1995

Semiautomatic disabbreviation of technical text

Rowe, Neil C.

Monterey, California. Naval Postgraduate School

Information Processing and Management, 31, no. 6 (1995), 851-857.

Semiautomatic disabbreviation of technical text

Neil C. Rowe and Kari Laitinen

Department of Computer Science

Code CS/Rp, U. S. Naval Postgraduate School Monterey, CA USA 93943 (rowe@cs.nps.navy.mil)

Abstract

Abbreviations adversely affect information retrieval and text comprehensibility. We describe a software tool to decipher abbreviations by finding their whole-word equivalents or "disabbreviations". It uses a large English dictionary and a rule-based system to guess the most-likely candidates, with users having final approval. The rule-based system uses a variety of knowledge to limit its search, including phonetics, known methods of constructing multiword abbreviations, and analogies to previous abbreviations. The tool is especially helpful for retrieval from computer programs, a form of technical text in which abbreviations are notoriously common; disabbreviation of programs can make programs more reusable, improving software engineering. It also helps decipher the often-specialized abbreviations in technical captions. Experimental results confirm that the prototype tool is easy to use, finds many correct disabbreviations, and improves text comprehensibility.

Acknowledgements: This work was sponsored by the Defense Advanced Research Projects Administration, as part of the I3 Project under AO 8939, and by the Technical Research Centre of Finland (VTT). This paper appeared in *Information Processing and Management, 31,* no. 6 (1995), 851-857.

Introduction

Abbreviations of words and phrases pose a special challenge to information retrieval because of their frequent ambiguity. To be useful, an abbreviation must be significantly shorter than its source, but the loss of much information usually means several other words or phrases correspond to each abbreviation. Clearly, people use context to decipher abbreviations. This paper explores some of these context-based methods.

Abbreviations in technical documents, and their subtype of acronyms, have long been criticized informally (e.g. Ibrahim (1989)). One kind of technical text is notorious for overuse of abbreviations, computer programs. Since names of procedures and variables must be repeated exactly everywhere they occur in a computer program, computer programmers have learned to shorten their procedure and variable names to make them easier to type. But what is good for constructing a computer program is bad for other people reading, understanding, and using the program. Thus abbreviating seriously affects one of the most critical issues in software engineering, finding and reusing old computer programs for new purposes. Schneiderman (1980, pp. 70-75) and Weissman (1974) showed experiments in which improved ability to answer questions about a program occurred when mnemonic names were used instead of abbreviated names in the program, and Laitinen (1992) showed programmers preferred to work with programs having natural-language-word or "natural" names. Natural names can be developed early in the software design cycle by a structured methodology like Laitinen and Mukari (1992). But not all programmers will accept this additional

imposition, and more importantly, this methodology cannot be used with existing programs that need maintenance. So we have explored automatic and semi-automatic replacement of abbreviations by natural names, and have developed a prototype tool, a "disabbreviator". Its output can either replace the original text or aid its use.

Some simple tools replace abbreviations in text according to a fixed list, including DataLift from Peoplesmith, Inc., Magic Typist from Olduvai Corp., and The Complete Writer's Toolkit from Systems Compatibility Corp. The Newton device from Apple uses a word list to reconstruct handwritten input. Laitinen (1992) mentions a more sophisticated tool that also checks for conflicts among replacement names. Unfortunately, abbreviations vary considerably between applications, and the general-purpose replacement lists of these programs are of only limited help for the often specialized task of software description. For example, "lbr" is a common abbreviation for "labor" in business, but not in a biological laboratory. Rosenberg (1992) lists an average of two different interpretations for every three-word abbreviation, even for the restricted domain of "information technology" and the restriction to common interpretations. An improvement would be application-specific abbreviation pairs plus "disabbreviation" rules that can intelligently guess replacements not in the list. But the challenge is to find a way to limit the seemingly unbounded number of words that could be checked as the source of an abbreviation.

We will first describe the ideas of the tool in regard to the data structures, design of the rule-based system, and analogy-finding. Experiment 1 considers computer programs. Experiment 2 validates the improved comprehensibility of tool output. Experiment 3 shows the tool works for another kind of technical text, the captions in a large photographic library.

Tool design

Data structures

Our disabbreviation tool uses the following support files, all hashed:

-- a dictionary list of 29,000 English words;
-- a list of reserved words for the programming language and/or operating system whose programs are being analyzed, if applicable;
-- an auxiliary list of valid English words found in previous runs but not in the preceding files;
-- common words of the application domain, used as additional comment words (optional);
-- standard abbreviations used in computer software, with their disabbreviations;
-- replacements accepted for previous text (optional).

The dictionary is necessary to define the acceptable natural names in the source text. To obtain it, we combined wordlists from the Berkeley Unix "spell" utility, the GNU Emacs "ispell" utility, the first author's papers, and some electronic mail. The word list had good coverage of words from the computing fields, but also included nearly all common English words. We removed abbreviations and misspellings from these sources manually, and were very free in removing unusual words under four letters in length. This gave us a reasonably broad vocabulary of about 29,000 English words. This dictionary is then supplemented with reserved words specific to the programming language being used if the source text is a program, and any new words confirmed by the user on previous runs. The user may also optionally include a file of important words of the real-world domain that the program addresses; these may be in the dictionary, but have special priority in trying to find disabbreviations.

We also manually compiled a "standard abbreviation list" of 168 entries from a study of programs written by a variety of programmers, supplemented by the list of Rosenberg (1992); example entries are "ptr -> pointer" and "term -> terminal". These are the abbreviations we observed repeatedly in a wide range of program examples. The list was kept short to reduce domain-dependent alternatives; but explicit listing does save time in disabbreviating the most common abbreviations, even those that can be deciphered from the disabbreviation rules discussed next. The list is supplemented by disabbreviations confirmed or provided by the user, as we will discuss.

Disabbreviation methods

Disabbreviating means replacing an abbreviated name with a more understandable "natural" one consisting of whole English words. To disabbreviate, we "generate-and-test": We select candidates and abbreviate them various ways until we match a given abbreviation. The abbreviation rules derive mostly from our study of example programs, with some ideas from Bourne and Ford (1961). (Uthurusamy et al (1993) describes a database system that does some automatic disabbreviation on data entry and uses some of these ideas.) Three "word abbreviation" methods are used: (1) match to a standard disabbreviation, previous user replacement, or "analogy" (see below), as confirmed by lookup in the corresponding hash table; (2) truncation on its right end of some word in the program's comments, as long as at least two letters remain and at least two letters are eliminated; and (3) elimination of vowels in a comment word, provided at least two letters remain including the first. Then a "full abbreviation" is either one (like "mgr"), two (like "asstmgr"), or three (like "actasstmgr") "word abbreviations" appended together. These methods together explain about 98 percent of abbreviations we observed in example programs, as abbreviations rarely get more complicated than three-word, although we do have ways to find longer abbreviations as discussed below. For an |m|-letter word, word-abbreviation method (1) generates |O (1)| abbreviations for matching, method (2) |O (m)|, and method (3) |O (1)|; so the whole algorithm generates |O (m sup 3)| full abbreviations. Cubic behavior was observed to be the limit if processing were to be fast enough.

A key feature of our approach to disabbreviation of programs is the restriction of word-abbreviation methods (2) and (3) to words in the comments on the source program, supplemented by optional application-domain words. With a 29000-word dictionary, there would be 25,000,000,000,000 combinations to abbreviate otherwise. We believe that if a program is properly commented, the comments should contain most of the natural names that would be appropriate to use in its variable and procedure names beyond the standard programming words like "pointer". But we do not insist that comment words be near their abbreviations, as comments can have global scope and can reference backward as well as forward. Note that word-abbreviation methods (2) and (3) leave intact the initial letter of a word or phrase, consistent with the programs we studied, and that eliminates many possibilities.

Since a word can have many abbreviations, the nondeterminism of Prolog is valuable, as it permits backtracking with depth-first search to get further disabbreviations until the user finds one acceptable (although the user can set a parameter limiting the number of suggested deabbreviations per word.) The above ordering of the methods is used heuristically in this depth-first search, so that direct hash lookups are tried before deleting characters, single-word abbreviations before two-word, and two-word before three-word. These heuristics derive from cost analysis of the methods. With two-word and three-word methods, an additional heuristic, based on study of example programs, considers rightmost splits first, so "tempvar" would be first split into "tempv" and "ar", then "temp" and "var".

Control structure

The control structure of the disabbreviation tool has similarities to those of spelling correctors like Peterson (1980), text-error detectors like Kukich (1992), and copy-editor assistants like Dale (1989). But abbreviated words are less like their source words than misspelled words are like theirs, so a disabbreviator requires more computation per word; and abbreviating loses so much more information than misspelling that we must rely on a guess-and-test approach.

Processing has three passes. On the first pass, the input text is read in, comment words more than 3 characters are stored, and non-comment words are looked up in the dictionary. Reserved words of the programming language, words within quotation marks, words containing numbers, and words already analyzed are ignored. Morphology and case considerations complicate the lookup. To simplify matters, our dictionary and standard abbreviations are kept in lower case, and comment and unknown non-comment words are converted to lower case for matching. Similarly, most of the dictionary is minus the standard English suffixes "s", "ed", and "ing", and unknown words for comparison are stripped of any such endings, using the appropriate morphological rules of English for undoubling of final consanants, adding a final "e", and changing "i" to "y". The unknown word and all stripped words are tried separately, to catch misleading words like "string" and "fuss".

The second pass then individually considers the words that did not match on the first pass. Full abbreviations are generated to try to match the given word. Plurals are also tried to match to the given word or pieces of it. Possible disabbreviations are individually displayed to the user for approval; if the user rejects them all, the user is asked if the word can be added to the dictionary. If not, the user must supply a disabbreviation replacement, which is recursively confirmed to contain only dictionary words, allowing underscores for punctuation. Replacements can be specified either global for the entire source text or local to the procedure definition. Note that the second pass insists that every input word must either be found a disabbreviation replacement or be confirmed as a dictionary word.

Alternative disabbreviations for a word can be ranked from word and abbreviation-method frequencies, and presented to the user in order (as Uthurusamy et al (1993)). We used ordering for the caption data in Experiment 3. But we did not use it for programs because comments were too sparse to get useful word frequencies, and it adds considerably to processing time.

The third pass changes the names in the source text as decided on the second pass. Name collisions can occur if user-approved disabbreviations are identical with words already used in the text, and the user is then given a warning.

Analogies

Replacements can also be inferred from analogies, and this is one of the most important features of the tool. For instance, if the user confirms that they want to replace "buftempfoobar" with "buffer_temporary_foobar", we can infer that a replacement for "buf" is "buffer" and for "temp" is "temporary", even if we do not recognize "buf" and "foobar". Previous user replacements can be found within an analogy too, to simplify matters, and multiple disabbreviations of the same abbreviation must be permitted to be inferred (although at lesser priority than user-sanctioned disabbreviations). We discovered that analogies are very helpful in disabbreviating real programs because user names are often interrelated. Such analogies should be found after a replacement is approved by the user, not during the generation of disabbreviations when there would be too many possibilities to consider. Caching of analogies that are rejected is important to prevent requerying them.

Replacements can also be inferred from analogies between two previous replacements. For instance, if the user confirmed that "temp" should be replaced by "temporary", and then confirmed "tempo" should be replaced by "temporary_operator", we should infer that one possible replacement for "o" here is "operator" even though "tempo" is an English word, "tempo" is a truncation of "temporary", and one-word abbreviations are quite ambiguous. Then if "tempb" occurs later and "b" is known to be replaceable by "buffer", the disabbreviation "temporary_buffer" will be the first guess. Analogizing can also infer disabbreviations more than three words long that the standard abbreviation rules cannot; for instance, if "tempg" is "temporary_global", "var" is "variable", and "nm" is "name", then "tempgvarnm" is "temporary_global_variable_name". With such analogizing, the system can become progressively more able to guess what the user means.

Acronyms are not recognized by the methods of section 2.2, so analogies are especially important for them, to find them within abbreviations. Acronyms were rare in programs we studied. But they were so common in the picture captions of Experiment 3 that we wrote additional rules to decipher them, explained below.

Previous user replacements (whether generated by the program or by the user) can be very helpful in analogies. We give the user the option of keeping the replacements (both explicit and inferred) from earlier text that was disabbreviated, so that related text can be treated similarly.

Experiments

Experiment 1: Recall and precision of the tool on programs

To test the tool on computer programs, 15 C programs were disabbreviated by the second author, 7 Prolog programs were disabbreviated by the first author, and two large Prolog programs were disabbreviated by Thomas Galvin. The programs were written by a variety of programmers for a variety of applications. CPU time averaged about 1.5 seconds per source-program word with semi-compiled Quintus Prolog running on a Sun SparcStation, with significant inter-program variation. The second, third, and fourth columns of Table I show tool performance. The precision is the ratio of the number of accepted disabbreviations to the number of proposed disabbreviations; the recall is the ratio of the number of accepted disabbreviations to the number of abbreviations in the input, the latter being the sum of the number of accepted disabbreviations and the number of explicit user replacements (assuming that all replacements were disabbreviations). The number of unrecognized symbols is less than the sum of the four integers below it because of analogy-finding. So the tool guesses an acceptable replacement about half the time, and eventually finds an acceptable replacement about three quarters of the time. It can also save users effort, since at 1.5 seconds real computation time per word on a dedicated machine, and 1 second on average per user decision, the C programs we tested for instance required about 3,200 seconds real time to scan 20,990 symbols, an effort that would have required near 20,990 far more tedious seconds if the programs were scanned manually. Our experimental results also indicate that multiple possible disabbreviations are not common, and support our decision not to rank disabbreviations of a word with programs.

Experiment 2: Comprehensibility of tool output

Another issue is whether disabbreviated programs produced by the tool are actually better than the original abbreviated programs. We tested some tool output on 24 subjects, third-quarter students of a M.S. program in Computer Science. We used two programs written previously for other purposes, one in C of 373 symbols,

and one in Prolog of 118. Half the subjects got the original C program and the disabbreviation of the Prolog program; the other half got the disabbreviation of the C program and the original Prolog program. We asked 10 multiple-choice comprehension questions in 20 minutes about the C program and 8 questions in 15 minutes about the Prolog program. Some questions asked about purposes ("Why are there four cases for median_counts?"), and some about execution behavior ("What happens if the word being inserted into the lexicon is there already?"). Subject performance was better on both disabbreviated programs: 7.69/10 versus 7.17/10 for the C program, and 4.64/8 vs. 3.62/8 for the Prolog program. Using Fisher's 2-by-2 test, we confirmed that output of the disabbreviator gave higher comprehensibility at significance level 0.048 (or a 4.8% chance of error).

Experiment 3: Disabbreviation of photograph captions

As a quite different test, we applied the tool to the full set of 36,191 military-photograph captions whose subset we studied in Rowe and Guglielmo (1993). These photographs cover the full scope of activities at a Navy base in 1991. 31,955 distinct symbols occur in the captions; a symbol was defined as something articulated by spaces, commas, colons, semicolons, or period-space combinations. Of these, 6,845 remain after we exclude those found in our natural-language-processing dictionary, morphological variants of words in the dictionary, recognizable codes like those for dates and identification numbers, the domain-specific wordlist of the previous work (763 words), recognizable person names (1861), recognizable place names (191), recognizable corporation names (118), punctuation errors involving insertion of punctuation characters within words (137), and single-letter spelling errors (1056).

Removal of spelling errors improves the success rate in subsequent disabbreviation of caption text. (It is unnecessary for debugged programs.) We did not use a standard spelling-correction tool because our domain has many specialized terms. Spelling correction here required another rule-based system; we only considered words 5 letters or more where transposition of two adjacent letters, deletion of a letter, insertion of a letter, or change of a letter could create a recognizable word. Example misspellings that we recognized are "reservior", "facilites", and "perimenter". We used a frequency ratio test developed by experiment, for statistics on the frequency |f sub m| of occurrence of the allegedly misspelled word and the frequency |f sub c| of its alleged correction, that |f sub c / (f sub m - 0.5) < 35 / n| where |n| is the number of letters in the correction; the 35 is changed to 10 for corrections recognized in the Unix Ispell dictionary. This ratio test reflects the theory that misspellings should be rarer than correct spellings, but that the frequency ratio should decrease with the frequency of the correct spelling and the number of letters in the correct spelling. The precision in spelling correction was 1063/1107 = 0.960, and this is critical; an example mistake was "Texas" for "Texan" since "Texas" occurs frequently. 86 definitely misspelled words (including 14 with multiple errors) were missed, but this overestimates recall because many mistypings of code names can only be detected by domain experts.

With automatically-found misspellings removed (but not all misspellings, which would require considerable work), we can turn to the abbreviations. The Wordnet thesaurus system of Miller et al (1990), from which we obtained information about the 15,000 common-English words used in the captions, provides 960 abbreviation-disabbreviation synonym pairs, of the words mentioned in the captions, where one word can be created from the other by vowel elimination or by right-truncation like "atm" for "atmosphere". Of these 960, all but 52 (for a precision of 0.95) are appropriate abbreviations for this domain.

Some acronyms have disabbreviations given by appositives in the captions, as in "project jmem, joint munitions effectiveness manual" or with extra prepositions, conjunctions, or articles as in "sam - the surface

to air missile". Many of these involve more than one letter from a word, like "alvrj, advanced low volume ramjet", requiring further rules to decipher. Such analysis is valuable because acronyms are so difficult to decipher otherwise. When our implementation was run on the caption data, both retrieval precision and recall were high (see the second column from the right in Table I), although there were errors like suggesting "antenna dome" as a disabbreviation of "and", and recall could be an overestimate since expertise concerning some of these pictures is quite rare and we had to assess relevance ourselves.

The rightmost column of Table I shows results for disabbreviation of the remaining words using the methods of Section 2. Examples found are "negatives" for "negs" and "military contract" for "milcon". We did not make our task easy: "Comment words" were interpreted as all 17,000 words of our natural-language lexicon. We did not exploit any standard abbreviations or reserved words since the domain was so broad. We ignored words with hyphens or digits as they are likely to be arbitrary codes, and we only considered words of three or more letters. We also limited search for multiword disabbreviations to two-word phrases since they were rare in this domain; we ruled out disabbreviations that break the abbreviation into two acceptable English nouns, like "outstanding stationery" for "outstation". Because of the large universe of discourse, we weighted disabbreviations, and only showed the best five to the user. Weighting was the product of three factors: the frequency of the replacement word or words (or if none known, a decreasing function of the word length), whether the replacement could be found in a standard English lexicon (which excluded person and equipment names, and most place names), and whether the replacement had been approved before. Recall was good but precision was low (0.149), reflecting the large number of code words and the remaining spelling errors.

We suggest that our respellings and disabbreviations be alternative keywords for information retrieval, not replacements, due to the chance they are incorrect.

Conclusion

Our experience and experiments show that a disabbreviation tool with inferencing can definitely improve the comprehensibility of text, especially computer programs. The tool is easy to use since most user interaction is just confirming a choice, and its computational requirements are modest. Its coverage is also complete, in that every abbreviated word must be presented to the user and must be assigned a valid disabbreviation. Our tool also shows success in recall for the harder task of deciphering abbreviations in unrestricted technical captions. Our tool could be educational, since it shows people how ambiguous their abbreviations are. But most importantly, it could help in the costly task of software maintenance.

References

Bourne, C. P. and Ford, D. F. (1961). A study of methods for systematically abbreviating English words and names. *Journal of the ACM, 8*, 538-552.

Dale, R. (1989). Computer-based editorial aids. In *Recent developments and applications of natural language processing*, London: Kogan Page, 8-22.

Ibrahim, A. M. (1989, March). Acronyms observed. *IEEE Transactions on Professional Communication, 32(1)*, 27-28.

Kukich, K. (1992, December). Techniques for automatically correcting words in text. *ACM Computing*

Surveys, 24(2), 377-439.

Laitinen, K. (December, 1992). Using natural naming in programming: feedback from practitioners. Proceedings of the Fifth Workshop of the Psychology of Programming Interest Group, Paris. Also included in VTT Research Notes 1498, The principle of natural naming in software documentation, Technical Research Centre of Finland, 1993.

Laitinen, K. and Mukari, T. (1992). DNN -- Disciplined Natural Naming. Proceedings of the 25th Hawaii International Conference on System Sciences, vol. II, 91-100.

Miller, G., Beckwith, R., Fellbaum, C., Gross, D., and Miller, K. (1990, Winter). Five papers on Wordnet. *International Journal of Lexicography, 3(4)*.

Peterson, J. L. (1980, December). Computer programs for detecting and correcting spelling errors. *Communications of the ACM, 23(12)*, 676-687.

Rosenberg, J. M. (1992). *McGraw-Hill dictionary of information technology and computer acronyms, initials, and abbreviations*. New York: McGraw-Hill.

Rowe, N. and Guglielmo, E. (1993). Exploiting captions in retrieval of multimedia data. *Information Processing and Management, 29(4)*, 453-461.

Schneiderman, B. (1980). *Software psychology: human factors in computer and information systems*. Cambridge, MA: Winthrop.

Uthurusamy, R., Means, L. G., Godden, K. S., and Lytinen, S. L. (1993, December). Extracting knowledge from diagnostic databases. *IEEE Expert, 8(6)*, 27-38.

Weissman, L. M. (1974). A methodology for studying the psychological complexity of computer programs. Ph.D. thesis, Dept. of Computer Science, University of Toronto.

Experiment	*C programs (run by Laitinen)*	*Prolog programs (run by Rowe)*		*Prolog programs (run by T. Galvin)*		*Caption acronyms (run by Rowe)*	*Other caption abbreviations (run by Rowe)*
Symbols	*20,990*	*5,850*	*16,975*	*24,500*	*24,500*		
Unrecognized symbols	*1096*	*605*	*683*	*6983*	*6440*		
Accepted disabbreviations	*598*	*371*	*359*	*543*	*145*		
Rejected disabbreviations	*434*	*123*	*570*	*18*	*831*		
Explicit user replacements	*175*	*140*	*69*	*153*	*59*		
New words added to dictionary	*274*	*9*	*255*	*--*	*--*		
Precision	*0.579*	*0.751*	*0.386*	*0.968*	*0.149*		
Recall	*0.774*	*0.726*	*0.839*	*0.780*	*0.711*		

Table I: Experimental results

Go up to paper index