Learning Programs from Traces using Version Space Algebra

Tessa Lau
IBM TJ Watson Research
P.O. Box 704
Yorktown Heights, NY 10598
USA
tessalau@us.ibm.com

Pedro Domingos
Department of CS&E
Box 352350
University of Washington
Seattle, WA, 98195 USA
pedrod@cs.washington.edu

Daniel S. Weld
Department of CS&E
Box 352350
University of Washington
Seattle, WA, 98195 USA
weld@cs.washington.edu

ABSTRACT

While existing learning techniques can be viewed as inducing programs from examples, most research has focused on rather narrow classes of programs, e.g., decision trees or logic rules. In contrast, most of today's programs are written in languages such as C++ or Java. Thus, many tasks we wish to automate (e.g. programming by demonstration and software reverse engineering) might be best formulated as induction of code in a procedural language. In this paper we apply version space algebra [10] to learn such procedural programs given execution traces. We consider two variants of the problem (whether or not program-step information is included in the traces) and evaluate our implementation on a corpus of programs drawn from introductory computer science textbooks. We show that our system can learn correct programs from few traces.

Categories and Subject Descriptors

I.2.6 [**Artificial Intelligence**]: Learning—*Knowledge acquisition, induction*; I.2.2 [**Artificial Intelligence**]: Automatic programming—*Program synthesis*

General Terms

Algorithms, Experimentation, Human Factors

1. INTRODUCTION

Important tasks, like programming by demonstration or reverse-engineering of software artifacts, would be facilitated if one could induce programs from execution traces. But although machine learning algorithms induce a *type* of program from examples, only certain forms of target programs have been studied extensively. For example, decision-tree learners produce a single discrete-valued output via a sequence of input-value tests. Other learners, whether computing a single output value (e.g., classifiers) or a sequence of decisions (e.g., reinforcement learners), use similarly restricted representations. Attempts to learn programs of a more general form have focused mainly on functional and declarative languages (e.g., LISP, PROLOG), and preserve tractability by explicitly or implicitly embodying very strong biases. In reality, procedural languages like C++ or Java are much more prevalent, suggesting that much knowledge is expressed in such languages, and that it is important to easily incorporate a wide variety of biases when learning them.

Our goal here is to extend the applicability of machine learning by providing a framework and specific algorithms for inducing programs in procedural languages, avoiding as much as possible arbitrary *a priori* restrictions on the type of programs that can be learned. In order to make learning with such weak biases feasible with a realistic number of examples, we assume that, in addition to the program's inputs and outputs, the learner has access to a *trace* of the program's execution—the sequence of intermediate states produced by the program when applied to a given input. This is a realistic assumption in applications such as reverse engineering, where the state of the system may be observable though the source code is not, and programming by demonstration [11], where the intermediate states are results of a human's successive actions on a user interface, and the program being executed exists only in the user's mind.

This paper presents a language-neutral framework and implementation for learning programs in an imperative programming language (a subset of Python), given traces of a program's execution on specific inputs. We view a procedure as a set of program constructs (e.g., conditionals and loops) that join together sequences of

K-CAP'03, October 23–25, 2003, Sanibel Island, Florida, USA.

primitive program statements. Our approach is based on Lau *et al*'s version space algebra [10], which in turn extends Mitchell's version space framework [13]. Version space algebra allows an efficient exhaustive search of the space of programs consistent with the training traces.[1] It also enables explicit tailoring of the learning bias to include only the specific programming language statements defined by a grammar. As a result, our system learns correct programs from a remarkably short 5.1 traces on average.

This paper is organized as follows. Section 2 reviews prior work on version space algebra. Section 3 defines the problem of learning programs from traces. Section 4 presents and evaluates our approach for learning when program-step information is available, and Section 5 explains how learning is possible when execution traces are absent this information. Finally, we discuss related work and conclude.

2. VERSION SPACE ALGEBRA

In this section we review version space algebra [10]. A *hypothesis* is a function h that takes as input an element of its domain I_h and produces as output an element of its range O_h. A *hypothesis space* is a set of functions with the same domain and range. The *bias* determines which subset of the universe of possible functions is part of the hypothesis space; a stronger bias corresponds to a smaller hypothesis space. We say that a hypothesis h is *consistent* with a training example (i, o), for $i \in I_h$ and $o \in O_h$, if and only if $h(i) = o$. A *version space*, $\mathrm{VS}_{H,D}$, consists of only those hypotheses in hypothesis space H that are consistent with the sequence D of examples. When a new example is observed, the version space must be *updated* to ensure that it remains consistent with the new example. A version space may be *executed* on an input i to produce a set of outputs $\{o_1, o_2, ..., o_N\}$ such that each $o_j = h_j(i)$ for some h_j in the version space. In our framework, a *preference bias* may be set by the application designer by defining a probability distribution over the hypotheses in the hypothesis space, from which we can compute the probability of each hypothesis in the version space. In many cases, version spaces may be represented efficiently by constraints on the set of consistent hypotheses, such as the boundaries of the set relative to a partial order (one that is convex and definite [8], but not necessarily the generality ordering [10]).

For example, we define a $ConstInt$ version space containing functions of the form $f(x) = C$ for every integer-valued C. Given a training example (input: 0, output: 5), the version space is updated to contain only the function $f(x) = 5$. Further training examples will either be consistent with the version space's single hypothesis, or cause the version space to collapse to contain no hypotheses. This simple version space becomes more powerful when used as a building block in the version space algebra.

Version space algebra is a method for composing together version spaces to build a complex version space out of simpler parts. Version space algebra operators include

- **Union**: combine two or more version spaces to form one space containing the union of the functions in the member spaces,
- **Join**: combine two or more version spaces to form one space containing the cross product of the functions in the member spaces, subject to a consistency predicate, and
- **Transform**: convert one version space into another by mapping its functions to a different domain and range.

The transform operator is key to the modularity of the version space algebra. A simple version space such as $ConstInt$ may be transformed into a number of different domain-specific version spaces by the suitable definition of a pair of transformation operators that convert the inputs and outputs for one version space into inputs and outputs for the other. For example, we represent a version space containing constant assignment statements (e.g., $y = 4$) as a transform of a $ConstInt$ space, mapping each function $f(x) = C$ into an assignment statement $y = C$.

3. PROBLEM STATEMENT

We define the problem of learning programs from traces as follows. Let a *state* be the environment visible to our system during the execution of a program. We formalize a *program* as a procedure that, given an initial state S_0, produces a *trace*—a sequence of states $S_0, S_1, ..., S_n$ resulting from the execution of the procedure on S_0, such that the execution of a single program statement in state S_i produces state S_{i+1}.[2] The program's inputs and outputs are encoded in the trace. We say that a program is *consistent* with an example trace $T = \langle S_0, S_1, S_2, ..., S_n \rangle$ iff the program, when executed from state S_0 and given the inputs included in T, produces trace T.

Let the *bias* be the language denoting the set of allowable programs. We state the learning problem as follows:

> Given a language and one or more traces of a target program, output a program in the

[1] Although our approach is currently incompatible with noisy data, we plan to lift this limitation in future work, as Norton and Hirsh [16] have for concept learning.

[2] Nonterminating programs are an area for future work.

language that is consistent with all of the examples and generalizes correctly to future examples.

The contents of the program state affect the difficulty of the learning problem. We identify four configurations:

Incomplete The state contains a subset of the variables available to the program; some relevant variables are hidden.

State variables The state includes the complete set of variables available to the program.

Program step identifier The state includes the complete set of variables available to the program, and unique identification of the program step executed between each pair of consecutive states.

Fully visible The state includes the complete set of variables available to the program, unique identification of the program step, and a set of change predicates indicating whether each variable changes between each pair of consecutive states.

When the state is fully visible, the problem of learning programs is simply a matter of inferring the conditionals present in the program and the variabilized program statements that cause the observed changes in the state. The change predicates enable the learner to identify the portion of the state that has changed in this program step; without them, certain program statements (such as assigning to a variable the value it previously held) are more difficult to learn. If the program step identifier is not available, the program structure (i.e., loops and conditionals) must be inferred from the trace as well. Finally, if part of the state is hidden, the learner must consider a much larger space of hypotheses since the correct program may reference variables not visible in the state. The next section shows how to learn programs with loops and conditionals, given knowledge of the program step identifier. In section 5, we relax the program-step-identifier assumption and demonstrate learning a certain class of programs with only state variable information: loops with a fixed but unknown number of statements in the loop body. Learning with an incomplete state remains an item for future work.

4. LEARNING PROGRAMS FROM TRACES

A simple example illustrates the problem of learning programs from traces given knowledge of the program step identifier. Figure 1(a) shows an execution trace for a program that computes the greatest common divisor of two values. Figure 1(b) shows one program that is capable of producing this trace. Our SMARTpython system learns such a program given fourteen traces like the one shown in Figure 1(a).

Prog step	Var values i	j	k	Label
1	18	12	0	S_0
2	18	12	0	S_1
3	18	12	6	S_2
4	12	12	6	S_3
1	12	6	6	S_4
2	12	6	6	S_5
3	12	6	0	S_6
4	6	6	0	S_7
1	6	0	0	S_8
5	6	0	0	S_9

(a)

```
1: while j > 0:
2:         k = i mod j
3:         i = j
4:         j = k
```

(b)

Figure 1: (a) Sample execution trace for a program that computes the greatest common divisor of two values. (b) A program consistent with the trace.

Table 1: Grammar describing the class of programs learnable by our system. Italicized non-terminals are defined by domain-dependent grammars.

⟨Program⟩ ::= ⟨Statement⟩ [; ⟨Program⟩]
⟨Statement⟩ ::= ⟨*PrimitiveStatement*⟩
⟨Statement⟩ ::= IF ⟨*Condition*⟩ THEN ⟨Program⟩
ELSE ⟨Program⟩
⟨Statement⟩ ::= WHILE ⟨*Condition*⟩ DO ⟨Program⟩

4.1 Learning program structure

The grammar shown in Table 1 defines the language recognized by the learner, including control flow constructs such as loops and conditionals. The grammar is parameterized by two language-specific components: primitive program statements and Boolean conditional expressions. The grammar with instantiated components defines the set of programs in the hypothesis space of the program learner.

Control flow statements may be either a conditional IF statement that tests a condition and branches to one or another subprogram, or a WHILE loop that executes a subprogram as long as a condition holds. Assuming the program step identifier is available, the problem of learning programs reduces to learning each program statement individually based on the state before and after that statement's execution. For example, in Figure 1, states S_1/S_2 and S_5/S_6 are examples of states before and after program step 2 is executed.

Figure 2 shows the version space used to learn programs expressed in the grammar in Table 1. A program is a join of version spaces representing invididual program statements. The number of version spaces in the join is lazily determined after the first training example has been observed, allowing our system to learn programs

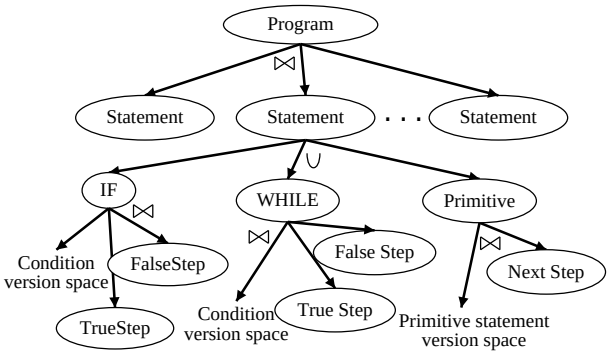

Figure 2: Version space hierarchy that defines the program structures corresponding to the grammar shown in Table 1. ∪ indicates a union and ⋈ indicates a join.

of any length. The *Primitive* space joins the space of primitive statements provided by the application designer with a *NextStep* space that contains the index of the next program step. The *IF* and *WHILE* spaces join a condition space (provided by the application designer) with two spaces indicating the next program step when the condition is true or false (the next program step version spaces are transformed $ConstInt$ spaces).

This version space is parameterized by the language-specific version spaces required to recognize statements and conditionals for a particular programming language. The next section describes our SMARTpython implementation for a simple imperative language.

4.2 Learning statements and conditionals

A program statement is a function that maps from one program state to another. A program statement is consistent with a pair of states $\langle S_i, S_j \rangle$ iff when executed in state S_i the statement produces state S_j. For example, an assignment statement executed in one state produces another state in which a new value has been assigned to the variable. A conditional statement produces a state in which the variable values are unchanged, but the program step has been updated, indicating which branch to follow.

For example, consider states S_1 and S_2 in Figure 1(a). In state S_1, variable k has value 0, while in state S_2, k has value 6. Assignment statements that are consistent with this state change include the correct statement $k = i \bmod j$, as well as the statements $k = 6$, $k = i - j$, $k = j - 6$, and so on.

Table 2 shows the grammar that defines the program statements recognized by the SMARTpython system. Figure 3 shows the corresponding version space. The version space hierarchy can be constructed in a straightforward manner from the grammar; future work will investigate automatic derivation of a version space from a grammar specification.

Table 2: Grammar describing the conditions and primitive statements supported in the SMARTpython system. The % symbol denotes the modulo operator. Combined with the grammar in Table 1, this grammar defines the language of programs expressible in SMARTpython.

⟨*Condition*⟩ ::= ⟨Var⟩ < ⟨Var⟩
⟨*Condition*⟩ ::= ⟨ArrayVar⟩ < ⟨Var⟩
⟨*Condition*⟩ ::= ⟨Var⟩ >= ⟨Var⟩
⟨*Condition*⟩ ::= ⟨ArrayVar⟩ >= ⟨Var⟩
⟨*Condition*⟩ ::= ⟨Var⟩ = ⟨Var⟩
⟨*Condition*⟩ ::= ⟨ArrayVar⟩ = ⟨Var⟩
⟨*Condition*⟩ ::= ⟨Var⟩ > Constant
⟨*Condition*⟩ ::= ⟨Var⟩ < Constant

⟨*PrimitiveStatement*⟩ ::= ⟨Var⟩ := ⟨Value⟩
⟨*PrimitiveStatement*⟩ ::= ⟨Var⟩ := ⟨Var⟩ ⟨Op⟩ ⟨Var⟩
⟨*PrimitiveStatement*⟩ ::= ⟨ArrayVar⟩ := ⟨Value⟩
⟨*PrimitiveStatement*⟩ ::= ⟨ArrayVar⟩ := ⟨Var⟩
⟨Var⟩ ::= i | j | k
⟨ArrayVar⟩ ::= A[⟨Var⟩] | A[0] | A[1] | ... | A[N]
⟨Value⟩ ::= Constant | ⟨Var⟩ | ⟨ArrayVar⟩
⟨Op⟩ ::= | + | - | * | / | mod

Each node in the tree represents a version space. A stacked node represents multiple parameterized version spaces. For instance, the *VarValue* node represents three version spaces containing assignment statements that assign to the variables i, j, and k. The *ArrayValue* node similarly represents assignments to array variables for constant indices 0 through N, and variable indices i, j, and k.

The union combines multiple version spaces of the same type together into a single collection of functions. For example, *VarValue(i)* is a collection of assignment statements to variable i, including constant assignment, constant increment, constant multiplier, and functions of other variables. The *VarValue* and *ArrayValue* spaces have the property that the consistency of all hypotheses in the space can be checked against most training examples with a single test; this information is provided by the application designer. When the property holds, large portions of the search space can be eliminated without examining each individual hypothesis. For example, if it is known that variable i's value has changed, then no assignment to a different variable can be consistent.

Edge labels in Figure 3 indicate transform functions. For example, the `i+=C` transform function converts a set of $f(x) = C$ functions into functions of the form $f(i) = i + C$. A transform (not shown) on the *Assignment* version space converts from the value of the $i + C$ expression into a program state in which the variable has been assigned its new value.

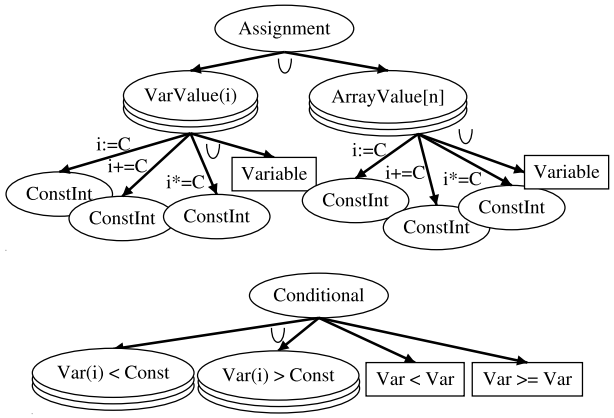

Figure 3: Version space for learning primitive statements (top) and Boolean expressions (bottom) in SMARTpython. Nontrivial transforms are shown as edge labels. Rectangular leaf nodes indicate enumerated version spaces, while oval leaf nodes indicate boundary-represented spaces.

Conditional statements cause control flow to branch to one part of the program or another depending on the truth value of a Boolean expression. By examining which step the program branches to after the conditional, we distinguish states in which the condition is true from those in which it is false. Equivalently, these states are positive and negative examples of the target condition. In Figure 1, states S_1 and S_5 are positive examples of the condition $j > 0$ in program step 1, while state S_9 is a negative example. The version space that represents conditionals of the form $j > C$ is represented using boundary sets. After the first positive example, in which j has the value 12, $\mathrm{VS}_{j>C}$ contains hypotheses of the form $j > C$ for $C > 12$. Since these hypotheses are totally ordered, we represent this version space using the boundaries 13 and ∞. Figure 3 shows the complete set of Boolean expressions supported as conditionals in SMARTpython.

4.3 Evaluation

We evaluate our approach on sixteen programs from the following sources:

- An introductory Fluency in Information Technology class designed for non-CS-majors; quiz and exam programs were translated from Visual Basic to the subset of Python supported by our system
- Introductory programming textbooks [4, 7, 3]
- Algorithms of our own implementation, e.g. bubble sort

The average program length is six statements. For each program, we randomly generate a set of 20 training examples and 100 test examples. Each example consists of a random initial state contructed by assigning a random integer between -100 and 100 to each named variable and to each element of a length-5 array. For the training examples, we execute the target program on the example's initial state to generate a trace. We then train the learner incrementally on the 20 training traces, testing its accuracy after each new training example. If the version space has not converged to a single program, we choose a highest probability program to execute.[3] The accuracy is the fraction of the 100 examples for which the learned program generates the correct trace. We repeat this procedure 25 times and average the results.

Figure 4 shows the accuracy versus number of training examples for the six most complex programs. On average, our system requires 5.1 examples to reach 100% accuracy. The greatest-common-divisor program achieves 100% accuracy with 14 traces. Learning from so few traces is possible because of the large amount of information in each trace; each iteration through a loop generates another example of the statements in the loop body. Visual inspection of learned programs showed them to be semantically equivalent to the correct program; where differences were observed, they were of the form "if i $\geq$ 0 then B else A" instead of "if i $<$ 0 then A else B". Conditionals proved to be the hardest to learn because randomly-generated traces did not adequately illustrate the decision boundary. If a human were to provide traces, such as in a programming by demonstration context, traces could be selected so as to provide maximal benefit to the learner, reducing the number of traces required for learning. While preliminary, these results indicate the feasibility of our approach and show that nontrivial programs can be learned from a small number of traces.

5. LEARNING WITHOUT A PROGRAM STEP IDENTIFIER

The previous section showed how to learn programs from traces given knowledge of the program step being executed at each point in the trace. However, in programming by demonstration (PBD) applications, where the system generalizes a program from actions performed directly on the user interface, it is not feasible to require the user to specify a program step for each action.

The SMARTedit system [10], for example, automated repetitive text-editing tasks by learning the actions a user performed in a text editor. The system assumed that the user was performing a task whose program structure consisted of a single loop with a fixed-length body. Instead of requiring explicit program step labelling, SMARTedit required users to manually *segment* the program trace by pressing a button at the beginning

[3]We use uniform probability distributions in all leaf spaces except the *Var(i) op Const* family, which are biased towards more specific hypotheses.

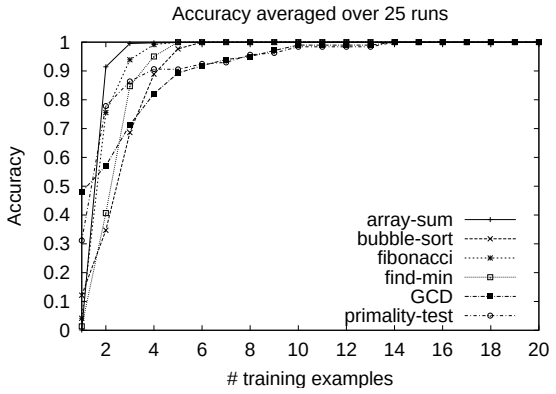

Figure 4: Accuracy versus number of training examples for a representative selection of programs.

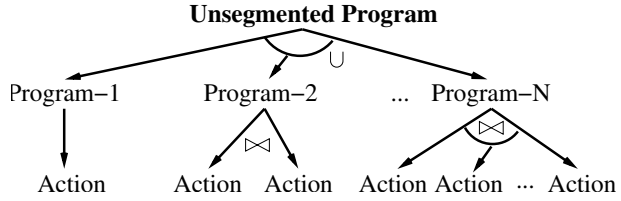

Figure 5: Version space for learning loops of unknown length

and end of every iteration of the task. For fixed-length loops, this segmentation enabled SMARTedit to infer the program step corresponding to each action in the demonstration. However, we believe that even manual segmentation imposes too much of a burden on the user.

We can extend our learning method to perform automatic segmentation by generalizing the version space. Specifically, we define an *UnsegmentedProgram* version space (Figure 5) as a union of one-step, two-step, ... N-step programs. These programs are composed of sequences of text-editing actions, which are the primitive program statements defined in the original SMARTedit system, and include moving the cursor to a new position, and inserting/deleting text. Actions are functions that map from one text-editing state to another; text-editing states include the contents of the file being edited, the location of the insertion cursor, and the contents of the clipboard; see prior work [10] for a complete description. The *Action* verion space in Figure 5 is the primitive statement version space of Figure 2.

We assume that the program structure consists of a single loop with a fixed number of statements in its body. Since the user will not provide segmentation information, the learner must infer the correct iteration boundaries from the trace. The version space is lazily constructed on receipt of the first training example,

Table 3: Exerimental results showing number of actions required to learn without program step identifier. The Iter column shows the same result measured in iterations; the Orig column shows the number of iterations required by the original SMARTedit system.

Scenario	Total	Train	Iters	Orig
columns	98	7	**1**	1
addressbook	84	27	**1.93**	2
grades	14	2	**1**	1
commentstyle	25	5	**1**	1
HTML-to-LaTeX	56	7	**0.875**	3
bindings	77	39	**5.57**	6
game-score	98	34	**2.43**	3
country-codes	32	8	**1**	1
pickle-array	468	73	**18.25**	19
xml-attribute	192	8	**1**	1

which bounds the maximum length of a single iteration.[4] For instance, suppose the user performs three text-editing actions: moving the cursor to a new position, inserting text, and moving the cursor again. The version space contains program hypotheses with one, two, or three actions per iteration. The *Program-1* version space shown in the figure is trained with each of the three actions in the trace as if they were all examples of the same program step; this version space collapses immediately because no hypothesized step is consistent with both moving the cursor and inserting text. The *Program-2* version space represents a program in which the two movement steps are each examples of the first step in the program. The *Program* version spaces are weighted probabilistically such that programs of length i are twice as likely as programs of length $i + 1$.

We evaluated this approach on a number of text-editing scenarios, listed in Table 3. The first five were used to test the original SMARTedit system. The remainder are: `bindings`, source code refactoring; `game-score`, used to evaluate the TELS PBD system [14]; `country-codes`, used to evaluate the WIEN wrapper induction system [9]; `pickle-array`, converting a list of numbers into a serialized form; and `xml-attribute`, converting elements in an XML file to a different schema.

Each scenario is comprised of an initial state, and a sequence of actions required to complete the task starting from the initial state. We measure the system's performance by training the learner on a prefix of the target sequence, and testing its performance on the remainder. The results of this experiment are shown in Table 3. The Train column shows the minimum num-

[4]We assume that the user performs at least one complete iteration of the target program.

ber of actions (out of the Total number of actions in the scenario) required to learn a program model that performs correctly on the remainder of the scenario.

The results show that the system is able to accurately identify the correct loop length, and infer the program step identifier, without requiring the user to manually segment each trace. For comparison, the table also shows the number of iterations required by both our system and the original SMARTedit system. Although our system is given strictly less information, in half the cases it outperforms the original system, requiring fewer iterations to learn a correct program. This result is due in part to the fact that the original experiment measured only complete iterations, whereas our experiment employs a finer-grained metric, measuring individual actions within each iteration. If a step early in each iteration requires more training examples than a later step, then the system will learn the correct program after being trained on only the first portion of an iteration, effectively reducing the number of iterations required to learn the program.

However, other factors allow our system to perform significantly better than the original. For example, the HTML-to-LaTeX scenario (e.g., turning <HTML> into $<$HTML$>$) shows a remarkable improvement: what used to require 3 iterations is learnable by our system with 0.875 iterations. The original SMARTedit was trained with 8 text-editing actions per iteration, four each to locate and escape the left and right angle brackets. Multiple iterations were needed in order to disambiguate the action that located the next angle bracket. By contrast our system, which makes no assumptions about the length of the iteration, found that a four-step program was able to accomplish the task, by generalizing a procedure to escape any angle bracket. With a 4-step iteration, seven training actions gave the system the two examples it required to correctly disambiguate the procedure, resulting in a large improvement over the original system.

6. RELATED WORK

Our work is closely related to automatic programming [18, 2, 6, 1, 12, 17]. Unlike language-specific systems, version space algebra provides a general method for learning programs in a wide variety of languages; its bias is defined by an input grammar. Our approach also takes advantage of version space decomposition and boundary set representability to efficiently conduct an exhaustive search of the space of programs.

The role of version space algebra in learning procedural programs is similar to the use of declarative biases in inductive logic programming [15]. In contrast to the heuristic search typically used in ILP systems, version space algebra allows efficient exhaustive search; future work will investigate its application to learning logic programs.

Compared to previous work using version space algebra for programming by demonstration [10], our work improves expressiveness by learning programs with loops, conditionals, and state variables, and shows that programs can be learned without knowledge of the program step identifier. In addition, we have formalized conditions as boundary-set representable version spaces and added a probabilistic framework. Our work on conditions is very similar to work on detecting invariants [5], which could be used as a module in our learner.

7. CONCLUSION

In this paper, we:

- Formalize the problem of learning programs from traces using version space algebra;
- Propose a domain-independent method for learning complete programs from traces, given a specification of statements and conditions in the target language;
- Apply this method to learning programs from traces in a high-level language with arrays, conditionals, and loops;
- Use it to learn text-editing programs by demonstration without explicit program step identifiers; and
- Experimentally validate our approach, showing it learns correct programs from a small number of examples.

Relaxing the assumptions we have made in this work is our primary goal for future work. Handling noisy or incomplete data is an important first step. One idea is to integrate heuristic search with the version space algebra. Although this paper has presented the version space algebra as combining independent version spaces, we expect that different learners, such as decision trees or neural networks, can be substituted for individual version spaces in the version space hierachy where appropriate.

Particularly important for the programming by demonstration application is learning from fewer traces. We plan to apply active learning to learn more accurately from fewer traces. We also plan to extend our model to handle more expressive control flow constructs (e.g. subroutines and recursion) and program statements (e.g. conjunctive conditions). Finally, we will conduct a formal analysis of our approach and investigate tradeoffs between expressiveness and learnability as we scale to larger and more complex programs.

8. REFERENCES

[1] D. R. Barstow. An experiment in knowledge-based automatic programming. *Artificial Intelligence*, 12:73–119, 1979.

[2] A. W. Biermann. On the Inference of Turing Machines from Sample Computations. *Artificial Intelligence*, 3:181–198, 1972.

[3] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. MIT Press, 1991.

[4] Edsger W. Dijkstra. *The Discipline of Programming*. Prentice Hall, Inc., Englewood Cliffs, NJ, 1976.

[5] M. Ernst, A. Czeisler, W. Griswold, and D. Notkin. Quickly detecting relevant program invariants. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE 2000)*, pages 449–458, Limerick, Ireland, June 2000.

[6] C. Green and D. Barstow. On program synthesis knowledge. *Artificial Intelligence*, 10:241–279, 1978.

[7] David Gries. *The Science of Programming*. Springer-Verlag, New York, NY, 1981.

[8] Haym Hirsh. Theoretical underpinnings of version spaces. In *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence*, pages 665–670. San Francisco, CA: Morgan Kaufmann, July 1991.

[9] N. Kushmerick, D. Weld, and R. Doorenbos. Wrapper Induction for Information Extraction. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence*, pages 729–737. San Francisco, CA: Morgan Kaufmann, 1997.

[10] Tessa Lau, Pedro Domingos, and Daniel S. Weld. Version space algebra and its application to programming by demonstration. In *Proceedings of the Seventeenth International Conference on Machine Learning*, pages 527–534, June 2000.

[11] Henry Lieberman, editor. *Your Wish is My Command: Giving Users the Power to Instruct their Software*. Morgan Kaufmann, 2001.

[12] Z. Manna and R. Waldinger. Knowledge and reasoning in program synthesis. *Artificial Intelligence*, 6, 1975.

[13] T. Mitchell. Generalization as search. *Artificial Intelligence*, 18:203–226, 1982.

[14] Dan Hua Mo. Learning Text Editing Procedures from Examples. Master's thesis, University of Calgary, December 1989.

[15] C. Nédellec, C. Rouveirol, H. Adé, F. Bergadano, and B. Tausend. Declarative bias in ILP. In L. de Raedt, editor, *Advances in Inductive Logic Programming*, pages 82–103. IOS Press, Amsterdam, the Netherlands, 1996.

[16] S. W. Norton and H. Hirsh. Classifier learning from noisy data as probabilistic evidence combination. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, pages 141–146, Menlo Park, CA, 1992. AAAI Press.

[17] C. Rich and R. Waters. The programmer's apprentice: A research overview. *IEEE Computer*, 21(11):10–25, 1988.

[18] David E. Shaw, William R. Swartout, and C. Cordell Green. Inferring lisp programs from examples. In *Proceedings of the Fourth International Joint Conference on Artificial Intelligence*, pages 260–267, Tblisi, Georgia, USSR, 1975. San Francisco, CA: Morgan Kaufmann.