SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 15(11), 1025–1040 (NOVEMBER 1985)

A File Comparison Program

WEBB MILLER AND EUGENE W. MYERS
Department of Computer Science, University of Arizona, Tucson, AZ 85721, U.S.A.

SUMMARY

This paper presents a simple method for computing a shortest sequence of insertion and deletion commands that converts one given file to another. The method is particularly efficient when the difference between the two files is small compared to the files' lengths. In experiments performed on typical files, the program often ran four times faster than the UNIX *diff* command.

KEY WORDS Edit distance Edit script File comparison

INTRODUCTION

A file comparison program produces a list of differences between two files. These differences can be couched in terms of lines, e.g. by telling which lines must be inserted, deleted or moved to convert the first file to the second. Alternatively, the list of differences can identify individual bytes. Byte-oriented comparisons are useful with non-text files, such as compiled programs, that are not divided into lines.

The approach adopted here is to generate only instructions to insert or delete entire lines. Since lines are treated as indivisible objects, files can be treated as containing lines consisting of a single symbol. In other words, an n-line file is modelled by a string of n symbols.

In more formal terms, the file comparison problem can be rephrased as follows. The *edit distance* between two strings of symbols is the length of a shortest sequence of insertions and deletions that will convert the first string to the second. The goal, then, is to write a program that computes the edit distance between two arbitrary strings of symbols. In addition, the program must explicitly produce a shortest possible edit script (i.e. sequence of edit commands) for the given strings.

Other approaches have been tried. For example, Tichy[1] discusses a file-comparison tool that determines how one file can be constructed from another by copying blocks of lines and appending lines. However, the ability to economically generate shortest-possible edit scripts depends critically on the repertoire of instructions that are allowed in the scripts.[2]

File comparison algorithms have a number of potential uses beside merely producing a set of edit commands to be read by someone trying to understand the evolution of a program or document. For example, the edit scripts might be text editor instructions that are saved to avoid the expense of storing nearly identical files. Rather than storing

0038–0644/85/111025–16$01.60

Received 1 October 1984
Revised 21 January 1985

two long files, just one of the files and a (presumably short) file containing instructions like

```
replace lines 6–8 by the line "*s++ = *t++;"
```

is stored. Rochkind[3] and Tichy[4] discuss version control systems based on this technique.

As further testimony to the range of uses for file comparison techniques, one of the earliest algorithms for the problem was invented simultaneously by biologists interested in comparing long molecules such as proteins and by speech processing experts trying to compare spoken words with known words.[5] These algorithms have also been used in information retrieval systems to cope with spelling mistakes[6] and for video redisplay, where the problem is to send the computer terminal a minimal set of display modification commands that will bring the image up to date.[7]

STRAIGHTFORWARD APPROACHES

The simplest method of file comparison is to look through the two files line by line until they disagree, then search forward in the files until a matching pair of lines is found. Regardless of the strategy for resynchronization, this simple approach suffers from the defect of sometimes producing edit scripts that are much longer than necessary.

To see what goes awry, let the first string consist of n repetitions of the six symbols $axxbxx$ and let the second string be derived by adding bxx to the front of the first string. Most of the simple resynchronization strategies will match xs. For the case $n = 1$, the matching looks as follows:

Simple strategy

```
axxbxx
 ||  ||
bxxaxxbxx
```

Optimal strategy

```
axxbxx
  \\\\\
bxxaxxbxx
```

For each of the n segments in the general case, a and b are deleted, then inserted in the opposite order; bxx is inserted at the end. This produces an edit script of length $4n + 3$. On the other hand, the minimal edit script just inserts bxx at the front.

In practice, it is common for file comparison programs to require that several consecutive lines match before resynchronization. In the above example, the same edit script of length $4n + 3$ is produced even if resynchronization requires that two contiguous symbols must match. But if three symbols must match before resynchronization is achieved, then the optimal script is found. None the less, the example can be modified to show that any resynchronization strategy that looks for k aligning symbols is not optimal. Moreover, for large k more time is required and matching substrings of length less than k will be missed.

The failure of simple file comparison algorithms is more than just a theoretical curiosity; it can easily happen in practice. For example, suppose that a procedure is added to the beginning of a source file for a program and this file is then compared

against the original file. Where is the first point that lines of the files match? Quite possibly, the match occurs at the ends of the first procedures in each file where there are, for example, several blank lines. If the file comparison program resynchronizes at this point by, in effect, removing the first procedure from each file, then the resulting situation is the same as at the start: *file2* is just *file1* with an additional procedure tacked on the front. Thus the algorithm may report that the two files are entirely different, except for blank lines.

AN EFFICIENT ALGORITHM

The algorithm described below always produces a shortest possible edit script. It works very well on files where the differences are small, and poorly only when the files are quite different. These performance characteristics make for a very efficient comparison tool in the frequently occurring situations where differences are expected to be small. An earlier algorithm[8] enjoys the same general property, but is substantially more complicated. The detailed analysis verifying the correctness and time complexity of the algorithm is deferred until the next section.

All the information needed to compute the edit distance between strings A and B can be determined by comparing every element of A with every element of B. Thus, denoting the length of A by m and the length of B by n, $m \times n$ comparisons suffice. A systematic approach is developed that uses three rules to build up a solution from the solutions to the subproblems that are obtained by considering initial segments of the given strings. Which subproblems need to be solved cannot be determined until the final solution is in hand. This fact is confirmed by the straightforward algorithms, which fail because they fix on specific edit instructions before the two files are completely known.

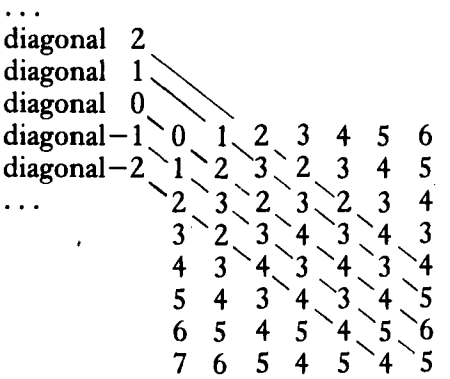
Let $D[i,j]$ be the edit distance between the first i symbols of A (denoted $A[1:i]$) and the first j symbols of B (denoted $B[1:j]$). $D[i,j]$ makes sense even when i or j is zero; for example $D[i,0]$ is the edit distance between a string of i symbols and a string of 0 symbols, which obviously equals i. These values are arranged as a matrix with $1+m$ rows (one row giving the values $D[0,j]$ and one row for each entry of A) and $1+n$ columns (one column giving the values $D[i,0]$ and one for each entry of B). For example, if $A =$ *abcabba* and $B =$ *cbabac*, the matrix of edit distances is

0	1	2	3	4	5	6	
1	2	3	2	3	4	5	*a*
2	3	2	3	2	3	4	*b*
3	2	3	4	3	4	3	*c*
4	3	4	3	4	3	4	*a*
5	4	3	4	3	4	5	*b*
6	5	4	5	4	5	6	*b*
7	6	5	4	5	4	5	*a*
	c	*b*	*a*	*b*	*a*	*c*	

In this example, the entry $D[5,4]$, which lies at the intersection of row 5 and column

4, (row and column numbers start with 0) is the edit distance between *abcab* (the string labelling rows 1–5) and *cbab* (the string labelling columns 1–4). The value in that position is 3 because *abcab* can be transformed into *cbab* by deleting the leading *a* and the *c* from the first string, then inserting a *c* at the front, but there is no shorter edit script for the transformation. (There may be more than one shortest edit script for a given pair of strings; in this case, delete *a* and *b* and insert a *b*.)

The above matrix exhibits several useful patterns that appear for any choice of the two input strings. The *i*th row (for any *i*) begins with *i*, and each pair of adjacent values differs by 1. The same is true of columns. Diagonals, however, have a different structure. To make this precise, number the diagonals of D as follows.



The values of D along diagonal k begin with $|k|$, then jump to $|k| + 2$, then to $|k| + 4$, and so on. The validity of this pattern follows from another of particular algorithmic interest: all occurrences of a given value d are on diagonals $-d$, $-d+2$, ..., $d-2$ and d. That is, the entries with value d lie along alternate diagonals in a band of half-width d, centered around diagonal 0. A formal proof of these observations is not given here but can be inferred from the algorithm's proof of correctness given beow.

The file comparison algorithm systematically constructs a solution by using three rules to fill values in the D matrix. Along with each value $D[i,j]$, the algorithm accumulates an edit script of length $D[i,j]$ that converts $A[1:i]$ to $B[1:j]$. The algorithm begins operation by determining all entries in the D matrix that are 0. As is evident, these entries are just the values $D[i,i]$ on diagonal 0 where $A[k] = B[k]$ for all $k \leq i$. In other words, the algorithm starts by finding identical prefixes of A and B. Then the algorithm applies the three rules to determine all entries in D that equal 1. Then it fills in the 2s, then the 3s and so on. This continues until the 'south-east' value $D[m,n]$ is determined, at which point the algorithm has found a shortest possible edit script for converting the first input string to the second. In specifying the rules, the notation $A[i]$ denotes the ith symbol of A and $B[j]$ denotes the jth symbol of B.

Rule 1: 'Move right'

Suppose that:

(i) $D[i, j-1]$ (the value just to the left of $D[i, j]$) is known.

(ii) An edit script of length $D[i, j-1]$ that converts $A[1:i]$ to $B[1:j-1]$ is known.

(iii) $D[i, j]$ is unknown.

Then $D[i, j] = D[i, j-1] + 1$, and adding the command 'Insert $B[j]$ after symbol i' to the edit script of (ii) produces a shortest edit script for converting $A[1:i]$ to $B[1:j]$.

If the algorithm has determined the value of $D[i, j-1]$, but not that of $D[i, j]$, then $D[i,j]$ must be greater than $D[i,j-1]$. Consequently, $D[i,j]$ must equal $D[i,j-1]+1$, since the rule shows how to construct a script of this length.

As an example, consider $D[3,3]$, the edit distance from *abc* to *cba* in the sample problem. Suppose that (i) $D[3,2]$, the edit distance from *abc* to *cb*, is 3, that (ii)

Delete symbol 1
Delete symbol 2
Insert *b* after symbol 3

is a shortest possible edit script for converting *abc* to *cb*, and that (iii) $D[3,3]$ has not been filled in. Rule 1 then asserts that $D[3,3]$ must equal 4, and appending the command 'Insert *a* after symbol 3' to the edit script given in (ii) yields a shortest edit script for converting *abc* to *cba*.

Edit commands refer to symbol positions in the *original* string. Thus the second delete command in the script above removes the *b* in *abc* and not the *c*. Also, a sequence of insertions after the same position are assumed to occur in order. Thus the first insert command in the script above places a *b* after the *c* in *abc* and the second 'Insert after symbol 3' places an *a* after the *b* just inserted. These 'parallel' scripts are not sequentially executable by common text editors but are oriented towards users, who cannot retain the intermediate states of a long edit. None the less, these scripts perform the desired transformation if executed sequentially in reverse order.

Rule 2: 'Move down'

Suppose that:
(i) $D[i-1, j]$ (the value just above $D[i, j]$) is known.
(ii) An edit script of length $D[i-1, j]$ that converts $A[1:i-1]$ to $B[1:j]$ is known.
(iii) $D[i, j]$ is unknown.
Then $D[i, j] = D[i-1, j] + 1$, and adding the command 'Delete symbol i' to the edit script of (ii) produces a shortest edit script for converting $A[1:i]$ to $B[1:j]$.

As an example, again consider $D[3,3]$ in the sample problem, which is the edit distance from *abc* to *cba*. Suppose that (i) $D[2,3]$, the edit distance from *ab* to *cba*, is 3, that (ii)

Delete symbol 1
Insert *c* after symbol 1
Insert *a* after symbol 2

is a shortest possible edit script for converting *ab* to *cba*, and that (iii) $D[3,3]$ has not been filled in. Rule 2 then asserts that $D\ [3,3] = 4$, and appending the command 'Delete symbol 3' to the edit script given in (ii) yields a shortest edit script for converting *abc* to *cba*.

Rule 3. 'Slide down the diagonal'

Suppose that

(i) $D[i-1, j-1]$) (the value just above and to the left of $D[i,j]$) is known.

(ii) An edit script of length $D[i-1, j-1]$ that converts $A[1: i-1]$ to $B[1: j-1]$ is known.

(iii) $A[i] = B[j]$.

Then $D[i, j] = D[i-1, j-1]$, and the edit script of (ii) is a shortest edit script for converting $A[1: i]$ to $B[1: j]$.

As an example, look at $D[5,4]$ in the sample problem. Rule 3 says that any edit script of length $D[4,3]$ for *abca* and *cba*, such as

Delete symbol 1
Delete symbol 2
Insert a *b* after symbol 3

is an edit script for *abcab* and *cbab* of length $D[5,4]$

For an illustration of how values of D are determined, suppose that the 0s, 1s and 2s have been filled in the sample matrix of edit distances:

0	1	2	.	.	.	.	
1	2	.	2	.	.	.	*a*
2	.	2	.	2	.	.	*b*
.	2	.	.	.	.	.	*c*
.	.	.	.	.	.	.	*a*
.	.	.	.	.	.	.	*b*
.	.	.	.	.	.	.	*b*
.	.	.	.	.	.	.	*a*
	c	*b*	*a*	*b*	*a*	*c*	

Now apply Rules 1–3 to see how far 3s extend down diagonal 1. Rules 1 and 2 imply that any unfilled position that is immediately to the right of, or immediately below, a 2 must contain a 3. Thus these rules place 3s in positions [1,2], [2,3] and [3,4]. Rule 3 then permits a 3 to fill in the [4,5] location, since the row and column labels there are both *a*, and since this position has a 3 immediately to its north-west. The same sort of reasoning fills in the 3s shown on the next page in diagonals -3, -1 and 3.

The example illustrates that the algorithm finds the d entries of the D matrix by moving right or down from a $d-1$ entry and then making a sequence of diagonal moves. It follows that d values are filled in only on diagonals $-d$, $-d+2$, . . . ,$d-2$,d. As already mentioned, all 0s go on diagonal 0. The 1s are filled in by either moving down from diagonal 0 to diagonal -1, or moving right from diagonal 0 to diagonal 1, then sliding down a diagonal.

0	1	2	3	.	.	.	
1	2	3	2	3	.	.	*a*
2	3	2	3	2	3	.	*b*
3	2	3	.	3	.	3	*c*
.	3	.	3	.	3	.	*a*
.	.	3	.	3	.	.	*b*
.	.	.	.	.	.	.	*b*
.	.	.	.	.	.	.	*a*
	c	*b*	*a*	*b*	*a*	*c*	

For the 2s, the algorithm moves right or down from diagonals ± 1, from which it can reach only diagonals -2, 0 and 2, then slides diagonally. Continuing inductively shows the general pattern to be true.

For efficiency, not all the entries of D are explicitly computed and stored. The algorithm only records those d entries nearest the bottom of D in each relevant diagonal (i.e. diagonals $-d$, $-d+2$, ..., $d-2, d$). The notation *last_d*[k] indicates the last row containing the most recent value of d to be filled in diagonal k. This simplification is valid since the sought-after corner $D[m,n]$ is such an entry. Furthermore, since the diagonals containing $(d-1)$s alternate with diagonals containing ds, the algorithm can use the same array to hold both the $(d-1)$ entries and the d entries it is computing from them.

Assuming that *last_d*[$k-1$] and *last_d*[$k+1$] indicate the positions of the last $(d-1)$s on diagonals $k-1$ and $k+1$, how is the last d on diagonal k located? The first problem is to find a value of d on diagonal k. One possibility is to move right from the last $d-1$ in diagonal $k-1$ to the row *last_d*[$k-1$] on diagonal k. The other possibility is to move down from the last $d-1$ in diagonal $k+1$ to the row *last_d*[$k+1$] + 1 on diagonal k. The algorithm selects the more advantageous of the two moves. Thus if *last_d*[$k+1$] $\geq$ *last_d*[$k-1$] it moves down, otherwise it moves right. (Special care is needed for $k = \pm d$, i.e. for the diagonals that have $(d-1)$s on only one side.) Once on diagonal k, the algorithm slides as far as is permitted by Rule 3.

For an example, return to the problem of filling 3s in diagonal 1 of the sample problem. Just after the 2s have been filled in, *last_d*[0] = 2 and *last_d*[2] = 2. Moving right from diagonal 0 would yield a 3 in row 2 of diagonal 1, whereas moving down from diagonal 2 yields a 3 in row 3. Hence the algorithm applies the move down rule to reach position [3,4], slides down to row 4 using Rule 3, and sets last_d[1] to 4.

Given this method of determining edit distances, producing a corresponding edit script is straightforward. An edit script denoted *script*[k] is associated with each *last_d*[k], and updated according to Rules 1–3. The algorithm fragment that locates the last d on diagonal k is given in Figure 1.

The test

```
if (k = −d or (k ≠ d and last_d[k+1] ≥ last_d[k−1]))
```

```
/* Find a d on diagonal k. */
if (k = −d or (k ≠ d and last_d[k+1] ≥ last_d[k−1])){
     /* Moving down from the last d−1 on diagonal k+1 */
     /* puts you further along diagonal k than does */
     /* moving right from the last d−1 on diagonal k−1. */
     row = last_d[k+1] + 1
     script[k] = script[k+1] with the command
       'Delete the rowth symbol' appended
} else {
     /* Move right from the last d−1 on diagonal k−1. */
     row = last_d[k−1]
     script[k] = script[k−1] with the command
       'Insert B[row+k] after the rowth symbol' appended
}
/* Column where row intersects diagonal k */
col = row + k
/* Slide down the diagonal. */
while (row < m and col < n and A[row+1] =B[col+1]) {
     row = row + 1
     col = col + 1
}
last_d[k] = row
```

Figure 1. Locating the last d on diagonal k

takes special care with the cases $k = \pm d$. When k is $-d$, the algorithm is working on the lowest diagonal that contains ds. The diagonal above it (diagonal $k+1$) contains one or more $(d-1)$s, but the diagonal just below (diagonal $k-1$) contains none, and *last_d*$[k-1]$ is not defined. The clause 'k = −d' guarantees that the move down rule will be applied. Similarly, the clause 'k ≠ d' guarantees that the move right rule is applied when k equals d.

The next step is to enclose the above algorithm fragment in loops that vary d and k appropriately. Perhaps the first pair of looping statements that one might try is:

```
for d = 1,2,3, ...
    for k = −d, −d+2, ... , d−2, d
```

However, an alternative approach (Figure 2) helps to guarantee that array references stay in bounds and further restricts the 'search band' of the algorithm. For example, suppose *row* reaches m, meaning that the algorithm has hit the bottom of the D matrix, say on diagonal k. It is pointless to fill in values to the left of diagonal k, since they cannot contribute to the value of $D[m,n]$. The algorithm arranges that diagonal $k+1$ will be the lowest diagonal that is considered when d is incremented. This is done by using a pair of bounds, *lower* and *upper*, that give the range of diagonals to consider at each iteration. Normally, *lower* is decremented by one and *upper* is incremented by one in each iteration. However, in the situation above the algorithm sets *lower* to $k+2$ and lets

```
/* Initialize: 0 entries in D indicate identical prefixes */
row = min (i:A[i+1] ≠ B [i+1])
last_d[0] = row
script[0] = NULL
if (row = m) lower = 1 else lower = −1
if (row = n) upper = −1 else upper = 1
if (lower > upper)
    Report that the files are identical and terminate execution

/* for each value of the edit distance */
for d = 1,2, . . . , max_d {
    /* for each relevant diagonal */
    for k = lower, lower + 2, . . . , upper − 2, upper {

        Locate the last d on diagonal k, as in Figure 1.

        if (row = m and col = n)
            Print the edit script pointed to by script[k]
            and terminate execution.
        if (row = m)
            /* Hit last row; don't look to the left. */
            lower = k + 2
        if (col = n)
            /* Hit last column; don't look to the right. */
            upper = k − 2
    }
    lower = lower − 1
    upper = upper + 1
}
Print a message indicating that the edit distance is greater than max_d.
```

Figure 2. The high-level structure of the algorithm

the default decrement set it correctly to $k+1$. The bound *upper* is similarly set when *col* reaches n on a given diagonal.

The algorithm terminates after *max_d* iterations of the outermost loop. If there are no edit scripts of length at most *max_d*, then the algorithm quits and reports this fact. This characteristic is desirable in the common case that one does not want to see the difference if it is too large to be useful. If *max_d* is set to $m + n$, the algorithm finds the difference regardless of its size.

An implementation of the complete algorithm in C is given in the Appendix.

ANALYSIS OF THE ALGORITHM

Correctness

The algorithm employs the simple strategy of finding each d entry by moving right or down from a $d-1$ entry and then making a sequence of diagonal moves. It has yet to be shown that this strategy correctly fills in the D matrix and hence that the algorithm is

correct. An inductive argument is needed to show that at stage d every d entry in the D matrix is found by the strategy. In effect, an argument about *all* edit scripts is required. Certainly, scripts that insert and subsequently delete a given symbol are not optimal and need not be considered. For the remaining scripts, assume without loss of generality that the commands of a script are ordered according to the position in A affected by the command. The scripts produced by Rules 1–3 have this property.

Suppose that S is a shortest edit script of length d for entry $D[i,j]$. For the basis, $d=0$, it follows that $i=j$ and $A[k] = B[k]$ for all $k \leq i$. But the algorithm correctly fills in these entries in its initialization step. Proceeding inductively, suppose that $d > 0$ and let R be the first $d-1$ commands of S. The algorithm's strategy is correct if R is a shortest script for an entry $D[p,q]$ that can reach $D[i,j]$ via a move right or down and a sequence of diagonal moves. Suppose that the last command of S is 'Delete symbol k', where k is between 1 and i. Then the last $i-k$ symbols of $A[1:i]$ must match the last $i-k$ symbols of $B[1:j]$. Thus S must also be a shortest script for converting $A[1:k]$ into $B[1:j-(i-k)]$. If it were not, then the shorter script would, by Rule 3, be a shorter script for $D[i,j]$, which is a contradiction. It then follows that R must be a shortest script for converting $A[1:k-1]$ into $B[1:j-(i-k)]$. Again, if it were not then the shorter script plus the command 'Delete symbol k' would, by Rules 1 and 3, be a shorter script for $D[i,j]$, which is a contradiction. But entry $D[i,j]$ can be reached from $D[k-1, j-(i-k)]$ by a move down and a sequence of diagonal moves. Similar reasoning reveals that if the last command of S is 'Insert an x after symbol k' where k is between 1 and i and x is arbitrary, then R is a shortest script for converting $A[1:k]$ to $B[1:j-(i-k)-1]$. But again, entry $D[i,j]$ can be reached from $D[k,j-(i-k)-1]$ by a move right and a sequence of diagonal moves. Thus the strategy and algorithm are correct.

A more detailed discussion of the formal issues for the edit distance problem and its generalizations is given in Reference 9.

Efficiency

At worst, the algorithm must determine all the entries of D, hence its running time is at worst proportional to $m \times n$. This is expected, since it is known that for every one of a certain large class of file comparison algorithms there exists some pair of files for which the algorithm takes time proportional to $m \times n$.[10]

The advantage of the algorithm is that it performs efficiently when the size of the output is small compared to m and n. Since the algorithm stops as soon as $D[m,n]$ is filled in, only diagonals $-d$ to d are considered, where $d = D[m,n]$. Thus the running time is proportional to the number of entries on those diagonals, which is less than $(2d+1)\min(m,n)$. Moreover, the expected running time is proportional to $\min(m,n) + d^2$ under appropriate distributional assumptions.[11]

Most file comparison algorithms do not perform this well when the two files are nearly identical. Some take time proportional to $m \times n$ regardless of d.[12] Other algorithms depend heavily on the number r of pairs $[i,j]$ where $A[i]$ equals $B[j]$, which may be large in cases where d is small.[13]

An algorithm whose performance depends on r is used in the UNIX *diff* command.[13, 14] To show how disastrous this dependence on r can be, *diff* was compared to *fcomp*, the new algorithm's implementation as listed in the Appendix. Both programs were run with *file 1* consisting of 1000 blank lines, and *file 2* consisting of *file 1* with a single non-blank line added to both ends. This choice makes $d = 2$ and $r = 10^6$. The *fcomp* program took

about half a second of computer time (on a VAX 11/780), whereas *diff* took over 2·5 minutes.

To turn the tables, the two programs were run on a pair of 1000-line files with no lines in common. This choice makes $d = 2000$ and $r = 0$; *fcomp* ran out of memory after one minute (and reported that the edit distance was at least 617), whereas *diff* solved the problem in about 8 seconds. (This mode of failure for *fcomp* can be avoided since the algorithm can be modified to run in space proportional to $m+n$.[11]) Thus there exist pathological cases where *fcomp*'s performance is much worse than *diff*'s, and vice versa.

To obtain a more realistic idea of how *fcomp* compares with *diff*, figures were gathered for their relative performance comparing 1000-line files of C programs, with various values of d between 5 and 50. Typical values of r were 10,000–20,000. (If one out of every 10 lines were blank, then the pairs of blank lines, one from each file, would contribute 10,000 to r.) On these problems, *fcomp* typically ran about 4 times faster than *diff*. The only circumstances in which this trend was broken were cases when all differences between the files fell in a small range of lines. *diff* begins operation by stripping away lines that match at the fronts and rears of the two files. For example, if all differences between the two files occur in lines 1–100, *diff* quickly reduces the problem to that of comparing two 100-line files, then applies its main algorithm. For such problems, *fcomp* ran about 2–3 times faster than *diff*.

It is notoriously difficult to judge the relative merits of two programs, since performance often depends critically on the data and the programming details. To evaluate the effect of coding differences, *diff*'s underlying algorithm was implemented in the spirit of *fcomp*. In particular, each line of the two files was saved using a call to the storage allocation routine *malloc*. (To conserve space, *diff* stores internally only the lines' hash values. This strategy costs the time to compute those values and to read each file twice, but speeds the process of comparing lines.) Again, *fcomp* was more efficient by a factor of around four. The program listing given in the Appendix will facilitate efforts to corroborate, extend or invalidate this experimental conclusion.

ACKNOWLEDGEMENTS

We would like to thank Chris Fraser, Gary Levin and Titus Purdin for suggesting several improvements to the paper. Walter Tichy pointed out References 5 and 8 and the referees made several helpful suggestions. This work was supported by National Science Foundation under Grants MCS82–10722 and MCS82–10096.

APPENDIX

The following program pp. 1036–1039 implements the file comparison algorithm in C.

```
/*
* fcomp - a file comparison program
*
* A command line has the form
*       fcomp [-n] file1 file2
* where the optional flag  n  is an integer constant that limits the size of
* edit scripts that will be considered by fcomp.  If all edit scripts changing
* file1 to file2 contain more than  n  insertions and deletions, then a message
* to that effect is all that is printed.  If no  n  is specified, then arbitrarily
* long edit scripts are considered.
*/

#include <stdio.h>
#define MAXLINES 2000                   /* maximum number of lines in a file */
#define ORIGIN  MAXLINES                /* subscript for diagonal 0 */
#define INSERT 1
#define DELETE 2

/* edit scripts are stored in linked lists */
struct edit {
        struct edit *link;              /* previous edit command */
        int op;                         /* INSERT or DELETE */
        int line1;                      /* line number in file1 */
        int line2;                      /* line number in file2 */
};

char *A[MAXLINES], *B[MAXLINES];        /* pointers to lines of file1 and file2 */

main(argc, argv)
int argc;
char *argv[];
{
        int     max_d,                  /* bound on size of edit script */
                m,                      /* number of lines in file1 */
                n,                      /* number of lines in file2 */
                lower,                  /* left-most diagonal under consideration */
                upper,                  /* right-most diagonal under consideration */
                d,                      /* current edit distance */
                k,                      /* current diagonal */
                row,                    /* row number */
                col;                    /* column number */

        /* for each diagonal, two items are saved: */
        int last_d[2*MAXLINES+1];       /* the row containing the last d */
        struct edit *script[2*MAXLINES+1];   /* corresponding edit script */

        struct edit *new;
        char *malloc();

        if (argc > 1 && argv[1][0] == '-') {
                max_d = atoi(&argv[1][1]);
                ++argv;
                --argc;
        } else
                max_d = 2*MAXLINES;
        if (argc != 3)
                fatal("Fcomp requires two file names.");
        /* Read in file1 and file2. */
        m = in_file(argv[1], A);
        n = in_file(argv[2], B);
```

```
/* Initialize: 0 entries in D indicate identical prefixes. */
for (row = 0; row < m && row < n && strcmp(A[row], B[row]) == 0; ++row)
        ;
last_d[ORIGIN] = row;
script[ORIGIN] = NULL;
lower = (row == m) ? ORIGIN + 1 : ORIGIN - 1;
upper = (row == n) ? ORIGIN - 1 : ORIGIN + 1;
if (lower > upper) {
        puts("The files are identical.");
        exit(0);
}

/* for each value of the edit distance */
for (d = 1; d <= max_d ; ++d) {
        /* for each relevant diagonal */
        for (k = lower; k <= upper; k += 2) {
                /* Get space for the next edit instruction. */
                new = (struct edit *) malloc(sizeof(struct edit));
                if (new == NULL)
                        exceed(d);

                /* Find a d on diagonal k. */
                if (k == ORIGIN-d || k != ORIGIN+d && last_d[k+1] >= last_d[k-1]) {
                        /*
                        * Moving down from the last d-1 on diagonal k+1
                        * puts you farther along diagonal k than does
                        * moving right from the last d-1 on diagonal k-1.
                        */
                        row = last_d[k+1]+1;
                        new->link = script[k+1];
                        new->op = DELETE;
                } else {
                        /* Move right from the last d-1 on diagonal k-1. */
                        row = last_d[k-1];
                        new->link = script[k-1];
                        new->op = INSERT;
                }
                /* Code common to the two cases. */
                new->line1 = row;
                new->line2 = col = row + k - ORIGIN;
                script[k] = new;

                /* Slide down the diagonal. */
                while (row < m && col < n && strcmp(A[row], B[col]) == 0) {
                        ++row;
                        ++col;
                }
                last_d[k] = row;

                if (row == m && col == n) {
                        /* Hit southeast corner; have the answer. */
                        put_scr(script[k]);
                        exit(0);
                }
                if (row == m)
                        /* Hit last row; don't look to the left. */
                        lower = k+2;
                if (col == n)
                        /* Hit last column; don't look to the right. */
                        upper = k-2;
```

```
                }
                --lower;
                ++upper;
        }
        exceed(d);
}

/* in_file - read in a file and return a count of the lines */
in_file(filename, P)
char *filename, *P[];
{
        char buf[100], *malloc(), *fgets(), *save, *b;
        FILE *fp, *fopen();
        int lines = 0;

        if ((fp = fopen(filename, "r")) == NULL) {
                fprintf(stderr, "Cannot open file %s.\n", filename);
                exit(1);
        }
        while (fgets(buf, 100, fp) != NULL) {
                if (lines >= MAXLINES)
                        fatal("File is too large for fcomp.");
                if ((save = malloc(strlen(buf)+1)) == NULL)
                        fatal("Not enough room to save the files.");
                P[lines++] = save;
                for (b = buf; *save++ = *b++; ) /* copy the line */
                        ;
        }
        fclose(fp);
        return(lines);
}

/* put_scr - print the edit script */
put_scr(start)
struct edit *start;
{
        struct edit *ep, *behind, *ahead, *a, *b;
        int change;

        /* Reverse the pointers. */
        ahead = start;
        ep = NULL;
        while (ahead != NULL) {
                behind = ep;
                ep = ahead;
                ahead = ahead->link;
                ep->link = behind;      /* Flip the pointer. */
        }

        /* Print commands. */
        while (ep != NULL) {
                b = ep;
                if (ep->op == INSERT)
                        printf("Inserted after line %d:\n", ep->line1);
                else {  /* DELETE */
                        /* Look for a block of consecutive deleted lines. */
                        do {
                                a = b;
                                b = b->link;
                        } while (b!=NULL && b->op==DELETE && b->line1==a->line1+1);
                        /* Now b points to the command after the last deletion. */
```

```
                change = (b!=NULL && b->op==INSERT && b->line1==a->line1);
                if (change)
                        printf("Changed ");
                else
                        printf("Deleted ");
                if (a == ep)
                        printf("line %d:\n", ep->line1);
                else
                        printf("lines %d-%d:\n", ep->line1, a->line1);
                /* Print the deleted lines. */
                do {
                        printf("  %s", A[ep->line1-1]);
                        ep = ep->link;
                } while (ep != b);
                if (!change)
                        continue;
                printf("To:\n");
        }
        /* Print the inserted lines. */
        do {
                printf("  %s", B[ep->line2-1]);
                ep = ep->link;
        } while (ep != NULL && ep->op == INSERT && ep->line1 == b->line1);
    }
}

/* fatal - print error message and die */
fatal(msg)
char *msg;
{
    fprintf(stderr, "%s\n", msg);
    exit(1);
}

/* exceed - the difference exceeds d */
exceed(d)
int d;
{
    fprintf(stderr, "The files differ in at least %d lines.\n", d);
    exit(1);
}
```

REFERENCES

1. W. Tichy, 'The string-to-string correction problem with block moves', *ACM Trans. Comp. Systems*, **2,** (4), 309–321 (1984).
2. M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Complete Problems*, W. H. Freeman, 1979.
3. M. J. Rochkind, 'The source code control system', *IEEE Trans. Software Engineering*, **1,** (4), 364–370 (1975).
4. W. Tichy, 'RCS — a system for version control', *Software — Practice and Experience*, **15**, (7), 637–654 (1985).
5. D. Sankoff and J. B. Kruskal, *Time Warps, String Edits and Macromolecules: The Theory and Practice of Sequence Comparison*, Addison-Wesley, 1983.
6. P. A. V. Hall and G. R. Dowling, 'Approximate string matching', *Computing Surveys*, **12,** (4), 381–402 (1980).
7. J. A. Gosling, 'A redisplay algorithm', *SIGPLAN Notices*, **16,** (6), 123–129 (1981).
8. N. Nakatsu, Y. Kambayashi and S. Yajima, 'A longest common subsequence algorithm suitable for similar text strings', *Acta Informatica*, **18,** 171–179 (1982).

9. R. A. Wagner and M. J. Fischer, 'The string-to-string correction problem', *Journal of ACM*, **21**, (1), 168–173 (1974).
10. A. V. Aho, D. S. Hirschberg and J. D. Ullman, 'Bounds on the complexity of the longest common subsequence problem', *Journal of ACM*, **23**, (1), 1–12 (1976).
11. E. W. Myers, 'An O(ND) difference algorithm and its variations', *Technical Report 85–6*, Department of Computer Science, University of Arizona, 1985.
12. D. S. Hirschberg, 'A linear space algorithm for computing maximal common subsequences', *Comm. ACM*, **18**, (6), 341–343 (1975).
13. J. W. Hunt and T. G. Szymanski, 'A fast algorithm for computing longest common subsequences', *Comm. ACM*, **20**, (5), 350–353 (1977).
14. J. W. Hunt and M. D. McIlroy, 'An algorithm for differential file comparison', *Computing Science Technical Report 41*, Bell Laboratories, 1975.