.*

Proof: Because repeated variables are not allowed, each branch of a tree pattern can be generalized independently of the other branches in a bottom-up way, using the SQ oracle to decide when to stop generalizing (Figure 6.1). Generalization is performed by trimming the bottom level of leaves or 1-level subtrees under one node, replacing that node with a variable, and keeping only those changes which are accepted by SQ. The algorithm forms one hypothesis tree from each example. Compactness (Lemma 6.1) guarantees no superfluous trees are generated. The number of queries is bounded by the sum of all the nodes in the examples plus the number of tree patterns in the target. Hence the algorithm runs in polynomial time in the size of the examples presented. □

Corollary 6.3 *μ-UF is learnable from EQ and MQ.*

```
Initialize hypothesis h = {} %start with empty hypothesis
while EQ returns a positive example x
      %(SQ guarantees negative examples won't occur)
   t = x
   t = bot-up-generalize(x)
   h = h ⋃ t
return h

bot-up-generalize(s): %uses t as global variable
      % and s is a pointer within t
      % which is modified destructively.
   if s has children
      then for i = 1 to number of children
            bot-up-generalize(s_i)
         if all of the children of s are now variables
            replace s with a new variable
            if not SQ(t) then undo this change
      else if s is a constant
         change s to a variable
         if not SQ(t), undo the change.
```

FIGURE 6.1. μ-UF-EQSQ algorithm

Proof: We can use the same "width-doubling" or label-selection techniques as for OF with an unbounded number of children (Theorem 3.12). SQ is simulated using MQ by selecting successive constant subtrees to substitute for variables and rerunning the SQ-based learning until EQ does not return any negative counterexamples (which imply the simulation was imperfect since the SQ-based algorithm would never overgeneralize). These constant subtrees can start with width $w = 1$ meaning the trees have only one child. On each retry of the algorithm w would be doubled. The second and third variables in a tree pattern to be given to MQ would use constant subtrees with $w + 1$ and $w + 2$ children, etc. Labels could be substituted for variables starting with label number c (with numbers $c+1$ and $c+2$ for the second and third variables, etc.). Each time the simulation failed, increase c by the maximum number of variables in any tree pattern in which this substitution was used. □

Corollary 6.4 *μ-UT is learnable from SQ and one positive example.*

Proof: The one example can be generalized in the same bottom-up fashion. Further, SQ is required to always give correct answers, so EQ is not needed to assist in a simulation of SQ by MQ. □

6.3. Learning Unordered Trees with Superset Queries

This section describes a learning algorithm for unordered trees with onto semantics that uses equivalence and superset queries (or just one positive example and superset queries). The unordered tree learning algorithm (Figure 6.2) 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

```
% 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 6.2. SupQ-Based Algorithm for UT

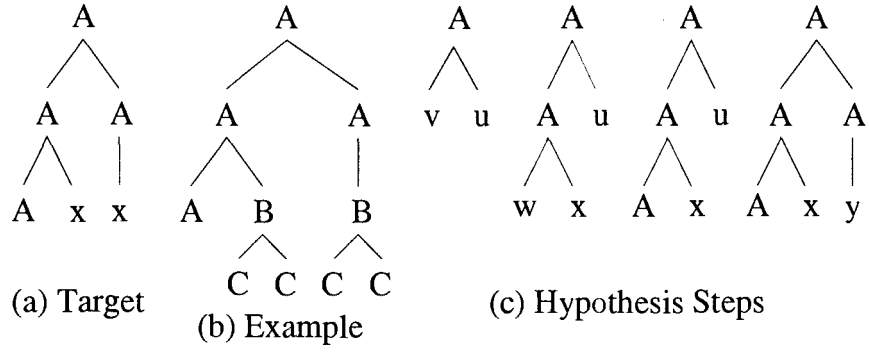


FIGURE 6.3. UT Learning Example

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 6.3, the training example (b) is generated for the target tree (a). The algorithm then forms a series of hypothesis trees (c) by successively specializing its current hypothesis. A simple variables hypothesis is first tried and then a tree is formed by copying the top level of the example and making each child be a distinct variable. Then subtrees are formed and specialized, and only 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 a hypothesis tree pattern independently of other parts by **grow-tree**.

Lemma 6.5 *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 7.5, $h \succeq t$. There is a substitution σ and mapping μ (by Definitions 7.2 and 7.1, respectively) 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 7.5.

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 i^{th} 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 one-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 $L(h) \supseteq L(t)$. □

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 (Figure 6.3 (c)), the algorithm makes the two variables x and y identical, 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 6.2). Any such identification of variables, once accepted by SupQ, need not be undone, as shown by the following theorem. When the algorithm terminates, the resulting hypothesis is equivalent to the target.

Theorem 6.6 *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 6.2). 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 6.5, 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 6.5, and the hypothesis is over-refined. Since the refinement of h occurs either by replacing a single variables with a constant or by adding one more level to a single subtree. 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-var**s 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 7.5. 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 7.5, and hence there is no such substitution. Since each change in each step 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 $n(n+l+v^2)$. □

6.4. Learning Unordered Forests from EQ and SupQc

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

Lemma 6.7 *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.*

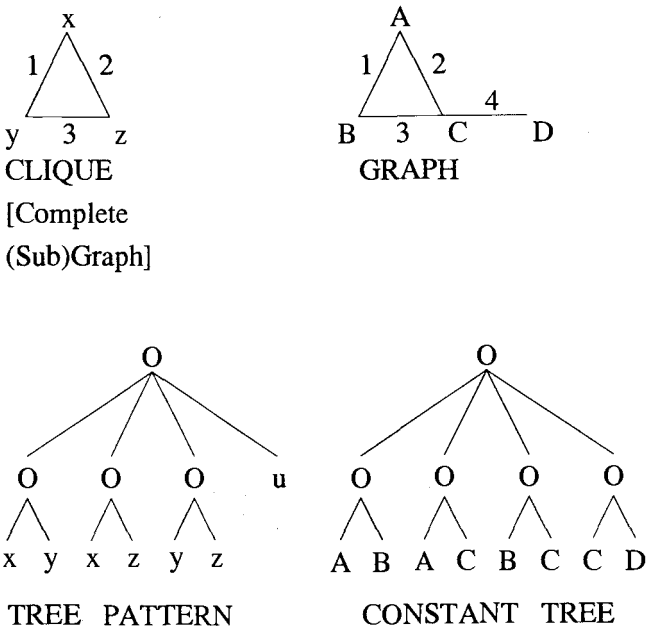


FIGURE 6.4. CLIQUE as Unordered Tree

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 label 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.

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. 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 6.4 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 & Johnson, 1979) except that UT trees are unordered whereas the arguments in the expressions or functions in the latter are ordered.

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. A version of the SupQ oracle which returns counterexamples is used:

Definition 6.1 *SupQc(F) is a (Forest) Superset Query (with counterexample) oracle which is given a forest (F) as an argument and returns either just "true" (if $F \supseteq$ target) or "false" with a counterexample.*

The biggest problem facing the algorithm (Figures 6.6 and 6.7) is that matching a counterexample to the hypothesis trees is too hard (Theorem 6.7). 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) making the hypothesis not cover the target. In effect, additional, specialized trees must added "blindly" until the result is again a superset of the target. 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 by EQ. 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 parallel/breadth-first fashion until a specialization which works is found.

The algorithm (initial phase in Figure 6.6 and main loop in Figure 6.7) will now be described in more detail along with a sample execution (Figure 6.5). The initial phase first checks for two default cases and then generates the one-level top covers of the target trees. The target could be either the empty/null or universal hypotheses, in which case the algorithm is done. If EQ returns a negative counterexample, showing the universal hypothesis (a single tree pattern variable) is overgeneral, that hypothesis is effectively "split" or specialized into a set of one-level tree patterns called top covers as follows ("TOPCOVER:" in Figure 6.6). The empty hypothesis is given to EQ, and a series of positive counterexamples are collected and converted a tree pattern with the same root

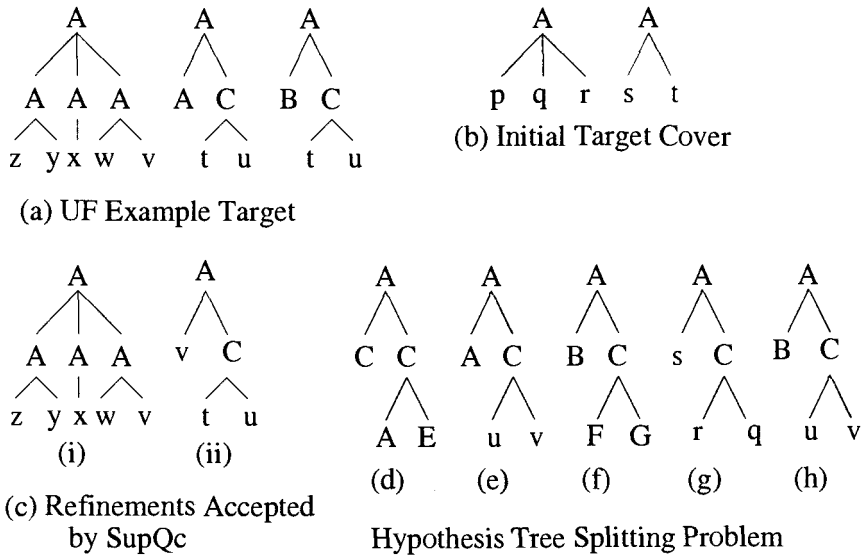


FIGURE 6.5. EQSupQc UF Example

label and number of children as the top level of the example, but with a distinct variable in place of each subtree. For example, let the target be Figure 6.5(a), then the top covers are Figure 6.5(b). SupQc can then be used to guide the specialization of each such tree as much as is possible without adding any new trees but keeping the result a superset of the target ("SPECTOP:"). This step yields the hypothesis in Figure 6.5(c) for the sample execution. These trees are the hypothesis at the end of the initial phase and the start of the main loop (Figures 6.6 and 6.7).

Each main loop execution ("MAINLP:" through "FINALSPEC:" in the Figure 6.7 code which is all of that figure except the final return statement) successfully performs a 1-level specialization on one of the tree patterns t_i in

```
Initialize hypothesis h = {}  %start with empty hypothesis
If SupQc(h) returns 'true' then return(h)
      else save x = counterexample
h = v  %universal hypothesis (single variable)
If EQ(h), return(h)   %keep the simpler, top-level
TOPCOVER:                % tree split separate from main loop:
specialize h to top level of x %(root label+number of children)
while SupQc(h) gives example x  %until cover target
   h = h ⋃ top level of x
  % Also keep track of examples x_i used to create t_i to guide spec-
  % ialization of those trees later. At this point, h=all root label-#
  % of children combinations that are needed to cover the target.
SPECTOP:
for each tree t in h
   while SupQc(h) returns 'true'
      specialize all parts of t as guided by corresponding example
         (undo changes not accepted)
```

FIGURE 6.6. Initial part of Algorithm EQSupQc for Learning UF

the current hypothesis h. Where a 1-level specialization is the minimum amount a tree can be specialized and still be changed to represent a proper subset of the previous tree. That is, either expand a variable leaf by changing it to a label with children–each of which is a variable or make two leaf variables identical. These specializations are guided by the example originally used to form each tree, and the algorithm keeps track of this hypothesis tree-example correspondence, although this aspect is not explicitly shown in the pseudocode. Specializations are tested by by SupQc at several points in the code. The

MAINLP: %Main loop does additional tree splits:
while not EQ(h) %(negative example–need to split some tree)
define t_i so $\bigcup_{i=1}^{r} t_i = h$ %(i.e., h is trees $t_1 \ldots t_r$)
for $i = 1$ to r
$g_i = h - t_i$
ALLSPEC:
for j indexing all 1-level specializations of t_i based on x_i:
$h_{ij} = g_i \bigcup$ that specialization
COVER:
for all h_{ij} in parallel:
repeat %add trees to each h_{ij} until one covers target:
$x_{ij} = \text{SupQc}(h_{ij})$
%make u_{ij} cover as well as t_i does by:
u_{ij} = as specialized as possible based on x_{ij}
while $g_i \bigcup u_{ij}$ still covers target
SPECCOVER:
$h_{ij} = h_{ij} \bigcup$ all 1-level specializations of u_{ij}
until $\text{SupQc}(h_{ij})$ is true for one of the h_{ij}
PRUNE:
for all trees v in that $h_{ij} - g_i$ %eliminate unneeded trees:
if $\text{SupQc}(h_{ij} - v)$ $h_{ij} = h_{ij} - v$
FINALSPEC:
for all trees v in g_i- that h_{ij} %and generalizations:
specialize v as long as SupQc accepts h_{ij}
$h = h_{ij}$
Return(h)

FIGURE 6.7. Main loop of Algorithm EQSupQc

main loop continues until EQ indicates no more specialization is needed. Each potential specialization (tree t_i in the current hypothesis h and node within t_i) and its corresponding resulting hypothesis h_{ij} is refined separately and in parallel until one succeeds in covering the target again with a more specific hypothesis. This refinement process gives each h_{ij} repeatedly to SupQc to get additional examples x_{ij} which are converted into an additional specialized trees that are combined with h_{ij}. These additional trees fill the gap created by the specialization and eventually cover the target again. The parallelism is needed because the learner has no way to tell beforehand which specialization will succeed and how many trees need to be added to the corresponding h_{ij}.

Each iteration of the main loop starts with EQ returning a negative counterexample, but the learner doesn't know which tree is overgeneral or how the tree should be specialized to avoid covering the example. The code at the beginning of of the main loop ("MAINLP:" through "ALLSPEC:" in Figure 6.7) therefore tries all possible specializations of each hypothesis tree t_i (as guided by the examples originally used to form them). For each t_i, g_i represents the rest of the hypothesis h, and h_{ij} is g_i combined with the jth specialization of t_i. In the Figure 6.5 sample execution, (c) is h, (ii) could be labeled t_2; since there are only two trees in h, g_2 is just (i), and suppose (d) is the negative counterexample (not shown in code) from EQ at "MAINLP:". Then "ALLSPEC:" then generates (e) as a specialization of (ii) which represents h_{21} when combined with g_2 = (i) and is the one of many specialized versions of t_2 which eventually yields a useful result.

When a possibly overgeneral tree t_i is specialized to form a hypothesis h_{ij} and a new example x_{ij} is returned which represents the part of the target now uncovered, the algorithm doesn't know how to change that example into a new tree pattern to help fill in the gap between h_{ij} and h. Further, even the problem of discovering how the example is related to t_i is complicated by the fact *matching* a tree pattern to an instance is *NP-Complete*. In the code,

"COVER:" and "SPECCOVER:" call SupQc to obtain an example x_{ij}. This example is generalized to u_{ij} just enough to cover part of the target as well as t_i does with the help of g_i (i.e., each leaf is trimmed bottom up until $g_i \cup u_{ij}$ covers the target, then each part of u_{ij} is specialized as long as this union still covers the target). Next all possible 1-level specializations of u_{ij} consistent with x_{ij} are combined into the common hypothesis h_{ij} (rather than being maintained separately as in the other specialization loop). This covering phase loop repeats in parallel with each thread building its own hypothesis h_{ij} until one of the h_{ij} successfully covers the target again. In the Figure 6.5 sample execution, (ii) is a hypothesis tree which is more general than the target, but any one value for the left child is too specific to cover the target; therefore an adequate hypothesis can't be generated by generalizing or specializing this one tree. A step we call *hypothesis tree splitting* is needed. Tree (f) represents the counterexample x_{21}, (g) represents the generalization to u_{21} which covers the target with the help of g_1 (i) as well as t_2 (ii) does. Tree (h) is the one specialization of (g) which eventually proves useful. Hypothesis tree (ii) is then effectively split into (e) and (h).

If the hypothesis generated from the previous step is used without change in succeeding iterations of the main loop, its size will grow exponentially with executions of the main loop. But most of the trees that were added to the hypothesis are not useful and must therefore be pruned, and then the remaining trees are specialized as much as SupQc will accept. This step will be called the *pruning phase* in the proof section below. The code in "PRUNE:" therefore tests each tree in the successful hypothesis h_{ij} to see if it really needed. If elimination of that tree still leaves h_{ij} a superset of the target, then that tree is dropped. Then "FINALSPEC:" makes each tree as specialized as possible (again guided by the example used to produce each tree) while h_{ij} is still a superset of the target, Finally, h_{ij} is made the new hypothesis h and the main loop repeats. These steps are not explicitly shown in the sample execution. However, several

specializations of the Figure 6.5(g) tree are created and added to h_{21}. Each leaf in the former can be changed to look like example (f). But trees with constants in place of the r and q leaves in (g) either do not help to cover the target or are redundant. These other trees are therefore pruned. In this example, no further specialization of (g) is needed, but some learning problems will need specialization similar to that which was done to get both trees in (c) from (b).

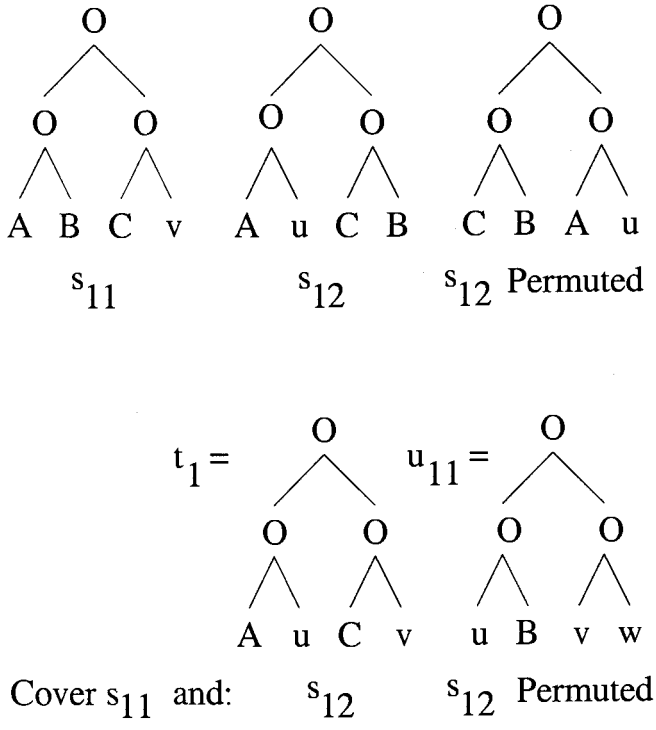


FIGURE 6.8. Two Different least-general generalizations Covering Examples s_{11} and s_{12}

The following lemma shows that the justification of algorithm EQSupQc is not quite straightforward. It might seem at first that u_{ij} in the algorithm

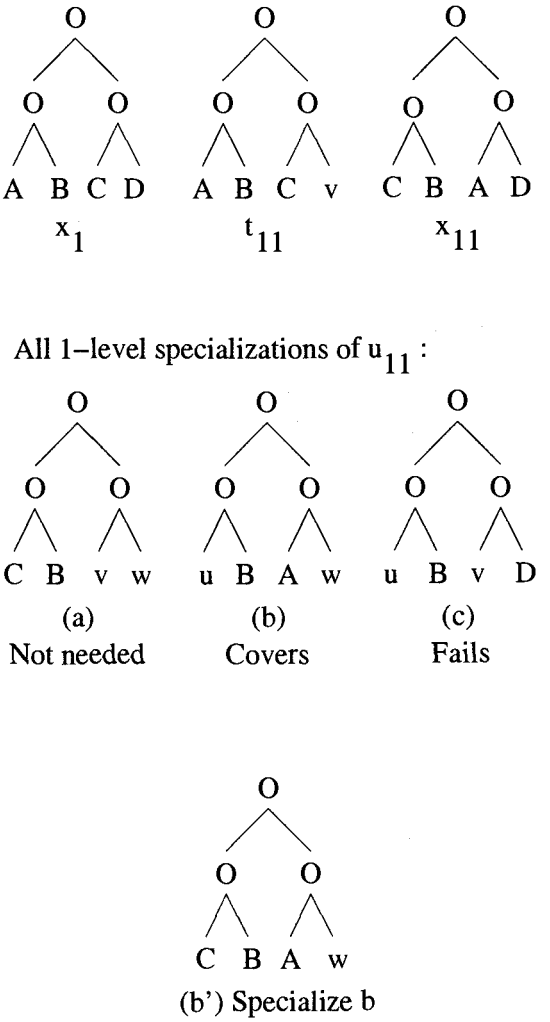


FIGURE 6.9. EQ and SupQc Algorithm Covering/Pruning

(Figure 6.5(g)) would always cover (matches) the same set of instances as t_i (Figure 6.5(c)), since u_{ij} was generalized from the counterexample generated by the removal of t_i from a hypothesis covering (representing a superset of) the target. But as the lemma below shows, this assumption is incorrect. A more flexible proof which takes this discrepancy into account must therefore be found.

Lemma 6.8 *The least-general unordered tree pattern that covers a set S of (unordered) tree instances is not unique.*

Proof: Let the example trees be as in Figure 6.8. Two examples are used, and a permuted version of the second is shown. The first tree is then generalized with each permutation of the second, giving the two tree patterns below. Then, depending on how the children in the two trees are paired, 2 different *minimal* covering generalizations are possible. These covers are generated by taking the *ordered lgg* of the two tree patterns. □

A detailed justification of the remainder of algorithm EQSupQc follows. It is split into two major parts: the covering and pruning phases. This is followed by a correctness proof of the algorithm (main theorem). Where appropriate, notation is used which corresponds to the variable names used in the pseudocode. The notation for the first tree is used (t_1 and u_{11}) but the lemmas apply equally well for all instances. Note that there could be other parts of the hypothesis h (i.e., $h - t_1$). This remaining hypothesis part is not shown and will not affect the following because it will be kept covered on all SupQc queries.

Covering Phase: The set $S_1 = \bigcup_f s_{1f}$ of trees represents the target trees left uncovered when t_1 is eliminated from the hypothesis (this set is therefore covered by t_1 as well as by u_{1j} for each j). This set and the variables s_{1f} are different from all previous variables mentioned in the algorithm, hence the use of different names (i.e., S is not necessarily the same as any $\bigcup_j t_{1j}$ where the t_{1j} represents the trees generated by splitting hypothesis tree t_1). The following

lemmas assume the target is expressed in minimal form in the sense that no target trees represent a subset of other target trees. First it is shown the fact that u_{ij} is not the same as t_i does not matter.

Lemma 6.9 *Given a nonempty subset $\{t_i\}$ of the hypothesis trees in h, each of which covers more than one target tree, then the covering phase of the EQ-SupQc algorithm will discover a way to split one of the t_i while still maintaining hypothesis h as a superset of the target.*

Proof: For each hypothesis tree t_i in the current hypothesis h, let $S_i = \{s_{ik}\}$ represent the set of target trees t_i covers which are not completely covered by $g_i = h - t_i$. The algorithm derives u_{ij} from counterexample x_{ij} to cover S_i (with the possible help of g_i). But by *compactness* of UT and the fact none of the s_{ik} are completely covered by g_i, u_{ij} covers all of S_i.

For each i with t_i covering more than one target tree, some 1-level specialization of covering tree pattern u_{ij} separates the set of target trees into at least two groups by the following argument. There is more than one target tree in S_i, u_{ij} covers all of them, and none of these target trees are redundant. Therefore, all the s_{ik} represent *proper subsets* of u_{ij}. Since u_{ij} is a proper superset of each s_{ik}, and therefore the former matches the latter, the top-level structure of that target tree is the same as u_{ij}. A given s_{ik} is therefore a specialization of u_{ij} in some leaf l of the latter, so some 1-level specialization of u_{ij} in l will still cover s_{ik}. The s_{ij} are not all specialized in l or u_{ij} could have been specialized in l and still cover all of S_i. Therefore specializing u_{ij} in l will still make it cover s_{ik} but not some other tree in S_i.

In the parallel covering loop, each thread is building a particular hypothesis h_{ij}. The specializations generated from the corresponding u_{ij} from each iteration will cover at least one target tree by the above argument. Some of the threads will attempt to split a single target tree and would therefore never terminate. But the negative counterexample obtained at the beginning of the main loop guarantees some thread will eventually be successful after a finite

number of cover-loop iterations. Therefore, one of these threads will eventually create a more specific hypothesis more specific than the previous h which covers the target and the hypothesis h_{ij} assigned to that thread will be reassigned to h. □

The algorithm does not specifically try to find all configurations of a specific subtree (e.g., like the left child of the right two trees in Figure 6.5(a) having values A and B). Instead, each iteration uses a divide-and-conquer approach to split the set of remaining target trees (when there are more than one) and guarantee at least one more tree is covered. Even within one thread, successive iterations of the cover loop are likely to split the set of remaining target trees on different parts of the trees.

Pruning: The above lemma shows that the target really does get covered with more specialized trees (the 1-level specializations of u_{11}). But there is still a need to show the number of these trees does not become excessive and threaten the polynomial bound.

Lemma 6.10 *After the algorithm's pruning phase, for every target tree pattern in S, there is at most one hypothesis tree pattern in h_{ij}.*

Proof: Recall that the pruning phase tries eliminating one of the specializations of u_{ij} at a time from the hypothesis and restores it only if the target becomes uncovered. Those 1-level specializations can be separated into two classes: those that completely cover one of the target trees and those that don't. By (the contrapositive of) *compactness* (lemma 6.1), a collection of specializations each of which only covers part of the target tree can't completely cover that tree. The latter class of specializations will be pruned from the hypothesis because there must be others which cover the target. Each hypothesis tree that completely covers a target tree will be kept if and only if it is the only remaining tree to do so. Therefore, for each target tree, all but one of the hypothesis trees which cover it will be pruned.

Hence, the number of remaining hypothesis trees after pruning will be bounded by the number of target trees. □

Example: Figure 6.8 is used again for the first steps demonstrating the above aspects of algorithm EQSupQc and Figure 6.9 completes the example. Assume that $s_{11} \bigcup s_{12}$ is the remainder of the target yet to be covered (there might be other trees not shown in this example). Let t_1 (middle bottom of Figure 6.8) be the tree pattern used to cover this part of the target during an earlier phase of the algorithm and x_1 be the example from which t_1 was generated. The algorithm then creates all possible 1-level specializations of t_1 and tests them separately. For the rest of this example, we consider just one of those specializations: t_{11} (top of Figure 6.9). SupQc is then called with $h - t_1 \bigcup t_{11}$ and the example x_{11} is obtained (which is therefore covered by t_1 but not t_{11}).

This example is then used to guide the top-down formation of a pattern to cover the target $(s_{11} \bigcup s_{12})$ and the result could be u_{11}. Notice that the latter has a different number of variables than t_1 even though it is intended to cover the same set of trees as the latter–and therefore looks like it could cause trouble. But the algorithm handles this supposed difficulty perfectly. Next, all possible one-level specializations of u_{11} (as guided by an example–not shown–whose children are CB and AD) are created (trees a, b, and c in Figure 6.9). This collection of specializations succeeds in covering the target showing that t_{11} was a successful specialization choice (the other specializations not shown would be unsuccessful). The pruning phase then decides which of these trees are not needed when used with t_{11} to cover the target; s_{11} is covered by t_{11}, so the new trees only need to cover s_{12}. There is no problem if (a) is eliminated because (b) still covers s_{12}. If both (a) and (b) are eliminated, the target is no longer covered, so (b) or at least one of these two must be retained. Tree (c) fails to cover s_{12}, so it is useless and is eliminated. Finally, the remaining tree (b) is specialized as much as possible while keeping the target covered (b'). Now we are ready to prove the main result of this section.

Theorem 6.11 *Unordered forests with one-to-one onto semantics are learnable from EQ and SupQc.*

Proof: The separate first/top-level iteration ("SPECTOP:") pass is guaranteed to find all the root-number-of-children combinations in the target. "MAINLP:" is guaranteed to cover the target with a more specialized set of trees on each iteration by lemma 6.9 and to prune the result to something no more complex than the target by lemma 6.10. Eventually no more specializations will be possible while keeping the target covered and the algorithm will terminate by EQ returning true.

Bounds: Learning starts with the "TOPCOVER:" phase, which creates no more hypothesis trees than target trees. These trees are supersets of target trees so they have no more leaves than the target. The initial hypothesis is therefore bounded by the target size.

The dominant bound for the number of queries superficially at first glance seems to be determined by two sets of 3-level loops (Figure 6.7). The main loop "MAINLP:" with "while not EQ(h)" is the outermost loop for both sets of 3-level loops. The first set of loops is the specialization part of the code (first part of the main loop and "ALLSPEC:") with loops "for i" and "for j". The second set of loops is "COVER:" and "SPECCOVER:". But these two sets of loops are not really disjoint/independent even though they superficially appear to be in the code. The second set operates in parallel for all i and j which were created by the first and therefore inherits the complexity of the first. Further, the second set also has a repeat ... until loop and the loop implied by "all 1-level specializations of u_{ij}". On this basis, it could be argued that the second loop set is really a 5-level loop. But even these loops do not quite reflect the actual execution time of the algorithm because the number of possible 1-level specializations is not linear in the tree size–as will be explained below. There is also another loop level which examines each node within a tree. "PRUNE:"

and "FINALSPEC:" do not have as many nested loops and therefore do not affect the order of the execution time.

Let f be the total number of nodes in the target forest, and t be the number of nodes in the largest target tree. The number of possible 1-level specializations (i, j index loops) will be determined by the total number of leaves in the target. The first of three ways of specializing a tree is to turn a leaf into a one-level subtree (as guided by the example used to form that tree). A second mode of specialization turns a variable leaf into a constant. But this specialization is mutually exclusive of the above one because the specialization must be guided by an example and therefore has only one choice. A third specialization mode is to make two leaf variables identical. This last specialization mode dominates because it depends on pairs of leaves rather than just a single leaf. The dominant term in the bound on the number of specializations ("j loop") is therefore $t^2/2$ for an individual hypothesis tree or $ft/2$ for the target forest.

By an argument like the *metric* for OT (Theorem 3.7), the number of iterations of "MAINLP:" is bounded by f. The "for i" loop iterates over the hypothesis (and potentially all target) trees. "ALLSPEC:" (for j) will potentially try $t^2/2$ possible specializations for an individual tree, giving a total factor of f times $ft/2$ or $f^2t/2$ so far. The "COVER:" loop potentially executes as many times as target trees, and the "SPECCOVER:" loop could call SupQc for each pair of nodes in the u_{ij} giving a bound of $ft/2$ again, and an overall bound of $f^3t^2/4$.

By lemma 6.10, the hypothesis after pruning will be maintained with a complexity less than the target so the above bounds will hold with successive "MAINLP:" iterations. The overall queries bound would therefore be of order $O(f^3t^2)$. Execution time could be that quantity multiplied by tree size: $O(f^3t^3)$.
□

6.5. Learnability of UF Using SupQc and SQ or MQ

The availability of SQ allows the learner to directly test which trees are overgeneral thus significantly simplifying the algorithm for learning unordered forests when SQ and SupQc are available. This test avoids the need for parallelism in the covering loop. But the specialization step and covering loops are still needed. The combination of SQ and SupQc allows the learner to achieve the equivalent of an EQ test except for negative counterexamples. But SQ can be used to test each tree so the lack of negative counterexamples causes no difficulty. The process of splitting an overgeneral tree is still somewhat complicated.

Theorem 6.12 *UF is learnable from SQ and SupQc.*

Proof: The learning algorithm for this class will start with a null/empty hypothesis (no trees). Then a topcover is formed by repeatedly calling SupQc with the current hypothesis and creating trees from the top level of the example. (These trees are therefore as general as possible without covering the entire instance space.) After SupQc verifies the target is covered, each tree is specialized as long as it is accepted by SupQc. The part of the algorithm up to this point is the same as the initial part of algorithm EQSupQc.

SQ is then used to test which hypothesis trees are still overgeneral. Those that are overgeneral are specialized/split by trying the following steps. First, an overgeneral hypothesis tree t_i is replaced by all of its 1-level specializations. If the result is a superset of the target, prune the specialized trees which are not needed and repeat these steps. Otherwise generate new trees using a process very similar to the covering loop of the EQSupQc algorithm. These steps are repeated until SQ verifies each tree is not overgeneral and SupQc verifies the target as a whole is a superset, proving equivalence of the hypothesis and target. □

The following lemma shows that the MQ and SupQc combination of queries (unlike SupQc and EQ or SQ) lacks the capability to tell the learner when it has found the correct target.

Lemma 6.13 *UF is not learnable from SupQc and MQ.*

Proof: Any learner with these two oracles cannot correctly know whether the target is the universal set, which is represented by a tree with a single node which is a variable, because there will always be other targets which are also consistent with the information that the learner has received. Specifically, if the learner reports that it has learned the universal set (one possible target), then all of the calls to MQ must have resulted in the responses of "*true.*" Further, the calls to SupQc can only have resulted in a *yes* or in a positive example. While the universal hypothesis is consistent with their data, another target consisting of all received positive examples is also consistent with this data. If the learner reports either one of these two targets as its hypothesis, the teacher can always choose the other, and the learner will have failed to learn the correct target. □

6.6. Learnability of Bounded Unordered Trees and Forests

This section discusses the learnability of UT and UF with a finite label alphabet and a bound on the number of children below each node. For all of the learning problems given so far, a training example (or a source of examples) is necessary. This claim applies in general whenever an infinite label alphabet is used because then the learner can't guess a label in a finite number of queries. It is also true when subset (rather than superset) queries are used because the former requires learning to be bottom up and must have an initial example as a starting point (e.g., the algorithm for μ-UF, Theorem 6.2).

But there is an exceptional concept class which does not require any training examples. Adapting Definition 4.3 of Section 4.2, this class can be

called $UT_{l,b}$, meaning the concept class of (single) unordered trees with a label alphabet of l symbols and a bound b on the number of subtrees below any node (with both l and b finite).

Theorem 6.14 *$UT_{l,b}$ is learnable with SupQ and no training examples.*

Proof: The algorithm for this class works the same as for the UT algorithm, except where the single example is used as a guide for specializing the hypothesis. Instead of using an example as a guide to refine the hypothesis, this algorithm tries out all single-step refinements exhaustively until one succeeds. In other words, it tries all $l(b+1)$ possible ways of refining a given node by replacing a variable with any of l labels and from 0 to b children (with new variables as the children), tests the result with SupQ and keeps any changes that are accepted. It is easy to see that when the algorithm terminates, it would have exactly learned the target tree. □

The problem of learning unordered forests with a bounded label alphabet and number of children using superset and equivalence queries is similar to the unbounded alphabet and children case; but there are additional, subtle problems.

Open Question 5 *$UF_{l,b}$ is learnable with equivalence and superset queries where both return counterexamples ?*

A proof might be attempted along these lines: The bounded class is not *compact* (Definition 3.5). but that might not matter because The algorithm has to maintain a hypothesis which represents a superset of the target and which is no more complex than necessary. It would be highly desirable to guarantee there are no more hypothesis trees than target tree patterns.

The version of the algorithm for unbounded trees tries specializing a particular node in the hypothesis (say p) and then adds more tree patterns until

the target is covered. When the alphabet is infinite, this approach necessarily only succeeds when the corresponding target tree is more specialized than the hypothesis. But with a bounded alphabet and children, a finite set of specialized trees could cover the same instances as the original tree–if there are at least $l(b+1)$ trees. The result would make the hypothesis too complex. The algorithm for $UF_{l,b}$ must be adjusted to stop just before this happens. There is still another problem; the target could include all $l(b+1)$ refinements of a particular hypothesis node–but require some of them to be further specialized. This possibility requires the algorithm to try generating all specializations of node p and then exploring further specializations of the resulting trees. Since the algorithm should not use exploration this complex unless it is really needed, refinement to greater depths would not be used until all lesser choices are exhausted.

The problem is that the proof for EQSupQc requires compactness to bound the size of the hypothesis. A learning algorithm for $UF_{l,b}$ needs to bound the complexity of its hypothesis which essentially needs the hypotheses to be restricted to those which don't violate the compactness property. That is, no set of hypothesis trees should include all $l(b+1)$ values of node label and number of children for a particular tree node.

Presumably if $UF_{l,b}$ could be learned then DNF could also be learned with EQ and SupQc. But the problem of deciding whether a DNF hypothesis violates a compactness property is equivalent to deciding whether the hypothesis covers all instance values represented by a particular conjunct. The latter problem is equivalent to the dual problem of finding satisfying instance for a CNF expression and therefore to solving *SAT*–which is *NP-Complete*.

6.7. Summary

Unordered trees without repeated variables are learnable from one example and subset queries. Unordered trees and forests without repeated variables are learnable from equivalence and either subset or membership queries.

Like the class of ordered trees, the class of unordered trees is compact. But unlike for ordered trees, matching for unordered trees with repeated variables is NP-Complete. Unordered trees with repeated variables are learnable from superset queries and one positive example. Unordered forests with repeated variables are learnable from superset and equivalence queries–both with counterexamples. Unordered forests are learnable from superset queries with counterexamples and subset queries. But unordered forests are not learnable from superset queries with counterexamples and membership queries.

Unordered trees with repeated variables and with a finite label alphabet and bound on the number of children are learnable from superset queries (without counterexamples) and no training examples (or even a source of examples). The question of whether unordered forests with a finite label alphabet and bound on the number of children is learnable from equivalence and superset queries with counterexamples is an open problem. These results seems to suggest that superset queries are at least in some sense more powerful than subset queries. This distinction is really tied to the conjunctive nature of tree matching (see the discussion in Section 9.1).

7. LEARNING UNDER ONE-TO-ONE INTO SEMANTICS

Chapter 5 showed that learning unordered trees with *one-to-one onto* semantics using only EQ and either SQ or MQ is hard. In Chapter 6, we say that learning this class is easy with superset and equivalence queries. But suppose superset queries were unavailable or not practical? The difficulty of learning unordered tree patterns can also be circumvented by using *one-to-one into semantics.* In this chapter, we describe a bottom-up algorithm for unordered forests with one-to-one into semantics and give an analysis of the algorithm.

7.1. Introduction

First-order predicate *Horn clauses* can be learned without superset queries even though there is no constraint on the order of the predicates within the clause (Reddy & Tadepalli, 1999). This capability contrasts with the above limitations in learning unordered tree patterns with onto semantics. Yet there seems to be a connection between tree and predicate learning. Both have variables which match structures and under certain conditions can use algorithms such as lgg. But Horn-clause learning allows extra literals in an instance matched by a predicate clause. One-to-one onto tree matching is quite rigid in comparison. To get a better comparison, tree matching is altered to allow extra children in an instance matched by a tree pattern. This changes the mapping of the children of a tree pattern node to the children of a tree instance node from one-to-one onto to one-to-one into.

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, & 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 et al., 1999) as will be shown below.

Unfortunately, the lgg algorithm as used with Horn clauses still won't work with unordered trees even with this altered semantics. A different algorithm similar to the algorithm for μ-UT (Theorem 6.2) will work. This algorithm starts with one example target tree to be learned and repeatedly tries all possible ways of generalizing the resulting pattern until no further generalization is possible. The result is one of the target tree patterns. This algorithm depends on there being only polynomially many ways to generalize a tree pattern at each step. This property is not true for one-to-one onto semantics because the number of ways to generalize a hypothesis can be exponential for repeated variables/constants. But a variation in the approach makes an end-run around this obstacle possible. The algorithm still uses a bottom-up generalization approach from a single example. But instead of generalizing directly from a tree pattern with repeated variables, the algorithm first makes the pattern more specific by adding extra subtrees to a node. Then further generalization becomes possible.

7.2. General Definition of Tree Matching Semantics

The standard, most-often used mapping is called one-to-one onto which means that each (one) pattern child maps *to* exactly *one* instance child, and *onto* requires that all instance children are included in this mapping. Further, no instance child is mapped from more than one pattern child. This mapping therefore requires that the numbers of pattern and instance children are the same; the mapping is a *permutation* if it is also allowed to be *unordered* (or is the identity mapping if it is required to be *ordered*).

How do tree patterns represent/correspond to expressions? If an expression is one of the tree instances represented by a tree pattern, then each node in the pattern must correspond to some node in the instance. The children of the node in the pattern must correspond to the children of the corresponding instance node in some fashion. This correspondence will be called *matching semantics* and will be defined by the type of mapping used. The simplest mapping type is *one-to-one onto*. But difficulties with learning with this semantics and the desire to compare with other logical formulations inspired a study of one-to-one into and many-to-one into semantics, both of which allow an instance to have more children than a matching pattern. One-to-one into semantics is the same as one-to-one onto except that the requirement that all of the instance children take place in the mapping is removed.

The semantics are classified according to the kind of mapping allowed between subtrees of corresponding trees. In *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*.

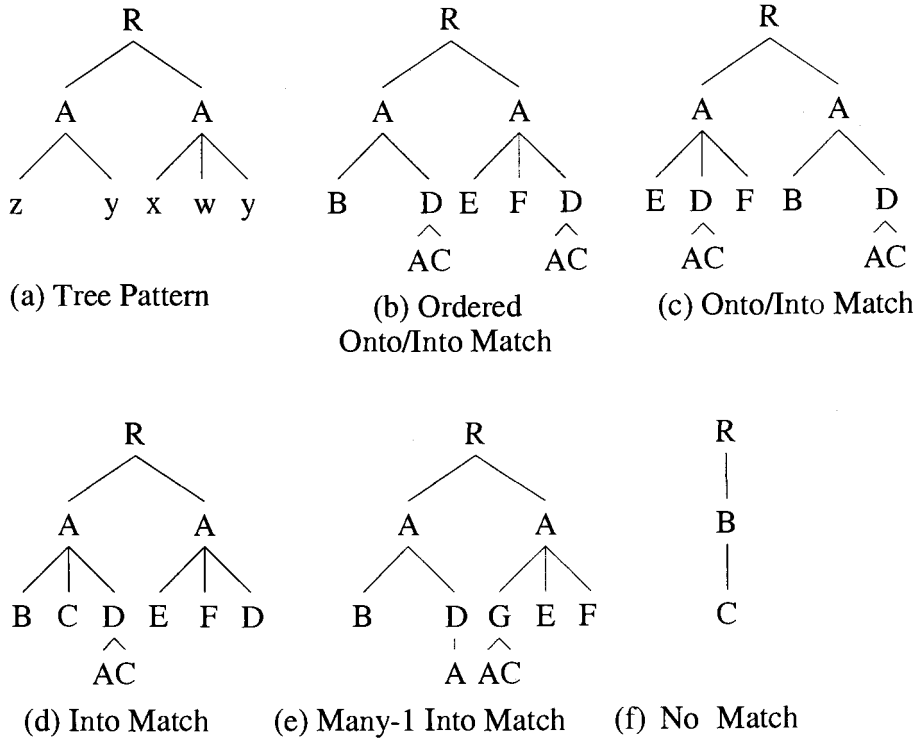


FIGURE 7.1. Match Semantics Example

In Figure 7.1, 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, $D(A\ C)$ 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 (many- or one-to-one) into semantics. But this instance is not matched using one-to-one onto semantics because of the extra leaf C under the left 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 with no children and $D(A\,C)$). Tree pattern (a) does not match instance (e) by one-to-one onto/into semantics because there is no subtree with ≥ 3 children 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) and 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., $R(A($ 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]

For ordered tree matching, just a simple substitution is needed for matching. 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 7.1 *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)*

---

[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.

$r_0 = s_0$, *(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* $1 \leq j \leq l$*). Where* $r_i\mu = s_j$ *and* $r\mu = s$*, and we 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 ($\text{label}(r_f) = \text{label}(s_g)$, meaning r_f and s_g are the same constant–or variable).

Definition 7.2 *A tree pattern* Γ matches *a tree instance (or pattern)* t *according to a given semantics* Ψ*, denoted by* $\Gamma \succeq_\Psi t$*, iff (*t *is empty hypothesis* ϕ *or) there is a substitution* σ *for* Γ*'s 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.*

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 7.3 $L_\Psi(P)$ *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 (i.e., apply in left to right order).

Definition 7.4 *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* $\sigma_1 \circ \sigma_2 = \{x_1/s_1\sigma_2, \ldots, x_n/s_n\sigma_2\} \bigcup \{y_i/t_i | y_i \notin \{x_1, \ldots, x_n\}\}$.

Lemma 7.1 *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 7.2 *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 type Ψ as well. □

Lemma 7.3 *For every pattern* P, *substitution* σ, *and* Ψ*-consistent mapping* μ *there exists a* Ψ*-consistent mapping* μ' *such that* $P\mu\sigma = P\sigma\mu'$.

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 $P\sigma\mu' = P\mu\sigma$. □

The lemma implies a string of substitutions and mappings can be reordered with all substitutions before all mappings (using different mappings). The reverse direction $P\mu'\sigma = P\sigma\mu$ doesn't work because μ could map substitutions for identical variables to nonidentical subtrees.[2]

---

[2]For example, given $R(x\ x)\sigma\mu = R(A(B)\ A(B))\mu = R(A(B)\ A(B\ C))$. Then σ can't produce both $A(B)$ and $A(B\ C)$. $R(x\ x)\sigma\mu = R(A(B\ C)\ A(C\ B))$ shows one-to-one onto semantics can have the same difficulty.

Lemma 7.4 *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 7.3. Further, $P(\sigma \circ \sigma')(\mu'' \circ \mu') = R$ by Lemmas 7.2 and 7.1. Finally, $P \succeq R$ by Definitions 7.1 and 7.2. □

Theorem 7.5 *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 7.4, and $I \in L(P)$. Hence $L(P) \supseteq L(Q)$.
only if: Given $L(P) \supseteq L(Q)$. Since the alphabet of constant labels 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 7.3, there is a mapping μ'' so $P\sigma'\sigma^{-1}\mu'' = P(\sigma' \circ \sigma^{-1})\mu'' = Q$. Hence, $P \succeq Q$. □

Trees are compact for all matching semantics:

Lemma 7.6 *The classes of (Ordered or) Unordered tree patterns (UT) with one-to-one onto, one-to-one into, many-to-one onto, 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 7.3, $\mathcal{V}_i\sigma'\sigma^{-1}\mu = \mathcal{Z}$ for another mapping μ. Substitution composition via Lemma 7.1 gives $\mathcal{V}_i(\sigma' \circ \sigma^{-1})\mu = \mathcal{Z}$ or $\mathcal{V}_i \succeq \mathcal{Z}$ for some i. By Theorem 7.5, $L(\mathcal{V}_i) \supseteq L(\mathcal{Z})$ for some i. □

7.3. One-To-One Into Algorithm Description

The main part of the unordered forests learning algorithm for one-to-one into-semantics using EQ and SQ (equivalence and subset queries) is based on a bottom-up generalization approach from single examples and is shown in Figure 7.2. 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 instance 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 that corresponds 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 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 7.2. 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

```
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 7.2. Into-Semantics Main (left) and Pruning Routines (right)

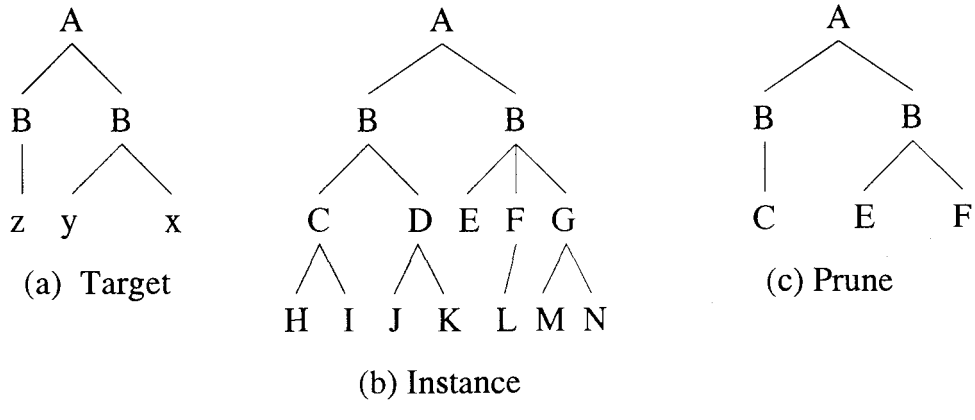


FIGURE 7.3. Bottom-Up Pruning Sample Execution

for pruning. The process repeats as long as pruning yields a tree pattern that is accepted by SQ.

Figure 7.3 illustrates how the simple pruning technique works. A possible target is given in (a) and single training example could be as in (b). The bottom-up pruning 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 7.4) 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 acceptable by SQ. 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 step repeats until partitioning or variablization is unsuccessful on each remaining non-singleton set of identically labeled constants or variables. The actual turning of constants into 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 (called c's below and in routine **variablize**) 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 5.5). 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

```
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 7.4. UF 1-to-1 Into Repeated Variable Partition Algorithm

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 particular 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 7.4) 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.

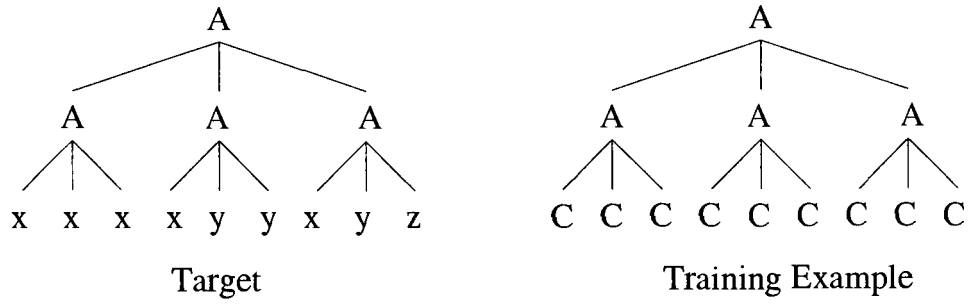


FIGURE 7.5. Partition/Repeated Variables Bottom-Up Example

Figure 7.5 shows the start of 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. 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 $CCCrrr\ CCCrrr\ CCCrrr$ (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 a C, then one of the r's, yielding $CCrr\ CCCrrr\ CCCrrr$. 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 $Crr\ CCCrrr\ CCCrrr$. 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 *Crr Crr CCCrrr*. Eliminating one of each from the third subtree gives *Crr Crr CCrr*, but no subtree can match the 3 x's. Backtracking and eliminating r's gives *Crr Crr CCC*. Another call to partition the C's changes all C's into a variable, say s, giving *srr srr sss*. 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*).

7.4. Proof of Algorithm

We will now give a correctness proof of our algorithm. The proofs cover the following three tasks: prune all leaf nodes as much as possible, convert constant leaves to variables, and partition sets of identical constants or variables as much as possible.

Theorem 7.7 *After* **prune** *(Figure 7.2), the hypothesis tree will be isomorphic to the target tree (aside from repeated variables).*

Proof: Since the target matches the hypothesis, there is a corresponding σ and μ. If there are extra nodes/edges that do not have maps, they would be removed by **prune**. 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 7.5. 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. It follows that there is a correspondence between $\mathcal{T}$ and p represented by the mapping μ of Definition 7.1, and this correspondence is one-to-one onto (otherwise, the extra subtrees permitted by the into mapping would have been removed by routine **prune**). Therefore, **prune** removes all the nodes which are not needed to correspond to and be matched by some target tree. □

Lemma 7.8 *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 as arguments, routine* **partition** *will generalize the set of o's in t if that set is less general than some target tree.*

Proof: The generalization occurs by either turning constants into variables or by splitting a set of repeated variables into sets of two or more distinct variables (or by splitting a set of repeated constants into sets of constant and a variable). By Lemma 7.6, 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 $\mathcal{T} \succeq t$.

If the set of o'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 a variable 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 7.9 *Routine* **variablize** *will learn one of the target trees from a pruned example.*

Proof: 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 7.8. This process continues until t is eventually equivalent to $\mathcal{T}$. □

Theorem 7.10 *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 7.2 and 7.1, respectively, there exists a substitution σ and mapping μ so $\mathcal{T}\sigma\mu = t$. If μ is not one-to-one onto, by Lemma 7.7, routine **prune** can remove leaves until the mapping satisfies that criteria. Then, by Lemma 7.8, 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 7.11 *UF with one-to-one into semantics is learnable with equivalence and subset queries.*

Proof: By Lemma 7.6 this class is compact. Therefore in the algorithm in Figure 7.2 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 7.12 *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. □

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 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 (the number of 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, (Theorem 6.2 in Section 6.2).

7.5. Summary

This chapter showed that although *one-to-one onto* semantics is not learnable with equivalence and subset queries, the flexibility of *one-to-one into* semantics permits an algorithm which generalizes from one example per target tree to work. An lgg algorithm similar to what was used for *ordered one-to-one onto* semantics will apparently not work for this semantics. But an algorithm related to that for μ-UT (Lemma 6.3) does work. Unlike the algorithm for the latter class, this algorithm can't directly generalize a set of repeated variables because there are too many choices for the direct approach. The algorithm has to perform an end-run around this obstacle by first specializing a hypothesis and then generalizing it.

8. OTHER SEMANTICS

We wanted to make a connection between tree learning and predicate Horn-clause learning. Since more than one predicate can be satisfied by being matched to the same instance literal, one-to-one into semantics is not flexible enough to represent predicate learning. This discrepancy also explains why lgg doesn't work for one-to-one into semantics. Many-to-one semantics (both onto and into) will therefore be studied and compared to Horn-clause learning.

A Horn clause is a conjunction of predicates. *Horn clauses* are unordered and variable in number as far as the literals (individual predicates) are concerned. Further, an instance is allowed to have extra laterals which are not matched by a predicate clause containing variables. But the arguments within the predicates tend to be fixed in number and in a specific order, and an instance matches only if there are exactly the same number of arguments in each predicate. Predicate learning thus resembles a mixture of unordered into and ordered onto tree matching. This issue will be explored.

8.1. Many-To-One Onto Semantics

With trees of more than one level, the rules for matching semantics work recursively. Figure 8.1 shows a pattern which matches an instance according to unordered many-to-one onto semantics. The two x's can map to the same subtree in the instance (as allowed by the word, many, in the semantics name); assume this is the subtree with the C. The subtree $A(Byy)$ (in parent(child ...child) notation) maps to the other subtree of the instance by having the two y's map to the same child (say D). This fact and the other two pattern children matching the first instance subtree therefore satisfy the requirements for matching at the top level and for the trees as a whole. Since either instance subtree can serve either role, there is more than one way for the match to work.

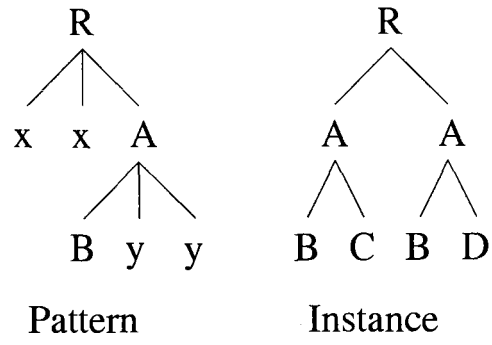


FIGURE 8.1. Recursive Many-to-One Onto Mapping

It is easiest to understand the effect of *many-to-one onto* semantics on matching and subset relationships with one-level trees having all children identical. Then a tree pattern with p children matches trees with 1 through p children (compared to ≥ 1 child for *many-to-one into* semantics). If another child variable is added to a tree pattern, the set represented is a superset of the original and taking away children produces a subset. Adding and pruning children in a tree pattern therefore has the opposite effect for many-to-one onto as with many-to-one or one-to-one into semantics.

The T2 class (single tree hypothesis with two levels, subtrees all the same number of children, and just three distinct variables) in the nonlearnability proof for UT with one-to-one onto semantics (Theorem 5.5) can be learned under many-to-one onto semantics. This fact is true even if the matricies representing the trees have only 1's and 0's, i.e., there are no repeated variables within a subtree–but there are in the tree as a whole and the trees correspond to *Boolean matrices*. Assume there are s subtrees, and the learner knows this fact from an

initial training example. Further assume these subtrees are interchangeable as far as matching is concerned, so any pattern subtree could match any instance subtree if the numbers of each distinct variable or constant are appropriate. Recall that T2 for Theorem 5.5 was represented with a matrix notation with each entry giving the count of a particular variable/constant in a particular subtree. If all entries in the matrix representing the target were ≥ 1, then an algorithm could learn by just creating a hypothesis with all 1's and increasing each entry as long as the result is still a subset of the target–by the above argument explaining matching and subset relationships for many-to-one onto. The many-to-one onto learnability question therefore reduced to that of learning the tree equivalent of arbitrary Boolean matrices.

Suppose the target variables are x, y, and z. If some entries in a target matrix in the T2 class are 0, then learner needs to discover the count of the number of subtrees with only x, count of those with just x and z, of all three variables, etc. Only the counts of these subtrees (not the positions) matters since matching is unordered. The task is therefore to split the s subtrees into 7 groups each of which corresponds to one of the nonempty subsets of $\{x, y, z\}$. Since the learner knows there are a total of s subtrees, a hypothesis which is a subset of the target can be found in $O(s^6)$ tries. Searching for three groups of subsets of only two variables (at a time in a hypothesis h–say x and y) will not work because the onto part of the semantics requires a match of a target t to h to have a variable in a subtree of t corresponding to any z in h. Further, the match must uniformly use the same row (i.e., tree variable) correspondence for all columns/subtrees. Then the learner can proceed as in the case where all matrix entries are ≥ 1. Even if the variables are interchangeable, $O(s^6)$ cases would still have to be tested. This approach gives polynomial-time learnability for any fixed number of target variables, but not for an indefinite number of variables, since $O(s^{2^v-2})$ tries would be required for v variables.

Theorem 8.1 *UT with many-to-one onto semantics is not learnable.*

Proof (Sketch): To show that UT with many-to-one onto semantics is not learnable, first note that the above algorithm sketch for T2 depends on the subtrees being interchangeable. If the subtrees with a variable to be partitioned were on many tree levels (or have different root nodes labels) and therefore not interchangeable (much like the UF-nonlearnability proof, Theorem 5.9), then exponential combinations would have to be tried to partition one variable into two. Other query/tree pattern combinations or attempts to combine examples would encounter similar difficulties. The nonlearnability proof must also retain some interchangeable subtrees to make it difficult to figure out how to combine multiple examples. A class similar to that used for UF nonlearnability (Theorem 5.9) could be used. □

An lgg algorithm (similar to the cartesian-product style used for many-to-one into semantics) will apparently not work for many-to-one onto (or even one-to-one into) semantics because of difficulties with producing a true lgg. The algorithm for one-to-one into semantics (which specializes before generalizing) can't be adapted to many-to-one onto, either.

Corollary 8.2 *UF with many-to-one onto semantics is not learnable.*

Proof (Sketch): Use a more complex version of the same subclass of tree patterns representable with Boolean matrices. Then the targets can be shown to be consistent with an exponential number of examples (as in the one-to-one onto UF nonlearnability proof). □

8.2. 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).[1]

The primitives of predicate clauses are constants, variables, *function symbols*, and *predicate symbols*. Both constants and variables are considered to be *"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 (interpreted conjunctively) 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.

---

[1] An earlier version of this work and parts of other chapters is in (Amoth, Cull, & Tadepalli, 2001)

Definition 8.1 *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 8.2 *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 8.3 *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 8.3). 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 8.3 is the standard *ordered-tree lgg algorithm* ordlgg in Figure 3.3 or LGG in the introduction 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.

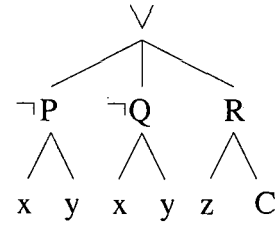


FIGURE 8.2. A Clausal-Tree equivalent to $\neg P(x,y) \vee \neg Q(x,y) \vee R(z,C)$

Theorem 8.3 (Plotkin, 1970) *The* **productlgg** *algorithm (Figure 8.3) 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 trees (Figure 8.2), 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 into 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) ∨

$\sigma_1 = \{\}$; $\sigma_2 = \{\}$;
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 $x_{i,j} = \mathsf{ordlgg}(t_i, r_j)$ 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 8.3. Cartesian-Product *lgg* Algorithm: **productlgg**

$\neg$Father(x, n) $\vee$ $\neg$Mother(y, z) $\vee$ GrandFather(x, z) with substitutions σ_1 = $\{x$/John, y/MotherOf(Lisa), q/John, z/Lisa, o/Sam, n/Sam$\}$ and σ_2 = $\{x$/Peter, y/Alice, q/Mark, z/Mary, o/Alice, n/Mark$\}$.

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 & Tadepalli, 1999) to learn single-predicate, non-recursive Horn programs can be directly adapted to learning CT forests. Figure 8.4 shows the learning algorithm. The main routine of the algorithm maintains the hy-

main: initialization: $h = \{\}$
while EQ(h) gives counterexample x (which is positive):
  If $\exists$ a tree pattern p in h such that SQ(**productlgg**(p, x))
    then replace the first such p with **reduce**(**productlgg**(p, x))
    else $h = h \bigcup x$. % 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 8.4. Algorithm for CT Forests

pothesis as a set of Clausal tree 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 tree 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 still represents subset of the target. It is sufficient to go through the top level subtrees in sequence, and prune any subtree when the result is accepted 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 of the first two examples 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 would be matched according to their ordering and generalized by the *ordered lgg algorithm.*

Multiple clause targets are equivalent to unions of clausal trees–or clausal forests. One complication in learning forests is to decide which sets of examples should be combined into a single tree. Following (Reddy & 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 $\neg$Father(Paul,Bob) $\vee$ $\neg$Father(Bob, Ted) $\vee$ GrandFather(Paul, Ted), which captures a second type of GrandFather relation. The lgg of this example with the current hypothesis clause $\neg$Father(x,y) $\vee$ $\neg$Mother(y,z) $\vee$ GrandFather(x,z) gives $\neg$Father(x,y) $\vee$ $\neg$Father(w,z) $\vee$ 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 8.4 *Clausal trees and forests are learnable from equivalence and membership queries.*

Proof: Based on the learnability of single-predicate non-recursive Horn clauses (Reddy & Tadepalli, 1999). □

8.3. Summary

This chapter showed a connection between *many-to-one into* trees and *Horn-clauses* (conjunction of *predicate clauses*). Predicate clause matching to instances (instantiated predicates) behaves like trees that match in a *many-to-one into* fashion in the top level and in a *one-to-one onto* fashion at all levels below.

A sketch explaining why *many-to-one onto* semantics trees/forests is not learnable with equivalence and subset/membership queries much as *one-to-one onto* semantics is not. The precise subclass used for the proofs differ significantly, but the technique is basically the same. the classes and the arguments differ.

9. CONCLUSIONS

In this chapter, the results are summarized in tables showing which classes are learnable with which sets of queries and which algorithms are used to learn those classes. The implications are then discussed. Finally, future work including open problems and other uses of the theory are described.

9.1. Summary of the Results

Figure 9.1 is a graph summarizing which tree classes are learnable with which algorithms. The blank spaces correspond to classes which are not learnable according to proofs in Chapters 4 and 5 (see the tables below). The row and column labels specify the query sets and concept classes specify the learning problem for each small rectangle, and the shading patters indicate which algorithms are applicable to a given query set-class combination. The query set "1ex" indicates only one example is given to the learner (possibly with another query oracle). The algorithms are: "Dup.Child" is the algorithm used in the proof for one-to-one into semantics which generalized repeated variables by duplicating those variables. "Bottom-Up" is the algorithm that generalize from a single example for the μ-UT class, "Top-Down" is the SupQ-based algorithm for UT (both in Chapter 6), and "lgg" is the least-general-generalization algorithm (Chapter 3). Figure 9.2 shows the same information for forests. Forests are never learnable from single examples or without equivalence queries, so those cases are not included in this latter figure.

The tables summarize the conditions under which tree classes are learnable or not learnable. The mapping is a function of the following 3 major features: the semantics (one/many-to-one onto/into), the tree class (ordered/unordered tree/forest/repeated variables or not), and the set of queries used (some subset of EQ, MQ/SQ, and SupQc with counterexamples or SupQ without counterexamples). The mapping yields either learnable with a proof

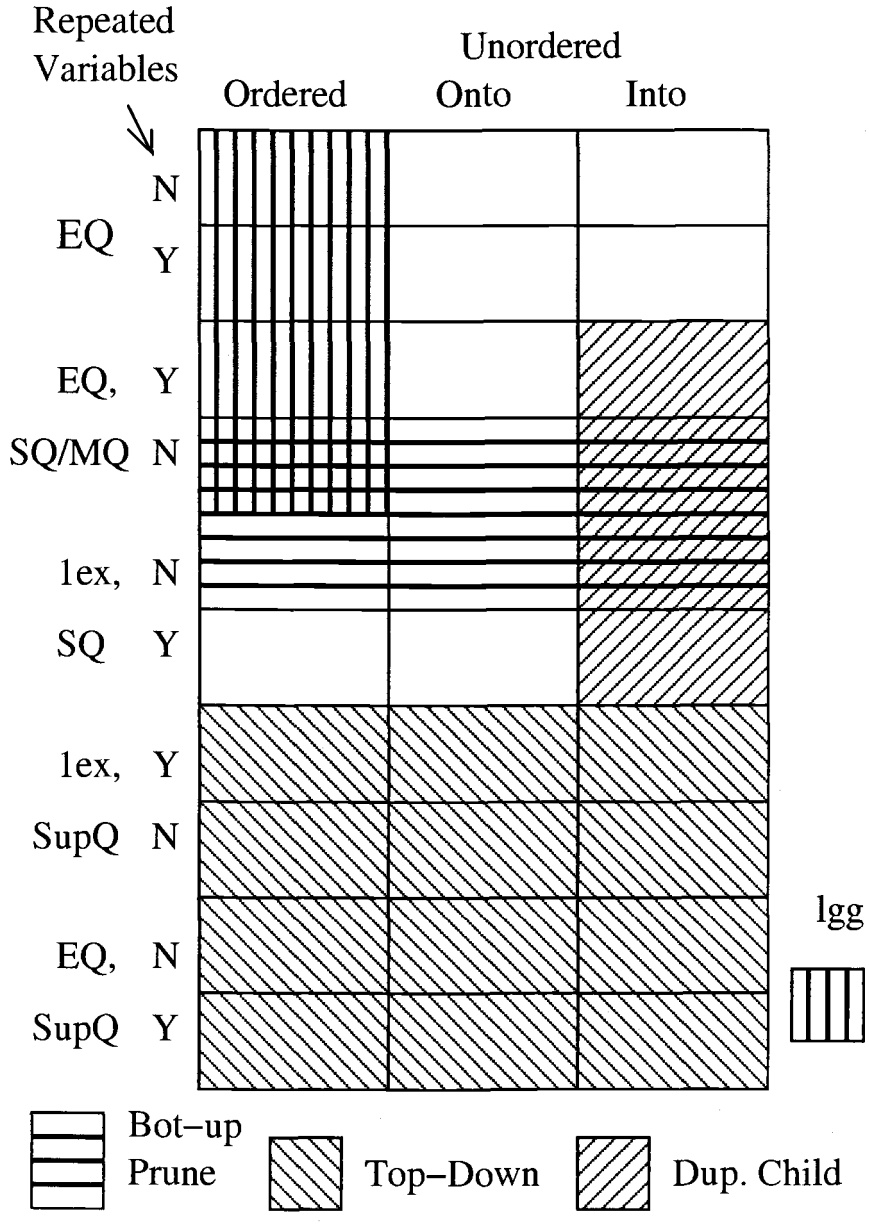


FIGURE 9.1. Trees Classes Algorithms Summary

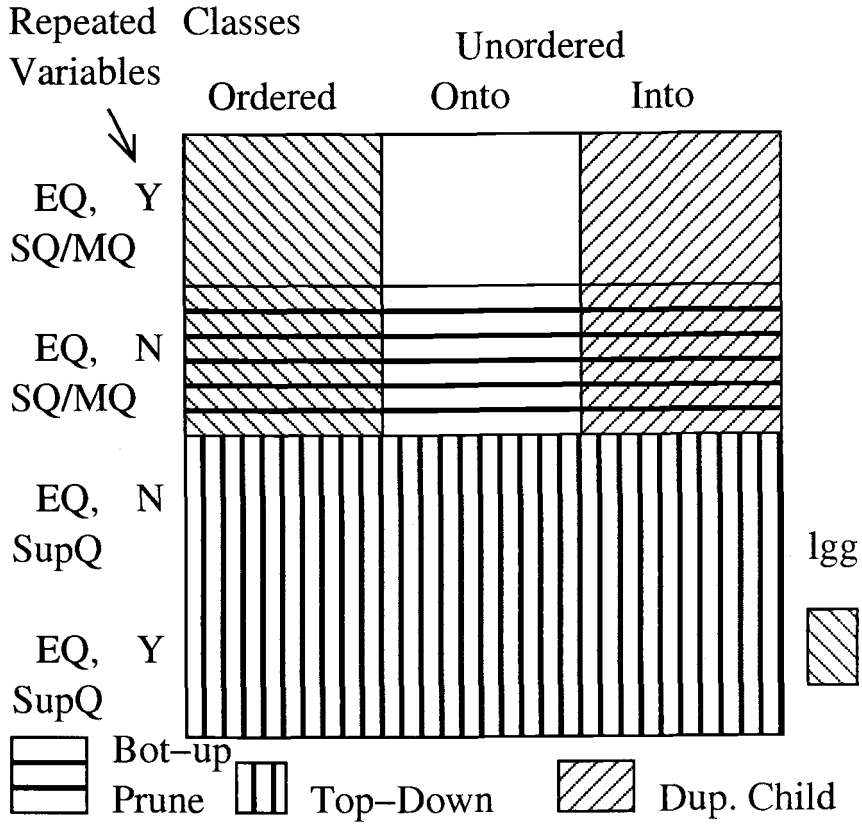


FIGURE 9.2. Forest Classes Algorithms Summary

reference and abbreviated description of the algorithm, or nonlearnable and the proof technique. Many other combinations are implied by those given in the tables. Taking away a query (thereby weakening the learnability) from a combination that is not learnable preserves that result. Conversely, strengthening the query set (e.g., by changing MQ to SQ) preserves learnability.

Class	Queries	Justification	Algorithm
OT	EQ only	Theorem 3.7	lgg
OF	EQ+MQ	Theorem 3.12	lgg + MQ
μ-UT	SQ+1ex.	Corollary 6.4	Bot-Up 1ex
μ-UF	EQ+MQ	Corollary 6.3	Bot-Up 1ex
UT	SupQ+1ex.	Theorem 6.6	Top-Down 1ex
UF	SupQc+SQ or	Theorem 6.12	Simulates EQ
	SupQc+EQ	Theorem 6.11	Top-Down Parallel

TABLE 9.1. Class/Query Combinations Learnable with One-to-One Onto Semantics

For the classes studied in this thesis, if a class is learnable for forests, then the corresponding tree class is learnable. The contrapositive applies equally well: if trees are not learnable, then corresponding forests class is also not learnable. Although this relationship is not true in general, e.g., the proof that UT is not learnable uses a subclass of UT with only 3 tree pattern variables (Chapter 5, and this particular subclass is learnable in UF (by calling EQ until all ways of partitioning the set of variables are exhausted), but the UF class as a whole is not learnable. (The proof of UT nonlearnability depends on the class of hypotheses given to queries being restricted to the target class and therefore a

separate proof is required for UF.) Similarly the unordered tree/forest classes which are learnable with the sets of queries given are also learnable as ordered tree/forest classes with the same sets of queries.

Table 9.1 summarizes the class/set-of-query combinations that are learnable with one-to-one onto semantics. The theorem proving the result is given along with an abbreviated description of the algorithm technique. Recall that O,U,T,F stand for ordered, unordered, tree, and forest. The prefix $\mu-$ represents no repeated variables. In the technique column, 1ex represents starting from a single example (per target tree where applicable). SupQc is the superset query with counterexamples, but SupQ is without counterexamples. Table 9.2 gives similar results for one-to-one onto class-query combinations which are not learnable.

Class	Queries	Justification	Technique
OF	EQ only	Lemma 4.8	DNF$\leq$
μ-OF$_{l,b}$	EQ+MQ	Lemma 4.13	DNF$\leq$
μ-UT	EQ only	Lemma 5.2	DNF$\leq$
UT	EQ+SQ	Theorem 5.5	Combinatorial
UF	SupQc+MQ	Lemma 6.13	Not Terminate
UF	EQ+SQ	Theorem 5.9	Combinatorial

TABLE 9.2. Not Learnable with One-to-One Onto Semantics

Table 9.3 summarizes the learnability results for semantics other than one-to-one onto of the unordered tree classes using EQ and MQ or SQ. For each query set and semantics for matching a tree to the instance tree, a reference

to the relevant proof is given along with a an abbreviated description of the algorithm or proof technique.

Learnable				
Concept Class	Queries	Matching Semantics	Justification	Algorithm
UT	EQ,MQ	1-1 **into**	Theorem 7.10	bottom-up
UF	EQ,MQ	1-1 **into**	Corollary 7.12	bottom-up
Clausal trees(UF)	EQ,MQ	many-1 **into**/ ordered 1-1 onto	Theorem 8.4	lgg/prune
Not Learnable				
Concept Class	Queries	Matching Semantics	Justification	Technique
UF	EQ,SQ	many-1 **onto**	Section 8.1	Combinatorial

TABLE 9.3. UT/F Learnability With Other Semantics

9.2. Discussion

In this chapter we considered the learnability of ordered and unordered tree patterns under various definitions of match semantics and related it to learning first order clauses. The main results of this thesis are summarized in Tables 9.1 to 9.3. Learning of tree patterns offers a rich variety of classes and an interesting set of connections to other classes.

We began with ordered one-to-one onto semantics. In previous work, it was shown that it is possible to learn unions of ordered tree patterns when the degree or alphabet sizes are not bounded (Arimura et al., 1995; Page, 1993),

(Theorems 3.7, 3.11 and their accompanying algorithms give details). In this thesis, we have shown that DNF learning reduces to learning unions of ordered tree patterns with bounded degree and bounded alphabet size (bounded ordered forests) without repeated variables (Theorem 4.8). The reduction proofs for the exact learning model (Angluin, 1988) also apply to the PAC-predictability with and without membership queries (Pitt & Warmuth, 1990; Angluin & Kharitnov, 1995). It is simpler to reduce the classes in the prediction frameworks because the hypotheses of the learning algorithm need not be translated back by the reduction algorithm.

Unfortunately, negative results showed the class of *unordered* tree patterns is not learnable with examples and membership queries when the mapping from the subtrees of the tree pattern to subtrees of the tree instance is required to be one-to-one onto regardless of the computational power of the learner (Theorems 5.5 and 5.9). 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.

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 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.

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 which 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 & Tadepalli, 1999).

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 the following sources of complexity in tree learning: *onto*-semantics, unordered trees, repeated variables, and restriction to subset queries (instead of superset queries). Only when all four of these are present, is learning hard (Theorems 5.5 and 5.9). With ordered trees, it is possible to combine multiple examples. If there are no repeated variables, the tree can be learned in a bottom-up, divide-and-conquer fashion, since all the subtrees are independent (Theorem 6.2). With superset queries, the correct partitioning of a set of repeated variables can be found in a step-by-step fashion. 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 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, 1984b), 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 (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 mathematics 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/oracle 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.[1] 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 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. Because of the conjunctive nature of the hypothesis

---

[1] If the target is also declaratively available to the teacher, he may simply tell it to the learner, thus making the learning problem superfluous.

space, generating instances from a superset of the hypothesis (e.g., the entire rest of the instance space) and filtering by the target before testing them on the hypothesis would be be too inefficient in any nontrivial domain.

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 $A \bigcap T = \{\}$ (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 or 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.

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*.[2] 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.

9.3. Future Work

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.

The open problems in this thesis are worthy of future study: Reduction of DNF from a higher base to a lower base seems to have the same difficulty as reduction of DNF to ordered forests with bounded label alphabet and number of children. Reduction of bounded OF without repeated variables to DNF which apparently needs to be solved to be able to reduce bounded OF with repeated variables to DNF (i.e., if the latter reduction works then presumably the former does; maybe a "reduction between reductions" could be proven). These four open questions are all in section 4.7. Although the two problems within each pair of open problems seem to be related, the following possible, abstract connection between the pairs is worth investigating: all four problems have difficulty with any fixed, a priori hypothesis translation scheme. Section 6.6 discusses a fifth

---

[2]The limited number of constants in planning domains requires trying the query on more than one problem.

open problem: the learnability of unordered forests with a finite label alphabet and a bound on the number of children below each node (the discussion in that section ties this problem to SAT).

The effects of the following variations of the tree learning problems and other problems are worth studying:

- Comparison of tree learning with learning of Horn programs as a whole rather than just individual predicate clauses (analogous to one tree pattern) or a set of clauses with a common antecedent (a forest)
- Variations of matching semantics–such as many-to-one into or "converses" of the semantics already studied (while satisfying the constraint that each variable has only one value)
- Variations on tree classes–such as mixed ordered/unordered
- Heuristics for applications–for many nonlearnable classes, the hard cases are likely to be so obscure in practice that heuristics could be readily used
- Applications to mathematics and linguistic processing
- Learning tree transformations (e.g., for symbolic mathematics)
- Learning from noisy examples
- Learning from *incomplete membership oracles* (meaning the oracle does not always provide an answer)

BIBLIOGRAPHY

Aho, A. V., Sethi, R., & Ullman, J. D. (1987). *Compilers Principles, Techniques, and Tools.* Addison-Wesley.

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. 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. ACM.

Amoth, T. R., Cull, P., & Tadepalli, P. (2001). On exact learning of unordered tree patterns. *Machine Learning, forthcoming.*

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

Angluin, D., Frazier, M., & Pitt, L. (1992). Learning conjunctions of Horn clauses. *Machine Learning, 9*, 147–164.

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

Angluin, D. (1990). Negative results for equivalence queries. *Machine Learning, 5*, 121–150.

Angluin, D., & Kharitnov, M. (1995). When won't membership queries help?. *Journal of Computer and System Sciences, 50*, 336–355. Also extended abstract in STOC-91, 444-451.

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.

Birkhoff, G. (1967). *Lattice Theory.* American Mathematical Society.

Califf, M. E., & Mooney, R. J. (1997). Relational learning of pattern-match rules for information extraction. In *Proceedings of the ACL Workshop on Natural Language Learning, Madrid, Spain*, pp. 9–15.

Cardie, C. (1997). Empirical methods in information extraction. *AI Magazine, 18*, 65–80.

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.* W.H. Freeman, San Fransisco, CA.

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.

Kearns, M. J., & Vazirani, U. V. (1994). *An Introduction to Computational Learning Theory.* The M.I.T. Press, Cambridge, MA.

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

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

Mitchell, T. (1980). The need for biases in learning generalizations. In Shavlik, J., & Dietterich, T. (Eds.), *Readings in Machine Learning*, pp. 184–191. Morgan Kaufmann, San Mateo, CA. Originally published as Rutgers University technical report.

Natarajan, B. (1991). *Machine Learning: A Theoretical Approach.* Morgan Kaufmann, San Mateo, CA.

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 Muggleton, S. H. (Ed.), *Inductive Logic Programming*, pp. 29–61. Academic Press.

Pitt, L., & Warmuth, M. (1990). Prediction preserving reducibility. *Journal of Computer and System Sciences, 41*(3), 430–467.

Plotkin, G. (1970). A note on inductive generalization. In Meltzer, B., & Michie, D. (Eds.), *Machine Intelligence*, Vol. 5, pp. 153–163. Elsevier North-Holland, New York.

Reddy, C., & Tadepalli, P. (1998). Learning first-order acyclic horn programs from entailment. In *Proceedings of the 15th International Conference on Machine Learning; (and Proceedings of the 8th International Conference on Inductive Logic Programming).* Morgan Kaufmann.

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. Morgan Kaufmann.

Ullman, J. D. (1982). *Principles of Database Systems.* Computer Science Press.

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

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

INDEX