Int. J. Man–Machine Studies (1993) **39**, 353–390

Supporting command reuse: empirical foundations and principles

SAUL GREENBERG

Department of Computer Science, The University of Calgary, Calgary, Alberta T2N 1N4, Canada

IAN H. WITTEN

Department of Computer Science, University of Waikato, Private Bag 3105, Hamilton, New Zealand

(Received 18 December 1989 and accepted in revised form 1 February 1993)

Current user interfaces fail to support some work habits that people naturally adopt when interacting with general-purpose computer environments. In particular, users frequently and persistently repeat their activities (e.g. command line entries, menu selections, navigating paths), but computers do little to help them to review and re-execute earlier ones. At most, systems provide ad hoc *history mechanisms* founded on the premise that the last few inputs form a reasonable selection of candidates for reuse.

This paper provides theoretical and empirical foundations for the design of a reuse facility that helps people to recall, modify and re-submit their previous activities to computers. It abstracts several striking characteristics of repetitious behaviour by studying traces of user activities. It presents a general model of interaction called "recurrent systems". Particular attention is paid to the repetition of command lines given a sequential history list of previous ones, and this distribution can be conditioned in several ways to enhance predictive power. Reformulated as empirically-based general principles, the model provides design guidelines for history systems specifically and modern user interfaces generally.

1. Introduction

There is much repetition in computer use. Yet most interfaces offer little help for reviewing and reusing previous submissions. As a result, users often find themselves retyping command lines or reselecting menu items that they had entered previously. To relieve the tedium, some systems incorporate a *history mechanism* that allows a user to recall old submissions. These are invariably based on the premise that recently entered submissions are a reasonable working set of candidates to keep available to the user for reselection. But is this premise correct? Might other strategies work better? Indeed, is the user–computer dialogue sufficiently repetitive to warrant some type of reuse facility in the first place? As existing history mechanisms were designed by intuition rather than from empirical knowledge of human–computer interactions, it is difficult to judge how effective they really are and what scope there is for improvement.

This paper assesses the extent to which people reuse their previous activities. Although the idea of reuse is simple—anything used before may be used again—it is only effective when recalling old activities is less work for the user (cognitively and physically) than submitting new ones. Consider a user who has submitted n

0020-7373/93/090353 + 38$08.00/0

activities to the system over time (say $n > 100$) and whose next activity is identical to some previous one. An optimal reuse facility would be an oracle that correctly predicted when an old action would be reused, and then selected the correct one and submitted it to the system in the user's stead. In contrast, a non-predictive system that merely presents the user with all previous n submissions would be less effective, for the user's overhead now includes scanning (or remembering) the complete interaction history and selecting the desired action. Real systems are situated between these extremes. A small set of reasonable predictions p is offered to the user ($p \ll n$), sometimes ranked by probability. The intention is to make the act of selecting a prediction less work than entering it anew; the metric for "work" is, of course, ill-defined.

Schemes for activity reuse are based upon the assumption that the human–computer dialogue contains repetition. Yet very little critical attention has been given to this assumption. Section 2 introduces a model of human–computer dialogue called *recurrent systems*, where most users predominantly repeat their previous activities. Such systems suggest potential for activity reuse because there is opportunity to give preferential treatment to the large number of repeated actions. A few suspected recurrent systems from non-computer domains are summarized in this context to help pinpoint salient features.

The remainder of this paper investigates recurrences exhibited by 168 subjects using the UNIX command interpreter *csh*. Section 3 describes the method of data collection and introduces some terminology, while Section 4 reviews previous studies on how people enter UNIX *commands* (as opposed to complete *command lines*). However, we believe that studies of commands have only limited utility. As commands often act on objects and are qualified with options, it is important to look at the command line as a whole.

Next, three questions particularly relevant to reuse facilities are addressed, all concerning the statistics of complete command lines entered by the user to UNIX. First, how often do users repeat themselves? Section 5 details how often a UNIX user actually repeats command lines over the course of a dialogue. Particular attention is paid to the variation in this rate between groups and between individuals, and its stability over the number of command lines entered. Second, how well does recency behave as a predictor of future events? Section 6 gives the probability distribution that the next command line will match a previous input, measured as a function of the number of entries that have elapsed from the matched input to the current one. Third, given that people repeat themselves, what is the best way to predict (and offer for reuse) what will be done next? Section 7 examines several predictive schemes for reuse through an empirical study that derives the probability distribution of the next activity given a list of previous ones.

The paper closes by reformulating the study's findings as empirically-based general principles that govern how users repeat their activities on computers. These provide a basis for design guidelines for history mechanisms specifically and modern user interfaces generally.

2. Recurrent systems

Reuse facilities presuppose that people repeat their activities. But do they do so enough to repay the overhead of learning and using a reuse facility? And how are

these activities repeated? Are patterns of repetitions arbitrary or system-specific, implying that reuse facilities must be customized to be worthwhile? Or can general patterns be found in most dialogues, implying the possibility of generic reuse facilities? This section provides evidence for the latter view by defining a model of human–computer dialogue called "recurrent systems".

An *activity* is defined as the command formulated by the user and submitted to the system. Its execution is expected to gratify the user's immediate intention. Activities are the units entered to *incremental interaction,* a human–computer dialogue characterized by successive requests that are submitted to the computer and responded to in turn (Thimbleby, 1990). Entering command lines, querying databases, choosing items from a palette, and locating and selecting items in a menu hierarchy are some examples. Copy typing is not, as it is continuous rather than incremental, and is not a cognitive activity (at least, not for the skilled typist).

A *recurrent system* is defined as an open-ended system where users predominantly repeat activities they have previously submitted to the computer. In other words, although many activities are possible, most (but not all) of a user's entries are repetitions of their previous ones.

The frequency of repeats, called the *recurrence rate,* is the probability that any activity is a repeat of a previous one. Let *total activities* be the number of all submissions a user has entered, and the *vocabulary size* be the count of different submissions in the set. The recurrence rate R over a set of user activities is calculated as:

$$R = \frac{\text{total activities} - \text{vocabulary size}}{\text{total activities}} \times 100\%.$$

Although many old activities are repeated, new ones are constantly added to the repertoire. The rate at which new activities are composed and introduced to the dialogue is the *composition rate* C, simply expressed as $C = 100 - R$. As there are a very large number of possible activities available, new activity formation within recurrent systems is open-ended. Even when new activities are continually generated, it is expected that these will remain a small subset of the activities that could be formed by any one user.

We have performed several empirical studies of recurrent systems. These included information retrieval from manuals, telephone dialling patterns, UNIX command and command line use (reported in this paper), and the way people use a functional programming language (Greenberg, 1993). From these, we abstract typical characteristics of how individuals interact with such systems:

1. Many activities are repeated, with the recurrence rate R ranging between 40–85%; the exact value depends on the application domain and the individual's usage patterns. That is, there is a probability of 40–85% that the next activity the user submits to the system is an exact duplicate of a previous submission.
2. Recurrent systems incorporate new activities regularly, where the composition rate C falls between 15–60%. That is, there is a probability of 15–60% that the next activity submitted by the user is novel.

3. The set of activities invoked by any particular user is typically a small subset of the total activities usually available.
4. Although the overall recurrence rate remains more or less constant over time, the frequency at which particular items are repeated over the course of the interaction waxes and wanes.
5. The probability of an activity recurring increases (although not linearly) with its recency of selection. That is, recently-submitted items are more likely to be repeated than those last submitted a long time ago.
6. While there is some overlap, the sets of activities invoked by different users of the system are mostly disjoint.
7. Different people may repeat the few activities in common at quite different rates.

This definition and list of properties is not a strong one, for the boundary between recurrent and non-recurrent systems is not well-defined. Such a boundary specification, even a "fuzzy" one, would be subjective and would also depend upon other aspects of the system being investigated. For example, time between recurrences might be a consideration, where short-term recurrences are counted but those repeated only after long intervals are discarded. Still, the properties provide a reasonable checklist for judging whether particular systems have potential for reuse.

It may seem that, at least on the surface, recurrent systems are just another way of denoting patterns of behaviour already well described by Zipf's law (Zipf, 1949). However, major differences exist:

- Many human-oriented observations characterized by Zipf's law are based upon data pooled over the entire population. One study, for example, examined the statistics of all terms used to retrieve items over all users of two separate bibliographic data bases, and describes how they conform to Zipf's law (Bennett, 1975). Similar large-scale statistics have been applied to many facets of library science; a list is provided by Peachey, Bunt and Colbourn (1982). Yet there is no evidence that the same distribution applies to individuals. Recurrent systems, on the other hand, are centred around the statistics of activities of individuals, rather than the pooled statistics of large groups.
- Zipf's law typically deals with very large numbers, and tends to break down with few observations [see Bennett (1975) for one example]. Recurrent systems do not break down, and the analysis below shows that patterns within recurrent systems are apparent within small slices of sequential activities entered by a single individual.
- As Zipf's law describes a frequency distribution, it does not account well for items that are heavily used in a short-term interaction but rarely used afterwards, or ones whose frequencies fluctuate over time [(4) above]. Recurrent systems handle this well since they emphasize recency as well as frequency of use [(5) above].

3. Data collection methodology

The remainder of this paper presents our empirical study and discussion of UNIX as a recurrent system. This section introduces some terminology and describes the way data was collected in our UNIX study.

3.1. DEFINITIONS

A *command line* is a single complete line (up to a terminating carriage return) entered by the user. This is a natural unit to consider as an "activity" because commands are only interpreted by UNIX *csh* when the return key is typed, and the complete line is a more detailed reflection of one's intentions than just the command itself. Command lines typically comprise an *action* (the command), an *object* (e.g. files, strings) and *modifiers* (options that modify the behaviour of the command). The *command* is the verb of the command line.

A *history list* is a sequential record of command lines entered by a user over time, ignoring boundaries between login sessions. Unless stated otherwise, the history list is a true sequential record of every single syntactically correct command line typed (erroneous submissions noticed by *csh* are not included). Duplicate activities, for example, are included.

The *distance* between two command lines is the difference between their positions on the history list. A *working set* is a small subset of items on the history list. The different entries in the history list (i.e. with duplicate entries removed) make up the command-line *vocabulary*. All white space separating words in a command line are considered equal, so two lines that differ only in their use of white space are equivalent. However, syntactically different but semantically identical command lines are considered distinct.†

3.2. DATA COLLECTION

Command-line data was collected from users of the UNIX *csh* command interpreter (Joy, 1980). The selection and grouping of subjects, and the method of data collection, are as follows.

Subjects

The subjects were 168 unpaid volunteers, all university students or employees. Four target groups were identified, representing a total of 168 male and female users with a wide cross-section of computer experience and needs. Salient features of each group are described below, while the sample sizes are indicated in Table 1.

- *Novice Programmers.* Conscripted from an introductory Pascal course, these had little or no previous exposure to programming, operating systems, or UNIX-like command-based interfaces. They spent most of their computer time learning how to program and to use the basic system facilities.
- *Experienced Programmers.* Members were senior computer science undergraduates, expected to have a fair knowledge of programming languages and the UNIX environment. As well as coding, word processing, and employing more advanced UNIX facilities to fulfill course requirements, these subjects also used the system for social and exploratory purposes.
- *Computer Scientists.* This group, comprised of faculty, graduates and researchers from the Department of Computer Science, had varying experience

† For example, the command lines *ls-las* and *ls-lsa* are treated as different vocabulary items, even though they mean the same thing. Although this strategy overestimates the vocabulary size, a semantic analysis was deemed too expensive for the large data set covered.

TABLE 1
Sample group sizes and statistics of the command lines recorded

Name	Sample size	Total number of command lines	Number of command lines excluding errors		
			total	mean	S.D.
Novice programmers	55	77,423	73,288	1333	819·8
Experienced programmers	36	74,906	70,234	1950	1276·0
Computer scientists	52	125,691	119,557	2299	2022·9
Non-programmers	25	25,608	24,657	986	1155·6
Total	168	303,628	287,736	1712	1498·8

with UNIX, although all were experts with computers in general. Tasks performed were less predictable and more varied than other groups, spanning advanced program development, research investigations, social communication, maintaining databases, word processing, satisfying personal requirements, and so on.

- *Non-programmers.* Word processing and document preparation was the dominant activity of this group, made up of office staff and members of the Faculty of Environmental Design. Little program development occurred—tasks were usually performed with existing application packages. Their knowledge of UNIX was usually the minimum necessary to get the job done.

Since users were assigned to subject groups only through their membership in identifiable user groups (e.g. Computer Science graduate students), their placement in the categories above cannot be considered strictly rigorous. Although it is assumed that they generally follow their group stereotype, uniform behaviour is not expected.

Instructions to subjects

As part of the solicitation process, subjects were informed verbally or by letter that:

- data on their normal UNIX use would be monitored and collected at the command-line level only;
- the data collected would be kept confidential through use of anonymous reference;
- at any time during the study period the subject could request that data collection stop immediately;
- there would be no noticeable degrading of system performance;
- if requested, data collected from a subject would be made available to him or her.

Subjects did not require nor did they receive any additional instructions during the actual study period. No subject asked to be withdrawn from the experiment, and no-one asked to see their personal data.

Apparatus

A modified version of the Berkeley 4.2 UNIX *csh* command line interpreter was installed on three VAX 11/780's located in the Department of Computer Science and one VAX 11/750 in the Faculty of Environmental Design, both within the University of Calgary. Many different terminals were available to participants, most of which were traditional character-based VDUs. In addition, Corvus Concept workstations running the Jade Window Manager were available to members of the Experienced and Computer Scientist groups. As with most window systems, these allowed users to create many terminal windows on their display, each running *csh* (Greenberg, Peterson & Witten, 1986).

Method

Command-line data was collected continuously for four months from users of the modified *csh* command-line interpreter mentioned above. From the user's point of view, monitoring was unobtrusive—the modified interpreter was identical in all visible respects to the standard version. The total numbers of command lines recorded per group are listed in Table 1.

Data was collected by recording lines expanded by *csh*. Instead of catching keystrokes as they are entered, the complete line submitted was captured as a chunk after it had been entered and processed by *csh*. Editing operations that helped form the line were ignored (e.g. backspace and corrected keystrokes). Extra information known to *csh* was trapped and recorded as well by placing "hooks" within *csh*. History and alias use (both features of *csh*) were noted, as well as the current working directory of the user and the error status after execution was attempted. More specifically, login sessions were distinguished by a record that noted the start and end time of each session. Records associated with each user input were then listed subsequently, each annotated with the command line, the current working directory, alias substitution (if any), history use and error status. The total number of command lines collected over a group (excluding errors) and the mean number collected per individual are listed in Table 1.

Data selection

If subjects did not log in at least 10 times and execute at least 100 commands during the study period, their data was not considered. By these criteria, 12 of the original 180 participants were rejected.

Motivation

Participants used UNIX as usual. Users were neither encouraged nor expected to alter their everyday use of the system. As subjects had few reminders that their command-line interactions were being traced, they were largely oblivious to the monitoring process.

Availability and confidentiality of data

All data collected is available to other researchers. A research report describes its format, and includes a tape of the data (Greenberg, 1988). As all subjects were promised confidentiality, data has been massaged to remove the identity of subjects.

Limitations

Tracing lines expanded by *csh* is a tradeoff between recording too much and too little information. Several limitations of our data collection method are noted below.

First, due to implementation difficulties, the details of *csh*'s history invocations were not recorded. The altered *csh* indicates only that history has been used, and notes the command line retrieved through history. It does not record the actual history instructions used to produce the modification.

Second, user activity outside of *csh* is not captured. Although recording *csh* lines works well for batch-style programs that execute and return without user intervention (i.e. incremental interaction), it does not capture activity within the interactive applications used (e.g. editors). Interactive information is lost since collected data captures the *csh* command line only.

Third, the actual processes spawned by the command line are not noted. There are many ways to execute programs in UNIX; directly by name, indirectly through an alias or *csh* variable, or as a suite of programs through a script. Because of this diversity, users can invoke the same program by many different names. For example, *e, emacs* and *ed* may all invoke the same editor. As only the text typed to *csh* is collected, the actual processes executed are left as an educated guess. Still, the method employed here seems a reasonable approach, especially when contrasted to other methods employed by researchers to collect data on UNIX use [see Chapter 2 in Greenberg (1993) for a comparison of data collection methods].

4. Related work: recurrence of UNIX commands

Several independent researchers have studied how individual commands in UNIX—the verbs of the command lines—are selected. Statistics have been presented in a variety of ways: as single results pooled over the population; as comparisons between groups within a population; and as individual patterns of use. Variables investigated have been the frequency distribution of commands; the overlap of command sets between groups and individuals; the growth of command vocabulary over time; and the interactions between sequences of commands. The significant results of each approach are summarized here. We will argue that the statistics tell us more of differences between people than their similarities.

FREQUENCY DISTRIBUTION OF COMMANDS FOR LARGE GROUPS

Many investigators have examined the frequency of command usage by a user population, where all individuals are pooled into a single population statistic (Peachey, Bunt & Colbourn, 1982; Hanson, Kraut & Farber, 1984; Ellis & Hitchcock, 1986; Greenberg & Witten, 1988*a*). All studies report results approximated by a Zipf distribution, which has the property that a relatively small number of items have high usage frequencies, and a very large number of items have low usage frequencies (Witten, Cleary & Greenberg, 1984; Zipf, 1949). A looser characteristic of this kind of rank distribution is the well-known 80–20 rule of thumb that has been commonly observed in commercial transaction systems—20% of the items in question are used 80% of the time (Knuth, 1973; Peachey, Bunt & Colbourn, 1982). Hanson, Kraut and Farber (1984), for example, state that 10% of the 400–500

commands available account for 90% of the usage. Greenberg and Witten (1988*a*) report a similar finding of 10% of the commands accounting for 84–91% of all activity.

USAGE FREQUENCY OF PARTICULAR COMMANDS BETWEEN GROUPS

The Zipf distribution of pooled command use over the complete sample can be misleading. Greenberg and Witten (1988*a*) contrasted command use between the four groups identified in Section 3, and data from Hanson's similar study (Hanson, Kraut & Farber, 1984). They found that commands do not necessarily retain their same rank order between different user groups. That is, although the shape of each group's distributions are similar, particular commands may appear at quite different locations between them (if at all). Although there are several frequently used commands in common for all groups,† the comparison emphasized the group's differences in their choice of system utilities, such as compilers, editors, and task-specific items.

FREQUENCY DISTRIBUTION AND COMMAND OVERLAP BETWEEN INDIVIDUALS

These differences between groups are a reflection of the tremendous differences between individuals within a group. Draper (1984) found that very little of each individual's command vocabulary was used by all the population, a little more was shared to some degree by other users, and the rest was used by the individual user alone.‡ Greenberg and Witten (1988*a*) pursued this matter further. They found that only 0·2% (i.e. 3) of the 1307 different commands entered to the system were shared by more than 90% of the users. More surprisingly, fully 92% of all shared commands invoked were used by fewer than 10% of the users, and 68·8% of all commands were not shared at all. Even users with apparently similar task requirements and expertise had astonishingly little vocabulary overlap.

SIZE AND GROWTH OF AN INDIVIDUAL'S COMMAND VOCABULARY

Sutcliffe and Old (1987) suggested that the size of a user's command set (the command vocabulary) grows with their usage of the system. They found a significant correlation between the overall command use by the user and the number of unique commands employed. Greenberg and Witten (1988*a*) actually observed vocabulary acquisition by particular users and noted that the vocabulary growth rate was quite slow: $C = 1\%$ or less ($R = 99\%$). Vocabulary growth was by no means regular—long periods of quiescence were followed by flurries of activity. Although a total of 1307 different commands were invoked on the system over the period of study, each person used, on average, only 50 different commands (but the standard deviation was 32·5).

RELATIONS IN COMMAND SEQUENCES

The previous discussion says nothing about possible relations and dependencies between commands. Through a multivariate analysis of UNIX commands invoked by the site population, Hanson, Kraut and Farber (1984) examined the interaction

† The shared commands mostly concerned the basic UNIX commands for navigating, manipulating and finding information about the file store.

‡ Commands in his study were the processes actually executed.

effects between commands. They noted that some commands follow or congregate around others in quite regular ways. However, the dependencies and clustering observed may be an artefact of pooled statistics resulting from, say, a small handful of people using a set of related commands frequently. Sutcliffe and Old (1987) replicated and extended Hanson, Kraut and Farber's work by eliminating all dependencies but those that were significant for at least five or more individual users. The resulting network was a fragmented subset of the population network. They concluded that only a small number of command chains were used in common tasks.

DISCUSSION

In spite of the high value of R, command use is not, strictly speaking, a recurrent system, for new activities are not incorporated regularly (C is only 1%). This summary of the command analyses tells us more about individual differences between users than their similarities, and the results do not suggest any general new directions in interface design.

Perhaps undue attention has been paid to command usage. Commands, after all, are only the verbs of the command line. They also act on objects, are qualified with options, and may redirect input and output to other commands. These other facets are surely important and should not be ignored. For example, UNIX lines sharing the same initial command may have completely different meanings. Consider the two command lines below. The first just sorts a file, while the second produces a frequency count of the identical lines in the file:

sort file
sort file/uniq -c/sort -r.

Another problem is that the same command line may satisfy rather different intentions. One person might invoke the UNIX command line *ls -l* to distinguish between ordinary files and directories, whereas another could use the same sequence to discover file creation dates and sizes (Ross, Jones & Millington, 1985). Still, a reasonable approach to meaningful analysis of UNIX as a recurrent system is to consider the complete command line entered. Accordingly the UNIX usage data, analysed here in terms of commands, is re-analysed in the following sections in terms of command lines.

5. Recurrences of command lines

As mentioned above, command use is not really a recurrent system since C is so low. A different question is whether complete command lines submitted to command-based environments follow the properties of recurrent systems. If they do, what patterns do these recurrences exhibit?

Although only a few commands account for all actions of a particular user, it is not known how often new command lines are formed and old ones recur. This is important, as it is the recurrence rate—the probability that the next item has been entered previously—that existing history systems exploit best. One might expect command lines to recur infrequently, given the limitless possibilities and combinations of commands, modifiers, and arguments. Surprisingly, this is not the case.

We investigated how often lines are repeated by counting the command line vocabulary size. Let $t_{cmd\ lines}$ be the total number of command lines (activities) entered by the user (i.e. the size of the history list), and $v_{cmd\ lines}$ be the vocabulary size, or number of distinct items in that set. The overall recurrence rate and composition rate can then be calculated as described in Section 2. But first, we have to see how "stable" R is over a lengthy transaction record.

Do users extend their vocabularies continuously and uniformly over the duration of an interaction? If not, then the recurrence rate, measured locally, will change over time as the user's history list grows. Furthermore, calculating group means for R could be confounded by the large variation between the number of command lines each user enters, which was noted in Table 1. As R is a function of $v_{cmd\ lines}$ and $t_{cmd\ lines}$ it is necessary to investigate how the vocabulary size depends upon the actual number of commands entered. If users never extend their vocabulary after some short initialization period, little correlation with $t_{cmd\ lines}$ is expected. On the other hand, a strong correlation is likely if new command lines are composed regularly by a user.

A simple regression analysis was performed by contrasting $t_{cmd\ lines}$ and $v_{cmd\ lines}$ for each subject. The regression line is plotted in Figure 1, where each point in the scattergram represents the value observed for each subject at the end of the study period. A statistically significant and strong correlation was found ($r = 0{\cdot}918$, $df = 167$, $p < 0{\cdot}01$). The moderate slope ($C = 23\%$) of the regression line makes the correlation practically significant as well.

It seems reasonable from the scattergram of Figure 1 that $v_{cmd\ lines}$ increases linearly with $t_{cmd\ lines}$, indicating that the recurrence rate is independent of the actual number of lines entered. This was checked in two ways. The first was a simple regression analysis of $t_{cmd\ lines}$ with R, where each point represents the recurrence rate observed for each subject at the end of the study. A statistically significant correlation was found ($r = 0{\cdot}253$, $df = 167$, $p < 0{\cdot}01$), indicating that the recurrence

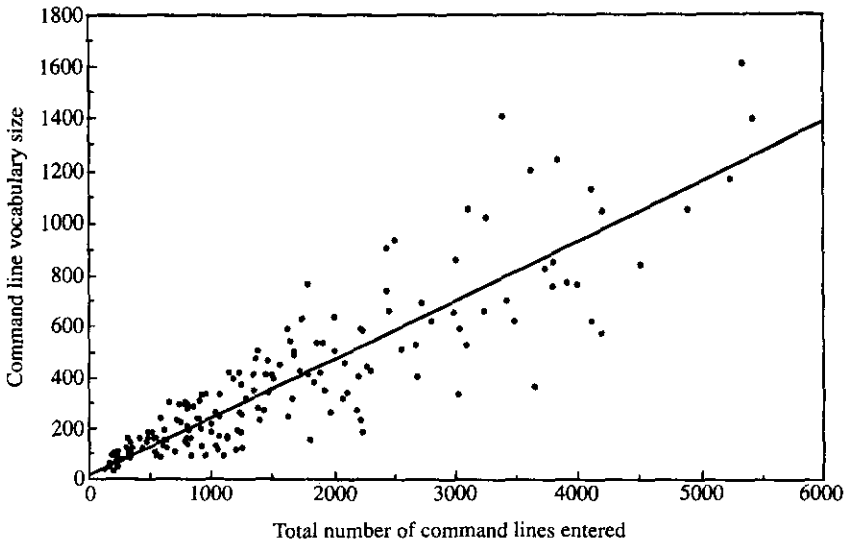

FIGURE 1. Regression command line vocabulary size vs. the total command lines entered by each subject. Slope = 0·23.

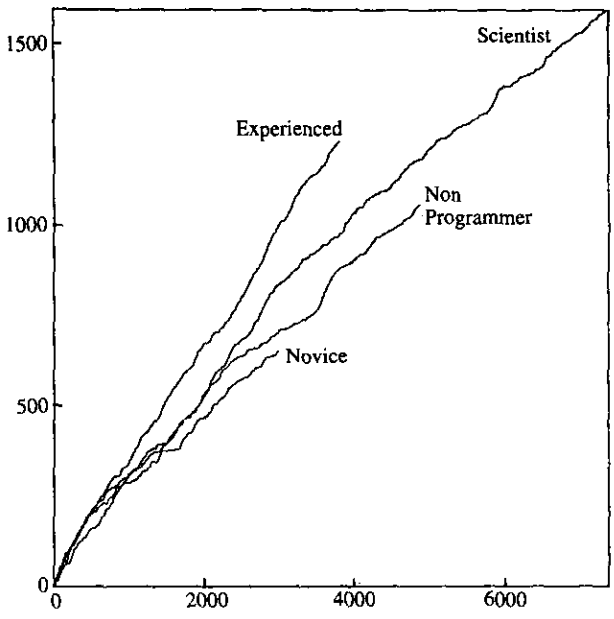

FIGURE 2. Command line vocabulary size vs. the number of commands entered for four typical individuals.

rate increases with the number of commands entered. However, the high variance of data points around the line ($r^2 = 0{\cdot}064$), and its low slope (0·002), make this finding insignificant for practical purposes. Consequently, R is considered independent of $t_{cmd\ lines}$.

The second and perhaps more convincing way of observing the independence of the recurrence rate is by examining in detail the vocabulary growth of particular individuals as opposed to the group statistics. The formation of new command lines is surprisingly linear and regular, as illustrated by Figure 2, which shows the command line vocabulary growth for four typical users, one from each group. The horizontal axis represents the number of lines entered so far, while the vertical axis indicates the size of the command line vocabulary. For example, the scientist subject has composed close to 1400 new command lines after 6000 lines were entered.

Table 2 lists the mean recurrence rate, standard deviation, and ranges of R for each subject group. An analysis of variance of raw scores rejects the null hypothesis that these means are equal [$F(3, 164) = 21{\cdot}42$, $p < 0{\cdot}01$]. The Fisher PLSD multiple comparison tests suggest that all differences between group means are significant ($p < 0{\cdot}01$), excepting the Non-programmers vs. Scientists. As the Table indicates, the mean recurrence rate for groups ranges between 68% and 80%, with Novice Programmers exhibiting the highest scores.

Although recurrence rate depends upon user category, and very slightly on the number of command lines entered, it is reasonable to simplify this descriptive statistic by assuming the mean R over all users to be 75% and C to be 25%, independent of $t_{cmd\ lines}$. In other words, an average of three out of every four command lines entered by the user already exist on the history list. Conversely, an average of one out of every four command lines appears for the first time.

TABLE 2
The average recurrence rate of the four sample Unix user groups

	Recurrence rate		Range	
Sample name	mean	S.D.	minimum	maximum
Novice programmers	80·4%	7·2	64·7%	91·7%
Experienced programmers	74·4%	9·7	51·4%	90·0%
Computer scientists	67·7%	8·2	46·4%	82·0%
Non-programmers	69·4%	8·1	50·0%	84·3%
Total	73·8%	9·6	46·4%	91·7%

6. Command line frequency as a function of distance

For any command line entered by a user, the probability that it has been entered previously is quite high. But how do previous items contribute to this probability? Do all items on the history list have a uniform probability of recurring, or do the most recently entered submissions skew the distribution? If a graphical history mechanism displayed the previous p entries as a list (e.g. HISTMENU, Bobrow, 1986), what is the probability that this list includes the next entry?

The recurrence distribution as a measure of distance was calculated for each user. First, let $R_{s,d}$ be the recurrence rate at a given distance for a single person, obtained by processing each subject's data. Figure 3 shows the algorithm used to obtain all

Given:

- a trace numbered from 1 through n, where n is the last line entered;
- an array of counters used to accumulate the number of recurrences at a particular distance.

Algorithm:

```
/* For each item, find its nearest match on the history list */
/* and record it */
for (i := 1 to n)
        for (j := i-1 downto 1)
                if (submission_i = submission_j) then begin
                        distance := i-j;
                        counter[distance] := counter[distance] + 1;
                        break; /* jump out of inner loop */
                end
/* The averaged value found in each counter is R_s,d */
for (distance := 1 to n)
        counter[distance] := (counter[distance]/n) * 100;
```

FIGURE 3. Processing a subject's trace for all values of $R_{s,d}$.

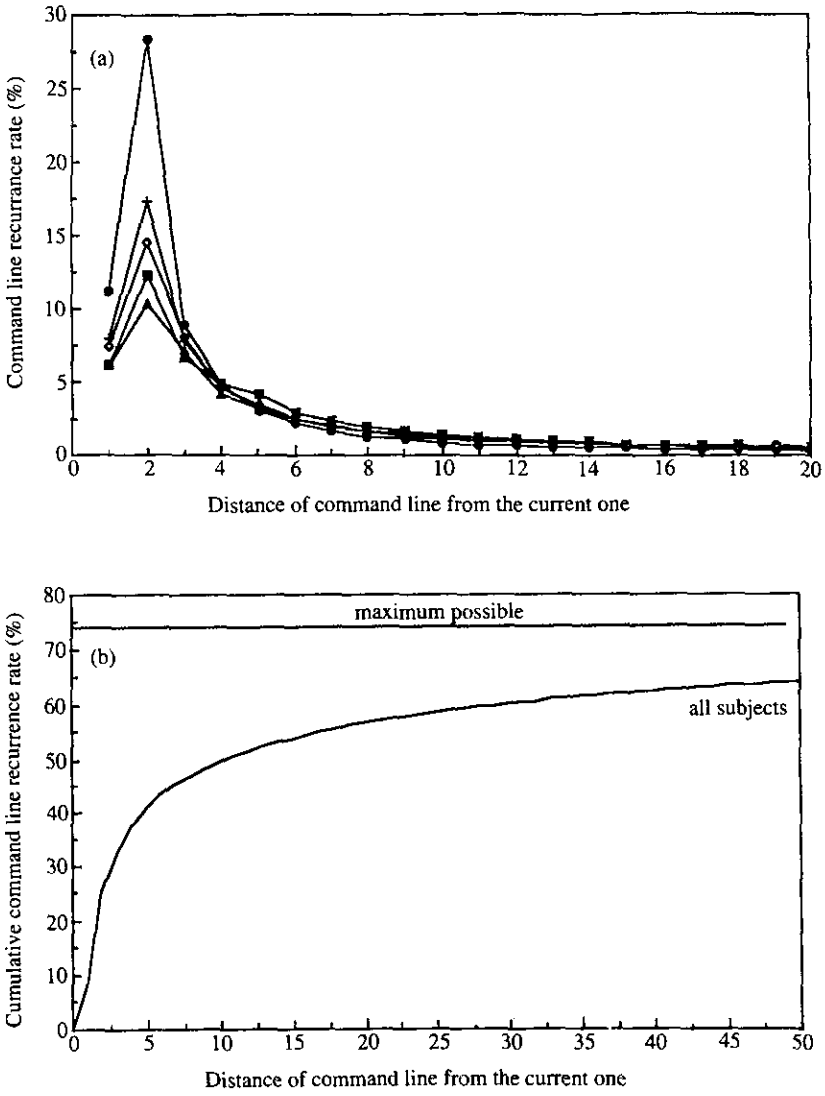

FIGURE 4. (*a*) Recurrence distribution as a measure of distance. ●, novice programmers; ■, experienced programmers; ▲, computer scientists; ◇, non-programmers; +, all subjects. (*b*) Cumulative recurrence distribution as a measure of distance.

values of $R_{s,d}$ from a subject's trace. The mean recurrence rate for a given distance d over all S subjects in a particular group is then calculated as:

$$R_d = \frac{1}{S}\sum_{s=1}^{S} R_{s,d}.$$

These group means are plotted in Figure 4(*a*). The vertical axis represents R_d, the rate of command line recurrences, while the horizontal axis shows the position of the repeated command line on the history list relative to the current one. Taking Novice Programmers, for example, there is an $R_{d_1} = 11\%$ probability that the

current command line is a repeat of the previous entry (distance = 1), $R_{d_2} = 28\%$ for a distance of two, $R_{d_3} = 9\%$ for three, and so on. The most striking feature of the Figure is the extreme recency of the distribution.

The previous seven or so inputs contribute the majority of recurrences. Surprisingly, it is not the last but the second to last command line that dominates the distribution. The first and third are roughly the same, while the fourth through seventh give small but significant contributions. Although the probability values of R_d continually decrease after the second item, the rate of decrease and the low values make all distances beyond the previous ten items equivalent for practical purposes. This is illustrated further in Figure 4(*b*), which plots the same data for the grouped total as a running sum of the probability over a wider range of distances. The running sum of the recurrence rate up to a given distance D for a single person is called R_D. Its mean value over a group of subjects is calculated as

$$R_D = \frac{1}{S} \sum_{s=1}^{S} \sum_{d=1}^{D} R_{s,d}.$$

As Figure 4(*b*) shows, the most recently entered command lines on the history list are responsible for most of the cumulative probabilities. For example, there is an $R_{D_{10}} = 47\%$ chance that the next submission will match a member of a working set containing the ten previous submissions ($R_{D_{10}}$ is an abbreviation of R_D with $D = 10$). In comparison, all further contributions are slight (although their sum total is not). The horizontal line at the top represents a ceiling to the recurrence rate, as $C = 26\%$ of all command lines entered are first occurrences.

Figure 4(*a*) also shows that the differing recurrence rates between user groups, noted previously in Table 2, are mostly attributed to the three previous command lines. Recurrence rates are practically identical elsewhere in the distribution. This difference is strongest on the second to last input, the probability ranging from a low of 10% for Scientists to a high of 28% for Novice Programmers.

The statistics of UNIX *csh* use, as with the recurrent systems mentioned in Section 2, indicate that the most recently submitted activities are the most likely to be repeated. These statistics confirm the potential of reuse facilities in general, and verify the assumptions, made by many existing history mechanisms, of using recency.

7. Increasing the opportunities for reuse in UNIX csh

In the last section, particular attention was paid to the recurrence of command lines during *csh* use, and to the probability distribution of the next line given a sequential history list of previous ones. We saw that the most striking feature of the collected statistics is the tremendous potential for a historical reuse facility: the recurrence rate is high and the last few submissions are the likeliest to be repeated. One may predict with a reasonable degree of success what the user will do next by looking at those recent submissions.

Yet there is still room for improvement, since a significant portion of recurrences are *not* recent submissions. For example, consider a working set of the ten previous items on the *csh* history list that is displayed to a user for selection. Although there is a $R_{D_{10}} = 47\%$ chance that the next command line can be successfully predicted by this display [Figure 4(*b*)], there is still a 27% chance that it last appeared further

back. When combined with the $C = 26\%$ chance that the next submission has not appeared before, then the history list will fail to be of any benefit 53% of the time. Can predictions of the user's next step be offered that better these figures?

This section proposes alternative strategies of arranging a user's command line history to condition the distribution in different ways, firstly to increase the recurrence probabilities over a working set of a given size, and secondly to improve the overall "quality" of predictions offered. Each method is applied to the UNIX traces, and their predictive quality is measured and contrasted against each other. The following subsections explain how quality is assessed, describe a variety of conditioning techniques, and apply these conditions to the traces that have been collected.

7.1. THE QUALITY OF PREDICTIONS

Predictions of activities for reuse are only effective when the search for and selection of an offering is less work for the user than submitting it afresh. Work is therefore used to measure prediction quality. The smaller the amount of work required for reuse as opposed to resubmission, the higher the quality of the set of predictions offered. The selection of a high-quality prediction either reduces the cognitive effort of reconstructing the original activity or minimizes the physical work required to enter that activity to the system.

There are several ways to calculate the "work" involved for reuse vs. resubmission. First, we can observe users interacting with a particular reuse facility; the catch is that the results may be heavily dependent on a particular implementation. Second, we can model the task by cognitive modelling methods such as GOMS (Card, Moran & Newell, 1983), which take into account mental and physical operations. While the results may be more general than those supplied by a usability study, the operations are still tied to a particular style of interaction. Lee (1992) used GOMS to generate predictions about the mental and physical effort involved with several different styles of reuse facilities, and concluded that:

- reuse is indeed a tradeoff of more cognitive effort for selection vs. less physical effort for typing;
- increased mental effort may be alleviated by designing history tools favouring simple mental operations;
- non-expert typists using history tools expend less overall effort.

Because physical work is an essential parameter within this model, reuse facilities that are good predictors will lessen the overall amount of work required of the user. This leads to the third method that we use in this paper. We calculate a metric for work that accounts for both how well a reuse method predicts the next submission, and how many characters will be predicted per submission. This relates directly to the sizes of lists a user must scan, and the amount of typing that may be saved. While the metric does not indicate whether the tradeoff of reuse vs. resubmission is worthwhile, it does allow us to contrast a variety of prediction methods.

The metric for work introduced here is called M_D, and comprises two components that estimate a prediction's quality. The first is R_D, the probability that the desired item appears on a displayed list of length $p = D$. Its calculation was given in Section

4. The second, called $\bar{c}_d$, is the average number of characters saved by reusing the matching activities at exactly a particular distance d. Incorporating string length as a partial indicator of work assumes, of course, that longer strings are harder to recall and re-enter than short ones. M_D indicates the average number of characters saved over all submissions when repeated activities are selected from a list of candidates of length D. By using M_D, predictive methods can be numerically compared and ranked accordingly.

The calculation of M_D and its components proceeds as follows. First, let $\bar{c}_{s,d}$ be the average number of characters saved by a subject s per recurrence at distance d, calculated as:

$$\bar{c}_{s,d} = \frac{c_{s,d}}{r_{s,d}}.$$

The term $c_{s,d}$ is the total number of characters saved by the subject reusing all matching recurrences at a particular distance, and $r_{s,d}$ is the number of matching recurrences at that distance. When $\bar{c}_{s,d}$ is averaged over all subjects S, we get $\bar{c}_d$, calculated as:

$$\bar{c}_d = \frac{1}{S}\sum_{s=1}^{S} \bar{c}_{s,d}.$$

But $\bar{c}_d$ just gives the average characters saved by using a correct prediction at a particular distance. An alternative approach is to include the probability that the prediction is correct. Specifically, M_d is the mean number of characters saved at a particular distance over all subjects:

$$M_d = \frac{1}{S}\sum_{s=1}^{S} \bar{c}_{s,d} R_{s,d},$$

where $R_{s,d}$ is a particular subject's probability of a recurrence at the given distance, as defined in Section 6. Note that M_d differs from $\bar{c}_d$ because it is the average saving *per submission* rather than per recurrence. The final step in calculating M_D shows the cumulative average saving in characters per submission when one through D predictions are available for selection:

$$M_D = \frac{1}{S}\sum_{d=1}^{D} M_d = M_{d_1} + M_{d_2} + \cdots + M_{d_D}.$$

Both R_D and M_D will be reported in this paper as metrics for evaluating working sets of particular sizes. Values of R_d (defined in Section 5) and $\bar{c}_d$ may be found in Greenberg (1993).

7.2. DIFFERENT CONDITIONING METHODS

A variety of conditioning methods are described here. As well as conditions that are expected to perform quite well, poorer ones that have been implemented in existing reuse facilities are also included. For each method we indicate how the recorded data was analysed to assess its effectiveness. The algorithms used to find $R_{s,d}$ for

each case are not elaborated (they are minor variations of the one shown in Figure 3). Results will be presented in Section 7.3, and will show how effective—or ineffective—these conditioning methods really are.

Sequential ordering by recency

This conditioning method was described in Section 6 and is simply an ordered history list of 14 UNIX command lines numbered by order of entry. The most recent submission appears on the top, and the history list—as with all other examples in Figure 5—is intended to be reviewed top-down.

There are two virtues of using a simple recency strategy in a reuse facility. First, the items presented would be the ones a user has just entered and still remembers. He knows they are on the list without having to scan through it. Second, unlike some other methods, such as frequency-ordered lists, there is no initial startup

Sequential *starting in ~/text*

14	cd ~/figs
13	print draft
12	cd ~/text
11	edit fig1
10	ls
9	graph fig1
8	edit fig2
7	edit fig1
6	ls
5	cd ~/figs
4	edit draft
3	print draft
2	edit draft
1	ls

Duplicates Removed

original position

12	cd ~/text
9	graph fig1
8	edit fig2
7	edit fig1
5	cd ~/figs
3	print draft
2	edit draft
1	ls

latest position

14	cd ~/figs
13	print draft
12	cd ~/text
11	edit fig1
10	ls
9	graph fig1
8	edit fig2
4	edit draft

Frequency Order

secondary key is recency

10	ls	3
14	cd ~/figs	2
13	print draft	2
11	edit fig1	2
4	edit draft	2
12	cd ~/text	1
9	graph fig1	1
8	edit fig2	1

secondary key is reverse-recency

10	ls	3
4	edit draft	2
11	edit fig1	2
13	print draft	2
14	cd ~/figs	2
8	edit fig2	1
9	graph fig1	1
12	cd ~/text	1

Alphabetic *duplicates removed*

14	cd ~/figs
12	cd ~/text
4	edit draft
11	edit fig1
8	edit fig2
9	graph fig1
10	ls
13	print draft

Directory Sensitive

directory context is ~/text

14	cd ~/figs
3	print draft
5	cd ~/figs
4	edit draft
13	print draft
2	edit draft
1	ls

directory context is ~/figs

12	cd ~/text
8	edit fig1
10	ls
9	graph fig1
11	edit fig2
7	edit fig1
6	ls

with duplicates removed, events saved in latest position

1	ls
4	edit draft
13	print draft
14	cd ~/figs
8	edit fig2
9	graph fig1
10	ls
11	edit fig1
12	cd ~/text

Commands *recency, no duplicates*

14	cd
13	print
11	edit
10	ls
9	graph

FIGURE 5. Examples of history lists conditioned by different methods. In UNIX, users changed directories through the cd command. The "~" is shorthand for the home directory: following "/" indicates sub-directories.

instability of deciding what to present to the user when only a few items are available.

Pruning duplicates from the history list

The sequentially-ordered history lists mentioned so far maintain a record of every single command line typed. Duplicate lines are not pruned off the list. On a displayed history list of limited length, duplicates occupy space that could be used more fruitfully by other command lines.

There are two obvious strategies for pruning redundancies, as described by Barnes and Bovey (1986). The first saves the activity in its original location on the history list, a method employed by RECENT, the history system built into the Macintosh HYPERCARD (Goodman, 1987). The second saves it in its latest position, the method chosen by the MINIT history system that combines command processing and history into its single "window management window" (Barnes & Bovey, 1986). It is expected that the latter approach would give better performance, as not only is local context maintained, but unique and low-probability command line entries will migrate to the back of the list over time.†

Consider, for example, the two pruned event lists in the second major column of Figure 5. Both are the same length, which is considerably shorter than the plain sequential one in the first column. But the order of entries are quite different. Even in this short list, the disadvantage of saving items in their original position is evident. Local context is weak (indicated by the scattered event numbers), and the regularly repeated *ls* command line is poorly positioned at the bottom of the list.

Data sets were re-analysed using both strategies of pruning duplicates off sequential history lists, where recurring items were either kept in their original position or moved to their latest position.

Frequency ordering

Perhaps the most obvious way of ranking activities is by frequency, where the most often-used command line appears at the front of the history list and the rarest one at the end. This approach is conservative. Old and frequently used items tend to stay around—unless there is a built-in decay factor—while newer submissions will not appear near the head of the list until they are repeated as often as the old ones. Still, it may do as well as or perhaps even better than recency.

When ordering items by frequency, it is necessary to consider a tie-breaking strategy for items of identical frequencies. One possibility uses recency as the secondary sorting key. For example, if the current submission is a recurrence, its frequency count is increased by one and it is relocated before all other recurrences with the same count. Another approach uses a secondary sort by reverse-recency, where the recurring item is placed at the tail of the list of items with identical frequencies. Contrasting these two methods gives a bound to the range of recency effects. Example are shown in the top right column of Figure 5, where the number to each item's right counts how often that line has been submitted.

It is expected that frequency ordering may do quite well, given that UNIX

† Saving recurring activities in their latest position only is equivalent to "self-organized files", where successfully located records are moved to the beginning of the sequentially accessed file. As briefly discussed by Knuth (1973), oft-used items tend to be located near the beginning of the file, and the average number of comparisons is always less than twice the optimal value possible.

command lines often consist of a frequently-executed command without arguments. But probably fewer characters are predicted, since short lines would tend to dominate the higher frequencies. Another disadvantage of frequency order is that counts must now be associated with every submission. At best, this just takes up some space and a little cpu time, which matters little in these days of cheap memory and fast machines. At worst, the derived probabilities associated with a young history list are quite unstable and may lead to very poor initial predictions, which could discourage a new user from placing faith in it (c.f. recency).

The data sets were analysed by ordering history lists by frequency and using two cases of secondary sorting: recency and reverse-recency. Since there is no advantage in keeping multiple copies of command lines, they were pruned from the list.

Alphabetic ordering

Sorting activities alphabetically is another possibility. Although items on alphabetic ordered lists are best found by binary search or pattern matching, surprisingly many systems use only scrolling for sequential searching. Most direct manipulation systems, for example, present lists of items (such as files) in scrollable alphabetic lists. A history system example is MINIT's window management window, which provides it as a display option (Barnes & Bovey, 1986). We would expect poor performance of a distribution derived from alphabetic ordering. Letter frequencies aside, it should do no better than a random ordering of events. Performance is easily evaluated by seeing how many pages of previous activities would have to be scrolled on average before the desired item is found.

User's traces were re-analysed by placing their command lines on a history list in ASCII order. If a new submission was identical to one already on the list, it was ignored. An example of an ASCII-ordered list is included on the bottom left of Figure 5.

Context-sensitive history lists by directory

Users of computer systems perform much task switching (Bannon *et al.*, 1983), where each task represents an independent or interacting context. Since many command line submissions are specific to the task at hand, it is reasonable to hypothesize that context-sensitive history lists will give better local predictions.

Ideally, the reuse facility would infer the context of every submission entered and place it on an appropriate history list, creating a new one if needed. Events common to multiple contexts could perhaps be shared between lists. The system would then infer the likely context of the next submission and offer its predictions for reuse only from the appropriate list.

Associating a user's activities with their tasks or goals is not easy, and such inferences cannot be made reliably. Instead, a simple heuristic provides a reasonable guess of the true context. UNIX furnishes a hierarchical directory system for maintaining files. As many user actions reference these files, we hypothesize that the current working directory defines a context for command lines. This grouping of command lines by the current directory (or perhaps by the obvious alternative of windows) is just an approximation—possibly a poor one—to actual task contexts.

When data was collected, each user submission was annotated with the directory it was run in. The traces were re-analysed by creating a new history list for each new

directory visited and placing the command line on that list. The recurrence distance for each submission was then calculated by retrieving the history list for the current directory of the next submission and searching it for the most recent match.

The second main column in the lower half of Figure 5 illustrates the directory-sensitive condition applied to the sequential input, where each sub-column is sensitive to a particular directory. No pruning of duplicates is done. Most command lines refer to files in that directory, and would rarely be used in other directories. However, some command lines, for example *ls* for listing files, are common to more than one directory.

Ordering commands by recency

Section 4 showed that most individuals use only a handful of the available commands, and that the frequency distribution of command selection is very uneven. It would be interesting to see how a history list comprised of recency-ordered commands (not command lines) would perform. Although we expect the probability of a matching prediction R_D to be quite high, the characters predicted per recurrence would be lower, since the rest of the command line is ignored (see the example in Figure 5, bottom right column).

User traces were re-analysed over history lists of commands. Duplicate commands were pruned, with a single copy of the command kept in its position of latest occurrence.

Partial matches

Instead of the next command line matching a previous one exactly, partial matching may be allowed. This is helpful when: people make simple spelling mistakes; the same command and options are invoked on different arguments; command lines are extended with additional arguments; and so on.

However, the potential benefit is highly user- and situation-dependent, for the user must alter the selected sequence before it is invoked. Consider the next submission s and its partial match to a previous event e on the history list. If selecting and modifying e is easier and more reliable than entering s, then it is an attractive strategy. If s is long, for example, and differs from e by a single character, selecting and fixing e is probably faster. If s is short, it is unlikely that the user would bother.

Partial matches by prefix were investigated. A command line is matched whenever it is a prefix of the next submission. If $s =$ "*edit fig2*", for example, some partial matches on prefix for e could be "*ed*", "*edit*", "*edit fig*", and "*edit fig2*".

In partial matching, history lists are not altered. Rather, it is the definition of recurrence that has changed. Any increase in predictive probability comes at the expense of fewer useful characters predicted. Effects of partial matching were shown for a recency-ordered history list both with duplicates retained and with duplicates pruned.

A hierarchy of command lines and command-sensitive sublists

One way of increasing the effectiveness of a history list is by using existing displayed items as a hierarchical entry point to related items. More specifically, consider a history list of command lines where each item can further raise a secondary list of

all lines that share the same initial command (which we call a *command-sensitive sublist*). One first scans down i entries in the normal list for either an exact match which terminates the search, or for a line that starts with the desired command. In the latter case, the command-sensitive list is displayed (perhaps as a pop-up menu) and the search continues until an exact match is found j entries later. The distance of a matching recurrence is simply $i+j$. Given the sequential list in Figure 5, for example, the command sensitive sublist on item 11 (*edit fig1*) would contain *edit fig2* and *edit draft*.

Such a scheme could do no worse than the original method of displaying the history list, and has potential to do much better. This method was tested by using recency-ordering of both the primary and command-sensitive history lists with duplicates saved in their latest position only.

Combinations

The strategies above are not mutually exclusive, and can be combined in a variety of ways. The bottom half of column 2 of Figure 5 shows one such possibility, where the event list is conditioned by directory sensitivity and pruning. Data sets were re-analysed using combinations of a few conditions mentioned above.

7.3. EVALUATING THE CONDITIONING METHODS

Data selection

Conditioning by directory context is no different from standard sequential history if subjects only work within a single directory. As not all subjects used multiple directories, this portion of the analysis was restricted to the experienced programmers, each of whom used several directories.† All other groups had subjects who used one directory exclusively (17 of the 55 novice programmers, 6 of the 25 non-programmers, and 2 of the 52 computer scientists). Each subject from the experienced programmer group was re-analysed using the various conditioning methods and some of their combinations for redefining both the history list and the method of determining recurrences.

Length of command lines and M_D

Before delving into details of how each method performs according to the quality metric, we need to determine the best performance possible. To start, the average length of command lines is 7·58 characters, where terminating line feeds are not counted and duplicate lines are included. This was calculated by finding the average line length for each subject, and averaging those results over all subjects. These numbers will under-estimate the actual characters typed, for editing sequences (such as backspaces and correction keystrokes) are not included.

Since reuse facilities can only predict lines that have been entered previously, it is

† Another reason for limiting the number of subjects analysed is more pragmatic—about 4–8 h of machine time were required to process a single condition for each group.

important to know if recurring lines have a different average length than those appearing only once. Further analysis shows that the average length of submissions that already exist on the history list is 5·97 characters, while those that appear for the first time are 12·29 characters long. This is not as surprising as it might seem at first, for short lines with few arguments are usually more general-purpose (and therefore reusable) than complex lines. We would expect frequently-appearing lines to be shorter than lines that are rarely or never repeated.

The maximum possible value for M_D is therefore $5{\cdot}97R/100$, for M_D is calculated over all submissions. As R is 74·4% for experienced programmers, M_D for an optimal conditioning method is 4·43 characters predicted per submission.

7.4. RESULTS

Results for all conditions are summarized in two tables, each presenting various distributions over the last 50 items of the history list. Table 3 presents the percentage frequency of submissions recurring as a running sum over distance (R_D). This includes the total recurrence rate over the complete history list, which differs with certain conditions.† Figure 6 graphs the results of Table 3. As with Figure 4(*b*), the horizontal axis shows the position of the repeated command line on the history list relative to the current one, while the vertical axis represents R_D, the rate of accumulated command line recurrences, as a percentage.

The next table involves the length in characters of recurrences. Table 4 displays the metric M_D, which shows how many characters are saved for an average submission. This value accounts for recurring and non-recurring submissions, and assumes that the user can select from D predictions. Figure 7 graphs the performance of each conditioning method over distance using this metric. Tables of values of R_d (defined in Section 5) and $\bar{c}_d$ are found in Greenberg (1993).

Standard sequential

$R_{D_{10}}$ is 44·4% for the experienced programmer group (Table 3), which is around 60% of the maximum value it could have ($R = 74{\cdot}4\%$). The metric $M_{D_{10}}$ for the same group is 2·48 characters per submission (Table 4), which is 55% of its maximum value of 4·43 characters. These figures will be used as a benchmark for comparing other conditioning methods.

Pruning duplicates

Although pruning duplicates off the history list does not alter the recurrence rate, it does shorten the total distance covered by the distribution (i.e. the history list is smaller).

First, how does saving single copies of recurring activities in their original position on the history list compare with saving items in their latest position? A quick glance at the tables and graphs shows that the former gives exceedingly poor predictive performance. Curiously, saving activities in their original position gives a much higher average length of predicted strings than for any other conditioning method for lines recurring over small distances (see Greenberg, 1993). But it is the

† The recurrence rate differs when the way of determining matching submissions changes (partial matching, commands only) and when the history list is split into multiple lists (directory sensitivity).

TABLE 3

Cumulative probabilities in percent of a recurrence over distance (R_D) for various conditioning methods

Condition method	Distance														
	1	2	3	4	5	6	7	8	9	10	20	30	40	50	*R*
Recency, duplicates saved:															
always	6·12	18·41	25·12	29·94	34·06	37·00	39·36	41·33	42·99	44·39	52·67	56·82	59·58	61·47	74·42
in original position only	2·53	4·28	5·57	6·65	7·66	8·48	9·27	9·93	10·68	11·29	15·92	19·58	22·82	26·29	74·42
in latest position only	6·12	18·94	26·52	31·87	36·80	40·28	43·11	45·48	47·47	49·17	58·98	63·51	66·00	67·67	74·42
Frequency order:															
second key recency	13·13	21·08	26·32	30·29	33·66	36·48	38·95	41·06	42·85	44·41	55·35	60·98	64·31	66·48	74·42
second key reverse recency	13·16	20·89	26·05	29·90	33·09	35·83	38·21	40·12	41·84	43·37	53·63	58·85	62·02	63·93	74·42
Alphabetic order:															
duplicates removed	1·27	2·27	3·48	4·78	5·80	7·05	7·81	8·68	9·52	10·09	16·53	21·76	25·84	30·16	74·42
Directory-sensitive by recency:															
duplicates included	7·46	21·07	29·27	34·16	37·66	40·39	42·44	44·12	45·63	46·85	53·52	56·62	58·48	59·69	65·53
duplicates removed	7·46	21·75	31·15	36·93	41·06	44·18	46·54	48·60	50·13	51·51	58·80	61·56	62·93	63·74	65·53
Commands only by recency:															
duplicates removed	15·36	35·23	46·12	53·17	58·92	63·01	66·12	68·68	70·89	72·70	82·61	86·83	89·05	90·49	95·24
partial matching by recency:															
duplicates included	8·17	21·65	29·26	34·71	39·22	42·57	45·17	47·34	49·19	50·78	60·16	64·74	67·78	69·93	84·39
duplicates removed	8·17	22·23	30·83	36·90	42·25	46·14	49·20	51·84	54·10	56·02	66·90	72·04	74·94	76·88	84·39
Command hierarchy:															
recency, duplicates removed	6.12	20·01	29·36	35·96	41·52	45·56	48·74	51·44	53·71	55·54	64·81	68·38	70·17	71·21	74·42

R, total recurrence rate.

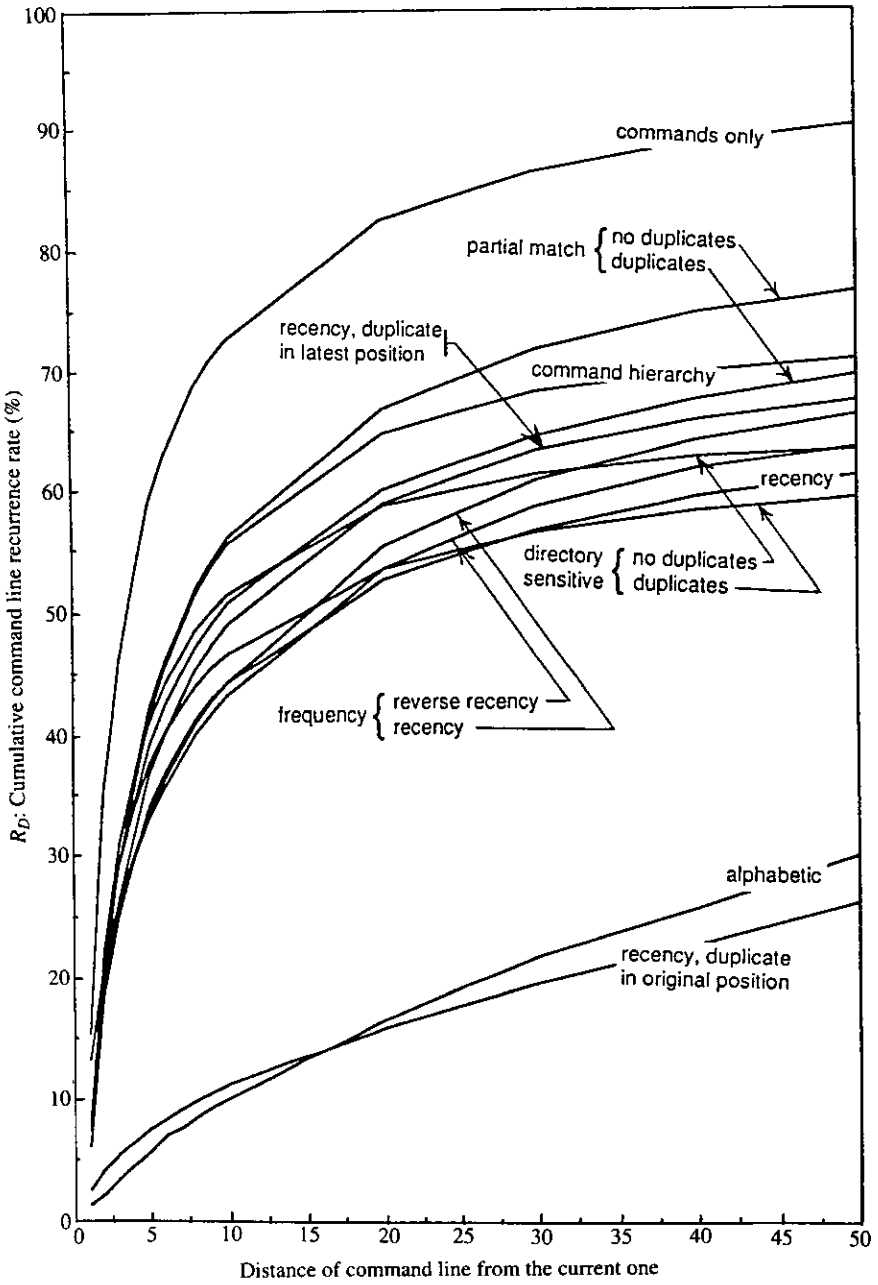

FIGURE 6. Cumulative probabilities of a recurrence (R_D) over distance for various conditioning methods.

low-frequency lines that must contribute most to this average, as high-frequency ones do not remain near the front of the list. This larger than expected line length supports the hypothesis that oft-repeated lines are shorter on average than rarely repeated ones. However, the low probability values associated with those recurrences reduce any benefit accrued by predicting longer lines.

TABLE 4

Cumulative average number of characters saved per submission over distance (M_D)

	Distance													
Conditioning method	1	2	3	4	5	6	7	8	9	10	20	30	40	50
Recency, duplicates saved:														
always	0·37	0·99	1·35	1·63	1·87	2·04	2·19	2·31	2·40	2·48	2·99	3·25	3·43	3·55
in original position only	0·27	0·46	0·59	0·69	0·78	0·86	0·94	0·99	1·05	1·10	1·48	1·75	1·98	2·20
in latest position only	0·37	1·02	1·44	1·76	2·05	2·25	2·42	2·56	2·68	2·78	3·40	3·69	3·86	3·98
Frequency order:														
second key recency	0·32	0·64	0·93	1·16	1·34	1·49	1·62	1·75	1·86	1·96	2·74	3·19	3·49	3·68
second key reverse recency	0·33	0·63	0·91	1·14	1·30	1·45	1·57	1·69	1·79	1·89	2·50	2·99	3·26	3·43
Alphabetic order:														
duplicates removed	0·03	0·08	0·15	0·24	0·31	0·43	0·48	0·54	0·61	0·65	1·12	1·45	1·69	1·91
Directory-sensitive by recency:														
duplicates included	0·48	1·28	1·76	2·05	2·27	2·44	2·57	2·67	2·76	2·83	3·25	3·45	3·57	3·65
duplicates removed	0·48	1·32	1·88	2·24	2·45	2·68	2·83	2·95	3·04	3·13	3·59	3·77	3·87	3·93
Commands only by recency:														
duplicates removed	0·50	1·03	1·34	1·55	1·73	1·86	1·95	2·03	2·10	2·15	2·46	2·59	2·67	2·71
Partial matching by recency:														
duplicates included	0·45	1·11	1·50	1·79	2·04	2·21	2·36	2·47	2·56	2·64	3·12	3·35	3·51	3·62
duplicates removed	0·45	1·14	1·60	1·93	2·21	2·41	2·57	2·71	2·82	2·92	3·47	3·72	3·86	3·96
Command hierarchy:														
recency, duplicates removed	0·37	1·11	1·68	2·09	2·43	2·68	2·88	3·04	3·18	3·30	3·90	4·12	4·23	4·29

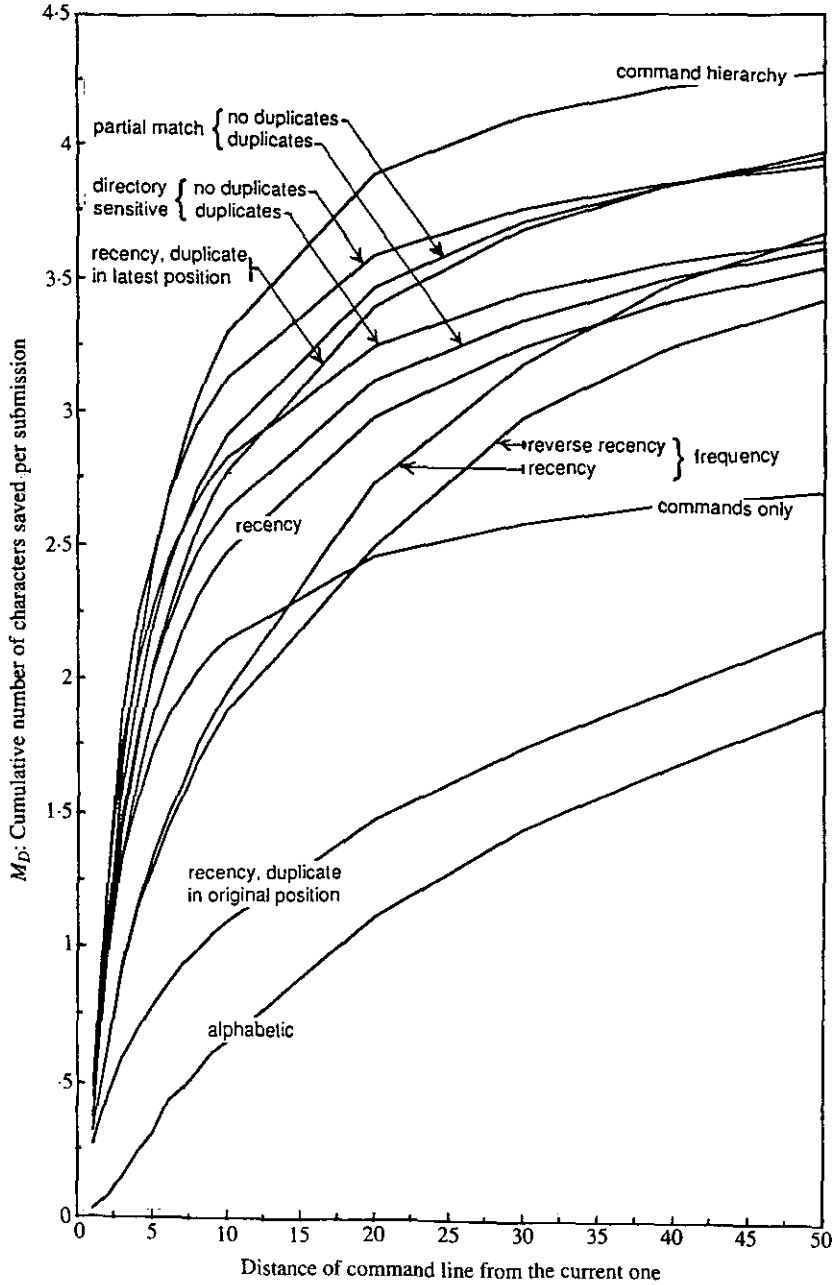

FIGURE 7. Cumulative average number of characters saved per submission (M_D) over distance.

Second, how good is the strategy of saving duplicates in their latest position? Consider a 10-item working set. The probabilities $R_{D_{10}}$ of a recurrence falling in that set are 11% and 49% for the original and the latest position respectively, and the corresponding values of $M_{D_{10}}$ are 1·10 and 2·78 characters per submission. Saving activities in their original position is clearly ineffective. Unless stated otherwise, the

remainder of this paper assumes that history lists with duplicates pruned will always save the single copy in its position of latest occurrence.

As the working set size increases, so does the value of R_D associated with a duplicates-pruned list when compared to the standard sequential list (Table 3 and Figure 6). Pruning duplicates increases the overall probability of a 10-item working set by 4·8% ($R_{D_{10}} = 49{\cdot}1\%$ vs. 44·4%), and $M_{D_{10}}$ is increased by 0·3 characters per submission to 2·78.

Frequency order

Using recency as a secondary sort in a frequency-ordered list is marginally better than sorting by reverse-recency. The overall probability that a 10-item working set will contain the next submission is 1·1% higher, and 0·1 character more is predicted per submission. Since these reflect the bounds of these two conditions, it is hardly worth worrying about how to do the secondary sort. Still, whenever frequency-ordered lists are discussed in this paper, the better secondary sort of recency is assumed unless stated otherwise.

Frequency-ordered history lists do not do as well as strict sequential ones, even though duplicates are not included in the former. Although the probability of a hit in a 10-item working set is about the same ($R_{D_{10}} = 44{\cdot}4\%$), lines predicted are shorter (as expected). The metric $M_{D_{10}}$ is 0·6 characters less per submission.

Alphabetic order

As anticipated, alphabetic ordering of history lists gives the poorest performance of any conditioning technique (this assumes sequential searching through the list). With a 10-item display, $R_{D_{10}} = 10{\cdot}1\%$, and only 0·65 characters are predicted per submission. If a user were scrolling through this display, fully 100 items (or 10 pages) must be reviewed on average to match $M_{D_{10}}$ for the strict sequential list!

Context-sensitive history lists by directory

Creating context-sensitive directory lists with duplicates retained decreases the overall recurrence rate for experienced programmers from 74·4% in the strict sequential case to 65·5%, because command lines entered in one directory are no longer available in others. Although this reduction means that plain sequential lists out perform directory-sensitive ones over all previous entries, benefits were observed over small working sets. The first three directory-sensitive items are more probable than their sequential counterparts, approximately equal for the fourth, and slightly less likely thereafter. The accumulated probabilities R_D cross over with a working set of 27 items (Figure 6). With a working set of 10 items, directory-sensitivity increases the overall probability that the next item will be in that set by 2·5% ($R_{D_{10}} = 46{\cdot}9\%$). The length of lines predicted in the directory-sensitive condition is also longer than that for lines predicted by a strict sequential list, and $M_{D_{10}}$ is 0·35 characters per submission higher.

Ordering commands by recency

When all aspects of a command line are ignored except for the initial command word, the recurrence rate jumps to 95·2%. The accumulated probabilities of recurrences are also very high when compared to the strict sequential list:

$R_{D_{10}} = 72{\cdot}7\%$ vs. 44·4%. However, the high predictability is offset by the low number of characters predicted. $M_{D_{10}}$ actually drops 0·3 characters per prediction.

Partial matches

Pattern matching by prefix increases the recurrence rate to 84·4%, where the recurrence rate is now defined as the probability that any previous event is a prefix of the current one. As partial matches are found before more distant (and perhaps non-existent) exact matches, an increase is expected in the rate of growth of the cumulative probability distribution. This increase is illustrated in Table 3 and Figure 6. Conditioning by partial matching increases $R_{D_{10}}$ of a 10-item working set by 6·4% when compared to a strict sequential list (Table 3), although lines predicted are shorter. Still, $M_{D_{10}}$ is increased slightly by 0·16 characters per submission.

A hierarchy of command lines and command-sensitive sublists

The history list comprised of recency-ordered non-duplicated lines and command-sensitive sublists shows the best performance of all conditions evaluated. The accumulated probability of a 10-item display is $R_{D_{10}} = 55{\cdot}5\%$ out of the 74·4% possible. $M_{D_{10}}$ is 3·3 characters per submission, compared to the 4·4 character maximum for an optimal system.

Combinations

When conditioning methods are combined, the effects are slightly less than additive. A few possible combinations are included by removing duplicates from both the directory-sensitive and partial matching conditions. Each improves as expected, as illustrated by Tables 3 and 4, and Figures 6 and 7. Where feasible, conditioning methods can be combined even further. For example, a partially-matched, pruned and directory-sensitive history mechanism increases $R_{D_{10}}$ over a strict sequential history mechanism by 12·7%, with a working set of 10 items (Greenberg & Witten, 1988*b*).

7.5. DISCUSSION

The recurrence rate R provides a theoretical ceiling on the performance of a reuse facility using literal matches. It is reached only if one reuses old submissions at every opportunity. However, finding and selecting items for reuse could well be more work than entering them afresh, especially if it is necessary to search the complete history list. Pragmatic considerations mean that most reuse facilities choose a small set of previous submissions as predictions, and offer only those for reuse. While the last section demonstrated that temporal recency is a reasonable predictor, the conditioning methods described and evaluated here illustrate that simple strategies can increase predictive power even further.

We saw that up to 55% of all user activity in *csh* can be successfully predicted with working sets of 10 predictions for literal matches, depending upon the conditioning method chosen. However, the best a perfect literal reuse facility could do is, on

average, $R = 75\%$. The best predictive method described here is therefore 55/75 effective, or around 75% of the optimum possible.

When the quality metric is incorporated, we observe that the best method correctly predicts 3·3 characters per submission (with a working set of 10 items), compared to the 4·4 optimum calculated previously. Again, the method is about 75% as effective as the optimum.

In marked contrast, a few conditioning methods perform poorly. Saving duplicates in their original position has no benefit, and alphabetic ordering of the history list is questionable. Although frequency ordering does not fare badly, other methods give better results.

The number of characters saved per submission may seem quite small. The skeptic would conclude that reuse facilities are perhaps not worth the fuss. But a few points should be considered. First, the number of characters saved in practice would be considerably higher, for the string is already formed and editing is not necessary. Actual savings are likely double the theoretical ones, due to the extra editing and correction keystrokes e.g. backspaces, erroneous characters, and so on (Whiteside *et al.*, 1982). That is, selecting an item through history will, in practice, replace around 6·6 keystrokes on average. Second, recognizing and selecting an activity is generally considered easier than recalling or regenerating it. Third, it may all depend upon the user's focus of attention. If he is selecting items from a history list with (say) a mouse, he may continue to do so rather than switch to the keyboard. The reverse is also true.

There is no guarantee that any of the conditioning methods described here will be effective in practice, for the cognitive and mechanical work required for finding and selecting items for reuse from even a small list may still be too costly. Research is required in three areas. First, other conditioning methods should be explored that further increase the probability of a set of predictions (up to the value of R). One candidate is the use of a model similar to that employed by the REACTIVE KEYBOARD (Darragh & Witten, 1992), an adaptive system that bases its prediction on a frequency-based Markov model. Another candidate, explored by Lee (1992), uses *locality* of command recurrences to account for history use. While promising in principle, her study showed that command line recurrences exhibit poor locality (31%). Also, a predictive method based on locality remains to be constructed. Second, the size of the working set should be reduced. Ideally, only one correct prediction will be suggested. Third, the cognitive effort required for reviewing a particular conditioned set of predictions must be evaluated (see the GOMS modelling of reuse of Lee, 1992). One factor is whether the user knows beforehand if the item being sought appears in the set, otherwise she may face an exhaustive and ultimately fruitless search. Another factor is whether the item can be found rapidly. Given these factors, it is possible that one conditioning technique may give better practical performance than another theoretically superior one.

8. Principles: how users repeat their activities

The preceding sections analysed command-line recurrence in dialogues with the UNIX *csh*. Based on the empirical results, the first part of this section formulates general principles that characterize how users repeat their activities on computers.

Design Guidelines

- ⊙ Users should be able to recall previous entries.
- ⊙ It should be cheaper, in terms of mechanical and cognitive activity, to recall items than to re-enter them.
- ⊙ Simple reselection of the previous five to ten submissions provides a reasonable working set of possibilities.
- ⊙ Conditioning of the history list, particularly by pruning duplicates and by further hierarchical structuring, could increase its effectiveness.
- ⊙ History is not effective for all possible recalls, since it only lists a few previous events. Alternative strategies must be supported.
- ⊙ Events already recalled through history by the user should be easily reselected.

FIGURE 8. Design guidelines for reuse facilities.

Some guidelines are also tabulated for the design of a reuse facility that allows users to take advantage of their previous transaction history. The second part steps back from the empirical findings and presents a broader view of reuse.

We abstract empirical principles governing how people repeat their activities from the UNIX study described earlier. They are summarized in Figure 8 as empirically-based general guidelines for the design of reuse facilities. Although there is no guarantee that these guidelines generalize to all recurrent systems, they do provide a more principled design approach than uninformed intuition.

8.1. PRINCIPLES

A substantial portion of each user's previous activities are repeated

In spite of the large number of options and arguments that could qualify a command, command lines in UNIX *csh* are repeated surprisingly often by all classes of users. On average, three out of every four command lines entered by the user have already appeared previously. UNIX is classified as a recurrent system by the definition in Section 2.

This high degree of repetition justifies the intent of reuse facilities. Users should be re-entering their recurring inputs more easily than their original entries. The aim is to reduce both physical tedium and the cognitive overhead of remembering past inputs. Reuse facilities should not be targetted only at experts—they can help everyone.

New activities are composed regularly

Although many activities are repeated, a substantial proportion are new. One out of every four command lines entered to UNIX *csh* is a new submission. Composing command lines is an open-ended activity.

Many modern interfaces provide pop-up menus as a way of structuring and packaging common activities. Though useful for well-understood domain-specific systems that collect specialized tools together, a package of tailored activities will not suffice as a front end to the open-ended recurrent systems addressed by this paper. Although the few facilities shared by users should be somehow enhanced, user composition of new activities must be supported as well.

Users exhibit considerable temporal recency in activity reuse
The major contributions to the recurrence distribution are provided by the last few command lines entered.

Most reuse facilities are history mechanisms designed to facilitate re-entry of the last few inputs (Greenberg & Witten, 1993). Systems that do not have explicit and separate displays of the event list rely on a user remembering his own recent submissions, or on the visibility of the dialogue transcript on the (usually small) screen. Given the high recency effect, we expect limited success by memory alone.

Yet the principle does pinpoint design weaknesses of existing systems. First, the second to last command line recurs more often than any other single input. But many reuse facilities favour access to the last entry instead. For example, typing the shortcuts "redo" and "!!" in the INTERLISP Programmer's Assistant (Xerox, 1985) and UNIX *csh* respectively defaults to the previous submission, and it is slightly harder to retrieve other items. In history through editing, where one sees the interaction record as a scroll of intermixed input and output, a user would have to search through two previous input and output sequences before finding the second to last entry (Greenberg & Witten, 1993).

Second, the major contributions to the recurrence distribution are provided by the previous 7 ± 3 inputs. Yet most graphical history mechanisms display considerably more than 10 events. HISTMENU's graphical menu of history items, for example, defaults to 51 items (Bobrow, 1986), and MINIT's window management window is illustrated with 18 slots in the original paper (Barnes & Bovey, 1986). Considering the high cost of real estate on even large screens, and the user's cognitive overhead of scanning the probabilities, a lengthy list is unlikely to be worthwhile. For example, a menu of the previous 10 UNIX events covers, on average, 45% of all inputs. Doubling this to 20 items increases the probability by only 5%.

The cost/benefit tradeoff of encompassing more distant submissions could also be used to tune other predictive systems that build more complex models of all inputs [see Greenberg and Witten (1993) and Greenberg *et al.* (1993) for a description of several predictive systems oriented towards reuse]. The high recency effect associated with recurrences suggests that a reasonable number of successful predictions can be formed on the basis of a short memory. Perhaps a recency-based short-term memory combined with a frequency-based long-term memory could generate better predictions.

Some user activities remain outside a small local working set of recent submissions
A significant number of recurrences are not covered by the last few items (about 40% of the recurring total with a working set of 10 events, using strict sequential ordering). Doubling or even tripling the size of the set does not increase this coverage much, as all but the few recent items are, for practical purposes, equiprobable.

Unfortunately it is just these items that could help the user most. Since their previous invocation occurred long ago, they are probably more difficult to remember and reconstruct than more recent activities. If the command line is complex, file names would be reviewed, details of command options looked up in a manual, and so on. Excepting systems with pattern-matching capabilities and scrolling—both questionable methods of recall—no implemented reuse facility provides reasonable

ways of accessing distant events. Although alternative strategies are not investigated here, some possibilities are described in Greenberg and Witten (1993).

Working sets can be improved by suitable conditioning

A perfect "history oracle" would always predict the next command line correctly, if it was a repeat of a previous one. As no such oracle exists, we can only contemplate and evaluate methods that offer the user reasonable candidates for re-selection. Although simply looking at the last few activities is reasonably effective—60% of all recurrences are covered by the previous 10 activities—pruning duplicates, context sensitivity, partial matches, and hierarchies of command-sensitive sublists all increase coverage to some degree. Combining these methods is also fruitful. But they have drawbacks too.

Pruning duplicates increases the coverage of a fixed-size list. However, if sequences of several events can be selected [as in INTERLISP's Programmers Assistant (Xerox, 1985)], pruning may destroy useful sequences. In addition, events no longer follow the true execution order, which may confound a user's attempt to recall them by position. Pruning problems also arise when the history list serves other purposes. Consider, for example, the undo facility in the Programmer's Assistant. As side effects of activities are stored along with the text of the activity, undoing two textually-equivalent items may have different results. In this case, items cannot be pruned without compromising the integrity of the undo operation (Thimbleby, 1990).

Conditioning the working set on the current working directory may eliminate useful context-independent items from the history list with only a slight gain in predictive power. But the usefulness of references may improve, since viewing the history list may help remind the user of the specialized and perhaps more complex directives submitted in that context.

Retrieval by partial matching allows a user to select any event and edit it for spelling corrections or minor changes. There is no guarantee that the editing overhead will be less than simple re-entry. The possibility of erroneously retrieving an undesired event must be considered as well.

When command-sensitive sublists are included but ignored, the potential for reuse is still at least as high as the primary list. Using the attached sublists can only increase the chance of finding a correct match. Still, these sublists involve more mechanical overhead for reuse unless they are on permanent display, and even then there is a cognitive overhead associated with hierarchical searches.

Some "obvious" or previously implemented ways of presenting predictions do poorly

Scrolling through alphabetically-sorted submissions is ill-suited to activity reuse. Yet this scheme pervades many modern, popular systems. The Apple Macintosh, for example, presents a scrollable alphabetic display of files for selection within its application. If file access is a recurrent system (which it probably is), then structuring file lists by temporal recency could give quicker selection, especially with the large file stores available on today's computers.

The previous section has shown that saving duplicates in their original position is an extremely poor predictive strategy for maintaining lists. Yet it is used by several history systems. It is the only method of reviewing cards visited in HYPERCARD

[through the RECENT facility (Goodman, 1987)] and is a presentation option in MINIT's window management window (Barnes & Bovey, 1986). Alternative strategies should be encouraged.

Ordering lists by frequency of use may or may not give any benefit over recency. Although used fruitfully by the dynamic menu system (Greenberg & Witten, 1985), the usability and predictive power of that system could increase if recent selections were treated preferentially, perhaps by giving them their own display space on the top-level menu screen.

Predicting commands without their arguments has little value. Although predictability is increased, the overall quality of prediction drops because mostly short sequences are offered. Perhaps inclusion of command-sensitive sublists could improve this fault.

When using history, users continually recall the same activities

UNIX *csh*-users generally employ history for recalling the same events within a login session (this result is based on the same study, and is described in the Appendix in Greenberg and Witten 1993). Once an event has been recalled, it should somehow be given precedence.

Functionally powerful history mechanisms in glass teletypes do poorly

UNIX *csh* history fails on two points, even though it is functionally powerful (Greenberg & Witten, 1993). First, most people (especially novices and non-programmers) never use it. Second, those who do, use it seldom. Only a fraction of all recurrences are recalled through history.

8.2. RECURRENCES: NATURAL FACT OR ARTEFACT?

Where do recurrences come from? Are they naturally part of a human–computer dialogue or are they artefacts imposed by poorly-designed interfaces? If the former, then reuse facilities are an essential component of a good interface. If the latter, they are merely add-on patches; the interface itself should be reconsidered. We will see that, depending upon the situation, recurrences can be either.

The recency effect seen in recurrent systems is probably due to repetitive actions responding to the interaction particulars of a situation that is changing only slightly. In a development task, for example, the situation may be debugging, where the usual responses to particular circumstances comprise a debug cycle. When the development is complete, the cycle terminates. Debug cycles are seen throughout the UNIX traces, and seem responsible for the recurrence probability peaking on the second to last submission. Consider this typical trace excerpt from a non-programmer developing a document:

```
nroff Heading2 Chapter1 | more          | cycle repeats four times
emacs Chapter 1                         |
nroff Heading2 Chapter1| lpr -Plq       | occurs once
```

The sequence shows the user developing a document by iteratively editing the

source text and evaluating the formatted result on the screen, using the *emacs* editor and the *nroff* typesetter. The user's evaluation of the situation determines how often the cycle is repeated. When she was satisfied with the document, she terminated the cycle by producing a final hardcopy.

Another extracted and slightly simplified sequence from a different user illustrates program development using the *fred* editor and the ADA compiler.

```
fred
ada -M concur -o q5.o q5.a        | cycle repeats 11 times
q5.0                              | cycle repeats 3 times
fred
ada -M concur -o q5.o q5.a        | cycle repeats 6 times
q5.0
```

This shows three debug cycles all related to the same development process. In the first, the user edits some source code until it compiles successfully (11 cycles), and then evaluates the executable program. Final tuning of the program is done by expanding the initial debug cycle to include editing, compilation, and execution.

These observations relate to Suchman's thesis that plans are derived from *situated action*—the necessarily ad hoc responses to the contingencies of particular situations (Suchman, 1987). We saw that the user's plan for the development process is necessarily vague, since bugs and difficulties cannot be predicted beforehand. The developer must, of necessity, respond to the particulars of each individual situation. These responses appear repetitious because the situation is altered only slightly after each action.†

In the case of debug cycles, it is certain that some recurrences are artefacts that can be eliminated by different interfaces. Interpreted or incrementally compiled programming environments, for example, remove the necessity for repeated recompilation of the source [see Reiss (1984) for an example]. In other domains, what-you-see-is-what-you-get text processors and spreadsheets not only remove the "compile" step from the cycle, but also show the current state of execution. No distinction is made between the source and developing product, and any changes update the display immediately.

However, other recurrences are not so easily eliminated. Repetitions are often a natural part of the task being pursued. Design work, for example, is fundamentally an iterative process. A second example is telephone dialling. The caller may dial the same number repeatedly when a connection is not made, or he may be a middleman arbitrating information between two or more other people. Retrieval of information in manuals gives another example of recurrences that arise from repetition of our intentions rather than from interface artefacts. Also, consider navigation on computers where people must locate and traverse the many structures necessary for their current context (e.g. navigating file hierarchies and menu-based command sets, and manipulating windows to find pertinent views): since context switching is common, these traversals would recur regularly.

† Although repetitions in the UNIX *csh* dialog shown are identical with the original, the interactions occurring within the editor between its different invocations are probably non-repetitious.

Other recurrences come from long-term context switching. In the UNIX traces, it is usual to see work on a particular task (say document development) occurring in bursts. In a single login session, these bursts may be just a single task interrupted by other dependent or independent diversions. Over multiple login sessions, tasks are constantly released and resumed.

In summary, some recurrences are artefacts arising from particular aspects of system design and implementation. Others are not, for they arise directly from the user's intention, independent of the computer system. Perhaps future systems will minimize the need for reuse facilities by eliminating the artefacts. For the present, reuse facilities remain a potentially viable and very general way of handling repetition.

9. Summary

We have investigated the empirical basis behind an interactive support facility that allows people to reuse their on-line activities. The chief difficulty with this enterprise is the dearth of knowledge of how users behave when giving orders in open-ended computer dialogues. As a consequence, existing reuse facilities—as surveyed in Greenberg and Witten (1993)—are based on ad hoc designs that may not adequately support a person's natural and intuitive way of working.

We began by exploring the notion of recurrent systems, where most users predominantly repeat their previous activities. A few observations of recurrent systems from both non-computer and computer domains were summarized in this context to help pinpoint salient features. Although people were seen to generate many new activities, old ones were repeated to a surprising degree. In UNIX, for example, the average recurrence rate for command lines was 75%. Further study showed that the probability distribution of the next submission repeating a previous one shows recency—the premise behind history mechanisms—to be a reasonable predictor of what the person will do next.

The potential opportunities for reuse were investigated further by describing and evaluating a variety of conditioning methods. Each method used differing strategies for choosing a small set of previous submissions as predictions of the next one. In particular, we saw that up to 55% of all user activity and 3.3 characters per *csh* submission can be predicted successfully with working sets of just 10 predictions. Since the best any literal predictive method could do is $R = 75\%$ on average, or 4·4 characters per submission, the conditioning methods are about 75% effective. There is still room for improvement.

In marked contrast to the theoretic potential of reuse facilities, the "popular" *csh* history is used poorly in practice (as reported in Greenberg & Witten, 1993). Most people, particularly those who are not computer sophisticates, do not use it. Those who do, use it rarely. Only 4% of all activity was reused, compared to the 75% possible! And in spite of the esoteric features available in *csh* history, only the simpler features were used with any regularity. It was suggested that the results observed are likely artefacts of using a poorly designed facility, rather than a human difficulty with the idea of reuse.

We then derived a set of principles that characterize how people repeat their activities on computers. These principles were reformulated as general guidelines

for the design of reuse facilities (Figure 8). Although there is no guarantee that they apply to all recurrent systems and applications, they do seem a reasonable starting point in the absence of more system-specific data.

There is plenty of scope for future research into reuse facilities. We have already built a reuse facility based upon the principles presented here, and have extended the idea further by integrating reuse with a tool that lets one *organize* activities as well (Greenberg, 1993; Greenberg & Witten, 1993). Following the metaphor of a handyman's workbench, this user support facility, through direct manipulation editing, allows the user to pick items off the reuse facility and stash them temporarily on a visible tool shelf or place then semi-permanently within a drawer hierarchy of a tool cabinet. If desired, items can be annotated with a help message and a label. The drawer, which may be displayed and modified at any time, becomes a task-specific toolkit for the user's activities. By using the history list as a primary source of tried and tested candidates for storage within the workbench organization, a person can rapidly create, annotate, and modify his personal workspace so that it responds to his situated needs.

References

BANNON, L., CYPHER, A., GREENSPAN, S. & MONTY, M. (1983). Evaluation and analysis of users' activity organization. *Proceedings of the ACM SIGCHI Human Factors in Computing Systems*, pp. 54–57, Boston: ACM Press. 12–15 December.

BARNES, D. J. & BOVEY, J. D. (1986). Managing command submission in a multiple-window environment. *Software Engineering Journal*, **1**, 177–183.

BENNETT, J. (1975). Storage design for information retrieval: Scarrott's conjecture and Zipf's law. *International Computing Symposium '75*, pp. 233–237, Amsterdam: North-Holland. 2–4 June.

BOBROW, D. (1986). HistMenu. *Lisp User Library Packages Manual, Koto Release.* Xerox Artificial Intelligence Systems. April.

CARD, S. K., MORAN, T. P. & NEWELL, A. (1983). *The Psychology of Human–Computer Interaction.* Hillsdale, NJ: Erlbaum.

DARRAGH, J. J. & WITTEN, I. H. (1992). *The Reactive Keyboard.* Cambridge Series on Human–Computer Interaction. Cambridge: Cambridge University Press.

DRAPER, S. W. (1984). The nature of expertise in Unix. *Interact '84—First IFIP Conference on Human–Computer Interaction*, **2**, pp. 182–186. 4–7 Sept.

ELLIS, S. R. & HITCHCOCK, R. J. (1986). The emergence of Zipf's law: spontaneous encoding optimization by users of a command language. *IEEE Transactions On Systems, Man, And Cybernetics*, **SMC-16**, 423–427.

GOODMAN, D. (1987). *The Complete HyperCard Handbook.* The Macintosh Performance Library. New York: Bantam Books.

GREENBERG, S. (1988). *Using Unix: collected traces of 168 users.* Research Report 88/333/45, Dept of Computer Science, University of Calgary, Calgary, Alberta. Magnetic tape of user data included with report.

GREENBERG, S. (1993). *The Computer User as Toolsmith: The Use, Reuse, and Organization of Command-based Tools.* Cambridge Series on Human–Computer Interaction. Cambridge: Cambridge University Press.

GREENBERG, S., DARRAGH, J., MAULSBY, D. & WITTEN, I. H. (1993). Predictive interfaces: what will they think of next? In A. EDWARDS, Ed. *Extra-Ordinary Human Computer Interaction.* Cambridge: Cambridge University Press. Forthcoming.

GREENBERG, S., PETERSON, M. & WITTEN, I. (1986). Issues and experiences in the design of a window management system. *Proceedings of the Canadian Information Processing Society National Conference*, 21–24 October.

GREENBERG, S. & WITTEN, I. H. (1985). Adaptive personalized interfaces—a question of viability. *Behaviour and Information Technology,* **4,** 31–45.

GREENBERG, S. & WITTEN, I. H. (1988*a*). Directing the user interface: how people use command-based systems. *Proceedings of the IFAC 3rd Man Machine Systems Conference,* Oulou, Finland, 14–16 June.

GREENBERG, S. & WITTEN, I. H. (1988*b*). How users repeat their actions on computers: principles for design of history mechanisms. *Proceedings of the ACM SIGCHI Conference on Human Factors in Computing Systems,* pp. 171–178, Washington: ACM Press. 15–19 May.

GREENBERG, S. & WITTEN, I. H. (1993). Supporting command reuse: mechanisms for reuse. *International Journal of Man–Machine Studies,* **39,** 391–425.

HANSON, S. J., KRAUT, R. E. & FARBER, J. M. (1984). Interface design and multivariate analysis of UNIX command use. *ACM Trans Office Information Systems,* **2,** 42–57.

JOY, W. (1980). *An introduction to the C shell.* University of California, Berkeley, California. November.

KNUTH, D. (1973). *The art of computer programming: Searching and sorting.* Reading, MA: Addison-Wesley.

LEE, A. (1992). *Investigation into history tools for user support.* PhD Thesis, University of Toronto, Department of Computer Science.

PEACHY, J. B., BUNT, R. B. & COLBOURN, C. J. (1982). Bradford-Zipf phenomena in computer systems. *Proceedings of the Canadian Information Processing Society National Conference,* pp. 155–161, Saskatoon, Saskatchewan, May.

REISS, S. P. (1984). Graphical program development with PECAN program development systems. *Proceedings of the ACM SIGSOFT/SIGPLAN software engineering symposium,* Pittsburgh, Pennsylvania, 23–25 April.

ROSS, P., JONES, J. & MILLINGTON, M. (1985). *User modelling in command-driven systems.* Research paper 264, Department of Artificial Intelligence, University of Edinburgh.

SUCHMAN, L. A. (1987). *Plans and Situated Actions: The Problem of Human–Machine Communication.* Cambridge: Cambridge University Press.

SUTCLIFFE, A. & OLD, A. (1987). Do users know they have user models? Some experiences in the practice of user modelling. In H. BULLINGER & B. SHACKEL, Eds. *Human–computer interaction—Interact '87,* pp. 35–41. Amsterdam: Elsevier/North-Holland.

THIMBLEBY, H. (1990). *User interface design.* Frontier Series. New York: ACM Press/Addison-Wesley.

WHITESIDE, J., ARCHER, N., WIXON, D. & GOOD, M. (1982). How do people really use text edition? *Proceedings of the ACM SIGOA Conference on Office Information Systems,* pp. 29–40, June, ACM Press.

WITTEN, J. H., CLEARY, J. & GREENBERG, S. (1984). On frequency-based menu-splitting algorithms. *International Journal of Man–Machine Studies,* **21,** 135–148.

XEROX (1985). *The Interlisp-D reference manual, Volume 2.* Xerox Artificial Intelligence Systems, April.

ZIPF, G. K. (1949). *Human Behaviour and the Principle of Least Effort.* Ontario: Addison-Wesley.