SUPPORTING COMMAND REUSE: EMPIRICAL FOUNDATIONS AND PRINCIPLES

Ian H Witten and Saul Greenberg
1989-375-37
December 1, 1989

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 (eg command lines, menu selections), but computers do little to help them to review and re-execute earlier ones--at most providing ad hoc "history mechanisms" founded on the premise that the last few inputs form a reasonable selection of candidates for re-use. This paper provides theoretical and empirical foundations for the design of a general facility that helps people to recall, modify and re-submit their previous activities to computers. It abstracts several striking characteristics of repetition behaviour from usage data gleaned from many users of different systems. 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 reuse facilities specifically and modern user interfaces generally. A brief case study of actual use of a widely-available history system is included.

Department of Computer Science, University of Calgary, Calgary, Alberta, CANADA.

1 Introduction

Most interfaces offer little help for reviewing and reusing previous activities. Generally activities must be completely re-typed, or reselected through menu navigation. Some systems incorporate *ad hoc* "history" mechanisms that employ a variety of recall strategies, invariably based on the premise that the last n recent user inputs are a reasonable working set of candidates for re-selection. But is this premise correct? Might other strategies work better? Indeed, is the dialog sufficiently repetitive to warrant some type of activity reuse facility in the first place? As existing reuse facilities were designed by intuition rather than from empirical knowledge of user interactions, it is difficult to judge how effective they really are and what scope there is for improvement.

Consider a user who has submitted n activities to the system (say $n > 100$) and whose next activity is identical to a previous one. An optimal reuse facility would be an oracle that correctly predicted when an old action could be reused and submit 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 ($p << n$), 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.

This paper assesses the extent to which people reuse their previous activities. A companion paper (Greenberg & Witten, 1989) lays the groundwork by surveying interactive reuse facilities that allow users to recall, modify, and re-submit their previous entries to computers. 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. As noted in the survey, the main differences between reuse facilities arise from their ability to offer a small but good set of candidates for possible reuse, and from the user interface available to manipulate these candidates.

Schemes for activity reuse are based upon the assumption that the human-computer dialog contains repetition. Yet very little critical attention has been given to this assumption. Section 2 introduces the general model of *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 examined in this context to help pinpoint salient features. Section 3 pursues recurrent systems further through a long-term study of UNIX usage by four different categories of users.

Given that people repeat themselves, what is the best way to predict (and offer for reuse) what will be done next? Section 4 examines several predictive schemes for reuse through an empirical study that derives

the propability distribution of the next activity given a history list of previous ones. A brief case study of actual use of a widely available history system is included in Section 5 as a graphic example of how poorly a supposedly successful system can perform.

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 the 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 dialogs, implying the possibility of generic reuse facilities? This paper provides evidence for the latter view through a model of human-computer dialog called *recurrent systems.*

To motivate the idea, consider how the following real-world examples suggest likely characteristics of the model.

- A cookbook has a subset of recipes referred to repeatedly by a single homemaker. However, usage patterns differ as not all people favor the same recipes. Some cooks prefer tried and true recipes, and will thus use a small set many times. Others desire variety and select from a larger recipe set with less repetition. A similar analogy may be made to favored selections from a book of verse, readings in the bible, or sections and columnists read in a newspaper.
- An audiophile listens to different records repeatedly. Some are heard more than others, and new styles come into favor while old ones die out.
- Within tool-oriented contexts, tradespeople and artisans use some tools more often than others. The actual tools chosen (and repeatedly used) may be task-dependent.
- Procedures carried out by most office workers are routine. Still, special procedures are sometimes followed for rarer conditions and exceptions, while new ones are created to handle unexpected situations.

This section defines recurrent systems and summarizes empirical evidence supporting their existence in two domains: telephone usage and retrieving topics from technical manuals. Section 3 presents a detailed study of a command line interface as a recurrent system.

2.1 Definition and properties

A *recurrent system* is defined as an open-ended system in which users predominantly repeat activities they have invoked previously. In other words, although many activities are possible, most (but not all) are repetitions of previous ones rather than being freshly generated. This definition presupposes incremental interaction, defined as a style of human-computer dialog characterized by successive requests that are submitted to the computer and responded to in turn (Thimbleby, 1990). An *activity* is loosely defined as the formulation and execution of one or more actions whose result is expected to gratify the user's immediate intention. It is the unit entered to incremental interaction systems. Entering command lines, querying databases, choosing items from a pallete, and locating and selecting items in a menu hierarchy are some examples. Copy typing is not. It is continuous rather than incremental, and it is not a cognitive activity (at least, not for the skilled typist).

The fundamental notion behind recurrent systems is that activities are repeated. The frequency of repeats, called the *recurrence rate*, is the probability that any activity is a repeat of a previous one. The *total activities* is the number of all submissions the user has entered, while the *vocabulary size* counts only those that are different. The recurrence rate $\mathcal{R}$ over a set of user activities is calculated as:

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

For a system to be classed as "recurrent," the recurrence rate may exhibit a moderate variation across users, provided that the average rate is fairly high.

Although many old activities are repeated, new ones are constantly added to the repertory. The rate at which new activities are composed and introduced to the dialog is the *composition rate* $\mathcal{C}$:

$$\mathcal{C} = \frac{vocabulary\ size}{total\ activities} \times 100\% = 100 - \mathcal{R}$$

Activity formation within recurrent systems is open-ended, as there are a very large number of possible activities available. Even when new activities are constantly generated, it is expected that these will remain a small subset of the activities that could be formed by any one user.

The following characteristics of recurrent systems have been derived from empirical studies described in the next few sections.

1. Many activities are repeated. The recurrence rate $\mathcal{R}$ ranges between 40–85% .
2. Recurrent systems incorporates new activities regularly, that is, $\mathcal{C}$ falls between 15–60%.
3. The set of activities invoked by any particular user is typically a small subset of the activities usually available.

4. Although the overall recurrence rate remains more or less constant, 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.
6. The set of activities invoked may be disjoint or overlapping for different users of the system.
7. Different people may repeat those 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 only short-term recurrences are counted but those repeated only after long intervals are considered different. Still, the properties provide a reasonable checklist for judging whether particular systems have potential for reuse.

It would seem that, at least on the surface, recurrent systems are just a weaker way of denoting patterns of behaviour already well described by Zipf's law (Zipf, 1949). However, major differences exist. First, many human-oriented observations characterised by Zipf's law are based upon accumulated results of the 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 centered around the statistics of activities of individuals, rather than large groups. Second, 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 are quite comfortable with small numbers. Empirical work has shown that patterns within recurrent system are apparent within small slices of sequential activities entered by one individual. Finally, 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. Recurrent systems handle this well since they emphasise recency as well as frequency of use.

2.2 Empirical evidence for recurrent systems

Previous work has confirmed the existence of recurrent systems, and verified the properties listed above. Recurrent systems are not merely an artifact of computer use, but manifest themselves in everyday activities, as shown by the studies of telephone usage and information access in manuals that are summarized below. Other studies involving human-computer interaction will be described later.

Telephone use is an example of a recurrent system, where an activity is simply a number being dialed. This seems natural, for we know from experience that:

- many calls are to people/firms that have been called before;
- some calls are new ones not made before;
- numbers are called with differing frequencies;
- usage patterns evolve slowly over time;
- actual numbers dialed are quite different between people.

A modest study was conducted on individual telephone use by five people over a one to three month period to determine patterns of recurrence in the numbers dialed (Greenberg, 1984; Greenberg & Witten, 1985). The following characteristics were observed.

1. The average recurrence rate $\mathcal{R}$ of calls made by each user is about 57% (standard deviation=7.7).
2. New telephone numbers were dialed regularly, suggesting that telephone use is not restricted to a few numbers dialed repeatedly, but is, in fact, open-ended ($\mathcal{C} = 43\%$).
3. When each subject's calls were ranked by frequency, there was a wide spectrum between highly and rarely repeated numbers. This decreasing trend can be loosely modeled by the Zipf distribution, although the Zipf decrease is significantly more pronounced than in the telephone usage distribution. However, there is a danger of ascribing a fixed frequency profile to what is most likely a fluctuating frequency recurrent system. There seems to be five general categories of calls: highly popular numbers which are called quite often; moderately popular ones called infrequently; once-only calls which are never or very rarely repeated; new ones never seen before that are incorporated in the repertory; and calls whose popularity fluctuates over time.
4. Telephone numbers that have just been dialed are more likely to be repeated than those dialed long ago. Using one subject as an example, 10% of calls were a repeat of the last number dialed, 8% repeated the second from last, 5% the third from last, and so on with decreasing probability. 41% of all calls made by that subject (or 61% of the recurring calls, $\mathcal{R} = 67\%$) repeated one of the previous 10 dialed.

Information retrieval is another recurrent system. Intuitions about the recurrence rate of such systems are perhaps not so immediate as with telephone access. Still, a few arguments for suspecting recurrences follow. First, it is usually difficult to remember particular details of information retrieved, especially if it is obscure, technical or numerical in nature. Retrieval recurrences over short time periods is therefore likely, since details of a document require constant review. Second, different information fragments are not sought equally. People may recall "important" information fragments repeatedly over long time periods. Finally,

previously acquired information may become stale. As information is rarely static, the same question may be posed repeatedly and the answer checked for changes. Airline arrival and departure information available through broadcast and teletext environments is one example of dynamic information. Another example is the slowly changing standards described in technical manuals, which become obsolete over time.

Empirical evidence supports the contention that manuals are recurrent systems. An analysis of work logs of Boeing engineers showed that up to $\mathcal{R} = 70\%$ of all lookups of hardcopy manuals (*eg* standards, product manuals) were to specific things the engineers had seen before but had forgotten (reported in Dumais & Landauer 1982). The high figure is perhaps not surprising in retrospect, for technical details found in engineering manuals do not lend themselves to easy recall.

Topic retrieval in computer-based technical manuals is also characterised by high recurrences. All retrievals of topic pages in the UNIX on-line manual by students and employees in a Computer Science department were collected for one month (Bramwell, 1983) and analyzed for recurrences (Greenberg, 1984). A total of 4978 correct retrievals were made by 443 users. The salient findings follow.

1. The recurrence rate of retrievals was generally high, approaching an average of $\mathcal{R} = 50\%$ for each user after relatively few retrievals.
2. Moderate variation in the recurrence rate was noted between individuals. For example, users who had made between 17 – 19 retrievals had a standard deviation of 17.1% over the average rate of $\mathcal{R} = 45.2\%$ for that class. Extremes were 12% and 71%.
3. Each user retrieved only a small subset of the topics available.
4. Few common retrievals were noted between users, even when user tasks were similar.
5. The shape of the frequency distribution of the topics retrieved by an individual varied substantially from user to user. The general trend was for each user to access most of their retrieved topics between one and three times, with a smaller set being called on more often.

In general, we conclude that retrieving topics in technical manuals is a recurrent system. It seems likely that these results generalize to most structured documents, such as those found in hypertext systems, and to general information retrieval facilities provided by standard data bases.

3 Unix as a recurrent system

This section investigates recurrences exhibited by subjects using the UNIX command interpreter *csh*. After introducing some terminology and describing the method of data collection, previous studies on how people enter UNIX *commands* (as opposed to complete *command lines*) are reviewed. However, we believe that

studies of commands only have 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, two questions particularly relevant to reuse facilities are addressed, both concerning the statistics of complete command lines entered by the user to UNIX. First, we examine how often a user actually repeats command lines over the course of a dialog. 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, the probability distribution that the next command line will match a previous input is described. This distribution is measured as a function of the number of entries that have elapsed from the matched input to the current one.

3.1 Definitions

In the following discussion, a *command line* is a single complete line (up to a terminating carriage return) entered by the user. This is a natural unit 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 activity than just the command itself. Command lines typically comprise an action (the command), an object (*eg* files, strings) and modifiers (options). The *command* is the verb of the command line entered. A sequential record of command lines entered by a user over time, ignoring boundaries between login sessions, is known as a *history list*. Erroneous submissions noticed by *csh* are not included. Unless stated otherwise, the history list is a true sequential record of every single command line typed. Duplicate activities, for example, are included. The *distance* between two lines is the difference between their positions on the list. A *working set* is a small subset of items on the history list. The number of different entries in the history list is the command line *vocabulary*. Although white space is ignored, syntactically different but semantically identical command lines are considered distinct.[1]

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.

[1] 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.

Subject use. 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.

—Table 1 around here—

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 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. Knowledge of UNIX was the minimum necessary to get the job done.

Since users were assigned to subject groups only through their membership in identifiable user groups (*eg* 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 (Joy, 1980) 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 VDU's. In addition, Corvus Concept workstations running the Jade Window Manager were available to members of the Experienced and Computer Scientist groups. These allowed users to create many "virtual terminal" windows, each running *csh*, on a single screen (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 number 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 form the line were ignored. Extra information known to *csh* was trapped and recorded as well by placing "hooks" within *csh*. History use and alias use are noted, as well as the current working directory of the user and the error status after execution is attempted. More specifically, login sessions are distinguished by a record that notes the start and end time of each session. Records associated with each user input are then listed subsequently, each annotated with the command line, the current working directory, alias substitution (if any), history use and error status. The total and average number of command lines collected (excluding errors) are listed in Table 1.

Data Selection. If subjects did not log in at least ten times and execute at least 100 commands during the study period, their data was not considered. By these criteria, 12 of the 180 original 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 cartridge tape of the data (Greenberg, 1988). As all subjects were

promised confidentiality, data is massaged to remove the identity of subjects.

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

First, due to implementation difficulties, the details of history directives 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 directive used to produce the modification.

Second, and more seriously, not all user activity is captured. Although recording *csh* lines works well for "batch" style programs that execute and return without user intervention (*ie* incremental interaction), it does not capture activity within the interactive applications used (*eg* editors). Interactive information is lost since collected data concerns the *csh* command line only. Also, commands can be very different. For example, consider a trace containing only two UNIX commands: *ls* for listing files; and *emacs* which invokes a sophisticated interactive editor. Whereas file listing is accomplished almost immediately, an editing session can last for hours. This distinction is not captured here.

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 is 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 Greenberg, 1989 Chapter 2 for a comparison of methods).

3.3 Recurrence of commands

Several independent researchers have studied how individual commands in UNIX—the verbs of the command lines—are selected. The significant results are summarized here.

Frequency distribution of commands for large groups. Many investigators have examined the frequency of command usage by a user population (Peachey *et al*, 1983; Hanson, Kraut & Farber, 1984; Ellis & Hitchcock, 1986; Greenberg & Witten, 1988a). 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 (Zipf, 1949; Witten, Cleary & Greenberg, 1984). 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 *et al*, 1982). Hanson *et al* (1984), for example, state that 10% of the 400-500 commands available account for 90% of the usage. Greenberg and Witten (1988a), 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 commands over the complete sample is misleading. Greenberg and Witten (1988a) contrasted command use between the four groups identified in this study, and data from Hanson's similar study (Hanson *et al*, 1984). They found that commands do not necessarily retain their same rank order between different user groups. Although there are several frequently used commands in common for all groups.[2] the comparison emphasised 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 few of each individual's command vocabulary[3] were used by all the population, a few more shared to some degree by other users, and the rest used by the individual user alone. Greenberg and Witten (1988a) pursued this matter further. They found that only 0.2% (*ie* 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 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 he employed. Greenberg and Witten (1988a) actually observed vocabulary acquisition by particular users, and noted that the vocabulary growth rate was quite slow: $\mathcal{C} = 1\%$ or less ($\mathcal{R} = 99\%$). Vocabulary growth was by no means regular—they observed long periods of quiescence followed by a flurry 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).

[2]The shared commands mostly concerned the basic Unix commands for navigating, manipulating and finding information about the file store.

[3]Commands in his study were the processes actually executed.

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 *et al* (1984) examined the interaction effects between commands. They noted that some commands follow or congregate around others in quite regular ways. But the dependencies and clustering observed may be an artifact 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'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 $\mathcal{R}$, command use is not, strictly speaking, a recurrent system, for new activities are not incorporated regularly ($\mathcal{C}$ 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 *sort file* and *sort file | uniq -c | sort -r*. The first just sorts a file, while the second produces a frequency count of the identical lines in the file. 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, analyzed here in terms of commands, is re-analyzed next in terms of command lines.

3.4 Recurrences of command lines

As mentioned above, command use is not really a recurrent system since $\mathcal{C}$ is so low. A different question is whether complete command lines submitted to general-purpose 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 reuse facilities 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 (*ie* 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 is calculated as described in Section 2.1.

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 $\mathcal{R}$ could be confounded by the large variation between the number of command lines each user enters, which was noted in Table 1. As $\mathcal{R}$ 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 ($r = .918$, $df = 167$, $p < .01$). The moderate slope ($\mathcal{C} = 23\%$) of the regression line makes the correlation practically significant as well.

—Figure 1 around here—

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 $\mathcal{R}$, where each point represents the recurrence rate observed for each subject at the end of the study. A statistically significant correlation was found ($r = .253$, $df = 167$, $p < .01$), indicating that the recurrence rate increases with the number of commands entered. However, the high variance of data points around the line ($r^2 = .064$), and its low slope (0.002), makes this finding insignificant for practical purposes. Consequently, $\mathcal{R}$ 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.

—Figure 2 around here—

Table 2 lists the mean recurrence rate, standard deviation, and ranges of $\mathcal{R}$ for each subject group. An analysis of variance of raw scores rejects the null hypothesis that these means are equal ($F(3,164) = 21.42, p < .01$). The Fisher PLSD multiple comparison tests suggests that all differences between group means are significant ($p < .01$), excepting the Non-programmers versus Scientists. As the Table indicates, the mean recurrence rate for groups ranges between 68% and 80%, with Novice Programmers exhibiting the highest scores.

—Table 2 around here—

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 $\mathcal{R}$ over all users to be 75% and $\mathcal{C}$ of 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.

3.5 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 (*eg* HISTMENU, Bobrow 1986), what is the probability that this includes the next entry?

The recurrence distribution as a measure of distance was calculated for each user. First, let $\mathcal{R}_{s,d}$ 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 values of $\mathcal{R}_{s,d}$ 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:

$$\mathcal{R}_d = \frac{1}{S}\sum_{s=1}^{S}\mathcal{R}_{s,d}$$

These group means are plotted in Figure 4a.

—Figure 3 around here—

—Figure 4 around here—

The vertical axis represents $\mathcal{R}_d$, 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 a $\mathcal{R}_{d_1} = 11\%$ probability that the current command line is a repeat of the previous entry (distance = 1), $\mathcal{R}_{d_2} = 28\%$ for a distance of two, $\mathcal{R}_{d_3} = 9\%$ 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 $\mathcal{R}_d$ 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 4b, 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 $\mathcal{R}_D$. Its mean value over a group of subjects is calculated as

$$\mathcal{R}_D = \frac{1}{S} \sum_{s=1}^{S} \sum_{d=1}^{D} \mathcal{R}_{s,d}$$

The most recently entered command lines on the history list are responsible for most of the cumulative probabilities. For example, there is a $\mathcal{R}_{D_{10}} = 47\%$ chance that the next submission will match a member of a working set containing the ten previous submissions ($\mathcal{R}_{D_{10}}$ is an abbreviation of $\mathcal{R}_D$ 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 $\mathcal{C} = 26\%$ of all command lines entered are first occurrences.

Figure 4a also shows that the differing recurrence rate 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.

Summary The last two sections introduced the notion of recurrent systems, and provided empirical evidence of their existence in both natural and computer domains. The three diverse examples studied—telephone usage, information retrieval, and command-line interfaces—show remarkable similarity in the way activities are repeated. All satisfy the characterization of recurrent systems set out in Section 2.1.

The statistics of UNIX *csh* use, and to lesser extent telephone dialing, 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 of recency made by history mechanisms.

4 Reuse Opportunities 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 $\mathcal{R}_{D_{10}} = 47\%$ chance that the next command line can be successfuly predicted by this display (Figure 4b), there is still a 27% chance that it last appeared further back. When combined with the $\mathcal{C} = 26\%$ 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 betters 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.

4.1 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.

The metric for work introduced here is called M_D, and comprises two components that estimate a prediction's quality. The first is $\mathcal{R}_D$, the probability that the desired item appears on a displayed list of length $p = D$. Its calculation was given in Section 3.5. The second, called $\overline{c}_d$, 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 $\overline{c}_{s,d}$ be the average number of characters saved by a subject s per recurrence at distance d, calculated as:

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

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 $\overline{c}_{s,d}$ is averaged over all subjects S, we get $\overline{c}_d$, calculated as:

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

But $\overline{c}_d$ 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:

$$M_d = \frac{1}{S}\sum_{s=1}^{S}\overline{c}_{s,d}\mathcal{R}_{s,d}$$

where $\mathcal{R}_{s,d}$ is a particular subject's probability of a recurrence at the given distance, defined in Section 3.5. Note that M_d differs from $\overline{c}_d$ because it is the average savings *per submission* rather than per recurrence. The final step in calculating M_D shows the cumulative average savings in characters per submission when one through D predictions are available for selection:

$$M_D = \sum_{d=1}^{D} M_d = M_{d_1} + M_{d_2} + \ldots + M_{d_D}$$

Both $\mathcal{R}_D$ and M_D will be reported in this paper as metrics for evaluating working sets of particular sizes. Values of $\mathcal{R}_d$ (defined in Section 3.5) and $\overline{c}_d$ are included for reference.

4.2 Different conditioning methods

A variety of conditioning methods are described here. As well as conditions that are expected to perform quite well, weak ones that have been implemented in existing reuse facilities are also included. For each method we indicate how the recorded data will be analyzed to assess its effectiveness. The algorithms used to find $\mathcal{R}_{s,d}$ for each case are not elaborated (they are minor variations of the one shown in Figure 3). Results are presented in the next section that show how effective—or ineffective—these conditioning methods really are.

Sequential ordering by recency. This conditioning method was described in the previous section, and is simply a time-ordered list of all submissions entered by the user. The first column of Table 3 illustrates the sequentially-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 the Table—is intended to be reviewed top-down.

—Table 3 around here—

There are two virtues of using recency 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 adaptive methods, there is no initial startup instability of deciding what to present 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 which could more fruitfully be used 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 MINIT's history system that combines command processing and history into a single "window management window" called *wmw* (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 entries will migrate to the back of the list over time.[4]

Consider, for example, the two pruned event lists in the second major column of Table 3. 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 frequently used *ls* command line is poorly positioned at the bottom of the list.

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

[4] Saving recurrencing 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.

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 1 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. Examples are shown in Table 3, 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 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 (*cf* recency).

The data sets are analyzed 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 are 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. One example is MINIT's *wmw*, 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-analyzed by placing their command lines on a history list in ASCII order. If a new submission is identical to one already on the list, it is ignored. An example of an ASCII-ordered list is included in Table 3.

Context-sensitive history lists by directory. Users of computer systems perform much task switching (Bannon, Cypher & Greenspan, 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 hypothesise 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 one's 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 are 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 is 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 Table 3 illustrates the directory-sensitive condition applied to the sequential input, where each sub-column is sensitive to a particular directory. Most command lines refer to files in that directory, and would rarely be used in other directories. Some command lines, however, are common to more than one directory (for example, *ls* for listing files).

Ordering commands by recency. Section 3.3 showed that most individuals use few 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 $\mathcal{R}_D$ 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 Table 3).

User traces are re-analyzed over history lists of commands. Duplicate commands are 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, 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 are 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 (called a command-sensitive list). 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 Table 3, 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 Table 3 shows one such possibility, where the event list is conditioned by directory sensitivity and pruning. Data sets were re-analyzed using combinations of a few conditions mentioned above.

4.3 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.[5] 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 is re-analyzed 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 are not included.

Since reuse facilities can only predict lines that have been entered previously, it is 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 $5.97\mathcal{R}/100$, for M_D is calculated over all submissions. As $\mathcal{R}$ is 74.4% for experienced programmers, M_D for an optimal conditioning method is 4.43 characters predicted per submission.

Results. Results for all conditions are summarized in four tables, each presenting various distributions over the last fifty items of the history list. Table 4 presents the percent frequency of submissions recurring at a particular distance ($\mathcal{R}_d$), while Table 5 provides the same information as a running sum over distance ($\mathcal{R}_D$). The latter includes the total recurrence rate over the complete history list, which differs with certain

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

conditions.[6] Figure 5 graphs the results of Table 5. As with Figure 4 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 $\mathcal{R}_D$, the rate of accumulated command line recurrences, as a percentage.

The next two tables involve the length in characters of recurrences. Table 6 shows the average number of characters saved for a recurrence at a given distance (the value of $\overline{c}_d$). Table 7 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 6 graphs the performance of each conditioning method over distance using this metric.

—Table 4 around here—

—Table 5 around here—

—Figure 5 around here—

—Table 6 around here—

—Table 7 around here—

—Figure 6 around here—

Standard sequential. $\mathcal{R}_{D_{10}}$ is 44.4% for the experienced programmer group (Table 5), which is around 60% of the maximum value it could have ($\mathcal{R} = 74.4\%$). The metric $M_{D_{10}}$ for the same group is 2.48 characters per submission (Table 7), 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 (*ie* 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 gave a much higher average length of predicted strings than any other conditioning method for lines recurring over small distances (Table 6). But it is the 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

[6] 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).

hypothesis that oft-repeated lines are shorter on average than rarely repeated ones. The low probability values associated with those recurrences reduce any benefit accrued by predicting longer lines. Consider a 10-item working set. The probabilities $\mathcal{R}_{D_{10}}$ 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 $\mathcal{R}_D$ associated with a duplicates-pruned list when compared to the standard sequential list (Table 5 and Figure 5). Pruning duplicates increases the overall probability of a ten-item working set by 4.8% ($\mathcal{R}_{D_{10}}$ = 49.1% 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 of a ten-item working set 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 ten-item working set is about the same ($\mathcal{R}_{D_{10}}$ = 44.4%), 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 ten-item display, $\mathcal{R}_{D_{10}}$ = 10.1%, and only 0.65 characters are predicted per submission. If a user were scrolling through this display, fully 100 items (or ten 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. As Table 4 illustrates, 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 $\mathcal{R}_D$ cross over with a working set of twenty-seven items (Figure 5). With a working set of ten items, directory-sensitivity increases the overall probability that the next item will be in that set by 2.5% ($\mathcal{R}_{D_{10}} = 46.9\%$). The length of lines predicted in the directory sensitive condition are also longer than those 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: $\mathcal{R}_{D_{10}} = 72.7\%$ *vs* 44.4%. But 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 5 and Figure 5. Conditioning by partial matching increases $\mathcal{R}_{D_{10}}$ of a ten-item working set by 6.4% when compared to a strict sequential list (Table 5), although lines predicted are shorter (Table 6). 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 ten-item display is $\mathcal{R}_{D_{10}} = 55.5\%$ 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 4 through 7 and Figures 5 and 6. Where feasible, conditioning methods can be combined even further. For example, a partially-matched, pruned and directory sensitive history mechanism increases $\mathcal{R}_{D_{10}}$ over a strict sequential one by 12.7% with a working set of ten items (reported in Greenberg & Witten, 1988b).

4.4 Discussion

The recurrence rate $\mathcal{R}$ 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 ten predictions for literal matches, depending upon the conditioning method chosen. Given that $\mathcal{R} = 75\%$ on average, which is the best a perfect literal reuse facility could do, this means that the best predictive method described here is about 75% effective, at least potentially.

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% effective.

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 (Whiteside, Archer, Wixon & Good, 1982). 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 describe 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 $\mathcal{R}$). One candidate could use the model similar to that employed by the *Reactive Keyboard* (Darragh, 1988), an adaptive system that bases its prediction on a frequency-based Markov model (summarized in Greenberg & Witten 1989). 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. 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.

4.5 Corroboration

The preceding discussion was based on the UNIX findings. As there is no guarantee that they generalize to all recurrent systems and applications, it is useful to see if studies of other systems would produce the same results.

Data on 20 heavy users of the GLIDE functional programming language and environment (Toyn, 1987). became available after we had completed the UNIX study (Beale, Finlay, Austin & Harrison, 1989). The analysis performed was similar to the UNIX one described previously, although not nearly as extensive (see Greenberg, 1989 for a detailed description of the GLIDE study).

The GLIDE analysis corroborates the findings of the UNIX study. The most glaring difference is a lower recurrence rate (50% versus 75%). Part of this difference could arise from the fact that arguments in GLIDE functions are lists. Since lists are generally not as persistent as filenames, arguments (and their lines) would not recur as often. A small part of this difference could be an artifact of data collection, for white space and errors were handled differently.

When conditioning methods were applied to GLIDE data, they followed the same rank ordering as that produced by *csh* use. Although there are fewer recurrences with GLIDE, the predictive power of the conditioning methods is relatively greater. For example, up to 43% of all user activity can be successfully predicted with working sets of ten predictions. Given that $\mathcal{R} = 50\%$, which is the best a perfect reuse facility could do, the top-ranked predictive method is 85% effective for GLIDE recurrences (*cf* 75% for UNIX). When the quality metric is incorporated, up to 4.43 characters are submitted per submission when 10 predictions are available. Since the maximum value of M_D found for GLIDE is 4.87 characters, the best method is about 90% effective (*cf* 75% for UNIX).

In summary, despite the numeric differences in the analyses, the general findings of the UNIX study are corroborated by subjecting GLIDE to the same analysis. Although new activities are composed regularly by users (around 50%), a substantial portion of their activities are repeated (50%). Users exhibit considerable recency in activity reuse in the same way as UNIX. The major contributions are provided by the previous 7 ± 3 submissions, and the second to last command line recurs more often than any other input. Although some user activities still remain outside a small working set containing the recent submissions, the predictive power of these sets can be improved by suitable conditioning. Command-sensitive sublists remain particularly

effective.

5 Actual Use of Unix History

We have seen that user dialogs are highly repetitive and the last few command lines have a high chance of recurring— the premise behind most history systems. There are certainly plenty of opportunities for reuse, especially when appropriate conditioning methods are engineered into the presentation of items. But are current history mechanisms used well in practice? And how are they used? In this section, we investigate how well the reuse facilities supplied by the UNIX shell are used in practice (Joy 1980; summarized in Greenberg & Witten 1989). This was investigated by analyzing each user's *csh* history use. During data collection, all *csh* history uses were noted, although the actual form of use was not. Results should be interpreted carefully, for they may be artifacts arising from idiosyncrasies of the *csh* facilities, rather than from fundamental characteristics of reuse.

The recurrence rate and its probability distribution, studied previously, give a theoretical value against which to assess how effectively history mechanisms are used in practice. The average rate of re-selecting items through a true sequential history list (as used by *csh*) cannot exceed the average value of $\mathcal{R}$. By comparing the user's actual re-selection rate with this maximum, the practical effectiveness of a particular history mechanism can be judged.

5.1 Results

Table 8 shows how many users of UNIX *csh* in each sample group actually used history. Although 54% of all users recalled at least one previous action, this figure is dominated by the computer sophisticates. Only 20% of Novice Programmers and 36% of Non-Programmers used history, compared to 71% for Computer Scientists and 92% for Experienced Programmers.

—Table 8 around here—

Those who made use of history did so rarely. On average, 3.9% of command lines referred to an item through history, although there was great variation (*std dev* = 3.8; *range* = 0.05% *to* 17.5%). This average rate varied slightly across groups, as illustrated in Table 8, but an analysis of variance indicated that differences are not statistically significant ($F(3, 86) = 1.02$).

In practice, users did not normally refer very far back into history. With the exception of novices, an average of 79 - 86% of all history uses referred to the last five command lines. Novice Programmers achieved this range within the last two submissions. Figure 7a illustrates the nearsighted view into the past. Each

line is the running sum of the percent of history use accounted for (the vertical axis) when matched against the distance back in the command line sequence (the horizontal axis). The differences between groups for the last few actions (left-hand side of the graph) reflect how far back each prefers to see.[7]

—Figure 7 around here—

Since most activities revolve around the last few submissions, the distribution bears closer examination. The data points in Figure 7b now represent the percent of history use accounted for by each reference back. High variation between groups is evident. Although most uses of history recall the last or second last entry, it is unclear which is referred to more often.

It was also noticed that history was generally used to access or slightly modify the same small set of command lines repeatedly within a login session. If history was used to recall a command line, it was highly probable that subsequent history recalls will be to the same command.

A few *csh* users were queried about history use. They indicated that they are discouraged from using *csh* history by its difficult syntax and the fact that previous events are not normally kept on display. (The latter point is important, for it enforces the belief that candidates for reuse should be kept visible.) Users also stated that most of their knowledge of UNIX history was initially learnt from other people—the manual was incomprehensible. Also, the typing overhead necessary to specify all but the simplest retrievals makes them feel that it is not worth the bother.

5.2 Corroboration and extensions

Another researcher, Alison Lee, also examined history usage within various command interpreters available to the UNIX environment. Some of her qualitative findings corroborate and add to the observations noted in this section (Lee, 1988).

1. There were very few uses of *csh* history.
2. Those uses made were of the simpler features, the most popular being "!!" (retrieve the last event) and "!pattern" (retrieve the most recent event beginning with the given pattern).
3. People rarely retrieved items by absolute or relative event number.
4. Although the history list is available for viewing by special request, users rarely asked to see it.

[7] Actual figures are probably higher than those indicated here, due to inaccuracies in distance estimates. As the *csh* monitor only noted that history was used and not how it was used, the actual event retrieved was determined by searching backwards for the first event exactly matching the current submission. If the submission was a modified form of the actual recalled event, the search would terminate on the wrong entry. These are assumed to be a small percent of the total.

5. Modifiers for editing were rarely used. When used, they tended to be of the form $\wedge pattern1 \wedge pattern2 \wedge$, which does simple substring replacement on the previous submission.

6. Other observed ways of modifying events were by using recalled events as prefixes or suffixes. This technique allows one to add more parameters to previous events or to add a new command sequence in a pipeline.

7. Occasional instances were noted of recalling the last word in the previous event (ie !$) and of printing events without executing them.

Lee also examined *tcsh*, another history mechanism available to UNIX users that uses a very simple and familiar *emacs*-like editing paradigm to retrieve, review and edit previous events. Although better use of history is expected due to the improved editing power and visualization of the history list, only a marginal increase was noted (although the still-available *csh* history was used less). The visual scrolling and editing capabilities available in *tcsh* were used to some extent.

5.3 Discussion

Many people never use UNIX *csh* history. Those who do tend to be sophisticated UNIX users. Yet even they do not use it much. On average, less than 4% of all submissions were retrieved through history out of the 74% potentially possible! The history facility supplied by *csh* is obviously poor.

Some reasons for the failure of *csh* history follow. First, the complex and arcane syntax discourages its use. Those who did use history indicated that only the simplest features of UNIX history were selected. As one subject noted, "it takes more time to think up the complex syntactic form than it does to simply retype the command." Also, it takes at least two characters to recall an event in *csh*. As most simple UNIX recurrences are short (6 characters on average), users may feel that it is not worth the bother. Second, it is hard to find out about it. *Csh* history details are buried in a single on-line manual entry that runs to thirty-one pages(!), the text is quite technical, and examples are sparse. Third, the event list is usually invisible. As previous events are not normally kept on display, frailty of human memory usually limits recall to the last few items. These deficiencies of *csh* hit Novice Programmers especially hard. Even though they have the highest recurrence rate of all groups and could benefit the most from history, they are effectively excluded from using it.

It is too soon to condemn the ideas provided by *csh*, because some of the observations are likely artifacts of using a poorly designed facility, rather than a human difficulty with the idea itself. Still, it is worthwhile reviewing some of the common retrieval methods it provides.

Retrieval through absolute or relative position. It is fairly difficult to associate and remember the

number of a previous event, as it is an indirect reference. Visibly tagging events with numbers offers benefit only for those interfaces without direct selection and only when no better strategy is available. Perhaps its only viable use is as a redundant way of retrieving events when other selection methods are available.

Scrolling and hidden views. If events are not on display, they will not be asked for. Hidden history lists were rarely recalled, and little use was made of the scrolling facilities in *tcsh*.

Pattern matching. Simple pattern matching, especially by prefix specification, seems promising as a textual way of retrieving events. But matching is potentially dangerous, as users may accidentally retrieve and execute an interposed but undesired event that fits the specification.

Simple methods for recall/selection of very recent events. The syntactically simplest methods are used most to recall very recent events. For example, the "!!" directive was heavily used, even though it does not recall the most probable event. This likely reflects the shortness of short-term memory—users only use "!!" because the last item is the only thing they can both remember reliably and retrieve quickly. Overloading a reuse facility with complex functionality would not make it better.

Editing events. Although people do edit command lines as they compose them, they may not be willing to modify previous events much. Often the cognitive and physical overhead of recall and editing previous events makes simple re-entry more effective. Still, some editing does occur and probably has some value.

6 Principles: How Users Repeat Their Activities

The preceding sections analyzed command line recurrence in dialogs with the UNIX *csh*. Based on the empirical results, the first part of this section formulates general principles which characterize how users repeat their activities on computers. 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 Table 9 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 un-informed intuition.

—Table 9 around here—

6.1 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 2.1.

This high degree of repetition justifies the intent of reuse facilities. Recurring inputs should be re-entered more easily than the user's original entry, with the aim of reducing both physical tedium and the cognitive overhead of remembering past inputs. Reuse facilities should not be targetted only to experts—they can help everyone.

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

Many modern interfaces provide transient 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, 1989). 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 dialog 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-D 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, a user would have to search through two previous input and output sequences before finding the second to last entry.

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 ten events. HISTMENU's graphical

menu of history items for example, defaults to 51 items (Bobrow, 1986), and MINIT's *wmw* 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 possibilities, a lengthy list is unlikely to be worthwhile. For example, a menu of the previous ten UNIX events covers, on average, 45% of all inputs. Doubling this to twenty 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 & Witten, 1989 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 ten 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 (1989).

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 ten 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-Ds Programmers Assistant, Xerox 1985), pruning may destroy useful sequences. And events no longer follow the true execution order, perhaps confounding attempts 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 too.

When command-sensitive sublists are included but ignored, the potential for reuse is still as 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 considerably more mechanical overhead for reuse unless they are on permanent display, and even then there is a cognitive overhead.

Some seemingly 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 applications. 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, and is a presentation option in MINIT's *wmw* (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. 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. 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.

6.2 Recurrences: natural fact or artifact?

Where do recurrences come from? Are they naturally part of a human-computer dialog or are they artifacts 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 interactional 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 emacs Chapter 1	*repeats 4 times*
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	*repeats 11 times*
q5.o	*repeats 3 times*
fred ada –M concur –o q5.o q5.a q5.o	*repeats 6 times*

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, 1985). 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.[8]

In the case of debug cycles, it is certain that some recurrences are artifacts 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.

But 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 dialing. 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 is another example of recurrences that arise from repetition of our intentions rather than from interface artifacts. Or consider navigation on computers where people must locate and traverse the many structures necessary for their current context (*eg* 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.

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

[8] 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.

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 artifacts 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 artifacts. For the present, reuse facilities remain a potentially viable and very general way of handling repetition.

7 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 dialogs. As a consequence, existing reuse facilities—as surveyed in Greenberg and Witten (1989)—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 suspected recurrent systems from both non-computer and computer domains were examined 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 ten predictions. Since the best any literal predictive method could do is $\mathcal{R} = 75\%$ 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. 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 artifacts of using a poorly designed facility, rather than a human difficulty with the idea of reuse.

All findings above were used to derive 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. 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, 1989). 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 them 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., and Monty, M. (1983) Evaluation and analysis of users' activity organization. In *Proceedings of the ACM SIGCHI '83 Human Factors in Computing Systems*, pages 54–57, Boston, December 12-15.

Barnes, D. and Bovey, J. (1986) Managing command submission in a multiple-window environment. *Software Engineering Journal*, 1(5):177–183, September.

Beale, R., Finlay, J., Austin, J., and Harrison, M. (1988) User modelling by classification: a neural-based approach In Taylor, J. and Mannion, C. (1988), editors, *New developments in neural computing*, pages 103–110. IOP Publishers.

Bennett, J. (1975) Storage design for information retrieval: Scarrott's conjecture and Zipf's law In Gelenbe and Potier (1975), editors, *International Computing Symposium 1975*, pages 233–237, Amsterdam, June 2-4. North-Holland.

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

Bramwell, B. (1988) An automatic manual. Master's thesis, Department of Computer Science, University of Calgary, Alberta.

Darragh, J. (1988) Adaptive predictive text generation and the Reactive Keyboard. Master's thesis, Department of Computer Science, University of Calgary, Alberta, September.

Draper, S. (1984) The nature of expertise in Unix. In *Interact '84 — First IFIP Conference on Human-Computer Interaction, 2*, pages 182–186, London, UK, Sept 4-7.

Dumais, S. and Landauer, T. (1982) Psychological investigations of natural terminology for command and query languages In Badre and Shneiderman (1982), editors, *Directions in human/computer interaction*, pages 95–110. Ablex Publishing Co., Norwood, New Jersey.

Ellis, S. and Hitchcock, R. (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(3):423–427, May/June.

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

Greenberg, S. (1984) User modeling in interactive computer systems. Master's thesis, Department of Computer Science, University of Calgary, Alberta.

Greenberg, S. (1988) Using Unix: Collected traces of 168 users. Research report 88/333/45 plus tar-format cartridge tape, Department of Computer Science, University of Calgary, Calgary, Alberta, Canada.

Greenberg, S. (1989) *Tool use, reuse, and organization in command-driven interfaces.* PhD thesis, Department of Computer Science, University of Calgary, Calgary, Alberta, Canada. also available as Research Report 89/336/48.

Greenberg, S., Peterson, M., and Witten, I. (1986) Issues and experiences in the design of a window management system. In *Proceedings of the Canadian Information Processing Society Edmonton Conference*, pages 33–50, Edmonton, Alberta, October 21-23.

Greenberg, S. and Witten, I. (1985) Adaptive personalized interfaces — a question of viability. *Behaviour and Information Technology*, 4(1):31–45.

Greenberg, S. and Witten, I. (1988a) Directing the user interface: how people use command-based systems. In *Proceedings of the 3rd IFAC Conference on Man-Machine Systems*, Oulu, Finland, June 14-16.

Greenberg, S. and Witten, I. (1988b) How users repeat their actions on computers: principles for design of history mechanisms. In *Proceedings of the ACM SIGCHI '88 Human Factors in Computing Systems*, pages 171–178, Washington, D.C., May 15-19.

Greenberg, S. and Witten, I. (1989) A survey of reuse facilities. *International Journal of Man Machine Studies.* Submitted paper.

Hanson, S., Kraut, R., and Farber, J. (1984) Interface design and multivariate analysis of UNIX command use. *ACM Transactions on Office Information Systems*, 2(1), March.

Joy, W. (1980) *An introduction to the C shell, volume 2c.* Unix Programmer's Manual. University of California, Berkely, California, seventh edition.

Knuth, D. (1973) *The art of computer programming: searching and sorting.* Addison-Wesley.

Lee, A. (1988) Study of command usage in three UNIX command interpreters. In *Interactive Poster in the ACM/SIGCHI 1988 Conference on Human Factors in Computing Systems*, Washington, D.C., May 15-19.

Peachey, J., Bunt, R., and Colbourn, C. (1982) Bradford-Zipf phenomena in computer systems. In *Proc Canadian Information Processing Society National Conference*, pages 155–161, Saskatoon, Saskatchewan, May.

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

Ross, P., Jones, J., and Millington, M. (1985) User modelling in command-driven systems. DAI Research Paper 264, Department of Artificial Intelligence, University of Edinburgh.

Suchman, L. (1985) Plans and situated actions: The problem of human-machine communication. PhD Thesis ISL-6, Intelligent Systems Laboratory, Xerox PARC Research Center, Palo Alto, California.

Sutcliffe, A. and Old, A. (1987) Do users know they have user models? Some experiences in the practice of user modelling In Bullinger, H. and Shackel, B. (1987), editors, *Human-computer interaction — Interact '87*, pages 35–41. Elsevier Science Publishers B.B. (North Holland).

Thimbleby, H. (1990) *The user interface design book.* Addison Wesley.

Toyn, I. (1987) *Exploratory environments for functional programming.* PhD thesis, Department of Computer Science, University of York, Heslington, York, February.

Whiteside, J., Archer, N., Wixon, D., and Good, M. (1982) How do people really use text editors? In *Proceedings of the ACM SIGOA Conference on Office Information Systems*, pages 29–40, June.

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

Zipf, G. (1949) *Human behaviour and the principle of least effort.* Addison-Wesley, Ontario.

List of Figures

List of Tables

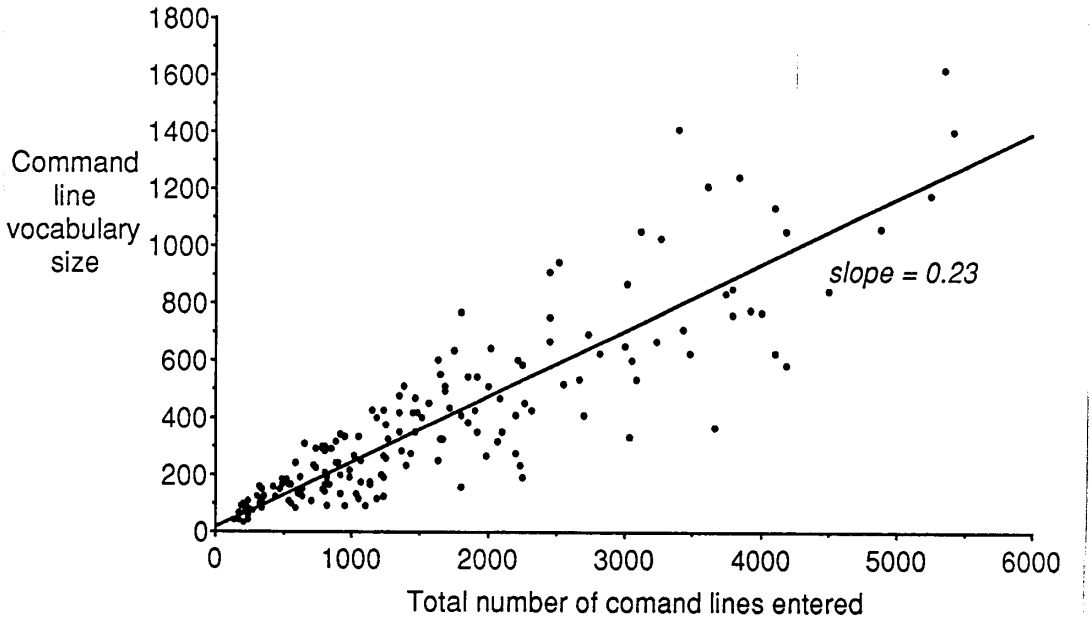

Figure 1: Regression: Command line vocabulary size versus the total commands lines entered by each subject

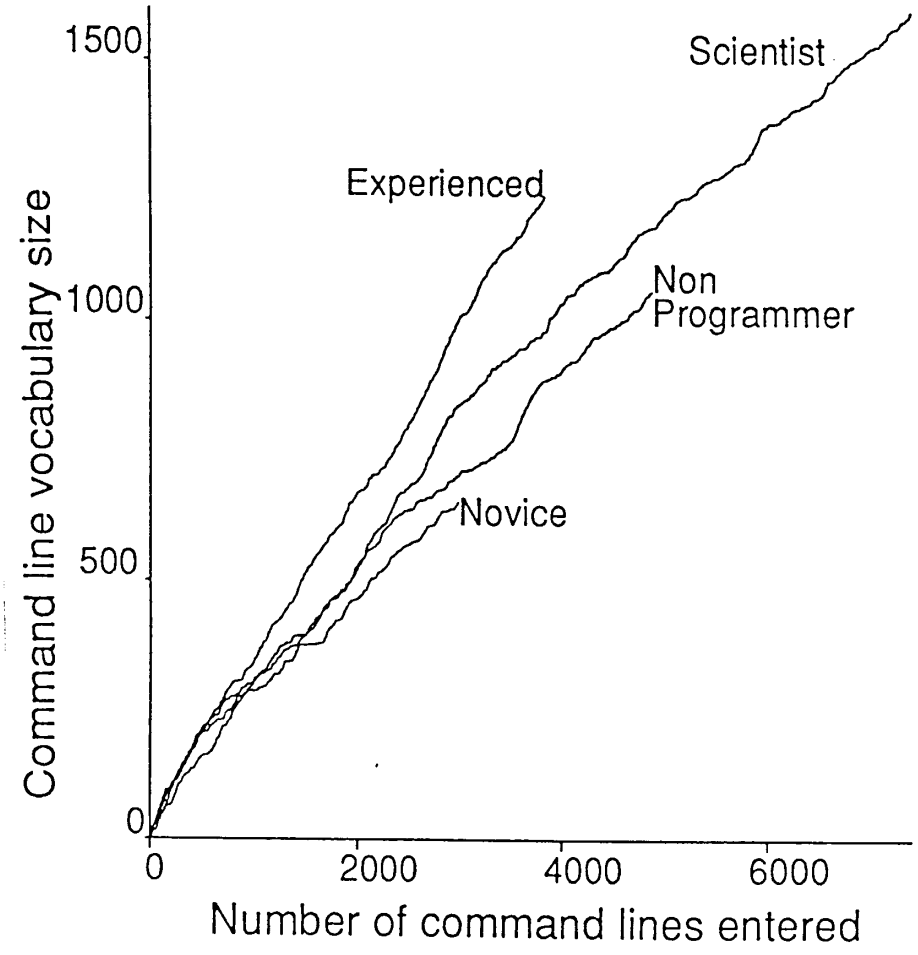

Figure 2: Command line vocabulary size versus the number of commands entered for four typical individuals

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 $\mathcal{R}_{s,d}$

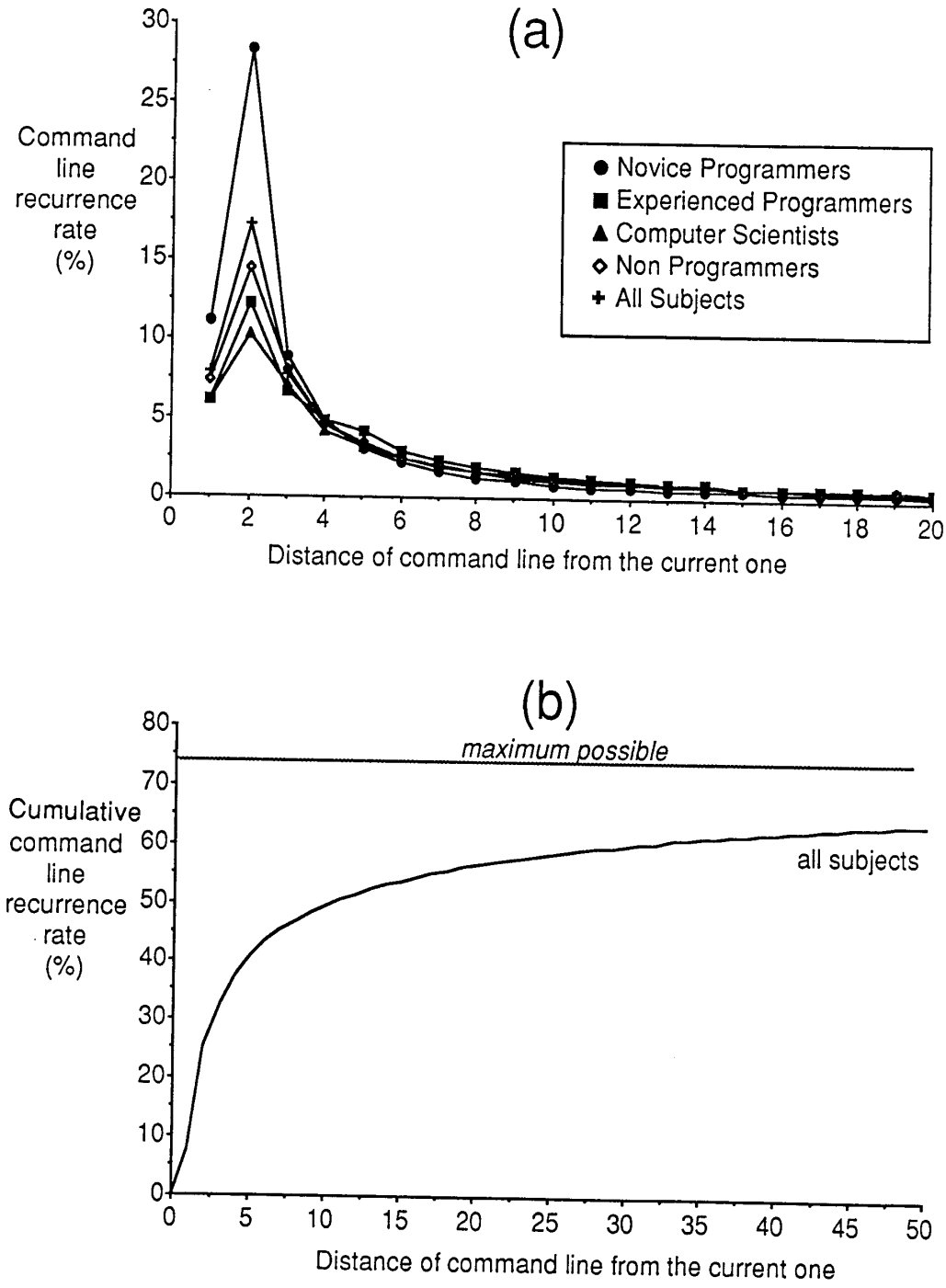

Figure 4: a) Recurrence distribution; and b) cumulative recurrence distribution as a measure of distance

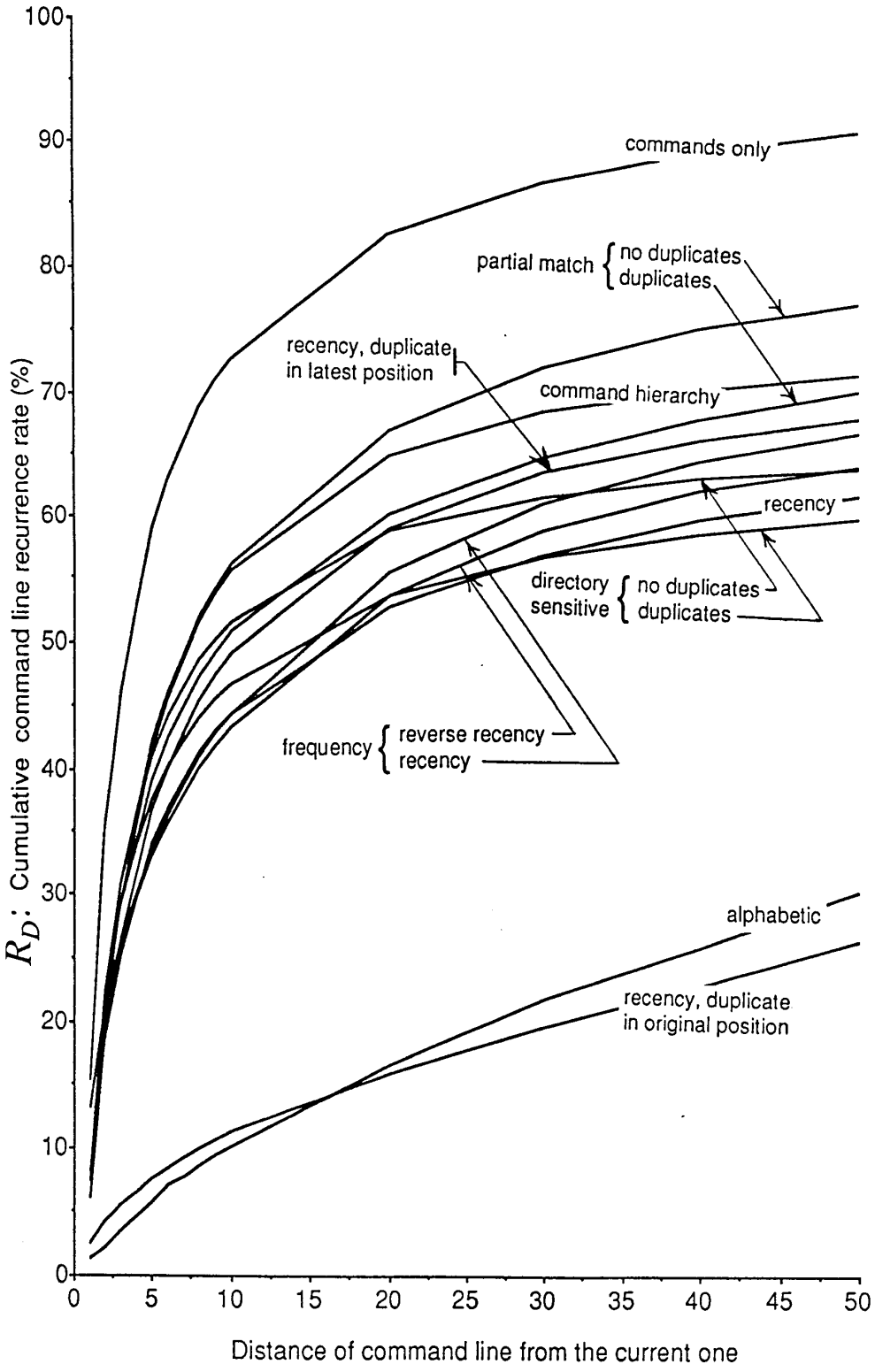

Figure 5: Cumulative probabilities of a recurrence over distance for various conditioning methods

94

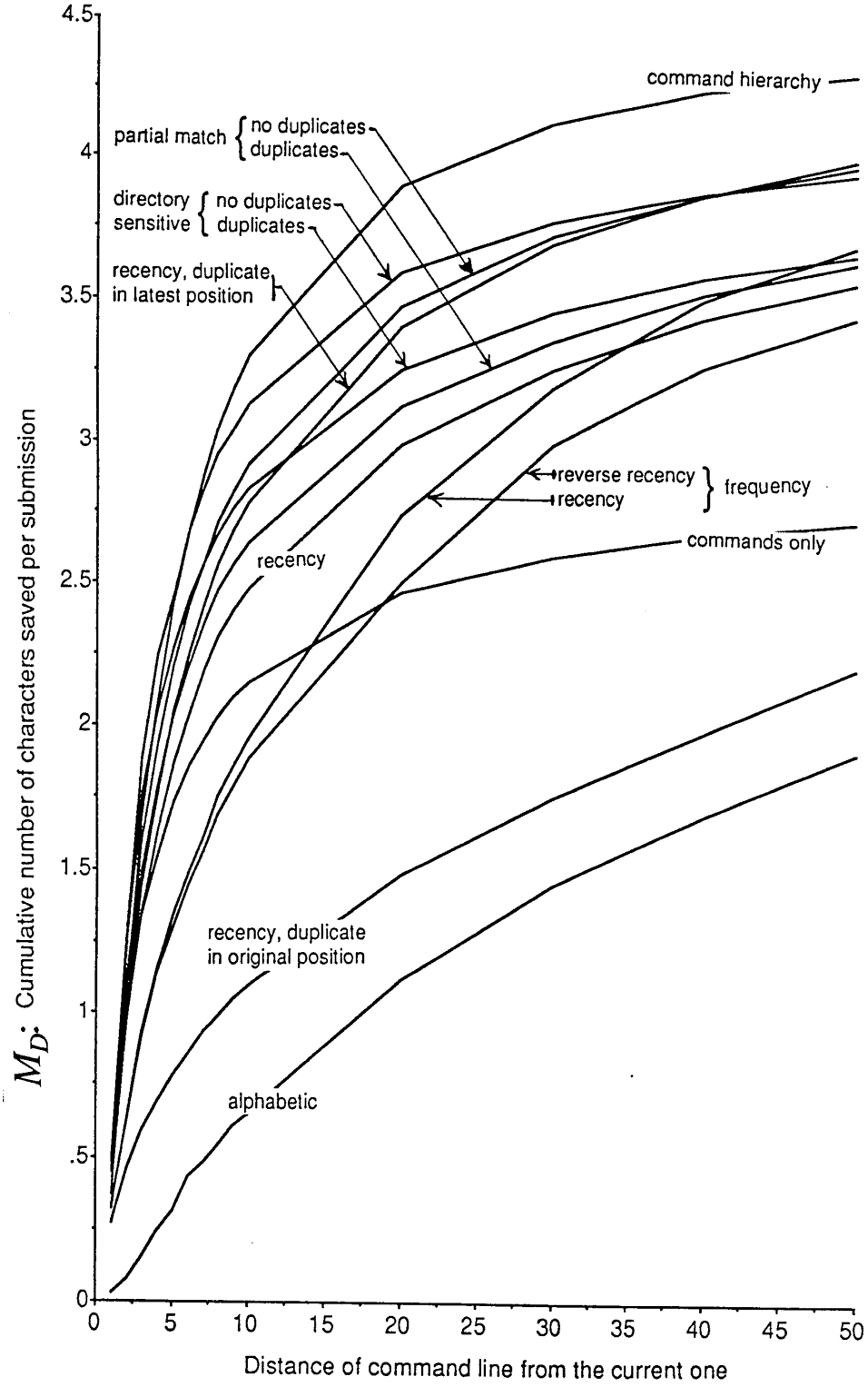

Figure 6: Cumulative average number of characters saved per submission over distance

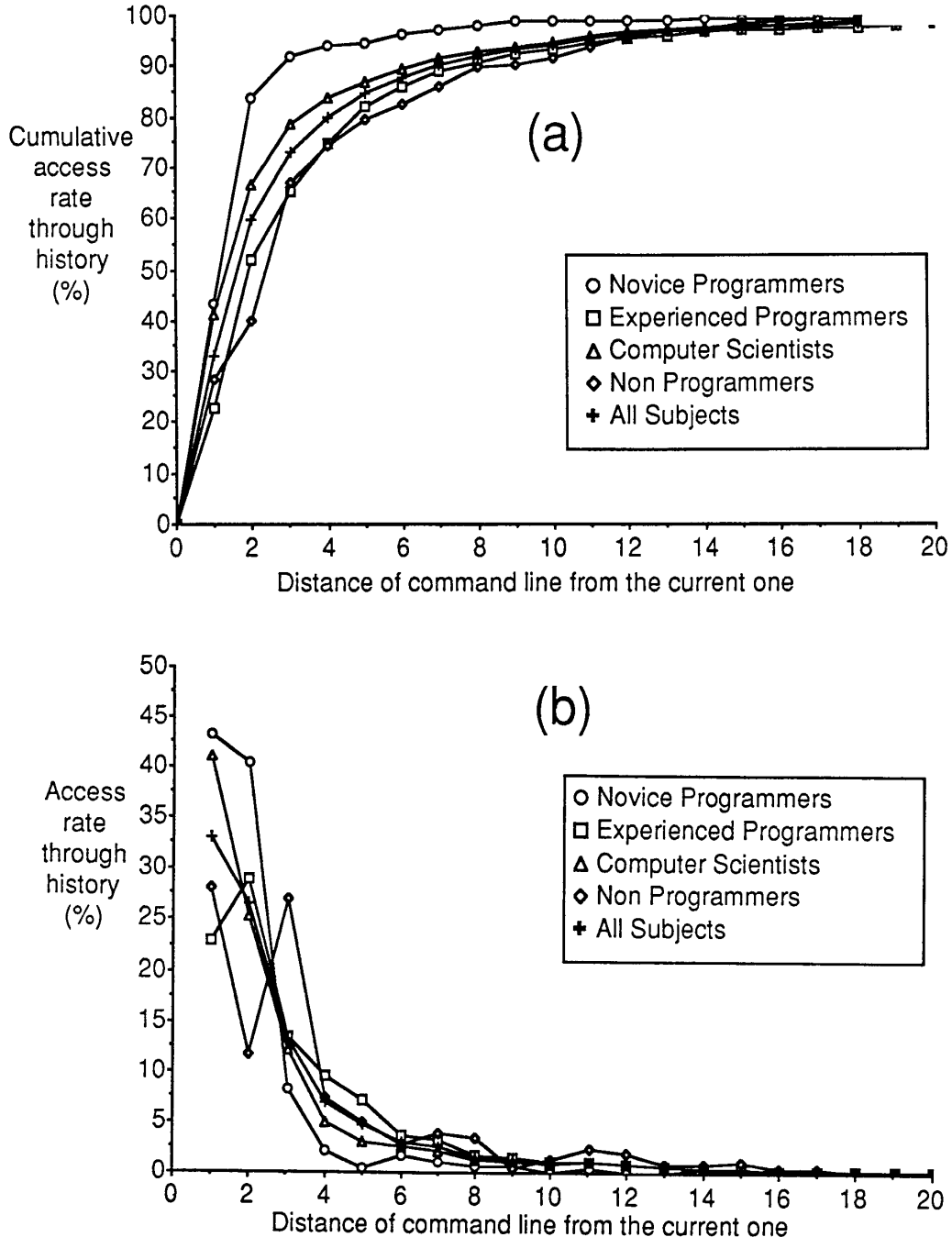

Figure 7: a) Cumulative distribution of history; and b) distribution of history use as a measure of distance

Name	Sample size	Total number of command lines	Number of command lines excluding errors		
			total	mean	std dev
Novice Programmers	55	77423	73288	1333	819.8
Experienced Programmers	36	74906	70234	1950	1276.0
Computer Scientists	52	125691	119557	2299	2022.9
Non-Programmers	25	25608	24657	986	1155.6
Total	168	303628	287736	1712	1498.8

Table 1: Sample group sizes and statistics of the command lines recorded

Sample Name	Recurrence Rate		Range	
	mean	std dev	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%	84.3%
Total	73.8%	9.6	46.4%	91.7%

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

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

In Unix, users change directories through the cd *command. The "~" is shorthand for the home directory. Following "/"'s indicate sub-directories.*

Table 3: Examples of history lists conditioned by different methods

Probability of a recurrence at the given distance d in percent ($\mathcal{R}_d$)														
Conditioning method	Distance 1	2	3	4	5	6	7	8	9	10	20	30	40	50
Recency, duplicates saved:														
always	6.12	12.29	6.71	4.83	4.12	2.94	2.36	1.97	1.66	1.40	0.59	0.32	0.21	0.16
in original position only	2.53	1.75	1.30	1.08	1.01	0.82	0.79	0.66	0.75	0.61	0.35	0.34	0.32	0.23
in latest position only	6.12	12.82	7.58	5.35	4.93	3.48	2.83	2.38	1.99	1.70	0.59	0.30	0.18	0.14
Frequency order:														
second key recency	13.13	7.95	5.24	3.98	3.37	2.83	2.47	2.11	1.79	1.56	0.73	0.49	0.26	0.20
second key reverse recency	13.16	7.74	5.16	3.84	3.20	2.74	2.38	1.91	1.73	1.53	0.74	0.44	0.24	0.16
Alphabetic order:														
duplicates removed	1.27	1.00	1.21	1.30	1.02	1.25	0.76	0.87	0.85	0.57	0.68	0.48	0.32	0.52
Directory sensitive by recency:														
duplicates included	7.46	13.61	8.20	4.89	3.50	2.73	2.06	1.67	1.52	1.22	0.44	0.28	0.15	0.12
duplicates removed	7.46	14.29	9.39	5.78	4.13	3.11	2.37	2.06	1.53	1.38	0.39	0.18	0.11	0.08
Commands only by recency:														
duplicates removed	15.36	19.87	10.89	7.05	5.75	4.09	3.11	2.56	2.21	1.81	0.64	0.28	0.16	0.14
Partial matching by recency:														
duplicates included	8.17	13.49	7.61	5.45	4.51	3.35	2.60	2.18	1.85	1.59	0.63	0.34	0.26	0.16
duplicates removed	8.17	14.07	8.60	6.07	5.34	3.89	3.06	2.64	2.26	1.92	0.65	0.33	0.23	0.18
Command hierarchy:														
recency, duplicates removed	6.12	13.89	9.35	6.60	5.56	4.03	3.19	2.70	2.26	1.83	0.52	0.22	0.13	0.09

Table 4: Probability of a recurrence over distance for various conditioning methods

Cumulative probabilities of a recurrence up to a given distance d in percent ($\mathcal{R}_D$)															
Conditioning method	Distance 1	2	3	4	5	6	7	8	9	10	20	30	40	50	$\mathcal{R}$
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

Table 5: Cumulative probabilities of a recurrence over distance for various conditioning methods

The average number of characters saved over all subjects per recurrence at a given distance ($\bar{c}_d$)														
Conditioning method	Distance													
	1	2	3	4	5	6	7	8	9	10	20	30	40	50
Recency, duplicates saved:														
always	5.94	5.04	5.31	5.57	5.79	5.61	6.11	5.84	5.64	5.51	6.26	5.39	4.83	5.78
in original position only	10.60	10.24	9.76	9.51	8.41	8.83	9.49	7.97	8.17	8.00	7.18	6.70	6.11	5.42
in latest position only	5.94	5.06	5.49	5.72	5.86	5.51	5.88	5.76	5.74	5.87	6.21	5.83	5.31	5.67
Frequency order:														
second key recency	2.57	3.94	5.34	5.76	5.30	5.58	5.17	6.10	6.45	6.50	6.91	7.61	8.73	8.94
second key reverse recency	2.60	3.93	5.35	5.65	5.10	5.56	5.30	6.19	6.08	6.38	7.31	7.58	7.30	8.15
Alphabetic order:														
duplicates removed	3.52	6.67	5.67	5.66	5.66	7.09	6.12	7.10	6.90	6.79	6.33	4.65	5.97	4.84
Directory sensitive by recency:														
duplicates included	6.47	5.70	5.62	5.95	6.24	6.08	6.56	6.14	5.61	5.69	6.08	6.17	5.23	5.08
duplicates removed	6.47	5.71	5.76	6.10	6.14	6.05	6.21	6.04	5.88	5.98	6.43	4.81	6.04	4.36
Commands only by recency:														
duplicates removed	3.25	2.68	2.82	2.96	3.08	3.00	3.07	3.05	3.06	3.15	2.96	3.05	2.78	2.98
Partial matching by recency:														
duplicates included	5.54	4.86	5.02	5.18	5.32	5.22	5.45	5.02	4.94	4.90	5.24	4.09	4.30	3.99
duplicates removed	5.54	4.87	5.18	5.25	5.38	4.99	5.14	5.02	4.99	5.11	4.65	4.27	3.84	4.00
Command hierarchy:														
recency, duplicates removed	5.94	5.36	5.85	6.11	6.15	6.23	6.02	6.11	6.30	6.48	5.91	6.33	4.44	4.37

Table 6: Average number of characters saved over distance per recurrence

Cumulative average savings in characters of D predictions over all submissions (M_D)														
Conditioning method	Distance													
	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

Table 7: Cumulative average number of characters saved per submission over distance

Sample Name	Users of History		Mean rate of history uses (%)
	actual	(%)	
Novice Programmers	11/55	20%	2.03
Experienced Programmers	33/36	92%	4.23
Computer Scientists	37/52	71%	4.04
Non-Programmers	9/25	36%	4.35
Total	90/168	54%	3.89

Table 8: History uses by sample groups

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.

Table 9: Design Guidelines for reuse facilities