Examining the algorithms of the diff utility

WHAT'S THE DIFF?

urbanheart, Fotolia

Diff finds the differences between two versions of a file. We'll show you how diff finds changes and matches in files without affecting a system's resources. **BY ANDREAS ROMEYKE**

For a user at the command line, discovering the differences between two text files is easy: a simple command, such as *diff Version_1.txt Version_2.txt*, is all it takes. On closer inspection, however, it turns out that diff needs a large amount of memory and some ingenious algorithms to compare files.

This article investigates how diff manages to find changes and matches in multiple megabyte files without affecting a system's resources.

	t	i	e	r
t	0	11	15	2
o	5	6	10	3
r	2	9	13	0

Figure 1: The matrix view makes matches (zero values) visible, even though the position of the characters has changed between the two files.

Editing Distance

Every string can be changed into any other string by inserting, deleting, or replacing individual characters.

One possible method of converting *tier* into *tor* would be to perform the following changes: *tier -> ter -> tr -> tor*. However, an alternative solution with fewer intermediate steps would be: *tier -> ter -> tor*.

The smallest number of steps required for a change provides us with a metric for evaluating the similarity of two strings. This metric is referred to as the Levenshtein or editing distance, and this method is the basis for marking changes in diff.

THE AUTHOR

After graduating in telecommunications and information science, Andreas Romeyke now works as a software developer for the Max Planck Institut for Neuro and Cognition Sciences in Leipzig, Germany. Andreas co-founded the Linux Usergroup Leipzig and the Society for the Application of Open Systems.

	o	t	t	e	r
l	3	8	8	7	6
o	0	5	5	10	3
t	5	0	0	15	2
t	5	0	0	15	2
o	0	5	5	10	3

Figure 2: Matching passages are visible as zero diagonals running parallel to the main diagonals in the matrix.

	l	a	g	e	r
r	6	17	11	13	0
e	7	4	2	0	13
g	5	6	0	2	11
a	11	0	6	4	17
l	0	11	5	7	6

Figure 4: Palindromes (reversed character orders) show up in the matrix as diagonals that run from bottom left to top right.

In practical applications, the larger part of the files will be unchanged for most comparisons. Thus, the first step is to exclude the identical passages. To discover the changes, even if they have been shifted with respect to the original, we need to organize the text in a matrix, as shown in Figure 1.

The numbers in the table refer to the differences between the byte values of the individual characters. Thus, a zero represents an unchanged character. The longest match is referred to as the longest common subsequence or LCS.

The editing distance can be derived from the length of the LCS by applying the following formula $d(X, Y) = n + m - 2 * |LCS(X, Y)|$ with $X = x_1 \ldots x_n$ and $Y = y_1 \ldots y_m$

In a matrix of this kind, shifts are very easy to detect: comparing *otter* and *lotto* (Figure 2) the zeroes (the matches) are located along a descending line parallel to the main diagonal of the matrix (the diagonal that runs from top left to bottom right).

	t	i	e	r
t	0	11	15	2
e	15	4	0	17
i	11	0	4	13
r	2	13	17	0

Figure 3: Swaps show up as interrupted zero diagonals in the matrix. The characters that were swapped are located on a line at 90 degrees to the diagonal.

Swaps (*teir* to *tier*, Figure 3) are shown as interruptions in the matrix with zeroes at 90 degrees to the main diagonal running through their centers.

Palindromes (reversed character orders) show up as a sequence of zero values that runs from top right to bottom left (the adjacent diagonal) in the matrix (Figure 4).

Runtime Optimization

The matrix size depends on the length of the texts. If you have two 10 KB files, the number of comparisons is surprisingly high: 10000 * 10000 = 100000000, and this means you need 100 MB of RAM just to store the matrix. Searching for matches requires some more memory.

A computational process that calculates values multiple times can be optimized. Dynamic programming (see the "Dynamic Programming" box) reduces memory consumption and saves computation time.

Dynamic programming keeps the number of comparisons low when comparing two versions of a text in a matrix: instead of the bitwise difference between two characters, the matrix shown in Figure 5 stores the number of matching characters since the start of the string. Listing 1 provides the Perl code used to implement this approach.

Using the values shown in Figure 5, a backtracking algorithm can quickly determine the longest common subsequence in a string:

1. Start with the maximum value. Select the largest entry above and to the left, or to the left, or above the current position.
2. If multiple entries with equally large values exist, take the path above and to the left.

Listing 1: Searching for the LCS

```
01 sub lcs {
02    my $refmatrix=shift;
03    my $refxlst=shift;
04    my $refylst=shift;
05    my $m=scalar @$refxlst-1;
06    my $n=scalar @$refylst-1;
07    foreach my $i (1 .. $m) {
08       foreach my $j (1 .. $n) {
09          if ($refxlst->[$i] eq
   $refylst->[$j]) {
10
   $refmatrix->[$i]->[$j] =
   $refmatrix->[$i-1]->[$j-1]+1;
11          } elsif
   ($refmatrix->[$i-1]->[$j] >=
   $refmatrix->[$i]->[$j-1]) {
12
   $refmatrix->[$i]->[$j] =
   $refmatrix->[$i-1]->[$j];
13          }
14          else {
15
   $refmatrix->[$i]->[$j] =
   $refmatrix->[$i]->[$j-1];
16          }
17       }
18    }
19    return $refmatrix;
20 }
```

3. Walking through the matrix; the LCS is found if multiple entries with the same maximum value occur.

Figure 6 shows the path that this algorithm takes through the matrix. Listing 2 implements the matching algorithm in Perl. To allow the script to terminate gracefully, the string must contain a sequence of null values at the start, as shown in the figure.

You don't need to add much to the algorithm discussed in the last section to output the differences between two files or strings just like diff. Whenever the tracking path through the matrix changes direction upward or to the left, a character has been deleted or inserted in the new version.

The script in Listing 3 detects these changes. The *while* loop in Lines 43 and 47 makes sure the algorithm takes the characters represented by zeroes in the matrix into consideration.

Although dynamic programming avoids multiple calculations, the developers behind the diff tool for Unix (later known as the diff-utils, [1]) had to pull another card out of their sleeves.

The diff tool was mainly designed for use with source code. To be able to handle typical file sizes with the memory that computers had in the 1980s, diff does not compare letter by letter, but line by line.

To do so, the program first calculates a hash for each line, before calculating the differences between the hashes in a second step.

The program does not need to compare the lines letter by letter if the hashes are identical. This approach saves a great deal of memory.

In 1986, Eugene Myers developed a fast algorithm that is the basis of the popular diff-utils [6]. GUI-based alternatives to the diff command line program, such as Meld [7] or the KDE Kompare [8] tool, are all based on the approach. In fact, despite the fancy graphics, Kompare actually relies on the legacy diff tool under the hood.

Dynamic Programming

Dynamic programming is an important concept in computer science. and it is also often the best approach for resolving optimization problems. In many cases, it is easier to break a problem down, resolve the individual subtasks, and use the results in an additional processing step.

Calculating powers is a simple example that dates back to the days in which computational resources were scarce: to calculate the eighth power of a number, you can break down the calculation *n*n*n*n * n*n*n*n* into intermediate steps of *((n*n) * (n*n)) * ((n*n) * (n*n))*. If you temporarily store the results of *(n*n)* and *((n*n)*(n*n))*, three multiplications are required, rather than seven.

More Applications

The technique that diff uses is not only suitable for discovering differences in the source code. Instead of discovering differences, the diff algorithm can also find matches, and thereby prove that code has been reused. For larger scale software projects, the occurrence of

Listing 2: Backtracking

```
01 # run backtracking on lcs-matrix
02 sub backtracking_lcs {
03    my $refmatrix=shift;
04    my $ref_xlst=shift;
05    my $ref_ylst=shift;
06    my @lcs;
07    my $x=scalar @$ref_xlst -1;
08    my $y=scalar @$ref_ylst -1;
09    while ($y>0 && $x>0) {
10       my $actual_value=$refmatrix->[$x]->[$y];
11       my $actual_x=$x;
12       if (
13
($refmatrix->[$x-1]->[$y-1] >= $refmatrix->[$x-1]->[$y]) &&
14
($refmatrix->[$x-1]->[$y-1] >= $refmatrix->[$x]->[$y-1])
15       ) { # go left upper
16          $x--; $y--;
17       } elsif ($refmatrix->[$x-1]->[$y] >= $refmatrix->[$x]->[$y-1]) { # go left
18          $x--;
19       } else { # go upper
20          $y--;
21       }
22       # check if value is changed, then push to @lcs
23       if ($actual_value > $refmatrix->[$x]->[$y]) {
24          push @lcs, $actual_x;
25       }
26    }
27    @lcs=reverse @lcs; # reverse because backtracking
28    return \@lcs;
29 }
30
31 # print out lcs matrix
32 sub print_lcs {
33    my $ref_matrix=shift;
34    my $ref_xlst=shift;
35    my $ref_ylst=shift;
36    print "LCS: '";
37    foreach my $i (@{ backtracking_lcs($ref_matrix, $ref_xlst, $ref_ylst) }) {
38       print $ref_xlst->[$i];
39    }
40    print "'\n";
41 }
```

INFO

[1] GNU Diffutils Manual, 2002: *http://www.gnu.org/software/diffutils/manual/diff.html*

[2] Darren C. Atkinson and William G. Griswold, "Effective pattern matching of source code using abstract syntax patterns": Software – Practice and Experience, 36 (4), p. 413-447, 2006.

[3] K. Nandan Babu and Sanjeev Saxena, "Parallel algorithms for the longest common subsequence problem", January 20, 1999.

[4] J. W. Hunt and M. D. McIlroy "An algorithm for differential file comparison": Technical Report CSTR 41, Bell Laboratories, Murray Hill, NJ, 1976.

[5] Moritz G. Maaß, "Matching statistics: efficient computation and a new practical algorithm for the multiple common substring problem": Software – Practice and Experience, 36 (3), p. 305-331, March 2006.

[6] E. W. Myers, "An o(ND) difference algorithm and its variations": Algorithmica, 1 (2), p. 251-266, 1986; *http://citeseer.ist.psu.edu/myers86ond.html*

[7] Meld: *http://meld.sourceforge.net*

[8] Kompare: *http://www.caffeinated.me.uk/kompare*

		t	i	e	r
	0	0	0	0	0
t	0	1	1	1	1
e	0	1	1	2	2
e	0	1	1	2	2
r	0	1	1	2	3

Figure 5: Instead of entering the differences between the character values, it is more efficient to write the length of the subsequences on initial parsing.

		t	i	e	r
	0	0	0	0	0
t	0	1	1	1	1
e	0	1	1	2	2
e	0	1	1	2	2
r	0	1	1	2	3

Figure 6: To discover the longest subsequence, start at the maximum value in the table and backtrack through the fields, using a simple algorithm.

many code duplicates is proof of successful refactoring. A variant on the diff theme is even able to compare notes played with the notes a musician is asked to play.

If the distance matrix (Figure 4) shows the difference between the keypresses on a computer keyboard (this referred to as the typewrite distance). Instead of the difference between the character codes, it can be applied to incorrectly typed words to guess what a person meant to type. One interesting application for diff is in biology, where it is used to sequence and catalog genes. ■

Listing 3: Diff Algorithm

```
01 # run backtracking on lcs-matrix
02 sub backtracking_lcs {
03    my $refmatrix=shift;
04    my $ref_xlst=shift;
05    my $ref_ylst=shift;
06    my @lcs;
07    my $x=scalar @$ref_xlst -1;
08    my $y=scalar @$ref_ylst -1;
09    while ($y>0 && $x>0) {
10       my $actual_value=$refmatrix->[$x]->[$y];
11       my $actual_x=$x;
12       my $actual_y=$y;
13       my $actual_direction;
14       if (
15 ($refmatrix->[$x-1]->[$y-1] >= $refmatrix->[$x-1]->[$y]) &&
16 ($refmatrix->[$x-1]->[$y-1] >= $refmatrix->[$x]->[$y-1])
17       ) { # go left upper
18          $x--; $y--;
19          $actual_direction="ul";
20
21       } elsif ($refmatrix->[$x-1]->[$y] >= $refmatrix->[$x]->[$y-1]) { # go left
22          $x--;
23          $actual_direction="l";
24       } else { # go upper
25          $y--;
26          $actual_direction="u";
27       }
28       # check if value is changed, then push to @lcs
29       if ($actual_value > $refmatrix->[$x]->[$y]) {
30          # push @lcs, $actual_x;
31          push @lcs, "(".$ref_xlst->[$actual_x].")";
32       } else {
33          if ($actual_direction eq "u") {
34             push @lcs, "+(".$ref_ylst->[$actual_y].")";
35          } elsif ($actual_direction eq "l") {
36             push @lcs, "-(".$ref_xlst->[$actual_x].")";
37          } else {
38             push @lcs, "+(".$ref_ylst->[$actual_y].")";
39             push @lcs, "-(".$ref_xlst->[$actual_x].")";
40          }
41       }
42    }
43    while ($y > 0) { # get last stuff of ylst
44       push @lcs, "+(".$ref_ylst->[$y].")";
45       $y--;
46    }
47    while ($x > 0) { # get last stuff of xlst
48       push @lcs, "-(".$ref_xlst->[$x].")";
49       $x--;
50    }
51    @lcs=reverse @lcs; # reverse because backtracking
52    return \@lcs;
53 }
54
55 # print out lcs matrix
56 sub print_diff {
57    my $ref_matrix=shift;
58    my $ref_xlst=shift;
59    my $ref_ylst=shift;
60    print "DIFF: '";
61    foreach my $i (@{ backtracking_lcs($ref_matrix, $ref_xlst, $ref_ylst) }) {
62       print $i;
63    }
64    print "'\n";
65 }
```