

An introduction to diffs

Finding the similarities (and differences) between two strings

In this issue...

▶ WHAT'S COVERED

A common programming task is using a tool to view the differences between two text files, the files usually being different versions of the same source code file. We're interested in which lines were removed and which were added. The basic algorithm to determine these 'edits' is known as the longest common subsequence (LCS) algorithm, and it's easier to grasp if you consider the difference between two strings.

We've all spent time playing those children's word puzzles where you change one word into another by altering a single letter at a time, making all the intermediary steps recognisable words as well. To take a simple example, changing CAT into DOG would require the following steps: CAT, COT, COG, DOG.

These word games work by merely deleting a letter and inserting a new one at the same position at each step. If we didn't have the limitations imposed by the rules of the puzzle, we could certainly transform any word into another by deleting all the old characters and inserting all the new ones. That's the sledgehammer approach, but we'd like to be a little subtler.

Suppose our goal were to find the smallest number of edits needed to convert one word to another. Let's think about changing LAMB into RAM. Looking at the words, it's easy to see that we must delete the B and replace the L with an R. So how do we implement this as an algorithm?

Common subsequences

One way is to look at the subsequences of each word and see if we can find a common subsequence between the two words. A subsequence is formed by removing characters from the string and leaving the remaining characters in their original order. For example, the three-letter subsequences of LAMB are AMB, LMB, LAB and LAM. The two-letter subsequences are LA, LM, LB, AM, AB and MB, and there are four single-letter ones. So for

		R	A	M	to get RAM from LAMB:
L	0	0	0	0	"add R"
A	0	0	1	1	"keep A"
M	0	0	1	2	"keep M"
B	0	0	1	2	"remove B"

▲ Figure 1: The LCS matrix to go from LAMB to RAM.

a four letter word there are a total of 15 possible subsequences, and, in fact, it can be shown that for an n letter sequence the number of subsequences is about 2^n .

The brute-force algorithm would look at the two words LAMB and RAM and enumerate their three-letter subsequences to see if any match. There isn't, so next we do the same for the two-letter subsequences of each

word. This time we have a match: AM. This is the longest common subsequence (LCS) between the two words. From this starting point, we can work out which letters to delete and which to insert.

For small words, like our example, this process isn't too complex, but imagine if we were looking at enumerating all of the subsequences of a 100-character string. As I stated, the number of possible subsequences is about 2,100: a huge amount. The brute-force algorithm is exponential; it's not feasible to use.

Stepwise progression

The subsequence idea does have its merits though; we just need to approach it from a different angle. Instead of enumerating all of the subsequences in the two words and comparing, let's see if we can do it in a stepwise progression. We'll use a dynamic programming method to split up the calculation

Nugget

One technique for discovering the diff between two XML files is to use not characters or lines but elements as the fundamental thing we compare. The reason is that in XML files, whitespace is frequently ignored as not being part of the data or structure of the XML file. Hence we have to use the structure of the XML itself as the basis of our comparisons, and not the fact that in one file an element is spread over several lines and in another the same element isn't. ■

Nugget

The LCS algorithm works in the same way when you compare two text files. In this case, of course, we would be comparing lines in the text files and not individual characters. It goes without saying that the text files are probably going to have hundreds of lines rather than the few characters we've been discussing. However the process is the same: compare the end line from the two files, if equal you have a diagonal and you reduce both sets, if not, you find the maximum of the two possible LCSs by ignoring the final lines. Repeat through recursion. ■

of the LCS for two longer words into calculating LCSs for smaller words and then somehow 'joining' them together.

Imagine we are trying to find the LCS for two different words. First, we compare the final letter of both words. If they're the same, we can immediately say that this letter is part of the LCS (the final part). To calculate the rest of the LCS, we can delete the final letter from both words and then calculate the LCS of these two smaller words – a smaller problem. We can then add this final letter to the solution of this LCS calculation to produce the complete LCS for the two words.

To use some simple notation: $LCS("fooX", "barX") = LCS("foo", "bar") + "X"$.

However, what happens if the final letters don't match? In this case, we shall have two problems to solve instead of one. We'll have to calculate the LCS of the first word minus its final character and the second word, and then calculate the LCS of the first word and the second word minus its final character. Once we have these two values for the LCS of these two slightly smaller problems, we have to choose the larger LCS as the one we want; after all, we're aiming for the longest common subsequence.

In our simple notation: $LCS("fooX", "barY") = \max(LCS("foo", "barY"), LCS("fooX", "bar"))$.

Minimising the calculations

In both cases (either the final letter is the same or it isn't), we're reducing the problem of finding the LCS of two long words into

one or two LCS calculations of shorter words. By continuing this process of reducing the problem step by step, we'll eventually be comparing single characters.

However, in doing so we may find ourselves calculating the same LCS values over and over, so we should store the results somehow, allowing us to re-use them. This is one of the hallmarks of a dynamic programming problem: not only do we reduce the problem to smaller ones and combine the results of solving the smaller problems, but we also store the results of the smaller problems for re-use.

Since we may compare every substring of the first word with every substring of the second, it makes sense to use a matrix. The rows would be all the starting substrings of the first word, and the columns would be that of the second. We could store the LCS of the row substring and the column substring in each cell, but this isn't too helpful: it helps us calculate the LCS, yes, but it doesn't help us in determining which characters need to be deleted or inserted in order to generate the string we want from the original.

One item of information we really need is the length of the LCS for every cell. Once we have this information, we can easily work out the length of the LCS for the two complete words using the recursive algorithm. To be able to generate the LCS string itself, though, we'd need to know which path we took through the matrix. To do this, we'd need to store a

Spotlight on...

Iteration and recursion

The exercise we did to calculate the LCS matrix of LAMB and RAM was an iterative method: calculating the value in every cell from the cells we'd just calculated, until the whole matrix was filled in.

What happens in the recursive case? This is, after all, where we started when discussing the algorithm. Here we would essentially work our way backwards through the

matrix. Would we also calculate all the cells in the matrix?

Generally we would, but in the case where there are very few changes between the two words (say, between MALE and FEMALE) a lot of the cells would not be calculated at all due to the great similarity between the sequences. In 'Figure 2', I've left the cells that don't need calculating blank. That wouldn't be calculated with a recursive approach. ■

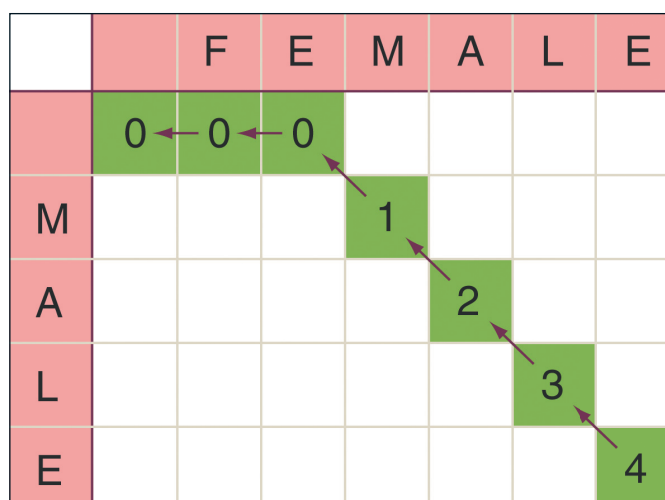
pointer at each cell that pointed to the previous cell that was used to build the LCS for this one.

Using the matrix

Let's calculate the LCS matrix by hand for the LAMB/RAM case. We'll have a 5x4 matrix (we take into account empty substrings, so we should start indexing at 0). Rather than fill in the matrix recursively, we'll calculate all the cells iteratively from the top left all the way down to the bottom right, going from left to right along each row. The first row and column are easy: all zeroes, because the LCS between an empty string and any other word has zero length. From this we can start working out the LCS for cell (1,1), or the two strings L and R. The two final characters of these one-character strings are not equal, so the length of the LCS is the maximum of the previous cells above and to the right. These are

both zero, so their maximum value and hence the value of this cell is zero. Cell (1,2) is for the strings L and RA. Again, zero. Cell (2,1) is for LA and R: the LCS length is zero again. Cell (2,2) is interesting: it's the LCS length for the two substrings LA and RA. Since the final letters are the same, we add one to the LCS length for the two substrings without the A, which is 0. Continuing like this, we're able to fill in all 20 cells in the matrix.

'Figure 1' doesn't just show the answer. It also reveals some more information: for each cell, it indicates the previous cell – whose length value was used in calculation – with a small arrow. We can follow the arrows back from the bottom-right corner (which has the length of the complete LCS) to calculate the LCS itself: every time we move on a diagonal, it indicates an equal letter, and hence is part of the LCS. In fact, we can infer even more about this path: if we move horizontally it indicates a letter being inserted, vertically means a letter is removed. Not only can we calculate the LCS; we can also calculate the edits needed to convert one word to another. We've worked out the diff. ■



▲ Figure 2: The same binary tree only showing the links.

Write an
XNA game

Use your coding skills
to create your own
Xbox game,
page 121

Julian M Bucknall has worked for companies ranging from TurboPower to Microsoft and is now CTO for Developer Express. feedback@pcplus.co.uk