# Chapter 7

# Sequence Alignment

## 7.1 Problem

The next problem we will tackle is similar to sequence labeling. Previously, we were given a collection of sequence pairs: a sequence of words and a sequence of labels. Suppose that we are given a collection of sequence pairs, but not necessarily of the same length, and we want to be able to predict one from the other. For example:

- Given misspelled words (or "alternatively spelled" words like teh, pleez, etc.) and their correct spellings, can we learn to correct the spelling of new words?
- Given Japanese words that have been borrowed from English, and the original English words, can we learn to predict the English origin of other English loanwords in Japanese?
- Given English words and their pronunciations, can we learn to pronounce new English words?

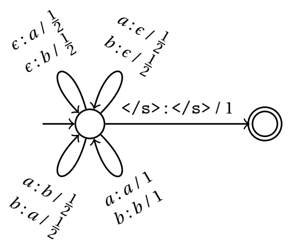
## 7.2 Spelling correction

Let's think about spelling correction, although exactly the same machinery might work well for the other examples mentioned above as well. You may already be familiar with one possible method that can be thought of as a sequence alignment problem: *string edit distance* or *Levenshtein distance.* The string edit distance is defined in terms of three basic operations:

- Insert a symbol, with cost 1
- Delete a symbol, with cost 1
- Change a symbol into a difference symbol, with cost 1

The minimum edit distance problem is: Given two strings $w$ and $w'$, find the minimum number of operations needed to transform $w$ into $w'$.

We can represent these operations using a finite-state transducer. Something new that we will use below is transitions that have empty ($\epsilon$) input or output.



This is for the alphabet $\{a, b\}$; larger alphabets would need more arcs. We gave transitions with edits a weight of $\frac{1}{2}$, which is arbitrary. Then the weight of any path is $2^{-k}$, where $k$ is the number of edits. Note that this is a weighted FST but not a probabilistic FST.

A consequence of the presence of empty transitions is that given an input string $w$ and an output string $w'$, there is more than one path through the FST that maps $w$ to $w'$. For example, to change abba into baba, a couple of ways are:

$$\begin{tikzcd}
\text{a} \arrow[d] & \text{b} \arrow[d] & \text{b} \arrow[d] & \text{a} \arrow[d] & \\
\text{b} & \text{a} & \text{b} & \text{a} &
\end{tikzcd} \quad \text{2 substitutions}$$

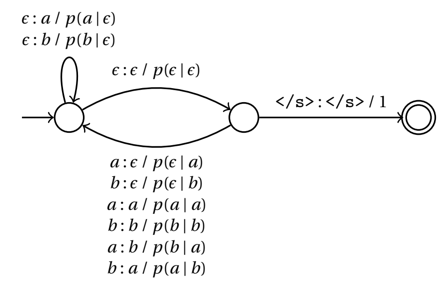
$$\begin{tikzcd}
\text{a} & \text{b} \arrow[dl] & \text{b} \arrow[d] & \text{a} \arrow[d] \\
\text{b} & \text{a} & \text{b} & \text{a}
\end{tikzcd} \quad \text{1 deletion, 1 insertion}$$

So (as you already know if you're familiar with the minimum edit distance problem) computing the number of edits is not trivial. We can compose this FST with automata representing $w$ and $w'$ to obtain a FST that accepts only $w$ as input and outputs only $w'$, and every path through the FST represents a possible sequence of edits. Running the Viterbi algorithm on this FST to find the highest-weight path is exactly the same as the standard minimum edit distance algorithm.

But there are a number of problems we need to solve:

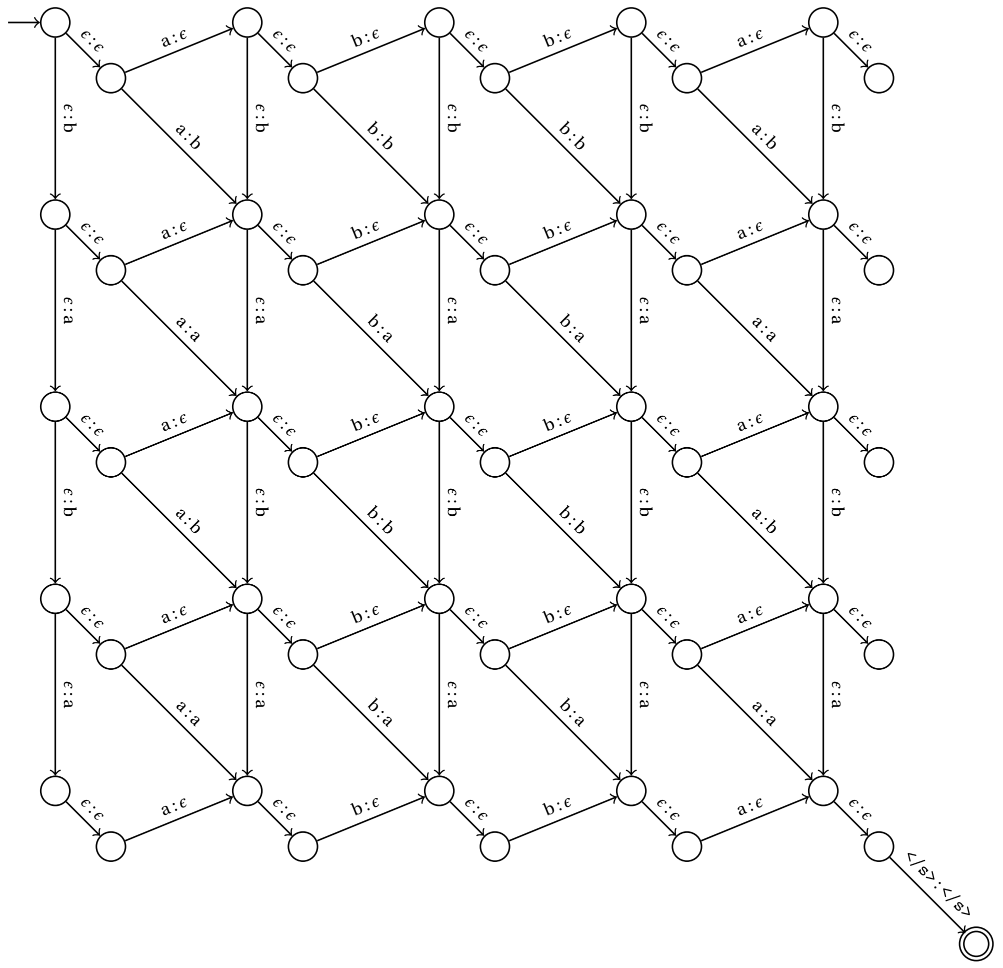
- How do we use this for spelling correction or text normalization? That is, given $w'$, find the word $w$ that it is closest to, taking into account that some words are more likely than others?
- What if the three operations shouldn't have equal weight, or equal weight for each symbol? How can we learn them from data?
- What if don't even have access to a gold-standard dictionary or text?

We will tackle these problems using probability models (which edit distance is not). In order for a weighted FST with empty-input transitions to be probabilistic, we need to impose the following restriction: for each state, all outgoing transitions either all have empty input, or none of them do. Then, a weighted FST is probabilistic if, for each state, all outgoing transitions with the same input symbol sum to one. (If we allowed nonempty-input transitions and empty-input transitions to mix, it would be tricky to define the sum-to-one constraint.) That means that we need to modify the above FST into the following:



We've replaced the numerical weights with parameters $p(\sigma' \mid \sigma)$ that now need to be estimated. For each $\sigma$ (including $\epsilon$), we should have $\sum_{\sigma'} p(\sigma' \mid \sigma) = 1$ (where the summation includes $\sigma' = \epsilon$).

When intersected with FSTs representing an input string $w$ and an output string $w'$, we get something that looks like this (taking $w$ = abba and $w'$ = baba):

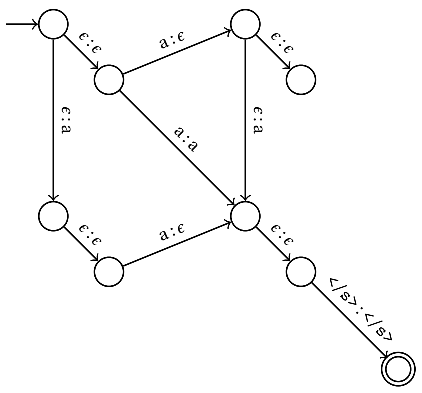


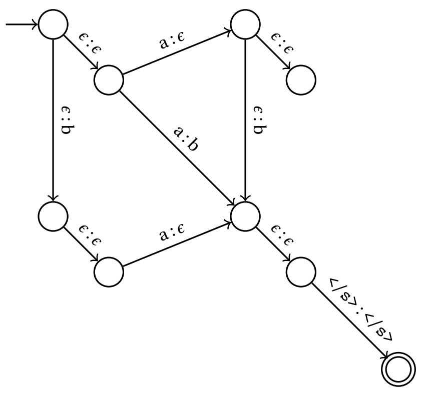
where each edge labeled $\sigma : \sigma'$ has probability $p(\sigma' \mid \sigma)$.

## 7.3 Training

Because of the nondeterminism discussed above, we can't just train the weights by counting and dividing. This is an example of a partially-supervised learning problem: we are given $w$ and $w'$, but we're not given the specific sequence of operations used to get from $w$ to $w'$. So, just as when we did unsupervised sentence clustering and topic modeling, we need to use EM.

In the following, let $M_{w,w'}$ be the FST formed by composing $M$ with FSTs for $w$ and $w'$. The four-letter example from above will be unwieldy to illustrate training, so let's switch to two shorter string pairs:





$w = \text{a}$ $w' = \text{a}$ $w = \text{a}$ $w' = \text{b}$

We initialize the parameters uniformly:

$$p(\text{a} \mid \text{a}) = 1/3 \qquad p(\text{a} \mid \text{b}) = 1/3 \qquad p(\text{a} \mid \epsilon) = 1/3$$
$$p(\text{b} \mid \text{a}) = 1/3 \qquad p(\text{b} \mid \text{b}) = 1/3 \qquad p(\text{b} \mid \epsilon) = 1/3$$
$$p(\epsilon \mid \text{a}) = 1/3 \qquad p(\epsilon \mid \text{b}) = 1/3 \qquad p(\epsilon \mid \epsilon) = 1/3$$

### 7.3.1 Hard EM

E step: In hard EM, all we do for the E step is to choose the single best path. That's just the Viterbi algorithm. For both training examples, that's the path that has no insertions or deletions. The counts for the two training examples are:

$$c(\epsilon,\epsilon) = 2 \qquad c(\epsilon,\epsilon) = 2$$
$$c(\text{a},\text{a}) = 1 \qquad c(\text{a},\text{b}) = 1$$

Summing over training examples, we get

$$c(\text{a},\text{a}) = 1$$
$$c(\text{a},\text{b}) = 1$$
$$c(\epsilon,\epsilon) = 4$$

M step: Count the number of times each parameter occurs along all the best paths and normalize. So

$$\begin{aligned} c(\epsilon,\epsilon) &= 4/4 = 1 \\ c(\mathrm{a},\mathrm{a}) &= 1/2 \\ c(\mathrm{a},\mathrm{b}) &= 1/2 \end{aligned}$$

```
repeat
    ▷ E step
    for each string pair $w, w'$ do
        compute best path through $M_{w,w'}$
        for each parameter $p(\sigma' \mid \sigma)$ along path do        ▷ $\sigma$ or $\sigma'$ can be $\epsilon$
            $c(\sigma,\sigma') \leftarrow c(\sigma,\sigma') + 1$
        end for
    end for
    ▷ M step
    for all $\sigma$ do
        $Z \leftarrow \sum_{\sigma'} c(\sigma,\sigma')$
        for all $\sigma'$ do
            $p(\sigma' \mid \sigma) \leftarrow c(\sigma,\sigma')/Z$
        end for
    end for
until done
```

### 7.3.2 Real slow EM

If we want to do real EM, we need to compute the probability of each path, *given $w$ and $w'$*, and pretend that we saw each path that many times. For each of our training examples, there are three paths.

| first example | | | | | | second example | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| path | | | | weight | count | path | | | | weight | count |
| $\epsilon:\epsilon$ | a : $\epsilon$ | $\epsilon$ : a | $\epsilon:\epsilon$ | 1/81 | 1/5 | $\epsilon:\epsilon$ | a : $\epsilon$ | $\epsilon$ : b | $\epsilon:\epsilon$ | 1/81 | 1/5 |
| $\epsilon:\epsilon$ | a : a | $\epsilon:\epsilon$ | | 1/27 | 3/5 | $\epsilon:\epsilon$ | a : b | $\epsilon:\epsilon$ | | 1/27 | 3/5 |
| $\epsilon$ : a | $\epsilon:\epsilon$ | a : $\epsilon$ | $\epsilon:\epsilon$ | 1/81 | 1/5 | $\epsilon$ : b | $\epsilon:\epsilon$ | a : $\epsilon$ | $\epsilon:\epsilon$ | 1/81 | 1/5 |

Summing over training examples and then normalizing, we get

$$\begin{aligned} c(\mathrm{a},\mathrm{a}) &= 3/5 & p(\mathrm{a}\mid\mathrm{a}) &= 3/10 \\ c(\mathrm{a},\mathrm{b}) &= 3/5 & p(\mathrm{b}\mid\mathrm{a}) &= 3/10 \\ c(\mathrm{a},\epsilon) &= 4/5 & p(\epsilon\mid\mathrm{a}) &= 4/10 = 2/5 \\ c(\epsilon,\mathrm{a}) &= 2/5 & p(\mathrm{a}\mid\epsilon) &= 2/24 = 1/12 \\ c(\epsilon,\mathrm{b}) &= 2/5 & p(\mathrm{b}\mid\epsilon) &= 2/24 = 1/12 \\ c(\epsilon,\epsilon) &= 4 & p(\epsilon\mid\epsilon) &= 20/24 = 5/6 \end{aligned}$$

```
▷ E step
for each string pair $w, w'$ do
    $Z \leftarrow$ total weight of all paths through $M_{w,w'}$
```

```
        for each path $\pi$ through $M_{w,w'}$ do
            for each edge $e$ along $\pi$ do
                for each parameter $p(\sigma' \mid \sigma)$ on $e$ do          ▷ $\sigma$ or $\sigma'$ can be $\epsilon$
                    $c(\sigma,\sigma') \leftarrow c(\sigma,\sigma') + \text{weight}(\pi)/Z$
                end for
            end for
        end for
    end for
    ▷ M step: same as above
```

But don't implement this algorithm! The loops over all paths give it an exponential time complexity.

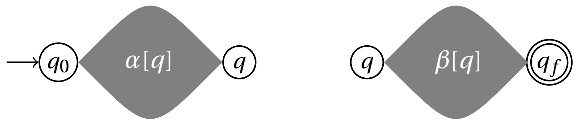
### 7.3.3 Real fast EM

The way to make this efficient is analogous to the way that we made the Viterbi algorithm efficient using dynamic programming. First, observe that the slow algorithm visits each edge many times, each time adding a (usually very small) number to $c(\sigma,\sigma')$. We can reorder the loops so that we visit each edge $e$ exactly once, and the increment that we add to $c(\sigma,\sigma')$ is the *total* weight of all the paths that go through $e$. Call this $\gamma[e]$.

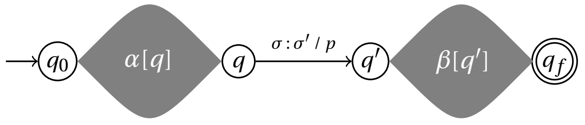
```
▷ E step
for each string pair $w, w'$ do
    $Z \leftarrow$ total weight of all paths through $M_{w,w'}$
    for each edge $e$ of $M_{w,w'}$ do
        $\gamma[e] \leftarrow 0$
        for each path $\pi$ that goes through $e$ do
            $\gamma[e] \leftarrow \gamma[e] + \text{weight}(\pi)$
        end for
        for each parameter $p(\sigma' \mid \sigma)$ on $e$ do          ▷ $\sigma$ or $\sigma'$ can be $\epsilon$
            $c(\sigma,\sigma') \leftarrow c(\sigma,\sigma') + \gamma[e]/Z$
        end for
    end for
end for
▷ M step: same as above
```

How do we calculate $Z$? This is a simple extension of the Viterbi algorithm. If Viterbi finds the maximum weight of all paths because at each step it performs a maximum operation at each step, then we can find the sum of the weights of all paths if at each step we perform a sum operation.

```
procedure FORWARD($M$)
    $\alpha[q_0] \leftarrow 1$
    $\alpha[q] \leftarrow 0$ for $q \neq q_0$
    for each state $q'$ in topological order do
        for each incoming transition $q \rightarrow q'$ with weight $p$ do
            $\alpha[q'] \leftarrow \alpha[q'] + \alpha[q] \times p$
        end for
    end for
    return $\alpha$
end procedure
```



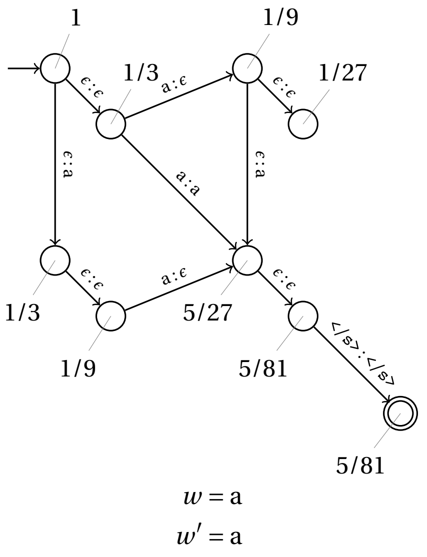
(a) (b)

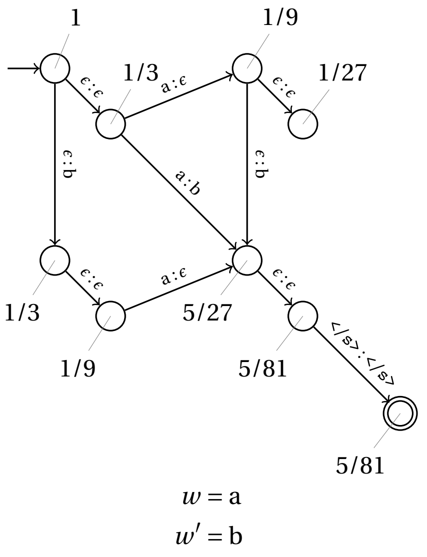


(c)

Figure 7.1: (a) The forward probability $\alpha[q]$ is the total weight of all paths from the start state to $q$. (b) The backward probability $\beta[q]$ is the total weight of all paths from to $q$ to the final state. (c) The total weight of all paths going through an edge $q \to q'$ with weight $p$ is $\alpha[q] \times p \times \beta[q']$.

The $\alpha[q]$ are called *forward probabilities.* Each is the total weight of all paths from the start state to $q$ (see Figure 7.1a). So $Z$ is the forward probability of the final state.





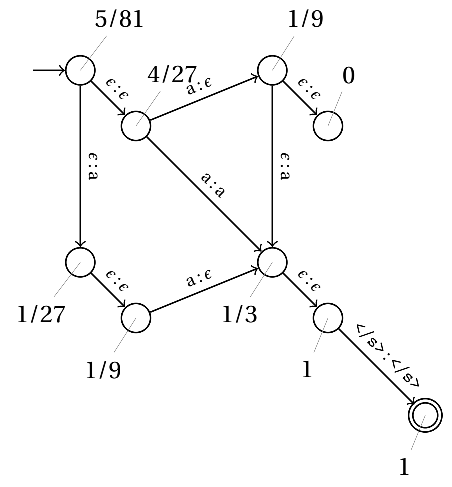
It's easy to do the same computation in reverse, to obtain *backward probabilities* $\beta[q]$, each of which is the total weight of all paths from $q$ to the final state (see Figure 7.1b).

```
procedure BACKWARD($M$)
    $\beta[q_f] \leftarrow 1$
    $\beta[q] \leftarrow 0$ for $q \neq q_0$
    for each state $q$ in reverse topological order do
```

```
            for each outgoing transition $q \to q'$ with weight $p$ do
                $\beta[q] \leftarrow \beta[q] + p \times \beta[q']$
            end for
        end for
        return $\beta$
    end procedure
```

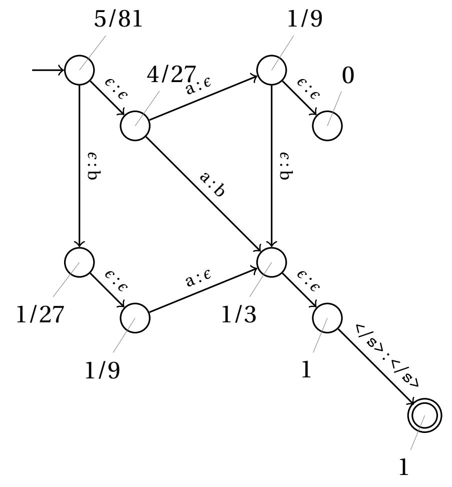
If it's not convenient to iterate over the outgoing transitions of a state, then this version is equivalent:

```
    procedure BACKWARD($M$)
        $\beta[q_f] \leftarrow 1$
        $\beta[q] \leftarrow 0$ for $q \neq q_0$
        for each state $q'$ in reverse topological order do
            for each incoming transition $q \to q'$ with weight $p$ do
                $\beta[q] \leftarrow \beta[q] + p \times \beta[q']$
            end for
        end for
        return $\beta$
    end procedure
```



$w = \text{a}$

$w' = \text{a}$



$w = \text{a}$

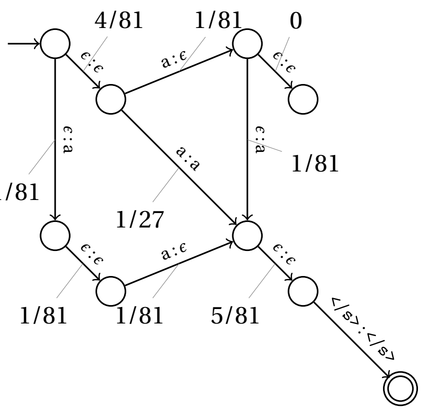
$w' = \text{b}$

Notice that the backward probability of the initial state equals the forward probability of the final state – this must always be true, and is a good way to check for errors.

What do backward probabilities buy us? If $e$ is a transition $q \to q'$ with weight $p$, we can compute $\gamma[e]$: it is $\alpha[q] \times p \times \beta[q']$ (see Figure 7.1c). If you prefer to think about it in equations, let $\pi_{<e}$ be the prefix

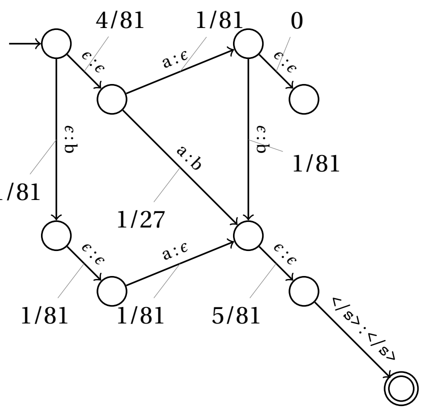
of path $\pi$ up to but not including $e$, and similarly for $\pi_{>e}$; then

$$\begin{aligned}
\gamma[e] &= \sum_\pi \text{weight}(\pi) \\
&= \sum_\pi \text{weight}(\pi_{<e}) \times p \times \text{weight}(\pi_{>e}) \\
&= \left(\sum_\pi \text{weight}(\pi_{<e})\right) \times p \times \left(\sum_\pi \text{weight}(\pi_{>e})\right) \\
&= \alpha[q] \times p \times \beta[q'].
\end{aligned}$$



$w = \text{a}$

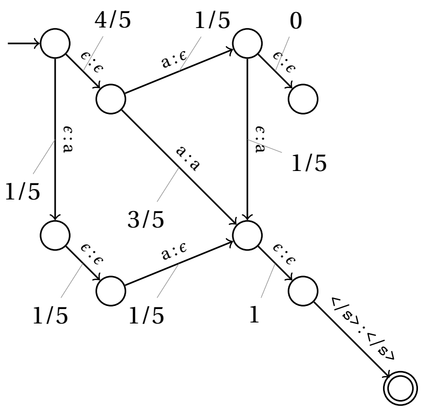
$w' = \text{a}$



$w = \text{a}$

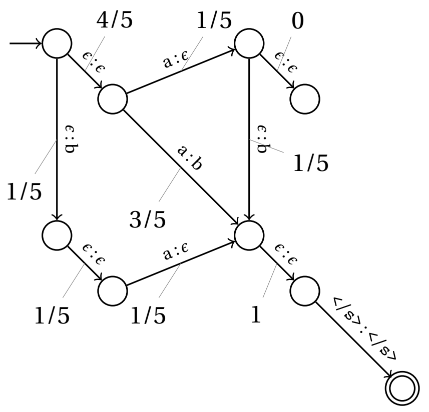
$w' = \text{b}$

To get the probability that each edge is used, *given $w$ and $w'$*, we divide each weight by $Z$.



$w = \text{a}$

$w' = \text{a}$



$w = \text{a}$

$w' = \text{b}$

Then, sum up the counts for each parameters. You can check that it comes out exactly the same as when we enumerated all the paths.

Putting it all together, we get the *forward-backward algorithm*:

▷ E step

```
for each string pair $w, w'$ do
    $\alpha \leftarrow \textsc{Forward}(M_{w,w'})$
    $\beta \leftarrow \textsc{Backward}(M_{w,w'})$
    $Z \leftarrow \alpha[q_f]$
    for each edge $e = q \rightarrow q'$ of $M_{w,w'}$ do
        $p \leftarrow$ weight of $e$
        $\gamma[e] \leftarrow \alpha[q] \times p \times \beta[q']$
        for each parameter $p(\sigma' \mid \sigma)$ on $e$ do        ▷ $\sigma$ or $\sigma'$ can be $\epsilon$
            $c(\sigma,\sigma') \leftarrow c(\sigma,\sigma') + \gamma[e]/Z$
        end for
    end for
end for
▷ M step: same as above
```

## 7.4 Aligning and decoding

Once we've trained the model, what do we do with it? Given $w$ and $w'$, we can try to find the most probable way of getting from $w$ to $w'$. This is useful, for example, if we are interested in aligning $w$ with $w'$ to see which symbols are in correspondence. To do that, form $M_{w,w'}$ exactly as before, and run the Viterbi algorithm.

Or, given $w'$, we can find the most probable $w$. This problem is sometimes called *decoding*. Previously, we composed the edit-model FST with a FST for $w$ on the input side and a FST for $w'$ on the output side. Now, on the input side, we compose with a language model. This FST can input any string but only output $w'$, and the probability of each path is

$$\text{weight}(\pi) = P(w)P(\pi \mid w).$$

Then use the Viterbi algorithm to find the best path. The most probable $w$ should be

$$\begin{aligned} w^* &= \arg\max_w P(w \mid w') \\ &= \arg\max_w P(w)P(w' \mid w) \\ &= \arg\max_w P(w) \sum_{\pi \mid \pi \text{ outputs } w'} P(\pi \mid w), \end{aligned}$$

but that is unfortunately an NP-hard problem! We can find the best path, and that is usually good enough.