Machine Learning 2: 39–74, 1987
© 1987 Kluwer Academic Publishers, Boston – Manufactured in The Netherlands

A Version Space Approach to Learning Context-free Grammars

KURT VANLEHN (VANLEHN@A.PSY.CMU.EDU)
WILLIAM BALL (BALL@A.PSY.CMU.EDU)
Department of Psychology, Carnegie-Mellon University, Pittsburgh, PA 15213 U.S.A.

(Received: June 17, 1986)
(Revised: January 21, 1987)

Keywords: Induction, grammatical inference, version space, context-free grammars, learning from examples.

Abstract. In principle, the version space approach can be applied to any induction problem. However, in some cases the representation language for generalizations is so powerful that (1) some of the update functions for the version space are not effectively computable, and (2) the version space contains infinitely many generalizations. The class of context-free grammars is a simple representation that exhibits these problems. This paper presents an algorithm that solves both problems for this domain. Given a sequence of strings, the algorithm incrementally constructs a data structure that has nearly all the beneficial properties of a version space. The algorithm is fast enough to solve small induction problems completely, and it serves as a framework for biases that permit the solution of larger problems heuristically. The same basic approach may be applied to representations that include context-free grammars as special cases, such as And-Or graphs, production systems, and Horn clauses.

1. Introduction

The problem addressed here arose in the course of studying how people learn arithmetic procedures from examples (VanLehn, 1983a; VanLehn, 1983b). Our data allowed us to infer approximations of the procedures the subjects had learned and the examples they received during training. Thus, the inputs and outputs to the learning process were known, and the problem was to describe the learning process in detail. However, because the subjects' learning occurred intermittently over several years, we were not immediately interested in developing a detailed cognitive simulation of their learning processes. Even if such a simulation could be constructed, it might be so complicated that it would not shed much light on the basic principles of learning in this task domain. Therefore, our initial objective was to find principles that could act as a partial specification of the learning process. The principles we sought took the form of a representation language for procedures, together with inductive biases that would postdict the procedures learned by our subjects. More precisely, our problem was:

- Given:
    - a training sequence, consisting of examples of a procedure being executed, and
    - a set of observed procedures, represented in some informal language (i.e., English),
- Find:
    - a representation language for procedures, and
    - a set of inductive biases, expressed as predicates on expressions in the representation language,

  such that the set of all procedures that are consistent with the examples and preferred by the biases

    - includes the observed procedures, and
    - excludes implausible procedures (e.g., ones that never halt).

This method for studying the structure of mental representations and processes has much to recommend it (VanLehn, Brown & Greeno, 1984; Fodor, 1975), but here we wish to discuss only the technical issues involved in implementing it. The central technical problem is calculating the sets mentioned above. The calculation must be done repeatedly, once for each combination of representation language, biases and training sequence. Although the calculations could be done by hand, it is probably easier to program a computer to perform them. Rather than build one program that could handle all combinations, or one program for each combination, we chose a hybrid approach.

The approach was to build a different program for each representation language. The programs are *induction* programs, in that they take a sequence of training examples and calculate expressions in the representation language that are generalizations of those examples. The inducers are *unbiased*, in that they produce *all* expressions in the language consistent with their inputs. An unbiased inducer provides a framework on which we can install explicit biases in an attempt to fit its output to the data. The advantage of this approach is that tuning an unbiased inducer is much easier than building a different biased inducer for each set of biases. The main technical problem of implementing this approach is devising an unbiased inducer for each of the hypothesized representation languages.

It is very important to understand that these inducers are merely tools for generating certain sets that we are interested in studying. They are not meant to be models of human learning processes.

This approach works fine for some representation languages, but not for others. Some procedure representation languages (e.g., those used by Anderson, 1983, and VanLehn, 1983c) are based on recursive goal hierarchies that are isomorphic to context-free grammars.[1] For several reasons, it is impossible to construct an inducer that produces the set of all context-free grammars consistent with a given training set. First, such a set would be infinite. Second, the standard technique for representing such a set, Mitchell's (1982) version space technique, seems inapplicable because the crucial 'more-specific-than' relationship is undecidable for context-free grammars.[2] The proofs for these points will be presented later. Although we could have abandoned exploration of procedure representation languages with recursive goal hierarchies, we chose instead to attack the subproblem of finding a suitable induction algorithm for context-free grammars.

The impossibility of an unbiased inducer means that a biased one must be employed as the framework on which hypothesized biases are installed for testing their fit to the data. Because we will not be able to test the fit with the built-in bias removed, the built-in bias must be extremely plausible a priori. Moreover, there must be an algorithm for calculating the set of grammars consistent with it, and that set must be finite.

We found such a bias and called it *reducedness.* A grammar is *reduced* if removing any of its rules makes it inconsistent with the training examples. Later, we will argue for the plausibility of reducedness and, more importantly, we will prove that there are only finitely many reduced grammars consistent with any given training sequence. This proof is one of the main results presented in this paper.

The proof contains an enumerative algorithm for generating the set of reduced grammars consistent with a training sequence, but the algorithm is far too slow to be used. In order to experiment with biases, we needed an algorithm that could take a training sequence of perhaps a dozen examples, and produce a set of reduced grammars in a day or less time.

The obvious candidate for a faster algorithm is Mitchell's (1982) version space strategy. Applying the strategy seems to involve conquering the undecidability of the 'more-specific-than' relationship for grammars. However, we discovered that it was possible to substitute a decidable relationship for 'more-specific-than' and thereby achieve an algorithm that had almost all the beneficial properties of the version space technique. In particular, it calculates a finite, partially ordered set of grammars that can be represented compactly by the maximal and minimal grammars in the

[1]A context-free grammar is a set of rewrite rules, similar to a simple production system. The next section gives precise definitions of the relevant terms from formal language theory.

[2]The version space technique is explained in the next section.

order. Unfortunately, the set is not exactly the set of reduced grammars, but it does properly contain the set of reduced grammars. We call it the *derivational* version space.

The derivational version space satisfies our original criterion: it is a set of consistent grammars which is arguably a superset of the set of grammars *qua* procedures that people learn. Moreover, the algorithm for calculating it is fast enough that small training sequences can be processed in a few hours, and the structure of the algorithm provides several places for installing interesting biases. The derivational version space is the second result to be presented in the paper.

The main interest for machine learning researchers lies in the generality of the techniques we used. The reducedness bias can be applied directly to many representation languages. For instance, an expression in disjunctive normal form (i.e., a disjunction of conjunctions) is reduced if deleting any of its disjuncts makes the expression inconsistent with the training examples. The finiteness result for reduced grammars suggests that sets of reduced expressions in other representations are also finite and effectively computable.[3] Moreover, the technique of substituting an easily computed relation for the 'more-specific-than' relation suggests that such sets of reduced expressions can be efficiently computed using the derivational version space strategy.

Indeed, the fact that substituting another relation for 'more-specific-than' leads to a useful extension of the version space strategy suggests looking for other relationships that provide the benefits of version spaces without the costs. This idea is independent of the idea of reducedness. Both ideas may be useful outside the context of grammar induction.

There are four main sections to the paper. The first introduces the relevant terminology on grammars, grammar induction and version spaces. The second discusses reducedness and the finiteness of the set of grammars consistent with a set of examples. The third discusses the derivational version space, while the fourth presents the induction algorithm for this structure and demonstrates the results of incorporating certain biases. The concluding section speculates on the larger significance of this work.

[3]It might be argued that although the bias can be applied to other representation languages, one might not want to. However, reducedness is already obeyed by all constructive induction programs that we are familiar with, including the systems of Quinlan (1986), Michalski (1983), and Vere (1975). (Reducedness is not usually obeyed by enumeration-based induction algorithms, such as those found in the literature on language identification in the limit (Osherson, Stob & Weinstein, 1985).) Apparently, the designers of constructive inducers believe that it is natural for an induced generalization to include only parts (e.g., rules, disjuncts) that have some support in the data. Reducedness is a precise statement of this belief.

2. Terminology

2.1 Introduction to grammars and grammar induction

A grammar is a finite set of rewrite rules. A rule is written $\alpha \rightarrow \beta$, where α and β are strings of symbols. Grammars are used to generate strings by repeatedly rewriting an initial string into longer and longer strings. In this article, the initial string is always 'S'. For instance, the following grammar

$$\begin{aligned} &S \rightarrow b \\ &S \rightarrow aS \end{aligned}$$

generates the string 'aab' via two applications of the second rule and one application of the first rule:

$$S \rightarrow aS \rightarrow aaS \rightarrow aab$$

Such a sequence of rule applications is called a *derivation.* There are two kinds of symbols in grammars. *Terminals* are symbols that may appear in the final string of a derivation, whereas *nonterminals* are not allowed to appear in final strings. In the above grammar, a and b are terminals, and S is a nonterminal.

The grammar induction problem is to infer a grammar that will generate a given set of strings.[4] The set of strings given to the learner is called the *presentation.* It always contains strings that the induced grammar should generate (called *positive* strings) and it may or may not contain strings that the induced grammar should not generate (called *negative* strings). For instance, given the presentation

$$-\ a,\ +\ ab,\ +\ aab,\ -\ ba$$

the grammar given earlier (call it Grammar 1) could be induced because it generates the two positive strings, 'ab' and 'aab,' and it cannot generate the two negative strings, 'a' and 'ba'. A grammar is said to be *consistent* with a presentation if it generates all the positive strings and none of the negative strings.[5]

There are usually many grammars consistent with a given presentation. For instance, here are two more grammars consistent with the presentation mentioned above:

[4]Grammar induction is studied in at least three fields – philosophy, linguistics, and artificial intelligence. For reviews from the viewpoints of each, see, respectively, Osherson, Stob and Weinstein (1985), Pinker (1979), and Langley and Carbonell (1986). In addition, Cohen and Feigenbaum (1983) give an excellent overview.

[5]Some authors use 'deductively adequate' (Horning, 1969) or 'consistent and complete' (Michalski, 1983) for the same concept. We use the term 'consistent' in order to bring the terminology of this paper into line with the terminology of Mitchell's (1982) work on version spaces.

Grammar 2	*Grammar 3*
$S \rightarrow A$	$S \rightarrow Sb$
$A \rightarrow b$	$Sb \rightarrow ab$
$A \rightarrow aA$	$S \rightarrow aa$

Grammar 2 is equivalent to grammar 1 in that it generates exactly the same set of strings: {b, ab, aab, aaab, aaaab, ...}. The set of all strings generated by a grammar is called the *language* generated by the grammar. The language in this case is an infinite set of strings. However, languages can be finite. The language generated by Grammar 3 is the finite set {ab, aa, aab}.

Grammar induction is considerably simpler if restrictions are placed on the class of grammars to be induced. Classes of grammars are often defined by specifying a format for the grammars that are members of the class. For instance, grammars 1 and 2 obey the format restriction that the rules have exactly one nonterminal as the left side. Grammars having this format are called *context-free grammars.* Grammar 3 is not a context-free grammar.

2.2 Version spaces

One of the most widely studied forms of machine learning is learning from examples, or *induction*, as it is more concisely called. The following is a standard way to define an induction problem:[6]

- Given:
  - A representation language for generalizations;
  - A predicate of two arguments, a generalization and an instance, that is true if the generalization matches the instance;
  - A set of 'positive' instances (which should be matched by the induced generalizations) and a set of 'negative' instances (which should not be matched);
  - A set of biases that indicate a preference order for generalizations;
- Find: One or more generalizations that are
  - consistent with the instances; and
  - preferred by the biases;

where 'consistent' means that the generalization matches all the positive instances and none of the negative instances.

This formulation is deliberately vague in order to encompass many specific induction problems. For instance, the instances may be ordered. There may be no negative instances. There may be no biases, or biases that rank

[6]Throughout, we follow Mitchell's (1982) choice of terminology, with two exceptions. First, we use Generalizes(x,y) instead of more-specific-than(y,x). Second, if x Generalizes y, then we visualize x as *above* y; Mitchell (1982) would say that x is *below* y.

generalizations on a numerical scale, or biases that partially order the set of generalizations. Much work in machine learning is encompassed by this definition.

Mitchell defines a *version space* to be the set of *all* generalizations consistent with a given set of instances. This is just a set, with no other structure and no associated algorithm. However, Mitchell also defines *the version space strategy* to be a particular induction technique, based on a compact way of representing the version space. Although popular usage of the term 'version space' has drifted, this paper will stick to the original definitions.

The central idea of the version space strategy is that the space of generalizations defined by the representation language can be partially ordered by generality. One can define the relation Generalizes(x,y) in terms of the matching predicate:

Definition 1 Generalizes(x,y) is true if and only if the set of instances matched by x is a superset of the set of instances matched by y.

Note that the Generalizes relationship is defined in terms of the denotations of expressions in the representation language, and not the expressions themselves. This will become important later, when it is shown that the Generalizes relation is undecidable for context-free grammars.

It is simple to show that the Generalizes relation partially orders the space of generalizations. Thus, no matter what the specific induction problem may be, one can always imagine its answer as lying somewhere in a vast tangled hierarchy which rises from very specific generalizations that cover only a few instances, all the way up to generalizations that cover many instances.

Given a presentation, the version space for that presentation will also be partially ordered by the Generalizes relation. Given some mild restrictions (e.g., that there are no infinite ascending or descending chains in the partial order), the version space has a subset of maximal elements and a subset of minimal elements. The maximal set is called G, because it contains the set of maximally *general* generalizations. The minimal set is called S, because it contains the maximally *specific* generalizations. The pair [S,G] can be used to represent the version space. Mitchell proved that

> Given a presentation, x is contained in the version space for that presentation if and only if there is some g in G such that g Generalizes x and there is some s in S such that x Generalizes s.

Three algorithms are usually discussed in connection with the [S,G] representation of version spaces:

- `Update(i,[S,G]) → [S',G']`
  The Update function takes the current version space boundaries and an instance that is marked as either positive or negative. It returns boundaries for the new version space. If the instance makes the version space empty (i.e., there is no generalization that is consistent with the presentation, as when the same instance occurs both positively and negatively), then some marker, such as Lisp's NIL, is returned. The Update algorithm is the induction algorithm for version space boundaries. Its implementation depends on the representation language.
- `DoneP([S,G]) → true or false`
  Unlike many induction algorithms, it is possible to tell when further instances will have no effect because the version space has stabilized. DoneP is implemented by a test for set equality, S = G.
- `Classify(i,[S,G]) → +, -, or ?`
  Classify an instance that is not marked positively or negatively, using the version space boundaries. It returns '+' if the instance would be matched by all the generalizations in the version space. It returns '−' if it would be matched by no generalizations. It returns '?' otherwise. Classify is useful for experimental design. If instances are marked by some expensive-to-use teacher (e.g., a proton collider), then only instances that receive '?' from Classify are worth submitting to the teacher.

Applying the version space strategy to a representation language means that one must devise only an appropriate Update function, because the Classify and DoneP functions come for free with the strategy. This is sometimes cited as the chief advantage of the version space approach. In our work on skill acquisition, we make only minor use of them. Our main reason for preferring the version space strategy over other induction strategies is that it computes exactly the set we need, the version space, and represents it compactly.

3. Reduced version spaces

The first problem encountered in applying the version space strategy to grammar induction is that the version space will be always be infinite. This does not necessarily imply that the version space boundaries will be infinite; a finite S and G can represent an infinite version space. However, for grammars, the boundaries also turn out to be infinite. To begin, let us consider a well-known theorem about grammar induction, which is:

Theorem 1 For any class of grammars that includes grammars for all the finite languages, there are infinitely many grammars in the class that are consistent with any given finite presentation.

That is, the version space is infinite for any finite presentation. This theorem has a significance outside the context of version space technology. For instance, it has been used to justify nativist approaches to language acquisition (Pinker, 1979). This section is written to address both the concerns of version space technology and the larger significance of this theorem.

3.1 Normal version spaces are infinite

Three ways to prove the theorem will be presented. Simply amending the statement of the theorem to prevent the use of each of the proof techniques yields a new theorem, which is one of the results of this article.

All three proofs employ mathematical induction. The initial step in all the proofs is the same. Because the class of grammars contains grammars for all finite languages, and the positive strings of the presentation constitute a finite language, we can always construct at least one grammar that is consistent with the presentation. This grammar initializes the inductions. The inductive steps for each of the three proofs are, respectively:

1. Let α be any string not in the presentation. Add the rule S $\rightarrow$ α to the grammar. The new grammar generates exactly the old grammar's language plus α as well. Since the old language was consistent with the presentation, and α does not appear in the presentation, the new grammar is also consistent with the presentation. Because there are infinitely many strings α that are not in the presentation, infinitely many different grammars can be constructed this way. One might object that the rule S $\rightarrow$ α may be in the grammar already. However, because a grammar has finitely many rules, there can be only finitely many such α, and these can be safely excluded when the α required by the proof is selected.

2. Let A be a nonterminal in the grammar, and let B be a nonterminal not in the grammar. Add the rule A $\rightarrow$ B to the grammar. For some or all of the rules that have A as the left side, add a copy of the rule to the grammar with B substituted for A. These additions create new grammars that generate exactly the same strings as the original grammar. Because the original grammar is consistent with the presentation, so are the new grammars. This process can be repeated indefinitely, generating an infinite number of grammars consistent with the presentation.

3. Form a new grammar by substituting new nonterminals for every nonterminal in the old grammar (except S). Create a union grammar whose rules are the union of the old grammar's rules and the new grammar's rules. The union grammar generates exactly the same language as the original grammar, so it is consistent with the presentation.

The union process can be repeated indefinitely, yielding an infinite set of grammars consistent with the presentation.

It is hard to imagine why a machine or human would seriously entertain the grammars constructed above. The grammars of the last two proofs are particularly worthless as hypotheses, because they are notational variants of the original grammar. In a moment, we will add restrictions to the class of grammars that will bar such irrational grammars.

We have mentioned that an infinite version space can, in principle, be represented by finite boundaries. Unfortunately, this does not work for grammars. The second two proofs above will generate infinitely many grammars that generate exactly the same language as the initial grammar. If the initial grammar is from S, then S can be made infinite; similarly, G can be made infinite. The G set can also be made infinite by the first proof above. These comments prove the following theorem:

Theorem 2 If the representation language for generalizations specifies a class of grammars that includes grammars for all finite languages, then for any finite presentation, the version space boundaries, S and G, are each infinite.

3.2 Reducedness makes the version space finite

One way to make the version space finite is to place restrictions on the grammars to be included in it. As some of these restrictions are most easily stated as restrictions on the form of grammar rules, we will limit our attention to context-free grammars, although the same general idea works for some higher order grammars as well (as shown in Appendix 1). The first restriction blocks the grammars produced by the second proof:

Definition 2 A context-free grammar is *simple* if (1) No rule has an empty right side,[7] (2) if a rule has just one symbol on its right side, then the symbol is a terminal, and (3) every nonterminal appears in a derivation of some string.

The class of simple grammars can generate all the context-free languages. Hopcroft and Ullman (1979) prove this (theorem 4.4) by showing how to turn an arbitrary context-free grammar into a simple context-free grammar. For our purposes, the elimination of rules of the form A $\rightarrow$ B, where both A and B are nonterminals, blocks the second proof.

Proofs 1 and 3 can be blocked by requiring that all the rules in an induced grammar be necessary for the derivation of some positive string in the given presentation. To put this formally:

[7]This reduces the expressive power of the class somewhat, because a grammar without such epsilon rules, as they are commonly called, cannot generate the empty string.

Definition 3 Given a presentation P, a grammar is *reduced* if it is consistent with P and if there is no proper subset of its rules that is consistent with P.

Removing rules from a grammar will only decrease the size of the language generated, not increase it. So removing rules from a grammar will not make it generate a negative string that it did not generate before. However, deleting rules may prevent the grammar from generating a positive string, thus making it inconsistent with the presentation. If any deletion of rules causes inconsistency, the grammar is reduced.

In proof 1, adding the rules $S \rightarrow \alpha$ creates a new grammar that is reducible. Similarly, the union grammar formed by proof 3 is reducible. This leads to the theorem:

Theorem 3 Given a finite presentation, there are finitely many reduced simple context-free grammars consistent with that presentation.

The proof of this theorem is presented in Appendix 1. We call the version space of reduced, simple grammars a *reduced* version space for grammars.

Any finite partially ordered set has a finite subset of minimal elements and a finite subset of maximal elements. Define the *reduced* G and S as the maximal and minimal sets, respectively, of the reduced version space under the partial order established by the Generalizes relation. It follows immediately that:

Theorem 4 Given a finite presentation, the reduced G and S sets are each finite.

3.3 The behavior of reduced version spaces

This subsection describes some of the ways in which a reduced version space differs from a normal version space.

Normally, a version space can only shrink as instances are presented. As each instance is presented, generalizations are eliminated from the version space. With a reduced version space, negative instances cause shrinking, but positive instances usually expand the reduced version space. To see why, suppose that at least one of the grammars in the current version space cannot generate the given positive string. There are usually several ways to augment the grammar in order to generate the string. For instance, one could add the rule $S \rightarrow \alpha$, where α is the string. Or one could add the rules $S \rightarrow A\beta$ and $A \rightarrow \gamma$, where $\alpha = \gamma\beta$. Each way of augmenting the current grammar in order to generate the new string contributes one grammar to the new version space. So positive strings cause the reduced version space to expand.

Because presenting a positive string can cause the reduced version space to expand, the equality of S and G no longer implies that induction is done. That is, the standard implementation of DoneP does not work. We conjecture that Gold's (1967) theorems would allow one to show that there is no way to tell when induction of a reduced version space is finished.

The S set for the reduced version space turns out to be rather boring. It contains only grammars that generate the positive strings in the presentation. We call such grammars *trivially specific* because they do nothing more than record the positive presentation. The version space Update algorithm described below does not bother to maintain the S set, although it could. Instead, it maintains P+, the set of positive strings seen so far. In order to illustrate the efficiency gained by this substitution, consider the Classify function, whose normal definition is: where i is an instance to be classified, if all s in S match i, then return '+'; else if no g in G matches i, then return '−'; else return '?'. With P+, the first clause of the definition becomes: if i is in P+, then return '+'. Because S contains only the trivially specific grammars, these two tests are equivalent. Clearly, it is more efficient to use P+ instead of S. Similar efficiencies are gained in the implementation of the Update algorithm. Nowlan (1987) presents an alternative solution to this problem with some interesting properties.

3.4 Why choose reducedness for an inductive bias?

The basic idea of reducedness applies to other representation languages. For instance, suppose the representation is a first order logic whose expressions are in disjunctive normal form (i.e., a generalization is one large disjunction, with conjunctions inside it). The rules in a grammar are like disjuncts in a disjunction. Therefore, a disjunctive normal form expression is reduced if removing a disjunct makes it inconsistent with the presentation. We conjecture that the reduced version space for disjunctive normal forms will turn out to be finite. There may be a general theorem about reducedness and finiteness that would apply, at the knowledge level perhaps (Newell, 1982; Dietterich, 1986), to many representation languages.

As mentioned earlier, it seems that reducedness is a 'common sense' restriction to place on induction. All heuristic concept induction programs with which we are familiar (e.g., Michalski, 1983; Vere, 1975; Quinlan, 1986) consider only reduced concepts. Reducedness seems to be such a rational restriction that machine learning researchers adopt it implicitly.

There are other ways to restrict grammars so that there are only finitely many grammars consistent with a finite presentation. For instance, there are only finitely many simple, trivially specific grammars consistent with a finite presentation. However, the restriction to reduced, simple grammars seems just strong enough to block the procedures that produce an infinitude of grammars without being so strong that interesting grammars are blocked as well. This makes it an ideal restriction to place on version spaces for grammars. The chief advantage of version spaces is that they contain *all* the generalizations consistent with the presentation. In order to retain the basic spirit of version spaces while making their algorithms effective, one should add the weakest restrictions possible. For grammars, the conjunction of reducedness with simplicity seems to be such a restriction.

4. Applying the version space strategy to reduced version spaces

The proof of theorem 3 puts bounds on the size of reduced grammars and their rules. In principle, the reduced version space could be generated by enumerating all grammars within these bounds. However, such an algorithm would be too slow to be useful. This section discusses a technique that yields a much faster induction algorithm.

4.1 The undecidability of the Generalizes relationship

The version space strategy is the obvious choice for calculating a reduced version space, but it cannot, we believe, be applied directly. The problem is that the version space strategy is based on the Generalizes relationship, which is defined by a superset relationship between the denotations of two generalizations. If the generalizations are grammars, then the denotations are exactly the languages generated by the grammars. Implementing Generalizes(x,y) is equivalent to testing whether the language generated by x includes the language generated by y. This test is undecidable for context-free grammars or grammars of higher orders (Hopcroft & Ullman, 1979, theorem 8.12). This means that there is no algorithm for implementing Generalizes(x,y) over the context-free grammars.

This result does not prove that the version space strategy is inapplicable, because only the Update algorithm is required in order to construct a version space, and there is no proof (yet) that a computable Generalizes is necessary for a computable Update. On the other hand, we have never seen a version space Update algorithm that did not call Generalizes as a subroutine, and we have no idea how to build a Generalizes-free Update algorithm for grammars. So the undecidability of the Generalizes predicate is a practical impediment, at the very least.

The Generalizes predicate may be decidable if its arguments are restricted to be reduced grammars for the same presentation. If so, then it may be possible to use Generalize in an Update algorithm that only works for the reduced version space, and not the normal version space. We did not explore this approach. Instead, we sought a way to apply the spirit of the version space strategy while avoiding the troublesome Generalizes predicate entirely.

The 'trick' to the version space strategy is using the boundaries of a partial order to represent a very large, partially ordered set. In principle, this trick can be based on any partial order, and not necessarily on the partial order established by Generalizes. This idea led us to seek a partial order that was 'like' Generalizes and yet was computable. Moreover, the partial order had to be such that there was an Update algorithm for the sets of maximal and minimal elements in the order.

It was not difficult to find a computable partial order on grammars, but we never found an Update algorithm that could maintain sets that were the boundaries of *exactly* the reduced version space. Instead, we discovered one for a *superset* of the reduced version space. In particular, we found:

- A set, called the *derivational version space*, that is a superset of the reduced version space and a subset of the version space.
- A computable predicate, called *FastCovers*, that is a partial order over grammars in the derivational version space.
- An Update algorithm for the maximal and minimal elements in FastCovers of the derivational version space.

The remainder of this section presents the derivational version space and the FastCovers relation. The next section discusses the Update algorithm.

4.2 The derivational version space

In order to define the derivational version space, it will be helpful to define some ancillary terms first. A *derivation tree* is a way to indicate the derivation of a string by a grammar. (These are sometimes called *parse trees.*) The derivation tree's leaves are the terminals in the string. The non-leaf nodes of the tree are labelled by nonterminals. The root node is always labelled by the root nonterminal, S. An algorithm can 'read off' the rules used by examining mother-daughter subtrees. If the label of the mother is A and the labels of the daughters are B, C and D, then the rule A $\rightarrow$ B C D has been applied. This reading off process can be used to convert derivation trees into a grammar.

For simple grammars, derivation trees are constrained to have certain possible shapes. Simple grammars have no rules of the form A $\rightarrow$ B, where both A and B are nonterminals. Therefore, if a node in the derivation tree has a single daughter, that daughter must be a terminal, because a rule may have a singleton right side only if that side consists of a terminal. Let us call trees with this shape *simple trees.* The definition of simple grammars makes it impossible for a simple tree to have long, unbranching chains. Consequently, there are only finitely many unlabelled simple trees for any given string.

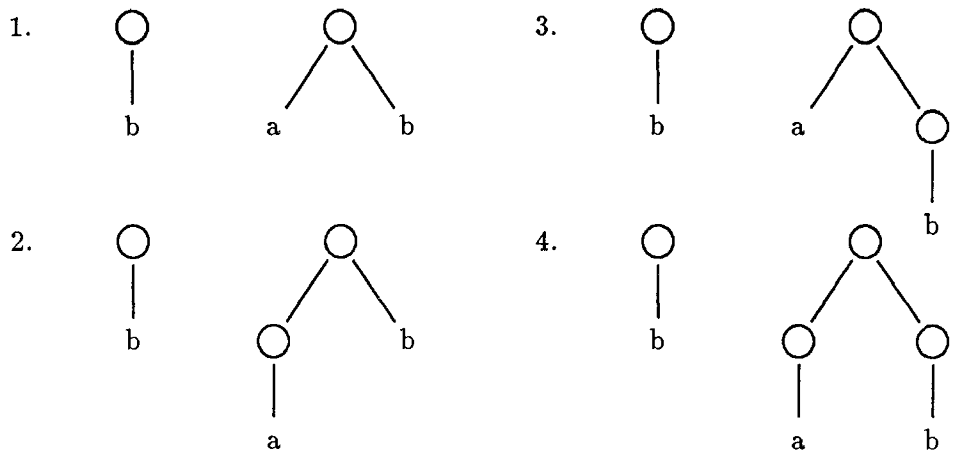


Figure 1. The simple tree product for the presentation 'b', 'ab'.

If a string has more than one element, then there is more than one unlabelled simple tree. Given a finite sequence of strings, one can calculate all possible sequences of unlabelled simple trees by taking the Cartesian product over the sets of unlabelled simple trees for each string. Let us call this set of simple tree sequences the *simple tree product* of the strings. Because there are only finitely many unlabelled simple trees for each string, the simple tree product will be finite. The definition of the derivational version space can now be stated:

Definition 4 Given a set of positive strings, the derivational version space is the set of grammars corresponding to all possible labellings of each tree sequence in the simple tree product for those strings. Given a set of positive and negative strings, the derivational version space is the derivational version space for the positive strings minus those grammars that generate any of the negative strings.

An example may clarify this definition. Suppose the positive strings are 'b' and 'ab.' The construction of the derivational version space begins by considering the simple tree product for the strings. There is one unlabelled tree for 'b.' There are four unlabelled trees for 'ab.' So there are four tree sequences in the Cartesian product of the trees for 'a' and the trees for 'ab.' These four tree sequences constitute the simple tree product, which is shown in Figure 1. For each of the four tree sequences, the construction process partitions the nodes in the trees and assigns labels. Figure 2 illustrates how the fourth unlabelled tree sequence is treated. At the top of the figure, the unlabelled tree sequence is shown with its nodes numbered. Trees 1 through 5 show all possible partitions of the four nodes and the labellings of the trees that result. Because the root nodes of the trees must always receive the same node label, S, they are given the same number, which forces them to be in the same partition element, and hence receive the same labelling. Each of the resulting labelled tree sequences is converted to a grammar. These grammars are shown in the third column of the figure. The derivational version space is the union of these grammars, which derive from the fourth tree sequence, with the grammars from the other tree sequences.

The motivation for the derivational version space is the following: If a grammar is going to parse all the positive strings, then there must be a sequence of simple derivation trees, one for each string. Such a sequence must be some possible labelling of some possible sequence of unlabelled simple trees. The derivational version space is constructed from all such sequences, i.e., from the simple tree product. Consequently, it must somehow represent all possible grammars, except those grammars which have rules that were not used during the parsing of those strings. Those grammars are, by definition, the reducible grammars. So the derivational version space contains all the reduced grammars. These observations lead to the following theorem, which is proved in Appendix 2:

Theorem 5 The derivational version space for a given presentation contains the reduced version space for that presentation.

Usually, the reduced version space is a *proper* subset of the derivational version space. That is, the derivational version space often contains reducible grammars. In the illustration discussed earlier, where the positive strings (P+) were 'b' and 'ab,' no reducible grammars would be generated. However, if P+ was {'b,' 'ab,' 'ab'} or if P+ was {'b,' 'ab,' 'abb'}, then many reducible grammars would be generated. In general, if a subset of P+ is sufficient to produce grammars that will generate all of it, then the derivational version space will contain reducible grammars.

The following theorem shows that the 'version space' component of the name 'derivational version space' is warranted:

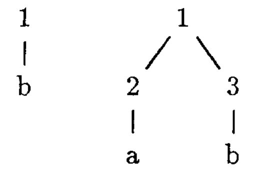
Theorem 6 The derivational version space for a given presentation is contained in the version space for that presentation.

The proof follows from the observation that the grammars in the derivational version space were constructed so that each positive string has a derivation, and grammars that generate negative strings are filtered out. Consequently, the grammars are consistent with the presentation. Lastly, we note that:

Theorem 7 The derivational version space for a finite presentation is finite.

The proof follows from the earlier observation that the simple tree product is finite. Because each tree sequence in the product has only finitely many

Unlabelled trees:



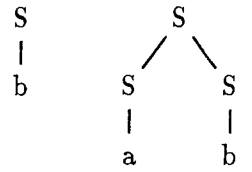
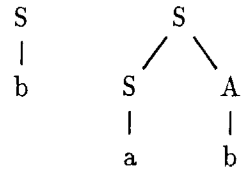
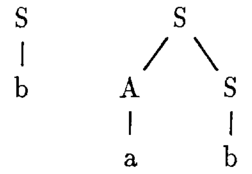
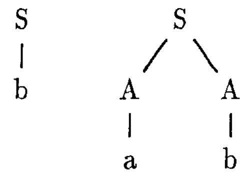
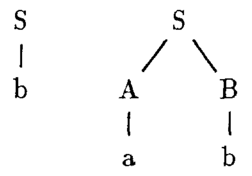
	Partition	*Labelled Trees*	*Grammar*
4.1	{1,2,3}		S → b S → a S → SS
4.2	{1,2} {3}		S → b S → a S → SA A → b
4.3	{1,3} {2}		S → b S → AS A → a
4.4	{1} {2,3}		S → b S → AA A → a A → b
4.5	{1} {2} {3}		S → b S → AB A → a B → b

Figure 2. Partitions, labelled trees, and grammars of tree sequence 4.

nodes and there are only finitely many ways to partition a finite set into equivalence classes, there are only finitely many ways to label each of the finitely many simple tree sequences. Hence, the derivational version space for P+ is finite. The derivational version space for the whole presentation is a subset of the one for P+, so it too is finite.

The derivational version space is a finite set that contains all the reduced simple grammars, and moreover, all its members are consistent with the presentation. This set suffices for the purposes we outlined in the introduction. It contains all the 'plausible' grammars and it is finite. We show next that there is a partial order for the set that allows a boundary updating algorithm to exist.

4.3 The FastCovers predicate

The definition of the partial order is simplified if grammars in the derivational version space are represented in a special way, as a triple. The first element of the triple is a sequence of unlabelled simple derivation trees, with the nodes numbered as in Figure 1. The second element of the triple is a partition of the trees' nodes. The third element is the grammar's rules. For instance, grammar 4.4 of Figure 2 is represented by the following triple:

```
Tree sequence:
    (1 b), (1 (2 a)(3 b))
Partition:
    {1}, {2 3}
Rules:
    S → b
    S → AA
    A → a
    A → b
```

The triple representation allows the FastCovers relation to be defined as follows:

Definition 5 Given two grammars, X and Y, in triple form, grammar X FastCovers grammar Y if (1) both grammars are labellings of the same tree sequence (i.e., the first elements of their triples are the same), and (2) the partition (i.e., second element of the triple) of Y is a refinement[8] of the partition of X.

For instance, a grammar whose partition is ({1},{2},{3}) is FastCovered by the grammar above. In contrast, a grammar whose partition is ({1 2},{3}) is not FastCovered by the above grammar, nor does it FastCover that grammar.

[8]A partition PY is a *refinement* of another partition PX if and only if every partition element of PY is a subset of some partition element of PX.

FastCovers is named after Covers, a partial order used in early work on grammar induction (Reynolds, 1968; Horning, 1969; Pao, 1969). Although we will not pause to define Covers, it can be shown that FastCovers(x,y) implies Covers(x,y), but Covers(x,y) does not imply FastCovers(x,y). FastCovers is used instead of Covers for ordering the derivational version space because it is faster to compute than Covers and it makes the Update algorithm simpler.

It is simple to show that the FastCovers relationship is transitive and reflexive, because these hold for the refinement relationship. Moreover, because every grammar in the derivational version space has a triple form, FastCovers applies to every pair of grammars in a derivational version space. Thus, FastCovers partially orders the derivational version space.

A second property of FastCovers is needed to show that the Update algorithm is correct:

Theorem 8 For any two grammars, X and Y, in triple form, FastCovers(X,Y) implies Generalizes(X,Y).

The proof follows from observing that the refinement relationship between the nonterminals (= partition elements) of X and the nonterminals of Y establishes a mapping that takes Y's nonterminals onto X's nonterminals. Every derivation in grammar Y can be turned into a derivation in grammar X by mapping Y's nonterminals onto X's nonterminals. Thus, every string that has a derivation in Y must have a derivation in X as well. So the language generated by Y is a subset of the language generated by X, i.e., Generalizes(X,Y).

Given a derivational version space, there is always a finite set of maximal elements in FastCovers and a finite set of minimal elements. The finiteness of the boundaries follows from the finiteness of the space itself. We will call the maximal and minimal sets the *derivational* G and S, respectively. From the preceding theorem, it follows immediately that:

Theorem 9 The derivational G (S) includes the subset of the derivational version space that is maximal (minimal) with respect to the Generalizes relationship.

Given a derivational [S,G], the FastCovers relationship can be used to determine whether a given grammar is contained in the derivational version space represented by the pair.

Theorem 10 Given a grammar x in triple form and a derivational [G,S], x is in the derivational version space represented by [G,S] if and only if there is some g in G such that g FastCovers x, and some s in S such that x FastCovers s.

The proof of the theorem is given in Appendix 3.

5. An Update algorithm for the derivational version space

The preceding section discussed the definition of the structure that we wish to generate. This section presents the algorithm that generates the structure, then reports the results of several experiments with it. It begins by presenting an informal account of what happens as positive and negative strings are presented.

5.1 The 'sliced bread' metaphor for derivational version spaces

The derivational version space under FastCovers is a set of partition lattices,[9] one lattice for each tree sequence in the simple tree product. One can visualize the space as a loaf of sliced bread, one slice for each tree sequence. All the FastCovers relationships run inside slices; none cross from one slice to another. Each slice is a partition lattice. It has one maximal partition on top and one minimal partition on the bottom. The top partition has just one element, and the element has all the nodes in the tree sequence for that slice. The top partition for tree sequence of Figure 2 is $(\{1, 2, 3\})$. The bottom partition in each lattice has a singleton partition element per node in the tree sequence. The bottom partition for the tree sequence of Figure 2 is $(\{1\},\{2\},\{3\})$. All the slices/lattices have unique top and bottom partitions.

If there are no negative instances in the presentation, then G consists of the top partition in each lattice. As negative instances are presented, the maximal set for each lattice may descend. Thus, the G set expands and the derivation version space shrinks as negative strings are presented. The S set always consists of the bottom partition in each lattice. Presentation of negative instances does not effect the S set.

When a new positive instance is presented, the derivational version space grows horizontally, so to speak (i.e., the loaf gets more slices, and the slices get larger). If the newly added positive string has more than one member, there will be more than one unlabelled simple derivation tree for it. Hence, the simple tree product will increase in size and the set of partition lattices will increase as well (i.e., the loaf gets more slices). Moreover, each of the new tree sequences is longer than the corresponding old one, because some unlabelled derivation tree for the new string has been added to it. The new, longer tree sequence will have more nodes (again, assuming that the string has more than one member). With more nodes available for partitioning, the partition lattices will expand. Thus, the loaf's slices get larger. In

[9] A lattice is a partial order with the additional property that every pair of nodes in the lattice has a singleton maximal set and a singleton minimal set. A partition lattice consists of the set of all partitions of some finite set of objects, ordered by the refinement relationship.

short, presenting a positive string increases the number of partition lattices and the sizes of the partition lattices.

Presenting a new positive string affects the derivational S and G sets in the following ways. The increase in the number of partitions implies that the derivational S grows because its members are always the bottom partitions of the partition lattices. The effect on the derivational G is more subtle. If there are no negative instances, then G grows because its members are the top elements of the partition lattices. If there are negative instances, then G *may* grow as positive instances are presented, but we have no proof that it *must* grow. Although the number of maximal sets grows, the size of the sets may decrease, leaving the overall G set the same size, or perhaps even decreasing it.

5.2 The Update algorithm

As mentioned earlier, our algorithm does not bother to maintain the S set, although it could easily do so. Instead, it maintains P+, the set of positive strings seen so far. This makes the algorithm more efficient.

The Update algorithm is incremental. It takes an instance and the current [P+, G] pair and returns a revision of the pair that is consistent with the given instance. If there is no such revision, then the algorithm returns some error value, such as NIL. The following describes the top level of the algorithm:

1. If the string is positive and a member of P+, then do nothing and return the current version space. If the string is not a member of P+, then add it to P+ and call Update-G+.
2. If the string is negative and a member of P+, then return NIL. If the string is not a member of P+, then call Update-G−.

The subroutine Update-G− is simpler than Update-G+, so it will be described first.

The task of Update-G− is to modify G so that none of the grammars will parse the negative string. The easiest way to do this is with a queue, which is initialized to contain all the grammars in G. The basic cycle is to pick a grammar off the queue, and see if it parses the negative string. If it does not, then it can be placed in New-G, the revised version of G. If it does parse the string, then the algorithm refines the node partition once, in all possible ways. That is, it takes a partition such as ({1 2 3},{4 5}), and breaks one of the partition elements in two. In this case, there are four possible one-step refinements:

1. {1}, {2 3}, {4 5}
2. {1 2}, {3}, {4 5}
3. {1 3}, {2}, {4 5}
4. {1 2 3}, {4}, {5}

Each of these corresponds to a new grammar. These grammars have the property that they are FastCovered by the original grammar, and there is no grammar that FastCovers them and not the original grammar. That is, they are just below the original grammar in the partial order of FastCovers. This process is called *splitting* in the grammar induction literature (Horning, 1969).[10]

All the grammars produced by splitting are placed on the queue. Eventually, the new grammars will be taken off the queue, as described above, and tested to see if they parse the negative string. Those that fail to parse the negative string are placed in New-G. Such grammars are maximal in the FastCovers order in that there is no grammar above them that fails to parse the negative string. The basic cycle of testing grammars, splitting them, and queuing the resulting new grammars continues until the queue is exhausted. At this point, New-G contains the maximal set of the grammars that fail to parse the negative string.

There is one technical detail to mention. It is possible for the same grammar to be generated via several paths. Before queuing a new grammar, the queue and New-G are checked to make sure the new grammar is not FastCovered[11] by some existing grammar.

New-G should contain only grammars that are (1) simple, (2) consistent with the positive presentation, (3) consistent with the negative presentation, and (4) maximal in the FastCovers partial order. The following comments prove that the Update-G− algorithm satisfies those criteria.

1. The grammars are simple, because the unlabelled derivation trees from which they are constructed are simple.
2. The grammars are consistent with the positive presentation, because they are a labelling of a set of derivation trees for those strings. Therefore, they are guaranteed to parse those strings.

[10]Normally, splitting takes one nonterminal (i.e., a partition element), and divides all its occurrences in two. Some of the occurrences are replaced by a new nonterminal. Thus, the old nonterminal is 'split' in two. The triple representation presented earlier allows a very simple implementation of splitting, but it applies only to grammars that can be represented as triples, and it generates only a subset of the grammars that normal splitting would produce.

[11]In order to make the algorithm function correctly, FastCovers must be used for this and not Covers. Filtering grammars that are covered by grammars from distinct unlabelled tree sequences will prune search paths that may lead to valid New-G members. This is the main reason for working with FastCovers rather than Covers.

3. The grammars are consistent with the the negative string just received, because the test puts the grammars in New-G only if they fail to parse that string. The grammars are consistent with the negative strings received prior to this one, because the grammars from the old G were consistent, and splitting moves down the FastCovers order, so splitting reduces the language generated by a grammar and never expands it.
4. The grammars are maximal in the FastCovers order because splitting moves down the order one step at a time, and the movement is stopped as soon as the grammar becomes consistent with the presentation.[12]

This completes the discussion of Update-G−. We now turn to Update-G+, the function that revises the G set when some of the grammars in it do not parse a newly received positive string.

The easiest way to explain Update-G+ is to first describe an algorithm that is not incremental: it takes the whole presentation at once and builds the appropriate G set. The non-incremental algorithm proceeds in the following steps:

1. Form the simple tree product by taking the Cartesian product of the unlabelled simple derivation trees for each positive string.
2. For each such tree sequence in the simple tree product, form a triple to represent the grammar that has only one nonterminal, S. The partitions for these grammars all have just one partition element, and the element contains all the nodes in the derivation tree sequence. These grammars are the maximal grammars in the FastCovers partial order. They would be the G set if the presentation had only the positive strings.
3. For each string in the set of negative strings, apply Update-G−.

This algorithm is not incremental, since it must process all the positive strings before it processes any of the negative strings. An incremental Update algorithm must be able to handle strings in the order they are received. The incremental algorithm should take a G set whose grammars may have already been split somewhat by Update-G−, and modify it to accommodate a new positive string.

In the non-incremental algorithm, the effect of adding a new string is to increase the length of the sequences of unlabelled derivation trees, and hence to increase the number of nodes in the partitions. In the incremental algorithm, this must be done in all possible ways, so the resulting Update algorithm is:

[12]A complete proof would require using the fact that the derivational version space is a set of lattices, and lattices are particularly well-connected.

1. Given a positive string, form the set of all unlabelled simple derivation trees for that string.
2. For each grammar in the old G and for each tree for the new positive string,
   (a) append the tree onto the end of the tree sequence of the grammar's triple, and
   (b) allocate the new tree's nodes to the partition elements in all possible ways. Thus, if there are N partition elements in the partition, then there are N choices for where to put the first tree node, N choices for where to put the second tree node, etc. If the tree has M nodes, then N^M new partitions will be generated. Each one becomes a grammar that is a candidate for New-G.
3. Place all the candidate grammars generated in the preceding step on the queue for the Update-G− algorithm. However, instead of testing that a grammar is consistent with just one negative string, as the Update-G− algorithm does, test that the grammar is consistent with all the negative strings in the presentation that have been received so far.

The first two steps generalize the old grammars by adding rules to them. The new grammars might be too general, in that they may parse some of the negative strings given earlier in the presentation. Hence, the last step must check *all* the negative strings. This requires saving all the negative strings as they are presented. Thus, the version space needs to be a triple: [P+, P−, G].

This means that one of the usual benefits of the version space technique is lost. Usually, version space induction allows the learner to forget about an instance after having processed it. This algorithm requires the learner to remember the instances in the presentation. However, it is still an incremental algorithm. After each string is presented, an up-to-date G set is produced. Moreover, it is produced with less processing and memory than would be required to generate that same G set completely from scratch using the entire presentation. In short, the algorithm is an incremental version space update with respect to computation, but not with respect to instance memory.

5.3 Illustrations of the algorithm's operation

In order to illustrate the operation of the algorithm, this subsection presents a simple example. The next subsection will continue this example by showing how the algorithm performs when it is modified to incorporate certain biases.

Table 1. Learning a command language.

Instances	*Size of G set*	*CPU seconds*
+ delete all-of-them	4	0.03
− all-of-them delete	5	0.25
− delete delete	5	0.52
+ delete it	25	11.80
− it it	25	0.32
+ print it	197	526.00
+ print all-of-them	2580	20300.00

The illustration is based on learning a command language for a file system. The algorithm receives strings of command words, marked positive or negative, and from these data it must infer a grammar for the command language. Suppose the first string is positive: 'delete all-of-them.' There are four possible unlabelled simple trees for this string, and they lead directly to four grammars for the G set. These grammars are listed below in their triple representation.

1. (1 delete all-of-them)
   {1}
   (S $\rightarrow$ delete all-of-them)
2. (1 delete (2 all-of-them))
   {1 2}
   (S $\rightarrow$ delete S) (S $\rightarrow$ all-of-them)
3. (1 (2 delete)(3 all-of-them))
   {1 2}
   (S $\rightarrow$ S all-of-them)(S $\rightarrow$ delete)
4. (1 (2 delete)(3 all-of-them))
   {1 2 3}
   (S $\rightarrow$ S S)(S $\rightarrow$ delete)(S $\rightarrow$ all-of-them)

The first three grammars generate the finite language consisting only of the single string 'delete all-of-them.' The fourth grammar generates all possible strings over the two word vocabulary of 'delete' and 'all-of-them.' Suppose the next string is a negative string, 'all-of-them delete.' This string cannot be parsed by grammars 1, 2 or 3, so they remain unchanged in the G set. The fourth grammar is overly general, so it is split. There are only three legal partitions. Two of them survive, becoming grammars 5 and 6 shown below. The other partition, {1}{2 3}, yields a grammar that parses the negative string, so it is split further, into {1}{2}{3}. This partition is FastCovered by the two survivors, so it is abandoned. The survivors are:

5. (1 (2 delete)(3 all-of-them))
   {1 2}{3}
   (S → S A)(S → delete)(A → all-of-them)
6. (1 (2 delete)(3 all-of-them))
   {1 3}{2}
   (S → A S)(A → delete)(S → all-of-them)

Suppose the next string is 'delete delete,' a negative instance. None of the grammars in G parse this string, so the G set remains unchanged. This illustrates that the algorithm has made an inductive leap while processing the preceding strings. This string is new, but there is no change in the version space.

Suppose the next string is positive, 'delete it.' There are four possible unlabelled simple derivation trees for this string. Each is paired with each of the five grammars in the current G, yielding 20 combinations. The resulting 20 grammars are queued for testing against P−. Some splitting occurs during the processing of the queue. When the queue is finally exhausted, New-G has 25 grammars.

Table 1 summarizes the results so far and shows what happens as more instances are presented. As a rough indication of the practicality of the algorithm, the table shows the number of CPU seconds used in processing each instance by a Xerox 1109 running Interlisp. The combinational explosion inherent in the Update-G+ algorithm is quite evident. However, the algorithm is fast enough to construct small version spaces.

Table 2. A bias for minimum number of nonterminals.

Instances	*Size of G set*	*CPU seconds*
+ delete all-of-them	4	0.03
− all-of-them delete	3	0.28
− delete delete	3	0.05
+ delete it	7	1.66
− it it	7	0.09
+ print it	17	5.87
+ print all-of-them	55	19.40

5.4 Biasing the Update algorithm

Better performance can be obtained by using the Update algorithm as a framework upon which biases can be mounted. There are several places in the algorithm where biases can be installed. One place is in the queue-based loop of Update-G−. Currently, new grammar triples are placed on the queue only if they are not FastCovered by existing grammar triples.

Table 3. The effects of limiting the splitting ply.

Instances	*one ply*		*two ply*	
	G	*Secs.*	G	*Secs.*
+ delete all-of-them	4	0.02	4	0.01
– all-of-them delete	5	0.39	5	0.54
– delete delete	5	0.09	5	0.08
+ delete it	25	6.99	25	13.50
– it it	25	0.31	25	0.32
+ print it	188	75.20	197	204.00
+ print all-of-them	2406	1110.00	2580	3460.00

This filter can be made stronger. For instance, suppose we queue only grammars that have a minimal *number* of nonterminals, that is, grammar triples with partitions of minimal cardinality. Table 2 shows the results of running the previous illustration with this bias installed.

The bias reduces the G set from 2580 grammars to 55 grammars. All of these grammars happen to use a single nonterminal, e.g.,

S → S all-of-them
S → S it
S → delete
S → print

Processing time is drastically reduced since many grammar triples – those with partitions having cardinality larger than that of some existing consistent grammar triple – are not even generated.

Another filter that can be placed on the Update-G– loop is one which limits how deeply into the partition lattice the search may delve. We implemented a filter which allows the user to set a 'ply.' If a grammar triple with partition of cardinality m needs to be split, the search will proceed only to partitions of cardinality m+n, where n is the ply set by the user. Table 3 indicates that this bias approximates the results of the unbiased algorithm more closely than does minimizing the number of nonterminals. Note especially that for a ply of two, all the grammars of the unbiased algorithm were produced at a fraction of the processing time.

The desirability of these biases will depend on the task domain. The point is only that the algorithm provides a convenient framework for implementing such biases.

Another place to install biases is in the subroutine of Update-G+ that generates unlabelled derivation trees. This placement allows the biases to control the *form* of the grammars. For example, if the tree generator produces only binary trees, then the induced grammars are in Chomsky normal form. If the tree generator is constrained to produce only right branching trees, then only regular grammars are considered. In the latter case, there is only one right branching simple tree for each string. Consequently, there is only one unlabelled tree sequence for any given presentation. Under these circumstances, FastCovers is equivalent to Covers, and our algorithm becomes similar to Pao's (1969) algorithm for learning finite state machines. The main difference is that Pao's algorithm employs an explicit representation for the whole version space, whereas our algorithm uses the more compact [P+, P−, G] representation. Table 4 shows the results of our algorithm on the test case discussed above when the bias for regular grammars is introduced. Tables 5 and 6 show the results for a more challenging case, inferring the syntax of Unix file names.

The point of this section is that the Update algorithm is good for more than just exploring induction problems. It can be used as a framework for the development of practical, task-specific learning machines.

Table 4. Inducing regular grammars for file system commands.

Instances	*Size of G set*	*CPU seconds*
+ delete all-of-them	1	0.07
− all-of-them delete	1	0.02
− delete delete	1	0.01
+ delete it	1	0.07
− it it	1	0.02
+ print it	1	0.11
+ print all-of-them	1	0.12

6. Conclusions

The introduction motivated the results presented here from a narrow perspective, viz. their utility in our research on cognitive skill acquisition. However, they may have wider application. This section relates the results to more typical applications of grammar induction, which, for purposes of discussion, can be divided into three classes:

- Grammar induction is used as a formal account of natural language acquisition (Osherson, Stob & Weinstein, 1985; Berwick, 1985; Langley & Carbonell, 1986; Pinker 1979). Learning the syntax of a language is regarded by some as an important component of learning a language, and grammar induction is one way to formalize the syntactic component of the overall language-learning task.

- Grammars are sometimes used in software engineering; e.g., for command languages or pattern recognition templates (Gonzalez & Thomason, 1978). Some applications require that the grammars change over time or in response to different external situations. For instance, a command language could be tailored for an individual user or a pattern recognizer might need to learn some new patterns. Grammar induction may be useful in such applications (Fu & Booth, 1975; Biermann & Feldman, 1972).
- Knowledge bases in AI programs often have recursive hierarchical structures, such as calling structures for plans or event schemata for stories. The hierarchical component of such knowledge is similar to a grammar. Grammar induction can be used to acquire the hierarchical structure, although it must be used in combination with other induction engines that acquire the other structures. For instance, the Sierra learner (VanLehn, 1987) represents knowledge as hierarchical production rules. It uses a grammar induction algorithm to learn the basic skeleton of its rules, a BACON-like function inducer (Langley, 1979) to learn details of the rules' actions, and a version space algorithm (Mitchell, 1982) to learn the exact content of the rules' conditions.

In all these applications, the problem is to find an algorithm that will infer an 'appropriate' grammar whenever it receives a 'typical' training sequence. The definition of 'appropriate grammar' and 'typical training sequence' depends, of course, on the task domain. However, it is usually the case that only one grammar is desired for any given training. If so, then the Update algorithm for derivational version spaces is not immediately applicable, because it produces a set of grammars. In fact, the set tends to grow larger as the training sequence grows longer. This is why we do not claim that the algorithm models human learning, even though it was developed as part of a study of human learning. The algorithm produces a set, whereas a person probably learns only one (or a few) grammars *qua* procedures. Similarly, the algorithm is not a plausible model of how children learn natural language, even in the liberal sense of 'plausible' employed by studies of language identification in the limit.[13]

[13]It might seem that after a large number of examples had been received, all the grammars in the derivational version space will generate exactly the same language, and that language is the target language. This would imply that our algorithm identifies languages in the limit, even though it produces a set of grammars instead of a single grammar. Kevin Kelly and Clark Glymour (personal communication) have proved this conjecture false. Kelly (personal communication) has shown that any identifiable class of context-free grammars is identifiable by a machine whose conjectures are always reduced, simple grammars that are consistent with the data. Kelly and Glymour conclude, along with us, that our results have little bearing on the language identification literature.

Table 5. Learning regular grammars for Unix file names.

Instances	*Size of G set*	*CPU seconds*
+ foo . bar	1	0.01
+ foo	1	0.01
+ bar	1	0.01
+ / gram / foo	1	0.04
− foo / / foo	5	1.03
− / / foo	2	0.82
+ / usr / vsg / bar	43	88.30
− / / bar	32	20.90
− / / / bar	25	9.49
− / usr / / bar	16	19.50
− / / gram / bar	32	9.76
− / / vsg / bar	14	6.20
− vsg / / usr / bar	22	10.10
− / usr / / gram / bar	15	17.20
− / usr / / foo	10	10.40
− / usr / / gram / foo	5	7.55
− / usr / / vsg / ba	2	8.72

Of course, not all applications of grammar induction desire an algorithm to produce just one grammar. An application might have an inducer to produce a set of grammars and leave some other process to choose among them. For instance, when tailoring a command language, one might have the user choose from a set of grammars generated by the grammar inducer.

If grammar induction is viewed as search, then producing just one grammar is a form of depth-first or greatest-commitment search. The version space strategy, as pointed out by Mitchell (1982), is a form of breadth-first or least commitment search. This gives it the capability of producing informative intermediate results. Halfway through the presentation of the training sequence, one can determine unequivocally which generalizations have been rejected and which generalizations are still viable. If this capability, or any other feature of the least commitment approach, is important, then the algorithm presented here should be considered.

Even if the application does not use grammars as the representation language or even as a component of the representation, the techniques presented here may be useful. The definition of reducedness extends readily to many representation languages. For instance, a Prolog program is reduced if deleting any of its Horn clauses makes the program inconsistent with the training examples. Similarly, the technique for building a derivational version space can be extended to representations other than grammars. It remains to be seen whether there is any utility in these extensions, but they are at least possible.

Table 6. The contents of the final G set of Table 5.

Grammar 1	*Grammar 2*
S → / A	S → / A
A → gram S	A → gram S
A → usr S	A → usr S
A → vsg S	A → vsg S
S → / foo	S → / foo
S → / bar	S → / bar
S → . bar	A → . bar
S → foo	S → foo
S → bar	S → bar
S → foo S	S → foo A

However, the applications that seem most likely to benefit from the derivational version space approach are those that are most similar to our application, in that their research problem is to understand the influence of representation languages and biases on learning. This amounts to studying the properties of induction *problems*, rather than studying the properties of induction *algorithms*. In such research, it is often a useful exercise to study the set of generalizations consistent with a given training sequence and see how that set changes as the biases and representation language are manipulated. Such sets are exactly what the derivational version space strategy calculates.

Acknowledgments

This research was supported by the Personnel and Training Research Programs, Psychological Sciences Division, Office of Naval research, under Contract N00014-86-K-0349. We would like thank Peter Gordon, Brian MacWhinney, Tom Mitchell, and Peter Reimann for their critiques. Special thanks go to Steve Nowlan and Kevin Kelly for debugging our claims, and to Pat Langley and Tom Dietterich for debugging our exposition.

References

Anderson, J. R. (1983). *The architecture of cognition.* Cambridge, MA: Harvard University Press.

Berwick, R. (1985). *The acquisition of syntactic knowledge.* Cambridge, MA: MIT Press.

Biermann, A. W., & Feldman, J. A. (1972). A survey of results in grammatical inference. In S. Watanabe (Ed.), *Frontiers of pattern recognition.* New York: Academic Press.

Cohen, P. R., & Feigenbaum, E. A. (1983). *The handbook of artificial intelligence.* Los Altos, CA: Morgan Kaufmann.

Dietterich, T. G. (1986). Learning at the knowledge level. *Machine Learning*, *1*, 287–316.

Fodor, J. A. (1975). *The language of thought.* New York: Crowell.

Fu, K., & Booth, T. (1975). Grammatical inference: Introduction and survey. *IEEE Transactions on Systems, Man, and Cybernetics*, *5*, 95–111.

Gold, E. M. (1967). Language identification in the limit. *Information and Control*, *10*, 447–474.

Gonzalez, R. C., & Thomason, M. G. (1978). *Syntactic pattern recognition.* Reading, MA: Addison-Wesley.

Hopcroft, J. E., & Ullman, J. D. (1969). *Formal languages and their relation to automata.* Reading, MA: Addison-Wesley.

Hopcroft, J. E., & Ullman, J. D. (1979). *Introduction to automata theory, languages, and computation.* Reading, MA: Addison-Wesley.

Horning, J. J. (1969). *A study of grammatical inference* (Technical Report CS–139). Stanford, CA: Stanford University, Department of Computer Science.

Langley, P. (1979). Rediscovering physics with BACON.3. In *Proceedings of the Sixth International Joint Conference on Artificial Intelligence* (pp. 505–507). Tokyo, Japan: Morgan Kaufmann.

Langley, P., & Carbonell, J. G. (1986). Language acquisition and machine learning. In B. MacWhinney (Ed.), *Mechanisms of language acquisition.* Hillsdale, NJ: Lawrence Erlbaum.

Michalski, R. S. (1983). A theory and methodology of inductive inference. In R. S. Michalski, J. G. Carbonell, & T. M. Mitchell (Eds.), *Machine learning: An artificial intelligence approach.* Los Altos, CA: Morgan Kaufmann.

Mitchell, T. M. (1982). Generalization as search. *Artificial Intelligence*, *18*, 203–226.

Newell, A. (1982). The knowledge level. *Artificial Intelligence*, *18*, 87–127.

Nowlan, S. (1987). *Parse completion: A technique for inducing context-free grammars* (Technical Report AIP–1). Pittsburgh, PA: Carnegie-Mellon University, Department of Psychology.

Osherson, D., Stob, M., & Weinstein, S. (1985). *Systems that learn.* Cambridge, MA: Bradford Books/MIT Press.

Pao, T. W. (1969). *A solution of the syntactical induction-inference problem for a non-trivial subset of context-free languages* (Interim Report

69–19). Philadelphia, PA: University of Pennsylvania, Moore School of Electrical Engineering.

Pinker, S. (1979). Formal models of language learning. *Cognition*, *7*, 217–283.

Quinlan, J. R. (1986). The effect of noise on concept learning. In R. S. Michalski, J. G. Carbonell, & T. M. Mitchell (Eds.), *Machine learning: An artificial intelligence approach* (Vol. 2). Los Altos, CA: Morgan Kaufmann.

Reynolds, J. C. (March 1968). *Grammatical covering* (Technical Memorandum 96). Argonne National Laboratory.

VanLehn, K. (1983a). Human skill acquisition: Theory, model and psychological validation. In *Proceedings of the National Conference on Artificial Intelligence* (pp. 420–423). Washington, D.C.: Morgan Kaufmann.

VanLehn, K. (1983b). *Felicity conditions for human skill acquisition: Validating an AI-based theory* (Technical Report CIS–21). Palo Alto, CA: Xerox Palo Alto Research Center.

VanLehn, K. (1983c). The representation of procedures in repair theory. In H. P. Ginsberg (Ed.), *The development of mathematical thinking.* Hillsdale, NJ: Lawrence Erlbaum.

VanLehn, K. (1987). Learning one subprocedure per lesson. *Artificial Intelligence*, *31*, 1–40.

VanLehn, K., Brown, J. S., & Greeno, J. G. (1984). Competitive argumentation in computation theories of cognition. In W. Kintsch, J. Miller, & P. Polson (Eds.), *Methods and tactics in cognitive science.* Hillsdale, NJ: Lawrence Erlbaum.

Vere, S. (1975). Induction of concepts in the predicate calculus. In *Proceedings of the Fourth International Joint Conference on Artificial Intelligence* (pp. 281–287). Tbilisi, USSR: Morgan Kaufmann.

Appendix 1. Proof of Theorem 3

The following proof shows that there are finitely many reduced context-sensitive grammars for any given finite presentation. Context-sensitive grammars are used in the theorem not only because they give the theorem broader coverage, but because they make the proof simpler. The proof is a counting argument, and context-sensitive grammars are defined by counting the relative sizes of the left and right sides of their rules.

Definition 6 A grammar is a context-sensitive grammar if for all rules $\alpha \rightarrow \beta$, we have $|\alpha| \leq |\beta|$, where $|x|$ means 'the length of string x'.[14]

[14]Another definition of context-sensitive grammars requires that the rules have the

Definition 7 A context-sensitive grammar is *simple* if (1) for all rules $\alpha \rightarrow \beta$, $|\alpha| = |\beta|$ implies that β has more terminals than α, and (2) every nonterminal occurs in some derivation of some string.

Lemma 1 The longest derivation of a string s using a simple context-sensitive grammar is $2|s| - 1$.

Proof: Consider an arbitrary step in the derivation, $\alpha \rightarrow \beta$. If $|\alpha| = |\beta|$, then β must contain at least one more terminal than α, because the grammar is a simple one. Consequently, there can be at most $|s|$ such steps in the derivation, because there are $|s|$ terminals in the string. For all the other steps in the derivation, β must be at least one longer than α. There can be at most $|s| - 1$ such steps in the derivation, because the string is only $|s|$ long. So the longest possible derivation using a simple grammar is $2|s| - 1$, where $|s|$ steps have $|\alpha| = |\beta|$ and $|s| - 1$ steps have $|\alpha| \leq |\beta|$.

Theorem 3 There are finitely many simple reduced context-sensitive grammars for any given finite presentation.

Proof: There are finitely many positive strings in the presentation. By the lemma just proved, the longest derivation of each string α is $2|\alpha| - 1$, Therefore, the largest number of rule firings in deriving the positive strings is less than 2T, where T is the total of the lengths of positive strings:

$$T = \sum_{\alpha \in P+} |\alpha|$$

where P+ designates the set of positive strings in the presentation. If a grammar has more than 2T rules, then there must be rules that were not used in any string's longest derivation. Such rules can be eliminated from the grammar without affecting the existence of the longest derivations. So such a grammar is reducible. Thus, the largest possible reduced grammar will have less than 2T rules.

This result alone is not enough to show that there are finitely many reduced grammars, because the rules could in principle be arbitrarily long or contain arbitrarily many distinct symbols. However, if a rule is used in a derivation of some string in P+, its right side must be shorter than L, where L is the length of the longest string in P+. The rule's left side must also be shorter than L. If we could show that the grammar cannot have arbitrarily many symbols, then we would be done. Because the terminals in the rules must appear in the strings and there are finitely many strings, there are finitely many possible terminals in the rules. In fact, the largest

form $\alpha A \beta \rightarrow \alpha \gamma \beta$, where A is a nonterminal and γ is nonempty. The definition given above is from Hopcroft and Ullman (1969), who comment that the two definitions are equivalent.

number of terminals is T. Because each nonterminal must participate in some string's derivation (by simplicity), each nonterminal must appear in some rule's left side. There are less than 2T rules, each with at most L symbols on the left side, so there are less than 2LT possible nonterminals. Thus, the number of simple reduced grammars is finite because: (1) the number of rules is less than 2T, (2) the length of the left and right sides is at most L, and (3) there are less than 2LT+T symbols used in the rules. This completes the proof of theorem 3.

Appendix 2. Proof of Theorem 5

Theorem 5 states that the derivational version space contains the reduced version space. The critical part of the proof deals with the positive strings, P+, because both version spaces specifically exclude grammars that generate negative strings. Hence, we prove the following theorem, from which Theorem 5 follows immediately.

Algorithm A Given a set of strings P+, produce the grammars corresponding to all possible labellings of all possible sequences of all possible simple unlabelled derivation trees for each string.

Theorem 11 Algorithm A produces a set of grammars that contains all the reduced, simple grammars for P+.

To prove the theorem, we need to show that every reduced grammar is generated by the algorithm. Suppose that R is a reduced grammar. Because R generates every string in P+, it generates at least one derivation tree for every string in P+. First we will consider the case where R generates exactly one derivation tree for each string, then consider the case where R generates more than one derivation tree for some (or all) of the strings.

Since R generates exactly one derivation tree for each string, we merely need to show that that sequence of derivation trees is among the set of labelled parse tree sequences generated by the algorithm. Because R is simple, its parse trees must conform to the structural constraints that were imposed in generating the set of unlabelled derivation trees. In other words, if a node has just one daughter, then the daughter is a leaf. Moreover, the algorithm generates all such unlabelled derivation trees, so R's derivation trees must be among those generated by the algorithm. Thus, R's derivation trees are some labelling of one of the *unlabelled* derivation tree sequences. However, the algorithm generates all possible labellings of these. So R's parse trees must be among the set of *labelled* derivation tree sequences generated by the algorithm. The only way for R to have nonterminals other than those induced by labelling the unlabelled derivation trees would be for R to have rules that are not used in generating its derivation trees for P+. However, R is reduced, so this cannot be the case. Thus, R must be one of the grammars induced by the algorithm.

Next we consider the case where R generates more than one derivation tree for at least one string in P+. For each string p in P+, let Trees$_p$ be the set of derivation trees for p generated by R. As shown above, the algorithm generates at least one of the derivation trees in each Tree$_p$. From that derivation tree sequence, it generates rules for a grammar. The generated grammar will not be the same as R if some of the other derivation trees in Trees$_p$ use rules that are not in this derivation tree sequence. However, if this were the case, then those rules could be deleted from R, and yet all the strings could still be parsed. Thus, R would be reducible, contrary to hypothesis. So R must be generated by the algorithm. This completes the proof of the theorem.

Appendix 3. Proof of Theorem 10

The following theorem states that the derivational version space is properly represented by its boundaries:

Theorem 10 Given a grammar x in triple form and a derivational [G,S], x is in the derivational version space represented by [G,S] if and only if there is some g in G such that g FastCovers x, and some s in S such that x FastCovers s.

The 'if' half of the 'if and only if' follows immediately from the definition of G and S; if x is in the space, then there must be maximal and minimal elements above and below it. To show the 'only if' half, suppose that x is not in the space, and yet there is a g that FastCovers it and an s that it FastCovers. A contradiction will be derived by showing that x should be in the derivational version space. First, we show that x is consistent with the presentation. Because x FastCovers s, the language generated by x includes the language generated by s. Because s's language includes the positive strings of the presentation, so does x's language. Thus, x is consistent with the positive strings of the presentation. Because g FastCovers x, and g's language excludes all the negative strings, x's language must also exclude all the negative strings. So x is consistent with the negative strings. Therefore, x is consistent with the whole presentation. The remaining requirement for membership in the derivational version space is that x be a labelling of some tree sequence from the simple tree product of the presentation. Clearly, x is a labelling of the tree sequence which is the first element of its triple. Because x FastCovers s, it must have the same tree sequence as x, so its tree sequence is a member of the simple tree product. It follows that x should be in the derivational version space. This contradiction completes the proof of the theorem.