From: AAAI Technical Report FS-00-02. Compilation copyright © 2000, AAAI (www.aaai.org). All rights reserved.

Learning How to Edit Text

Tessa Lau, Pedro Domingos, and Daniel S. Weld
Department of Computer Science & Engineering
University of Washington
Box 352350
Seattle, Washington 98195–2350
tlau@cs.washington.edu

Abstract

Programming by demonstration systems acquire procedural knowledge from examples, and then use that knowledge to execute the learned procedure on new instances. We present an architecture, called version space algebra, for learning this type of procedural knowledge, and describe an implemented system for programming by demonstration in the text-editing domain. We present results showing the system's effectiveness at acquiring procedures from a small number of examples.

Introduction

Programming by demonstration, or PBD, has the potential to empower programmers and non-programmers alike to customize their applications and minimize the effort required to perform everyday tasks. The central component of a PBD system is procedural knowledge: the sequences of actions users perform in order to accomplish their tasks. PBD systems acquire this knowledge via demonstration: the system records the actions a user performs in the user interface, and converts these actions into an internal representation, which can later be executed automatically on behalf of the user. As a result, acquiring procedural knowledge is crucial to the operation of a programming by demonstration system.

Our work has focused on the problem of programming by demonstration in the text editing domain, which is a domain familiar to all computer users, from novice to expert. We view text-editing actions as functions that transform the application state into a new state. For instance, inserting a string of text causes the state of the editor to change to incorporate the new content. Given a pair of application states, the problem is to infer the action which caused that state change.

In this work, we use machine learning to infer action functions from examples of the application state both before and after the action. We introduce a new method for learning these functions which we call version space algebra. Using version space algebra, we build up a complex search space of functions by composing together smaller, more simple version spaces. For example, a function which moves the cursor to a new position in the text file could be composed of two simpler functions: one which predicts the new row position, and one which predicts the column position.

Copyright © 2000, American Association for Artificial Intelligence (www.aaai.org). All rights reserved.

We have implemented a PBD system called SMARTedit that learns sequences of text-editing actions from demonstrations using the version space algebra. The next section motivates the text-editing domain as a fertile area for PBD research. The following section describes the characteristics of programming by demonstration as a knowledge acquisition problem. Next, we present the version space algebra algorithm used in SMARTedit to learn procedural knowledge. Finally, we conclude with a summary of experimental results.

Text-editing domain

The text-editing domain has a number of benefits. It is a concrete, familiar domain; every computer user is familiar with the basics of manipulating text. As such, the types of tasks commonly encountered in this domain are easy to explain and motivate.

Examples of repetitive text transformation tasks include converting from one file format to another, processing structured data such as tables or lists, or manipulating semi-structured data such as addresses or bibliographic entries. Often the kinds of tasks that arise in a text-editing domain are simple one-shot tasks for which recording a macro would be too much work, but which are clearly repetitive. These repetitive tasks are prime candidates for automation by a programming by demonstration system: a system that infers a program by observing demonstrations of the behavior of the program on concrete examples. The user is doing the same actions she would normally do to complete the task, but after a few examples, the PBD system is able to take over and finish the remainder of the task.

The text-editing domain is also good from the standpoint of knowledge acquisition. Although the space of possible tasks is huge, each task is accomplished by sequencing together a number of simpler actions, such as inserting the string "foo", moving the cursor to the 3rd line and the first column, or deleting the next five characters. The large variance in the number of different actions comes from the variance in arguments to each of a number of simple action types. For example, a user could potentially insert or delete just about any string, but all insertion actions are of the same

general type of action. In the text-editing domain, the number of different action types is small and easily represented within the knowledge acquisition framework.

Although there a small number of action types, each demonstrated example does not provide enough information to uniquely identify the correct action. Any demonstration could have a number of different explanations. For example, if the user moves the cursor to a new location, she could have intended to move the cursor one row down, move to the next comma character, or move twenty characters forward. Although each of these actions is consistent with the observed example, not all of them will produce the correct effect on future examples; the system must infer the correct action by observation. If the user demonstrates a second example, the system can infer that the features common to both examples are important (such as the search text preceding the cursor), and those that differ (for example, the column position of the cursor location) are unimportant.

PBD Characteristics

We formalize the programming by demonstration problem as the machine learning problem as follows. Each action a user performs during a demonstration causes the state of the application to change. For example, if she moves the cursor in a text editor, the state of the editor (which includes the cursor position, the contents of the text buffer, and so on) changes to reflect the new cursor position. The learning problem is to induce the action that explains successive application state changes given observations of the sequence of states occurring as the user performs a demonstration.

Programming by demonstration has a number of characteristics that distinguish it from other types of knowledge acquisition and machine learning problems.

- **Training data is scarce.** When each training example is constructed by a human typing on the keyboard, the system won't have a lot of examples to work with.
- **The size of the domain is very large.** As mentioned in the introduction, there are a large number of possible text editing tasks, and many ways to accomplish each task. This large domain is built up out of compositions of atomic text-editing functions, such as moving the cursor to a new location, or inserting or deleting text.
- **Tasks can range from single-step to elaborate multi-step sequences.** For our target audience, however, it's likely that tasks requiring more than a dozen steps will be too complex to demonstrate reliably.
- **Task knowledge must be executable.** After the PBD system learns how to perform a task, it must be able to execute this task on new data.
- **Task knowledge should be human-readable.** In order for a user to trust that the system's learned procedure will work correctly in the future, she must be able to understand what it has learned.
- **The task is highly supervised.** Not only does the human user construct training examples for the system to learn from, but she also corrects and verifies the system's knowledge.
- **The task admits multiple levels of abstraction.** At the lowest level, the user's keypresses and mouse actions form a primitive level of abstraction. Our SMARTedit system abstracts these primitives into higher-level actions such as "move the insertion cursor" and "insert string"; these higher-level semantic actions are part of even higher-level tasks such as "convert this record to the new format".
- **The system learns passively from observed examples.** However, in current work we are investigating means for incorporating active learning to allow the system to learn more quickly.

Our SMARTedit system for learning text-editing procedures is based on a machine learning technique which we call version space algebra that allows us to learn complex text-editing programs from small numbers of demonstrated examples. The version space algebra is described in the next section.

Version Space Algebra

In this section, we define the version space algebra: a method for composing together many simple version spaces using algebraic operations such that the whole is also a version space. We first extend version spaces to apply with any partial order (not just generality, as in Mitchell (1982)). We then define the union, intersection, join, and transform operators over these extended version spaces, and describe the conditions under which the combined version space may be efficiently maintained.

A *hypothesis* is a function that takes as input an element of its domain and produces as output an element of its range. 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 $i \in domain(h)$ and $o \in range(h)$, is *consistent* with a hypothesis h if and only if $h(i) = o$. A *version space*, $\text{VS}_{H,D}$, consists of only those hypotheses in hypothesis space H that are consistent with the sequence D of examples. When a new example is observed, the version space must be *updated* to ensure that it remains consistent with the new example. We will omit the subscript and refer to the version space as VS when the hypothesis space and examples are clear from the context.

In Mitchell's (1982) original version space approach, the range of functions was required to be the Boolean set $\{0, 1\}$, and hypotheses in a version space were partially ordered by their generality. (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 $h_2(i) = 1$.) Mitchell showed that this partial order allows one to represent 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. We extend the approach to any supervised learning problem (*i.e.*, to learning functions with any range) by allowing arbitrary partial orders. We base this proposal on the observation 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 is 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.

We now introduce a version space algebra using these extended version spaces. We define an *atomic version space* to be a version space as described above, *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*, $VS_{H_1,D} \cup VS_{H_2,D}$, is equal to $VS_{H_1 \cup H_2,D}$.*

Hirsh proved that the union of two BSR version spaces is also BSR if and only if the union is convex and definite. In contrast, we allow unions of version spaces such that the unions are not necessarily boundary-set representable, by maintaining component version spaces separately; thus, we can efficiently represent more complex hypothesis spaces.

Proposition 1 (Efficiency of union) *The time (space) complexity of maintaining the union is a linear sum of the time (space) complexity of maintaining each component version space.*

Definition 2 (Version space intersection) *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 intersection*, $VS_{H_1,D} \cap VS_{H_2,D}$, is equal to $VS_{H_1 \cap H_2,D}$.*

The considerations made above for the version space union also apply to the version space intersection.

In order to introduce the next 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 3 (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 $i \in$ domain(H_1) and $o \in$ 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, $VS_{H_1,D_1} \bowtie VS_{H_2,D_2}$, is the set of ordered pairs of hypotheses $\{\langle h_1, h_2\rangle | h_1 \in VS_{H_1,D_1}, h_2 \in VS_{H_2,D_2}, C(\langle h_1, h_2\rangle, D)\}$.*

Joins provide a powerful way to build complex version spaces, but a question is raised about whether they can be maintained efficiently. 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 with boundary sets).

Proposition 2 (Efficiency of 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 of the same type. Define $D_2 = \{d_2^j\}_{j=1}^n$ and d_2 similarly. Let D be the sequence of n pairs of examples $\langle d_1^j, d_2^j\rangle$. If $\forall D_1, D_2\ [C(h_1, D_1) \wedge C(h_2, D_2) \Rightarrow C(\langle h_1, h_2\rangle, D)]$, then $\forall D_1, d_1, D_2, d_2$*

$$\begin{aligned} &S(VS_{H_1,D_1} \bowtie VS_{H_2,D_2}) \\ &\quad = \quad S(VS_{H_1,D_1}) + S(VS_{H_2,D_2}) + O(1) \\ &T(VS_{H_1,D_1} \bowtie VS_{H_2,D_2}, \langle d_1, d_2\rangle) \\ &\quad = \quad T(VS_{H_1,D_1}, d_1) + T(VS_{H_2,D_2}, d_2) + O(1) \end{aligned}$$

In many domain representations, the consistency of a hypothesis in the join depends only on whether each individual hypothesis is consistent with its respective training examples, and not on a dependency between the two hypotheses in a pair. In this situation we say there is an *independent join* in which the consistency of a pair of hypotheses in the version space join follows from the consistency of each individual hypothesis relative to its respective training examples. If the join is independent, then the hypotheses in the version space join are exactly the hypotheses in the Cartesian product of the two component version spaces, and the join may be updated by updating each of the two component version spaces individually. For instance, given VS_1 containing hypotheses $\{A, B\}$, and VS_2 containing $\{X, Y\}$, even though A and X are consistent with their respective data, it is not always the case that $\langle A, X\rangle$ is consistent with the joint data. Although we have not yet formalized the conditions under which joins may be treated as independent, a later section gives several examples of independent joins in the PBD domain.

Note that our union and intersection operations are both commutative and associative, which follows directly from the properties of the underlying set operations. The join operator is neither commutative nor associative.

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 $VS_1 = \{g | \exists_{f \in VS_2} \forall_j\ g(j) = \tau_o^{-1}(f(\tau_i(i)))\}$.*

Transforms are useful for expressing domain-specific version spaces in terms of general-purpose ones.

Text-editing version spaces

SMARTedit implements an editor that supports a subset of the Emacs command language. As the user is editing a file, when she notices that she is about to perform a repetitive task, she invokes the SMART recorder by clicking on a button in the user interface. SMARTedit then records the sequence of states that result from the user's editing commands, and learns functions that map from one state to another.

When the user has completed one instance of the repetitive task, she clicks another button to indicate that she has completed a single demonstration. At this point, SMARTedit initializes the version space using the recorded state sequence as the first training example. SMARTedit updates the version space lazily as the user provides training examples, which allows it to consider infinite version spaces that are only instantiated on receipt of a positive training example.

The learner is able to make useful predictions after just a single training example. When the user enters another state where the same repetitive task must be performed, she invokes the learned procedure step by step. The system chooses the most likely function in the version space, executes it, and presents the resulting state to the user. If the system's guess was incorrect, the user may press a button to switch to the next most likely state, and so on. At any point, she may choose to undo SMARTedit's last action, or override the system and perform edits manually. When the user chooses a state (either by selecting one of SMARTedit's choices or by performing the action manually), this state is interpreted as another example and used to update the version spaces appropriately.

Version Space Decomposition

We represent procedural knowledge as a function from one application state to another. In the text editing domain, the state is an ordered triple (T, L, P, S), where T is the contents of the text editing buffer, $L = (R, C)$ the row and column location of the insertion cursor, P the contents of the clipboard, and S the highlighted selection range (if any). After an action is performed (*e.g.*, inserting a string at the current cursor position), the resultant state incorporates the changes made by that action.

At the highest level, our composite version space describes a set of functions mapping one text-editing program state to another. The set of functions in the version space represents all text-editing transformations we are able to learn. The goal of the learner is to induce a function from one state to another by generalizing from training examples (in the form of a sequence of states demonstrating the desired state changes). We compose the target version space out of smaller, component version spaces. Figure 1 shows the hierarchy of version spaces corresponding to the target function in the text-editing domain. Although we have presented it here as a tree for clarity, the complete version space has an equivalent representation as a formula in our algebraic notation.

The target space Program represents the class of all functions learnable in our domain. It is composed of an independent join of a fixed number of Action version spaces. (In

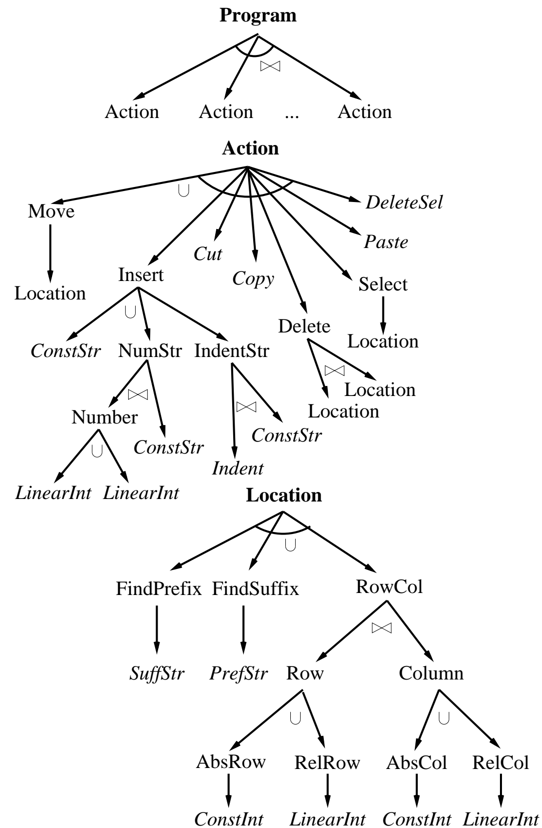

Figure 1: Version space structure for the text-editing domain. The upper tree shows the complete version space for a Program, expressed in terms of Action version spaces (middle tree). Action version spaces are in turn expressed in terms of Location version spaces (bottom tree). Italicized text denotes an atomic version space, while regular text denotes a composite version space.

practice, the number is determined lazily as the length of the first training example. Variable-length action sequences are a topic for future research.) Each Action function represents a simple command a user might perform in a text editor, such as moving the insertion cursor, inserting and deleting text at the current cursor location, and manipulating the clipboard (selecting text and copying it to and from the clipboard).

The leaf nodes in the version space hierarchy are the atomic version spaces. The ConstInt hypothesis space includes all functions of the form $f(int : x) = C$ for some integer constant C. The ConstInt version space is trivially maintained; after two or more examples, the version space collapses to one or zero hypotheses. The LinearInt hypothesis space includes all functions of the form $f(int : x) = x + C$, for some integer constant C. Its partial order and update function are analogous to ConstInt.

The AbsRow and AbsCol version spaces transform the ConstInt atomic version spaces from integer functions into functions on row or column values (*i.e.*, into functions that change the cursor position to an absolute row or column).

Similarly, the RelRow and RelCol version spaces transform LinearInt atomic version spaces into row and column functions (by changing the cursor position relative to its previous location). The Row composite version space consists of the union of AbsRow and RelRow version spaces, and likewise for the Column version space. The RowCol version space is the independent join of the Row and Column version spaces with a consistency predicate that is always true.

Besides row and column positioning, our domain representation supports positioning the cursor relative to the next occurrence of a string. If the cursor is positioned after (before) a string, we say that the user was finding 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", or the prefix "D". The FindPrefix and FindSuffix version spaces represent these types of string-searching hypotheses.

More formally, the PrefStr and SuffStr hypothesis spaces include all functions of the form $f() = T$ for some constant string T. We choose the partial order of PrefStr according to a string prefix relationship in the string T; if $f() = T_1$ and $g() = T_2$, then $f \prec g$ iff T_1 is a proper prefix of T_2. (SuffStr is defined similarly.) For clarity, we omit the function symbol and simply refer to the function as the string it produces. The least upper bound (LUB) and greatest lower bound (GLB) boundaries of the PrefStr 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 string), and the GLB becomes the singleton set $\{$"c"$\}$, where c is the character immediately following the cursor. After the first example the LUB and GLB will always contain at most one string 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. The GLB remains unchanged if the character immediately following the cursor is again c; otherwise the version space collapses to the null set.

The FindPrefix version space transforms each hypothesis in the SuffStr version space into a function from a state to a cursor position. For each function in the SuffStr version space, we create a corresponding function in FindPrefix that locates the first occurrence of this string, and returns the cursor position at the end of the matching occurrence. FindSuffix transforms PrefStr analagously, finding the beginning of each matching occurrence.

The various types of cursor-positioning functions are unioned together as the single Location version space, which is in turn transformed by many of the actions. For instance, the Move version space transforms Location to provide functions from one state to a new state with a different cursor location. The DeleteTo (SelectTo) version space transforms a Location version space to represent functions that delete (select) from the input cursor location to a new location, and output a new state in which the text between the two positions has been deleted (selected).

Scenario	Total # Exs.	# Train Exs.
OKRA	14	2
bindings	11	4
bold-xyz	50	4
column-reordering	14	2
outline	14	6
xml-comment-attribute	24	1

Figure 2: List of scenarios used to test the SMARTedit system, total number of examples in each, and number of training examples required by the system to induce a procedure that makes the correct predictions on the remaining examples.

Empirical results

We evaluated SMARTedit by testing its performance on a number of text-editing scenarios. These scenarios come from a variety of sources, including information extraction tasks drawn from the RISE information extraction repository[1], actual editing tasks gathered from real users, and artificial tasks that simulate the repetitive text manipulation arising during regular text-editor use.

Each scenario consists of a number of textual records that must be transformed. We scored SMARTedit based on the number of records that the user had to transform manually in demonstration before the system was able to learn a program that correctly transformed the remaining records. Note that this is a conservative estimate of SMARTedit's usefulness; often the system only needs minor corrections to refine its learned programs. In these cases, the user can step through SMARTedit's guesses and correct the incorrect ones, without having to demonstrate the entire transformation herself. However, for the purposes of evaluation, we have counted the examples in that refinement process as part of the hand-labelled training examples required for SMARTedit to learn.

Figure 2 shows the results for a representative collection of text-editing scenarios.

The `OKRA` scenario is an information extraction task. Given an HTML page returned by the OKRA white pages search service, the task is to extract the name, score, email address, and date of entry for each person on the page.

The `bindings` scenario is a programming task. For each call to the function `bind(arg1, arg2)`, the task is to add a second call immediately after the `bind()` call to a different function with the arguments `arg1` and the constant `b`, adhering to proper indentation conventions.

The `bold-xyz` scenario takes an HTML page with information about a company, and bolds all occurrences of the company name on the page. The company name appears both as "XYZFind" and "XYZ Find".

The `column-reordering` scenario operates on a text file containing data in whitespace-separated columns. The task in this scenario consists of moving the first column to the end of the line. The typical sequence of actions involved

[1] RISE can be found at http://www.isi.edu/ muslea/RISE/

in this task is to select the text in the first column, copy it to the clipboard, delete the selection, move the cursor to the end of the line, and paste the contents of the clipboard.

The `outline` scenario operates on an outline in emacs-outline format. The task is to number each top-level section heading starting with the number 1, and also copy all the top-level section headings to the beginning of the file to make a list.

The `xml-comment-attribute` scenario works on an XML data file. The task is to remove the XML comment tokens from commented XML elements, and add the attribute `on="no"` to those commented elements.

Related work

Unlike most previous text-editing PBD systems, SMARTedit uses a formal machine learning technique to describe the generalization that is performed by the system. Witten and Mo (1989) described the TELS system that recorded high-level actions similar to the actions used in SMARTedit, and implemented a set of expert rules for generalizing the arguments to each of the actions. TELS also used heuristic rules to match actions against each other in order to detect loops in the user's demonstrated program; it outperformed SMARTedit in this respect. However, TELS's dependence on heuristic rules to describe the possible generalizations makes it difficult to imagine applying the same techniques to a different domain, such as spreadsheet applications.

Nix (1985) described the Editing by Example (EBE) system that looked not at recorded actions, but at the input/output behavior of the complete demonstration. EBE attempted to find a program that could explain 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 is making 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) described the Dynamic Macro system for recording macros in the Emacs text editor. Dynamic Macro performed 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 performed no generalization, and it relied on several heuristics for detecting repetitive patterns of actions.

Maulsby and Witten's Cima system (1997) used 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 was able to learn a program to select all but one of the phone numbers given a single demonstration. The anomalous phone number lacked a preceding area code, and was also difficult for Cima to classify correctly.) Unlike other PBD systems, Cima allowed the user to give "hints" to the agent that focused its attention on certain features, such as the particular area code preceding phone numbers of interest. However, the knowledge gained from these hints was combined with Cima's domain knowledge using a set of hard-coded preference heuristics. As a result, it was never clear exactly which hypotheses Cima was considering, or why it preferred one over another. In SMARTedit, these types of hints could be used to bias the probabilities on its different hypotheses so as to prefer one over another.

Conclusion

We have presented the SMARTedit system for learning text-editing procedures by demonstration using the machine learning technique of version space algebra. Our PBD system acquires procedural knowledge through demonstrations of the procedure applied to a concrete example. The system infers the correct explanation for each demonstrated action and constructs a program that may be executed on a new example. The learned program, when used to automate the remainder of a repetitive task, saves the user time and effort over performing the entire task manually.

In future work, we plan to investigate the applicability of the version space algebra to other domains besides text-editing, consider the use of active learning or controlled experimentation to acquire concepts more quickly and with less user effort, and extend the core version spaces to handle disjunctive hypotheses or noisy training examples.

Acknowledgements

We thank Steve Wolfman for valuable discussion and feedback on the ideas presented in this paper.

References

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

Masui, T., and Nakayama, K. 1994. Repeat and Predict—Two Keys to Efficient Text Editing. In *Conference on Human Factors in Computing Systems (CHI '94)*, 118–123.

Maulsby, D., and 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.

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