An Algorithm for Differential File Comparison

J. W. Hunt

Department of Electrical Engineering, Stanford University, Stanford, California

M. D. McIlroy

Bell Laboratories, Murray Hill, New Jersey 07974

ABSTRACT

The program *diff* reports differences between two files, expressed as a minimal list of line changes to bring either file into agreement with the other. *Diff* has been engineered to make efficient use of time and space on typical inputs that arise in vetting version-to-version changes in computer-maintained or computer-generated documents. Time and space usage are observed to vary about as the sum of the file lengths on real data, although they are known to vary as the product of the file lengths in the worst case.

The central algorithm of *diff* solves the 'longest common subsequence problem' to find the lines that do not change between files. Practical efficiency is gained by attending only to certain critical 'candidate' matches between the files, the breaking of which would shorten the longest subsequence common to some pair of initial segments of the two files. Various techniques of hashing, presorting into equivalence classes, merging by binary search, and dynamic storage allocation are used to obtain good performance.

[This document was scanned from Bell Laboratories Computing Science Technical Report #41, dated July 1976. Text was converted by OCR and hand-corrected (last changed June, 2012). Figures were reconstructed. Some OCR errors may remain, especially in tables and equations. Please report them to doug@cs.dartmouth.edu.]

The program *diff* creates a list of what lines of one file have to be changed to bring it into agreement with a second file or vice versa. It is based on ideas from several sources[1,2,7,8]. As an example of its work, consider the two files, listed horizontally for brevity:

a	b	c	d	e	f	g
w	a	b	x	y	z	e

It is easy to see that the first file can be made into the second by the following prescription, in which an imaginary line 0 is understood at the beginning of each:

append after line 0:	w,		
change lines 3 through 4, which were:	c	d	
into:	x	y	z,
delete lines 6 through 7, which were:	f	g.	

Going the other way, the first file can be made from the second this way:

delete line 1, which was:	w,		
change lines 4 through 6, which were:	x	y	z
into:	c	d,	
append after line 7:	f	g.	

Delete, change and append are the only operations available to *diff*. It indicates them by 1-letter

abbreviations reminiscent of the qed text editor[3] in a form from which both directions of change can be read off. By exchanging 'a' for 'd' and line numbers of the first file with those of a second, we get a recipe for going the other way. In these recipes lines of the original file are flagged with '<', lines of the derived file are flagged with '>':

0 a 1,1	1,1 d 0
> w	< w
3,4 c 4,6	4,6 c 3,4
< c	< x
< d	< y
---	< z
> x	---
> y	> c
> z	> d
6,7 d 7	7 a 6,7
< f	> f
< g	> g

In mathematical terms, the goal of *diff* is to report the minimum number of line changes necessary to convert one file into the other. Equivalently, the goal is to maximize the number of lines left unchanged, or to find the longest common subsequence of lines that occurs in both files.

1. Solving the longest common subsequence problem

No uniformly good way of solving the longest common subsequence problem is known. The simplest idea—go through both files line by line until they disagree, then search forward somehow in both until a matching pair of lines is encountered, and continue similarly—reduces the problem to implementing the 'somehow', which doesn't help much. However, in practical terms, the first step of stripping matching lines from the beginning (and end) is helpful, for when changes are not too pervasive stripping can make inroads into the (nonlinear) running time of the hard part of the problem.

An extremely simple heuristic for the 'somehow', which works well when there are relatively few differences between files and relatively few duplications of lines within one file, has been used by Johnson and others[1, 11]: Upon encountering a difference, compare the kth line ahead in each file with the k lines following the mismatch in the other for k = 1,2,... until a match is found. On more difficult problems, the method can missynchronize badly. To keep a lid on time and space, k is customarily limited, with the result that longer changed passages defeat resynchronization.

There is a simple dynamic programming scheme for the longest common subsequence problem[4,5]. Call the lines of the first file A_i, $i = 1, \ldots, m$ and the lines of the second B_j, $j = 1, \ldots, n$. Let P_{ij} be the length of the longest subsequence common to the first i lines of the first file and the first j lines of the second. Evidently P_{ij} satisfies

$$P_{i0} = 0 \quad i = 0, \ldots, m,$$

$$P_{0j} = 0 \quad j = 0, \ldots, n,$$

$$P_{ij} = \begin{cases} 1 + P_{i-1,j-1} & \text{if } A_i = B_j \\ \max(P_{i-1,j}, P_{i,j-1}) & \text{if } A_i \neq B_j \end{cases} \quad 1 \leq i \leq m,\ 1 \leq j \leq n.$$

Then P_{mn} is the length of the desired longest common subsequence. From the whole P_{ij} array that was generated in calculating P_{mn} it is easy to recover the indices or the elements of a longest common subsequence.

Unfortunately the dynamic program is $O(mn)$ in time, and—even worse—$O(mn)$ in space. Noting that each row P_i of the difference equation is simply determined from P_{i-1}, D. S. Hirschberg invented a clever scheme that first calculates P_m in $O(n)$ space and then recovers the sequence using no more space and about as much time again as is needed to find P_m[6].

The *diff* algorithm improves on the simple dynamic program by attending only to essential matches, the breaking of which would change P. The essential matches, dubbed 'k-candidates' by Hirschberg[7], occur where $A_i = B_j$ and $P_{ij} > \max(P_{i-1,j}, P_{i,j-1})$. A k-candidate is a pair of indices (i, j) such that (1) $A_i = B_j$, (2) a longest common subsequence of length k exists between the first i elements of the first file and the first j elements of the second, and (3) no common subsequence of length k exists when either i or j is reduced. A candidate is a pair of indices that is a k-candidate for some k. Evidently a longest common subsequence can be found among a complete list of candidates.

If (i_1, j_1) and (i_2, j_2) with $i_1 < i_2$ are both k-candidates, then $j_1 > j_2$. For if $j_1 = j_2$, (i_2, j_2) would violate condition (3) of the definition; and if $j_1 < j_2$ then the common subsequence of length k ending with (i_1, j_1) could be extended to a common subsequence of length $k + 1$ ending with (i_2, j_2).

The candidate methods have a simple graphical interpretation. In Figure 1 dots mark grid points (i, j) for which $A_i = B_j$. Because the dots portray an equivalence relation, any two horizontal lines or any two vertical lines on the figure have either no dots in common or carry exactly the same dots. A common subsequence is a set of dots that can be threaded by a strictly monotone increasing curve. Four such curves have been drawn in the figure. These particular curves have been chosen to thread only (and all) dots that are candidates. The values of k for these candidates are indicated by transecting curves of constant k. These latter curves, shown dashed, must all decrease monotonically. The number of candidates is obviously less than mn, except in trivial cases, and in practical file comparison turns out to be very much less, so the list of candidates usually can be stored quite comfortably.

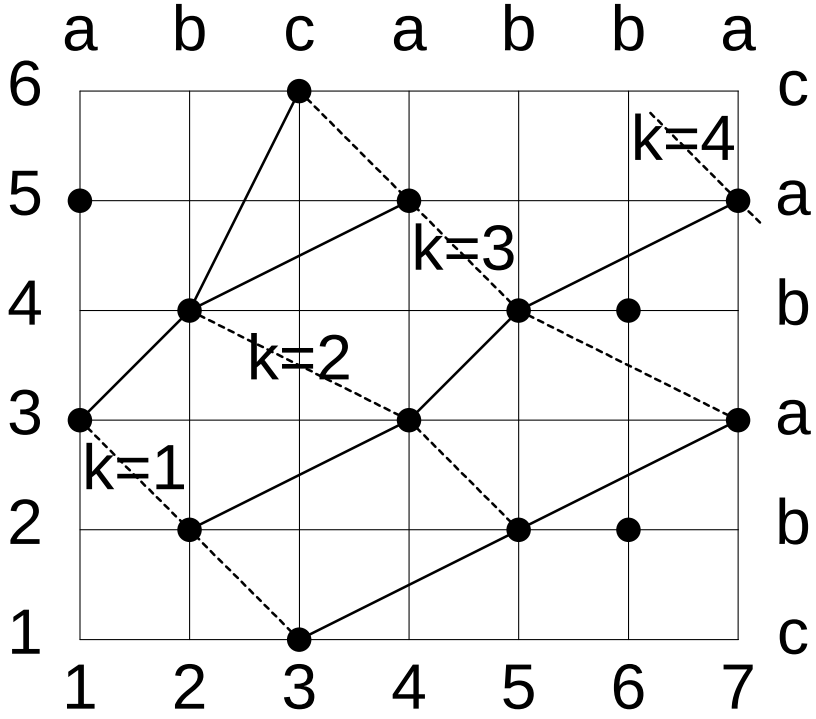


Figure 1. Common subsequences and candidates in comparing

```
abcabba
cbabac
```

2. The method of diff

The dots of Figure 1 are stored in linear space as follows:

(1) Construct lists of the equivalence classes of elements in the second file. These lists occupy $O(n)$ space. They can be made by sorting the lines of the second file.

(2) Associate the appropriate equivalence class with each element of the first file. This association can be stored in $O(m)$ space. In effect now we have a list of the dots for each vertical.

Having this setup, we proceed to generate the candidates left-to-right. Let K be a vector designating the rightmost k-candidate yet seen for each k. To simplify what follows, pad the vector out to include a dummy 0-candidate $(0, 0)$ and, for all k that do not yet have a candidate, a dummy 'fence' candidate $(m+1, n+1)$, whose components will compare high against the components of any other candidate. K begins empty, except for padding, and gets updated as we move right. Thus after processing the 4th vertical, marked 'a' in Figure 1, the list of rightmost candidates is

(0,0) (3,1) (4,3) (4,5) (8,7) (8,7) ...

Now a new k-candidate on the next vertical is the lowest dot that falls properly between the ordinates of the previous $(k-1)$- and k-candidates. Two such dots are on the 5th vertical in Figure 1. They displace the 2-candidate and 3-candidate entries to give the new vector K:

(0,0) (3,1) (5,2) (5,4) (8,7) (8,7) ...

The two dots on the 6th vertical fall on, rather than between, ordinates in this list and so are not candidates. Each new k-candidate is chained to the previous $(k-1)$-candidate to facilitate later recovery of the longest common subsequence. For more detail see the Appendix.

The determination of candidates on a given vertical is thus a specialized merge of the list of dots on that vertical into the current list of rightmost candidates. When the number of dots is $O(1)$, binary search in the list of at most $\min(m, n)$ candidates will do the merge in time $O(\log m)$. Since this case of very few dots per vertical is typical in practice, we are led to merge each dot separately by binary search, even though the worst case time to process a vertical becomes $O(n \log m)$, as against $O(m+n)$ for ordinary merging.

3. Hashing

To make comparison of reasonably large files (thousands of lines) possible in random access memory, *diff* hashes each line into one computer word. This may cause some unequal lines to compare equal. Assuming the hash function is truly random, the probability of a spurious equality on a given comparison that should have turned out unequal is $1/M$, where the hash values range from 1 to M. A longest common subsequence of length k determined from hash values can thus be expected to contain about k/M spurious matches when $k \ll M$, so a sequence of length k will be a spurious 'jackpot' sequence with probability about k/M. On our 16-bit machine jackpots on 5000-line files should happen less than 10% of the time and on 500-line files less than 1% of the time.

Diff guards against jackpots by checking the purported longest common subsequence in the original files. What remains after spurious equalities are edited out is accepted as an answer even though there is a small possibility that it is not actually a longest common subsequence. *Diff* announces jackpots, so these cases tend to get scrutinized fairly hard. In two years we have had brought to our attention only one jackpot where an edited longest subsequence was actually short—in that instance short by one.

Complexity

In the worst case, the *diff* algorithm doesn't perform substantially better than the trivial dynamic program. From Section 2 it follows that the worst case time complexity is dominated by the merging and is in fact $O(mn \log m)$ (although $O(m(m+n))$ could be achieved). Worst case space complexity is dominated by the space required for the candidate list, which is $O(mn)$ as can be seen by counting the candidates that arise in comparing the two files

```
a b c a b c a b c ...
a c b a c b a c b ...
```

This problem is illustrated in Figure 2. When $m = n$ the kite-shaped area in which the candidates lie is 1/2 the total area of the diagram, and (asymptotically} 1/3 of the grid points in the kite are candidates, so the number of candidates approaches $n^2/6$ asymptotically.*

In practice, *diff* works much better than the worst case bounds would indicate. Only rarely are more than $\min(m, n)$ candidates found. In fact an early version with a naive storage allocation algorithm that

* Direct counting shows that there are $\lfloor (4mn - m^2 - n^2 + 2m + 2n + 6)/12 \rfloor$ candidates when $m-1$ and $n-1$ differ by at most a factor of 2. The floor is exact whenever $n-1$ and $m-1$ are multiples of 6

provided space for just *n* candidates first overflowed only after two months of use, during which time it was probably run more than a hundred times. Thus we have good evidence that in a very large percentage of practical cases *diff* requires only linear space.

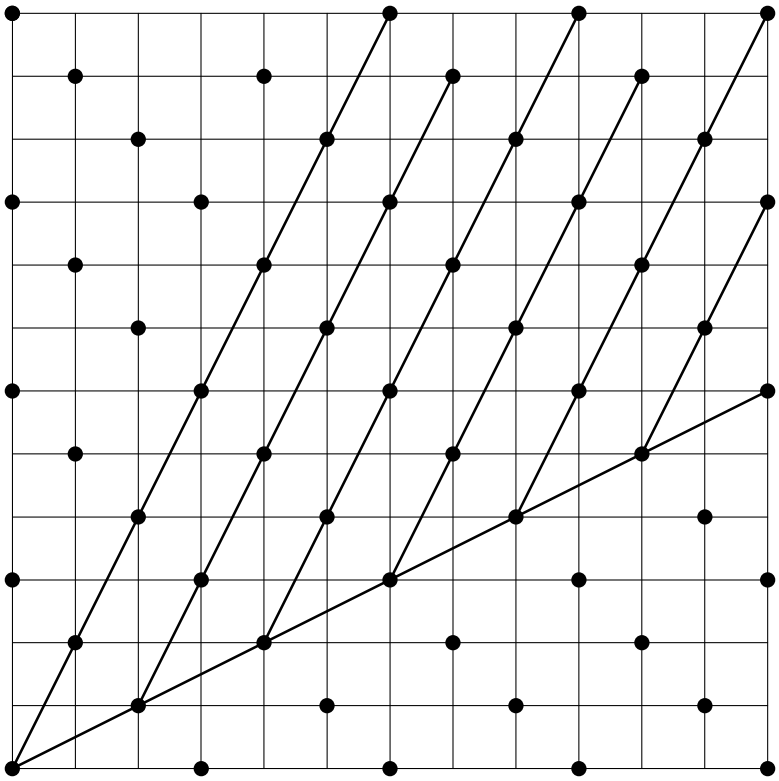


Figure 2. Common subsequences and candidates in comparing

```
a b c a b c a b c ...
a c b a c b a c b ...
```

As for practical time complexity, the central algorithm of *diff* is so fast that even in the biggest cases our implementation can handle (about 3500 lines) almost half the run time is still absorbed by simple character handling for hashing, jackpot checking, etc., that is linear in the total number of characters in the two files. Typical times for comparing 3500-line files range from 1/4 to 3/4 cpu minutes on a PDP11/45. By contrast, a speeded-up variant of Hirschberg's dynamic programming algorithm[6] took about 5 cpu minutes on 3500-line files. The heuristic algorithm sketched at the beginning of Section 1 typically runs about 2 or 3 times as fast as *diff* on long but trivially different files, but loses much of that advantage on more difficult cases that are within the competence of both methods. Since the failure modes of the two programs are quite different, it is useful to have both on hand.

References

[1] S. C. Johnson, ‘ALTER − A Comdeck Comparing Program,’ Bell Laboratories internal memorandum 1971.

[2] Generalizing from a special case solved by T. G Szymanski[8], H. S. Stone proposed and J. W. Hunt refined and implemented the first version of the candidate-listing algorithm used by *diff* and embedded it in an older framework due to M. D. Mcllroy. A variant of this algorithm was also elaborated by Szymanski[10]. We have had many useful discussions with A. V. Aho and J. D. Ullman. M. E. Lesk moved the program from UNIX to OS/360.

[3] ’Tutorial Introduction to QED Text Editor,’ Murray Hill Computing Center MHCC-002.

[4] S. B. Needleman and C. D. Wunsch, ’A General Method Applicable to the Search for Similarities in the Amino Acid Sequence,’ *J Mol Biol* **48** (1970) 443-53.

[5] D. Sankoff, ’Matching Sequences Under Deletion/Insertion Constraints’, *Proc Nat Acad Sci USA* **69** (1972) 4-6.

[6] D. S. Hirschberg, ’A Linear Space Algorithm for Computing Maximal Common Subsequences,’ *CACM* **18** (1975) 341-3.

[7] D. S. Hirschberg, ’The Longest Common Subsequence Problem,’ Doctoral Thesis, Princeton 1975.

[8] T. G Szymanski, ’A Special Case of the Maximal Common Subsequence Problem,’ Computer Science Lab TR-170, Princeton University 1975

[9] Michael L. Fredman, ’On Computing the Length of Longest Increasing Subsequences,’ *Discrete Math* **11** (1975) 29-35.

[10] T. G. Szymanski, ’A Note on the Maximal Common Subsequence Problem,’ submitted for publication. [The paper finally appeared as H. W. Hunt III and T. G. Szymanski, ‘A fast algorithm for computing longest common subsequences’, *CACM* **20** (1977) 350-353.]

[11] The programs called *proof*, written by E. N. Pinson and M. E. Lesk for UNIX and GECOS use the heuristic algorithm for differential file comparison.

Appendix

A.l Summary of the diff algorithm

Algorithm to find the longest subsequence of lines common to file 1, whose length is m lines, and file 2, n lines.

Steps 1 through 4 determine equivalence classes in file 2 and associate them with lines in file 1 in preparation for the central algorithm. (The *diff* program that is in actual use does the work of these steps somewhat differently.)

1. Let V be a vector of elements structured ($serial$, $hash$), where $serial$ is a line number and $hash$ is an integer. Set

$$V[j] \leftarrow (j, H(j)) \quad j = 1, \ldots, n.$$

   where $H(j)$ is the hash value of line j in file 2.

2. Sort V into ascending order on $hash$ as primary key and $serial$ as secondary key.

3. Let E be a vector of elements structured ($serial$, $last$). Then set

$$E[j] \leftarrow (V[j].\,serial,\ f(j)) \quad j = 1, \ldots, n,$$

$$E[0] \leftarrow (0, \textbf{true}),$$

   where

$$f(j) = \begin{cases} \textbf{true} & \text{if } j = n \text{ or } V[j].\,hash \neq V[j+1].\,hash \\ \textbf{false} & \text{otherwise} \end{cases}$$

   E lists all the equivalence classes of lines in file 2, with $last$ = **true** on the last element of each class. The elements are ordered by $serial$ within classes.

4. Let P be a vector of integers. For $i = 1, \ldots, m$ set

$$P[i] \leftarrow \begin{cases} j \text{ such that } E[j-1].\,last = \textbf{true} \text{ and } H(i) = V[j].\,hash \\ 0 \text{ if no such } j \text{ exists} \end{cases}$$

   where $H(i)$ is the hash value of line i of file 1. The j values can be found by binary search in V.
   $P[i]$, if nonzero, now points in E to the beginning of the class of lines in file 2 equivalent to line i in file 1.

Steps 5 and 6 are the longest common subsequence algorithm proper.

5. Let $candidate(a, b, previous)$ be a reference-valued constructor, where a and b are line numbers in file 1 and file 2 respectively and $previous$ is nil or a reference to a candidate.
   Let $K[0{:}\ \min(m, n) + 1]$ be a vector of references to candidates. Let k be the index of the last usefully filled element of K. Initialize

$$K[0] \leftarrow candidate(0, 0, nil),$$

$$K[1] \leftarrow candidate(m + 1, n + 1, nil),$$

$$k \leftarrow 0.$$

   $K[1]$ is a fence beyond the last usefully filled element.

6. For $i = 1, \ldots, m$, if $P[i] \neq 0$ do $merge(K, k, i, E, P[i])$ to update K and k (see below).

Steps 7 and 8 get a more convenient representation for the longest common subsequence.

7. Let J be a vector of integers. Initialize

$$J[i] \leftarrow 0 \quad i = 0, \ldots, m.$$

8. For each element c of the chain of candidates referred to by $K[k]$ and linked by previous references set

$$J[c.a] \leftarrow c.b.$$

The nonzero elements of J now pick out a longest common subsequence, possibly including spurious 'jackpot' coincidences. The pairings between the two files are given by

$$\{(i, J[i]) \mid J[i] \neq 0\}.$$

The next step weeds out jackpots.

9. For $i = 1, \ldots, m$, if $J[i] \neq 0$ and line i in file 1 is not equal to line $J[i]$ in file 2, set

$$J[i] \leftarrow 0.$$

This step requires one synchronized pass through both files.

A.2 Storage management

To maximize capacity, storage is managed in *diff* per the following notes, which are keyed to the steps in the preceding summary. After each step appear the number of words then in use, except for a small additive constant, assuming that an integer or a pointer occupy one word.

1. Storage for V can be grown as file 2 is read and hashed. The value of n need not be known in advance. [$2n$ words]

3. Though E contains information already in V; it is more compact because the *last* field only takes one bit, and can be packed into the same word with the *serial* field. E can be overlaid on $V.serial$. [$2n$ words]

4. P can be grown as was V in step 1. [$2n + m$ words]
V is dead after this step. Storage can be compacted to contain only the live information, E and P. [$n + m$ words]

5. Candidates can be allocated as needed from the free storage obtained in the previous compaction, and from space grown beyond that if necessary. Because they are chained, candidates do not have to be contiguous.

6 During the ith invocation of merge, the first i elements of P are dead, and at most the first $i + 2$ elements of K are in use, so with suitable equivalencing K can be overlaid on P. [$n + m + 3\times$(number of candidates)]

7. P and K are dead, so J can be overlaid on them. E is dead also. [$m + 3\times$(number of candidates)]

A.3 Summary of merge step

procedure *merge*(K, k, i, E, p)
K is as defined in step 5 above, by reference
k is index of last filled element of K, by reference
i is current index in file 1, by value
E is as defined in Step 3 above, by reference
p is index in E of first element of class of lines in file 2 equivalent to line i of file 1, by value

1. Let r be an integer and c be a reference to a candidate. c will always refer to the last candidate found, which will always be an r-candidate. $K[r]$ will be updated with this reference once the previous value of $K[r]$ is no longer needed. Initialize

$$r \leftarrow 0.$$

$$c \leftarrow K[0].$$

(By handling the equivalence class in reverse order, Szymanski[10] circumvents the need to delay updating $K[r]$, but generates extra 'candidates' that waste space.)

2. Do steps 3 through 6 repeatedly.

   3. Let $j = E[p].\mathit{serial}$.
   Search $K[r\colon k]$ for an element $K[s]$ such that $K[s] \to b < j$ and $K[s+1] \to b > j$. (Note that K is ordered on $K[.] \to b$, so binary search will work.)
   If such an element is found do steps 4 and 5.

      4. If $K[s+1] \to b > j$, simultaneously set

$$K[r] \leftarrow c.$$

$$r \leftarrow s+1.$$

$$c \leftarrow \mathit{candidate}(i, j, K[s]).$$

      5. If $s = k$ do: Simultaneously set

$$K[k+2] \leftarrow K[k+1] \quad \text{(move fence)},$$

$$k \leftarrow k+1.$$

      Break out of step 2's loop.

   6. If $E[p].\mathit{last} =$ **true**, break out of step 2's loop.
   Otherwise set $p \leftarrow p+1$.

7. Set $K[r] \leftarrow c$.