

1 Dynamic programming calculation of similarity

We now turn to the algorithmic question of how to compute, via dynamic programming, the similarity of two strings along with the accompanying optimal alignment.

NOTE: This treatment of dynamic programming is not actually what I did first in clas. Rather in lecture I developed a graphical way to think about this. In monday's lecture (April 15), I will introduce the formalisms that are presented in these notes.

Definition For two strings S_1 and S_2 , $V(i, j)$ is defined to be the similarity of prefixes $S_1[1..i]$ and $S_2[1..j]$, ie. the similarity of the prefix of S_1 up to the i 'th character, and the prefix of S_2 up to the j 'th character.

That is, $V(i, j)$ denotes the value of the optimal alignment of the *first* i characters of S_1 and the *first* j characters of S_2 . Using this notation, if S_1 has n letters and S_2 has m letters, then the similarity of S_1 and S_2 is *precisely* the value $V(n, m)$.

We will compute $V(n, m)$ by solving the more general problem of computing $V(i, j)$ for *all* combinations of i and j , where i ranges from zero to n and j ranges from zero to m . This is the standard *dynamic programming* approach used in a vast number of computational problems. The dynamic programming approach has three essential pieces – the *recurrence relation*, the *tabular computation* and the *traceback*. We will explain each piece in turn.

1.1 The recurrence relation

The recurrence relation establishes a *recursive* relationship between the value of $V(i, j)$, for i and j both positive, and values of D with index pairs *smaller* than i, j . When there are no smaller indices, the value of $V(i, j)$ must be stated explicitly in what are called the *base conditions* for $V(i, j)$.

For the string similarity problem, the base conditions are

$$V(i, 0) = -i,$$

and

$$V(0, j) = -j.$$

The base condition $V(i, 0) = -i$ is clearly correct (that is, it gives the number required by the definition of $V(i, 0)$) because the only possible alignment of the first i characters of S_1 and the first 0 characters of S_2 , puts those i characters against i spaces. Similar reasoning holds for $V(0, j) = -j$.

The recurrence relation for $V(i, j)$ when both i and j are strictly positive is

$$V(i, j) = \max[V(i-1, j) - 1, V(i, j-1) - 1, V(i-1, j-1) + t(i, j)],$$

where $t(i, j)$ is defined to have value 1 if $S_1(i) = S_2(j)$, and $t(i, j)$ has value -1 if $S_1(i) \neq S_2(j)$.

Correctness of the general recurrence

We establish correctness in the next two lemmas.

Lemma 1.1 *The value of $V(i, j)$ must be either $V(i, j - 1) - 1$ or $V(i - 1, j) - 1$ or $V(i - 1, j - 1) + t(i, j)$. There are no other possibilities.*

Proof Consider an optimal alignment of $S_1[1..i]$ and $S_2[1..j]$, i.e. an alignment of those two prefixes that maximizes the number of matches minus the number of mismatches minus the number of spaces, over all possible alignments of those two prefixes. Hence that alignment has value $V(i, j)$.

Now focus on the last column of that alignment. In that last column, either we have the characters $S_1(i)$ opposite the character $S_2(j)$, or we have character $S_1(i)$ opposite a space, or we have character $S_2(j)$ opposite a space. There are no other possibilities. We will examine each possibility in turn.

In the first case, when $S_1(i)$ is opposite $S_2(j)$ in the optimal alignment, the last column contributes value $t(i, j)$ to the value of the alignment, and the symbols in the alignment before the last column specify an alignment of prefix $S_1[1..i]$ to prefix $S_2[1..j]$. Moreover, those prior characters are aligned with value $V(i - 1, j - 1)$, for if they weren't then there is an alignment of $S_1[1..i]$ and $S_2[1..j]$ that is better than the one we assumed is optimal (just create the optimal alignment of $S_1[1..i - 1]$ with $S_2[1..j - 1]$ and then put $S_1(i)$ and $S_2(j)$ opposite each other). Therefore, in the first case, the value of the optimal alignment is $V(i - 1, j - 1) + t(i, j)$.

In the second case, when character $S_1(i)$ is opposite a space in the optimal alignment, the last column contributes -1 to the value of the alignment, and the symbols in the alignment before the last column specify an alignment of $S_1[1..i - 1]$ and $S_2[1..j]$. Again, that alignment must be optimal for those two strings, and hence must have value $V(i - 1, j)$. Therefore in the second case, the value of the optimal alignment is $V(i - 1, j) - 1$.

In the third case, when character $S_2(j)$ is opposite a space in the optimal alignment, the last column again contributes -1 to the value of the optimal alignment, and the prior symbols specify an alignment of value $V(i, j - 1)$. It again follows that the value of the optimal alignment, in this case, is $V(i, j - 1) - 1$.

We have covered all the cases, and hence the Lemma is proved. \square

Now we look at the other side.

Lemma 1.2 $V(i, j) \geq \max[V(i - 1, j) - 1, V(i, j - 1) - 1, V(i - 1, j - 1) + t(i, j)]$.

Proof The reasoning is very similar to that used in the previous lemma, but it achieves a somewhat different goal. The objective here is to demonstrate *constructively* the existence of an alignment achieving each of the three values specified in the inequality. Then since all three values are feasible, the maximum is certainly feasible.

First, it *is* possible to align $S_1[1..i]$ and $S_2[1..j]$ with a value exactly $V(i, j - 1) - 1$. Simply align $S_1[1..i]$ to $S_2[1..j - 1]$ in the optimal way, and then place character $S_2(j)$ opposite a space. By definition, the value of that alignment is exactly $V(i, j - 1) - 1$. Second, it *is* possible to align $S_1[1..i]$ to $S_2[1..j]$ with value exactly $V(i - 1, j) - 1$: Align $S_1[1..i - 1]$ to $S_2[1..j]$ with value $V(i - 1, j)$ and then place character $S_1(i)$ opposite a space. The value of this alignment is exactly $V(i - 1, j) - 1$. Third, it *is* possible to do the align $S_1[1..i]$ with $S_2[1..j]$ with value exactly $V(i - 1, j - 1) + t(i, j)$, using a similar argument. \square

Lemmas 1.1 and 1.2 immediately imply the next theorem.

Theorem 1.1 *When both i and j are strictly positive, $V(i, j) = \max[V(i - 1, j) - 1, V(i, j - 1) - 1, V(i - 1, j - 1) + t(i, j)]$.*

Proof Lemma 1.1 says that $V(i, j)$ must be *equal* to one of the three values $V(i - 1, j) - 1$, $V(i, j - 1) - 1$, or $V(i - 1, j - 1) + t(i, j)$. Lemma 1.2 says that $V(i, j)$ must be *greater than or equal* to the largest of those three values. It follows that $V(i, j)$ must therefore be *equal* to the largest of those three values, and therefore we have proven the theorem. \square

This completes the first piece of the dynamic programming method for edit distance, the recurrence relation, and its proof of correctness.

1.2 Tabular computation of edit distance

The second essential piece of any dynamic program is to use the recurrence relations to *efficiently* compute the value $V(n, m)$.

The recurrence relations we use are:

$$V(i, j) = \text{MAX}[V(i - 1, j) - 1, V(i, j - 1) - 1, V(i - 1, j - 1) + t(i, j)],$$

where $t(i, j)$ is -1 if characters $S_1(i)$ and $S_2(j)$ do not agree, and is +1 if they do agree.

Bottom-up computation

In the bottom-up approach, we first compute $V(i, j)$ for the *smallest* possible values for i and j , and then compute values of $V(i, j)$ for *increasing* values of i and j . The typical way to organize this bottom-up computation is with a dynamic programming *table* of size $(n + 1) \times (m + 1)$. The table holds the values of $V(i, j)$ for all the choices of i and j (see Figure 1). Note that string S_1 corresponds to the vertical axis of the table, while string S_2 corresponds to the horizontal axis. Because the ranges of i and j begin at zero, the table has a zero row and a zero column. The values in row zero and column zero are filled in directly from the base conditions for $V(i, j)$. After that,

V(i,j)			w	r	i	t	e	r	s
		0	1	2	3	4	5	6	7
	0	0	-1	-2	-3	-4	-5	-6	-7
v	1	-1							
i	2	-2							
n	3	-3							
t	4	-4							
n	5	-5							
e	6	-6							
r	7	-7							

Figure 1: Table to be used to compute the similarity of *vintner and writers*. The values in row zero and column zero are already included. They are given directly by the base conditions.

the remaining $n \times m$ subtable is filled in one row at a time, in order of increasing i . Within each row, the cells are filled in order of increasing j .

To get the idea of how to fill in the subtable, note that by the general recurrence relation for $V(i, j)$, all the values needed for the computation of $V(1, 1)$ are known once $V(0, 0)$, $V(1, 0)$ and $V(0, 1)$ have been computed. Hence $V(1, 1)$ can be computed after the zero row and zero column have been filled in. Then, again by the recurrence relations, after $V(1, 1)$ has been computed, all the values needed for the computation of $V(1, 2)$ are known. Following this idea, we see that the values for row one can be computed in order of increasing index j . After that, all the values needed to compute the values in row two are known and that row can be filled in, in order of increasing j . By extension, the entire table can be filled in one row at a time, in order of increasing i , and in each row the values can be computed in order of increasing j (see Figure 2).

Time analysis

How much work is done by this approach? When computing the value for a specific cell (i, j) , only cells $(i - 1, j - 1)$, $(i, j - 1)$ and $(i - 1, j)$ are examined, along with the two characters $S_1(i)$ and $S_2(j)$. Hence, to fill in one cell takes a *constant* number of cell examinations, arithmetic operations and comparisons. Maybe 10 or simple operations per cell are needed. There are $(n + 1) \times (m + 1) = \Theta(nm)$ cells in the table, so

Theorem 1.2 *The dynamic programming table for computing the similarity between a string of length n and a string of length m can be filled in with $\Theta(nm)$ simple operations. Hence, using dynamic programming, the similarity value $V(n, m)$ can be computed efficiently for n and m in the thousands.*

V(i,j)			w	r	i	t	e	r	s
		0	1	2	3	4	5	6	7
	0	0	-1	-2	-3	-4	-5	-6	-7
v	1	-1	-1	-2	-3	-4	-5	-6	-7
i	2	-2	-2	-2	-1	-2	-3	-4	-5
n	3	-3	-3	-3	-2	-2	-3	-4	-5
t	4	-4	-4	-4	-3	*			
n	5	5							
e	6	6							
r	7	7							

Figure 2: Similarity values are filled in one row at a time, and in each row they are filled in from left to right. The example shows the similarity values $V(i, j)$ to column 3 of row 4. The next value to be computed is $V(4, 4)$, where a * appears. The value for cell $(4, 4)$ is -1, since $S_1(4) = S_2(4) = t$ and $V(3, 3) = -2$.

The reader should be able to establish that the table could also be filled in *column-wise* instead of row-wise, after row zero and column zero have been computed. That is, column one could be first filled in, followed by column two, etc. Similarly, it is possible to fill in the table by filling in successive anti-diagonals. We leave the details as an exercise.

1.3 The traceback

Once the *value* of the similarity has been computed, how is the associated optimal alignment extracted? The easiest way (conceptually) is to establish *pointers* in the table as the table values are computed.

In particular, when the value of cell (i, j) is computed, set a pointer (arrow) from cell (i, j) to cell $(i, j - 1)$ if $V(i, j) = V(i, j - 1) - 1$; set a pointer from (i, j) to $(i - 1, j)$ if $V(i, j) = V(i - 1, j) - 1$; and set a pointer from (i, j) to $(i - 1, j - 1)$ if $V(i, j) = V(i - 1, j - 1) + t(i, j)$. This rule applies to cells in row zero and column zero as well. Hence, for most objective functions, each cell in row zero points to the cell to its left, and each cell in column zero points to the cell just above it. For other cells, it is possible, and common, that more than one pointer is set from (i, j) . Figure 3 shows an example.

The pointers allow easy recovery of an optimal alignment: Simply follow *any* path of pointers from cell (n, m) to cell $(0, 0)$. The optimal alignment is recovered (from the right end back to the left end) from that path. Each arrow (we also call this an "edge") along the path specifies another column in the alignment. If an arrow is a diagonal from cell (i, j) to cell $(i - 1, j - 1)$, then add a column containing character

V(i,j)			w	r	i	t	e	r	s
		0	1	2	3	4	5	6	7
	0	0	← -1	← -2	← -3	← -4	← -5	← -6	← -7
v	1	↑ -1	↖ -1	↖ ← -2	↖ ← -3	↖ ← -4	↖ ← -5	↖ ← -6	↖ ← -7
i	2	↑ -2	↖ ↑ -2	↖ -2	↖ -1	← -2	← -3	← -4	← -5
n	3	↑ -3	↖ ↑ -3	↖ ↑ -3	↑ -2	↖ -2	↖ ← -3	↖ ← -4	↖ ← -5
t	4	↑ -4	↖ ↑ -4	↖ ↑ -4	↑ -3	↖ -1	← -2	← -3	← -4
n	5	↑ -5	↖ ↑ -5	↖ ↑ -5	↑ -4	↑ -2	↖ -2	↖ ← -3	↖ ← -4
e	6	↑ -6	↖ ↑ -6	↖ ↑ -6	↑ -5	↑ -3	↖ -1	← -2	← -3
r	7	↑ -7	↖ ↑ -7	↖ -5	← ↑ -6	↑ -4	↑ -2	↖ 0	← -1

Figure 3: The complete dynamic programming table with pointers included. The arrow \leftarrow in cell (i, j) points to cell $(i, j - 1)$, the arrow \uparrow points to cell $(i - 1, j)$, and the arrow \swarrow points to cell $(i - 1, j - 1)$.

$S_1(i)$ opposite character $S_2(j)$. This column is added at the *left* end of the growing alignment; If the arrow is *horizontal*, from cell (i, j) to cell $(i, j - 1)$ then add a column, at the left, containing a space in the S_1 line opposite character $S_2(j)$; if the arrow is *vertical* from cell (i, j) to $(i - 1, j)$, then add a column, at the left, containing a space in the S_2 line opposite character $S_1(i)$. That fact that this traceback path specifies an optimal alignment is proved in a manner similar to the way that the recurrences for optimal alignment were established in the first place. We leave this as an exercise.

For example, there are two traceback paths from cell $(7, 7)$ to cell $(0, 0)$ in the example given in Figure 3. The paths are identical from cell $(7, 7)$ to cell $(1, 2)$ at which point it is possible to either go left or to go diagonally. The two optimal alignments are shown below.

$w \quad r \quad i \quad - \quad t \quad - \quad e \quad r \quad s$
 $v \quad - \quad i \quad n \quad t \quad n \quad e \quad r \quad -$

$w \quad r \quad i \quad - \quad t \quad - \quad e \quad r \quad s$
 $- \quad v \quad i \quad n \quad t \quad n \quad e \quad r \quad -$

If there is more than one pointer from cell (n, m) , then a path from (n, m) to $(0, 0)$ can start with *either* of those pointers. Each of them is on a path from (n, m) to $(0, 0)$. This property is repeated from any cell encountered. Hence a traceback path from (n, m) to $(0, 0)$ can start simply by following any pointer out of (n, m) , and then be extended by following any pointer out of any cell encountered. Moreover,

every cell except $(0, 0)$ has a pointer out of it, so no path from (n, m) can get stuck. Since any path of pointers from (n, m) to $(0, 0)$ specifies an optimal alignment, we have the following

Theorem 1.3 *Once the dynamic programming table with pointers has been computed, an optimal alignment can be found with $\Theta(n + m)$ simple operations. Hence this is even faster and more efficient than the task of filling in the DP table with V values.*

In class, we used an alternative way to find the optimal alignment, without using pointers, but the reasoning is very similar.

We have now completely described the three crucial pieces of the general dynamic programming paradigm, as illustrated by the optimal alignment problem.

1.3.1 The pointers represent *all* optimal alignments

The pointers that are built up while computing the values of the table do more than allow *one* optimal alignment to be retrieved. They allow *all* optimal alignments to be retrieved.

Theorem 1.4 *Any path from (n, m) to $(0, 0)$ following pointers established during the computation of $V(i, j)$ specifies an alignment of S_1 and S_2 with the maximum value. Conversely, any optimal alignment is specified by such a path. Moreover, since a path describes only one alignment, the correspondence between paths and optimal alignments is one-one.*

The theorem can be proven by essentially the same reasoning that established the correctness of the recurrence relations for $V(i, j)$, and is left to the reader.

References