Adventures with DIFF

Barry Dwyer

Abstract

`diff` is a Unix/Linux utility supported by the Free Software Foundation. It enables line by line comparison of two files. I shall discuss four things:

1. The proofreading task that spurred me to write this,
2. Using `diff` to compare student assignments, and how it led to a friend making a lot of money,
3. Using `diff` to compare directories and possibly save me from disaster.
4. How `diff` works, and *Dynamic Programming*.

`diff`

The simplest use of `diff` is as follows:

```
diff file1 file2
```

If `file1` and `file2` are text files, `diff` will compare them line by line and report where they differ. If they are binary files, `diff` will simply say whether they are the same. If they are directories, `diff` will report where differ, and it can be asked to search their subdirectories recursively.

`diff` has many options, for example, to ignore the difference between upper and lower case, to ignore white space, etc. If you want the full story, type

```
man diff | less
```

1 Fun with LaTeX.

As I'm sure I have made you aware, I have published a textbook, which contains lots of diagrams, tables and mathematical formulae. I submitted the manuscript to the publisher as series of `.tex` files, and they then typeset it. To illustrate how LaTeX works, I made the above section heading by typing,

```
\section{Fun with {\LaTeX}.}
```

and LaTeX set it in its default style. But publishers like to use their own house styles: different fonts, spacing, page geometry, etc. LaTeX makes this possible by letting the typesetter include a 'style' file at the start of the document—a bit like a cascading style sheet in `HTML`. The publisher then returned the result for me to proofread as a `PDF`. Where they had corrected something, but were not

sure they had done it right, they attached a note to the PDF. Unfortunately, they confidently 'corrected' a lot of other things they *didn't* tell me about. Here's an example of one of their corrections,

$$Has\ Parent = \{Peter \mapsto Mary, Peter \mapsto Mark,\ Mary \mapsto Jane, Mary \mapsto Paul\}$$

and here is my original,

$$Has\ Parent = \{Peter \mapsto Mary, Peter \mapsto Mark, Mary \mapsto Jane, Mary \mapsto Paul\}$$

Can you spot the difference? I was supposed to eyeball 500 pages of this stuff looking for differences like that. I decided that it was just too hard, so I asked them to send me *their* `.tex` files. When they *finally* agreed to do this, I ran `diff` to tell me what they had tampered with—which turned out to be quite a lot. Here is one of the things it told me,

```
997c997
< {\var{Peter}}\mapsto {\var{Mark}},\var{Mary}\mapsto \var{Jane},
---
> {\var{Peter}}\mapsto {\var{Mark}},\ \var{Mary}\mapsto \var{Jane},
```

which told me that at line 997 of each file there is a change (`c`): the typesetters had effectively replaced my version of the line by theirs. They had forced an extra space to appear between *Mark* and *Mary*. I wouldn't have minded, but, to be consistent, they should either have put extra spaces between *Mary* and *Peter*, and *Jane* and *Mary*, or simply have left well enough alone. Thank you, `diff`, for saving me from going word-blind!

2 Catching Cheats

One of the joys of teaching computer science was setting and marking practical work, which usually consisted of writing a program or two. Give a student twelve weeks to complete an assignment, and they will begin in week eleven. So we usually set assignments that could be completed in stages, and set deadlines every couple of weeks or so.

Practical work has to be worth a few marks, or no-one will do it; too many marks, and the poorer students will cheat. Catching the cheats is what marking is all about.

The first tool we used was `diff`. If two assignments showed only few differences, we could be pretty sure one student 'helped' the other, either wittingly or unwittingly.

- The first problem is that `diff` compares *lines*, so changing a few line breaks will cause `diff` to find more differences.
- The second problem is that the order in which parts of a program are defined is usually fairly flexible, so shuffling the code can defeat `diff` too.
- The third problem is, with a class of 100 students, there are 4,950 pairs of assignments to compare.

Our first solution to the first problem was to pass the programs through a filter that ignored the students' white space and line breaks. Instead it inserted

line breaks after semicolons, which mark the ends of logical statements in many programming languages. Unfortunately, there are other languages that use full-stops rather than semicolons, and yet others that actually use line breaks. In the end, we broke lines in an arbitrary, language-independent, way: If a group of four characters had ASCII codes that hashed to some magic value, we inserted a line break. This divided the file into what are now called *blocklets*.

To solve the second problem, we simply sorted the lines before passing them to `diff`.

To solve the third, we *hashed* each line. In case you don't know, a hash value is a number obtained from the text by some mystic calculation. If two lines have the same hash value, they are probably, but not necessarily, the same; if they have different values, they are definitely different.

One way to compute a hash value is to regard the binary representation of the text as a huge number, divide it by a suitably large prime, then take the remainder. The result must lie between zero and one less than the prime. The bigger you make the prime, the less the chance two different lines will have the same hash. This doesn't solve the problem of comparing 4,950 pairs of assignments, but it is much quicker to compare numbers than texts.

In the end, we found that the best way to discover cheats was to go back to `diff`, pure and simple. We just compared each stage of a student's work with the previous one. Slow and steady progress, no worries; a sudden surge in `diff` output, highly suspicious!

However, all those other ideas eventually made one of my friends a few million dollars—which is really the point of this whole story. He was completing a PhD thesis on data compression, and could see how these thoughts could be translated into a useful product. If two texts hash to the same value, they are probably the same. If we choose a big enough prime number, the probability becomes a virtual certainty. Combine that with a trapdoor algorithm, and the result is a *digital signature*. One of the properties of a digital signature is that it is very hard to forge: it is practically impossible to construct a different text that gives the same signature. A signature can be used to verify, for example, that a money transfer has not been tampered with.

Now combine signatures with blocklets. If we want to back up a file over the Internet for the first time, we have to transmit *all* its blocklets. But if we make updates, we only need to send the server the blocklets that have changed. What we do is to send their *signatures*. The back-up server checks if it already has blocklets with all those signatures, and tells us which ones it doesn't have. We only need to send the server the missing ones.

The outcome was that a big US remote back-up service provider paid my friend megabucks for his patent rights. It meant the provider could give their customers faster service, while needing less storage *and* paying less for data transmission.

3 A Scary Time with Directories

Like most of us, I use one account for general use ('`barry`'), and another, '`admin`' for system maintenance. Poking around in `admin` on my *laptop*, I was a bit surprised to see a '`Documents`' directory with apparently the same content as the one in `barry`. I thought maybe I should delete all the files in `admin`, which

were unwanted duplicates. To see if they were indeed the same, I typed,

```
cd /
sudo diff -r /Users/admin/Documents/ Users/barry/Documents/
```

My idea was to check for any differences between the two directories, recursively (`-r`). The amount of output was a overwhelming: `diff` displayed hundreds of files called '`.DS_Store`' in `barry`, but not in `admin`. It turned out that `.DS_Store` files are ones that *MacOS* uses to remember the positions of icons, so they were pretty much irrelevant. If there were any *relevant* differences, I couldn't find them. So I typed,

```
diff -r -x '.DS_Store' /Users/admin/Documents/ Users/barry/Documents/
```

which gave a much more sensible result, showing only a dozen or so differences. (The '`-x`' option allowed me to exclude all files matching `.DS_Store`.)

Where had the `Documents` files come from? At first I thought it had something to do with the magic of *iCloud*. I use the same *Apple ID* to download software to `admin` as I use to download music files to `barry`. Did the common *Apple ID* mean my laptop was uploading documents from `barry` to *iCloud*, then downloading them from *iCloud* to `admin`? If that was true, deleting the `admin` copy would delete the *iCloud* copy and delete the `barry` files too. Scary!

The fact that were some differences between the directories was encouraging, and so was the fact that my `admin/Documents` directory on my *desktop* computer was virtually empty.

So where did the `admin` documents come from? I eventually recalled that when I first bought the new laptop I chose an option to migrate everything across from my old laptop. *MacOS* unwisely decided that since `barry` was the proud owner of a new computer, `barry` would need `admin` privileges. It wouldn't let me then create a *second* `admin` account, so I had to rename `barry` to `admin`, then create an ordinary user called `barry`. My `admin` account now contained my migrated documents but my `barry` account had none. But once I had signed in to *iCloud*, *Apple* magic took over and my laptop's `Documents` directory had been synchronised with the desktop's. Panic over, and I could confidently delete the `admin` documents!

So much for logic! So much for confidence! It didn't happen straightaway. It happened overnight! The next day all my documents had disappeared from both accounts on both computers. And *iCloud*. And my *iPhone*. And my *iPad*.

Thank goodness for automatic back-ups!

4 How `diff` Works

If `file1` contains lines *A, B, C, D, E*, and `file2` contains lines *B, C, D, E, A*, then I want `diff` to tell me that `file2` has *A* missing before *B, C, D, E*, and *A* added after them. I don't want `diff` to say that `file2` has *B, C, D, E* added before *A*, and *B, C, D, E* missing after it. In other words, I want `diff` to find the fewest number of changes that can make the files agree. Obviously `diff` can't do this by comparing one line at a time. It needs to look ahead. The number of lines it looks ahead is called its *horizon*, which can be controlled by a command option. Even so, the problem of finding the fewest changes is not trivial.

Table 1 shows how the optimum choices are made. Each cell of the table represents a possible state of matching. For example, the cell in row **B**, column **D** represents a state where lines *A, B, C,* and *D* have been read from `file1` but only line *B* has been read from `file2`. The number in each cell is the minimum penalty to reach that cell, where we assign a penalty of 1 for each insertion or deletion, but zero penalty when lines match. We regard a changed line as an insertion plus a deletion.

	–	**A**	**B**	**C**	**D**	**E**
–	**0**	**1**	2	3	4	5
B	1	2	**1**	2	3	4
C	2	3	2	**1**	2	3
D	3	4	3	2	**1**	2
E	4	5	4	3	2	**1**
A	5	6	5	4	3	**2**

Table 1: Possible optimal states of matching two files. Moving left to right means accepting a line from `file1` without accepting a line from `file2`, and incurs a penalty of 1. Moving top to bottom means accepting a line from `file2` without accepting a line from `file1`, and also incurs a penalty of 1. Moving diagonally top-left to bottom-right means accepting a line from both files. It incurs zero penalty if the lines match, otherwise it incurs a penalty of 2. The optimal sequence of moves is highlighted in bold type.

The best way to get to row **B**, column **D**, for example, is to read line *A* from `file1`, match *B* from both files, then read *C* and *D* from `file1`. This incurs a penalty of 3. If we had chosen to read *A, B, C* and *D* from `file1`, then to read *B* from `file2`, that would have incurred a penalty of 5. (The state represented by the cell in row **B**, column **D** is *not* part of the optimum solution!)

The number of cells in the table is proportional to the product of the lengths of the files if they are short enough, otherwise it is proportional to the square of the horizon. Clearly, the size of the table is also a measure of how long it takes to find the best solution. With a horizon of 100 lines, we need to calculate 10,000 cells. However, we do not need to store the entire table. It can be developed row by row: we only need to store the current row and the next row.

Although this might seem tedious, it is better than the obvious approach. At each step we can either read `file1`, read `file2`, or read both. We must therefore make a 3-way decision at least 100 times. That gives over 3^{100} (about 5×10^{47}) alternatives. Quite impractical!

The way `diff` matches files is exactly analogous to how a spell checker counts spelling mistakes, and how *Auto-Correct* guesses what you are trying to type. They are both examples of *dynamic programming problems.*

This use of the word 'programming' has nothing to do with computer programs. It means organising a sequence of actions. The dynamic programming principle was originally devised in connection with NASA's attempts to land a man on the moon.

The dynamic programming principle is this:

> The best way to get from *A* to *B* via *C* consists of the best way to get from *A* to *C*, followed by the best way to get from *C* to *B*.

That seems so obvious that it can't be useful, especially since it doesn't tell us how to choose C. What it *does* tell us is that we only need to remember one way of getting from A to C—the best one. Then we can take each possible C in turn and figure out how to get to B.

Have you ever wondered why the output from LaTeX looks so much more professional than the output from a word processor?

A word processor justifies one line at a time. When a line overflows, it must decide whether to shrink the spacing to get the current word to fit, expand the spacing to let it overflow, or choose some place to hyphenate it. The result is that the line is split in absolutely the best place, given what came before. The problem is that splitting line 1 in the best place may leave line 2 with only some really bad choices. Does this problem begin to sound familiar?

LaTeX, on the other hand, sets a whole paragraph at a time—using dynamic programming. Look at line 3 of the previous paragraph. It is a little widely spaced. LaTeX probably debated whether to squeeze 'that' onto line 3, but decided not to, because line 4 would not be able to squeeze in the word 'problem' without hyphenating it or narrowing the spacing too much. It chose the break that incurred the least penalty for the whole paragraph.

LaTeX extends this policy to laying out a whole page, often expanding or shrinking the spacing between paragraphs a little to avoid widows and orphans.