Notes on string similarity measures
or
Three NLP algorithms for the price of one

Martin Jansche

2006-09-19
Corrected version

1 Classical pairwise string similarity

1.1 Alignments

Informally, an alignment of n strings is a sequence of n-tuples where by concatenating the kth components of all tuples in sequence one obtains the kth original string. More precisely, the kth component in each tuple contains either a symbol from the kth string or the empty string; and tuples consisting exclusively of empty strings are excluded.

Consider the strings ab and stu. Then possible alignments of these strings include $(a,s)\,(\varepsilon,t)\,(b,u)$ and $(\varepsilon,s)\,(\varepsilon,t)\,(a,u)\,(b,\varepsilon)$ where ε denotes the empty string.

Here are all 25 possible alignments of ab and stu:

```
a b             a   b           a b                 a b             a   b
    s t u     s   t   u       s   t u           s t u           s   t u

a   b           a     b       a   b             a b                 a b
  s   t u     s   t u         s t   u             s t u         s t   u

a     b             a b       a     b           a   b           a b
  s t   u     s t     u       s t u               s t u         s t u

a       b           a   b       a b               a b           a   b
  s t u       s t   u         s t   u           s   t u         s t u

  a b               a b         a   b           a     b           a b
s     t u     s t u           s t u               s t u         s t u
```

1.2 Alignment-based distance

For each tuple (a,b) define a cost $c(a,b)$.

The cost of an alignment is defined as

$$c((a_1, b_1) \dots (a_m, b_m)) = \sum_{k=1}^{m} c(a_k, b_k)$$

Let $A(x, y)$ denote the set of all alignments of strings x and y. The alignment-based distance $d(x, y)$ between x and y is defined as

$$d(x, y) = \min_{a \in A(x,y)} c(a) \tag{1}$$

1.3 Levenshtein distance

The Levenshtein distance between two strings is now typically understood to be the alignment-based distance with the following basic costs:

$$c_{\text{Lev}}(a, b) = \begin{cases} 0 & \text{if } a = b \\ 1 & \text{otherwise} \end{cases}$$

The distance based on the above costs is equal to the minimum number of insertions, deletions, and substitutions required to transform one string into the other.

Levenshtein originally defined a second notion of string similarity with the following basic costs:

$$c_{\text{indel}}(a, b) = \begin{cases} 0 & \text{if } a = b \\ 2 & \text{if } \varepsilon \neq a \neq b \neq \varepsilon \\ 1 & \text{otherwise} \end{cases}$$

The distance based on the above costs is equal to the minimum number of insertions and deletions (but not substitutions) required to transform one string into the other.

1.4 Computation

The challenge in computation lies in carrying out the minimization in (1) efficiently. This cannot be done naively, by enumerating all possible alignments, because their number grows exponentially in the length of the strings.

When aligning two strings of equal length n, the number of alignments is given by the Delannoy sequence, which grows faster than any polynomial in n:

n	$\sum_{k=0}^{n} \binom{n}{k}\binom{n+k}{k}$	n	$\sum_{k=0}^{n} \binom{n}{k}\binom{n+k}{k}$
0	1	5	1683
1	3	6	8989
2	13	7	48639
3	63	8	265729
4	321	9	1462563

The classic dynamic programming solution for computing alignment-based string similarity was (re)invented many times and is known by many different names (Needleman–Wunsch algorithm in biology, Wagner–Fischer algorithm in computer science, among others). The basic insight is that string similarity can be expressed recursively and the overall solution computed by combination of memoized sub-solutions.

Let x_1^i be the prefix of string x of length i. Let x_i be the ith element of string x. Let $d[i.j]$ be the Levenshtein distance between x_1^i and y_1^j. Then

$$d[i,j] = \min \left\{ \begin{array}{l} d[i-1,j] + c(x_i, \varepsilon), \\ d[i,j-1] + c(\varepsilon, y_j), \\ d[i-1,j-1] + c(x_i, y_j) \end{array} \right\}$$

for $i \geq 1$ and $j \geq 1$, with $d[0,0] = 0$ and $d[i,-1] = d[-1,j] = \infty$.

The dynamic programming algorithm stores d as a matrix, which is explored either row-by-row or column-by-column. The runtime is $\mathrm{O}(|x| \times |y|)$.

Example: Calculate the Levenshtein distance between ‘poetry’ and ‘theater’:

			t	h	e	a	t	e	r
		∞	∞	∞	∞	∞	∞	∞	∞
	∞	0	1	2	3	4	5	6	7
p	∞	1	1	2	3	4	5	6	7
o	∞	2	2	2	3	4	5	6	7
e	∞	3	3	3	2	3	4	5	6
t	∞	4	3	4	3	3	3	4	5
r	∞	5	4	4	4	4	4	4	4
y	∞	6	5	5	5	5	5	5	5

Exercises

1. Calculate the Levenshtein distance between the following pairs of strings:
   (a) ‘edit’ and ‘distance’
   (b) ‘two’ and ‘one’
   (c) ‘three’ and ‘thirteen’
   (d) ‘twentyfour’ and ‘fourteen’
2. State bounds on the Levenshtein distance $d(x, y)$ given only $|x|$ and $|y|$, the lengths of the two strings.

2 Generalizations based on FSTs

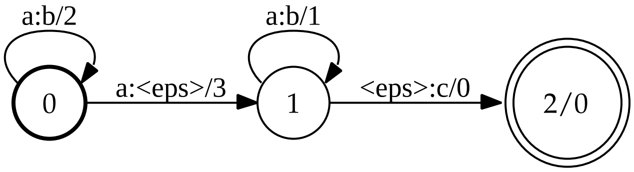
We now decompose the classic dynamic programming algorithm for calculating weighted string edit distance and re-express it in terms of operations on weighted finite state transducers (FSTs).

2.1 Weighted FSTs

An unweighted FST defines a relation between regular languages. Weighted FSTs are more appropriately conceptualized as defining a function which assigns a cost or weight to a pair of strings.

More precisely, given an FST T, we say that it assigns a cost $[\![T]\!](x, y)$ to a given pair of strings (x, y), defined as the minimum cost across all accepting paths labeled with (x, y) (notice how this parallels (1) above). The cost of a path is the sum of the cost of its edges.

Consider the following example, call it T_0:



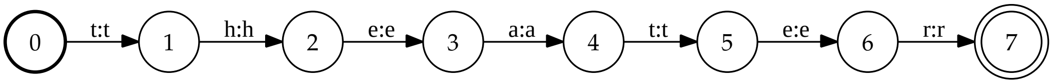
What is $[\![T_0]\!](aaa, bbc)$?

2.2 Representing individual strings

Given a string w, construct an FST $\text{Str}(w)$ with behavior

$$[\![\text{Str}(w)]\!](x, y) = \begin{cases} 0 & \text{if } w = x = y \\ \infty & \text{otherwise} \end{cases}$$

Example: $\text{Str}(\textit{theater})$ is shown below:



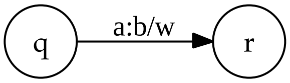
2.3 FST composition

The composition of two FSTs S and T is defined as

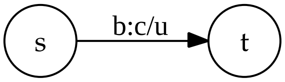
$$[\![S \circ T]\!](x, z) = \min_y [\![S]\!](x, y) + [\![T]\!](y, z)$$

The FST $S \circ T$ is constructed as the product of S and T: each state of $S \circ T$ is a pair whose first component is a state of S and whose second component is a state of T.

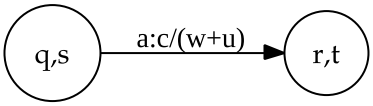
Edges of $S \circ T$ are formed as follows.

If 

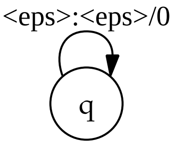
is an edge of S

and 

is an edge of T

then 

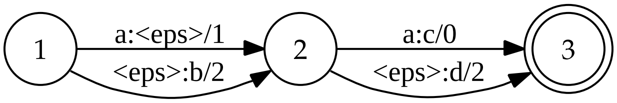
is an edge of $S \circ T$ (and nothing else is).

We assume that S and T have ε-loops 

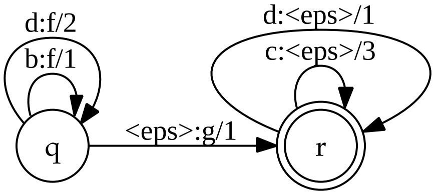
at each state q.

(This construction does not work in general – e.g. it needs to be modified to handle stochastic FSTs – but is correct for the special cases we consider here.)

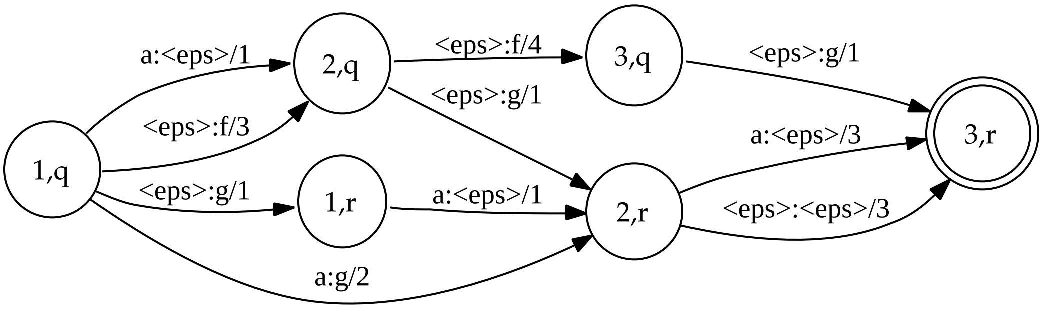
Example Let S be the transducer depicted here:



and let T be the transducer depicted below:

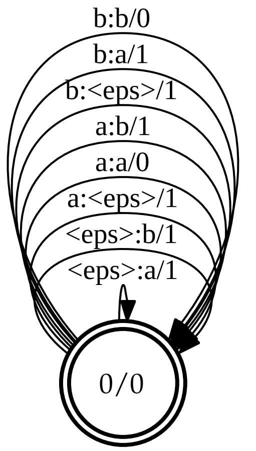


Then $S \circ T$ is the following FST:



2.4 Simple map FSTs and trellises

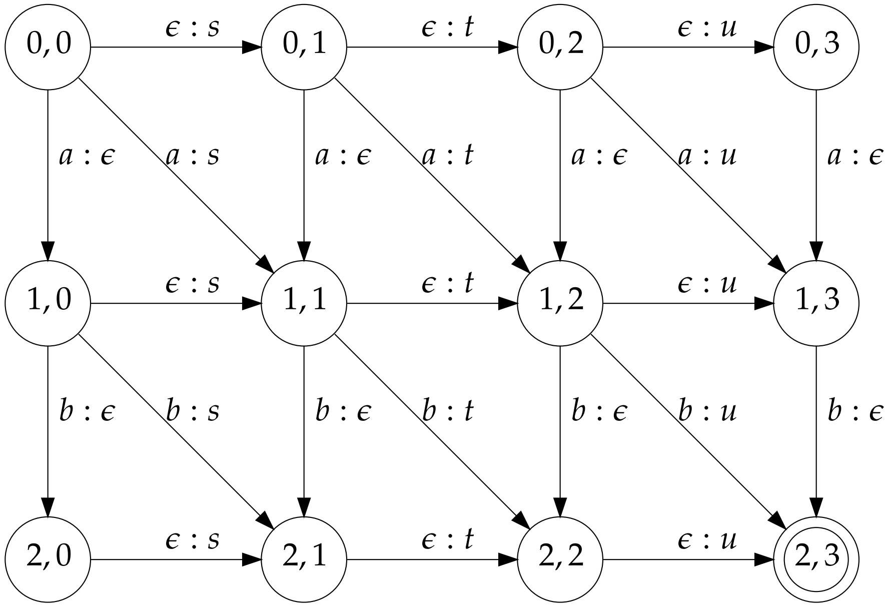
Consider the following FST, call it M, over the alphabet $\{a, b\}$ consisting of a single state and several loops:



Observe that $[\![M]\!](x, y)$ is precisely the Levenshtein distance between x and y.

To evaluate the Levenshtein distance of x and y, we first form the composite FST $\mathrm{Str}(x) \circ M \circ \mathrm{Str}(y)$, also known as a trellis.

Example The trellis representing all 25 alignments of ab and stu is shown below (costs are not shown):



2.5 Cost computation

The second step in evaluating the Levenshtein distance between x and y, once we have formed the trellis graph/FST, is to compute the least cost of all accepting paths through the trellis. Let $n = \max\{|x|, |y|\}$; then the trellis has $\mathrm{O}(n^2)$ vertices and $\mathrm{O}(n^2)$ edges.

We have several choices for computing the cost of the trellis. Since the cost of the whole trellis is precisely the cost of the least-cost path, we could use a single-source shortest path algorithm. The Bellman–Ford algorithm runs in time $\mathrm{O}(V\,E)$. which is $\mathrm{O}(n^4)$ in our case. Dijkstra's algorithm runs in $\mathrm{O}(V \log V + E) = \mathrm{O}(n^2 \log n)$ time.

However, since the trellis is the result of composing a map transducer on the left and right with two acyclic transducers, the trellis itself is always acyclic, i.e., it is a directed acyclic graph (DAG). This means the DAG shortest path algorithm can be used, which runs in $\Theta(V + E) = \Theta(n^2)$ time.

Not only is the runtime of the DAG shortest path algorithm the same as that of the standard dynamic programming algorithm for evaluating the Levenshtein distance, on closer inspections it turns out that standard algorithm is a special case of the DAG shortest path computation.

The DAG shortest path algorithm explores the vertices of the graph in topological order. In general, one would have to first sort the vertices of a DAG topologically using depth-first search, but in our special case the topological order is known *a priori*.

2.6 Consequences and generalizations

We have just shown that the weighted string edit distance between x and y can be computed as

$$\mathrm{bestpath}(\mathrm{Str}(x) \circ M \circ \mathrm{Str}(y))$$

for a suitable choice of M, which defines the basic costs.

Several consequences of this formulation are immediately obvious:

1. x and y can be strings over distinct alphabets, as long as M defines a suitable correspondence. This is useful for transliteration and transcription problems.

2. The costs of M can be arbitrary (including negative costs and infinite costs). There is no requirement that exact matches must carry a cost of zero, that M must be symmetric, that all insertion costs must be homogeneous, etc. (unless one wants the resulting function to have certain properties, like being a proper distance metric). Useful for cost-sensitive correction, e.g. in OCR correction the cost of confusing 'f' with 't' should be lower than the cost of confusing 'f' with 'm'.

3. The topology of M is not restricted to a single state. M can have any desired shape. However, the number of states of M has an impact on the

overall absolute runtime (though not the asymptotic runtime in $|x|$ and $|y|$). This is useful for capturing accidental duplication of letters in spelling correction.

Next, the computation of

$$\mathrm{bestpath}(\mathrm{Str}(x) \circ M \circ \mathrm{Str}(y))$$

can be generalized to

$$\mathrm{bestpath}(\mathrm{Str}(x) \circ M \circ Y)$$

where Y is an arbitrary finite automaton. When Y is acyclic, the DAG shortest path algorithm still applies. For example, in automatic spelling correction, Y may represent a finite dictionary of known words; then the above computation will find the closest matching word for x.

Finally, when the meaning of the weights of the FSTs is changed, so that weights no longer represent costs but instead represent probabilities, the DAG shortest path computation on the composed trellis either remains a shortest path computation (in which case we get, after one tiny change, the Viterbi algorithm) or changes into an algebraic path computation (in which case we get the Forward algorithm without any changes). You will encounter these two algorithms later in this course.

Exercises

1. Specify an FST which computes the following edit distance: Matching symbols carry a cost of zero, substitution and deletion carry a cost of one, but insertions are treated specially – an insertion operation ordinarily carries a cost of one, but the cost is only 0.5 when a symbol is inserted that is identical to the preceding symbol on the second tape of the FST. For example, the alignment $(a, b)\,(\varepsilon, b)$ has total cost 1.5, since the cost of the substitution (a, b) is 1 and the cost of the insertion (ε, b) in this context is 0.5. On the other hand, the alignment $(b, c)\,(\varepsilon, b)$ has a total cost of 2.

2. Specify an FST which computes an edit distance which is identical to the standard Levenshtein distance except that all deletion costs are doubled after the first deletion of the letter x.