Programming Techniques

S.L. Graham, R.L. Rivest, Editors

A Technique for Isolating Differences Between Files

Paul Heckel
Interactive Systems Consultants

A simple algorithm is described for isolating the differences between two files. One application is the comparing of two versions of a source program or other file in order to display all differences. The algorithm isolates differences in a way that corresponds closely to our intuitive notion of difference, is easy to implement, and is computationally efficient, with time linear in the file length. For most applications the algorithm isolates differences similar to those isolated by the longest common subsequence. Another application of this algorithm merges files containing independently generated changes into a single file. The algorithm can also be used to generate efficient encodings of a file in the form of the differences between itself and a given "datum" file, permitting reconstruction of the original file from the difference and datum files.

Key Words and Phrases: difference isolation, word processing, text editing, program maintenance, hashcoding, file compression, bandwidth compression, longest common subsequence, file comparison, molecular evolution

CR Categories: 3.63, 3.73, 3.81, 4.43

Author's Address: Interactive Systems Consultants, P.O. Box 2345, Palo Alto, CA 94305.

1. Introduction

The problem of finding all differences between two files occurs in several contexts. This discussion is directed toward building a tool that locates differences in source programs and other text files. Once understood, the technique described is transferable to related applications.

A tool that compares different versions of text files can be used for source program maintenance in at least four ways: (1) as an editing aid, to verify modifications and detect spurious edits; (2) as a debugging aid, to find the differences between a version of a program that is known to work and one that does not work; (3) as a program maintenance aid for locating and merging program changes introduced by several different people (or organizations); and (4) as a quality control tool to highlight the differences between the output of the latest version of a program and the output of an earlier version.

This tool is also useful when documents, or any text files, are edited and maintained on a computer. Text-editing and word-processing systems such as QED [2] and WYLBUR [4] rarely have a file-compare capability. This capability is useful when new editions of documents are issued in which all changes from a previous edition are to be underlined or otherwise identified.

2. Current Methods of File Comparison

Two other methods of file comparison are known. In the first, the files are scanned, and whenever a difference is detected, the next M lines are scanned until N consecutive identical lines are found. DEC's FILECOM [3] is implemented this way. This technique has two disadvantages: (1) M and/or N usually have to be tuned by the user to get useful output, and (2) if a whole block of text is moved, then all of it, rather than just the beginning and end, is detected as changed. This can be misleading, for any changes made within a moved block are not highlighted.

The algorithm described here avoids these difficulties. It detects differences that correspond very closely to our intuitive notion of difference. In particular, if a block of text is moved, only its beginning and end are detected as different, and any other differences within the moved block will be highlighted.

The second method is the longest common subsequence (LCS) method, which has received some attention in the literature [1, 6, 7, 10]. Given strings (or

files) $A = a_1a_2 \ldots a_n$, $B = b_1b_2 \ldots b_m$, and $C = c_1c_2 \ldots c_p$, C is a common subsequence of A and B if $n - p$ characters can be deleted from string A to produce string C and $m - p$ characters can be deleted from string B to produce string C. Given strings A and B, the LCS problem is to find the longest string C that is a common subsequence of A and B. When this is done, the file comparison output can be generated by simultaneously scanning strings A, B, and C, flagging characters that appear in A but not in C one way, and flagging those that appear in B but not in C another way.

The LCS method has strong appeal: It is a simple formal statement of the problem that yields good results. However, it has two basically different disadvantages. The first is that it is not necessarily the correct formalization of the problem. In the two examples below, the longest common subsequence is probably not what is desired. In these examples, the differences are displayed by underlining the inserted characters and putting minus signs through the deleted ones. In the second example, the parentheses indicate a block of characters that has been moved relative to its neighboring blocks.

OLD STRING	NEW STRING	LCS METHOD	"BETTER"
AXCYDWEABE	ABCDE	~~A~~X~~B~~C~~Y~~D~~W~~E~~A~~B~~E~~	~~AXCYDWEABCDE~~
ABCDEG	DEFGAC	~~ABC~~DE~~F~~G~~AC~~	(DE~~F~~G) (~~ABC~~)

In the first example, the LCS finds four deleted strings and one inserted one, while the "better" display shows one deleted string and one inserted one. (Note: Minimization of the number of inserts plus deletes is not a good formulation of the problem since the differences between any two strings can be trivially displayed as one deleted string and one inserted one.) In the second example, the LCS method does not find a moved block of characters (it never does), but the "better" display both shows the moved block and finds edits within the blocks. The algorithm described here produces the "better" display in both examples.

Counterexamples in which the algorithm described here produces poorer results are easy to construct. The author's opinion is that there is no "correct" formulation of the problem, just as there is no "correct" way to determine which of two equivalent algebraic expressions is simpler. In both cases it depends on the preferences of the person reading it.

The second disadvantage of the LCS method concerns the computational problems associated with it [1, 6, 7]. In the worst case, it can take time $O(mn)$ – that is, time proportional to the product of the lengths of the two strings. The only implementation of LCS for file comparison of which the author is aware [7] takes linear time and space for most practical cases but takes worst-case $O(mn\log n)$ time and $O(mn)$ space. As a practical LCS implementation, it uses several clever techniques and is neither concise nor easy to understand.

The method described here avoids these computational problems. It is concise and easy to understand and takes linear time and space for all cases. It also produces output that is probably as good as or better than the LCS method in most practical cases.

3. The Algorithm

To compare two files, it is usually convenient to take a line as the basic unit, although other units are possible, such as word, sentence, paragraph, or character. We approach the problem by finding similarities until only differences remain. We make two observations:

1. *A line that occurs once and only once in each file must be the same line (unchanged but possibly moved).* This "finds" most lines and thus excludes them from further consideration.
2. *If in each file immediately adjacent to a "found" line pair there are lines identical to each other, these lines must be the same line.* Repeated application will "find" sequences of unchanged lines.

The algorithm acts on two files, which we call the "old" file O and the "new" file N. Our conception is that O was changed to produce N, although the algorithm is virtually symmetric. There are three data structures: a symbol table and two arrays OA and NA. We use the text of the line as a symbol to key the symbol table entries. Each entry has two counters OC and NC specifying the number of copies of that line in files O and N, respectively. These counters need only contain the values 0, 1, and many. The symbol table entry also has a field $OLNO$, which contains the line number of the line in the "old" file. It is of interest only if $OC = 1$.

The array OA (NA) has one entry for each line of file O (N); it contains either a pointer to the line's symbol table entry or the line's number in file N (O) and a bit to specify which.

The algorithm consists of six simple passes. Pass 1 comprises the following: (a) each line i of file N is read in sequence; (b) a symbol table entry for each line i is created if it does not already exist; (c) NC for the line's symbol table entry is incremented; and (d) $NA[i]$ is set to point to the symbol table entry of line i.

Pass 2 is identical to pass 1 except that it acts on file O, array OA, and counter OC, and $OLNO$ for the symbol table entry is set to the line's number.

In pass 3 we use observation 1 and process only those lines having $NC = OC = 1$. Since each represents (we assume) the same unmodified line, for each we replace the symbol table pointers in NA and OA by

the number of the line in the other file. For example, if $NA[i]$ corresponds to such a line, we look $NA[i]$ up in the symbol table and set $NA[i]$ to $OLNO$ and $OA[OLNO]$ to i. In pass 3 we also "find" unique virtual lines immediately before the first and immediately after the last lines of the files.

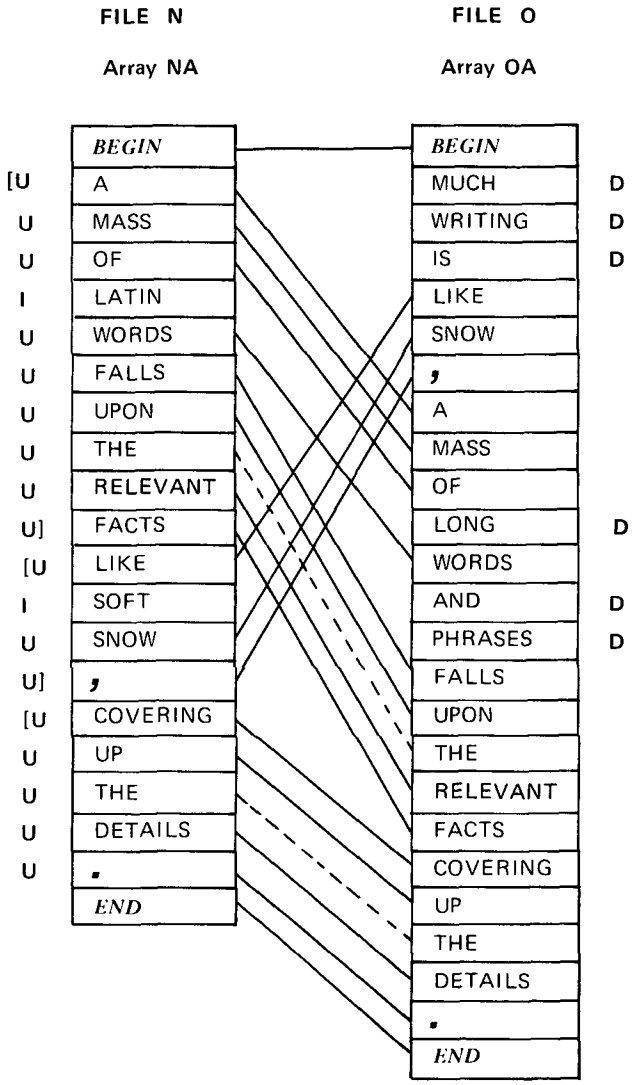
In pass 4, we apply observation 2 and process each line in NA in ascending order: If $NA[i]$ points to $OA[j]$ and $NA[i+1]$ and $OA[j+1]$ contain identical symbol table entry pointers, then $OA[j+1]$ is set to line $i+1$ and $NA[i+1]$ is set to line $j+1$.

In pass 5, we also apply observation 2 and process each entry in descending order: if $NA[i]$ points to $OA[j]$ and $NA[i-1]$ and $OA[j-1]$ contain identical symbol table pointers, then $NA[i-1]$ is replaced by $j-1$ and $OA[j-1]$ is replaced by $i-1$.

Array NA now contains the information needed to list (or encode) the differences: If $NA[i]$ points to a symbol table entry, then we assume that line i is an insert, and we can flag it as new text. If it points to $OA[j]$, but $NA[i+1]$ does not point to $OA[j+1]$, then line i is at the boundary of a deletion or block move and can be flagged as such. In the final pass, the file is output with its changes described in a form appropriate to a particular application environment. Figure 1 illustrates how this works.

A formal version of this algorithm is presented in an earlier version of this paper [5]. It outputs each line flagged by the type of change it finds: insert, delete, beginning of block, end of block, or unchanged.

Fig. 1. Difference isolation. The hashcodes for the lines in the files being compared are the corresponding entries of arrays OA and NA. The first and last locations are used for unique generated begin and end lines. In pass 3, all unique lines are connected (solid lines). In passes 4 and 5, identical but nonunique lines are found (dashed lines). The file comparison can then be generated. Each insert (I), delete (D), unchanged line (U), or moved block ([and]) can be detected and printed out. The output could be printed as:



4. Analysis of Potential Problems

The technique described here is prone to detecting false differences. Consider an unchanged sequence of lines none of which is unique. If the lines immediately above and below the sequence are changed, they will be (correctly) detected as different, but the unchanged sequence of lines between them will also be (falsely) marked as different.

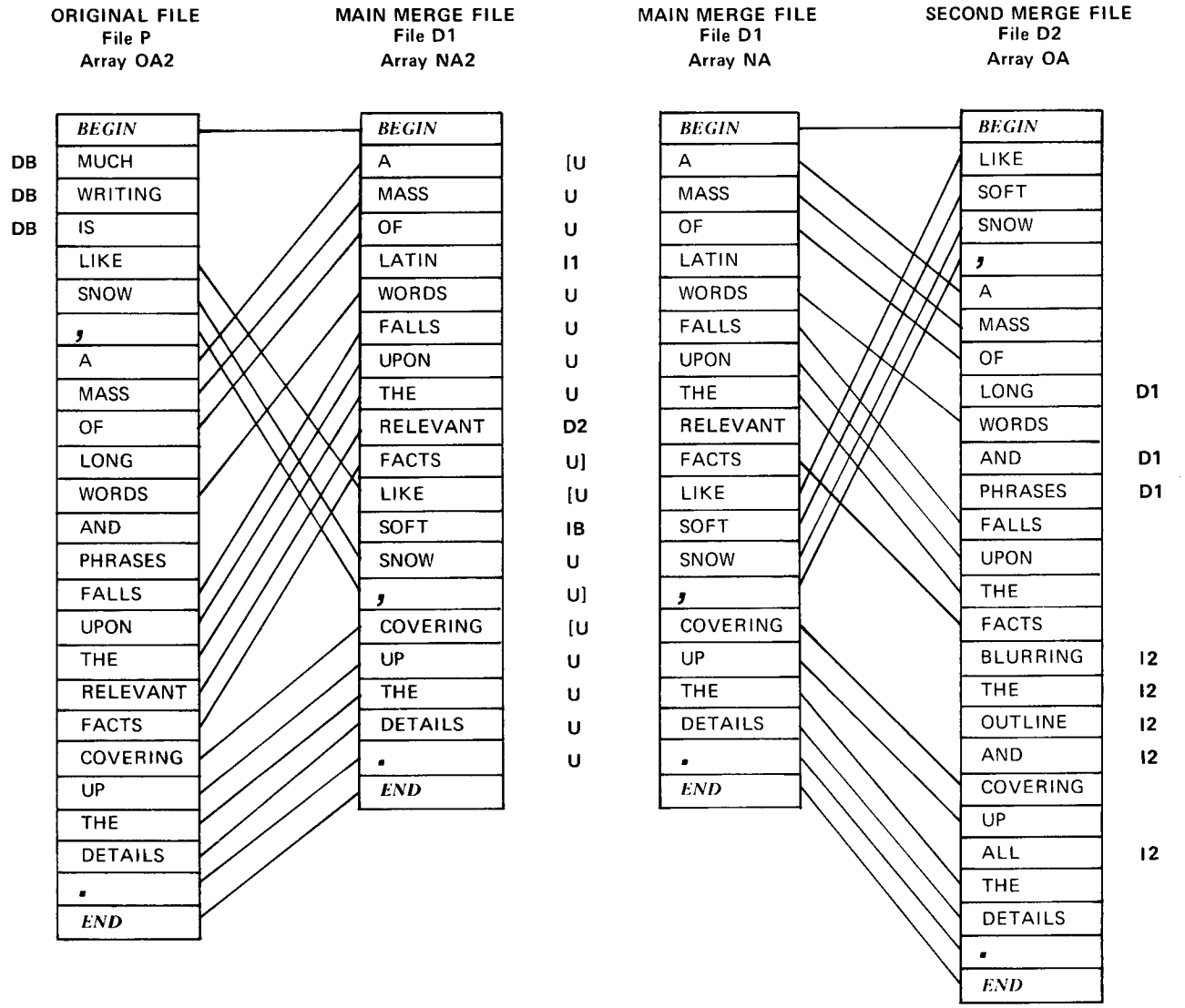
Because they occur at points to which attention is directed, false differences, if few in number, are annoying rather than serious. For applications where the files in question have less convenient characteristics, a different basic unit or a hierarchy of basic units could be chosen.[1]

The technique described can be modified by using a hashcode [9] as a surrogate for the characters in the line. Hashcoding buys greater speed, program simplicity, and storage efficiency at the cost of letting some changes go undetected. A good hashcoding algorithm with a large number of potential hashcodes will keep the number of undetected changes small. For example, a good 20-bit hashcode would mean that about one change in a million would go undetected. If a single line is changed, the chance it will be changed to a line with the same hashcode is $1/2^{20}$. A change to a line having the same hashcode as some other line would, at worst, detect the line as moved when, in fact, it had been changed. A complete analysis is beyond the scope of this paper.

This technique can be used on very large files. The symbol table is the only randomly accessed data structure that grows with the size of the files. The size of each symbol table entry can be reduced to two bits by combining the functions of NC and OC into one field and eliminating $OLNO$.

[1] For example, a three-line block could be chosen as the basic unit, with the exclusive or of the hashcodes for lines 1 to 3, 2 to 4, 3 to 5, etc. forming the indices into the hash table. Since the index for lines $i+1$ to $j+1$ can be computed with two exclusive ors from the index for lines i to j and the hashcode for line i, the time it takes to perform the algorithm for a k-line block is independent of k.

Fig. 2. File merging. We have three files, P, D1, and D2; P is the parent file, D1 and D2 are the descendant files, with D1 being the dominant file. Differences are isolated between D1 (in *NA*) and D2 (in *OA*), and between D1 (in *NA*2) and P (in *OA*2). The result is shown by the solid lines. The merged file can be generated by iterating through *NA* to generate lines that are: unchanged (U); deleted from file 1 (D1), file 2 (D2), both files (DB); inserted into file 1 (I1), file 2 (I2), both files (IB); or on block boundaries ([and]). The output could be printed as:



One seems to face a choice between a hashcode size having a large number of bits and minimizing undetected changes and one that has a small number and allows direct addressing of the hash table entry. However, one can get the advantages of both by using a double hashcode technique, where a long hashcode is stored in *NA* and *OA* and is rehashed into a smaller sized hashcode for accessing the symbol table directly. The number of extra entries required for such a symbol table depends on the number of differences in the files being compared. If the number of symbol table entries is two or three times the number of different lines, very few false differences will be detected unless the files have a great many changes. This modification means that the marginal storage cost can be reduced to three to six bits per unique line.

5. Encoding Files

Difference isolation can be used to produce a differential encoding of one of two versions of a file. This encoding can be stored or transmitted more efficiently than the complete file. The original file can later be reconstructed from its differential encoding and the other version. Two applications are: (1) A file system can keep several versions of a file by storing one version *in toto* and the others as differential encodings; and (2) copies of files on remote computers can be updated with minimum communications cost.

The analysis of the problem of undetected changes for the encoding process is different from that for source comparison since the decoding algorithm will not tolerate an occasional false moved block, while the

user who reads a source comparison listing will. Two techniques can be used to mitigate this problem. First, checksums of the original and reconstructed files can be computed and compared, and, if they are not the same, the original file can be transmitted in full. Second, the extra pass, which "unfinds" any single line blocks, can be inserted into the algorithm. Most undetected changes, since they look like single line blocks, would be eliminated at the small cost of transmitting all single line blocks.

Efficient encoding can be done even if the old version is not saved on the "new file" computer. The encoding algorithm does not require the text of the old file, but only the array of hashcodes OA. The array of hashcodes can be transmitted from the "old file" computer to the new file computer and the encoding sent back. In general, this procedure would require less bandwidth than would be needed to transmit the whole file.

The bandwidth used can be reduced even more by using a recursive scheme with hierarchy of basic units (e.g. chapters, sections, paragraphs, and sentences). The "new file" computer would load the array NA, and the "old file" computer would load the array OA with hashcodes. During phase 1, the "old file" computer would scan OA to calculate the hashcodes for all the chapters and then transmit them with the beginning and ending line numbers. The "new file" computer would then compare these hashcodes with those that it calculated for the chapters; differences can be isolated at the chapter level, with each line of the chapter linked for each identical chapter found. In the second phase, only the changed chapters are divided into sections, and their hashcodes computed and transmitted in like manner. Only during the last phase is the unchanged basic unit (lines) transmitted if it is detected as changed.

6. Merging Files

Another application of difference isolation involves the merging of text changes. This can happen when an organization maintains its own version of a vendor's program and wishes to merge vendor changes with its own whenever a new version of the program is distributed. IBM has such a system [8], but all changes must be especially encoded by the user. The method for generating the merged output should be fairly easy to understand from Figure 2; an algorithm for this purpose is given in the appendix to [5]. Where blocks of lines have been moved, the order of the output lines is determined by choosing one of the files as the main file. Thus two possible merged files can be generated, depending on which of the modified files is chosen as dominant.

Acknowledgments. The algorithm described here was implemented on an XDS-940 about 5 years ago. More recently, Bill Frantz of Tymshare implemented both PL/1 and 370 machine language versions. Codie Wells made some technical suggestions including the double hashcode and recursive schemes. Mark Halpern, Butler Lampson, Tom Weston, Glenn Manacher, and Bonnie Simrell made several useful editorial suggestions which have improved the presentation of the ideas in this paper. Glenn Manacher also pointed out the relevance of the longest common subsequence problem [1, 6, 7, 10].

Received April 1976; revised December 1976

References

1. Aho, A., Hirschberg, D., and Ullman, J. Bounds on the complexity of the longest common subsequence problem. *J. ACM 23*, 1 (Jan. 1976), 1-12.
2. Deutsch, P., and Lampson, B. An online editor. *Comm. ACM 10*, 12 (Dec. 1967), 793-799.
3. Digital Equipment Corp. DEC System 10 Assembly Language Handbook, 3d ed., 1972, pp. 931-942.
4. Fajman, R., and Borgelt, J. WYLBUR: An interactive text editing and remote job entry system. *Comm. ACM 16*, 5 (May 1973), 314-322.
5. Heckel, P. A technique for isolating differences between files. Tech. Pub. 73, Interactive Systems Consultants, Palo Alto, Calif.
6. Hirschberg, D. A linear space algorithm for computing maximal common subsequences. *Comm. ACM 18*, 6 (June 1975), 342-343.
7. Hunt, J., and McIlroy, M. An algorithm for differential file comparison. Compt. Sci. Techn. Rep. 41, Bell Telephone Labs, Murray Hill, N.J., Aug. 1976.
8. IBM Corp. IBM Virtual Machine Facility/370 Command Language Guide for General Users, Release 2, 225-226.1 (UPDATE).
9. Knuth, D.E. *The Art of Computer Programming, Vol. 3: Sorting and Searching*. Addison-Wesley, Reading, Mass., 1973, p. 509.
10. Wagner, R., and Fischer, M. The string-to-string correction problem. *J. ACM 21*, 1 (Jan. 1974), 168-173.