Machine Learning Techniques to Make Computers Easier to Use

Hiroshi Motoda [a] and Kenichi Yoshida [b]

[a] *The Inst. of Scientific and Industrial Research, Osaka University, Mihogaoka, Ibaraki, Osaka 567, Japan*

[b] *Advanced Research Laboratory, Hitachi, Ltd., Hatoyama, Saitama 350, Japan*

Abstract

Identifying user-dependent information that can be automatically collected helps build a user model by which 1) to predict what the user wants to do next and 2) to do relevant preprocessing. Such information is often relational and is best represented by a set of directed graphs. A machine learning technique called graph-based induction (*GBI*) efficiently extracts regularities from such data, based on which a user-adaptive interface is built that can predict next command, generate scripts and prefetch files in a multi task environment. The heart of *GBI* is pairwise chunking. The paper shows how this simple mechanism applies to the top down induction of decision trees for nested attribute representation as well as finding frequently occurring patterns in a graph. The results clearly shows that the dependency analysis of computational processes activated by the user commands which is made possible by *GBI* is indeed useful to build a behavior model and increase prediction accuracy.

1 Introduction

Computers are still not easy to use. The main reason is their ignorance about the user. Each user has different goals (tasks, resources, criteria, ...) and different preferences (habits, abilities, styles, ...). Computer systems do not understand these things. It is knowledge that makes understand possible, and the knowledge of the user is nowhere. The user information that is available to an interactive computer system is limited, and thus, the user model acquisition is a difficult problem. Classical acquisition methods like user interviews, application-specific heuristics, and stereotypical inferences are often not appropriate, and a better automated method is being sought.

Finding regularities in data is a basis of knowledge acquisition, and extracting behavioral patterns from the user information is one such problem. Since each user may do the same thing in a different way, identifying the information that can characterize the user and be automatically collected is crucial. Once such information is found and if an appropriate machine learning technique can induce regularities in each user's behavior to carry out his/her intended task, we can use them to guide the daily work and to do some preprocessing, which may facilitate easiness of usage and increase efficiency. In order for this to work satisfactorily, we rely on the assumption that situation, purpose, intention, meaning, concept are all embedded in some structure, and thus, extractable by mechanical operation.

We discuss three learning tasks, command prediction, script generation and file prefetching in a multi task environment. The scope of user behavior is limited to a sequence of task execution (*e.g.*, editing, formatting, viewing, etc.) using plural application programs.

Most studies that attempted to develop a user-adaptive interface system only analyzed the sequence of user behaviors, from which to automate the repetitions (See 8). In this setting, the data can easily be represented by attribute-value pairs, each attribute denoting the sequence order and its value, the command, and a standard classifier, *e.g.*, [22] can be directly applied to induce a set of classification rules without any difficulty. However, since the command sequence does not necessarily typify the user's behavior, the user model constructed from only the sequence information may not adequately capture the user's behavior (we have confirmed this and the results are shown later). We focused on the process I/O information that is also automatically collected along with the command sequence. Since this is dependency information and its relationship cannot be fixed in advance, it is not straightforward to represent this by attribute-value pairs and apply a standard classifier.

We show that graph-based induction [25] can nicely be applied to the three learning tasks. In this paper, we revisit *GBI*, show how it can extract typical patterns from a set of directed graphs and how it can induce classification rules using a similar technique in the Top Down Decision Tree (TDDT) induction algorithm. The first and the second learning tasks are implemented as *ClipBoard* which is a window like UNIX shell [26], and the third task is implemented as *Prefetch daemon* that is hidden from the user. The results clearly show that the dependency analysis of computational processes activated by the user's commands, which is made possible by *GBI*, is indeed useful. *ClipBoard* is in daily use and its prediction accuracy and response time are satisfactory. *Prefetch daemon* works as expected only for I/O intensive task due to an implementation problem, and thus needs further improvement.

The following section introduces the three learning tasks. Subsequent sections describe the learning method *GBI* and summarize the results of learning experiments performed to date. The last two sections consider lessons learned from this study and directions for future research.

2 Learning Tasks

Command prediction is a real time task that takes a user's operational history and predicts the next command. Figure 1 shows, in a simplified form, an example of operational history when a user is making a document using a *latex* document formatter. The bold arrows show the command sequence. The history includes, in addition to this, I/O relationships between commands, and thus, takes the form of a directed graph. Each link has a label that corresponds to a file extension. For example, the link connecting *emacs* to *latex* has a label *tex*. However, one link is reserved for sequence information. *ClipBoard* keeps recording and updating the history, and at any point of operation, predicts the next command. The learning task is to induce classification rules from the past history. It is a supervised learning. For each command in the past, a directed graph of a certain depth (number of sequentially connected links) and width (number of sibling links) are taken out. Each directed graph forms a training example. Its root is a class and the rests are considered to be nested attributes.

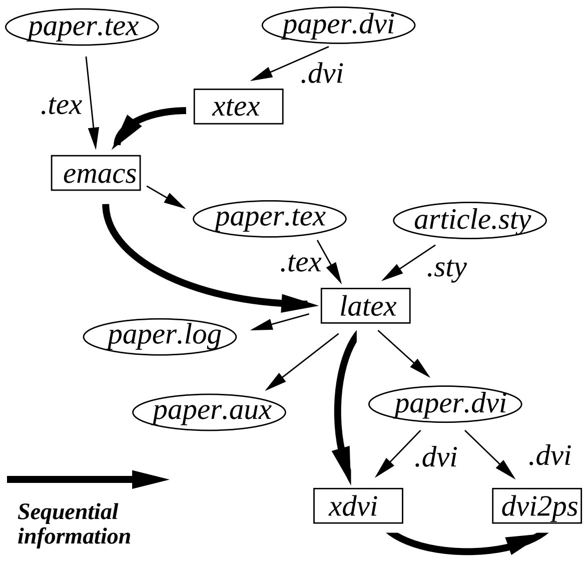


Fig. 1. I/O relationships between commands (applications)

Script generation is a batch task that extracts frequently occurring patterns from a large graph representing a history of order of days, generalizes the arguments and generates shell scripts to execute a sequence of operations by a single command. It is a kind of conceptual clustering and is unsupervised learning. Figure 2 shows an example of the generated scripts when a user repeatedly calls up *emacs*, *latex* and *xdvi*.

```
#! /bin/csh -f      # Document Processing Script
emacs  $1           # Edit document.
                    # Extension is assumet to be .tex.
latex  $1           # Format document.
xdvi  $1:r.dvi      # Preview result on screen.
                    # Extension is assumet to be .dvi.
```

Fig. 2. Example of a generated script

File prefetching is a real time task that predicts files to be used in the immediate future and prefetches them into the cache. Unlike the command prediction, prefetching must predict a few steps ahead, so not only the next command but also a few more together with the associated file I/O. The learning task is done in a batch mode using a large directed graph. It is unsupervised learning. The task is to extract frequently occurring patterns first like script generation, from each of which a prefetch rule is generated and then to merge them into a single trie structure (example shown in Figure 12). The prefetching is made in real time based on this trie. Since prefetching is automatic, this task is invisible.

3 Graph-based Induction

3.1 Finding Regularities in a Directed Graph

GBI was originally intended to find interesting concepts from inference patterns by extracting frequently appearing patterns in the inference trace. It uses a single heuristic: anything that appears frequently is worth paying attention to. In [25], it is shown that *GBI* was able to discover the notion of NOT and NOR from the qualitative simulation traces of an electric circuit. In this application, the original inputs were causal relations of voltage and current between various nodes of the circuit; there was no notion of logical operation. However, by finding regularities in the input traces, *GBI* was able to lift up the abstraction level and find more abstract concepts. Later, we showed that the same idea can be applied to other types of learning (speed up learning and classification rule learning) [27].

The original *GBI* was so formulated to minimize the graph size by repeatedly replacing each found pattern with one new node and contracting the graph. The graph size definition reflected the sizes of extracted patterns as well as the size of contracted graph. This prevented the algorithm from continually contracting, which meant the graph never became a single node. Because finding a subgraph is known to be NP-hard, the ordering of links is constrained to be identical if the found two subgraphs are to match, and an opportunistic beam search similar to genetic algorithm was used to arrive at suboptimal solutions. In this algorithm, the primitive operation at each step in the search was to find a good set of linked pair nodes to chunk (pairwise chunking). When applied to finding interesting concepts, GBI returned a set of subpatterns for which the graph size became minimum. Whether the found concepts are in deed interesting and useful depends on the definition of the graph size and is empirical. When applied to building a classifier, GBI returned a set of rules for which the predicted error rate (either by cross validation or by test data), the real measure, became minimum while using the graph size as a primary measure to minimize.

Because the search is local and stepwise, we can adopt an indirect measure rather than a direct estimate of the graph size to find the promising pairs. On the basis of this notion, we generalize the original GBI, and further extend it to cope with the classification problem. The idea of pairwise chunking is given in Figure 3, and the general algorithm of GBI in Figure 4.

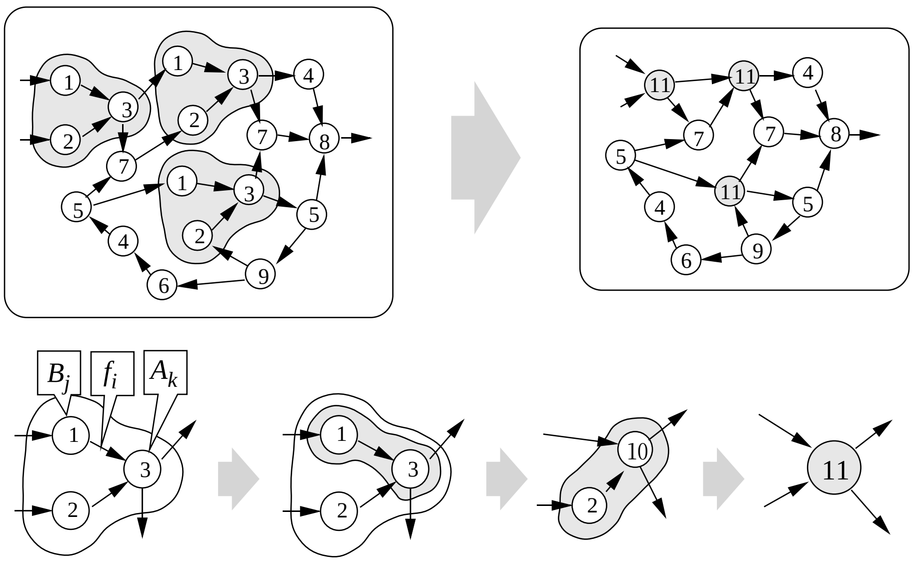


Fig. 3. The idea of graph contraction by pairwise chunking

The selection criterion of the pair nodes should be such that its use can find interesting patterns (*e.g.,* patterns occurring more frequently than others or patterns more easily identifiable than others). Proper termination condition must be used in accordance with the selection criterion (*e.g.,* iteration number, chunk size, change rate of selection measure, etc.). Examples of such measures are information gain [20], information gain ratio [22] and gini index [1].

We use information gain as a measure here because the pairwise chunking is a

```
$GBI(G)$
    Selection of pair nodes $(A_k, f_i, B_j)$ in $G$
    Chunk the pair nodes into one node: $c$
    $C := \{c\}$
    $G_c :=$ contracted graph of $G$
    $If$ termination condition not reached
        $C := C \cup GBI(G_c)$
    $end\_if$
    Return $C$
```

Fig. 4. Generalized algorithm of GBI

binary split. It works well for many cases, but the other indexes can be used in the same way. Unlike decision tree building where the measure is used for selecting a relevant attribute, here we have to select linked pair nodes. Each node has a value (color) and each link has a label. We can interpret the triplet (A_k, f_i, B_j) as saying that the value of the i-th attribute f_i of the parent A_k is B_j or when the i-th attribute f_i takes the value B_j, its immediate result is A_k. The problem is which (k, i, j) to select to chunk. A natural way is to focus on one of the three elements, and select the best remaining two to identify the chosen element. Three alternatives exist: a) focus on k, b) focus on i and c) focus on j. Case a) tries to find the attribute and its value pair that best characterizes the chosen immediate result. Likewise, case b) tries to find the result and the attribute value pair that best characterizes the chosen attribute, and case c) tries to find the attribute and its result pair that best characterizes the chosen attribute value. Which one to adopt depends on what the directed graph represents in terms of the original problem description. The default is to choose a) and we use this option for script generation and file prefetching.

In what follows, only case a) is described. The other two are obtained by permutating the subscripts. Let the underline in the subscript mean its complement (*e.g.*, $\underline{i}$ means the attributes other than the i-th.), and the superscript *yes* and *no* mean the result of the division by a test. The amount of information that is required to identify k before selecting the triplet is

$$I(n_k) = -\sum_{i,j} \frac{n_{k,i,j}}{N_k} \log_2 \frac{n_{k,i,j}}{N_k},$$

where N_k is the number of nodes that have value A_k and $n_{k,i,j}$ is the number of the triplets (A_k, f_i, B_j) (*i.e.,* the number of nodes that have value A_k and their i-th attributes have value B_j.).

The amount of the information that is required to identify k after the selection is

$$E(A_k, f_i, B_j) = \frac{N_{k,i,j}}{N_k} I(n_{k,i,j}^{yes}) + \frac{N_{k,\underline{i,j}}}{N_k} I(n_{k,i,j}^{no}),$$

where

$$I(n_{k,i,j}^{yes}) = -\frac{n_{k,i,j}}{N_{k,i,j}} \log_2 \frac{n_{k,i,j}}{N_{k,i,j}} = \frac{N_{k,i,j}}{N_{k,i,j}} \log_2 \frac{N_{k,i,j}}{N_{k,i,j}} = 0$$

$$I(n_{k,i,j}^{no}) = -\sum_{i',j'(\neq i,j)} \frac{n_{k,i',j'}}{N_{k,\underline{i,j}}} \log_2 \frac{n_{k,i',j'}}{N_{k,\underline{i,j}}}.$$

Here, note that the triplets that go into the *yes* branch are all identical, implying $I(n_{k,i,j}^{yes}) = 0$.

Info-gain$(k, i, j) = I(n_k) - E(A_k, f_i, B_j) =$

$$\frac{1}{N_k}\{\sum_{i',j'(\neq i,j)} n_{k,i',j'} \log_2 \frac{n_{k,i',j'}}{N_{k,\underline{i,j}}} - \sum_{i,j} n_{k,i,j} \log_2 \frac{n_{k,i,j}}{N_k}\}$$

The best attribute i and its value j for each k to select is

$$\text{Argmax}_{(i,j)}\{\text{Info-gain}(k, i, j)\} = (k, i_{0k}, j_{0k}).$$

Thus, the best triplet is determined to be

$$\text{Argmax}_k\{N_k \text{Info-gain}(k, i_{0k}, j_{0k})\} = (k_0, i_0, j_0).$$

This is recursively repeated until a termination condition is satisfied.

3.2 Inducing Classification Rules

In case of the classification problem, we interpret the root node as a class node and the links directly attached to it as the primary attributes. The node at the other end of each link is the value of the attribute, which has secondary attributes. Thus, each attribute can have its own attributes recursively, and the graph (*i.e.*, each instance of the data) becomes a directed tree (See Figure 5). In this case, the pairwise chunking must start at the root node and go backwards (from successor to predecessor) following the links. Here, we have to recursively select the attribute and its value pair that best characterizes the class. So the selection measure is slightly different from the normal *GBI* described above, *i.e.*, the chunking is made for the triplet $(*, f_i, B_j)$ where only the attribute f_i and its value B_j are specified.

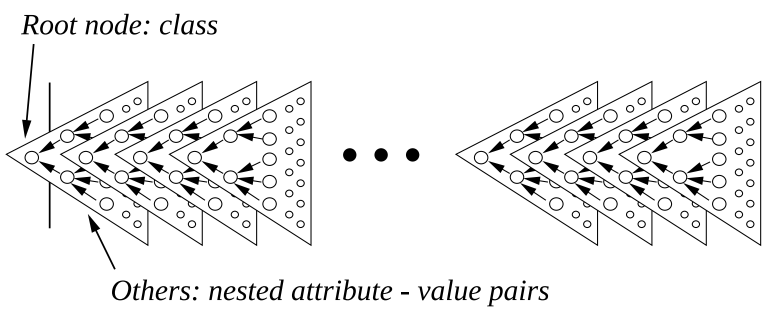


Fig. 5. Data representation for a classification problem

The amount of information before the selection is

$$I(n) = -\sum_k \frac{n_k}{N}\log_2\frac{n_k}{N},$$

where n_k is the number of nodes that have class value A_k, and $N = \sum_k n_k$. The amount of information after the test by the attribute f_i whose value is B_j is

$$E(f_i, B_j) = \frac{N_{i,j}}{N} I(n_{i,j}^{yes}) + \frac{N_{\underline{i,j}}}{N} I(n_{i,j}^{no}),$$

where

$$I(n_{i,j}^{yes}) = -\sum_k \frac{n_{k,i,j}}{N_{i,j}}\log_2\frac{n_{k,i,j}}{N_{i,j}},$$

$$I(n_{i,j}^{no}) = -\sum_k \frac{n_{k,\underline{i,j}}}{N_{\underline{i,j}}}\log_2\frac{n_{k,\underline{i,j}}}{N_{\underline{i,j}}},$$

$$N_{\underline{i,j}} = \sum_{i',j'(\neq i,j)} N_{i',j'}, \qquad n_{k,\underline{i,j}} = \sum_{i',j'(\neq i,j)} n_{k,i',j'}.$$

$$\text{Info-gain}(i,j) = I(\overline{n}) - E(f_i, B_j) =$$

$$\frac{1}{N}\{\sum_k\{n_{k,i,j}\log_2\frac{n_{k,i,j}}{N_{i,j}} + n_{k,\underline{i,j}}\log_2\frac{n_{k,\underline{i,j}}}{N_{\underline{i,j}}} - n_k\log_2\frac{n_k}{N}\}\}$$

Thus, the best attribute f_i and its value B_j to select for testing is

$$\text{Argmax}_{(i,j)}\{\text{Info-gain}(i,j)\} = (i_0, j_0).$$

This is recursively repeated until each subgroup, after testing, contains a single class value or some stopping condition is satisfied.

4 ClipBoard Interface

Figure 6 shows the system configuration for *ClipBoard Interface* and *Prefetch Daemon*. The process I/O recorder is a part of the operating system and records all the I/O operations of each command issued. This information is represented together with the command sequence by a directed graph as operation history. *GBI* program runs on this graph and generates prediction (classification) rules and typical (frequently appearing) patterns. The mouse-based command controller uses these to 1) select the next command, and to 2) create UNIX shell scripts. The prefetch daemon uses the typical patterns to generate prefetch rules and merges them into a trie structure to 3) prefetch files.

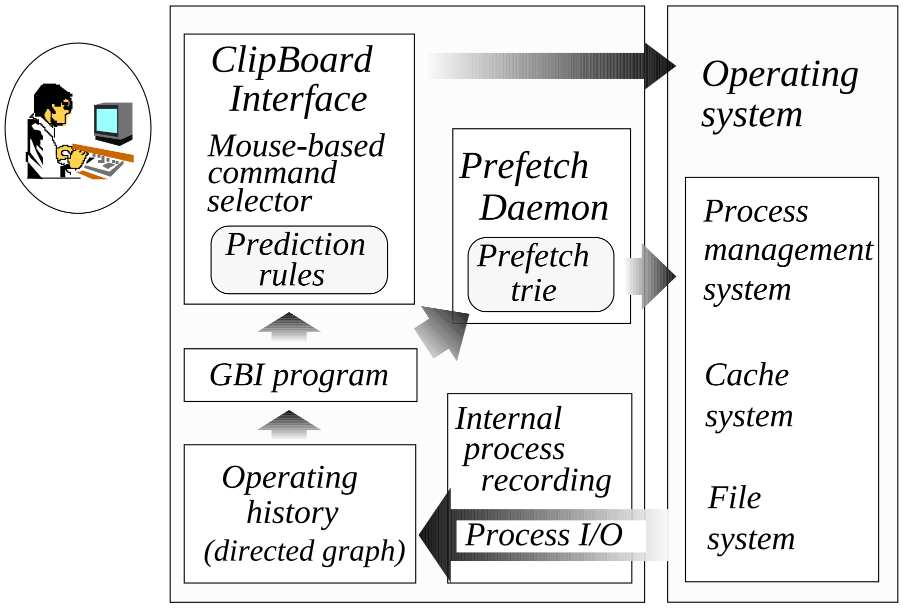
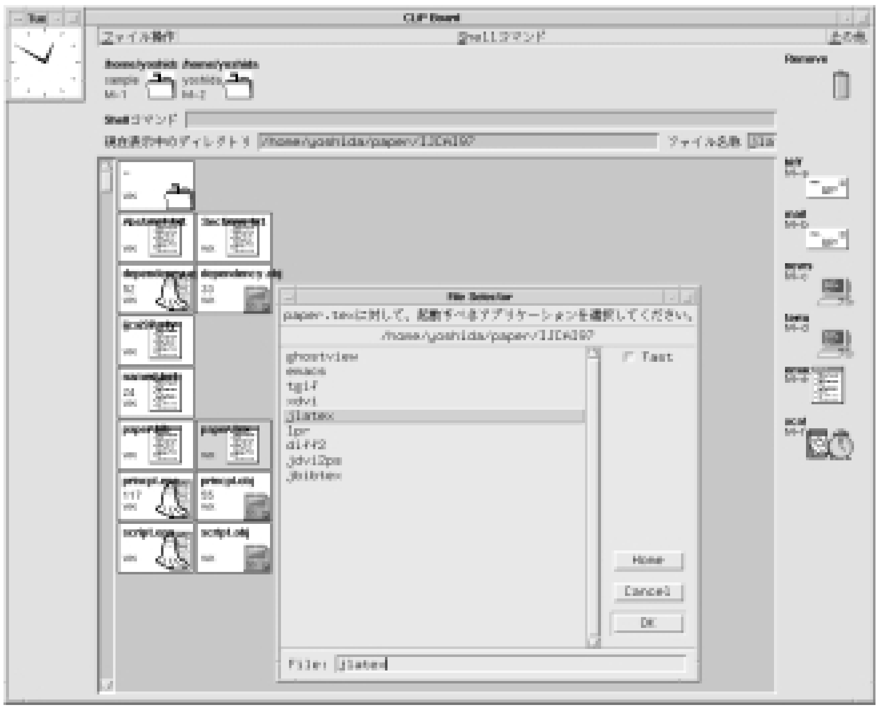


Fig. 6. System configuration of ClipBoard and prefetch daemon

Figure 7(a)(b) displays the screen images of *ClipBoard* during a simple document processing task. We have adopted the file metaphor. Rather than suggesting the next command directly, *ClipBoard* attaches an icon for the next command to each of the files that the user is now working on. Each small box on the screen represents a file. Each time a new file is created, a new box appears. When *ClipBoard* starts without any information, no icon appears in the box. In this case the user selects a file to be processed, then the dialogue box appears and the user can specify the command. The same dialogue box can be used to override the predicted command if the user does not want to run that command for the file s/he has selected. Figure 7(a) shows the latter. The selected file has an emacs icon, but the user wants to run latex. Entering a new command for the first time or overriding the predicted command triggers *ClipBoard* to initiate induction by *GBI* and update the prediction rules. *ClipBoard* never asks the user for information, thus it learns by being told. The user can always override *ClipBoard*'s recommendation. No learning takes places as far as the prediction made by *ClipBoard* is correct. Each time a new induction is initiated, a new data set is created from the past history including the one which *ClipBoard* has misclassified and bas been notified of. *ClipBoard*



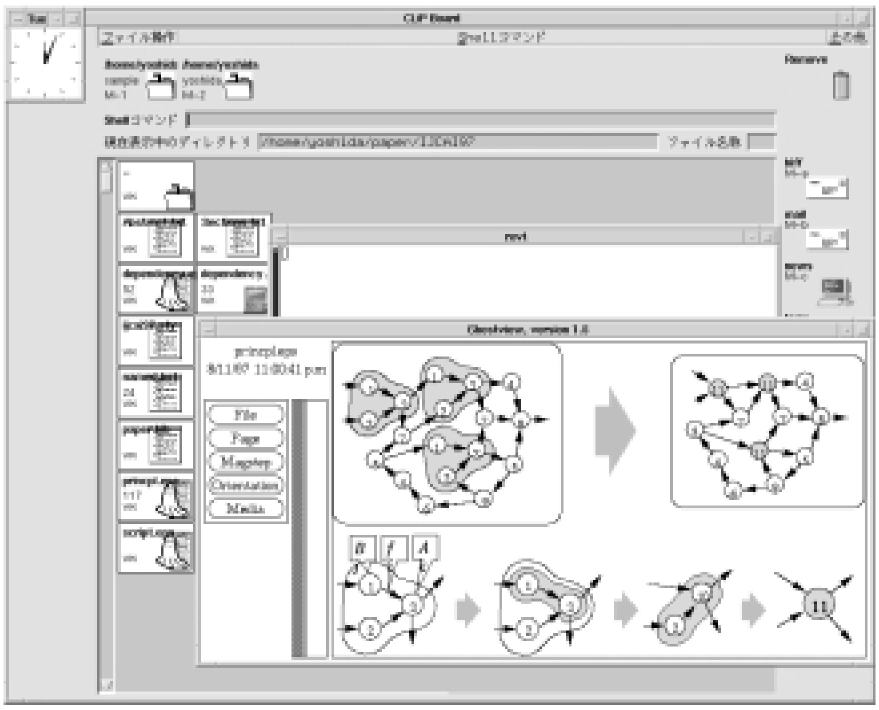
paper.texに対して、起動すべきアプリケーションを選択してください。

/home/yoshida/paper/IJCAI97

```
ghostview
emacs
tgif
xdvi
jlatex
lpr
diff2
jdvi2ps
jbibtex
```

File: jlatex

(a) Selecting *latex* command for a text file



(b) Clicking the *ghostview* icon on the postscript file to preview the text

Fig. 7. Screen images of *ClipBoard*

Table 1
An example of operation history

Step	Application	Input File
(A)	xtex	paper.dvi
	emacs	paper.tex
	latex	paper.tex
(B)	xdvi	paper.dvi
(C)	dvi2ps	paper.dvi

tries to learn the appropriate command for each file extension, and the files that have the same extension receive the same icon. The icon for the same file changes over time reflecting the context changes. The user clicks the icon to run the command. In Figure 7(b) the user clicked the *ghostview* icon that is attached to the postscript file and is viewing the document. Currently, *ClipBoard* interface is written by Tcl/Tk. The *GBI* program has both C and Lisp versions. The prefetch daemon is written by Java.

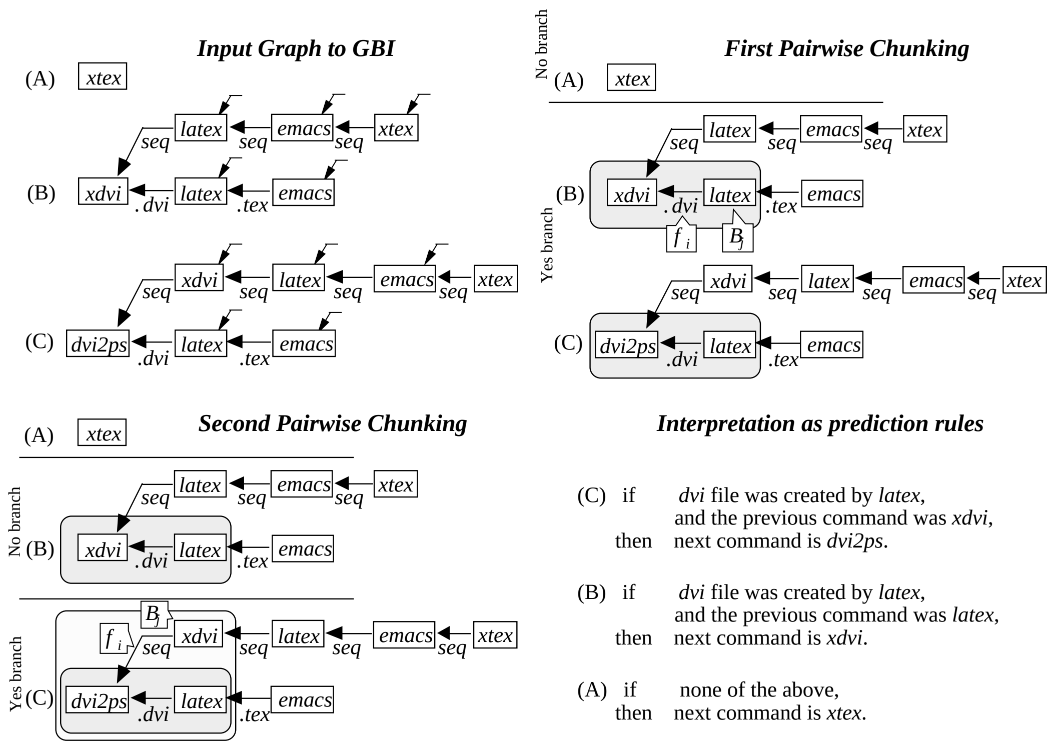
4.1 Command Prediction

4.1.1 I/O Information Analysis

Consider an operation history in Table 1. As shown in steps (A), (B), and (C), the file *paper.dvi* is processed by three different commands: *xtex*, *xdvi* and *dvi2ps*. The top left figure in Figure 8 shows the corresponding directed graphs that are the inputs to *GBI*. Every command has both sequential and dependency links, but for the sake of simplicity this is emphasized only for the root node. The algorithm described in 3.2 first chooses the *dvi* attribute (f_i) and its value *latex* (B_j) for testing, and chunks the triplets (*xdvi, dvi, latex*) in (B) and (*dvi2ps, dvi, latex*) in (C) (first pairwise chunking in Figure 8). The *no* branch contains only one instance, (A), and the *yes* branch contains two instances, (B) and (C). Next, the algorithm chooses the *sequential* attribute (f_i) and its value *xdvi* (B_j) for testing and chunks the triplet ((*dvi2ps, dvi, latex), seq., xdvi*) (second pairwise chunking Figure 8). This separates (C) from (B) and the induction stops[1]. The bottom right figure in Figure 8 is the interpretation of the induction results as prediction rules.

GBI assumes the existence of a strong correlation between the linked at-

[1] In reality, there are many occasions in history where *dvi* files are used by the same command that has different dependency, in which case the chunking process becomes more complicated.



(C) if *dvi* file was created by *latex*,
and the previous command was *xdvi*,
then next command is *dvi2ps*.

(B) if *dvi* file was created by *latex*,
and the previous command was *latex*,
then next command is *xdvi*.

(A) if none of the above,
then next command is *xtex*.

Fig. 8. Induction by pairwise chunking

tributes. As described in 3.2, the algorithm follows the standard TDDT induction, but the attributes to be selected are dynamically modified in the process. Note that it is impractical to represent the graph structure by a single table of attribute-value pairs.

4.1.2 Evaluation

The above algorithm for the classification problem was implemented and tested for the command prediction problem using both artificially generated and real operation data.

Artificial data were generated approximating user's behavior by a probabilistic model which comprises five different tasks that runs repeatedly with some probability distribution. Each task is also described by a probabilistic model. The model used is shown in Figure 9. Although not shown in this figure, the next state is probabilistically determined by a finite past history that includes file I/O dependency. About 2000 different sequences were generated. In going from one command to the next, noise was added according to the model shown in Figure 10. Such commands as *ls, ld, du*, etc., that do not directly depend on the previous command, were used as a noise. Three fold cross validation was used to evaluate the prediction accuracy. Because the data are sequential, use of cross validation could worsen the predictive accuracy. We assume that the data are stationary. The results are shown in Table 2 for three different levels of noise. This table includes the results obtained by other methods for comparison.

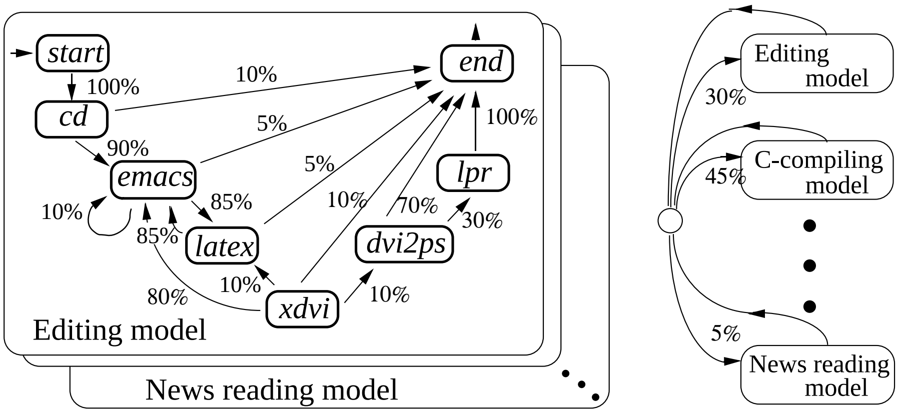


Fig. 9. Task representation by a probabilistic model

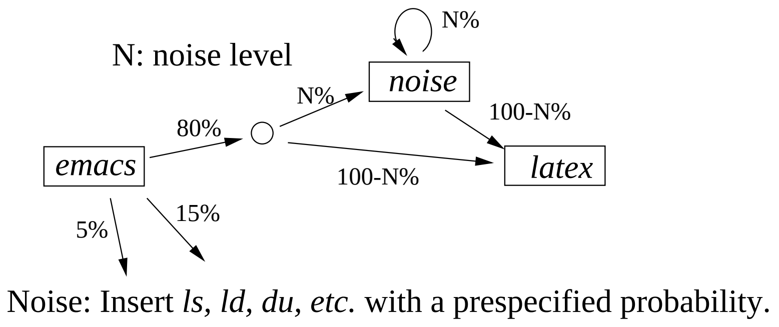

N%
N: noise level
noise
N%
100-N%
80%
emacs
latex
100-N%
5%
15%
Noise: Insert ls, ld, du, etc. with a prespecified probability.

Fig. 10. Noise model used in the artificially generated data

Table 2
Prediction accuracy for artificially generated data

Noise	Induction Methods				
	Default	*LD*	*CART*	GBI_1	GBI_2
15%	35.5	35.5	48.6	51.5	73.6
20%	33.8	33.8	45.2	52.1	73.7
25%	33.0	32.2	41.6	47.2	72.1

Default: Value for most frequently used command
LD: Linear Discrimination Method
GBI_1: Without dependency info. for the root node (command to predict)
GBI_2: With dependency info. for the root node

There are five induction methods in Table 2. *Default* is the simplest way of prediction that always assumes the most frequently used command to be the next command. *LD* is a linear discrimination method [9]. *CART* [1] is a well known decision tree classifier. There are two cases for GBI. GBI_1 is the case where dependency information is used only for the commands (nodes) preceding the root node. In other words, no dependency information is used for the root node. This reflects the fact that the argument is not known in advance to predict the next command. GBI_2 is the case where the dependency information for the root node (command to predict) is also used. This corresponds to a case where the file to process is specified, and this is exactly what the current *ClipBoard Interface* does. This is not a strong restriction because files associated with a given task are generally known and the prediction of the command for each of these files can be made with this method. In [26] the former is called command prediction and the latter, application selection.

The way the data were prepared for $CART$ and GBI_1 needs some elaboration. In Figures 5 and 8 the links directly attached below the root node are of two kinds: one for previous command (sequence information) and the other for input files (I/O dependency information). Since $CART$ can't handle the nested attribute representation (graph structure), last five consecutive commands without dependency information (except the command immediately before the root, which is already there) were moved below the root node. Thus, the root node has five links with no grandchildren. LD also used the same information as $CART$. To do a fair comparison, in GBI_1 the data were processed in the same way but with dependency information. Said differently, four copies of the dependency trees, each corresponding to one of the past four consecutive commands before the last one were attached to the root node. GBI_2 as described above used the dependency information at the root node, and no copy of the dependency trees for the past commands were attached (as in Figure 8). The depth and width were set at 10 and 100 respectively for both GBI_1 and GBI_2. The width 100 means that we use as many file I/O dependency as it occurs.

LD gave the same answer as the default and did not improve the accuracy. $CART$ gave much better results but less than GBI_1. We also used $C4.5$ [22] on the separate data set, but the results are almost the same. The difference between $CART$ and GBI_1 is the effect of dependency. To our disappointment, the difference is much smaller than we expected. It is about 5% in this artificially generated data set. However, as we show next, this is indeed big enough for the real data set. The result of GBI_2 indicates that the I/O dependency information immediately before the command to predict, plays an important role in increasing the accuracy of prediction.

The same algorithm was tested against the real data that had been taken from the log of daily usage over three months of a single user. The length of command sequence is about 2000, which includes about 100 different kinds of commands. Two-thirds of them was used as a training data set and the rest as a test data set. The result is shown in Table 3. It is clear that GBI

outperforms the other methods. Interestingly, as stated above, GBI_1 is much better than $CART$ in real data. This is probably because the number of commands actually used is much larger than the artificial data case and the noise level is also higher. Unfortunately the value for GBI_2 is not available for the same data set. It is instead estimated by the daily usage when the performance approached the steady state[2]. Once again, the role of I/O dependency is clear.

Table 3
Prediction accuracy for real data

Methods	$Default$	LD	$CART$	GBI_1	GBI_2
Accuracy %	22.6	22.6	34.6	57.8	~80.0

The non-essential commands such as *ls* and *df* can be naturally ignored by a mouse-based interface system. If we ignore these effects and focus on the important commands, we obtain the results shown in Table 4, which is by far better. While evaluation of *ClipBoard* is still ongoing, most of the important commands predicted by *ClipBoard* is quite adequate, and the user does not feel any burden in using it.

Table 4
Prediction accuracy of selected commands (GBI_1)

Command	*emacs*	*make*	*latex*	*backup*	*xdvi*
Accuracy %	69	85	92	86	100

4.2 Script Generation

4.2.1 I/O Information Analysis

In a multi-window and/or a multi-task environment, a single user can work on different shells simultaneously. Even though the I/O operation sequence of each task has regularity, the overall I/O sequence is affected by the subtle timing of each task progress. The graph structure can encode the correct information even in such an environment. To be precise, the I/O recorder keeps track of 1) all process creations in the operating system, and 2) all I/O operations (*open* system calls). Thus, it is possible to extract relationships between commands that may have been issued across the different shells (see Figure 11). We use the whole graph to extract patterns. The extracted patterns

[2] The depth was set 5 and the width 128 (this is maximum and automatically adjusted).

are frequently appearing ones in the history, and we convert them to shell scripts. The input file name is changed to the argument of the script with extensions retained (See Figure 2).

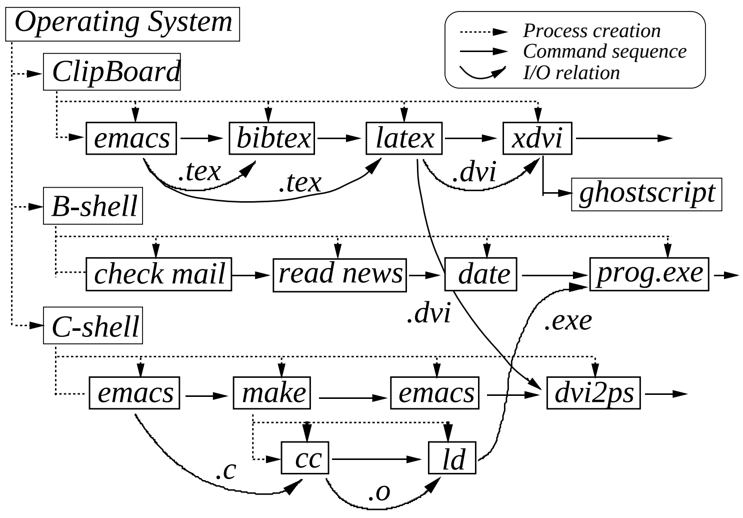


Fig. 11. Relationship between commands across the different shells

4.2.2 Evaluation

Table 5 lists the scripts with more than three commands that are generated from the sample history, which involves about 10,000 process creations and about 130,000 I/O operations. The number of processes includes system programs that were not invoked by the user (*e.g.*, telnet daemon, line printer spooler daemon, etc.), some user commands (*e.g.*, shell scripts), and created child processes. The number of the actual commands invoked by the user was approximately 2000, and the actual graph had about 2000 nodes and 16,000 links. The computation time to extract the frequently appearing patterns was about 20 min.

Table 5
Generated scripts with more than three commands

Scripts	Scripts	Scripts

1. `emacs $1`
`diff $1 $1.bak`
`cp $1 $1.bak`
	2. `emacs $1`
	`diff $1 $1.bak`
	`cp $1 $1.bak`
	`make`
		3. `cp $1 $1.bak`
		`chmod 500 $1`
		`rm $1.bak`

4. `emacs $1.c`
`cc $1.c`
`a.out`
	5. `emacs $1.tex`
	`latex $1.tex`
	`xdvi $1.dvi`
		6. `emacs $1.c`
		`cc $1.c`
		`strip a.out`

Since the algorithm only considers the frequency (more precisely equivalent as evaluated by the information measure), evaluation of the usefulness or importance of the generated scripts must be rendered to the user. Unlike the case for command prediction, there is no direct feedback from the user. The scripts in Table 5 have clear meanings except script 3. Without having knowledge about the C compiler, *ClipBoard* could generate scripts 4 and 6. *ClipBoard* did not use any pre-specified knowledge about *latex* and related commands in generating script 5. Script 1 is a unique script for this particular user. Without *ClipBoard* the user has to write this by him or herself. As we note, these scripts are not difficult for a user with standard knowledge to program. So this function is not used regularly.

5 Prefetch Daemon

5.1 I/O Information Analysis

In a multi-task environment different users also can work on the same machine for different tasks (*e.g.*, editing and programming). Just like in the case of script generation, *GBI* analyzes the process data and represents them by a set of directed graphs, from which it extracts typical patterns. Each of the patterns represents an aspect of the user (we call it user model for convenience). Figure 12 shows how these patterns are used to prefetch files. First, each of the patterns is converted into a prefetch rule. Unlike the command predictions, the point here is not to predict the root node from the rest, but to predict from the bottom (first) node in the sequence how certain files are going to be used along the subsequent command execution. Each rule consists of a sequence of events, *i.e.*, command executions and I/O operations, with a list of files to be prefetched. For example, in pattern A, when *emacs* is entered, it is known that four files (*bibtex, .bst, latex, .sty*) are going to be used in the immediate future. It is noted that the user is editing *.bib* and *.tex* files, thus these files are not in the candidates of prefetching. When *bibtex* is entered, it is known that three (*.bst, latex, .sty*) are going to be use soon.

Next, all the prefetch rules are merged into a single trie structure. For example, the first node of the two patterns are the same *emacs* and are thus merged. In order to improve the prefetch accuracy, the statistical information in the log is used to prune the files[3]. In the merged first node only two files (*make, bibtex*) are prefetched because there is a branch and the probability of going to each is known to be above a certain threshold. At the next node down right

[3] There are many patterns that partially overlap and/or are subpatterns of the others. A threshold can be set to the number of occurences of the files for them to be prefetched.

(*make*) only two files (*cc, .h*) are prefetched because the log indicates that a certain fraction of compiling operation is failure and it is not wise to prefetch all here. The generation of trie structure is performed as a batch process.

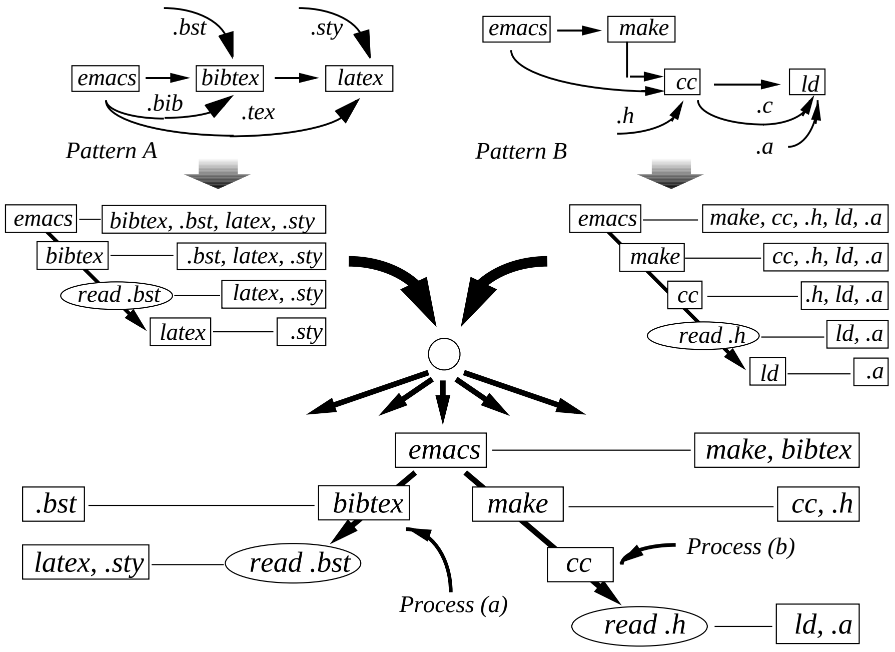


Fig. 12. Prefetch rules and a merged trie structure for prefetching

5.2 Evaluation

After the batch process constructs the trie structure, the prefetch daemon uses this trie structure to prefetch files. The daemon maintains the status information for each process. If a new process is activated, the prefetch daemon creates a new pointer which points the root node of the trie structure. If the process executes command *emacs* (*i.e.*, the program memorized in the succeeding trie node), the daemon prefetches program files *make* and *bibtex* and updates the pointer. In Fig. 12 *process (a)* shows the position of the pointer after it executed *emacs* and then *bibtex*. Likewise *process (b)* shows the position of the pointer after it executed *emacs*, *make* and *cc* in this order. Each time it updates the pointer, it also looks for the same command from the root (*i.e.*, the command just below the root node) as if a new process with this command was initiated. When it finds the command, it also prefetches the associated files. This is recursive. If the actual events of the process exhibit a different sequence from the trie, all the pointers for this process are removed and the prefetch daemon ignores the process until a new process is initiated.

The above prefetch mechanism was tested for the daily usage data (the length of the log was about 38,000). After removing the processes that were not invoked by the user, the size of the graph from which to extract frequently appearing patterns amounted to about 14,000 nodes. The prefetch cache size was automatically adjusted by OS (it varied 5MB to 50MB). The initial trie had approximately 1000 nodes and was pruned to about 1/10 using the statistical information from the log. Although further experiments are necessary, the preliminary experiments show that the trie structure has high prediction accuracy. For the experiment we conducted, the hit rate was almost 100%.

Unfortunately, even with the high hit rate, the current implementation slows down the CPU intensive tasks due to the CPU resources used by the prefetch daemon. We could only speed up I/O intensive tasks. It could indeed speed up the invocation of a large program such as *X-windows* and *mule* to the extent that we did not feel we had waited. The process switching overhead and the JAVA byte code interpretation are the sources of the problem. A kernel embedded file prefetcher that is coded by C and assembler would solve the problem.

6 Running Examples of ClipBoard

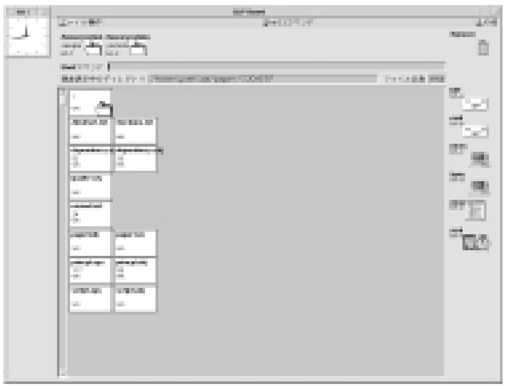
In this section, we briefly describe how *ClipBoard* display actually changes in response to user's operation. The first part (Figure 13(a) to (h)) is for before learning, and the second part (Figure 14(a) to (h)) for after learning. Figure 13(a) shows that there are twelve files in the directory where the task is editing a document. Since this is before learning, no predicted icons are shown yet. The user selects *emacs* from the dialogue box for the main input file, *paper.tex* (Figure 13(b)), which leads to Figure 13(c) where the user is editing the file. At this stage *ClipBoard* learns that a file with *.tex* extension must be an input to *emacs* and *emacs* icon has appeared in the *paper.tex* box for the first time. The user continues to browse by *emacs* one of the two text files with extension *. txt* both of which are called from the main input file (not shown). Now the *emacs* icons have appeared also to these two files that have the same extension (three *emacs* icons in Figure 13(d)). The user next views one of the *eps* files by *ghostview* and as before all the *eps* files have now the *ghostview* icons (three *ghostview* icons in Figure 13(d)). Then the user selects the main input file which has now *emacs* icon, and runs *latex* by overriding the *emacs* (dialogue box in Figure 13(e)). The icon of the main file has now been changed from *emacs* to *latex* and new files such as *paper.dvi, paper.aux, etc.* have been created (Figure 13(f)). Next the user selects the newly created *paper.dvi* file and runs *xdvi* to view it (Figure 13(g)). Note that the *xdvi* icon has appeared for the *paper.dvi* box (Figure 13(h)). *ClipBoard* keeps learning like this by being told and inducing the classification rules.

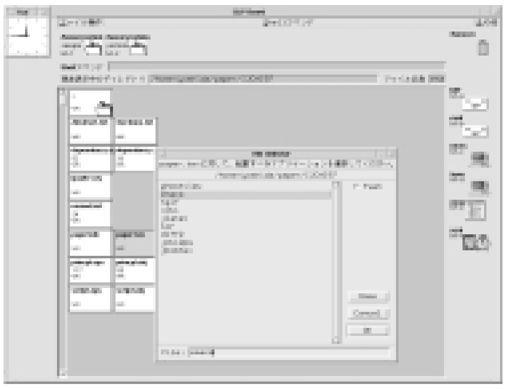
Figure 14(a) shows the files in the same directory after *ClipBoard* has learned enough. Note that the three text files have now the *emacs* icon and the *dvi* file has now the *dvi2ps* icon. Suppose that the user edits the file that is called by the main file (Figure 14(b)). Then the icon of the main file changes from *emacs* to *latex* because *ClipBoard* has learned that *latex* must be run when one of the input files has been changed although the main file remains the same (Figure 14(c)). The user then clicks the icon to run *latex*. Note that the icon has changed back to *emacs* and the icon for the *paper.dvi* has changed to *xdvi* because *ClipBoard* has learned that the next action is to view this file (Figure 14(d)). The user clicks this icon and views the paper (Figure 14(e)). The icon changes back to *dvi2ps* because the user has already viewed the file (not shown). Next the user edits the *bib* file by *emacs* and runs *bibtex* (not shown). Then the icon of the main file has changed from *emacs* to *latex* prompting that we need to run *latex* and the icon for *paper.dvi* has changed from *dvi2ps* back to *xdvi* (Figure 14(f)). The user then runs *latex* twice and the icon for *paper.dvi* changes back to *xdvi* (not shown). So the user clicks *xdvi* icon and view the final results (Figure 14(g)). The icon has changed again back to *dvi2ps* and the user clicks the *dvi2ps* icon to create a *ps* file, which can be viewed by *ghostview* and sent out to a printer (Figure 14(h)). As can be seen in this short running example, once *ClipBoard* has learned, all we need is in most cases simply to follow the predictions by clicking the icons. In summary, *ClipBoard* satisfies the following desirable features: It is a system that does not require a hand-coded knowledge base to model a user, learns in real time, is accurate enough, does not force a user to accept its recommendation (so user has a control), is easy to use, and learns to improve its performance over time.

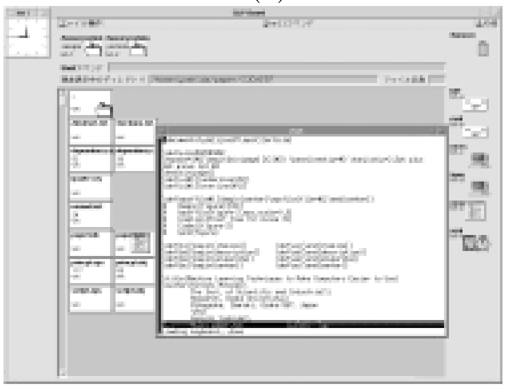
7 Discussion

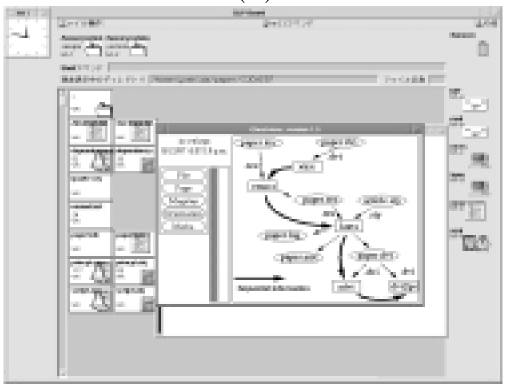
7.1 Learning Semantics from Syntax

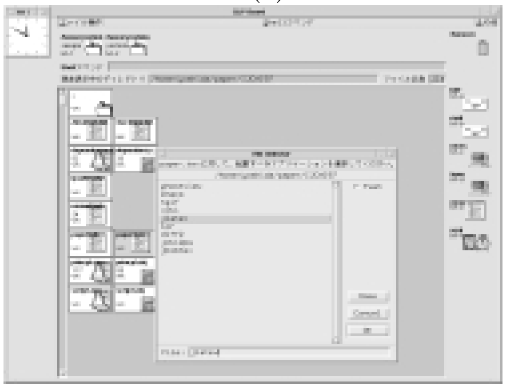
Although what *GBI* does is simply extracting the syntactic/statistical nature of what a user has done in the past, it is still possible to extract useful semantics of the user's behavior. The user never tells the start of his/her task to *ClipBoard*, but the scripts generated by *GBI* does capture a piece of meaningful tasks. Most crucial is the information source. The surface form of the user's input (*i.e.*, command sequence) was not enough. Other information that is hidden and invisible (*i.e.*, process I/O) contributed much. Standard techniques (*e.g.*, measures based on information theory, cross validation, etc.) that statisticians have developed are also important factors.

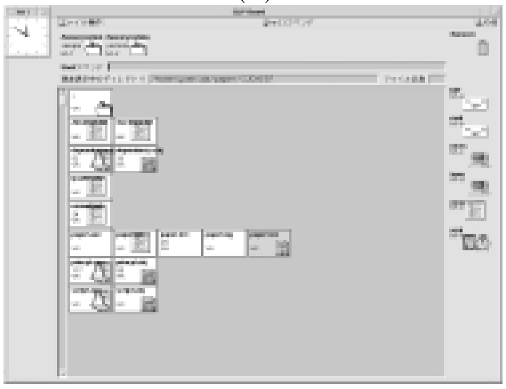

(a)

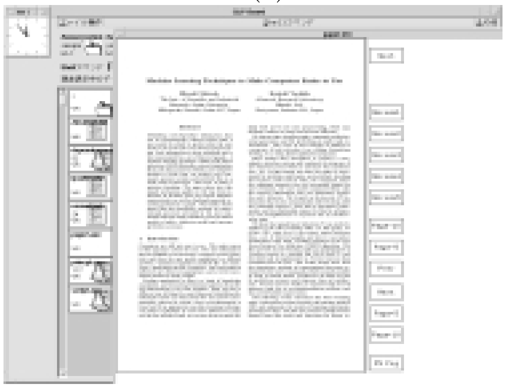

(b)


(c)


(d)


(e)


(f)


(g)

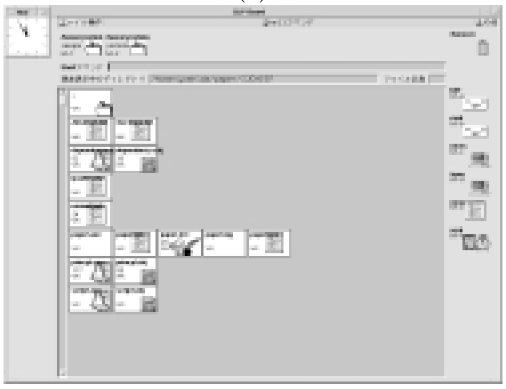
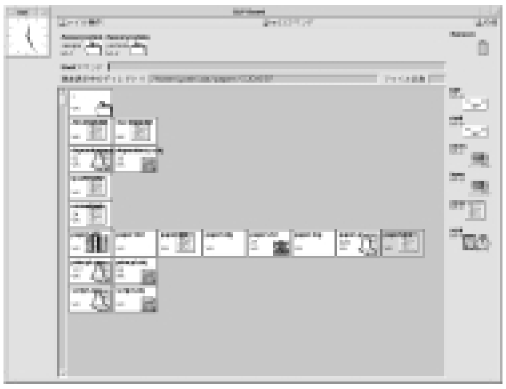

(h)

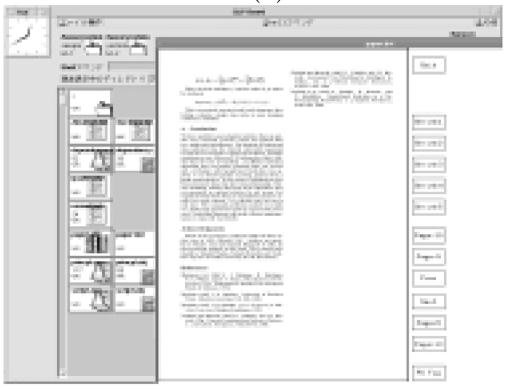
Fig. 13. Running example (before learning)

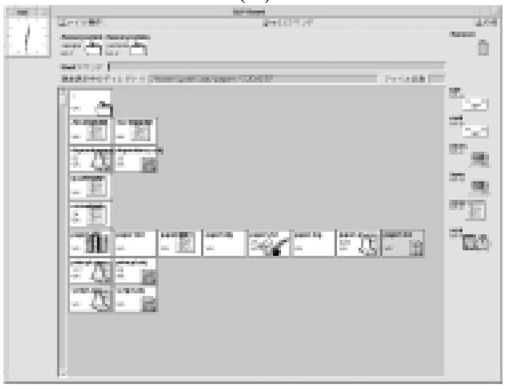

(a)

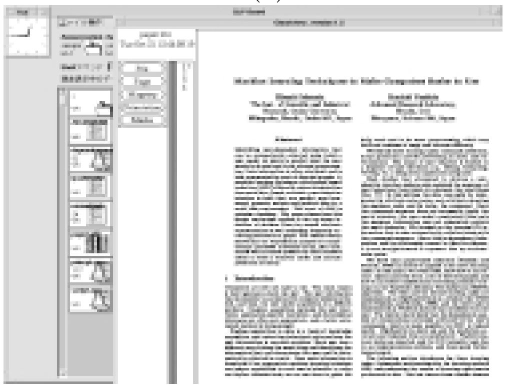
(b)

(c)

(d)


(e)


(f)


(g)

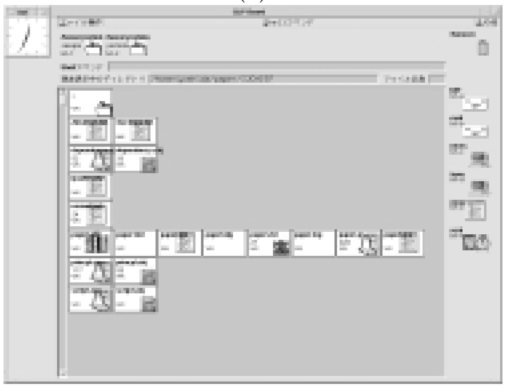

(h)

Fig. 14. Running examples (after learning)

[19] addresses the importance of the *context* in an interface system. File extensions we used in our analysis to capture the I/O information helped provide rich context. Other information that may help capture the user's behavior is command exit status and time of execution. For example, if the user fails to compile a program because of a simple syntactic error, the next step tends to be an editing task. If s/he succeeds, it tends to be a test run. Thus, the exit status seems to be informative. Since most users tend to check e-mail in the morning, the time of day also seems to be informative. Experiments using *ClipBoard* utilizing such information are currently under investigation.

The method of encoding information is also important. We encoded the I/O information from *how a file was* **made** *by application program*. The experimental results suggest the adequacy of this encoding, but this is not the only way to use the I/O information. For example, *how a file was* **used** *by application program* is another way of encoding. Figure 15 shows a graph format that was designed to emphasize this aspect. In this example sequence, a file *.tex* which was created by *emacs* are used by *latex* three times. This information is explicitly encoded in the lower graph (none for *latex*(a), once for *latex*(b) and twice for *latex*(c)). We confirmed that this encoding also works well in a version of *ClipBoard* that uses this as an alternative to the sequence information. Note that this encoding has a noise-tolerant nature. User errors, such as mistyping and wrong command selection, and unexpected interrupts, such as new mail arrival, sometimes cause noise in sequence information. The replaced I/O information is less affected by such noise.

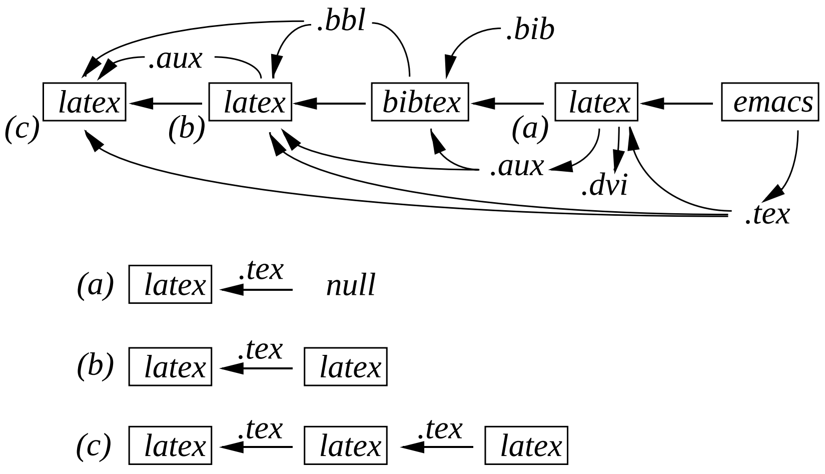


Fig. 15. Graph encoding the knowledge of *how a file was* **used**.

The use of I/O information exhibits its merits when multi tasks are being executed simultaneously as shown in Figure 11. *ClipBoard* distinguishes between the file names that have the same extension. Thus, for example, even when a user is editing two different document simultaneously *ClipBoard* can learn the correct classification rules and never mixes up the operations on these two documents.

7.3 Method of Analyzing User Behavior

If the user is always logical and consistent, the analytical methods, such as explanation-based learning, are adequate in making the user behavior model. Unfortunately, the user is sometimes illogical and inconsistent, and capriciousness makes it difficult to apply analytical methods to the interface problem. The statistical methods, such as linear discrimination and k-nearest-neighbor [9], and empirical learning methods, such as [20], seem to be more adequate. The errors, *i.e.*, mistyping and wrong command selection, are naturally ignored as noises in these methods. However, these methods are not suited to handle structural data as was the case for this study.

If we set the maximum width (number of input files) per command and the maximum depth (number of chains of I/O relationship), it is possible to design a table of attributes and values that can record all the necessary information. If we take the maximum width as 20 and the maximum depth as 5, a table with $\simeq 20^5$ attributes is created [4] . This is only for one instance. If the analysis requires 1000 cases, the table size becomes huge.

Inductive logic programming (ILP) [21,14,17], on the other hand, is more expressive and captures the relations most naturally in first-order logic. It can also handle noise [21,17]. To explore the potential of this approach, we tried to use *FOCL*, one of the most efficient ILP systems, to analyze the real data used in Section 4.1.2. However, *FOCL* took more than four hours to find the first test condition of the first rule; therefore we had to give up this approach [5] .

GBI's expressiveness lies in between the attribute-value pairs and the first-order logic. It is a limited form of propositional calculus. Its learning potential is much weaker than that of ILP, but stronger than that of the attribute-value representations and yet as efficient. We demonstrated that command prediction we addressed in this paper is a class of the problem that *GBI*'s framework fits well. Furthermore, *GBI* can handle both supervised learning (classification) and unsupervised learning (conceptual clustering) in a unified way. The former induces discrimination rules and the latter characteristic rules.

[4] Note that a typical (not maximum) single run of the *latex* command receives 50 input files (*e.g., .tex, . aug, . sty. .bbl, .eps, .tfm, .fmt, etc).*

[5] We have not taken advantage of the search strategy used in *GBI.*

7.4 Meta-level Learning and other improvements

Currently command prediction and script generation are treated as separate tasks. While using *ClipBoard*, repetition was frequently observed. This suggests the possibility of meta-level learning, that is learning regularity of *ClipBoard*'s behavior. Here the repetition is about the sequence in which the icons were clicked. Since those icons are attached to the files, this is different from the command sequence prediction. A simple mechanism which interactively compiles these found sequences into macros (or equivalently shell scripts) would be useful.

We are aware of some minor things that could improve *ClipBoard*'s ease of use. For example, we could improve *ClipBoard*'s selection function by highlighting the second suggestion shown in the dialog box (See Figure 7(a)) when the user wants to override *ClipBoard*'s first suggestion (which is displayed by icon).

7.5 Other Applications

The idea of *ClipBoard* seems to be useful in designing interface systems of other kinds such as automatic chart format selection in spread sheet and data base, naive-user guidance and installation guidance-and-diagnosis systems. The last two are meant to apply the knowledge learned from expert behavior to non-expert users. During the development of *ClipBoard*, we were able to use the I/O information itself, *i.e.*, the raw history data, for debugging purposes. A good display system of this information seems to be beneficial even for an expert user.

One promising application that goes beyond those within a single machine is dynamic World Wide Web caching. The rapid growth of information gathering through WWW causes a heavy network overload, and the resulting slow response is causing a problem. Distributed caching is a promising approach. Our preliminary study [24] by *GBI* shows that it is possible to reduce the overload of the backbone traffic by extracting frequent occurring data transmission patterns from the wide area network flow and using this to allocate distribute cache storage. The simulation assumed the situation where 32,000 WWW servers are accessed simultaneously by 16 clients. Each client and proxy had a 32 MB cache capacity. The data were taken from the access log of our proxy server that included 2.3 million data transfers (18.7 GB in size). Figure 16 shows how the backbone traffic changes with the time of day with and without cache, from which we observe 26% reduction of traffic between 10 am and 8 pm. The traffic reduction at the peak time amounts to 100 MB. Figure 17 compares the data flow for two different cache systems: the distribute caching by *GBI* and the conventional hierarchical caching. Both uses local caching and the figure shows how much reduction is made possible after the flow is reduced by the local cache. We can observe the reduction is 2.5 times larger on the average between 10 am and 8 pm.

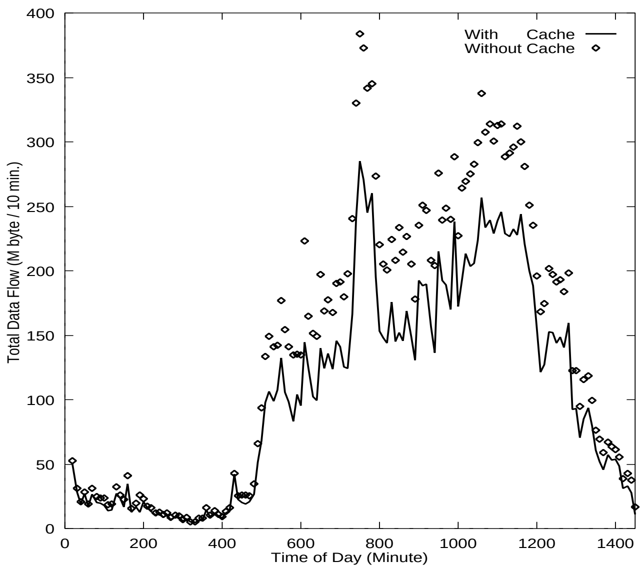


Fig. 16. Network traffic distribution over the time of day with and without cache

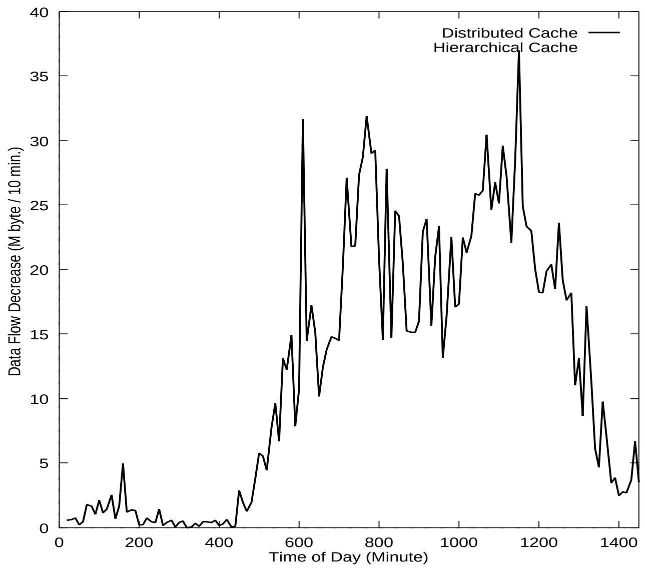


Fig. 17. Network traffic distribution over the time of day for different cache systems

8 Related Work

Intellectual assistance by computers has attracted many people, and various attempts have been undertaken with different approaches and for different tasks. There are many terms that characterize these approaches such as learning apprentice, software agent, learning agent, interface agent, programming by example or demonstration, personal knowledge based system, etc. What is common to many of them is that they observe repetition or regularity in the user's behavior and use them for automation, prediction and customization in one way or another.

The amount of knowledge that has to be provided in advance varies among the approaches. General remarks are that making the user program everything requires too much insight, understanding and effort from the user, and having to encode a lot of domain-specific background knowledge about the task and the user also requires a huge amount of work from the knowledge engineer. Both have fixed competence, and are hard to customize to individual user differences or changes of habits. Some sort of automatic knowledge acquisition that can capture each user's habits is needed.

EAGER [2] is an example of program by demonstration (PBD), which is a HyperText system that keeps watching a user's actions, detects an iteration and offers to run the iterative procedure to completion by generalizing the repetitions and making macros. Myers's demonstrational formatter [15] is also an example of PBD. It does not focus on the repetition, but generalizes a single example to create a template for later use, which enables the formatting of headers, itemized lists, tables, references, etc. Another example is Gold [16] which is a business chart editor. It is given the knowledge of properties of the data and the typical graphics in business charts to generalize a single, or a very few examples, by interpreting them as a combination of primitives.

[6] analyzes repetitive patterns in the UNIX command histories and observes some regularities. [13] also uses the repetitive nature for a predictive user interface. When a user types a repeat key after doing repetitive operations, an editing sequence corresponding to one iteration is detected, defined as a macro, and executed at the same time. Although being simple, it covers a wide range which had to formerly be covered by keyboard macro. [8,3] explores mechanisms for predicting the next command to be used for the UNIX command-line shell. To our knowledge their work is the closest to ours. They have collected command histories from 77 people, and have calculated the predictive accuracy over this dataset using $C4.5$. They use only sequence information and the best performance they obtained has an average online predictive accuracy of up to 38%, which is consistent to our result in Table 3. They have built a new shell called *ilash* by adding this predictive capability to *tcsh*. They argue that because many users use aliases which reduce the average command length, the saving of the keystrokes typed is not much even if a correct prediction could be inserted with a single character.

All of the above approaches except [8,3] do not use machine learning techniques although they do guess and generalize. The Interface agent of [12] takes a machine learning approach. They address the problem of self-customizing software at a much more task independent level. The core is to learn by observing the user, *i.e.*, by find reguralities in the user's behavior and using them for prediction. They also adapt two other learning modes: learning from user feedback and learning by being told. They used memory-based learning (k-nearest neighbor) which is good for explanation. Situations in the user are described in terms of a set of attributes which are hand-coded. The tasks that they applied are a calendar management agent and an electronic mail clerk.

The personal learning apprentice CAP [4] is similar to the above. It is an interactive assistance that learns continually from the user to predict default values. Their application is a calendar management apprentice which learns preferences as a knowledgeable secretary might do. Two competing leaning methods are used: decision tree learning and backpropagation neural net. The attribute value representation suffices for this purpose. Another related system addresses the task of form-filling [7]. They use decision tree learning to predict default values for each field on the form by referring to values observed on other fields and the previous form copy.

[23]'s pen-based interactive note taking system is a self-customizing software to eliminate the need for user customization. It starts with partially-specified software and applies a machine learning technique to complete any remaining customization. The system learns a finite state machine to characterize the syntax of user's notes and learns decision tree to generate predictions. Letizia [11] is an interface agent that assists a user browsing the WWW. It tracks user behavior and attempts to anticipate items of interest by doing concurrent, autonomous exploration of links from the user's current positions. Intelligent agent for information browsing is a hot area and many systems are being pursued (*e.g.*, [5,18]).

The research on prefetching is carried out by a separate community. The standard Least Recently Used (LRU) based caching offers some assistance, but ignoring any relationships that exist between file system events fails to make full use of available information. The closest work that uses the relationship would be [10]. They use trie structure to memorize previous I/O sequence but no explicit learning is performed. Their results indicate that the predictive caching gains on the average 15% more cache hits than the LRU based caching. However, since they are using only sequential information, their method does not work well in a multi-task environment.

All of the applications that use machine learning techniques do not require relational representations. The data are represented by a set of features. Analysis of sequential information is enough for the selected applications. Some require additional task specific knowledge. We showed in this paper that there are other applications that this success cannot be easily generalized, and proposed the *GBI* as a general induction mechanism for this type of applications.

9 Conclusion

We have modeled a user adaptive interface that can predict next command, generate scripts and prefetch files in a multi-task environment. The analysis of behavioral data indicated that the directly observable sequential records are not enough to capture the behavior, and that simultaneous use of process I/O information that is hidden from the user is beneficial. An efficient induction algorithm that can handle relational data was needed and a technique called graph-based induction was applied. It can find frequently occurring patterns from a graph representation. It also induces classification rules from structured data that have intra-relationship. Pairwise chunking, which is the heart of the algorithm, does not guarantee an optimal solution by any means, but empirical study shows that use of statistical measure results in a good solution. It is efficient and can run in real time. The command prediction module is in daily use. Shell script generation works as expected but is less used. Prefetching daemon still needs a better implementation to enjoy the real benefit.

Acknowledgments

Much of the work was conducted while the first author was at ARL, Hitachi, Ltd. Authors are grateful to Shojiro Asai and Katsuki Miyauchi of ARL for their generous support of this work. They extend their thanks to Takashi Washio, Tadashi Horiuchi and Toshihiro Kayama of Osaka University for the discussions.

References

[1] L. J. Breiman, H. Friedman, R. A. Olshen, and C. J. Stone. *Classification and Regression Trees.* Wadsworth & Brooks/Cole Advanced Books & Software, 1984.

[2] A. Cypher. Eager: Programming Repetitive Tasks by Example. In *Proc. of CHI'91*, pages 33–39, 1991.

[3] B. D. Davison and H. Hirsh. Toward An Adaptive Command Line Interface. In *Proc. of HCI'97 (Advances in Human Factors/Ergonomics Volume 21B - Design of Computer Systems: Social and Ergonomic Considerations*, pages 505–508, 1997.

[4] L. Dent, J. Boticario, J. McDermott, T. Mitchell, and D. Zabowski. A Personal Learning Apprentice. In *Proc. of AAAI'92*, pages 96–101, 1992.

[5] O. Etzioni94 and D. Weld. A Softbot-Based Interface to the Internet. *Commun. ACM*, 37(7):72–76, 1994.

[6] S. Greenberg and I. H. Witten. How Users Repeat Their Actions on Computers: Principles for Design of HISTORY Mechanisms. In *Proc. of CHI'88*, pages 171–178, 1988.

[7] L. A. Hermens and J.C. Schlimmer. A Machine-learning Apprentice for the Completion of Repetitive Forms. In *Proc. of the Ninth Conf. on Artificial Intelligence for Applications*, pages 164–170, 1993.

[8] H. Hirsh and B. D. Davison. An Adaptive unix Command-Line Assistant. In *Proc. of the First International Conference on Autonomous Agents*, pages 542–543, 1997.

[9] M. James. *Classification Algorithms.* A Wiley-Interscience Publication, 1984.

[10] T. M. Kroeger and D. D. E. Long. Predicting File System Actions from Prior Events. In *Proc. of CHI'94*, pages 319–328, 1994.

[11] H. Lieberman. Letizia: An Agent That Assists Web Browsing. In *Proc. of IJCAI'95*, pages 924–929, 1995.

[12] P. Maes and R. Kozierok. Learning Interface Agents. In *Proc. of AAAI'93*, pages 459–465, 1993.

[13] T. Masui and K. Nakayama. Repeat and Predict - Two Keys to Efficient Text Editing. In *Proc. of CHI'94*, pages 118–123, 1994.

[14] S. Muggleton and C. Feng. Efficient Induction of Logic Programs. In S. Muggleton, editor, *Inductive Logic Programming*, pages 281–298. Academic Press, 1992.

[15] B.A. Myers. Text Formatting by Demonstration. In *Proc. of CHI'91*, pages 251–256, 1991.

[16] B.A. Myers, J. Goldstein, and M. A. Goldberg. Creating Charts by Demonstration. In *Proc. of CHI'94*, pages 106–111, 1994.

[17] M. Pazzani and D. Kibler. The Utility of Knowledge in Inductive Learning. *Machine Learning*, 9(1):57–94, 1992.

[18] M. Perkowits and O. Etzioni. Category Translation: Learning to Understand Information on the Internet. In *Proc. of IJCAI'95*, pages 930–936, 1995.

[19] P. P. Piernot. The AIDE Project: An Application-Independent Demonstrational Environment. In A. Cypher, editor, *Watch What I do: Programming By Demonstration*, pages 387–405. MIT Press, 1993.

[20] J. R. Quinlan. Induction of Decision Trees. *Machine Learning*, 1:81–106, 1986.

[21] J. R. Quinlan. Learning Logical Definitions from Relations. *Machine Learning*, 5(3):239–266, 1990.

[22] J. R. Quinlan. *C4.5: Programs for Machine Learning.* Morgan Kaufmann, 1993.

[23] J.C. Schlimmer and L. A. Hermens. Software agents: Completing Patterns and Constructing User Interfaces. *Artificial Intelligence Research*, 1:61–89, 1993.

[24] K. Yoshida. WWW Cache Layout to Ease Network Overload. In *Proc. of Sixth International Workshop on Artificial Intelligence and Statistics, AISTATS97*, pages 537–548, 1997.

[25] K. Yoshida and H. Motoda. Clip: Concept Learning from Inference Pattern. *J. of Artificial Intelligence*, 75(1):63–92, 1995.

[26] K. Yoshida and H. Motoda. Automated User Modeling for Intelligent Interface. *Int. J. of Human Computer Interaction*, 8(3):237–258, 1996.

[27] K. Yoshida, H. Motoda, and N. Indurkhya. Graph-based Induction as a Unified Learning Framework. *J. of Applied Intelligence*, 4:297–328, 1994.