Similarity Percentage Computation in SIM

Dick Grune
dick@dickgrune.com

May 18, 2016

1 Introduction

The similarity testing facility of *sim* has been working satisfactorily since its inception in 1986, but the computation of similarity percentages, introduced in 2001, has been a continuous source of headache.

There seem to be two reasons for this. The first is that there is no clear definition of the notion "similarity percentage". The second is that all attempts to obtain similarity percentages have resulted in nonsensical results (often percentages far exceeding 100%), or have required quadratic time, or sometimes both. Quadratic time requirements are unacceptable because they make *sim* unsuitable for large-scale text comparisons like plagiarism detection, one of *sim*'s largest application areas.

2 The Definition of "Similarity Percentage"

Normally *sim* supplies the information

Files F_1 and F_2 have matches $(T_{1,1}, T_{1,2})$, $(T_{2,1}, T_{2,2})$, ..., $(T_{k,1}, T_{k,2})$

where the text segments T-s in the matches have the form (F, p, l), designating the text in file F, starting at position p with length l, if we view a file as an array of tokens. The matches have the following properties (assuming a minimum run length of 1):

- For any match $((F_1, p_1, l_1), (F_2, p_2, l_2))$ we have $l_1 = l_2$ and $F_1[p_1..p_1+l_1] = F_2[p_2..p_2 + l_2]$ (i.e. a match matches equal segments).
- For each token $F_1[a]$ for which there is a token $F_2[b]$ such that $F_1[a] = F_2[b]$, there is a match $((F_1, p_1, l_1), (F_2, p_2, l_2))$ such that $p_1 \le a < p_1 + l_1$ and $p_2 \le b < p_2 + l_2$ (i.e. if two files have a token in common, it is present in a match).
- The segments $T_{1,1}$, $T_{2,1}, ..., T_{k,1}$ do not overlap. (This does not apply to $T_{1,2}$, $T_{2,2}, ..., T_{k,2}$.)
- If there is a match $((F_1, p_1, l_1), (F_2, p_2, l_2))$ there are no matches $((F_1, p_1 + l_1, ...), (F_2, ..., ...))$ or $((F_1, p_1 - l, l), (F_2, ..., ...))$ (i.e. no match could be extended on either end, i.e. each match is maximal in length).

This information is well-defined and unambiguous. It is produced by *sim* in almost linear time.

Unfortunately this information does not reasonably extend to something like

> Files F_1 and F_2 have N% material in common.

One is tempted to conclude that F_1 and F_2 have material of length $l_1 + l_2 + ... + l_k$ in common, but $l_1 + l_2 + ... + l_k$ can easily be larger than F_1 or F_2, resulting in similarities of over 100%. This can, f.e., happen when $F_1 = BB$ and $F_2 = B$, where B is a block of text with length l_B. Then the matches found between the two files are $((F_1, 0, l_B), (F_2, 0, l_B))$ and $((F_1, l_B, l_B), (F_2, 0, l_B))$. So the two files have a common length of $l_B + l_B = 2l_B$, which gives F_2 (with length l_B) a similarity percentage of 200%.

A more fundamental approach to similarity is needed. The fullest and clearest representation of what an N-token file F_1 and an M-token file F_2 have in common is an $N \times M$ Boolean correlation matrix C with $C[n, m] = True$ if $F_1[n] = F_2[m]$ and $C[n, m] = False$ elsewhere.

Suppose F_1=`"a b c d e f g h i j"` (length = 10) and F_2=`"f g h v w x y z a b c d e"` (length = 13). Then the correlation matrix C is

		F_2												
		f	g	h	v	w	x	y	z	a	b	c	d	e
	a									+				
	b										+			
	c											+		
	d												+	
F_1	e													+
	f	+												
	g		+											
	h			+										
	i													
	j													

If we project the pluses onto the top and left axes, we get an indication of how much material F_1 can be found in F_2 and vice versa:

		F_2												
		f	g	h	v	w	x	y	z	a	b	c	d	e
		*	*	*						*	*	*	*	*
	a	*								+				
	b	*									+			
	c	*										+		
	d	*											+	
F_1	e	*												+
	f	*	+											
	g	*		+										
	h	*			+									
	i													
	j													

From this representation we see that

> F_1 consists for $8/10 = 80.0\%$ of F_2 material
> F_2 consists for $8/13 = 61.5\%$ of F_1 material

It is this information that *sim* tries – and frequently fails – to provide.

3 The Comparison Machine

To see how this information can be obtained by *sim* or why it cannot, we have to have a closer look at *sim*'s comparison machine.

Sim cannot use the $N \times M$ matrix. Its size is quadratic in the number of tokens in the combined files to be compared, which would restrict the total number of tokens to 100.000 at the utmost. Instead it uses a linear array of tokens, with some additional data, as follows.

3.1 The Algorithm

The comparison machine (the routine `lcs()` in `compare.c`) uses two indexes into T: `i0` and `i1`. The index `i0` steps through the array, and for each position the index `i1` is used to hunt for the match $((F_1, i_0, l), (F_2, i_1, l))$ with largest l and $i_0 < i_1$.

The skeleton algorithm is

```
i0 := 0;
WHILE i0 < end_of_text
    i1 := 0;
    WHILE i1 < end_of_text
        IF T[i0] = T[i1]
            find longest match starting from(i0, i1)
            IF longer than previous longest match starting from(i0)
                store as longest match starting from(i0)
        i1 := i1 + 1
    IF there is a longest match starting from(i0)
        i0 := i0 + length of longest match starting from(i0)
    ELSE
        i0 := i0 + 1
```

In short, the inner loop finds long matches; the outer loop finds the longest of these long matches.

Suppose we have two files F_1 =`"abcdefabc"` and F_2 =`"fabcdef"`; the algorithm then identifies the matches

$((F_1, 0, 6), (F_2, 1, 6))$ (matching `"abcdef"`),
$((F_1, 6, 3), (F_2, 1, 3))$ (matching `"abc"`).

If the files had been offered in the reverse order, the matches would have been

$((F_2, 0, 4), (F_1, 5, 4))$ (matching `"fabc"`),
$((F_2, 4, 3), (F_1, 3, 3))$ (matching `"def"`),

which shows that the results of the comparison algorithm are sensitive to the order in which the files are presented.

3.2 Details of the Increments

Even after finding a match in the inner loop, `i1` should only be incremented by 1, since right after a good match a better match may be found. Suppose F_1=`"aaabcd"` and F_2=`"aaaabcd"`. With `i0=0` and `i1=7` (just at the beginning of F_2) we find the match

$$((F_1, 0, 3), (F_2, 0, 3)) \text{ (matching "aaa")},$$

but with `i1=8` we find

$$((F_1, 0, 6), (F_2, 1, 6)) \text{ (matching "aaabcd")},$$

which is a better match starting at `i0=0`.

When a longest match is found in the outer loop, `i0` must be increased by the length of that match, to avoid repetitive matches. Suppose we have one file F=`"aaaaaaaa"`. Then the first match will be

$$((F, 0, 4), (F, 4, 4)) \text{ (matching "aaaa")}.$$

When we increase `i0` in steps of 1, the rest of the matches will be

$((F, 1, 3), (F, 4, 3))$ (matching `"aaa"`),
$((F, 2, 3), (F, 5, 3))$ (matching `"aaa"`),
$((F, 3, 2), (F, 5, 2))$ (matching `"aa"`),
$((F, 4, 2), (F, 6, 2))$ (matching `"aa"`),
$((F, 5, 1), (F, 6, 1))$ (matching `"a"`),
$((F, 6, 1), (F, 7, 1))$ (matching `"a"`),

for a total of $O(N)$ matches, many of them repeating earlier information.

If, however, we increase `i0` by the length of the match found, the rest of the matches are

$((F, 4, 2), (F, 6, 2))$ (matching `"aa"`),
$((F, 6, 1), (F, 7, 1))$ (matching `"a"`),

for a total of $O(\log N)$ matches, all informative.

The actual algorithm has many other features: establishing a minimum match length, options for avoiding to compare a file to itself, etc. There is one important optimization, concerned with incrementing `i1`. Incrementing `i1` by 1 makes the algorithm quadratic in the number of tokens, which is unacceptable. We have seen, however, that sometimes increasing by 1 is necessary. The problem is heuristically solved by having an array `Forward_Reference[]`, which for each position in the token array gives the index of the nearest larger position in the token array where matching text of at least the minimum required length can be found. This array is constructed in linear time by a prescan, using hashing extensively. Rather than incrementing `i1` by 1 when looking for the next match, `i1` is set to `Forward_Reference[i1]`.

In principle using this forward references array does not take away the quadratic component, but it multiplies it by ϕ, the density of the similarities. And since we usually have $\phi \ll 1$, this optimization makes *sim* usable. The linear component of course remains.

4 Problems and their Causes

The arrangement described above is not directly suitable for percentage computation, for three reasons.

1. The contents of F_k are explained in terms of matches with files $F_{k+1}, ..., F_n$; matches from $F_1, ..., F_{k-1}$ are not noticed.
2. Not all matches in $F_{k+1}, ..., F_n$ are noticed.
3. Only a single match for a block at a given position in F_k is reported; other lesser matches or equal matches further on are not noticed.

Each of these problems will now be considered in turn.

4.1 Ignoring Earlier Files

Earlier files are ignored because the `Forward_Reference` array is exactly that, it references forward. Normally this is not a problem, since any similarities between F_k and an earlier F_p (with $1 \leq p \leq k-1$) will be have been noted when F_p was analyzed.

One may be tempted to try to deduce the contribution of a file F_p ($1 \leq p \leq k-1$) to F_k from matches of the form $((F_p, ,), (F_k, ,))$, but that is not possible. Suppose $F_p = BaBBc$ and $F_k = aBc$, where B is a block of text of size l_B. The algorithm provides us with the matches

$((F_p, 0, l_B), (F_k, 1, l_B))$ (matching B),
$((F_p, l_B, 1 + l_B), (F_k, 0, 1 + l_B))$ (matching aB),
$((F_p, l_B + 1 + l_B, l_B + 1), (F_k, 1, l_B + 1))$ (matching Bc)

Note that in none of the matches the text in F_p overlaps.

These matches are perfect for determining the N in the statement "F_p consists for N% of F_k material": $N = (l_B + 1 + l_B + l_B + 1)/(l_B + 1 + l_B + l_B + 1) = (3l_B + 2)/(3l_B + 2) = 100\%$. But if we rely on these matches to compute the N in the statement "F_k consists for N% of F_p material", we arrive at the conclusion that $N = (l_B + 1 + l_B + l_B + 1)/(l_B + 2) = (3l_B + 2)/(l_B + 2) \approx 300\%$. It is clear that this is caused by the overlap of the matches in F_k, but this example makes it equally clear that it would be very difficult to disentangle such overlaps in the general case.

4.2 Selectively Ignoring Later Files

When there are a number of files, say four, each containing somewhere an identical block of text B, say $F_1 = abBcd, F_2 = efgB, F_3 = Bhi, F_4 = jklBmno$ (where $a..o$ are letters not occurring in B) *sim* produces the following three matches

$((F_1, 2, l_B), (F_2, 3, l_B))$,
$((F_2, 3, l_B), (F_3, 0, l_B))$,
$((F_3, 0, l_B), (F_4, 3, l_B))$

producing a linear number of matches, which is very desirable for most applications. But it means that the matches

$((F_1, 2, l_B), (F_3, 0, l_B))$,
$((F_1, 2, l_B), (F_4, 3, l_B))$,
$((F_2, 3, l_B), (F_4, 3, l_B))$

are missed. So percentages between F_1 and F_2, F_2 and F_3, and F_3 and F_4 could possibly be derived from these matches, but those between F_1 and F_2, F_1 and F_4, and F_2 and F_4 cannot.

As we have noted before, the output of *sim* is sensitive to the order in which the files are presented.

4.3 Ignoring Lesser Matches

Given the three files $F_1 = abBcd, F_2 = efgB, F_3 = bBhi$, *sim* produces the following two matches

$((F_1, 1, l_B + 1), (F_3, 0, l_B + 1))$, (matching bB)
$((F_2, 3, l_B), (F_3, 1, l_B))$, (matching B)

but the lesser match $((F_1, 2, l_B), (F_2, 3, l_B))$, relating F_1 to F_2, is missed. This is acceptable in similarity testing since the B in F_2 is caught by $((F_2, 3, l_B), (F_3, 1, l_B))$, but for similarity percentage computation we need all matches.

5 Solutions

5.1 Accessing Earlier Files

There are at least two ways to obtain both matches of the form $((F_p, ,), (F_q, ,))$ *and* of the form $((F_q, ,), (F_p, ,))$ with $p < q$: modifying the order in which the files are processed, and modifying the forward reference system.

5.1.1 Modifying the File Order

Running the entire program again with the order of the files reversed will provide the missing $((F_q, ,), (F_p, ,))$, at the expense of doubling the running time.

5.1.2 Modifying the Forward Referencing System

Each forward reference chain is particular to a specific string S of text, with $|S|$ the minimum required match length. The chain starts from the first occurrence of S in the text array, then leads to one or more subsequent occurrences of S, and ends in a NULL pointer at the last occurrence of S in the text array. When at a position in F_p we start using the forward reference chain for say S, we can reach only positions in F_q with $q > p$ (not $q \geq p$, since when computing percentages a file is not compared to itself).

Now suppose the chain, rather than terminating with a NULL pointer at the last occurrence of S, looped back to its beginning. Then when working on F_p we do not stop at the last occurrence of S in the text array while hunting for S but continue at the first occurrence of S in the text array, from where we may access files F_q with $q < p$. This statistically doubles the length of the chain, doubling the work of the comparison machine; but the effort in the preparation of the text array is not doubled, as it would be if the program were run twice.

A circular list is an awkward data structure and requires careful programming. During construction of a forward reference chain it can easily be made to loop, by remembering where it started and then rather than terminating it with a NULL pointer end it with a pointer to the starting point. When using the chain, following it must stop when we reach the position from which we started (which is certainly on the chain).

So the problem of how to access earlier files can be solved at the expense of roughly doubling the run time.

5.2 Full Coverage of Later Files

The problem is caused by the original algorithm hunting for the largest match in the rest of the files, whereas for percentage computation it should look for any match in each of the other files separately.

The possibility of comparing each file with each other file separately has already been provided by the **-e** option. Again this results in a quadratic time requirement, and again we invoke the low density of actual matches to reduce the weight of the problem.

The original algorithm produces the matches in a left-to-right largest match order, to minimize the number of matches; for the percentage computation we just want them all. Since the nature and number of matches depends on the order in which they are identified, one might wonder if the left-to-right largest match order may cause problems to percentage computation. Suppose F_p=`"abcdxe"` and F_q=`"abcdxxe"`, with a required minimum run length of 2. The matches found are

```
abcdxe        abcdxxe
abcdx         abcdx
     e             e
```

with the result that F_q consists for $5/7 = 71.4\%$ out of F_p material. The match `e` is ignored because it does not have the required minimum length. If the files were compared in the reverse order the following matches would be obtained

```
abcdxxe         abcdxe
abcdx           abcdx
     xe             xe
```

and F_q would consist for $(5 + 2)/7 = 100\%$ of F_p material

Considerations:

- The example is contrived and hinges on there being a minimum required length and on multiply overlapping patterns. No examples seem to exist without these ingredients.
- Multiply overlapping patterns do occur in practice, f.e. in lists of identifiers or numbers, but these are usually not very informative and of little interest.
- The algorithm for finding the maximum similarity percentage may conceivably require exhaustive search and be exponential.

It seems reasonable to ignore this problem for the moment, and use the normal *sim* matching algorithm in comparing one file to one file[1].

5.3 Lesser Matches

Since lesser matches can only occur when three or more files are involved, using the `-e` option solves this problem too, since it restricts all comparisons to two files only.

6 Conclusion

Using circular forward reference chains and the `-e`-option, correct similarity percentages can be obtained; the algorithm has a moderate quadratic component though.

A compromise can be obtained by omitting the `-e` option, which is the greatest source of quadratic behaviour, and use circular forward reference chains only. This makes the algorithm almost linear again, but will underreport percentages.

[1]The problem did show up in test runs, manifesting itself by minor differences in the percentages depending on the order of the input files, but went away when the run length was set to 1.