Sheepdog: Learning Procedures for Technical Support

Tessa Lau, Lawrence Bergman, Vittorio Castelli, Daniel Oblinger
IBM T.J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598 USA
tessalau@us.ibm.com

ABSTRACT

Technical support procedures are typically very complex. Users often have trouble following printed instructions describing how to perform these procedures, and these instructions are difficult for support personnel to author clearly. Our goal is to learn these procedures by demonstration, watching multiple experts performing the same procedure across different operating conditions, and produce an executable procedure that runs interactively on the user's desktop. Most previous programming by demonstration systems have focused on simple programs with regular structure, such as loops with fixed-length bodies. In contrast, our system induces complex procedure structure by aligning multiple execution traces covering different paths through the procedure. This paper presents a solution to this alignment problem using Input/Output Hidden Markov Models. We describe the results of a user study that examines how users follow printed directions. We present Sheepdog, an implemented system for capturing, learning, and playing back technical support procedures on the Windows desktop. Finally, we empirically evalute our system using traces gathered from the user study and show that we are able to achieve 73% accuracy on a network configuration task using a procedure trained by non-experts.

Categories and Subject Descriptors

I.2.6 [**Artificial Intelligence**]: Learning

General Terms

Algorithms, Design, Human Factors, Experimentation

Keywords

Programming by demonstration, Hidden Markov Models, alignment, user study, machine learning

1. INTRODUCTION

Computer users spend a significant fraction of their time maintaining their computing systems. Most of the time, users performing maintenance procedures are following well-worn paths that many users before them have trodden. Examples of such procedures include upgrading to the latest version of an application, reconfiguring network settings after a sitewide upgrade, or removing a virus.

IT departments are typically responsible for communicating common procedures to many users. The two main approaches to communicating procedures are hardcopy documentation and automated scripts. Hardcopy documentation is expensive to author. Moreover, as our preliminary user study indicates, users often have difficulty following written directions. At the other end of the spectrum, some IT departments author common procedures as short scripts that perform the procedure automatically on the user's behalf. In GUI environments, these scripts tend to be brittle, employing heuristics such as "wait 2 seconds for the window to appear" or "click in pixel location (23, 5)". These heuristics could fail if the system load is high or the font size has changed.

We define a procedure as a set of steps taken in pursuit of a single well-defined goal. Based on an informal survey of documented technical support procedures, as well as discussions with technical support personnel, we note the following characteristics of such procedures:

1. Technical support procedures typically have a number of branches that reflect variations in the environment within which they are performed. For example, a procedure for establishing network connectivity has different subprocedures depending on whether the connection is dialup or a LAN.

2. Documentation and/or scripts rarely cover the full set of conditions that are encountered by actual users. Unexpected errors or configurations occur frequently.

3. Both documentation and scripts quickly become obsolete when the underlying system changes, and must be updated often to remain useful.

4. Many maintenance procedures require human intervention at certain points and thus are not amenable to "one-click" automation. This is particularly true when unexpected errors occur, and the user must formulate a recovery strategy, or when the consequences of taking a wrong action are very costly.

We propose a solution to the problem of capturing and maintaining technical support procedures based on programming by demonstration. Our Sheepdog system learns from multiple experts, each performing the same procedure directly on a Windows desktop. Sheepdog records execution traces, and uses them to build a model of the procedure that can be interactively executed on a user's system to perform that task. By observing multiple experts performing the same task on differently configured systems, Sheepdog learns the well-worn paths through the procedure. As end users play back the learned procedure under a wide variety of configurations, they produce new execution traces. A future version of the system will be able to learn from these execution traces as well, enabling the procedure to evolve and cover more configurations over time.

Most previous programming by demonstration systems [11, 13] have focused on simple, repetitive tasks, where the user iterates over the same sequence of actions multiple times. These systems focus on the *parameter generalization* problem: inferring the parameters to an action given multiple examples of the action performed under different conditions. For example, if the user deletes one file in one example and a different file in another example, a system might generalize to "delete the oldest file". These previous systems assumed that the sequence of steps in the procedure was relatively fixed.

In contrast, we assume that the action parameters are relatively simple, but that the sequences of actions in each demonstration vary considerably due to conditional branches or other variations in the procedure. We define the *alignment* problem as the problem of learning the complex structure of a procedure given multiple execution traces. Specifically, technical support procedures tend to be very branchy, with multiple paths to follow depending on the configuration of the system, the error messages displayed on screen, or the result of previous steps. As a consequence, procedures for technical support must include conditional branches and loops. Learning these procedures from demonstrations requires the ability to create a single procedure by aligning multiple traces, matching up steps that perform the same role in the procedure even though they may occur at different points within different traces.

In this paper, we propose a novel approach to programming by demonstration based on Input/Output Hidden Markov Models (IOHMMs) [1]. We begin in the first section with a formalization of the alignment problem. The second section describes our technical approach to alignment using IOHMMs. The third section presents the results of a user study investigating how users perform technical support procedures following written directions. Based on these results, the fourth section presents our Sheepdog interface to interactive procedure execution. In the fifth section, we present an empirical evaluation of our learning system on the user study traces. Finally, we conclude with a summary of related work and directions for future work.

2. ALIGNMENT

We define a procedure Ψ as a graph of *procedure steps* ψ in service of a single goal. Each step ψ represents a particular point in the procedure, such as "opened the TCP/IP properties dialog, setting the DNS server." At each step, the user examines the state of the display, as well as the result of previous actions, and decides what step to go to next. She then selects an action calculated to advance her to that step. Note that actions may fail or behave unpredictably, resulting in the user advancing to a different step than expected. For example, a user trying to visit a web page may find that the network is disconnected, requiring her to jump to a different point in the procedure and repair the networking configuration.

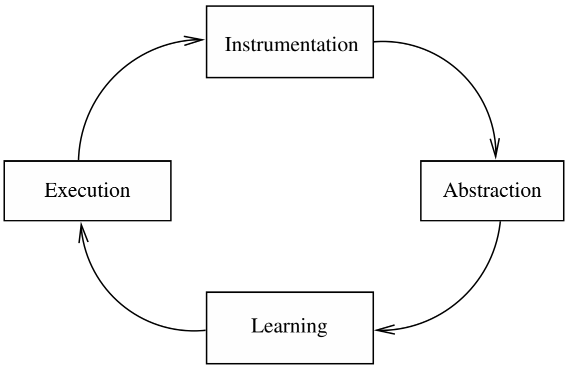


Figure 1: Architecture of the Sheepdog programming by demonstration system

An expert performing a procedure generates an execution *trace*. Each trace represents an expert's path through the procedure; different experts performing the same procedure on different operating configurations will produce different traces. At each point in time, the expert examines the *world state* S_i (the windows visible on the display, knowledge of previous actions, and other contextual information), performs an *action* A_i, and advances to the next step. We define a trace T as a sequence of *state-action pairs* t_i, where $t_i =< S_i, A_i >$. Note that the trace may contain noise — state-action pairs not relevant to the procedure.

We define an *alignment* ϕ of a trace onto a procedure as a mapping from each element in the trace to a procedure step ψ_j, such that $\phi(t_i) = \psi_j$. If $\phi(t_i) = \phi(t_j)$, then the expert passes through the same procedure step multiple times during the trace.

The alignment problem is defined as follows. Given a set of traces $T_1, T_2, ...T_N$, learn a procedure model Ψ such that each trace is optimally aligned onto the procedure. The optimal alignment maps all state-action pairs that are instances of the same logical procedure step onto the same ψ. Typically this alignment is only known by an expert familiar with the procedure. However, in practice, alignments may be evaluated using heuristics such as minimizing the number of procedure steps or maximizing the likelihood of observing a particular action at a particular step. In the next section we describe our implemention, called Sheepdog, which uses Hidden Markov Models to align traces and induce a procedure model.

3. LEARNING PROCEDURES

Figure 1 shows the high-level architecture of the Sheepdog programming by demontration system. *Instrumentation* captures events at the platform layer (e.g., the Windows operating system), and generates a stream of low-level events such as window creation and mouse click activity. This stream of events is passed to an *abstraction* component that turns the low-level events into a sequence of higher-

Table 1: Grammar used to segment low-level events into user actions

SingleClick(window w)	:= MouseDown(w)
DoubleClick(w)	:= SingleClick(w) SingleClick(w)
Keypress(w)	:= KeyDown(w)
StringEntry(w)	:= Keypress(w)
	:= StringEntry(w) Keypress(w)

level user actions and snapshots of the world state. Multiple such sequences are input into the *learning* component, which builds a model of the procedure. This procedure model is then passed to an *execution* component, which allows a user to execute the procedure on a target system.

We have implemented this architecture on the Windows operating system. The remainder of this section describes each of these phases in turn, beginning with the instrumentation and abstraction steps.

3.1 Capturing traces

We assume that an expert's interaction with a computer system consists of an alternating conversation in which first the expert performs an action, then the system responds with some change to the world state, followed by the expert performing another action, and so on. In a real-world system such as the Windows desktop, this fiction is difficult to maintain; system events such as status updates may happen asynchronously, and users often perform multiple tasks simultaneously. In this work we assume the user is concentrating on a single task, and that the user waits for the system update to complete before performing the next action. In future work we will investigate methods for learning in more dynamic environments.

In order to generate a sequence of state-action pairs, Sheepdog abstracts from low-level Windows events produced by the instrumentation layer (e.g., key `i` down, key `i` up, key `p` down, ...) into a clean, higher-level representation of user actions (e.g., type `ipconfig` into a console window). We employ a grammar parser to convert from sequences of low-level events into higher-level actions that become the individual actions in the learned procedure. Table 1 shows the set of rules encoded in our grammar; extending the grammar to recognize other actions such as drag-and-drop is an item for future work. Note that spurious actions (such as moving a window out of the way, or opening a window and then immediately closing it) are segmented by the grammar and converted into state-action pairs. We trust the alignment process to filter out this noise across multiple expert traces.

In order to learn dependencies between world state and user actions, a subset must be extracted from the large number of features in the world state. We make the assumption that most of the features relevant to the user's procedure will be displayed on screen, noting that well-designed GUIs are organized such that relevant information is displayed in the right place for the user to act on. Moreover, we make a further assumption: that the relevant information is displayed within what we call the *prime window*, the foreground window with which the user is about to interact.[1] Our system represents world state as a feature vector describing the prime window and its contents including the title of the window, the state of each of the input widgets in the window, and the contents of each of the editable widgets in the window.

A sequence of state-action pairs output by the instrumentation and abstraction layers forms a single trace. In order to learn the correct procedure including branches and loops, multiple traces of the same procedure must be aligned such that similar actions in different traces are grouped together into the same procedure step.

3.2 Aligning multiple traces

Different traces of the same procedure produce different sequences of state-action pairs. For example, if one branch of the procedure is only followed when a certain condition holds (e.g., install an update only if the current driver version is older than the desired version), then one trace may contain extra state-action pairs not present in the second trace. Thus, the procedure model must be able to produce different action probability distributions depending on attributes of the world state, such as the text displayed in the label showing the current version of the driver.

Our solution to this problem is to use an extension of Hidden Markov Models (HMMs) [14] known as Input/Output Hidden Markov Models [2]. A traditional HMM is trained with a sequence of outputs (actions in our model), producing a graph. Each node contains probability distributions over the possible next nodes and the possible outputs. Hence, if the HMM is in node X_{t-1} at time $t-1$, the HMM produces a probability distribution for the node at the next time step:

$$X_t \sim P(X_t \mid X_{t-1})$$

and it produces a probability distribution over possible actions:

$$A_t \sim P(A_t \mid X_t)$$

IOHMMs extend traditional HMMs by making these two probability distributions depend on the current world state:

$$X_t \sim P(X_t \mid X_{t-1}, S_t)$$

$$A_t \sim P(A_t \mid X_t, S_t)$$

Each node X in the graph is a hidden state representing the procedure step being executed. In other words, if a user is at step X_{t-1} and sees state S_t on the screen, she transitions to step X_t and produces action A_t.

We train an IOHMM using a modified Baum-Welch algorithm [6], which performs an iterative expectation-maximization computation. Pseudocode for the algorithm is shown in Table 2. The occupancy matrix $O_{T,t,n}$ for trace T denotes the probability of being in node n at step t in the procedure. This matrix aligns each state in the trace with the most likely HMM node for that state. The transition matrix R_{t,n_1,n_2} denotes the probability that the HMM is in node n_2 at step t given it was in node n_1 at time $t-1$. In an IOHMM, the transition matrix is also a function of the input state S_t.

[1]Although the prime window assumption has been sufficient for most of the procedures we have studied thus far, we plan to lift this assumption in future work as needed.

Table 2: IOHMM training algorithm

```
1: /* Initialization */
2: $O_{T,t,n} = randomOccupancyMatrix()$
3:
4: repeat until convergence:
5:         /* Maximization step */
6:         $R = computeTransitionMatrix(O)$
7:         $M = computeActionMatrix(O)$
8:         /* Expectation step */
9:         $O = computeOccupancyMatrix(R)$
```

During the maximization step, the transition matrix is recomputed given the best-known alignment of traces as specified by the occupancy matrix. For each node n_1 in the HMM, a classifier is trained as follows. For each of the (S, A) pairs in each of the training traces and each node n_2, a labelled training example (S, n_2, w) is constructed where S is the state observed during a trace, n_2 is the node to which to transition given that state, and w is the weight of the example, which is a scaled version of the probability that the HMM starts in node n_1 and transitions to node n_2. The action matrix, built on line 7 in Table 2, encodes the probability distribution over possible user actions given the current alignment. It is computed similarly to the transition matrix, except that its output is the user action taken at each step rather than the HMM node to which to transition. Our current system uses a nearest-neighbor learning algorithm for classification. These classifiers, one per HMM node, are applied to the world state S_t to compute the transition matrix for the next phase of the iteration.

Next, during the expectation step, the transition matrix is held fixed and the alignment is recomputed given this transition matrix, creating a new occupancy matrix. We use the standard forward-backward algorithm [14]. As in traditional HMM training, the expectation and maximization steps are repeated until the alignments have converged.

3.3 Executing learned procedures

Once trained, an IOHMM can be used to predict the next node and action given an observation of the current state. This formulation allows the procedure to represent conditional branches, where different actions are taken based on some attribute of the world state. For example, a procedure for modifying a system's DNS configuration could branch depending on whether the machine is configured for a static IP or for a dynamic IP.

During execution, our system captures the state-action pairs representing the execution trace just as if the user were demonstrating the procedure by hand. Steps performed by Sheepdog are treated identically to steps performed by the human. After the procedure is completed, the resulting execution trace can be incorporated into the procedure in order to update the learned model using information from this user's configuration. This capability enables our system to continue updating its model of a procedure as it changes over time, and incorporating knowledge about rare error conditions thay may only happen on a small fraction of desktops.

Our system allows the procedure to be executed one step at a time. At each point, the *execution cursor* represents a probability distribution over the possible steps in the procedure (i.e., nodes in the HMM). Given the cursor and the current world state, one can compute the distribution over next steps by classifying the state using each of the classifiers at each of the HMM nodes. The results are weighted by the node probabilities to produce a new probability distribution over procedure steps. The most likely action is predicted similarly given the new procedure step and the current state.

One of the challenges of procedure execution is knowing when to snapshot the system state in order to make the next prediction. System changes occur asynchronously: applications may take varying amounts of time to respond to user input; if an unexpected error occurs, the expected change to the system state may fail to occur at all. Human users of an interface have certain expectations about execution-timing based on past experience. Yet how many users have wondered whether their system has crashed or is just being slow when a window takes a long time to appear? Our working heuristic is to wait for *quiescence*, or a period of N seconds during which no system activity occurs. While this is not a long-term solution, it works well enough for the procedures with which we have experimented. In the future, we will investigate alternative mechanisms for solving this execution-timing problem. A possible approach is to attach preconditions and postconditions to each user action to determine at what time it would be applicable, potentially learning these conditions based on prior experience.

The IOHMM procedure representation allows the procedure to respond to changes in the world state that are not on the "normal" (i.e., straight-line) execution path. For example, if an error dialog pops up during a routine operation, the procedure could predict taking action to recover from that error, as long as that error was previously encountered by at least one of the experts. In our experience using the system, one author manually removed a DNS suffix from the listbox right after the system had added that entry. Sheepdog immediately noticed this change to the world state, and moved the execution cursor back to the step before the change was made, effectively ensuring that this step in the procedure was completed before continuing with the procedure.

To gain insight into possible user interfaces for our system, we conducted a preliminary user study investigating how end users perform technical support procedures while following printed directions. The next section presents the result of the study.

4. USER STUDY

We believe that a programming by demonstration approach to technical support will be an improvement over printed instructions, both for end users and procedure authors. To see how end users could benefit from such a system, we conducted a user study to observe how people follow printed instructions, and to gather data to be used in training and evaluating our system.

4.1 Methodology

In the study, we captured the mouse and keyboard actions of eleven subjects as they followed five pages of written instructions (including screenshots). These instructions were copied directly from IBM's internal technical support website, modified slightly by us to make them more readable. The instructions described a procedure to modify and verify the DNS configuration of a laptop computer. Each subject was presented with a different initial configuration, with

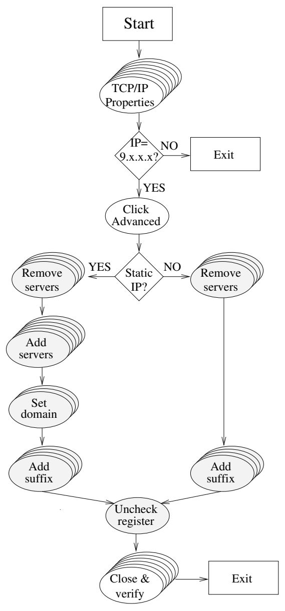


Figure 2: Network configuration procedure used in user study. Ovals denote user actions; shaded actions are performed only for certain initial configurations. Stacked ovals represent multiple actions in sequence. Diamonds indicate choice points.

the eleven configurations chosen to be distinct and to span the space of possible procedure pathways as well as possible. The subjects were instructed to follow the directions, using them to restore the system to the correct configuration. The complete procedure (shown in Figure 2) contains a conditional branch as well as a number of configuration-dependent steps — steps that are taken only for certain initial configurations.

The subjects were selected from a population of researchers and summer interns. We asked them to talk aloud as they followed the directions, and recorded their voice and their computer display on tape. We also had the participants fill out a post-study survey rating their knowledge of Windows networking configuration, the clarity of the instructions, their confidence that they followed the instructions correctly, and their confidence that the machine was configured correctly. We also asked an open-ended question about how they thought the printed instructions could be improved.

4.2 Observations

We noticed qualititive differences between subjects with little Windows expertise and those who rated themselves as having a lot of knowledge. Figure 3 shows the time duration of the trace as a function of the subject's Windows knowledge (1 means novice, 5 means expert). Duration was measured as the time in seconds from the time of the subject's first user action (mouse click or keypress) to the time of their last action. This metric does not take into account time spent reading the instructions prior to acting, though very few of our subjects actually read the directions before starting. The graph shows that novice users and expert users tended to spend more time following the instructions. The novice took a long time because she didn't understand

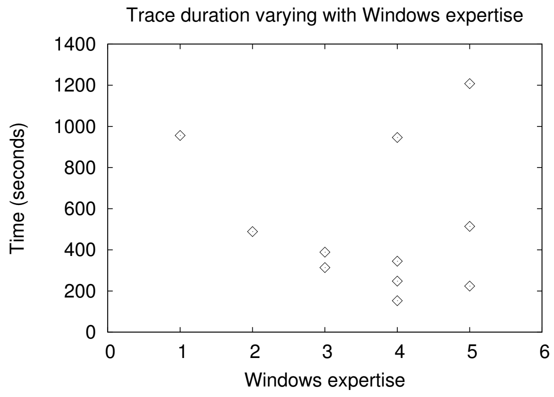


Figure 3: Time spent on the procedure plotted against Windows expertise.

the terms used, and confused "DNS server" with "IP address". The experts that took a long time skimmed over the procedure quickly, sometimes making incorrect assumptions about the configuration ("it's a laptop, therefore it must be using dynamic IP") and then had to go back and fix mistakes and steps missed due to haste.

Our observations confirmed that nearly all participants, regardless of Windows knowledge, had difficulties following printed instructions. In particular, we noted:

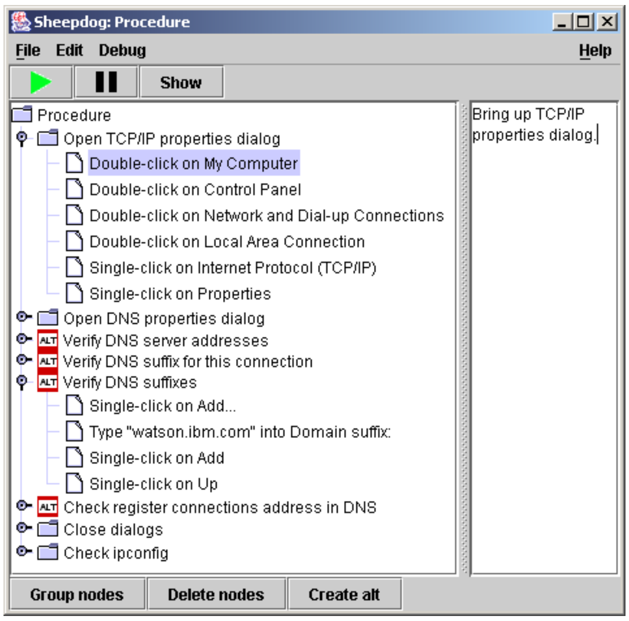
- people often had difficulty translating from text on the page to widgets on the screen;
- people tended to miss portions of the instructions, particularly separate portions of text that described branches of a conditional; and
- people had difficulty following out-of-order instructions and screenshots, especially those that weren't identical to what they saw on the screen.

4.3 Implications

It is clear to us that a guided approach to technical support procedures would be helpful to users. Novice users could benefit from having annotations explaining the key concepts in the procedure, and experts could benefit from partial automation that helps them avoid mistakes. We identified several ways in which an interactive help system could be an improvement over printed instructions:

- automatically highlighting the target widget on the screen,
- automatically detecting and branching on conditionals, and
- providing a visual indicator of which step the user is on in the procedure.

Based on observations from the study, conversations with technical support personnel, and examination of printed documentation, we identified a number of desiderata for user interfaces to interactive procedure playback. The next section describes our findings and the resulting interface design.



Procedure
- Open TCP/IP properties dialog
  - Double-click on My Computer
  - Double-click on Control Panel
  - Double-click on Network and Dial-up Connections
  - Double-click on Local Area Connection
  - Single-click on Internet Protocol (TCP/IP)
  - Single-click on Properties
- Open DNS properties dialog
- ALT Verify DNS server addresses
- ALT Verify DNS suffix for this connection
- ALT Verify DNS suffixes
  - Single-click on Add...
  - Type "watson.ibm.com" into Domain suffix:
  - Single-click on Add
  - Single-click on Up
- ALT Check register connections address in DNS
- Close dialogs
- Check ipconfig

Bring up TCP/IP properties dialog.

Figure 4: Sheepdog user interface for playing back a learned procedure.

5. USER INTERFACE

Our experience with technical support procedures led us to formulate a set of three desiderata for interactive procedure execution. First, the interface must be *collaborative* rather than automated. We believe that no system will ever be able to fully automate complex procedures such as technical support procedures; fully automated solutions are prone to failure. Second, the interface must provide *orientation* about where the user is in the procedure, what steps have been taken, and what steps remain. Third, the interface must be able to *explain* to the user what steps must be performed, and why.

We have designed a Sheepdog user interface (Figure 4) to satisfy the desiderata outlined above. We illustrate the interface on the technical support procedure employed in our user study: verifying and modifying a computer's DNS configuration. This procedure includes opening up the networking control panel, bringing up the TCP/IP properties dialog for the machine's ethernet connection, verifying that the appropriate DNS servers are set for the current configuration, and closing down the dialog boxes.

Experts using our system to generate procedure traces are presented with a macro recorder style interface that lets them start and stop recording a trace. While recording is active, our Windows instrumentation logs a trace of subwindow creation/deletion and low-level user events. During recording, the expert may add an annotation at any point by typing text into the recorder window. The annotation is associated with one or more actions (such as double-clicking on an icon) the expert is about to perform, and may describe the motivation behind a particular action or set of actions.

Once two or more traces have been recorded, the traces are processed by our learning module to produce a model. As part of this process, our system extracts a list of actions from the procedure. This list of actions can be loaded into the authoring/playback Sheepdog interface, where it may undergo further authoring. Figure 4 shows the procedure after it has been through the authoring process. The interface lets an expert use drag-and-drop to reorder steps, add hierarchical structure (such as the "Open TCP/IP properties dialog" shown in the screenshot), and define alternative branches (such as the "Verify DNS suffixes" shown). Annotations can also be added or edited in the learned procedure during the authoring phase, as shown in the right hand side of the window in Figure 4. Note that the authored procedure structure and the annotations are layered strictly on top of the learned procedure model; declaring that a branch exists during authoring supplies visual information to the consumer of the procedure, it does not actually change the structure of the HMM. In the future we plan to investigate methods for incorporating authored feedback into the learned model.

The same authoring interface is used to play back learned procedures on an end-user's desktop. The execution cursor, a blue bar highlighting a particular action in the procedure, starts at the first action in the procedure. Concurrently, the target of the first predicted action is highlighted on the user's screen. For example, the first step of this procedure is to double-click on the "My Computer" icon on the desktop, so Sheepdog visually highlights this icon by blinking it a few times. When the user presses the play button (green triangle), the system automatically performs that action, takes a new snapshot of the world state, and updates the execution cursor to point to the next action predicted by the classifiers in the HMM. If this step had been a choice point, the execution cursor would have jumped to the appropriate step depending on the features detected on-screen.

6. EVALUATION

During the user study, we used our instrumentation to capture the actions of our study participants as they performed the procedure described in the printed instructions. Since our participants were not true experts, all of the traces contained incorrect or extraneous actions. Several of the users failed to notice sections of the instructions, and thus did not produce the desired network configuration. Nonetheless, even true experts make mistakes or accomplish tasks in different ways. The strength of our system is its ability to learn the procedure despite imperfect traces.

The user study traces contained a total of 474 actions, of which 182 were unique. Figure 5 shows the number of actions as a function of their frequency in the traces. For example, 110 actions occurred only once in the data, while four actions occurred 14 times in the data. An example of one of the most-frequent actions was to click on the "Advanced..." button in the "Internet Protocol (TCP/IP) Properties" dialog box; all paths through the procedure required clicking on this button. This figure shows the wide variation in actions performed by various study participants. Because they were all performing the same procedure, one might expect that most of the actions performed would be common to many participants. However, the figure clearly shows that the vast majority of actions were performed only once.

To evaluate the performance of our learning algorithm, we input all eleven user-study traces into our system. We fixed the number of nodes for the HMM at fifteen and trained a model using a k-nearest-neighbor [4] classifier.

We evaluate the accuracy of a learning algorithm by com-

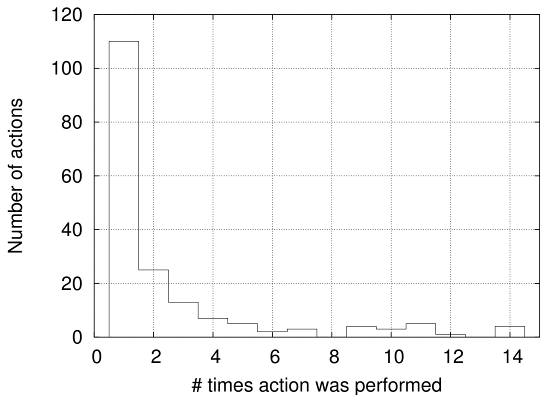


Figure 5: Number of actions that occur with a specific frequency.

Table 3: Categorization of mispredicted actions

Equivalent to actual	10
Valid but not in instructions	9
Instructions unclear	8
Other	45

paring its predicted actions against the actions an expert should have taken in the same situation. We produced a test set by having an expert (one of the authors) rerecord the complete procedure eleven times, starting with each of the different initial configurations. During recording, low-level system events such as window creation and mouse-down/mouse-up events are logged to a trace file. We then simulated the performance of the learning algorithm on each of these test traces in turn. Each test trace was fed into our system as if it were the data stream coming from a live Windows system. Whenever our system detects that a high-level user action was performed, it snapshots the world state just prior to that action, generating a pair (S, A). World state S is input to the IOHMM for classification. Given state S, the IOHMM updates its node probability distribution. Given the new node distribution and state S, the IOHMM outputs a probability distribution over possible next actions, with the maximally likely action being A'. If $A = A'$, we say the action is correct. This process repeats until the trace is completed. The accuracy of the IOHMM on this test trace is the fraction of correctly predicted actions out of the total number of actions in the trace.

Note that this metric is very conservative, and provides a lower bound on the accuracy of our system. It measures whether the HMM is able to produce exactly the same sequence of actions our expert performed. There are many ways to accomplish the same task and complete the procedure successfully; this metric does not take those into account.

Our model achieves on average 73% accuracy (min 55%, max 86%) on the test traces using the procedure trained on the user study traces. We believe this is a fairly strong result, given that the people recording the training traces failed to achieve the correct configuration of the system in several cases. The test traces contain a total of 248 actions. Of those, our model incorrectly predicts 72 of those actions. We classified the mispredicted actions into several categories, shown in Table 3. Ten mispredicted actions were equivalent to the actual expert action, but not equal, such as selecting one of two shortcuts to launch the command prompt, or pressing RETURN instead of clicking Add to dismiss a dialog box. Nine mispredicted actions were valid predictions, such as closing a dialog box, that were not explicitly mentioned in the instructions. Eight mispredictions were the result of poorly written instructions, causing a number of study participants as well as the expert to take incorrect actions.

Of the other mispredictions, most could be explained as the system choosing the wrong subtask in the procedure (such as predicting the action of removing a DNS server when the expert instead chose to add a DNS suffix). These predictions are the most difficult to choose correctly, in part because there may be multiple correct orderings of the subtasks. A focus in our future work will be to improve our system's ability to correctly predict these subtasks.

For comparison purposes, we developed a strawman algorithm that always predicts the most likely action among the actions in the user study traces. This algorithm produced 4% accuracy when tested on the expert-generated traces.

A procedure trained on traces gathered from novice users may not be representative of procedures trained by true experts. For comparison, we trained an HMM using the traces trained by an expert. Testing the HMM using the same expert traces, we observe 96% accuracy. Testing the expert-trained HMM on the user-study traces, however, we observe only 31% accuracy (min 0%, max 82%). Although some of this difference results from inconsequential actions (such as failing to press return at the end of a text field), this measure gives an indication of how poorly the test subjects' performance aligned with the sequence of instructions they should have followed by reading the directions.

7. RELATED WORK

Our work follows in the tradition of programming by demonstration established by Cypher [3] and Lieberman [12]. Most previous systems [11, 13] have addressed the generalization problem — learning action parameters when the sequence of actions is roughly static. In contrast, we learn procedures where the action parameters are relatively simple (e.g., click a button) but the complex program structure leads to varying action sequences across demonstrations.

Another area of related work is automatic programming [15] and learning programs from traces [10]. Our work is unique in applying IOHMM learning algorithms to the problem. We believe that the probabilistic finite-state model representation underlying our system is well-suited to the conditional structure of actual technical support procedures.

The learning algorithms we have used are adapted from previous work on IOHMMs by Bengio and Frasconi [2, 7] in the sequence-processing domain. We have extended the original work to support the use of arbitrary classifiers during the maximization step.

Our work is similar to prior work on task model acquisition [8] in the Collagen system. Collagen learns task models (represented as hierarchical plans) from traces of experts performing a task. Collagen expects human experts to annotate examples, marking optional steps or alternative paths. In contrast, our system uses many traces gathered from multiple experts to statistically identify common patterns in the procedure.

Another area of related work is in UNIX command line prediction [5, 9]. Such systems typically examine only the previouly entered commands, or features such as the current working directory. Our work differs in that our system examines the state of the display as well as the output of previous commands in order to predict the next action.

8. CONCLUSIONS

This paper has presented a novel approach to the alignment problem in programming by demonstration, and illustrated it in the domain of technical support for Windows desktops. Specifically, we make the following contributions:

- Formulation of the alignment problem in programming by demonstration;
- A solution to the alignment problem using Input/Output Hidden Markov Models;
- A study showing actual users' behavior when following printed instructions;
- Sheepdog, an implemented system for learning and playing back technical support procedures on the Windows platform; and
- An empirical evaluation of our system demonstrating 73% accuracy, trained on traces created by non-experts.

We see many opportunities for future work. We plan to conduct a user study of the Sheepdog system to determine its effectiveness at assisting end users with technical support procedures; results of the user study will inform Sheepdog's future development. We will investigate the problems that occur as we scale up to larger, more complex procedures; one area we want to improve on is the ability to incrementally learn the procedure model given new traces. Another improvement will be to combine our alignment approach with the parameter generalization work in previous programming by demonstration systems, enabling our system to learn loop variables and parameterized procedures. A final area of future work is extending Sheepdog to support mixed-initiative interaction, where the user can take over control from the system to demonstrate novel paths through the procedure. In particular, we plan to develop algorithms for detecting when a user deviates from the well-worn path in the procedure, and develop light-weight methods for incorporating the novel execution traces thus generated into an evolving procedure model.

9. REFERENCES

[1] Y Bengio and P Frasconi. Input-output HMMs for sequence processing. *IEEE Transactions on Neural Networks*, 7(5):1231 – 1249, September 1996.

[2] Y. Bengio and P. Frasconi. Input-Output HMM's for Sequence Processing. *IEEE Trans. Neural Networks*, 7(5):1231–1249, September 1996.

[3] Allen Cypher, editor. *Watch what I do: Programming by demonstration.* MIT Press, Cambridge, MA, 1993.

[4] B.V. Dasarathy, editor. *Nearest Neighbor Pattern Classification Techniques.* IEEE Computer Society, 1991.

[5] B. D. Davison and H. Hirsh. Predicting sequences of user actions. In *Predicting the Future: AI Approaches to Time Series Problems*, pages 5–12, Madison, WI, 1998. AAAI Press. Technical Report WS-98-07.

[6] A.P. Dempster, N.M. Laird, and D.B. Rubin. Maximum likelihood from incomplete data via the EM algorithm. *J. Royal Statistical Soc. B*, 39(1):1–38, 1977.

[7] P. Frasconi and Y. Bengio. An EM approach to grammatical inference: Input/output HMMs. In *Proc. IEEE Int. Conf. Pattern Recognition, ICPR '94*, pages 289–294, Jerusalem, October 9-13 1994.

[8] Andrew Garland, Kathy Ryall, and Charles Rich. Learning Hierarchical Task Models by Defining and Refining Examples. In *First Int. Conf. on Knowledge Capture*, 2001.

[9] Benjamin Korvemaker and Russell Greiner. Predicting unix command lines: Adjusting to user patterns. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence*, pages 230–235, Austin, Texas, July 2000. Menlo Park, CA: AAAI Press.

[10] Tessa Lau, Pedro Domingos, and Daniel S. Weld. Learning programs from traces using version space algebra. In *Proceedings of the Second International Conference on Knowledge Capture*, 2003. To appear.

[11] Tessa Lau, Steven A. Wolfman, Pedro Domingos, and Daniel S. Weld. Programming by demonstration using version space algebra. *Machine Learning*, 53(1-2):111–156, 2003.

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

[13] G. W. Paynter. *Automating iterative tasks with programming by demonstration.* PhD thesis, University of Waikato, February 2000.

[14] Lawrence R. Rabiner. A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition. *Proceedings of the IEEE*, 77(2):257–285, February 1989.

[15] L. Siklóssy and D. A. Sykes. Automatic program synthesis from example problems. In *Proceedings of the Fourth International Joint Conference on Artificial Intelligence*, pages 268–273, Tblisi, Georgia, USSR, 1975. San Francisco, CA: Morgan Kaufmann.