

Cluster-Based Find and Replace

Robert C. Miller
MIT Computer Science and AI Lab
77 Massachusetts Ave
Cambridge, MA 02139 USA
rcm@mit.edu

Alisa M. Marshall
Lockheed Martin Corporation
164 Middlesex Turnpike
Burlington, MA 01803 USA
alisa.marshall@lmco.com

ABSTRACT

In current text editors, the find & replace command offers only two options: replace one match at a time prompting for confirmation, or replace all matches at once without any confirmation. Both approaches are prone to errors. This paper explores a third way: *cluster-based find & replace*, in which the matches are clustered by similarity and whole clusters can be replaced at once. We hypothesized that cluster-based find & replace would make find & replace tasks both faster and more accurate, but initial user studies suggest that clustering may improve speed on some tasks but not accuracy. Users also prefer using a perfect-selection strategy for find & replace, rather than an interleaved decision-action strategy.

Categories & Subject Descriptors: H.5.2 [Information Interfaces and Presentation]: User Interfaces – evaluation/methodology, prototyping, user-centered design; H.1.2 [Models and Principles]: User/Machine Systems – human factors, human information processing; H.4.1 [Information Systems Applications]: Office Automation – word processing

General Terms: Design, Human Factors

Keywords: find & replace, text editing, error prevention, clustering

INTRODUCTION

The find & replace command in a typical text editor forces users to choose between two alternatives: replace one match at a time with confirmation, or replace all matches at once. When the document is long and the number of matches large, neither choice is ideal.

Confirming each match is slow and tedious. User thought and action is required for every replacement, which doesn't scale to large, complicated tasks. Worse, the tedium of the task leads the user to make errors. When most answers are Yes, a bored or impatient user eventually starts to press Yes without thinking, which makes confirmation pointless. Although some of these errors may be noticed, few find & replace interfaces provide an obvious Undo command, so fixing the error disrupts the user's task and reduces efficiency

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

CHI 2004, April 24–29, 2004, Vienna, Austria.

Copyright 2003 ACM 1-58113-702-8/04/0004... \$5.00.

further. Worse, some errors due to confirmation boredom may never be noticed at all.

The other choice, replacing all matches without confirmation, may be faster but no less error-prone. Replacing all matches requires the user to trust the precision of the search pattern, that it matches only text that should be replaced and nothing more. By *search pattern*, we mean not only the string of characters to be replaced, but also constraints like word boundaries, case sensitivity, and pattern matching operators. Designing a precise search pattern is a challenging task. It requires an understanding of the pattern constraints that are available and how they interact. It requires a familiarity with the document being edited, knowing which different variants of the pattern may appear and predicting the likelihood of false matches to a pattern. It may require tricks like searching for a longer pattern than you actually need to replace, or breaking a find & replace task down into several subtasks with different search patterns, or learning how to use regular expressions, in order to constrain context and eliminate false matches. In fact, precise global find & replace requires a process of abstraction and testing not unlike programming. Unfortunately, the traditional find & replace interface offers no support for this process — just a Replace All button, which is enough rope for users to hang themselves.

Errors apparently caused by find & replace have been found in published documents [6], among them:

- *arjppicial turf*, on a web site that evidently switched from TIFF images to JPEG;
- *eLabourated*, found in a Wall Street Journal article which also contained references to the British Labour Party;
- *AmriCzar*, in a Reuters wire bulletin about Amritsar, a city in India;
- *stogard*, instead of *standard*, in the Danish users' guide for Windows for Workgroups 3.11. The English word *and* translates to *og* in Danish, which suggests that find & replace by a translator was to blame.

It isn't clear whether these errors were due to tedious replace-with-confirmation or imprecise replace-all, but both techniques have flaws that lead to errors.

This paper explores a third interface for find & replace, which organizes matches into *clusters* based on similarity. This cluster-based find & replace method can combine

the advantages of replace-all and replace-with-confirmation. Large, identical clusters can be selected and replaced all at once, while unclusterable matches can be judged one by one.

One complication of this new approach is that reorganizing matches into clusters necessarily breaks the association between the order in which matches are presented for replacement and the order in which they occur in the document. Traditional replace-with-confirmation always presents matches in document order, which makes it easy for the user to limit replacements to a certain part of the document. Clustering, on the other hand, may bring together widely separated matches into the same cluster. Cluster-based find & replace is therefore global in nature. This paper describes a novel user interface that uses multiple text selections to reflect the global nature of cluster-based find & replace.

The effectiveness of clustering depends on the ability of the clustering algorithm to segregate true matches (which should be replaced) from false matches (which shouldn't). Ideally, the clustering should produce just two clusters, one containing the true matches, and the other, the false matches. Since this is impossible without understanding the user's intent, we aim for a more practical goal: clusters should be *homogeneous*, containing either all true matches or all false matches, and large, containing as many matches as possible. Homogeneous clusters can be quickly and safely replaced or skipped.

Clusters are formed by comparing *features* of matches, determined from content and context. Clusters will be homogeneous only if the features available to the clustering algorithm are sufficient to discriminate true matches from false matches. In general, this may be impossible; without a deep semantic understanding of the document, it may be very hard to discriminate between, say, uses of the word *program* that refer to software and uses of *program* that refer to an event schedule. In practice, however, lexical, syntactic, and stylistic features can substantially discriminate between true and false matches. Our clustering engine uses a wide range of such features, including capitalization, word boundaries, part of speech, style, location in page layout, and adjacent text.

The rest of this paper describes our experience with designing and evaluating a user interface for cluster-based find & replace. After surveying related work, we present the user interface, highlighting a number of design issues that were exposed and resolved by prototyping and pilot evaluations. Next, we describe a formal usability study we conducted to compare traditional find & replace with cluster-based find & replace. We then present the details of the clustering algorithm we used. Finally, we discuss some of the conclusions that can be drawn from our experience.

RELATED WORK

Clustering, also called *unsupervised learning*, has a long history of research in machine learning and information retrieval [1]. In traditional information retrieval, clustering is applied at the document level. A document's fea-

tures are the terms it contains (plus, for web pages, the incoming and outgoing links). Clustering is used in several web search engines, including Vivisimo, WiseNut, Northern Light, AllTheWeb, and Grouper [9], to organize search results and give the user an easy way to constrain searches. Clustering in search engines is particularly helpful for dealing with homonyms, words with diverse meanings. For example, *jaguar* refers to an animal, a car maker, an operating system from Apple, and a game platform from Atari. A clustering search engine like Vivisimo can separate the different senses of a search term into different clusters. Clustering can also suggest additional search terms, as when a search for *Java* produces clusters for *servlets*, *applets*, and *games*.

Cluster-based find & replace differs from document clustering in two important ways. First, the user's goal is not to satisfy an information need, as it is in information retrieval. Instead, the goal is a manipulation task, to modify the matches to the query. Find & replace therefore places more stringent demands on both precision and recall than information retrieval. Second, the matches to a find & replace query are tiny fragments of a document, not complete documents. The features used for whole-document clustering — terms — are not as useful for clustering fragments. Freitag confirmed this in his study of inductive learning for information extraction [2], which showed that a relational learner using features similar to ours was much more effective at extracting fragments from documents than a term-based learner.

Cluster-based find & replace is closely related to outlier finding [4], a technique for highlighting unusual instances in a pattern match or selection, on the expectation that outliers are likely to be errors. We use substantially the same algorithm to build clusters that outlier finding uses to detect outliers, although the output of the algorithm is applied to the user interface in different ways. Whereas outlier finding ignores large clusters in order to focus on unusual individuals, both large and small clusters are useful in cluster-based find & replace, so the interface presents both.

Techniques related to outlier finding can be found in other systems as well. For example, Microsoft Excel detects spreadsheet formulas that are inconsistent with the formulas in neighboring cells, highlighting them as possible errors. Morris and Cherry built an outlier-finding spell-checker [5] that computes trigram frequencies for a document and then sorts the document's words by their trigram probability, and found that it worked well on technical documents.

Find & replace itself has a hoary legacy, dating back to teletype text editors such as Unix *ed*, in which the *substitute* command was the primary way to change existing text. Although the importance of find & replace has faded somewhat with the rise of direct-manipulation text editing, virtually every text editor still includes a find & replace command.

In programming and web site maintenance, *multiple file* find & replace is often essential. A search of the shareware web

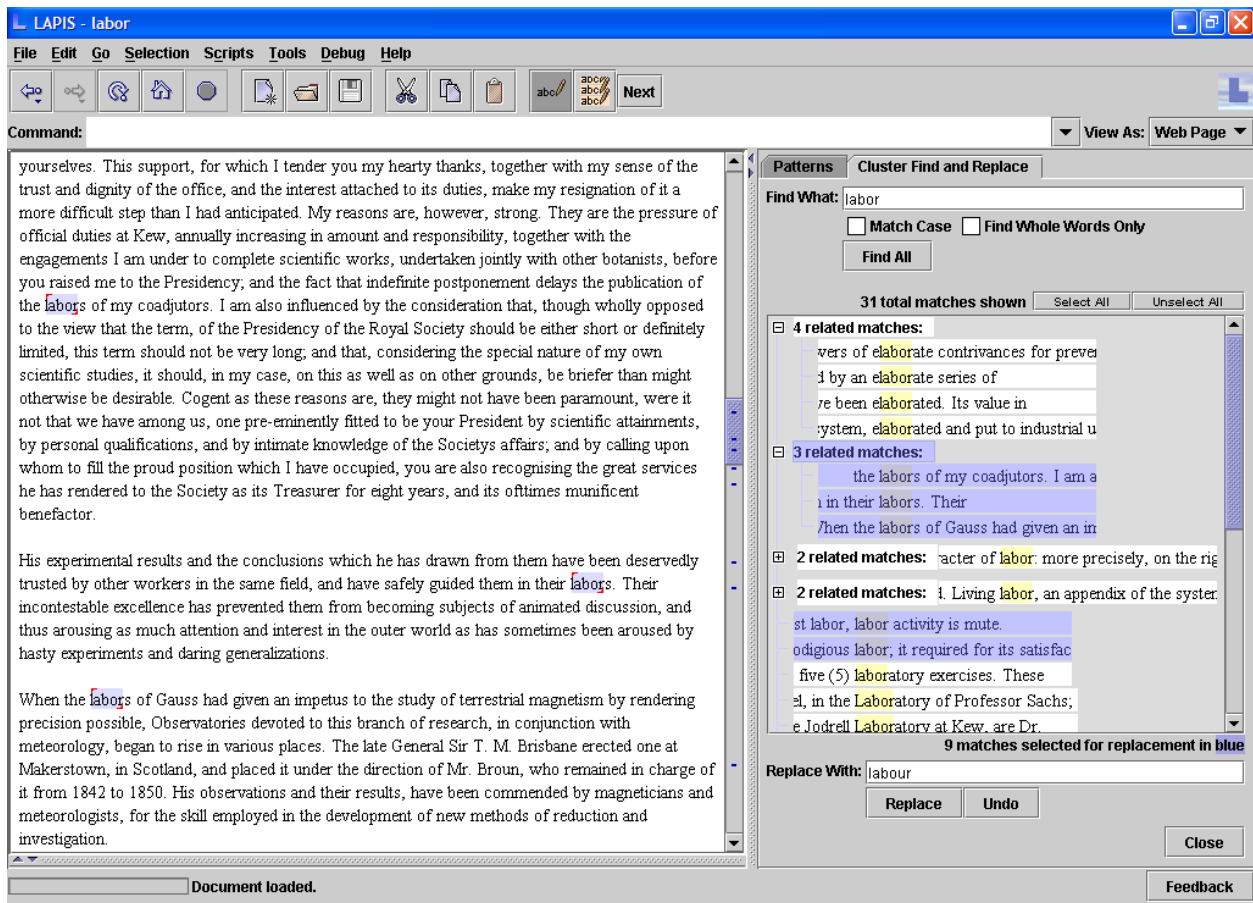


Figure 1: Cluster-based find & replace interface.

site Downloads.com finds a number of Windows programs specialized to this task, among them Advanced Find and Replace, HandyFile Find and Replace, Alias Find & Replace, Actual Search & Replace, and HTML Search and Replace. All these tools provide the traditional replace-all or replace-with-confirmation interface. Some of the users' comments on Downloads.com give insight into the problem: "Fast and easy but dumb (as they all are)... would need some intelligence in choosing where and when to change... now the user has to stay strict with the code." [7] Our clustering prototype currently works on a single document, but work is underway to generalize it to multiple-file tasks.

USER INTERFACE

This section describes our user interface for cluster-based find & replace. The interface was designed through four iterations of paper prototypes and pilot evaluations involving a few users each. The iterations explored a number of design alternatives, among them ways to display clusters, ways to describe the content of a cluster, and interaction techniques for selecting and replacing clusters. In this section, we first present the final interface that we implemented, and then discuss some of the lessons learned from exploring design alternatives.

The cluster-based find & replace interface is implemented inside LAPIS [3], an experimental text editor. LAPIS has several unusual features that are relevant to find & replace. First, the editor allows multiple discontinuous text selections (shown as blue highlights in the editor window in Figure 1). Multiple selections can be made several ways: by the mouse, by a pattern, or by inference from examples. Multiple selections can be used for editing; delete, cut, copy, paste, and typing affect all selections at the same time. Multiple selections may be spread throughout a file. Marks in the scrollbar provide cues to where selections are located, even if they are scrolled offscreen.

Figure 1 shows cluster-based find & replace in action. The interface sits in a side panel of the editor window. The panel is roughly divided into three parts. The top part of the panel contains controls for specifying the pattern: the text to search for, and check boxes for case sensitivity and word boundary constraints. These controls are conventionally provided in other find & replace interfaces as well.

The middle part of the panel displays all the matches to the pattern found in the document. Each pattern match is represented by a snapshot of document context around the match,

with the match itself highlighted in yellow. Matches are grouped into clusters by similarity (using an algorithm described later). The clusters are displayed in a standard tree widget, with each cluster described by a heading, e.g. “4 related matches”, which acts as a control point for collapsing or expanding the cluster. When a cluster is collapsed, one of its members (the first in the document) is displayed in the collapsed heading as a representative. Figure 1 includes two collapsed clusters.

The list of clusters is sorted by cluster size, with larger clusters appearing first. Singleton clusters — matches that were unique and unclusterable — appear last. Singleton clusters have no heading, and appear simply as leaves at the top level of the tree widget. The last five items in the cluster view of Figure 1 are singleton clusters.

Clicking on a match selects it, both in the match list and in the editor. The editor window scrolls automatically to bring the selection into view. An entire cluster can be selected by clicking on its heading or on the margin around it. Multiple clusters or matches can be selected by click-to-toggle selection. The Unselect All button clears the selection.

The bottom part of the panel controls replacement. A text field is provided for entering replacement text. When the Replace button is pressed, the selected matches are replaced in the editor and removed from the list of matches. The Undo button undoes previous replacements, undoing not only the effect on the editor but also restoring the replaced matches to the cluster display.

Design Lessons

The interface just described was developed through four iterations of low-fidelity prototyping, with each iteration tested on three fresh users drawn from a university research environment. The prototypes were hybrid paper and computer interfaces. The find & replace panel was prototyped on paper, while the existing LAPIS editor was used (under the control of the experimenter) to display the document on a computer screen, showing selected matches in context and showing the effect of replacements. Prototype users were given several find & replace tasks similar to those used in the formal user studies described later. For example, one task was a final exam schedule in which Monday was abbreviated to M, and users were asked to undo the abbreviation without affecting other M’s in the schedule (e.g., in an instructor’s initials).

One surprising result from prototype testing was users’ preferred replacement strategy. Nearly all users tried to make a perfect selection, selecting all the true matches and omitting all the false ones, before pressing Replace. An alternative strategy would interleave decision-making with action, choosing a cluster or a match and then immediately pressing Replace to lock in the choice and remove it from the list of matches. Traditional replace-with-confirmation forces the interleaved strategy; replace-all requires the perfect-selection strategy. Given a choice, however, users seem to prefer per-

fect selection. This preference was not affected by the location of the Replace button (either above or below the list of matches), nor was it restricted to paper prototyping, persisting throughout later computer implementations as well.

From this observation, it follows that some features we thought necessary were actually irrelevant, while others that seemed less important were actually vital. For example, early prototypes included a button (variously called Omit, Reject, Exclude, and Discard) that removed a false match or false cluster from the display without replacing it. This function is important to the interleaved strategy, because it acts as the dual to Replace — an action to take when the user has decided that a match is false. Our first prototype even made this duality explicit, by labeling the two buttons “Yes, Replace” and “No, Omit”. As it turned out, users never touched the Omit button during their tasks, regardless of how it was labeled. When asked afterwards what they thought the button meant, either they couldn’t guess or they believed that it might delete the match from the *editor*, instead of just removing it from the find & replace panel.

A perfect selection must in general be a multiple selection. In conventional tree and list widgets, however, multiple selection is fragile, because a single click can clear a carefully-constructed selection. Furthermore, many users are unfamiliar with the modifier keys used to create a multiple selection (under Windows, the Control key toggles selection and the Shift key extends a range). Although these problems did not appear in paper prototype testing, since the prototypes did not simulate the low-level interaction issues, it was easy to anticipate eventual problems with the computer interface. Indeed, early pilot tests of a computer interface with conventional multiple selection revealed that some users lost selections and others wanted to make multiple selections but didn’t know how. These observations drove us to use toggle selection instead of conventional list selection.

Another consequence of the perfect-selection strategy is the importance of expanding clusters and selecting or deselecting individual matches within a cluster. One of our early prototypes displayed each cluster as a row in a table, so that a cluster could only be selected and replaced as a unit. If a cluster were inhomogeneous — containing both true matches and false matches — then using a perfect-selection strategy with this prototype would lead to failure. Using an interleaved strategy, on the other hand, users would still be able to make progress with the homogeneous clusters, falling back to replace-with-confirmation (which was offered as an option in this prototype) to deal with the inhomogeneous clusters. Since perfect selection was preferred, however, users found this interface frustrating.

Prototypes also explored other ways to describe a cluster. One approach listed the features that were common to the members of the cluster, e.g. *just before Punctuation, just before LowerCaseWord, contains LowerCaseLetters, not just*

after Tag. As a rule, users found these explicit feature lists unreadable and unhelpful for deciding whether a cluster might have true matches or false matches. We also experimented with displaying a confidence rating based on the cluster's distance from a typical match in feature space [4], but users either found these ratings inexplicable or believed them to be a redundant indicator of the cluster's size.

We conclude from this experience that a cluster is best described by its own members. A single representative is used when the cluster is collapsed. Since users inevitably expand every cluster to check for inhomogeneity, however, it makes sense to display all the clusters expanded initially, in which case a cluster's default description is, in fact, its membership.

Two more lessons are worth mentioning. First, the early prototypes used two kinds of highlighting in the editor window. In addition to highlighting selected matches in blue, *all* matches were kept constantly highlighted in yellow, so that they were easy to find and examine in their original context. The yellow highlight in the editor window was consistent with the yellow highlight in the snapshot shown in the find & replace panel. Some users were confused by this extra highlighting, believing that the yellow highlights were selections and that pressing Replace would replace all of them (effectively a Replace All). Another user scanned for the yellow highlights in the editor, selected each one with the mouse, and manually typed the replacement text. Both pathological behaviors were caused by failing to notice that the list of clusters was actually selectable. Eliminating the yellow highlights solved these problems.

Finally, some find & replace errors are caused by disagreement of alphabetic case between the matched text and the replacement text. For example, literally replacing *jaguar* with *panther* everywhere is wrong if *Jaguar* is ever capitalized. Since clustering separates matches by capitalization, we speculated that users might notice and solve these problems themselves by modifying the case of the replacement text to match the cluster being replaced. In fact, the prevalence of the perfect-selection strategy suggests that this hope is unfounded, and we observed errors of this kind frequently in the paper prototype. Our computer implementation solves this problem in the conventional way used by other word processors and text editors: when the pattern is case insensitive, and the user types the replacement text with no capitalization, then the capitalization of the replacement text is automatically adjusted to agree with the text it is replacing.

ALGORITHM

We now turn to the details of the clustering algorithm, which is based on the outlier finder algorithm [4]. The algorithm takes as input a set of matches R (which are substrings of a document) and returns a partition of R such that the members of each subset are more similar to each other than to the other members of R . Similarity is determined by representing each match in R by a binary-valued feature vector. Features and

feature weights are generated automatically from R , optionally assisted by a knowledge base (in this case, a library of useful text patterns).

A feature is a predicate f defined over text regions. The clustering algorithm generates two kinds of features: *library features* derived from a pattern library, and *literal features* discovered by examining the text of the substrings in R .

LAPIS has a considerable library of built-in parsers and patterns, including Java, HTML, character classes (e.g. digits, punctuation, letters), English structure (words, sentences, paragraphs), and parts of speech for English words. The user can readily add new patterns and parsers to the library. Features are generated from library patterns by prefixing one of seven relational operators: *equal to*, *just before*, *just after*, *starting with*, *ending with*, *in*, or *containing*. For example, *just before Number* is true of a region if the region is immediately followed by a match to the Number pattern, and *in Comment* is true if the region is inside a Java comment. In this way, features can refer to the context around substrings, even nonlocal context like HTML font or paragraph style or Java syntax.

Literal features are generated by combining relational operators with literal strings derived from the substrings in R . For example, *starts with "http://"* is a literal feature. To illustrate how we find literal features, consider the *starts with* operator. The feature *starts with "x"* is useful for describing degree of membership in R if and only if a significant fraction of substrings in R start with the prefix x . To find x , we first find all prefixes that are shared by at least two members of R , which is done by sorting the substrings in R and taking the longest common prefix of each adjacent pair in the sorted order. We then test each longest common prefix to see if it matches at least half the strings in R , a trivial test because R is already in sorted order. For all prefixes x that pass the test, we generate the feature *starts with "x"*.

With a few tweaks, the same algorithm can generate literal features for *ends with*, *just before*, *just after*, and *equal to*. For example, the *ends with* version searches for suffixes instead of prefixes, and the *just before* version searches for prefixes of the text *after* each substring instead of in the substring itself. Only *in* and *contains* features cannot be generated in this way. The clustering algorithm does not presently generate literal features using *in* or *contains*.

Once the set of features has been generated, features that match every member of R are pruned, since they are useless for distinguishing clusters. Our algorithm takes the simplest possible approach to clustering the members of R : members with identical features are placed in the same cluster, while members which differ on at least one feature are placed in different clusters. This simple approach tends to produce smaller but more homogeneous clusters. More sophisticated clustering techniques are available, such as the commonly used k -means algorithm [1], but these algorithms require

Calendar: expand *M* to *Monday*, but only in exam dates

Dynamic Strategic Planning	R. de Neufville	1-190	M 12/16 Morning
Transportation Systems	J Sussman	1-190	F 12/20 Morning
Trans & Demand & Economics	M Ben-Akiva	1-242	M 12/16 Morning

Quote: replace ? with apostrophe or double quotes as needed

We hope to make the contents of our service courses more clearly known to other faculty, said Miller. We're always getting questions from faculty like: Do you really teach complex numbers in your courses? because students claim they've never seen them. Whether

Figure: remove the asterisk from *figure**, except in `\begin{figure*}` or `\end{figure*}`

within the document. Screenshots of the two interfaces are shown in `\Figure* \ref{screenTree1:Figure*}` and `\Figure* \ref{screenList1:Figure*}`. `\begin{figure*}`

Labor: change all *labor* word forms to *labour*, but not *elaborate* or *laboratory*

capitalist production. Labor is interaction. Therefore, in order to really understand postfordist laboring praxis, one must increasingly refer to Saussure, to Wittgenstein and to Carnap. These authors have hardly shown any interest in social relations of production; nonetheless, having elaborated theories and images of language, they

Figure 2: Tasks in user study.

more input from the user (e.g., a value for *k*, the number of clusters) and are more likely to produce inhomogeneous clusters, which are undesirable for find & replace. Since our primary purpose in this paper is to explore the user interface design, we opted for a simple algorithm that tends to produce homogeneous clusters.

EVALUATION

We compared cluster-based find & replace with traditional find & replace in a small user study. We obtained 16 participants by advertisements posted around a university campus. Participants ranged in age from 18 to mid-30's (most were college students), with 9 males and 7 females. All users reported being experienced in word processing, using at least one word processor or text editor regularly, and all users reported significant experience with find & replace (at least "enough to be comfortable"). Participants were paid \$5 per half hour for up to an hour of work.

Each participant was given four find & replace tasks. Portions of each task are shown in Figure 2. The tasks were chosen to represent a diverse range of texts, replacement patterns, and task sizes. For example, the calendar task was organized as a table, and correct replacements always appeared in the same column of the table, giving users an addi-

	Words	Pattern	Matches	Clusters
Calendar	4836	M	73/663	4/78
		M _(case)	73/366	4/33
		M _(case, word)	73/97	4/11
Quote	563	?	(17,17)/35	(8,13)/22
Figure	4165	figure*	35/55	19/25
Labor	3533	labor	38/54	33/44

Table 1: Task sizes. *Words* is the total document size. *Pattern* is a search pattern that might be used to solve the task; *case* means case-sensitive, and *word* means whole word only. *Matches* is the number of matches to replace / total number of matches for a pattern. (In the quote task, 17 matches must be replaced with apostrophes, and 17 with double quotes.) *Clusters* is the number of clusters to replace / total number of clusters found by the clustering algorithm for a pattern.

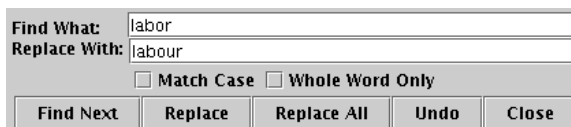


Figure 3: Traditional find & replace for user study.

tional spatial cue. By contrast, the quote task was organized in freeform paragraphs, and correct replacement depended strongly on the context. The figure task was an example of markup or code replacement, and the labor task tested word replacement, including multiple word forms. Table 1 shows the size of each task by several metrics.

Each user/task combination was assigned to one of two conditions: the clustering interface described previously, or a traditional find & replace interface modeled after Microsoft Word and Microsoft Notepad (Figure 3). This interface was implemented in the LAPIS editor in order to eliminate editor-related differences between the conditions. Each user did all four tasks – two in one condition, and two in the other. The order of tasks and assignment of tasks to conditions was balanced and randomly assigned to users.

Users were told to work as quickly and accurately as they could, and announce when they believed they had finished a task correctly. The resulting document was then compared with the expected correct result. If the user's result differed in a substantial way (extra text, missing text, different capitalization) from the correct result, then the user was told that there were errors, but not where or how many, and asked to stay on the task until all the errors were fixed. Users were free to fix errors in a variety of ways, including undo, manual editing, and restarting the task from the original input, but for find & replace they could only use the interface appropriate to the current condition. The total time required to reach an error-free result was measured (not including time spent by the experimenters to check for errors). Only one of the

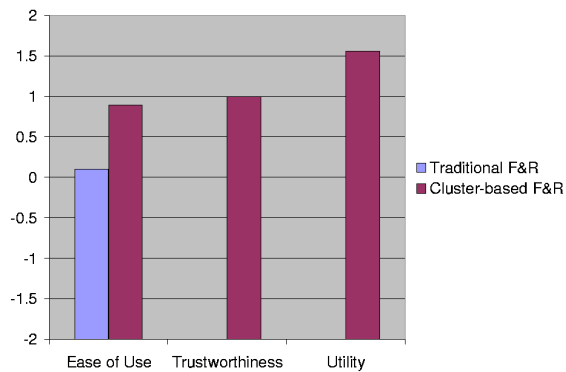


Figure 4: Subjective ratings. Ease of use was rated for both interfaces, but only clustering was rated for trustworthiness and utility.

64 user-task combinations failed to reach an error-free result within 30 minutes, where it was cut off. This failure occurred with the traditional find & replace interface on the calendar task, and its data was omitted from the results reported below.

Results

Times and error rates are shown in Table 2. The time is the total time the user spent to obtain an error-free result (omitting time spent by the experimenters to check for errors). The error rate is the fraction of tasks for which at least one error was found in the user’s first result, which represented the finished product in the user’s own judgement.

Using single-factor ANOVA on each task with the find & replace condition as the independent variable, the only significant differences were that cluster-based find & replace was faster in the calendar task ($p = .02$), more accurate in the calendar task ($p = .02$), and slower in the quote task ($p = .03$).

After the study, users rated the ease of use of both interfaces on a 5-point Likert scale ranging from *very hard* (-2) to *very easy* (+2). For the clustering interface only, users also rated how much they trusted the clustering to come up with useful groups of related matches (Trustworthiness), and whether they would use cluster-based find & replace if it were built into their favorite word processor (Utility). The results are shown in Figure 4. Users were generally very positive about cluster-based find & replace, and neutral about traditional find & replace, but the difference in ease of use ratings was not statistically significant.

Discussion

The calendar task and quote task deserve some discussion, since they produced the strongest and yet most contradictory effects in the study. On the calendar task, cluster-based find & replace was almost three times faster on average than traditional find & replace, largely because clustering was highly effective on this task. As Table 1 indicates, whereas the best pattern for traditional find & replace (“M”, with case-sensitivity and whole word enabled) had 97 matches

for the user to check, the clustering algorithm grouped these matches into only 11 clusters for cluster-based find & replace. On other tasks, clustering was less effective, producing more clusters and therefore providing less leverage.

On the quote task, in contrast, cluster-based find & replace was about two times *slower* on average than traditional find & replace. This task revealed flaws in the cluster-based user interface that had gone unnoticed in earlier prototyping. First, the task demanded more context than the cluster display alone could provide. Figure 2 shows an example: deciding how to replace the question marks in *courses??* requires reading the entire sentence around it. Thus, many matches from the cluster display had to be located in the document in order to see the larger context. Unfortunately, the user interface made this difficult; although selecting a match in the cluster list highlighted it in the document, the highlight was indistinguishable from other highlighted currently-selected matches. One user worked around the problem by toggling a match on and off to make it blink in the document, but most resorted to a slow visual scan of the document instead.

STATUS & FUTURE WORK

Our clustering interface actually differs in *two* substantial ways from traditional find & replace. Our interface not only reorganizes the matches into clusters of similarity, but also displays all the matches in a compact list, where they can be scanned and selected. Which is more important for usability – the organization provided by clustering, or the visibility and affordance provided by the list of matches? To answer this question, we have run an early pilot study of 13 users that included a third interface condition, identical to the clustering interface except that the list of matches was unclustered and sorted in document order. The details of the study are omitted for lack of space, but the results suggest that the unclustered list interface took the same time on average as traditional find & replace, while the clustering interface was faster (though none of the differences were significant).

Our decision to display all the clusters together in a list had another unfortunate effect. Iterative design suggested that the best way to represent the clusters generated by our algorithm is simply a list of the cluster’s members (rather than an abstract description like *boldfaced, starting Sentence*, etc.). When clusters are represented primarily by their membership, the list of clusters tends to look like a list of individual matches. As a result, several users in the user study clicked only one match at a time, instead of a cluster (or range of matches) at a time. In other words, these users were treating the interface as if it were an unclustered list. Having observed this problem in pilot studies, we tried to fix it with better mouse-over feedback, highlighting the entire cluster when the mouse was over the cluster’s margin. This reduced but didn’t eliminate the problem. Better graphic design distinguishing the clusters might also help. Alternatively, one could imagine a fourth interface condition that provided clustering but not visibility of all clusters at once,

		Calendar task	Quote task	Figure task	Labor task
Traditional F&R	time	8:29 ($\sigma = 5:31$)	4:27 ($\sigma = 2:17$)	4:43 ($\sigma = 2:43$)	4:57 ($\sigma = 2:14$)
	errors	71%	25%	50%	63%
Cluster-based F&R	time	2:58 ($\sigma = 1:57$)	10:05 ($\sigma = 6:08$)	3:29 ($\sigma = 1:20$)	5:21 ($\sigma = 5:04$)
	errors	13%	38%	38%	63%

Table 2: User study results. Time is the median time (in minutes:seconds) to reach an error-free result, with standard deviation σ given in parentheses. Error rate is the fraction of tasks for which the user’s first result had errors. Each task-condition combination is aggregated over 7 or 8 users.

instead presenting one cluster at a time in the style of replace-with-confirmation. Designing and evaluating this interface remains future work.

Turning to issues of effective clustering, our algorithm faces a tension between cluster size and cluster homogeneity, strongly affected by the number of features available to the algorithm. Using too many features might make every match a unique, singleton cluster, so clustering would confer no organizational advantage. On the other hand, with too few features, the clusters may not be homogeneous, forcing the user to look at the matches one at a time anyway. One way to address this tension is by assigning different weights to features, with high-weight features creating cluster boundaries while low-weight features are ignored.

The user could also guide the clustering process by choosing which features should be used for clustering. For example, a user might choose to cluster the matches by capitalization or word boundaries. Interestingly, none of the 40 or so users who tested variants of cluster-based find & replace asked for a way to control the clustering (although many asked for better explanation of the automatic clustering). Find & replace users may be more accustomed to controlling matches by changing the search pattern, rather than reorganizing a display of matches for easier selection.

Finally, cluster-based find & replace addresses only the errors caused by false matches. Pattern matching tasks also suffer from *false misses*, text that should have been replaced but was not matched by the user’s pattern. False misses can happen when the user’s pattern is too specific (e.g., using whole-word searching when the word sometimes takes endings) or when misspellings block matching. In order to detect false misses, a cluster-based find & replace interface would have to widen the scope of the user’s pattern, using techniques like *agrep*’s approximate matching [8] or outlier finding [4]. The cost would be significantly more potential matches for the user to examine. Integrating possible false misses into the find & replace interface remains future work.

The cluster-based find & replace interface described in this paper can be found in the experimental LAPIS editor, which is written in Java and freely downloadable from the Web at <http://graphics.csail.mit.edu/lapis>.

CONCLUSION

This paper has introduced cluster-based find & replace, a new technique designed to improve accuracy and speed on difficult find & replace tasks. Unlike the traditional replace-all and replace-with-confirmation approaches, clustering allows users to focus on groups of similar replacement decisions. By replacing or skipping large homogeneous clusters all at once, but considering unusual matches one at a time, users can focus their attention where it is most needed in the task.

Design experience showed that users prefer using a perfect-selection strategy for find & replace, rather than an interleaved decision-action strategy. We also discovered that the best description for a cluster is its membership, although this doesn’t keep users from wondering how clusters are related.

Cluster-based find & replace suggests ways that automatic reorganization might be applied to other confirmation problems that arise in user interfaces. We hope that these techniques will evolve to reduce tedium and increase correctness in difficult tasks that demand human judgement.

Acknowledgements

We gratefully acknowledge Min Wu, Matt Notowidigdo, and the anonymous referees for their help with this paper.

REFERENCES

1. Michael R. Anderberg. *Cluster Analysis for Applications*. Academic Press, 1973.
2. Dayne Freitag. *Machine Learning for Information Extraction in Informal Domains*. PhD thesis, Computer Science Department, Carnegie Mellon University, November 1998.
3. Robert C. Miller and Brad A. Myers. Multiple selections in smart text editing. In *Proc. IUI 2002*, pages 103–110.
4. Robert C. Miller and Brad A. Myers. Outlier finding: Focusing human attention on possible errors. In *Proc. UIST 2001, CHI Letters 3(2)*, pages 81–90.
5. Robert Morris and Lorinda L. Cherry. Computer detection of typographical errors. Technical Report 18, Bell Laboratories, July 1974.
6. Peter G. Neumann (moderator). Risks Digest: Forum on risks to the public in computers and related systems. <http://catless.ncl.ac.uk/Risks/v10/n23,v18/n24,v19/n12>.
7. Various. CNET user reviews for Search and Replace 98. <http://download.com.com/3302-2048-916215.html>, 2003.
8. Sun Wu and Udi Manber. Agrep – a fast approximate pattern searching tool. In *Proc. Winter USENIX 1992*, pages 153–162.
9. Oren Zamir and Oren Etzioni. Grouper: A dynamic clustering interface to web search results. In *Proc. WWW8*, 1999.