Machine Learning, 53, 111–156, 2003

Programming by Demonstration Using Version Space Algebra

TESSA LAU tessalau@us.ibm.com
IBM T.J. Watson Research, P.O. Box 704, Yorktown Heights, NY 10598, USA

STEVEN A. WOLFMAN wolf@cs.washington.edu
PEDRO DOMINGOS pedrod@cs.washington.edu
DANIEL S. WELD weld@cs.washington.edu
Department of Computer Science & Engineering, University of Washington, Seattle, WA 98195, USA

Editor: Haym Hirsh

Abstract. Programming by demonstration enables users to easily personalize their applications, automating repetitive tasks simply by executing a few examples. We formalize programming by demonstration as a machine learning problem: given the changes in the application state that result from the user's demonstrated actions, learn the general program that maps from one application state to the next. We present a methodology for learning in this space of complex functions. First we extend version spaces to learn arbitrary functions, not just concepts. Then we introduce the version space algebra, a method for composing simpler version spaces to construct more complex spaces. Finally, we apply our version space algebra to the text-editing domain and describe an implemented system called SMARTedit that learns repetitive text-editing procedures by example. We evaluate our approach by measuring the number of examples required for the system to learn a procedure that works on the remainder of examples, and by an informal user study measuring the effort users spend using our system versus performing the task by hand. The results show that SMARTedit is capable of generalizing correctly from as few as one or two examples, and that users generally save a significant amount of effort when completing tasks with SMARTedit's help.

Keywords: programming by demonstration, adaptive user interfaces, version spaces, complex function learning

1. Introduction

Despite exponential increases in processing power, computers are still difficult to use. Although graphical user interfaces have helped to bring computing to nonspecialists, only programmers are able to customize and optimize their computing experience. Applications that provide a means of customization typically limit this customization to a selection of features envisioned by the application designer, or require the user to script the application using its programming interface. The former method is inflexible, and the latter is beyond the means of most computer users; both are usually limited to a single application.

Programming by demonstration (PBD) has the potential to bridge the gap between programmers and non-programmers, and allow anyone to automate repetitive, everyday tasks (Cypher, 1993; Lieberman, 2001). Placed at the operating system level, PBD would facilitate the automation of complex tasks involving multiple applications (Paynter, 2000).

Rather than writing and debugging statements in an obscure programming language, which may be completely decoupled from the visual appearance of an application, a user constructs a program by simply demonstrating the desired behavior of the program using the interface with which she is already familiar. From this sequence of user actions, the PBD system infers a program that is consistent with the observed actions and generalizes to new instances of the task.

The macro recorders available in many applications today are a powerful but degenerate form of programming by demonstration. Typical macro recorders capture the exact keystrokes entered by the user, and play them back on demand. However, these macros are brittle, and recording a macro that performs correctly in a wide variety of different situations is a daunting endeavor, which may require several attempts. Many seemingly simple tasks (such as numbering lines consecutively) are difficult to automate with macro recorders that simply play back a recorded sequence of keystrokes.

PBD user interfaces may resemble the keystroke-based macro interface. Instead of recording a literal sequence of keypresses, however, a PBD system generalizes from the demonstrated actions to a robust program that is more likely to work in different situations.

Previous approaches to programming by demonstration have employed heuristic, domain-specific algorithms for generalizing from user actions to an executable program. In contrast, our work is based on a general, machine learning approach. We define the problem as follows. A repetitive task may be solved by a program with a loop; each iteration of the loop solves one instance of the repetitive task. Given a demonstration, in which the user executes the first few iterations of the program, the system must infer the correct program.

Each action the user performs during this demonstration causes a change in the state of the application. Therefore, we model this problem as a machine learning problem of inferring the function that maps one state to the next, based on observations of the state prior to and following each user action. In this article, we describe a method called version space algebra for learning such complex functions from examples of their inputs and outputs. A version space (Mitchell, 1982) contains the set of all functions in a given language consistent with a set of input-output examples. The version space algebra allows us to compose together simple version spaces, using operators such as union and join, in order to construct more complex version spaces containing complex functions.

PBD presents a particularly challenging machine learning problem. Since users teach the system by performing a task manually, the creation of additional training data imposes a burden on the user (in our study, users did not want to demonstrate more than one or two iterations of the program). Thus the learner must be able to generalize from a very small number of iterations. In addition, the learning system must be able to interact gracefully with the user. It must present comprehensible hypotheses, and take user feedback into account. Our version space algebra framework addresses both of these issues.

We illustrate our version space algebra approach by applying it to programming by demonstration in the text editing domain. Figure 1 provides a simple example of the sort of task faced by users in this domain: numbering items in a shopping list. Figure 1(a) shows an example of a shopping list, and figure 1(b) shows the list after it has been numbered.

Imagine a user faced with tens or hundreds of items to number. Macro recorders, which merely replay a fixed sequence of keystrokes, could not be used to automate a task in which

```
apples
flour
butter
milk
strawberries
```

(a)

```
1. apples
2. flour
3. butter
4. milk
5. strawberries
```

(b)

Insert the row number followed by the string .␣
Move the cursor to the beginning of the next line

(c)

Figure 1. A text-editing task of numbering the items in a shopping list: (a) the initial text; (b) the text after the task has been completed; (c) a generalized program that completes the task.

the text to be inserted changes from one example to the next. Can our user do better than to perform the numbering by hand, or to write a script in an abstract programming language to accomplish the task? With SMARTedit, after the user numbers one item in the shopping list, our system learns a procedure such as the one shown in figure 1(c).

Our version space approach to programming by demonstration learns a program to automate this numbering task, and more, often from as little as a single training example. Our solution to this challenging machine learning problem makes the following contributions:

- Extending version spaces to complex-function learning;
- An algebra for composing simple version spaces into complex spaces;
- An application of version space algebra to programming by demonstration;
- An implemented system for PBD in text-editing;
- Exploration of a design space of PBD user interfaces and version spaces; and
- Experimental validation of our approach on a number of repetitive text-editing scenarios.

The remainder of this article is organized as follows. We begin with a formal characterization of programming by demonstration as a machine learning problem, and a sample of repetitive bibliographic tasks in the text-editing domain. Section 3 presents the version space algebra framework we have created to address the problem of programming by demonstration, and Section 4 discusses how the framework can be implemented to construct complex machine learning applications. In Section 5 we present our implemented system, SMARTedit, by describing both its user interface and the version spaces used in its construction. In Section 6, we discuss our empirical evaluation of SMARTedit. We conclude with a discussion of related work (Section 7) and future work (Section 8).

2. PBD: A machine learning formulation

We cast programming by demonstration as the machine learning problem of inferring generalized actions based on examples of the state changes resulting from demonstrated actions. For example, in the desktop domain, the action of opening the newly-created file `expenses.doc` causes the state of the desktop to change to reflect the opening of this document; actions that can be inferred from this state change include "open the file named `expenses.doc`" and "open the most recently created file." While both of these hypotheses are consistent with the demonstrated action, only one will generalize correctly to future examples. In a graphical drawing domain, the state change resulting from the action of changing the width of a line could be explained by such actions as "change the line width to 4," "change the line width to double the previous width," and so on.

In this article, we focus on programming by demonstration in the text-editing domain. Text-editing actions are functions that transform the application state into a new state. The application state (T, L, P, S) of a text editor includes: the contents of the text buffer T (a sequence of tokens in some alphabet, usually the ASCII character set), the cursor location L (a tuple (R, C) representing a row and column position within the text buffer), the contents of the clipboard P (a sequence of tokens), and a contiguous region S in T denoting the selection (highlighted text).

We define a comprehensive set of text-editing actions, including: moving the cursor to a new location, deleting a region of text, inserting a sequence of tokens, selecting a region of text, cutting and copying the selection to the clipboard, and pasting the contents of the clipboard at the current cursor location. Note that each text-editing action may correspond to a sequence of keystrokes in the user interface. For instance, a sequence of cursor key presses would all be part of the same text-editing action of moving the cursor to a new location.

We assume that we are given the application state before and after each text-editing action. This representation can be derived from knowledge of the application state at the beginning of the demonstration, and observation of the sequence of keypresses and mouse actions the user performs. By learning from the state sequence instead of the action trace, our system is robust to a significant class of user errors. For instance, by looking only at the final position of the cursor in a move action, our system is robust to errors where the user overshoots the target location and then corrects herself. By paying attention only to the state after the user has finished inserting a text string, our system is robust to a user making a typo and then backspacing over the erroneous text.

Repetitive tasks can be automated by a program that loops over each instance of the task and transforms one instance in each iteration. In our shopping-list example, for instance, each iteration numbers a different item in the shopping list. The user's demonstration generates a sequence of states representing the changes that arise as the user modifies the text buffer. The trace represents a prefix of the execution trace of the complete program. We state the learning problem as follows:

> Given a prefix of the target program's execution trace, in the form of a sequence of application states, output a program that is consistent with the observed trace and produces the correct behavior for the remainder of the trace.

We represent each statement in the program as a function that maps from one application state to the next. For example, moving the cursor from the first column in row 5 to the first column in row 6 causes the application state to change from one in which the cursor is at position (5, 1) to one in which the cursor is at position (6, 1). The action "move to the next line" is consistent with this state change, as is the action "move to the beginning of line 6". However, the action "move to line 7" is not consistent with this change, nor is the action "insert the string `hello`". Given the single state change in this trace, the learning algorithm outputs the set of programs consistent with this example.

In this work, we describe a novel machine learning approach for inferring programs from a partial trace of the program's execution. We model program statements as complex functions, and introduce a method for learning complex functions called version space algebra. Using version space algebra, we build up a search space of functions by composing smaller, simpler version spaces. For example, a function that moves the cursor to a new row and column position in the text file could be composed of two simpler functions: one that predicts the new row position, and one that predicts the column position.

2.1. Motivating examples

The previous section presented one very simple repetitive task (numbering lines in a shopping list). To give a flavor for the scope of tasks that can be automated with our system, here we decribe a few more complex tasks in a bibliographic editing context.

Consider the tasks involved in converting a list of bibliography entries from one format to another. For instance, an entry in BibTeX intermediate format is shown in figure 2. We define several interesting tasks over data files containing bibliographic entries such as this one.

Bibitem-newblock. Convert a list of BibTeX entries to a human-readable form by making the following transformation: delete the first line of the entry (the one beginning with `\bibitem`) and delete each of the `\newblock` commands. The number of `\newblock` commands varies from entry to entry; some bibliographic entries have two, while others have three or four.

Citation-creation. Automatically construct a citation for each bibliography entry, of the form "[Cypher, 1993]", and insert the citation at the beginning of each bibliography entry on its own line. The citation should be extracted from the first argument to the `\bibitem` command.

Number-citations. Number each bibliography entry from 1 to N using the format `[i]`, and insert the number at the beginning of the bibliography entry on its own line. For example,

```
\bibitem[Cypher, 1993][Cypher][1993]{cypher-wwid}
Cypher, A. (Ed.). (1993).
\newblock Watch what I do: Programming by demonstration.
\newblock Cambridge, MA: MIT Press.
```

Figure 2. A bibliography entry in BibTeX intermediate format.

the first entry should be numbered `[1]`, the second `[2]`, and so on up to the last entry in the list.

Our SMARTedit system is able to learn programs to accomplish these tasks and more from as little as a single iteration of the program. Section 6 discusses our experimental results. First, however, we present the version space algebra learning framework that allows us to learn programs from a small number of iterations.

3. Version space learning framework

Programming by demonstration requires a machine learning algorithm capable of inferring functions that map complex objects into complex objects. In this section, we present a framework for complex-function learning. Note that our framework is not specific to programming by demonstration; it is generally applicable to supervised machine learning problems where the learner is provided with examples of the target function.

3.1. Beyond classification learning

Our framework is based on Mitchell's version space formulation (Mitchell, 1982), in which concept learning is defined as a search through a version space of consistent hypotheses. A concept learning hypothesis is a function that maps from a complex object into a binary classification (or an n-way classification, in classification learning).

In contrast, our work is concerned with learning functions that map from complex objects to complex objects, such as from one application state to another. For example, we want to learn a programming instruction that is consistent with the user's action of moving the cursor from one location to another in the text file, or an instruction for inserting a particular sequence of characters.

One might think that concept learning suffices for this learning problem. For example, one may classify tuples consisting of (input, output) states into those that exhibit the correct transformation, and those that do not. However, although this representation could be used in learning to classify (input, output) state pairs, in practice one is not given the output state to classify. The learner must be able to act on a novel input state, and produce a plausible output state, not merely classify a pair of proposed states. Moreover, the complexity and variability of these states prohibits a generate-and-test methodology.

In order to introduce our version space extensions we first define our terminology. A *hypothesis* is a function that takes as input an element of its domain I and produces as output an element of its range O. 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 training example (i, o), for $i \in \mathit{domain}(h)$ and $o \in \mathit{range}(h)$, is *consistent* with a hypothesis 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. (A sequential training set is required for the definition of the version space join, below; extending the formalism to apply to non-sequential data is a direction for future work.)

When a new example is observed, the version space must be *updated* to ensure that it remains consistent with the new example by removing the hypotheses that are inconsistent with it. We will omit the subscripts and refer to the version space as VS when the hypothesis space and examples are clear from the context.

In Mitchell's original version space approach, the range of hypotheses was required to be the Boolean set {0, 1}, and hypotheses in a version space were partially ordered by their generality.[1] Mitchell showed that this partial order allows one to represent and update the version space solely in terms of its most-general and most-specific boundaries G and S (i.e., the set G of most general hypotheses in the version space and the set S of most specific hypotheses). The consistent hypotheses are those that lie between the boundaries (i.e., every hypothesis in the version space is more specific than some hypothesis in G and more general than some hypothesis in S). We say that a version space is *boundary-set representable* (BSR) if and only if it can be represented solely by the S and G boundaries. Hirsh (1991) showed that the properties of convexity and definiteness are necessary and sufficient for a version space to be BSR.

Mitchell's approach is appropriate for concept learning problems, where the goal is to predict whether an example is a member of a concept. In this work, we extend Mitchell's version space framework to any supervised learning problem (i.e., to learning functions with any range). Our extended version spaces are capable of learning a superset of the functions learnable with Mitchell's original version space framework; in particular, they can learn disjunctive concepts and compositions of them.

The original version space formulation took advantage of a generality partial ordering over hypotheses in the version space in order to efficiently represent the space by its boundaries. Observe, however, that the efficient representation of a version space by its boundaries only requires that some partial order be defined on it, not necessarily one that corresponds to generality. The partial order to be used in a given version space can be provided by the application designer, or by the designer of a version space library. The corresponding generalizations of the G and S boundaries are the least upper bound and greatest lower bound of the version space. As in Mitchell's approach, the application designer provides an update function $U(\mathrm{VS}, d)$ that shrinks VS to hold only the hypotheses consistent with example d.

An example of a version space that uses a partial ordering other than generality is the FindSuffix space, which contains hypotheses that output the first location where a particular search string occurs. Each hypothesis searches for a different string, and we order these hypotheses by the prefix relation over strings, which is not a generality ordering. See Section 5.2 for a more complete description of the FindSuffix space.

3.2. Version space execution

Although we have described how to update version spaces, we must also define how to use them. We are interested not only in the literal hypotheses in a version space, but in the result of applying those hypotheses to a novel input. We say that a version space is *executed* on an input by applying every hypothesis in the version space to that input, and collecting the set of resulting outputs. More formally, the result of executing a version space V on an input i

is denoted by $\text{exec}_V(i)$ and defined as follows:

$$\text{exec}_V(i) = \{o : \exists f \in V,\ f(i) = o\} \tag{1}$$

Note that two distinct hypotheses may agree in their predictions when applied to a new input. In Section 5.1, we discuss how version space execution is used to communicate the contents of the version space to the user for critiquing. Below, Section 3.5 extends the notion of execution to incorporate probabilities on hypotheses in the version space.

3.3. Version space algebra

We now introduce an algebra over these extended version spaces. Our ultimate goal is to learn functions that transform complex objects into complex objects. However, rather than trying to define a single version space containing the entire function space, we build up the complex version space by composing together version spaces containing simpler functions. Just as two functions can be composed to create a new function, two version spaces can be composed to create a new version space, containing functions that are composed from the functions in the original version spaces.

We define an *atomic version space* to be a version space as described in the previous section, i.e., one that is defined by a hypothesis space and a sequence of examples. We define a *composite version space* to be a composition of atomic or composite version spaces using one of the following operators.

Definition 1 (Version space union). Let H_1 and H_2 be two hypothesis spaces such that the domain (range) of functions in H_1 equals the domain (range) of those in H_2. Let D be a sequence of training examples. The *version space union* of $\text{VS}_{H_1,D}$ and $\text{VS}_{H_2,D}$, $\text{VS}_{H_1,D} \cup \text{VS}_{H_2,D}$, is equal to $\text{VS}_{H_1 \cup H_2,D}$.

Unlike the BSR unions proposed by Hirsh (1991), we allow unions of version spaces such that the unions are not necessarily boundary-set representable. By maintaining component version spaces separately, we can thus efficiently represent more complex hypothesis spaces.

Theorem 1 (*Efficiency of union*)**.** *The time* (*space*) *complexity of maintaining a version space union is the sum of the time* (*space*) *complexity of maintaining each component version space plus an* $O(1)$ *term.*

Proof sketch[2]**:** This analysis follows directly from the fact that the union is maintained by maintaining each component version space separately. □

In order to introduce the join operator, let $C(h, D)$ be a consistency predicate that is true when hypothesis h is consistent with the data D, and false otherwise. In other words, $C(h, D) \equiv \bigwedge_{(i,o)\in D} h(i) = o$.

Definition 2 (Version space join). Let $D_1 = \{d_1^j\}_{j=1}^n$ be a sequence of n training examples each of the form (i, o) where $i \in \text{domain}(H_1)$ and $o \in \text{range}(H_1)$, and similarly for

$D_2 = \{d_2^j\}_{j=1}^n$. Let D be the sequence of n pairs of examples $\langle d_1^j, d_2^j \rangle$. The join of two version spaces, $\mathrm{VS}_{H_1,D_1} \bowtie \mathrm{VS}_{H_2,D_2}$, is the set of ordered pairs of hypotheses $\{\langle h_1, h_2 \rangle \mid h_1 \in \mathrm{VS}_{H_1,D_1}, h_2 \in \mathrm{VS}_{H_2,D_2}, \mathrm{C}(\langle h_1, h_2 \rangle, D)\}$.

The interpretation of the joint hypothesis $\langle h_1, h_2 \rangle$ is left to the application designer. One interpretation of the join is as function composition. In SMARTedit, a join is used to sequence individual actions together to form a program. Suppose we have a version space that defines the set of all text-editing actions supported by our system. A two-way join of one action version space with another instantiation of the same version space thus represents the space of programs containing two actions.

Joins provide a powerful way to build complex version spaces by maintaining the cross product of two simpler version spaces, subject to the consistency predicate. In the general case, the entire cross product must be materialized in order to maintain the join. In many domain representations, however, the consistency of a hypothesis in the join depends only on whether each component hypothesis is consistent with its respective training examples, and not on some joint property of the two hypotheses. In this situation we say there is an *independent join*.

Definition 3 (Independent join). Let $D_1 = \{d_1^j\}_{j=1}^n$ be a sequence of n training examples each of the form (i, o) where $i \in \mathrm{domain}(H_1)$ and $o \in \mathrm{range}(H_1)$, and similarly for D_2. Let D be the sequence of n pairs of examples $\langle d_1^j, d_2^j \rangle$. Iff $\forall\ D_1, D_2, h_1 \in H_1, h_2 \in H_2\ [C(h_1, D_1) \wedge C(h_2, D_2) \Rightarrow C(\langle h_1, h_2 \rangle, D)]$, then the join $\mathrm{VS}_{H_1,D_1} \bowtie \mathrm{VS}_{H_2,D_2}$ is independent.

For example, consider learning an axis-parallel rectangular decision boundary on the Cartesian plane where each example input is a point on the plane and the output is a label, positive or negative, for each coordinate. This concept could be expressed as a join of two version spaces, one learning the target range for the x-coordinate and another for the y-coordinate. If the hypothesis space contains only axis-parallel rectangles, then the version space can be represented as an independent join. In that case, the joint version space may then be represented as the Cartesian product of the x-coordinate's version space and the y-coordinate's space. If the hypothesis space is allowed to include non-axis-parallel rectangles, however, this efficient representation is lost and the full cross product must be maintained.

We expect that application designers will design joins to take advantage of the independence property. Although we have not yet formalized general conditions under which joins may be treated as independent, Section 5 gives several examples of independent joins in the text editing domain.

Independent joins can be stored efficiently by representing each member version space of the join individually, and computing the cross product only when needed. Let $T(\mathrm{VS}, d)$ be the time required to update VS with example d. Let $S(\mathrm{VS})$ be the space required to represent the version space VS perhaps using boundary sets.

Theorem 2 (*Efficiency of independent join*). *Let $D_1 = \{d_1^j\}_{j=1}^n$ be a sequence of n training examples each of the form (i, o) where $i \in$ domain(H_1) and $o \in$ range(H_1), and let d_1 be*

another training example with the same domain and range. Define $D_2 = \{d_2^j\}_{j=1}^n$ *and* d_2 *similarly. Let D be the sequence of n pairs of examples* $\langle d_1^j, d_2^j \rangle$, *and let* $VS_{H_1,D_1} \bowtie VS_{H_2,D_2}$ *be an independent join. Then* $\forall\ D_1, d_1, D_2, d_2$

$$S\big(VS_{H_1,D_1} \bowtie VS_{H_2,D_2}\big) = S\big(VS_{H_1,D_1}\big) + S\big(VS_{H_2,D_2}\big) + O(1)$$
$$T\big(VS_{H_1,D_1} \bowtie VS_{H_2,D_2}, \langle d_1, d_2\rangle\big) = T\big(VS_{H_1,D_1}, d_1\big) + T\big(VS_{H_2,D_2}, d_2\big) + O(1)$$

Proof sketch: If the join is independent, then the set of hypotheses in the join is the cross product of the hypotheses in the two component spaces. In this case, it may be maintained by maintaining the two component version spaces separately. The time and space requirements for maintaining the join are then merely the sum of the time and space requirements for each component version space, plus a constant factor. □

If the join is dependent, then in the worst case the complete materialization of the cross product must be maintained, and the time and space requirements grow with the product of the time and space requirements for each component in the join. Efficient representation of non-independent joins remains an item for future work.

The application designer can decide whether or not to take advantage of an independent join. This choice represents a tradeoff between efficient maintenance and expressivity. For instance, some sequences of actions might not be possible, such as a paste action that does not follow a cut action. The consistency predicate should be defined to weed out illegal joint hypotheses such as these. For efficiency, however, the application designer could choose to declare this join an independent join and gain efficient maintenance of the joint space at the cost of possibly including illegal hypotheses in the join.

Note that our union operation is both commutative and associative, which follows directly from the properties of the underlying set operation. The join operator is neither commutative nor associative (for instance, when sequencing actions together to form a program, the order matters). Although we do not define an intersection operator because we have found no application for it thus far, its definition is straightforward.

Let a child version space denote a version space that is a component of another version space (its parent). For instance, a member of a union is the child of the parent union. The functions in a child version space may need to be transformed into a different type (i.e., different domain and/or range) before inclusion in the parent version space. One situation in which this is necessary is when functions in the child version space act on only a portion of the full input available to the parent, and return values in a simpler domain. Another use of the transform is to convert from an application-independent domain or range (e.g. integers) to application-dependent types (e.g. rows in the text buffer). For this purpose, we introduce the transform operator. Transforms may be used between a parent and a single child in the absence of other version space operators, or they may be associated with each child in a union or join.

Definition 4 (Version space transform). Let τ_i be a mapping from elements in the domain of VS_1 to elements in the domain of VS_2, and τ_o be a one-to-one mapping from elements

in the range of VS_1 to elements in the range of VS_2. Version space VS_1 is a *transform* of VS_2 iff $VS_1 = \{g \mid \exists_{f \in VS_2} \forall_j \; g(j) = \tau_o^{-1}(f(\tau_i(j)))\}$.

Transforms are useful for expressing domain-specific version spaces in terms of general-purpose ones, or more generally, converting functions with one domain and range to functions with a different domain and range. Transforms allow modular reuse of domain-independent version spaces. For instance, a version space that performed linear regression (i.e., output the best line fitted to a set of example points) could be reused in different application domains by transforming its input and output to the quantities available in each domain. Section 5 and Appendix A provide more examples of transforms in the SMARTedit application.

3.4. PAC analysis

Having defined the version space algebra, we now discuss whether these complex version spaces can be learned efficiently. PAC learning theory (Valiant, 1984; Haussler, 1988; Kearns & Vazirani, 1994) studies the conditions under which a reasonably-correct concept can be learned with reasonable confidence, or in other words, the conditions under which a concept is *probably approximately correct*. Although PAC analysis was originally developed for concept learning, it is easily extended to complex-function learning.

We assume inputs are drawn from an unknown probability distribution $\mathcal{D}$. We have no guarantee that the learned function will be close to the target function, because it might be based on misleading similarities in the observed sequence of training examples. In addition, unless the learner is given training examples corresponding to every possible element in its domain, there may be multiple hypotheses consistent with the given training examples, and the learner is not guaranteed to output the correct one. For example, we might have as input a list of papers all by the same author, and learn a function that incorrectly assumes that all authors share the same first and last names.

The true error of a hypothesis h, $error_{\mathcal{D}}(h)$, with respect to target function f and distribution $\mathcal{D}$ is the probability that h will produce the wrong output when applied to an input i drawn at random according to $\mathcal{D}$:

$$error_{\mathcal{D}}(h) \equiv \Pr_{i \in D}[f(i) \neq h(i)] \tag{2}$$

Let ϵ denote an upper bound on the error of the learned hypothesis, and δ be a bound on the probability that the learner will output a hypothesis with error greater than ϵ. Given desired values for ϵ and δ, PAC analysis places a bound on the number of examples required for learning.

Definition 5 (PAC learnability). For a target function class F defined over a set of inputs I of length n and a learner L using hypothesis space H, F is *PAC-learnable* by L using H iff for all $f \in F$, distributions $\mathcal{D}$ over I, ϵ such that $0 < \epsilon < 1/2$, and δ such that $0 < \delta < 1/2$, learner L will output a version space VS such that $\forall\, h \in \text{VS}$, $error_{\mathcal{D}}(h) \leq \epsilon$

with probability at least $(1 - \delta)$, using a sample size and time that are polynomial in $1/\epsilon$, $1/\delta$, n, and size(f).

Blumer et al. (1987) derive the following relationship between the required sample size m, the parameters ϵ and δ, and $|H|$:

$$m > \frac{1}{\epsilon}(\ln |H| + \ln(1/\delta)) \tag{3}$$

Thus a necessary and sufficient condition for m to be polynomial when using a hypothesis space H is that $|H| = O(e^{P(n)})$, where $P(n)$ is an arbitrary polynomial function of n.

We say that a version space operator $\diamond$ *preserves* PAC learnability if, given function class F_1 that is PAC-learnable using H_1, and function class F_2 that is PAC-learnable using H_2, $F_1 \diamond F_2$ is PAC-learnable using $H_1 \diamond H_2$.

Theorem 3. *The version space union, join, and transform operators preserve PAC learnability.*

Proof sketch: To show that each operator preserves PAC learnability, it suffices to show that the number of examples and the time required to maintain the composite space remain polynomial in $1/\epsilon$, $1/\delta$, n, and size(f).

Let F_1 be a function class that is PAC-learnable using H_1, and similarly for F_2 and H_2. Then $|H_1| = O(e^{P_1(n)})$ and $|H_2| = O(e^{P_2(n)})$. For the union, $|H_1 \cup H_2| \leq |H_1| + |H_2| = O(e^{P_1(n)}) + O(e^{P_2(n)}) = O(e^{\max(P_1(n), P_2(n))})$. For the join, $|H_1 \bowtie H_2| \leq |H_1| \times |H_2| = O(e^{P_1(n)}) \times O(e^{P_2(n)}) = O(e^{P_1(n)+P_2(n)}) = O(e^{P_{12}(n)})$. The transform is a one-to-one mapping so $|H|$ remains constant across a transform. Since the size of the composite hypothesis space remains at most exponential in $P(n)$, the number of training examples remains polynomial in $\ln |H|$.

If F_1 is PAC-learnable using H_1, then the time required to learn it is polynomial, and similarly for F_2 and H_2. By Theorem 1 and the previous paragraph, the time required to learn $F_1 \cup F_2$ using $H_1 \cup H_2$ is the sum of the times required to learn F_1 and F_2, which is polynomial in n. By Theorem 2 and the previous paragraph, the time required to learn $F_1 \bowtie F_2$ using $H_1 \bowtie H_2$ is polynomial in n. The transform operator is a one-to-one mapping, so it requires no additional time to maintain. Since the number of examples and time remain polynomial across the union, join, and transform operators, the operators preserve PAC learnability. □

Note that the join of an unlimited number of version spaces does not preserve PAC-learnability, because the required number of examples is worst-case exponential in the number of version spaces being joined.

3.5. Probabilistic version space framework

We have constructed a probabilistic framework around the version space algebra that assigns a probability to each hypothesis in the version space. Such a probabilistic framework allows us to:

- *Choose better hypotheses during execution*: Even before the version space converges to a single hypothesis, it may still be executed on new instances to produce a set of consistent outputs. The probabilistic framework allows us to choose the most likely output.
- *Introduce domain knowledge*: An application designer probably has *a priori* knowledge about which hypotheses are more likely to occur than others. For example, when trying to reason about a user's decision to move the cursor through an HTML file, a hypothesis of "634 characters forward" is probably less likely than the hypothesis "after the next occurrence of the string `<TABLE>`". This knowledge can be gathered either empirically, such as by conducting user studies, or it may be set heuristically by the application designer. The probabilistic framework allows the designer to cleanly introduce this type of domain knowledge into the system. These probabilities could also be updated over time by adapting to the user and giving higher probability to the types of hypotheses she employs.
- *Provide support for noisy data*: In a traditional version space, inconsistent hypotheses have zero probability and are removed from consideration. In a probabilistic framework, one can assign a small but non-zero probability to inconsistent hypotheses, such that noisy data does not irretrievably remove a hypothesis from the version space.

To use the framework, the application designer may optionally provide *a priori* probabilities as follows:

- A probability distribution $\Pr(h \mid H)$ over the hypotheses h in a hypothesis space H, such that $\sum_{h \in H} \Pr(h \mid H) = 1$.
- A probability distribution $\Pr(V_i \mid W)$ over the version spaces V_i in a union W, such that $\sum_i \Pr(V_i \mid W) = 1$.

Each of these probability distributions defaults to the uniform distribution if not otherwise specified. The application designer is not required to specify any probabilities in order to use the probabilistic framework.

The probability of a hypothesis h in an atomic version space V, $P_{h,V}$, is initialized to $\Pr(h \mid H)$. As the atomic version space is updated, the inconsistent hypotheses are removed from the version space. The probabilities of the remaining hypotheses are then renormalized. A consistent hypothesis relative to the version space has probability $\Pr(h \mid V)$, and an inconsistent hypothesis relative to the version space has zero probability.[3]

Thus the overall probability of a hypothesis h in a version space V, $P_{h,V}$, is defined inductively up from the bottom of the version space hierarchy depending on the type of V:

Atomic: V is atomic. In this case, $P_{h,V} = Pr(h \mid V)$ and the probabilities are normalized.

Transform: V_1 is a transform of another version space V_2. Thus $V_1 = \{h \mid \exists_{f \in V_2} \forall_i \, h(i) = \tau_o^{-1}(f(\tau_i(i)))\}$. Since transforms are one-to-one, $P_{h,V_1} = P_{f,V_2}$ and no normalization is necessary.

Union: V is a union of version spaces V_i with corresponding probabilities w_i. Since the version spaces may not be disjoint, the probability of a hypothesis h is the sum of its

weighted probabilities:

$$P_{h,V} = \sum_{\forall\, i | h \in V_i} w_i \times P_{h,V_i} \tag{4}$$

If one of the version spaces in the union collapses and contains no hypotheses, then the probabilities of the remaining hypotheses in the union must be normalized.

Join: V is a join of a finite number of version spaces V_i. Thus, $V = \{\langle h_1, h_2, \ldots, h_n\rangle \mid h_i \in V_i, \mathrm{C}(\langle h_1, h_2, \ldots, h_n\rangle, D)\}$. In this case, $P_{h,V} = k \times \prod_i P_{h_i,V_i}$, where k is a normalizing constant. This constant accounts for the probability mass lost due to the consistency predicate C not holding for all the hypotheses in the Cartesian product. If the join is independent, the consistency predicate always holds, so the normalizing constant is unnecessary.

Using this framework, each hypothesis in the version space has an associated probability. If the execution of a version space V on an input i produces a set of outputs o, we assign a probability P_V^o to each of these outputs:

$$P_V^o(i) = \sum_{\forall f | f(i)=o} P_{f,V} \tag{5}$$

We employ version space execution and the probabilistic framework to present the results of the learner to the user. Our SMARTedit system presents a list of output states to the user, ranked by probability, and allows her to select the correct one by cycling through the different output states. Section 5 describes SMARTedit in more detail.

4. Application design with version space algebra

Choosing an appropriate representation is typically one of the most critical decisions in formalizing a problem as a machine learning problem (Langley & Simon, 1995). The version space algebra framework enables application designers to build machine learning applications more easily in domains unsuitable for the customary classification representation, such as the PBD domain. By providing a framework for learning complex functions (functions that map from a complex input to a complex output), we hope to facilitate the use of machine learning in new classes of domains.

In order to use the version space algebra to model an application, the application designer must specify a set of atomic and composite version spaces and designate a single target space, as shown in Table 1. The role of atomic spaces, composite spaces, and the target space in version space algebra is analogous to the role of terminal symbols, nonterminal symbols, and the start symbol in a context-free grammar. The simplest update function examines each hypothesis in the version space individually, and discards the inconsistent hypotheses. A more efficient update function represents only the boundaries of the consistent set, and updates only the boundaries given each training example. The partial order is a means to this end. Similarly, the simplest execution function separately computes the output for each hypothesis in the version space and assigns probabilities to outputs accordingly. However,

Table 1. Domain description to be provided by the application designer in order to use the version space algebra framework.

For each atomic version space:
1. Definition of the hypothesis space.
2. A partial order on the hypothesis space.
3. An update function that updates the version space to contain only those hypotheses consistent with a given example.
4. An execution function that computes a probability for each possible output given an input.
For each composite version space:
1. Formula expressing it in terms of atomic version spaces, previously-defined composite spaces, and version-space-algebraic operators.
2. Transformation functions that convert examples for the composite version space into the corresponding examples for each component version space.
3. An execution function that takes the outputs (and probabilities) of the component spaces' execution functions and produces a probability for each possible output of the composite space.
The target composite version space.
(One of the previously-defined composite spaces.)

it may be possible to find the probability for each output more efficiently; we show an example in the next section. When neither approach is feasible, approximate probabilities may be computed by sampling from the version space; this area holds promise for future research.

Many PBD researchers have found that incorporating domain knowledge into a learning system is difficult. While we do not claim to have completely solved this problem, we believe that the version space algebra framework makes it easier to incorporate domain knowledge into a PBD application by crafting the version space appropriately.

Our framework also enables creation of a library of reusable, domain-independent version spaces that can greatly speed up this process. In our experience creating the SMARTedit system, we have begun the construction of such a library of reusable version spaces. The framework supports incremental design of the learning application, with new version spaces being added as necessary. The modularity and simplicity of the version space approach support fast construction of new version spaces, and facilitate rearrangement of the hierarchical version space structure as the application matures. Knowledge acquisition tools to automate the version space construction task would be an interesting direction for future research.

One of the primary responsibilities of the application designer is to craft the tradeoff between expressiveness and sample complexity for a particular application by selecting appropriate version spaces for inclusion in the version space hierarchy. An appropriate choice of bias can make the difference between a system that learns from few examples and a system that is incapable of generalizing. In the following section, we present an extended example of application design with the version space algebra in the context of our SMARTedit system.

5. The SMARTedit PBD system

In order to test our version space algebraic framework for PBD, we implemented a system called SMARTedit in the domain of text editing. SMARTedit was implemented from scratch using Python and the Tk toolkit. We have also ported it to Emacs, where it was configured to replace the builtin macro recorder. It has been used for a wide range of repetitive tasks, including manipulation of highly-formatted columnar data, transformation of semi-structured XML or bibliographic data, editing programming language code, and writing unstructured texts.

SMARTedit's learning component uses the version space algebra presented in Section 3. In this section, we first describe SMARTedit's user interface, then give a tour of some of its more interesting version spaces. A complete summary of all version spaces used in SMARTedit is given in Appendix A.

5.1. SMARTedit user interface

We have designed SMARTedit's user interface to make it accessible to non-programmer users. Rather than constructing a program in an arcane programming language, we enable users to build the program by demonstrating its effects on concrete examples. Our target users are familiar with the basic operation of an application, and know how to accomplish tasks through direct manipulation. They are not required to know the programming interface to the application.

We illustrate SMARTedit's user interface on a simple repetitive task: deleting the HTML comments, including their contents, from a text file (figure 3).

The four buttons across the top of the interface control SMARTedit's programming by demonstration functionality. To begin creating a SMARTedit program, the user pushes the "Record iteration" button. The button turns red, indicating that her demonstration is being recorded, and its function changes to "Stop recording". The user then begins demonstrating what she wants SMARTedit to do. In this case, the user presses arrow keys to move the cursor to the beginning of the first HTML comment (in front of the text `<!--`) then presses the delete key until the extent of the comment (up until the `-->`) has been deleted. SMARTedit records the sequence of states that result from the user's editing commands, and learns functions that map from one state to another.

Unlike standard macro recorders, our PBD system disregards the actual keypresses a user performs at each step. For instance, a user may be performing a search, but if she sees the next occurrence of the search term on the screen she could use the mouse or cursor keys to position the cursor rather than invoking the search facility. SMARTedit is robust to this kind of behavior because it does not learn from the literal keypresses. Instead it collapses sequences of similar events together (e.g., series of cursor-key presses, invocations of the search dialog, and moving the cursor with the mouse) under the assumption that these all are part of the same high-level action (e.g., moving the cursor). SMARTedit thus learns from the application state before and after the entire sequence of events; the important feature is the resulting state change, not the keystrokes by which that state change was accomplished. An interesting direction for future work would be to use the literal keypresses to bias the

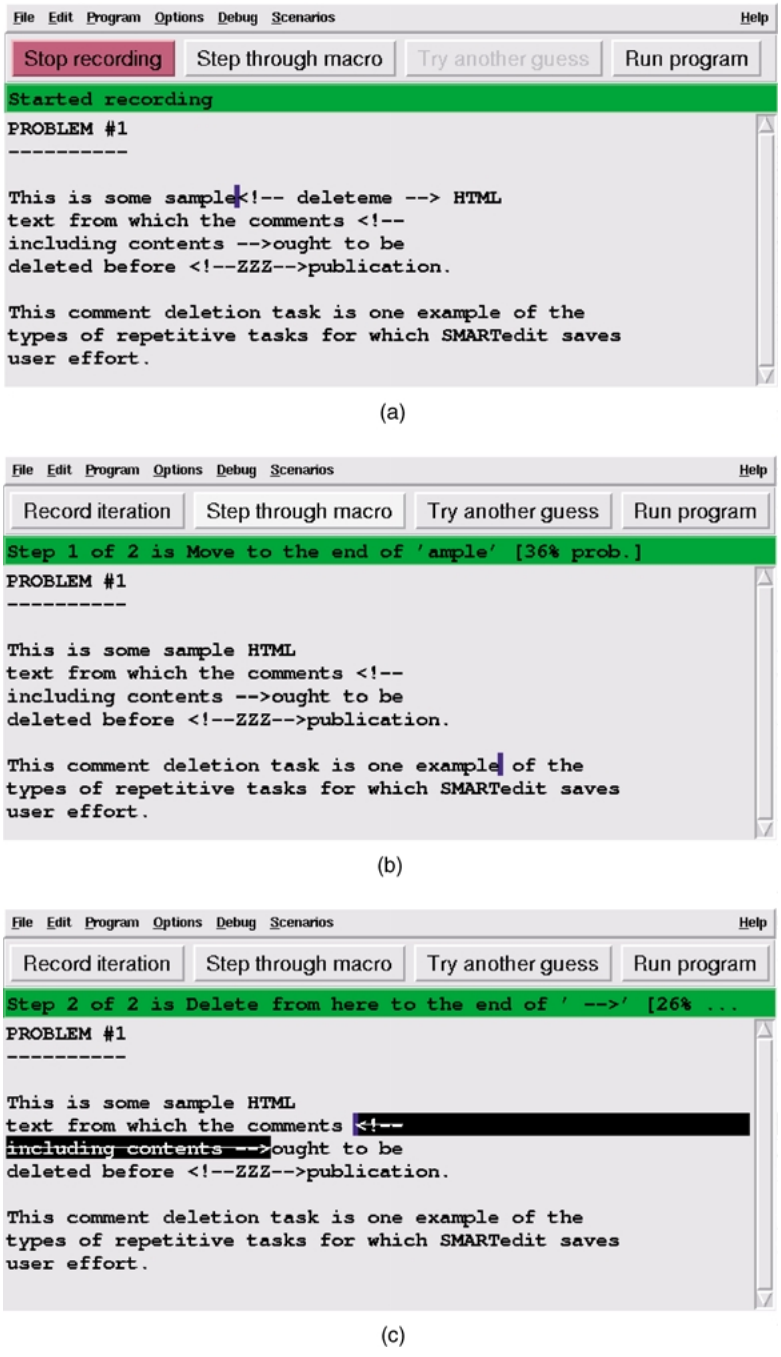

Figure 3. User interface for the SMARTedit system.

probabilities during learning. For instance, it is highly likely that the user is performing a search when she invokes the search facility.

Once the user has completed one iteration of the repetitive task, she clicks the red button to indicate that she has completed the iteration. At this point, SMARTedit initializes the version space using the recorded state sequence as an example. In this example, the user performed two actions: moving the cursor to the beginning of the HTML comment, and deleting to the end of the comment. This demonstration results in a sequence of three states: the initial state of the application (with the cursor at the top of the text file), the state after the cursor has been moved, and the state after the text has been deleted.

SMARTedit updates the version space lazily as the user provides more demonstrations, which allows it to consider infinite version spaces that are only instantiated on receipt of a training example. Given these observations, SMARTedit's version space contains a number of candidate hypotheses for the first step of the program, including:

– Move to row 4 and column 21,
– Move forward forty-two characters,
– Move to after the string `sample`,
– Move to after the string `ample`,
– Move to before the string `<`, and
– Move to before the string `<!--`

At this point the user may either provide more demonstrations (by continuing to execute the program on subsequent iterations), or ask SMARTedit to execute its learned procedure by pressing the "Step" button. In our example, pressing the "Step" button causes SMARTedit to display its first guess: it shows the cursor moving to the second paragraph of the text, at the end of the word `example`, with 36% probability (figure 3(b)). The best hypothesis (shown in the green bar at the top) explains that SMARTedit is moving after the word `ample` (because the word `sample` happened to precede the first HTML comment).

SMARTedit has performed the wrong action, so the user corrects its prediction by invoking the "Try another guess" button. Each press of this button causes SMARTedit to cycle to its next most likely prediction. Once SMARTedit shows the correct action, the user presses the "Step" button to continue on to the next step. The current implementation of SMARTedit always makes a prediction, even if the best hypothesis has very low probability. In a related paper (Wolfman et al., 2001), we describe a mixed-initiative interface layered on top of SMARTedit that automatically chooses the best interaction mode, such as presenting a hypothesis or asking the user to provide more demonstrations.

Continuing with our HTML comment deletion example, SMARTedit's second guess is correct (moving the cursor to the beginning of the next HTML comment, with 26% probability). The user then presses the step button to move on to the second step in the program. SMARTedit's best guess for the second action is correct: delete the extent of the comment, with 26% probability (figure 3(c)).

SMARTedit learns from the user's actions even during the stepping phase, so that on the third iteration of the program it will have learned both from the first iteration (demonstrated by the user) as well as from the second (in which the user corrected SMARTedit's

predictions). When the user selects the correct action, SMARTedit updates its version space with this new information. Any hypothesis that is inconsistent with this new example (including those hypotheses that supported SMARTedit's first, incorrect, prediction) are removed from the version space. For instance, the `sample` and `ample` hypotheses are discarded from the version space because they are inconsistent with the second iteration, as are all the row and column hypotheses. The correct hypothesis (move to the string `<!--`) is consistent with both iterations, so it remains in the version space.

In our example, when the user presses the "Step" button to begin deleting the third HTML comment, SMARTedit correctly moves the cursor to the third comment, with very high (93%) probability. Once the user is confident that SMARTedit has learned the correct program, she can use the "Run program" button to ask SMARTedit to execute its program on the remainder of the text file without any further user input.

Each of SMARTedit's predictions is drawn from the hypothesis space constructed using our version space algebra. The next section highlights some of the more interesting version spaces used in SMARTedit. A complete listing of the version spaces used in SMARTedit can be found in Appendix A.

5.2. SMARTedit version spaces

The complete version space hierarchy is shown in figure 4. First we describe how SMARTedit learns a single action. Then we show how to learn complete programs, an action at a time.

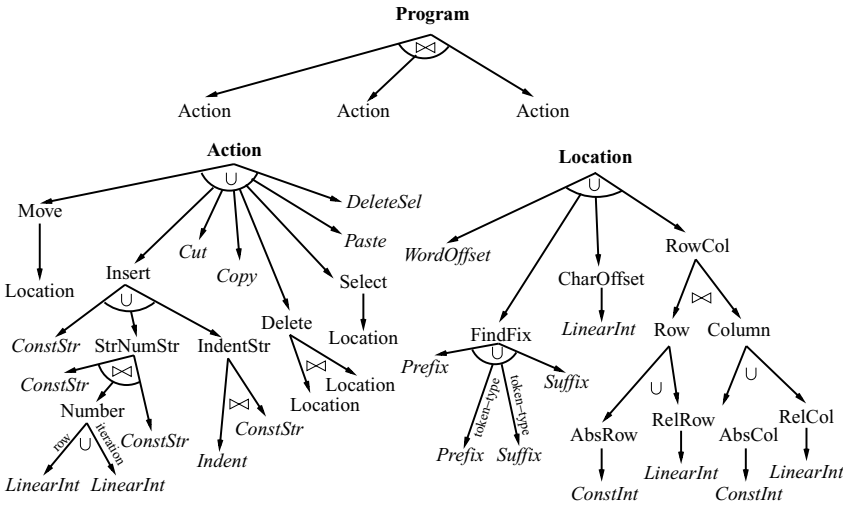

Figure 4. Version spaces used in the SMARTedit programming by demonstration system. The upper tree shows the complete version space for a Program, expressed in terms of Action version spaces (lower left tree). Action version spaces are in turn expressed in terms of Location version spaces (lower right tree). Italicized text denotes an atomic version space, while regular text denotes a composite version space. Edge labels denote transformation operations; for clarity, we omit most transformation operations from the figure.

5.2.1. Learning actions. The lower-left hierarchy in figure 4 shows the Action version space used to learn text-editing actions. Each function in the Action version space maps from one application state to another. A training example for this version space consists of the pair of states before and after the target action is performed, and the goal of learning is to produce the function (text-editing action) that represents the correct mapping from input state to output state.

The Action version space (figure 4) consists of a union of the different types of text-editing actions supported by SMARTedit: moving the cursor, inserting and deleting text, and manipulating the clipboard. For instance, the Move member of the Action union denotes the set of all actions that move the cursor to a different location in the text buffer. Similarly, the Select member of the union denotes the set of all actions that highlight a region of text starting from the current cursor position and ending in a new location.

Many of these text-editing actions are expressed in terms of locations in the text buffer. We encapsulate a useful collection of ways to specify a location within the buffer in the Location version space. The Location version space unions together a variety of different version spaces specifying different types of cursor positioning, such as string searching or row and column positioning. A version space transform is used to convert the functions in a Location space to the functions in the parent Move (or Select, etc.) space. For instance, consider a Location function that outputs the location on the same row but five columns to the right of the current cursor. Transformed, the corresponding Move function would map from the complete application state to a new state in which the cursor is positioned five columns to the right.

One way to specify a location in the buffer is by its row and column coordinates. We designed the RowCol version space (a member of Location) as an independent join of a Row version space and a Column version space. The Row functions that map from the current row coordinate to a new row and column functions are defined similarly. The Row space consists of a union of two spaces: AbsRow (absolute row positioning, such as "on row 5") and RelRow (relative positioning, such as "on the next row"). AbsRow is a transformation of a domain-independent version space drawn from our library of generic version spaces: ConstInt, which holds the set of functions $f(x) = C$ for any integral constant C. The transformation converts from functions with integral domain and range to functions with row coordinates as domain and range. Similarly, RelRow is a transformation of the domain-independent LinearInt version space, which contains functions $f(x) = x + C$ for any integral constant C. All of these row and column version spaces join together to allow a user to specify locations such as "on the next row and the 5th column" or "on the previous row and the first column".

Another way to specify a location in the text file is relative to the next occurrence of a string. If the cursor is positioned after (before) a string, we say that the user is searching for the next *prefix* (*suffix*) match. Suppose the user has moved the cursor to the end of the next occurrence of the string "PBD". From the system's point of view, the user may have been searching for the prefix "PBD", the prefix "BD", the prefix "D", or a superstring of "PBD". If, however, a user skipped over an occurrence of the string "D" while moving to "PBD", then the "D" hypothesis is not consistent. The FindPrefix and FindSuffix version spaces represent these types of string-searching hypotheses.

More formally, the FindPrefix and FindSuffix hypothesis spaces include functions that map from an input state to a location. Each function in the FindSuffix space maps from the input state to the location at the beginning of the next occurrence of some string T. We choose the partial order for FindSuffix according to a string prefix relationship in the string T; if f is a function of T_1 and g is a function of T_2, then $f \prec g$ iff T_1 is a proper prefix of T_2. (FindPrefix is defined similarly.) For clarity, we omit the function notation and simply refer to the function as the string on which it depends. The least upper bound (LUB) and greatest lower bound (GLB) boundaries of the FindSuffix version space are initialized to be $\mathcal{S}$ and $\mathcal{C}$ respectively, where $\mathcal{S}$ is a token representing the set of all strings of length K (some constant greater than the maximum text buffer size) and $\mathcal{C}$ is a token representing the set of all strings of unit length. When the first example is seen, the LUB becomes the singleton set containing the contents of the text buffer following the cursor (a sequence of tokens), and the GLB becomes the singleton set {"c"}, where c is the token immediately following the cursor. After the first example the LUB and GLB will always contain at most one member each. Given a new training example in which the string T follows the cursor, and the LUB contains the string S, the LUB is updated to contain the longest common prefix of S and T. If there is no common prefix, the version space collapses to the empty set. The GLB is updated based on the strings that were skipped over between the starting location and the final position. It is updated to contain the longer of its previous value and the shortest prefix of S which was not skipped (and thus invalidated as a hypothesis) between the starting and ending cursor locations. If S itself was skipped, the version space collapses.

Note that the FindPrefix and FindSuffix spaces are partially ordered by a string prefix/suffix relation, but not by the generality partial order. For example, if a FindSuffix hypothesis were to return the *set* of locations matching the search string, then two FindSuffix hypotheses would be ordered by the generality ordering, since one hypothesis would return a superset of the locations returned by the other. However, in our formulation, each string search hypothesis returns only a single location corresponding to the next occurrence of the string. Two string search hypotheses, one of which is a prefix of the other, may not necessarily produce the same location when executed in a given input state. For instance, the first occurrence of the string `th` in this paragraph is not the same as the first occurrence of the string `the`. Thus, while the two hypotheses are partially ordered by the string prefix relation, they are not ordered by the generality relation.

Note that we can execute the FindPrefix and FindSuffix version spaces efficiently without having to materialize the complete version space and execute each hypothesis individually. Consider the FindSuffix space (FindPrefix is handled similarly). This version space is equivalent to a set of strings, the longest of which is the string in the LUB, the others being some prefix of the string in the LUB. Intuitively, if one finds the longest string match first, then all prefixes which have not matched previously will also match at this location, and these shorter string hypotheses do not have to be executed individually. The execution maps each string in FindSuffix to the cursor location corresponding to its first occurrence in the text file following the location of the cursor. We can perform this search efficiently in time $O(st)$ where s is the length of the string and t is the length of the text file, by comparing the string against the text starting at every position in the text file following the cursor position. Let p_s denote the probability of the hypothesis containing the string s. The algorithm is as follows:

find the first location where the first k_1 characters of the text and search string match. Denote the matching string as $s[1, k_1]$. Output this location, with associated probability

$$p = \sum_{i=1}^{k_1} p_{s[1,i]} \tag{6}$$

Continue searching from this occurrence for a match of at least length $k_2 > k_1$. Output the next matching location with probability

$$p = \sum_{i=k_1+1}^{k_2} p_{s[1,i]} \tag{7}$$

Repeat until all prefixes have been matched or the end of text is reached, and return the set of output locations and their probabilities as the result of the execution.

Within each FindPrefix and FindSuffix version space, we provide a probability distribution over the hypotheses in the version space. We define a bell-shaped probability distribution such that strings consisting of exactly five characters are most likely, while shorter or longer strings are *a priori* less likely.[4] We chose this distribution heuristically, based on empirical observations. This probability distribution explains why, in the HTML comment deletion example above, SMARTedit's first guess was to move to the end of the string `ample`, which is five characters long, while the correct hypothesis (move to the string `<!--`) was less likely at four characters long.

SMARTedit uses version space transformations to provide a subset of regular-expression-style string searching, in addition to the strict substring matching performed by the FindPrefix and FindSuffix spaces. The "tokenization" transformation (shown in figure 4 beneath the FindFix space) converts the input text from a sequence of character tokens into a sequence of token types. Prefix and suffix searching is performed over these transformed sequences of token types. SMARTedit recognizes several classes of token types, including the sets of lowercase letters, uppercase letters, punctuation, digits, and whitespace. The tokenized string search version spaces allow SMARTedit to recognize hypotheses such as "the location after a sequence of five digits" (a postal ZIP code) or "the location before a sequence of two uppercase letters" (a state or province abbreviation).

As application designers, we believe that string searching is more likely than row and column positioning for most text-editing tasks, so we give the string search version space in the Location union higher prior probability than the row-column space. This weighting explains why a string search hypothesis showed up as the most likely prediction in the HTML comment deletion example described above, even though row-column hypotheses were present in the version space after the first training example. One direction for future work could be to change the prior probabilities based on the type of the file being edited. For instance, users editing structured tabular data may be more likely to use row and column positioning than string searching.

SMARTedit also recognizes several different types of insertion actions, as expressed in the Insert version space. The most straightforward insertion action is inserting a constant string. However, SMARTedit also recognizes regularly varying strings. For instance, the

StrNumStr space recognizes strings that consist of a constant string followed by a number followed by another constant string. The number can be a function of either the row number or the iteration number. For example, in the shopping-list task described in the introduction, one would want to insert the row number, followed by the string ".␣". SMARTedit's representation of actions as functions that map from one application state to another makes it easy to support functions like these, whose output depends on some variable in the state such as the row number of the insertion cursor.

At the leaves of the version space hierarchy are several instantiations of domain-independent version spaces, such as the ConstInt and LinearInt spaces (functions like $f(x) = C$ and $f(x) = x + C$).

Now that we have described how SMARTedit can learn a single action, we move on to learning programs as sequences of actions.

5.2.2. Learning complete programs. Repetitive text-editing tasks are generally solved by constructing a program with a loop that iterates over each of the repetitions. The body of the loop consists of the sequence of actions to perform one instance of the task, such as deleting a single HTML comment, and the loop is repeated until the task has been completed. Some tasks may require nested loops within the main loop. For instance, the main loop in a bibliography-editing task may iterate over the list of citations, and within each iteration of the main loop, a nested loop is required to iterate over each of the authors within the citation.

First we describe how SMARTedit learns programs with only a single loop and a fixed number of actions in the loop body. As the user executes the target program, she performs a sequence of actions (program statements) which are recorded by the system as a sequence of examples (input/output state pairs). The learning system must assign each of these state pairs to the program statement which was executed to create the example. If we knew the program counter, we could easily determine which step of the program was executed at each point. However, requiring a user to explicitly specify a program counter during the demonstration is unfeasible; instead we attempt to infer the program counter based on partial segmentation information in the trace. Hence, we have experimented with three user interfaces that enable the user to provide varying amounts of segmentation information:

- *Full segmentation*: the user presses a button before and after demonstrating each iteration of the program.
- *Start segmentation*: the user presses a button at the beginning of the demonstration, but does not need to press any button between iterations. The user may perform any number of full or partial iterations before completing the recording.
- *No segmentation*: the system is always recording the user's actions; the user does not have to tell the system when she begins a repetitive task.

Along with each of these interfaces we describe the version space structure required to learn from traces with these varying amounts of segmentation.

The *full-segmentation* interface generates a trace broken up into separate iterations of the same loop. The top tree in figure 4 shows the version space for learning programs with

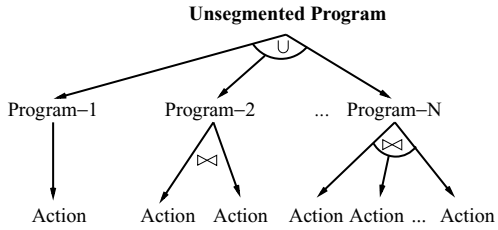

Figure 5. Unsegmented program version space used to learn programs without explicit segmentation information.

this structure: the Program version space consists of an N-way join of a fixed number of Action version spaces. The number of actions in the program is determined lazily once the user has demonstrated at least one iteration of the program. Each iteration of the program corresponds to a single training example for the Program version space, and consists of a sequence of application states, one more than the number of actions in the program. The first and second states are used to update the first Action space as a training example, the second and third states used to update the second Action, and so on.

The transformation function for the Program space thus takes a sequence of states $s_0, s_1, \ldots, s_n$ and constructs n examples of the form $\langle s_{i-1}, s_i \rangle$, such that the ith tuple (consisting of an input and an output state) is used to update the ith Action version space. Demonstrating a second iteration, or stepping through the program and correcting the system's predictions, both result in a similar sequence of states that is used to refine the version space.

Some application designers may feel that providing explicit segmentation (pressing a button before and after every iteration) is too burdensome for their users. For this reason we experimented with the *start-segmentation* user interface, which only requires a user to press a button at the start of the repetitive task. The result of such an interaction is a program trace of N actions. We assume that the user has demonstrated at least one complete iteration, and therefore know that the iteration length M (the number of actions in the loop body) is at most N. For instance, a user performing a program with a loop body of three steps could begin recording, demonstrate the first three actions in the first iteration, demonstrate two actions from the second iteration, and only then press the stop-recording button. The trace thus contains the application state before and after each of those five actions, but the iteration length of the target program is unknown and could be anywhere from one to five actions.

We have constructed the UnsegmentedProgram version space (figure 5) to learn programs from such unsegmented traces. The key idea is to maintain hypotheses for programs of all possible program lengths, from 1 to M. UnsegmentedProgram consists of a union of N Program version spaces, such that Program_1 consists of a single Action, Program_2 is a join of two Actions, and Program_N is a join of N Actions. Program_1's single Action version space is updated with all N observed actions as if each action were an example of a unit-length program. Program_2 is updated with $N/2$ pairs of actions, and so on. We also weight each member of the union such that shorter programs are given more probability than longer programs.

In practice, many of these Program version spaces collapse quickly because the action types do not match up except in programs of length M and multiples of M. In our five-step example, the two most likely choices would probably be either a length-3 program or a length-5 program. As soon as the user performs or confirms the sixth action, SMARTedit has a good chance of determining the correct program length.

Execution of the UnsegmentedProgram version space produces the union of the execution of each of the nested Program version spaces, and presents hypotheses to the user such as "Step 3 of 3 is Delete..." or "Step 1 of 5 is Move...".

A third possibility we have explored is the *no-segmentation* interface. In this interface, the system continually records the user's actions, and automatically detects when she performs a repetitive task. The user does not have to explicitly notify the system before she begins the repetitive task. A trace thus generated consists of the application state before and after every action the user has performed since she started using the application.

In order to learn a repetitive program from traces such as these, we assume that the user first performs an unknown number of unrelated actions, then performs at least one full iteration of the target program, followed by any number of full or partial iterations. If we knew where the program started, we could use an UnsegmentedProgram version space to learn the target program. Hence, to learn from such traces, we maintain hypotheses for every possible starting location in the trace. The StartlessProgram version space is a union of UnsegmentedProgram version spaces. The version space learns from an ongoing trace. As each new action is observed, a new UnsegmentedProgram is created such that its training examples begin at this action. The same action is also used to update all the other UnsegmentedProgram version spaces as if it were the next action in the sequence for each such version space.

As the expressiveness of the version space grows, its complexity also increases. In the simplest case, the Program version space maintains a number of Action version spaces that is proportional to the number of actions in one iteration of the target program. An UnsegmentedProgram space maintains a number of Action version spaces proportional to the square of the number of actions in the demonstration (which is assumed to be at least as long as a single iteration). A StartlessProgram space maintains a number of Action spaces proportional to the cube of the number of actions in the demonstration.

In practice, while unsegmented programs can be learned reasonably quickly with few training examples, the StartlessProgram version space is less useful, primarily due to the fact that it never stops growing; each new action may always be the start of a new program.

Both the UnsegmentedProgram and StartlessProgram version spaces are strict supersets of the basic Program version space; any program that could be learned from a fully segmented trace can also be learned from a trace with partial segmentation information. Since these alternatives define more complex hypothesis spaces, the price of reduced user effort is potentially requiring more examples to converge to a correct program.

Thus far, we have discussed the interfaces and version spaces for learning programs with a single loop and a fixed number of actions per loop iteration. Our system is also capable of learning programs with nested loops, in which a program may contain a sub-program that iterates several times before the main program resumes execution. Nested programs are straightforwardly expressed as a version space. In SMARTedit, we extend the Program

version space of figure 4 to support nested programs by replacing each Action member of the join with a union of Action and UnsegmentedProgram. This type of recursive definition is possible since the version space is lazily constructed.

When the program contains nested loops, knowledge of the program counter becomes even more important. We experimented with one user interface for demonstrating nested loops: a pair of buttons ("Start nested loop" and "End nested loop") that the user must push before executing the nested loop and after execution of the nested loop is completed. Thus the user must both segment each iteration of the main loop, as well as demarcate the beginning and end of the execution of the nested loop.

This interface gives the system just enough information about the program counter to determine whether each step belongs to the main loop or a nested loop. We hypothesized that asking the user to explicitly segment the nested loop into iterations was too much of a burden, and chose to learn the nested loop's program from the trace using an UnsegmentedProgram version space. Similarly, we limited the nesting depth to one so as not to confuse the user with multiple levels of nested loops. The user is still able to demonstrate programs with multiple sequential nested loops. In future work we plan to investigate alternative user interfaces for extracting the program counter information in the presence of loops, as well as experimenting with version spaces to learn from nested-loop traces without any segmentation information.

Another item for future work is to learn loop termination conditions. Upon execution of a nested loop, our current implementation requires a user to explicitly tell SMARTedit when to exit the nested loop and resume execution of the main program. Note that this decision must be made as the user is stepping through the program; unless otherwise directed, SMARTedit will continue executing the actions in the nested loop forever. We anticipate that a version space containing an appropriate set of condititionals could be joined to the nested loop's UnsegmentedProgram version space in order to learn loop termination conditions.

As this section has illustrated, our version space algebraic framework enables the application designer to tailor the version space to the abilities and preferences of the target audience. There is an inherent tradeoff between expressiveness and sample complexity; choosing an appropriate point on this spectrum is the responsibility of the application designer. In the next section, we report on experimental results conducted primarily on a version of SMARTedit equipped with the full-segmentation interface, as well as a few experiments with the nested loop interface.

6. Experimental results

We evaluate the SMARTedit system using two methods. First we summarize SMARTedit's performance on a collection of repetitive text-editing scenarios. Then we report on the results of a small user study.

6.1. Empirical evaluation

We have applied the SMARTedit system to a representative collection of repetitive text-editing scenarios; see Appendix B for a brief description of each scenario. Each scenario

Table 2. List of scenarios used to test the SMARTedit system, total number of iterations in each, and number of iterations required by the system to induce a procedure that behaves correctly on the remaining iterations.

Scenario	No. of training iterations	Total no. of iterations
bibitem-newblock*	1	13
c++comments	1	5
column-reordering	1	14
country-codes	1	4
modify-to-rgb-calls	1	20
number-fruits	1	14
prettify-paper-info	1	10
subtype-interaction	1	3
xml-comment-attribute	1	24
addressbook	2	6
citation-creation	2	13
grades	2	7
html-comments	2	13
latex-macro-swap	2	8
number-citations	2	13
number-iterations	2	7
smartedit-results	2	27
zipselect	2	6
game-score	3	7
html-latex	3	7
indent-voyagers	3	32
mark-format	3	6
bold-xyz	4	50
citation-to-bibtex	5	10
bindings	6	11
boldface-word	6	11
ul-to-dl	6	7
OKRA	10	14
outline	10	14
pickle-array	19	117

contains the trace of a target program executed for a number of iterations on a text file. A trace is a sequence of (T, L, P, S) states (text buffer, cursor location, clipboard contents, and selection region). Table 2 lists the scenarios we used to evaluate SMARTedit, along with the total number of iterations in each, and the number of iterations needed to train the system before the learned program performed correctly on all remaining iterations (i.e., applying the correct transformations to the remaining text). A * indicates that the task requires nested loops, and the result shown assumes that SMARTedit is given the termination

condition for the loop. The tasks shown in boldface are the bibliography tasks described in Section 2.1.

We test SMARTedit using an incremental learning methodology: train on the first iteration, then test the learned program's accuracy on the remaining iterations. If SMARTedit performs correctly on the remaining iterations, then we say that it requires only a single iteration to learn the task correctly. If not, we train on two iterations, test its accuracy on the remainder, and so on. The demonstrations were performed by an experienced SMARTedit user (one of the authors). In some cases, the system requires only a single iteration to correctly learn a program that performs correctly on the remainder of the text. Visual inspection of the learned programs indicates that SMARTedit's generalizations typically agree with the human's target program. Although we do not present timing results, the learning algorithm runs virtually in real time on a 600 MHz Pentium III.

Many of the scenarios were learnable with only two iterations. Some of the others required more training because of irregularities in the text data. For example, the pickle-array scenario involves converting an array of numbers from one format (a Python pickled format) to a different format. After two training iterations, the system learns a program that performed correctly on the third through eighteenth iterations. However, the nineteenth iteration is irregular: it crosses a row boundary in the array representation. Two competing hypotheses (which up until this point had both been consistent with the trace) now make differing predictions. In this case, the incorrect hypothesis has higher probability, requiring the user to correct SMARTedit's guess on this example. According to our metric, SMARTedit thus requires nineteen iterations to learn the task correctly, whereas if it had been given the nineteenth example earlier, it would have required only three iterations. We are currently investigating the use of active learning to identify anomalous examples earlier in the training process (Wolfman et al., 2001).

The bibitem-newblock scenario (marked with a * in Table 2) requires the use of nested loops to solve correctly; this task is described in Section 2.1. The table shows the number of iterations required to learn to perform this task correctly assuming that the termination condition is given (i.e., that the user indicates to SMARTedit when to terminate the nested loop and resume execution of the main program); SMARTedit is not yet able to predict when to terminate the nested loop on its own. In future work we plan to extend SMARTedit to learn the loop termination condition as well.

6.2. User evaluation

We conducted a pilot user study to gather user experiences with the SMARTedit system. We asked six undergraduate computer science majors to perform a set of seven tasks with and without SMARTedit's programming by demonstration capability. The study participants are not meant to be completely representative of our intended target audience (which includes non-programmer users); an item of future work will be to conduct a more comprehensive study with a more diverse set of participants. Our study determined that SMARTedit was useful for this set of users, and solicited feedback on its design prior to further development.

We began by giving each participant a five-minute introduction to SMARTedit's user interface, and guided them through one or two simple tasks using SMARTedit to familiarize them with the interface. We then asked them to perform a sequence of seven tasks, first using SMARTedit's programming by demonstration feature, and again using the same editor but without invoking the programming by demonstration capability (i.e., completing the repetitive task manually). If there were a practice effect from a user performing the same task twice in a row, the result would be biased against our system.

Each participant spent less than an hour total interacting with SMARTedit; none had had any prior experience with the program. Again, we chose this methodology to bias the results against our system; users with more experience using the system would be more efficient and show even greater gains than those shown in our results.

After the users had finished the tasks, we asked them to complete a form with questions about SMARTedit's helpfulness, usability, and whether or not they would consider using the program again.

The tasks we chose varied in difficulty and ranged from 4 to 27 iterations per task. They were chosen to reflect reasonably-sized tasks that we imagine users might naturally face. For each task, and for both SMARTedit and manual execution, we measured the time, the number of key presses, and the number of mouse clicks a user required to complete the task. The tasks were, in order: html-comments, country-codes, column-reordering, xml-comment-attribute, smartedit-results, number-citations, and bibitem-newblock.

The first six tasks were straightforward fixed-length programs, roughly increasing in difficulty; the seventh task (bibitem-newblock) required users to construct a program with a nested loop. We allowed users to give up on completing a task with SMARTedit if they became frustrated after five minutes of effort. All users completed the first six tasks with the exception of two users on the fifth task. Only one user successfully completed the seventh task using nested loops; another cleverly solved it with SMARTedit using two passes over the data.

All nested loop failures occurred because users failed to maintain the close attention required to guide SMARTedit back to the main program after execution of the nested loop. Users assumed that SMARTedit would learn how to terminate the loop on its own. More work needs to be done to improve the interface for demonstrating and executing nested loops; learning loop termination conditions is clearly a priority for future work.

We measured SMARTedit's performance with two metrics: the time savings gained by using SMARTedit over doing the task manually, and the percent of user actions (both keypresses and mouse clicks) saved using SMARTedit compared to manual.

Figure 6(a) shows the difference in seconds between the time taken to complete the task manually, and the time to complete it with SMARTedit. Bars above the zero line indicate that a user completed the task more quickly with SMARTedit. X's indicate missing data; in two cases, we failed to collect timing data during the experiment, and two users were unable to complete the fifth task. Users learned to use SMARTedit more effectively over time; by the fourth task, most users were able to benefit significantly from SMARTedit. Although not all users spent less time with SMARTedit than on the manual task, the results are encouraging given that none of the participants had any prior experience with the system.

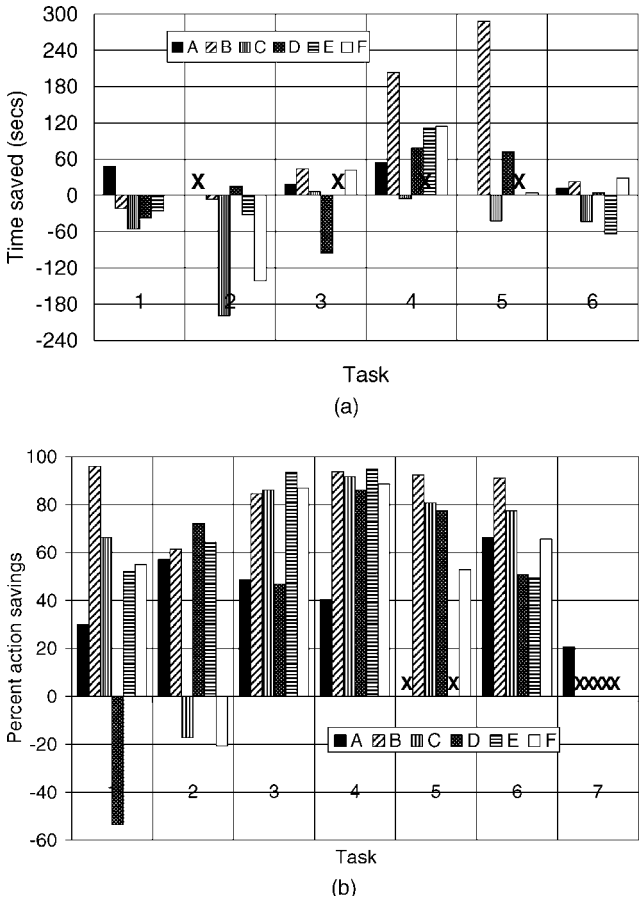

Figure 6. (a) Time and (b) user action savings using SMARTedit compared to performing the task manually, for users A–F. Bars above the zero point show increased savings. X's indicate missing data.

Figure 6(b) shows the percentage of user actions saved when using SMARTedit as a fraction of the number of manual actions (the number of manual actions minus the number of SMARTedit actions, divided by the number of manual actions). We define a user action to be either a keypress or a mouse click. Bars above the zero point show a savings with SMARTedit compared to manual execution. The higher the bar, the fewer actions a user performed with SMARTedit. X's indicate users that could not complete the task using SMARTedit. The figure suggests that most users were able to benefit from using the system as they gained practice with it. Recall that none of the users had used the system prior to the commencement of this study.

We also obtained user feedback in the form of responses to a questionnaire, summarized in Table 3. The feedback was quite positive. We asked users to rate (on a scale of 1 to 5, 5 being the best) SMARTedit's helpfulness in accomplishing the tasks, SMARTedit's usability, and whether they would use the system again. As the table shows, participants

Table 3. User feedback on the SMARTedit system. Each user rated SMARTedit on a scale of 1 to 5, 1 being the worst and 5 the best. The first question asked the user to rate SMARTedit's helpfulness in accomplishing the tasks, 5 being "very helpful". The second question rated SMARTedit's usability, 5 being "easy to use". The final question rated whether the user would use SMARTedit again, 5 being "yes, absolutely".

Question	User 1	User 2	User 3	User 4	User 5
Helpful?	4	4	4	4	5
Usable?	4	4	3	2	3
Use again?	4	5	4	5	5

found SMARTedit helpful and would use it again. One user wished he had had SMARTedit at work, saying "this would have been a nice thing to have." Others called it "cool" and "clever." Another noted the small number of user actions necessary to finish the task with SMARTedit and said, "Four keys and nine mouse clicks. I like that!"

The time saved by using SMARTedit on a task depends on the number of iterations in the task; one expects the savings to increase with the number of iterations. We averaged the time savings across all users for each task; figure 7(a) shows the average time savings on a task versus the number of iterations in the task. We fit a line to these points, obtaining r^2 of 0.97. The figure confirms our intuition, and indicates that for novice users, the break-even point (the number of iterations for which the total time spent operating SMARTedit equals the time spent performing the task manually) is about 15 iterations. We observed that participants became more proficient in using the system over the course of the study, so these results may be skewed by a practice effect.

Since all of our user study participants were untrained in the use of SMARTedit, much of their time was spent learning how to operate the system. To provide an alternative view of SMARTedit's usefulness, two experienced SMARTedit users (the first two authors) performed the smartedit-results task both with SMARTedit and without, recording the time spent per iteration. Figure 7(b) shows the cumulative time spent performing each iteration of the task for each user in both situations. Both users operated SMARTedit naturally: they demonstrated a single iteration, stepped through the second iteration while correcting SMARTedit's wrong guesses, and ran the program uninterruptedly starting from the third iteration. The uninterrupted run steps slowly through the first few actions of the learned program, to give the user time to observe that it's behaving correctly, then decreases the artificial delay between steps until it eventually runs at full speed. As shown in the figure, it took longer to teach SMARTedit than to perform the task manually on the first two iterations, but the total time was much faster with SMARTedit than without. The break-even point is 5 iterations. While this result may not generalize to tasks of arbitrary complexity, it suggests that the break-even point decreases as a user gains experience with the system.

The study indicated that SMARTedit clearly needs improved usability before it gains acceptance. Users found its primary failing to be the lack of a "multi-level undo" facility; once a user wrongly confirms one of SMARTedit's guesses by stepping through it to the next action, SMARTedit's version space could collapse, and users have no choice but to retrain the program from scratch. Improving SMARTedit's usability is a clear direction for future work, as is performing more extensive user studies.

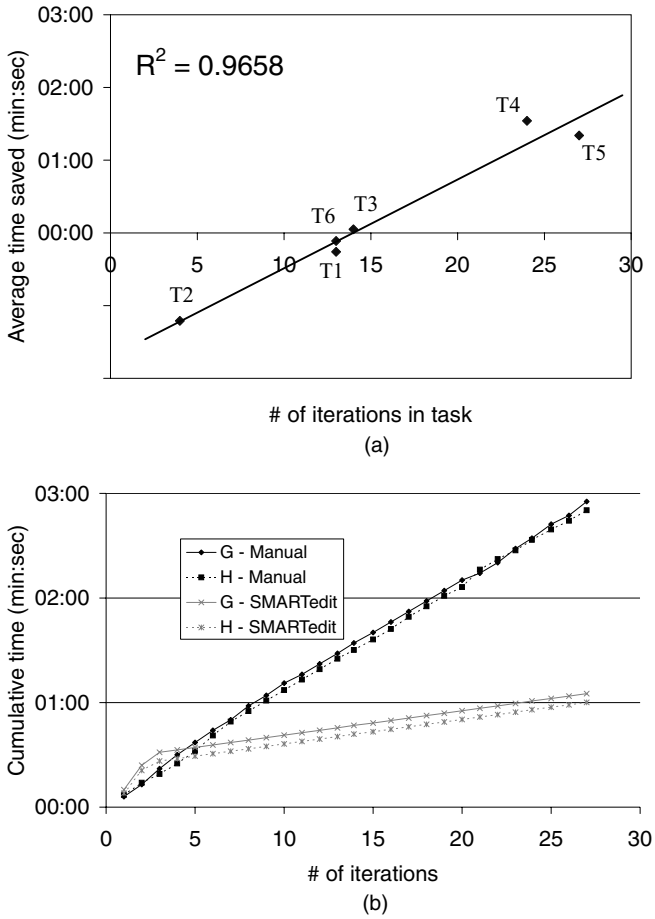

Figure 7. (a) Average time savings across all users for each task versus number of iterations in the task. (b) Cumulative time required for experienced users G and H to complete successive iterations of the smartedit-results task with SMARTedit and without, as a function of the number of iterations in the task.

7. Related work

Our work draws on a number of research areas, incorporating ideas from both human-computer interaction and artificial intelligence. In this section, we summarize related work in programming by demonstration, adaptive user interfaces, plan recognition, information extraction, and machine learning.

7.1. Programming by demonstration for text-editing

Other researchers have investigated programming by demonstration in the text-editing domain. Unlike most previous text-editing PBD systems, SMARTedit uses a formal machine

learning technique to describe the generalization that is performed by the system. Mo (1989) describes the TELS system that records high-level actions similar to the actions used in SMARTedit, and implements a set of expert rules for generalizing the arguments of each of the actions. TELS also uses heuristic rules to match actions against each other in order to detect loops in the user's demonstrated program. However, TELS's dependence on heuristic rules to describe the possible generalizations makes it difficult to understand the hypothesis space clearly, as well as to imagine applying the same techniques to a different domain such as spreadsheet applications.

Nix (1985) describes the Editing by Example (EBE) system. EBE generalizes not from recorded actions, but from the input/output behavior of the complete demonstration. EBE attempts to find a program that can reproduce the observed difference between the initial and final state of the text editor. In this respect, SMARTedit is a refinement of EBE that uses not only the initial and final state, but intermediate states as well. SMARTedit's approach has the drawback that it is sensitive to the order in which the user chooses to perform actions, but on the other hand it makes use of more information than EBE is given, and so SMARTedit is able to learn programs for more complex text transformations than EBE.

Masui and Nakayama (1994) describe the Dynamic Macro system for recording macros in the Emacs text editor. Dynamic Macro performs automatic segmentation of the user's actions—breaking up the stream of actions into repetitive subsequences, without requiring the user to explicitly invoke the macro recorder. Dynamic Macro performs no generalization, and it relies on several heuristics for detecting repetitive patterns of actions.

Maulsby and Witten's Cima system (1997) uses a classification rule learner to describe the arguments to particular actions, such as a rule describing how to select phone numbers in the local area code. (SMARTedit is able to learn a program to select all but one of the phone numbers in the Cima task given a single demonstration. The anomalous phone number lacks a preceding area code, and is also difficult for Cima to classify correctly.) Unlike other PBD systems, Cima allows the user to give "hints" to the agent that focus its attention on certain features, such as the particular area code preceding phone numbers of interest. However, the knowledge gained from these hints is combined with Cima's domain knowledge using a set of hard-coded preference heuristics, unlike SMARTedit's clean framework for incorporating knowledge. As a result, it is unclear exactly which hypotheses Cima is considering, or why it prefers one over another. In SMARTedit, these types of hints could be used to bias the probabilities on its different hypotheses.

7.2. Programming by demonstration in other domains

Many PBD systems have been discussed in the literature, in many domains other than text-editing; Cypher (1993) and Lieberman (2001) present overviews of the field. Witten (1995) surveys previous attempts to tie together programming by demonstration and machine learning.

In prior work (Lau & Weld, 1999), we formalized PBD in the email domain as an inductive learning problem using both a version space approach as well as an inductive logic programming approach. Compared to our previous work, the version space algebra

framework described in this article combines a more expressive and flexible bias with a more natural and straightforward domain representation.

Early work that formalized PBD as learning functions to transform application state or data includes Andreae's NODDY (1984) and Halbert's SmallStar (1993). Lieberman's Tatlin system (1998) infers user actions from observed application state changes. Tatlin learns how to cut and paste data from a calendar application to rows in a spreadsheet. Lieberman also discusses the problem of constructing appropriate data descriptions for the objects manipulated by the PBD system, such as "the window the user clicked on" or "the window named `Mail`"; perhaps the most important aspect of SMARTedit's learning component is its ability to infer the correct data description from examples.

The APE system (Ruvini & Dony, 2000) automates repetitive sequences of actions in a visual programming environment. It uses two learning algorithms: a classification learner, to determine whether (given the user's recent actions) the user is about to perform a repetitive task; and a concept learner based on candidate elimination (Mitchell, 1982) to generalize repetitive tasks from positive examples. Compared with our work, APE explores a different point in the programming by demonstration space. Our work explores learning richer program structures, such as nested loops, but requires explicitly segmented traces to do so. Our hypothesis space for actions is richer and contains more forms of generalization. On the other hand, APE performs non-supervised data mining on a trace of the user's actions, which is a direction we have only begin to pursue in our work (e.g., what we have called startless programs).

Bauer and Dengler (1999) and Bauer, Dengler, and Paul (2000) present TrIAs, a PBD system for learning wrappers for the purpose of training an information integration system. TrIAs is designed around a collaborative user interface in which a user teaches the system how to extract information from a web site by labelling a single example page. Unlike SMARTedit, TrIAs performs no generalization from multiple examples, instead using only heuristics and user hints to choose a hypothesis that is consistent with each part of the example. In comparison, SMARTedit has a well-defined hypothesis space, and the verion space framework clearly shows how each new bit of evidence affects the generalization process.

Paynter (2000) presents Familiar, a system for cross-application PBD on the AppleScript platform. Familiar employs a combination of machine learning algorithms to identify repetitive sequences of actions in two phases: first it identifies a repetitive sequence of action types (ignoring the parameters), and then it extrapolates the parameters for each of the actions. Within each of these stages, Familiar employs a set of heuristic rules to select the best hypothesis from among the guesses generated by the collection of recognizers. Unlike our work, which maintains the set of all consistent hypotheses along with associated probabilities, Familiar generates only a single hypothesis for the data. When an example is received that conflicts with the current hypothesis, Familiar must discard its hypothesis and re-train from the beginning.

7.3. Adaptive user interfaces

PBD is only one component of an adaptive user interface. Rich and Sidner (1997) describe the COLLAGEN collaborative agent system. COLLAGEN employs a hierarchical plan

library structured according to a user's goals and intentions, and uses it to model user tasks. Lesh, Rich, and Sidner (1999) apply plan recognition in COLLAGEN to infer a user's task from her actions. Using the task model, COLLAGEN enters into a collaborative discourse with the user to help her achieve shared goals. In our system, the collaborative task is to teach SMARTedit how to perform a particular repetitive edit. Although beyond the scope of this paper, we have also constructed a mixed-initiative interface for SMARTedit that could reduce user effort during a training session (Wolfman et al., 2001).

Researchers such as Maes and Kozierok (1993), Schlimmer and Hermens (1993), and many others have considered complementary approaches to adaptive interfaces, using machine learning techniques to predict the user's next actions. Space precludes a detailed discussion of the considerable work in this vein.

7.4. Plan recognition

We view programming by demonstration as a special case of the plan recognition problem. Plan recognition (Kautz & Allen, 1986; Charniak & Goldman, 1993; Vilain, 1990; Pollack, 1990) aims to understand an agent's plan or goal based on a partial observation of the agent's behavior. In an intelligent user interface, this knowledge may then be used to predict the agent's further actions or suggest more optimal methods for accomplishing the goal.

The issue of whether programs are plans has been debated at length (for example, see Agre & Chapman, 1990); however, for the purposes of this section we assume they are equivalent: a sequence of actions. Thus PBD can be viewed as the problem of observing a prefix of these actions, and generalizing to the complete plan.

Plan recognition systems typically require the use of a plan library: the set of plans from which the agent's target plan is assumed to be drawn. This plan library defines the plan recognizer's bias, or the space of hypotheses considered during the search. Kautz and Allen (1986) employ a plan library in the form of a hierarchy of plans connected via abstraction and decomposition relationships.

A second approach to building a plan library has been to represent plans as sentences in a grammar and phrase the plan recognition problem as parsing in a grammar (Vilain, 1990; Pynadath & Wellman, 2000). The parsing-based approaches suffer from the drawback that plans must be totally ordered, which may cause an exponential increase in the size of the plan library (to represent all total orderings over a partially ordered plan).

In order to scale to large domains, a third approach has investigated automatic construction of the plan library, such as proposed by Lesh (1997, 1998) and Lesh and Etzioni (1995, 1996). They point out the similarities between goal recognition and concept learning: given a sequence of actions, find the goal or goals consistent with the observations. Our approach falls into this category. The basic idea is to provide a set of primitives and a set of rules for composing together the primitives to construct hypothesis space implicitly. Related to automatic construction is the idea of efficient search through this space.

The choice of the plan library representation influences the types of plans that may be recognized by the system. In the Kautz and Allen representation, arbitrary plans may be represented; however, manual construction of the plan library limits the scalability of this approach. On the other hand, automatic construction of the library can scale up to

larger domains, but at the cost of reduced expressiveness. Our system is only capable of recognizing repetitive plans. However, many previous plan recognition systems assume that the agent is acting optimally, and only include optimal plans in the library. In contrast, we make no assumptions about the agent's behavior; our system is capable of recognizing the user's intent, whether or not the user performs optimally.

Another difference between the version space algebra approach and previous plan recognition systems is that we maintain the set of plans consistent with the observations, rather than finding a single consistent plan; and we allow probabilistic execution before the version space converges to a single plan. The Kautz and Allen approach produces the set of consistent plans in which the number of top-level actions is minimized, but provides no way to choose among them. In contrast, Charniak and Goldman (1991) present a probabilistic method for deciding between plan hypotheses, but do not address the problem of determining which plans are consistent with the data. Lesh and Etzioni maintain the set of all consistent goals, and support execution before convergence in special cases (when all goals in the version space are necessary or sufficient for the target goal); however, they provide no general mechanism for probabilistic execution or introduction of prior probabilities on hypotheses.

Our approach can be divided into three phases:

- *Translation*: converting from a sequence of low-level keypress and mouse actions to a sequence of state changes representing high-level actions;
- *Learning*: inducing high level actions from multiple examples of the state changes associated with each action; and
- *Recognition*: finding a repetitive plan consistent with the induced actions.

While one could do plan recognition over the low-level actions (such as sequences of cursor key presses), we believe that our method of abstracting away from the exact sequence of low-level actions leads to a representation closer to the user's model of the application, and thus more robust and understandable plans.

Most previous plan recognition systems perform only the third phase: recognizing a plan given a sequence of actions. In our work, the third phase is trivial: once the actions are known, the plan is merely a repetition of the sequence of actions. Instead, we have focused our efforts on the first two phases.

Lesh and Etzioni's approach is the most similar to our work; we compare the two systems in more detail in the remainder of this section. The BOCE (Lesh & Etzioni, 1996) system uses version spaces for goal recognition. Actions are represented in a STRIPS-like formalism (Etzioni et al., 1992), and a goal is a conjunction of literals that must be established by a plan (legal sequence of actions[5]). A goal is consistent with a sequence of observed actions if there exists a pseudo-optimal plan[6] containing those actions that, when executed in some state, asserts the goal. While BOCE represents goals explicitly in a predicate description language, our approach has no explicit goal representation.

Both BOCE and our approach use a version space representation of an implicit plan library, but the elements of each version space are very different. BOCE's version space contains *goals* ordered by generality, where each goal is a conjunction of literals.[7] In contrast, our approach constructs a version space of plans where each plan is represented as a function from state to state. Our version space algebra has a number of benefits over

a regular version space: other partial orderings, not necessarily generality; representation of version spaces with "holes"; combination of boundary-represented version spaces with enumerated version spaces in the hierachy; the join operator and decomposition of the target function into simpler components.

7.5. Wrapper induction and information extraction

Another body of related work is on techniques for wrapper induction and information extraction (Kushmerick, Weld, & Doorenbos, 1997; Ashish & Knoblock, 1997; Cowie & Lehnert, 1996; Hobbs, 1992). Wrapper procedures are programs that scan over semi-structured text, such as HTML tables, in order to extract tuples of information, such as names and phone numbers. Wrappers make use of certain regular features in the text, such as HTML opening and closing tags, to uniquely identify the location of each item of interest. Wrapper induction is the problem of learning such wrappers from labelled training examples. These wrapper procedures are very similar to the types of procedures learned by SMARTedit. Specifically, the language we have defined for expressing text-editing programs could be used (with few changes) to express programs for wrapper induction. In fact, several of the scenarios used to evaluate SMARTedit are drawn directly from the RISE repository of information extraction tasks (Muslea, 2000).

Kushmerick (2000) defines a variety of wrapper classes of increasing functionality. Kushmerick's LR wrapper class denotes a set of programs that is a subset of the class of single-loop programs with fixed-length loop bodies, and his HLRT wrapper class is equivalent to a program with a block of statements to find the head, a single loop with a fixed-length loop body (the LR component), followed by a block of statements to identify the tail. Our work has identified and shown how to learn these classes of programs, as well as more expressive programs containing nested loops within the single outer loop.

Muslea, Minton, & Knoblock (2000) describe the use of selective sampling (a form of active learning) using multiple views to pick the training example that is likely to cause the version space to converge the most quickly. A promising direction of future work is to apply active learning to SMARTedit; we have already taken some steps towards this goal (Wolfman et al., 2001).

7.6. Machine learning

Many researchers have extended version spaces in various ways. For example, Hirsh (1991) studies the algebra of boundary-set representable version spaces, and Hirsh, Mishra, and Pitt (1997) propose maintaining version spaces without boundary sets. We have extended Hirsh's work beyond concept learning to complex-function learning by allowing any partial order, defining the join operator, and supporting unions that are not represented by boundary sets.

A different extension of version spaces has been proposed by VanLehn and Ball (1987), for inducing context-free grammars from examples. Since generality is undecidable for context-free grammars, their work approximates the generality relation using an alternative partial order.

Subramanian and Feigenbaum (1986) describe version space factorization, which is very similar to our version space join. Rather than using the factored version space to express more complex hypotheses as our framework supports, however, they use factorization within the concept-learning framework to choose a sequence of instances that speeds convergence of the version space.

Explanation-based learning (EBL) is another class of machine learning algorithms that generalizes from few examples with the help of domain knowledge. For example, Shavlik & DeJong (1987) describe an explanation-based learning system that generalizes the concept of number. EBL is similar to version space algebra in that it has been used in learning for problem-solving, and is capable of learning from a single example. However, EBL is more limited in that it assumes a sound and complete domain theory, which is seldom available in practice. While EBL learns by reformulating a single domain theory, the version space algebra also supports efficient exhaustive search through a range of possible theories. In this respect it is also related to inductive logic programming (Muggleton, 1992). While most ILP systems maintain and return only a single theory, the version space algebra maintains all theories efficiently. In prior work (Lau & Weld, 1999), our TGEN$_{\text{FOIL}}$ system employed the FOIL ILP algorithm to learn programs in the email domain. Although we have not directly compared the two approaches on the same domain, we have found that the programs produced with version space algebra are generally easier to understand, and map more directly to a natural domain representation, than did the programs learned with TGEN$_{\text{FOIL}}$.

8. Future work

We are currently investigating several extensions to the research described in this article. Our future work falls roughly into three categories: application-specific extensions, version space algebra extensions, and consideration of new domains.

While the bias defined by SMARTedit's version space provides a useful set of text-editing actions, it is desirable to increase the expressiveness of SMARTedit's instruction language. In addition to extending the set of version spaces used in SMARTedit, we plan to investigate methods for automatically inferring loop termination conditions, biasing probabilities according to user preference or file type, and adding conditional statements to the instruction language supported by SMARTedit.

With an increase in expressiveness comes an increase in the size of the search space and the number of required training examples. As the search space increases, efficiency becomes a concern. The version space may reach a size where a complete search is not feasible. One direction of future work is to investigate means of combining boundary-represented version spaces with heuristically-searched spaces in order to efficiently learn procedures. In addition, we may need to investigate the use of sampling in order to efficiently execute such large version spaces. We may find it necessary to extend our algebra with additional operators, such as intersection.

We foresee several methods for reducing the number of required training examples in the presence of a large search space. First, we plan to investigate techniques for version space revision, or dynamically weakening the bias to include less-likely hypotheses if the stronger

initial biases fail. Second, we are actively pursuing methods including active learning and collaborative interfaces with the intent of reducing the number of required training examples. See recent work (Wolfman et al., 2001) for preliminary results in this area.

One assumption we have made in this work is the availability of pre-parsed examples that identify which part of the example is used to update each version space. For instance, an example for the RowCol version space can be parsed into its row and column components, and each used to update the appropriate version space. An area for future work is to investigate how we might automatically learn a decomposition of the version space into subspaces, when this decomposition is not given *a priori*. For example, in a speech recognition problem we might have a version space for each word, but we generally do not know where each word begins and ends. A related problem is that of credit assignment: deciding which version spaces should be modified in response to a given example, when the example can be parsed in more than one way.

Another assumption we have made is that the examples are noise-free. As mentioned in Section 3.5, one solution to deal with noise is to reduce the probability of inconsistent hypotheses, rather than eliminating them entirely from the version space. We will need to investigate mechanisms for efficiently maintaining such a probabilistic version space. Norton and Hirsh (1992) present an alternative method, where rather than maintaining a single version space containing potentially-inconsistent hypotheses, they maintain a set of version spaces, each of which is consistent with some interpretation of the data.

While the version space algebra has proven useful in the text-editing domain, an open area of future work is applying the version space algebra to other machine learning domains, either in support of programming by demonstration for a given domain, or in another application unrelated to the user interface. Another direction for future work is to empirically investigate alternative domain encodings, either for the text-editing domain or a different one.

Many of the version spaces used in SMARTedit are not application-dependent; they form the basis of a library of version spaces that can be ported directly to a new domain. Future work will expand this library to contain a larger variety of application-independent version spaces.

9. Conclusion

In this article, we describe a framework for complex-function learning called version space algebra, and apply it to programming by demonstration in the text-editing domain. We make the following contributions:

- We extend version spaces to complex-function learning. Efficient maintenance of a version space requires only a partial order, not necessarily the generality partial order.
- We describe a version space algebra for combining smaller, more restrictive version spaces into complex spaces. We define the union, join, and transformation operators for combining version spaces, and show that these operators preserve PAC learnability. We present a probabilistic framework for reasoning about the probability of each hypothesis in the composite version space.

– We apply the version space algebra to the problem of programming by demonstration, and describe the implemented SMARTedit system for PBD in the text-editing domain. We have also constructed a library of reusable component version spaces that may be applied to other domains.
– We have experimented with a number of user interfaces and version spaces for learning single-loop programs from traces with varying amounts of segmentation information, as well as learning programs with multiple nested loops.
– We validate our approach experimentally using a range of text-editing scenarios, including bibliographic editing tasks. We show that a program that generalizes correctly for each of these scenarios can be learned quickly in as few as one or two training examples. We report on an informal user study that confirms SMARTedit's usability and usefulness, and show how SMARTedit helps users perform tasks more quickly and with fewer actions.

Appendix A: SMARTedit version spaces

In this section, we summarize the version spaces used in the full-segmentation SMARTedit system, organized roughly in order of appearance starting from the target space Program.

Program The target version space. A Program consists of a join of a fixed number of Action version spaces. Functions in this space map from the initial application state to the state after one full iteration of the program has been performed. (Note that the learning process requires supplemental information in the form of a trace of intermediate program states.) The number of component version spaces is lazily determined from the first training example.

Action A union of all the types of text-editing actions known to SMARTedit: Move, Insert, Delete, Select, Cut, Copy, Paste, DeleteSelection. Functions in this space map from one application state to another.

Move A transformation of a Location version space. Functions in this space map from one application state to another. Functions in the Location space map from an input state to a cursor position. The τ_o^{-1} transformation converts from the cursor position to a state in which the cursor is positioned at that location.

Delete A transformation of a join of two Location spaces specifying the left and right endpoints of a region. Functions in this space map from one application state to another. Functions in the join of two Locations map from an input state to a pair of locations. The τ_o^{-1} transformation converts a pair of locations to a state in which the text between those locations has been deleted.

Insert A transformation of a union of three version spaces: ConstStr, StrNumStr, and IndentStr. Functions in this space map from one application state to another. The τ_o transformation for the ConstStr component converts the output state into the string that is inserted between the input and output states. The τ_o^{-1} transformation converts a string into a state in which the string has been inserted into the text buffer at the current cursor position.

ConstStr Atomic version space of functions that accept no input and return a constant string.

StrNumStr A join of three version spaces: ConstStr, Number, and ConstStr. Functions in this space map from an input state to a string. The τ_o transformation extracts the string inserted between the input and output state, and splits it into three substrings: a non-numeric string prefix, a number, and the remainder of the string. Each string is used to update the associated version space. The τ_o^{-1} transformation concatenates the strings returned by each of the three component version spaces.

Number A union of two LinearInt version spaces. Functions in this space map from the input state to a string representing a number. Functions in the component spaces map from an integer to an integer. The τ_i and τ_o transformations extract the row number and the iteration number from the input and output states, respectively; row and iteration number are used to update the two component version spaces. The τ_o^{-1} transformation converts an integer into its string representation.

LinearInt Atomic version space of functions from integer to integer of the form $f(x) = x + C$.

IndentStr A join of two version spaces: Identity and ConstStr. Functions in this space map from the input state to a string. The τ_o transformation extracts the string inserted between the input and output state, and splits it into two substrings: a sequence of whitespace tokens, and the remainder of the string. The τ_i transformation for the Identity space extracts the amount of whitespace indentation on the current line of the input state. The two whitespace strings are used to update the Identity space, and the remainder of the string updates the ConstStr space. The τ_o^{-1} transformation concatenates two strings and returns a single string.

Identity Atomic version space containing at most a single function: $f(x) = x$.

Select A transformation of a Location version space. Functions in this space map from one application state to another. Functions in the Location space map from an input state to a cursor position. The τ_o^{-1} transformation converts from the cursor position to a state in which the selection spans the region between the cursor position in the input state and the new cursor position.

Cut Atomic version space representing a cut action.

Copy Atomic version space representing a copy action.

Paste Atomic version space representing a paste action.

DeleteSelection Atomic version space representing the action of deleting the selected text without copying it to the clipboard.

Location A union of several verion spaces: FindFix, WordOffset, CharOffset, and RowCol. Functions in this space map from an input state to a location.

FindFix A union of several version spaces: two FindPrefix spaces and two FindSuffix spaces. A token typing τ_i transformation is applied to one each of the FindPrefix and FindSuffix spaces. This transformation converts the literal sequence of tokens in the text of the input state into a sequence of token *types*. Token types include lowercase letter, uppercase letter, whitespace, digit, and punctuation.

FindPrefix Atomic version space whose functions map from an input state to a location. The new location is the position at the end of the next occurrence (relative to the position in the input state) of a sequence of tokens. Each function in this space reflects a different sequence of tokens.

FindSuffix Atomic version space whose functions map from an input state to a location. The new location is the position at the beginning of the next occurrence (relative to the position in the input state) of a sequence of tokens. Each function in this space reflects a different sequence of tokens.

WordOffset A transformation of a LinearInt space. Functions in this space map from an input state to a location. The τ_i transformation converts from the input cursor position to an integral word offset relative to the beginning of the text. The τ_o transformation converts the output cursor position similarly. The τ_o^{-1} transformation converts from an integer representing a word offset to a position in the text.

CharOffset A transformation of a LinearInt space. Functions in this space map from an input state to a location. The τ_i transformation converts from the input cursor position to an integral character offset relative to the beginning of the text. The τ_o transformation converts the output cursor position similarly. The τ_o^{-1} transformation converts from an integer representing a character offset to a position in the text.

RowCol A join of two version spaces: Row and Column. Functions in this space map from the input state to a cursor position. The τ_i and τ_o transformations for Row extract the row value of the cursor position in the input and output state, respectively. For the Column component, the transformations extract the column value. The τ_o^{-1} transformation converts a row and column value into a cursor position.

Row A union of two version spaces: AbsRow and RelRow. Functions in this space map from a row value to a row value.

AbsRow A transformation of a ConstInt version space. Functions in this space map from nothing to a row value. The τ_o transformation converts from a row value to an integer. The τ_o^{-1} transformation converts from an integer to a row value.

RelRow A transformation of a LinearInt version space. Functions in this space map from a row value to a row value. The τ_i and τ_o transformations convert from the row value of the input/output states to integers, and the τ_o^{-1} transformation converts from an integer to a row value.

AbsCol A transformation of a ConstInt version space. Functions in this space map from nothing to a column value. The τ_o transformation converts from a column value to an integer. The τ_o^{-1} transformation converts from an integer to a column value.

RelCol A transformation of a LinearInt version space. Functions in this space map from a column value to a column value. The τ_i and τ_o transformations convert from the column value of the input/output states to integers, and the τ_o^{-1} transformation converts from an integer to a column value.

Appendix B: SMARTedit scenarios

In this section we provide a brief description of each of the scenarios used to test the SMARTedit system. Although some of the scenarios are artificial, many are derived from actual repetitive tasks faced by SMARTedit users.

OKRA Given a web page containing a list of names, email addresses, scores, and the date the name was entered, select each of those items in turn for each name on the page. This information extraction task comes from the RISE repository.

addressbook Convert single-line addresses into a multi-line format, suitable for printing on a mailing label.

bibitem-newblock Convert a list of bibliographic entries from BibTeX intermediary format to human-readable form.

bindings For each call to the `bind()` function in a Python source code file, add a second function call immediately following the `bind` whose first argument is the first argument to the `bind`. Ensure that the inserted code preserves the indentation level on the previous line.

bold-xyz Boldface the name of a company everywhere it appears on the web page, using the HTML bold tags. The company name sometimes has a space between the two words in the name.

boldface-word Boldface the word SMARTedit in a LaTeX document.

c++comments Replace C++-style comments (`//`) with C-style comments (`/* */`), assuming one comment per line of text.

citation-creation Automatically construct a citation for each bibliography entry, of the form "[Hirsh, 1997]". It should be extracted from the first argument to the `\bibitem` command in each entry.

column-reordering Rearrange the columns in a structured text file containing columns separated by spaces. The contents of the first column become the last column.

country-codes Extract (country, country code) tuples from an HTML page by converting the information in an HTML table to a list of comma-separated values. This is the example task for Kushmerick's wrapper induction system (Kushmerick, 2000).

grades For a list of grades and students, one per line, delete the name of the student so that only the grade remains.

html-comments Delete the HTML comments, including their contents, from a web page.

html-latex Convert HTML to LaTeX format by escaping less-than and greater-than characters into LaTeX's math mode.

indent-voyagers For plain text extracted from a web page, fix the indentation by removing the extraneous spaces at the beginning of every line, and make the text wrap by removing the newline at the end of each line of text.

latex-macro-swap Convert from LaTeX's `\tt` operator to a user-defined macro, by changing all occurrences of {`\tt text`} to `\prog{text}`.

mark-format In a Python script, change all occurrences of the statement `outstate.selection_extent[?]` to `mark_split(outstate.selection_extent[?])`, for any expression in place of the question mark.

number-fruits Number consecutively all the lines in the file using the format "*i*.".

number-iterations Number consecutively all the lines in the file that have a pound sign (#) at the beginning of the line.

outline For a file in Emacs outline format, number all the high-level section headers consecutively starting from 1, and copy each numbered high-level section header to the top of the file.

pickle-array Convert an array of numbers from one format (a Python pickled format) to another.

prettify-paper-info Given a text file with doubly-spaced lines indented to appear in the middle of the page, remove the double spacing and the extraneous indentation such that each line is flush with the left margin.

smartedit-results Convert from the results generated by SMARTedit's scenario test harness (whitespace-separated columns) into a LaTeX table. Use the & character to separate columns and end each line with two backslash characters.

subtype-interaction In a Python script, make each defined class inherit from a given base class by adding the string "(Interaction)" before the first colon in a class definition.

xml-comment-attribute Reformat XML text so that commented-out elements are replaced with the uncommented element with the addition of an extra attribute.

zipselect Given a file containing address data, one address per line, copy each zip code to the end of the file.

Acknowledgments

This research was funded in part by the Office of Naval Research Grant N00014-98-1-0147, National Science Foundation Grants IRI-9303461 and IIS-9872128, a National Science Foundation CAREER Award, the ARCS foundation Barbara and Thomas Cable fellowship, a National Science Foundation Graduate Fellowship, and a Microsoft Fellowship. Thanks to Corin Anderson, Dan Egnor, Oren Etzioni, Nick Kushmerick, Neal Lesh, and Rachel Pottinger for helpful comments on the research. Jim Guerber and Dutch Meyer implemented parts of the system. We thank all the user study participants for their efforts and feedback.

Notes

1. A hypothesis h_1 is more general than another h_2 iff the set of examples for which $h_1(i) = 1$ is a superset of the examples for which $h_2(i) = 1$.
2. In the interests of space, we provide only proof sketches of the theorems in this paper.
3. Future work will examine how to maintain the version space and associated probability distribution when inconsistent hypotheses are assigned a very small probability.
4. The weight of a string of length i is $e^{(-0.5*(i-5))^2}$, and the weights of consistent hypotheses are normalized to form probabilities.
5. More formally, a partially ordered set with causal links (Weld, 1994).
6. Every action in the plan must support some action in the plan or the goal. This requirement is weaker than having the plan be minimal and contain no irrelevant actions; in particular, it allows redundant sensory actions.
7. Note that the use of a STRIPS-like language is essential, but many domains (such as text editing) may be difficult to encode in this formalism.

References

Agre, P., & Chapman, D. (1990). What are plans for? *Robotics and Autonomous Systems, 6:1/2*, 17–34.

Andreae, P. (1984). Constraint limited generalization: Acquiring procedures from examples. In *Proceedings of the Fourth National Conference on Artificial Intelligence* (pp. 6–10).

Ashish, N., & Knoblock, C. (1997). Semi-automatic wrapper generation for internet information sources. In *Proceedings of the Second IFCIS International Conference on Cooperative Information Systems* (pp. 160–169). Los Alamitos, CA.

Bauer, M., & Dengler, D. (1999). TrIAs: Trainable information assistants for cooperative problem solving. In *Proceedings of the Third Annual Conference on Autonomous Agents* (pp. 260–267).

Bauer, M., Dengler, D., & Paul, G. (2000). Instructible information agents for web mining. In *Proceedings of the 2000 Conference on Intelligent User Interfaces*.

Blumer, A., Ehrenfeucht, A., Haussler, D., & Warmuth, M. (1987). Occam's razor. *Information Processing, 24:6*, 377–380.

Charniak, E., & Goldman, R. (1991). A probablistic model of plan recognition. In *Proceedings of the Ninth National Conference on Artificial Intelligence* (Vol. 1, pp. 160–165).

Charniak, E., & Goldman, R. P. (1993). A Bayesian model of plan recognition. *Artificial Intelligence, 64:1*, 53–79.

Cowie, J., & Lehnert, W. (1996). Information extraction. *C. ACM, 39:1*, 80–91.

Cypher, A. (Ed.). (1993). *Watch What I Do: Programming by Demonstration*. Cambridge, MA: MIT Press.

Etzioni, O., Hanks, S., Weld, D., Draper, D., Lesh, N., & Williamson, M. (1992). An approach to planning with incomplete information. In *Proceedings of the Third International Conference on Principles of Knowledge Representation and Reasoning* (pp. 115–125).

Halbert, D. C. (1993). SmallStar: Programming by demonstration in the desktop metaphor. In A. Cypher (Ed.), *Watch What I Do: Programming by Demonstration* (pp. 102–123). Cambridge, MA: MIT Press.

Haussler, D. (1988). Quantifying inductive bias. *Artificial Intelligence, 36:2*, 177–221.

Hirsh, H. (1991). Theoretical underpinnings of version spaces. In *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence* (pp. 665–670).

Hirsh, H., Mishra, N., & Pitt, L. (1997). Version spaces without boundary sets. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence* (pp. 491–496).

Hobbs, J. (1992). The generic information extraction system. In *Proc. 4th Message Understanding Conf.*

Kautz, H. A., & Allen, J. F. (1986). Generalized plan recognition. In *Proceedings of the Fifth National Conference on Artificial Intelligence* (pp. 32–37).

Kearns, M., & Vazirani, U. (1994). *An Introduction to Computational Learning Theory*. MIT.

Kushmerick, N. (2000). Wrapper induction: Efficiency and expressiveness. *Artificial Intelligence, 118:1/2*, 15–68.

Kushmerick, N., Weld, D., & Doorenbos, R. (1997). Wrapper induction for information extraction. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence* (pp. 729–737).

Langley, P., & Simon, H. A. (1995). Applications of machine learning and rule induction. *Communications of the ACM, 38*, 54–64.

Lau, T., & Weld, D. S. (1999). Programming by demonstration: An inductive learning formulation. In *Proceedings of the 1999 International Conference on Intelligent User Interfaces (IUI 99)* (pp. 145–152). Redondo Beach, CA, USA.

Lesh, N. (1997). Adaptive goal recognition. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence*.

Lesh, N. (1998). Scalable and adaptive goal recognition. Ph.D. thesis, University of Washington.

Lesh, N., & Etzioni, O. (1995). A sound and fast goal recognizer. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence* (pp. 1704–1710).

Lesh, N., & Etzioni, O. (1996). Scaling up goal recognition. In *Proceedings of the Fifth International Conference on Principles of Knowledge Representation and Reasoning* (pp. 178–189).

Lesh, N., Rich, C., & Sidner, C. (1999). Using plan recognition in human-computer collaboration. In *Proceedings of the Seventh Int. Conf. on User Modelling*. Banff, Canada.

Lieberman, H. (1998). Integrating user interface agents with conventional applications. In *Proceedings of the 2000 Conference on Intelligent User Interfaces* (pp. 39–46). San Francisco, CA.

Lieberman, H. (Ed.). (2001). *Your Wish is My Command: Giving Users the Power to Instruct their Software*. Morgan Kaufmann.

Maes, P., & Kozierok, R. (1993). Learning interface agents. In *Proceedings of AAAI-93* (pp. 459–465).

Masui, T., & Nakayama, K. (1994). Repeat and predict—Two keys to efficient text editing. In *Conference on Human Factors in Computing Systems (CHI '94)* (pp. 118–123).

Maulsby, D., & Witten, I. H. (1997). Cima: An interactive concept learning system for end-user applications. *Applied Artificial Intelligence, 11*, 653–671.

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

Mo, D. H. (1989). Learning text editing procedures from examples. Master's thesis, University of Calgary.

Muggleton, S. (Ed.). (1992). *Inductive Logic Programming*. London: Academic Press.

Muslea, I. (2000). RISE: Repository of online information sources used in information extraction tasks. Also available at http://www.isi.edu/~muslea/RISE/, retrieved on 10/4/2000.

Muslea, I., Minton, S., & Knoblock, C. A. (2000). Selective sampling with redundant views. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence* (pp. 621–626).

Nix, R. P. (1985). Editing by example. *ACM Transactions on Programming Languages and Systems, 7:4*, 600–621.

Norton, S. W., & Hirsh, H. (1992). Classifier learning from noisy data as probabilistic evidence combination. In *Proceedings of the Tenth National Conference on Artificial Intelligence* (pp. 141–146). Menlo Park, CA.

Paynter, G. W. (2000). Automating iterative tasks with programming by demonstration. Ph.D. thesis, University of Waikato.

Pollack, M. (1990). Plans as complex mental attitudes. In P. Cohen, J. Morgan, & M. Pollack (Eds.), *Intentions in Communication* (pp. 77–101). Cambridge, MA: MIT Press.

Pynadath, D. V., & Wellman, M. P. (2000). Probabilistic state-dependent grammars for plan recognition. In *Proceedings of the Conference on Uncertainty in Artificial Intelligence (UAI-2000)* (pp. 507–514).

Rich, C., & Sidner, C. (1997). Segmented interaction history in a collaborative agent. In *Third Int. Conf. Intelligent User Interfaces* (pp. 23–30).

Ruvini, J., & Dony, C. (2000). APE: Learning user's habits to automate repetitive tasks. In *Proceedings of the 2000 Conference on Intelligent User Interfaces* (pp. 229–232). New Orleans, LA.

Schlimmer, J., & Hermens, L. (1993). Software agents: Completing patterns and constructing user interfaces. *J. Artificial Intelligence Research*, 61–89.

Shavlik, J., & DeJong, G. (1987). An explanation-based approach to generalizing number. In *Proceedings of IJCAI-87* (pp. 236–238).

Subramanian, D., & Feigenbaum, J. (1986). Factorization in experiment generation. In *Proceedings of the Fifth National Conference on Artificial Intelligence* (pp. 518–522). Philadelphia, PA.

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

VanLehn, K., & Ball, W. (1987). A version space approach to learning context-free grammars. *Machine Learning, 2*, 39–74.

Vilain, M. (1990). Getting serious about parsing plans: A grammatical analysis of plan recognition. In *Proceedings of the Eighth National Conference on Artificial Intelligence* (pp. 190–197).

Weld, D. (1994). An introduction to least-commitment planning. *Artificial Intellligence Magazine, 15:4*, 27–61. Available at `ftp://ftp.cs.washington.edu/pub/ai/`.

Witten, I. H. (1995). Pbd systems: When will they ever learn? In *Workshop on Programming by Demonstration, ML'95* (pp. 1–9). Tahoe City, CA.

Wolfman, S. A., Lau, T., Domingos, P., & Weld, D. S. (2001). Collaborative interfaces for learning tasks: SMARTedit talks back. In *Proceedings of the 2001 Conference on Intelligent User Interfaces* (pp. 167–174). Santa Fe, NM, USA.

Received October 8, 2000
Accepted August 19, 2001
Final manuscript February 6, 2002