12 Symbolic MT 2: Weighted Finite State Transducers

The previous section introduced a number of word-based translation models, and introduced their parameter estimation methods and application to alignment. However, it intentionally glossed over an important question: how to *generate* translations from them. This section introduces a general framework for expressing our models graphs: **weighted finite-state transducers**. It explains how to encode a simple translation model within this framework, and how this allows us to perform search.

12.1 Graphs and the Viterbi Algorithm

Before getting into the details of expressing our actual models, let's look a little bit in the abstract about an algorithm to do search over a graph. Without getting into the details about how we obtained the graph, let's say we have a graph such as the one in Figure 33. Each edge of the graph represents a single word, with a weight representing whether the word is likely to participate in a good translation candidate or not. Actually, in these sorts of graphs, it is common to assume that higher weights are worse, and to search for the path through the graph that has the lowest overall score. Thus, of the hypotheses encoded in this graph "the tax is" is the best, with the lowest score of 2.5.

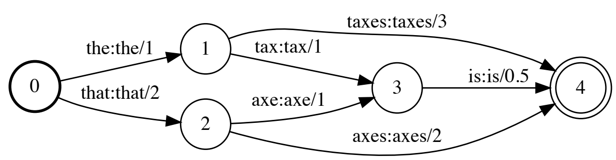

Figure 33: An example of a graph.

So how do we perform this search? While there are a number of ways, the most simple and widely used is called the **Viterbi algorithm** [9]. This algorithm works in two steps, a *forward calculation* step, where we calculate the best path to each node in the graph, and then a backtracking step, in which we *follow back-pointers* from one state to another.

In the forward calculation step, we step through the graph in topological order, visiting each node in an order so that when visiting a node, all preceding nodes have already been visited. For the initial node ("0" in the graph), we set its path score $a_0 \leftarrow 0$. Next, we define all edges g as a tuple $\langle g_\mathrm{p}, g_\mathrm{n}, g_\mathrm{x}, g_\mathrm{s} \rangle$, where g_p is the previous node, g_n is the next node, g_x is the word, and g_s is its score (weight). When processing a single node, we step through all its incoming edges, and calculate the minimum of the sum of the edge score and the path score of the preceding node,

$$a_i \leftarrow \min_{g \in \{\tilde{g}; \tilde{g}_\mathrm{n} = i\}} a_{g_\mathrm{p}} + g_\mathrm{s}. \tag{116}$$

We also calculate a "back pointer" to the edge that resulted in this minimum score, which we use to re-construct the highest scoring hypothesis at the end of the algorithm:

$$b_i \leftarrow \operatorname*{argmin}_{g \in \{\tilde{g}; \tilde{g}_\mathrm{n} = i\}} a_{g_\mathrm{p}} + g_\mathrm{s} \tag{117}$$

In the example above, the calculation would be equal to:

$$
\begin{aligned}
a_1 &= a_0 + g_{\text{the,s}} \\
&= 0 + 1 = 1 \\
b_1 &= g_{\text{the}} \\
a_2 &= a_0 + g_{\text{that,s}} \\
&= 0 + 2 = 2 \\
b_2 &= g_{\text{that}} \\
a_3 &= \min\left(a_1 + g_{\text{tax,s}}, a_2 + g_{\text{axe,s}}\right) \\
&= \min\left(1 + 1, 2 + 1\right) = 2 \\
b_3 &= g_{\text{tax}} \\
a_4 &= \min\left(a_1 + g_{\text{taxes,s}}, a_2 + g_{\text{axes,s}}, a_3 + g_{\text{is,s}}\right) \\
&= \min\left(1 + 3, 2 + 3, 2 + 0.5\right) = 2.5 \\
b_4 &= g_{\text{is}}
\end{aligned}
$$

The next step is the back-pointer following step. In this step, we start at the final state ("4") in the example, and iterate over the back-pointers g_{p} of each edge, one by one. First, we observe b_4, note the word $g_{\text{is,x}}$ is "is", then step to $g_{\text{is,p}} = 3$. We continue to follow b_3, note the word "tax", step to b_1, note the word "the", step to b_0 and terminate because we've reached the beginning of the sentence. This leaves us with the words "is tax the", which we then reverse to obtain "the tax is", our highest scoring hypothesis.

12.2 Weighted Finite State Automata and A Language Model

This sort of graph where $g = \langle g_{\text{p}}, g_{\text{n}}, g_{\text{x}}, g_{\text{s}} \rangle$ is also called a **weighted finite state automaton** (WFSA). These WFSAs can be used to express a wide variety of strings and their weightings over them,[35] and being able to think about various tasks in this way opens up possibilities for doing a wide variety of processing in a single framework. The following explanation describes some basic properties of WFSAs, and interested readers can reference [7] for a comprehensive explanation.

One example of something that can be expressed as an WFSA is the smoothed n-gram languages models described in Section 3. Let's say that we have a 2-gram language model interpolated according to Equation 9 over the set of words "she", "i", "ate", "an", "a", "apple", "peach", "apricot" calculated from the following corpus:

$$
\begin{aligned}
&\text{she ate an apple} \\
&\text{she ate a peach} \\
&\text{i ate an apricot}
\end{aligned}
\tag{118}
$$

We also assume that the interpolation coefficient is $\alpha = 0.1$.

[35]Specifically, they are able to express all **regular languages**, with a weight assigned to each string contained therein.

Given this, we will have two classes of probabilities, one where the bigram count is non-zero, such as $P(\text{apple} \mid \text{an})$, which (sparing the details) becomes the probability:

$$\begin{aligned} P(e_t = \text{apple} \mid e_{t-1} = \text{an}) &= (1-\alpha) P_{ML}(e_t = \text{apple} \mid e_{t-1} = \text{an}) + \alpha P_{ML}(e_t = \text{apple}) \\ &= 0.9 \frac{1}{2} + 0.1 \frac{1}{15} \\ &= 0.45\overline{6}. \end{aligned}$$

We also have probabilities where where the bigram count is zero, these are essentially equal to the unigram probability discounted by α. For example:

$$\begin{aligned} P(e_t = \text{apple} \mid e_{t-1} = \text{a}) &= (1-\alpha) P_{ML}(e_t = \text{apple} \mid e_{t-1} = \text{a}) + \alpha P_{ML}(e_t = \text{apple}) \\ &= 0 + 0.1 \frac{1}{15} \\ &= 0.00\overline{6} \end{aligned}$$

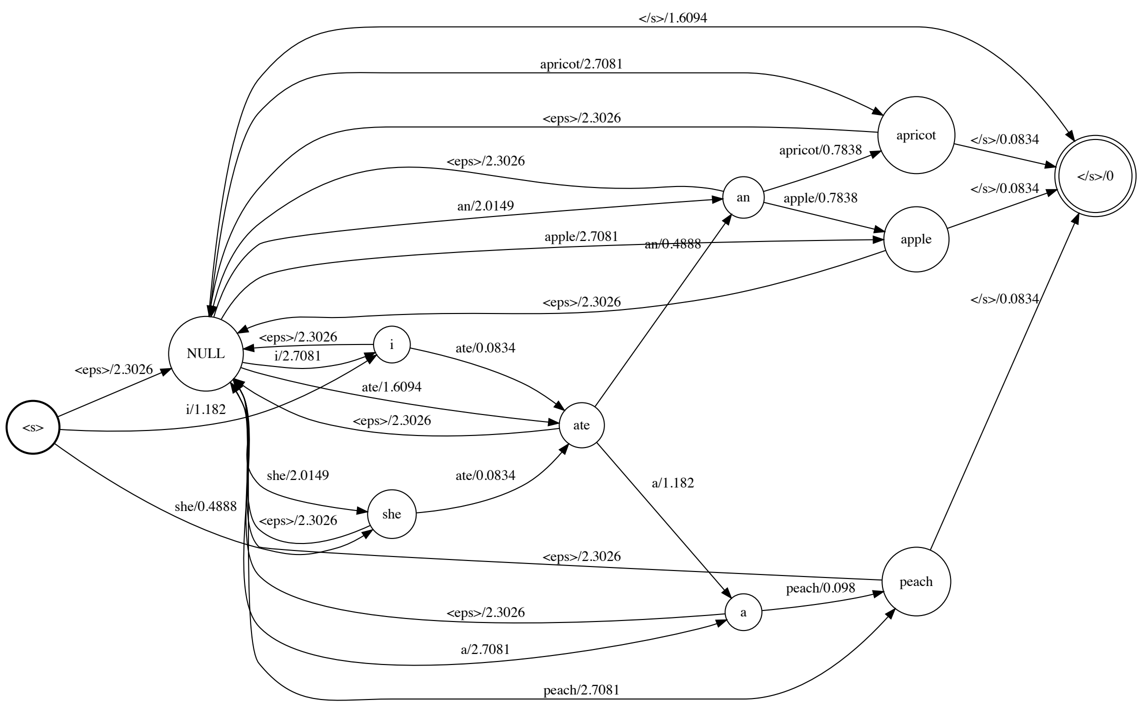

Figure 34: A 2-gram language model as a WFSA. Edge labels are "word/score", where the score is represented as a negative log probability. States are labeled with the context e_{t-1} that the represent, where "NULL" represents unigrams. "⟨eps⟩" represents an ϵ edge, which can be used to fall back from the unigram state to the bigram state.

The way we express this in a WFSA is shown in Figure 34. Each state label indicates the bigram context, so the state labeled "an" will represent probabilities $P(e_t \mid e_{t-1} = \text{an})$. Edges outgoing from a labeled state (with an edge label that is not "⟨eps⟩", which we will get to later), represent negative log bigram probabilities. So for example, $P(e_t = \text{apple} \mid e_{t-1} = \text{an}) = 0.45\overline{6}$, which indicates that the edge outgoing from the state "an" that is labeled with "apple" will have an edge weight of $-\log P(e_t = \text{apple} \mid e_{t-1} = \text{an}) \approx 0.7838$. We also have a state labeled "NULL", which represents all unigram probabilities $P(e_t)$. All outgoing edges here represent a unigram probability.

Now, to the ⟨eps⟩ edges, which are called ϵ edges or ϵ transitions. ϵ edges are basically edges that we can follow at any time, without consuming a token in the input sequence. In the case of language models, these transitions are used to express the fact that sometimes we won't have a transition that we can match for a particular context, and will instead want to fall back to a more general context using interpolation. For example, after "an" we may see the word "peach" and want to calculate its probability. In this case, we would fall back from the "an" state to the "NULL" state using the ϵ edge, which incurs a score of $-\log \alpha = 2.3026$, then follow the edge from the "NULL" state to the "peach" state, resulting in a probability of $-\log P(e_t = \text{peach}) = 2.7081$. Of course, we could also create an edge directly from "an" to "peach" with a probability $-\log \alpha P(e_t = \text{peach})$, but by using the ϵ edges we are able to avoid explicitly enumerating all pairs of words, improving our memory efficiency while obtaining the exact same results.

12.3 Weighted Finite State Transducers and a Translation Model

As could be seen from the previous section, WFSAs are able to express sets of strings with corresponding scores over them. This is enough for when we want to express something like a language model, but what if we want to express a translation model, that takes in a string and translates it into another string. This sort of string transduction can be done with another formalism called **weighted finite state transducers** (WFSTs). WFSTs are essentially similar to WFSAs with an addition symbol g_{y}, leading to $g = \langle g_{\text{p}}, g_{\text{n}}, g_{\text{x}}, g_{\text{y}}, g_{\text{s}} \rangle$. Thus, each edge takes in a symbol, outputs another symbol, and gives a score to this particular transduction.

To give a very simple example, let's assume a translation model that is even simpler than IBM Model 1: one that calculates $P(F \mid E)$ by taking one e_t at a time and independently calculates the translation probability of the corresponding word f_t

$$P(F \mid E) = \prod_{t=1}^{|E|} P(f_t \mid e_t). \tag{119}$$

Assume that we have the following Spanish corpus equivalent to our English corpus in Equation 118:[36]

$$\begin{gathered}\text{ella comió una manzana}\\ \text{ella comió un melocotón}\\ \text{yo comi un albaricoque}\end{gathered} \tag{120}$$

In this case, we can learn translation probabilities for each word, for example:

$$P(f = \text{ella} \mid e = \text{she}) = 1 \tag{121}$$

[36]Spanish allows dropping of the pronoun "yo" – equivalent to "i" – and a natural translation would probably do so. But for the sake of simplicity, let's leave it in to maintain the one-to-one relationship with the English words, and we'll deal with the problem of translations that are not one-to-one in a bit.

or

$$P(f = \text{comió} \mid e = \text{ate}) = 0.66\overline{6} \tag{122}$$
$$P(f = \text{comi} \mid e = \text{ate}) = 0.33\overline{3}. \tag{123}$$

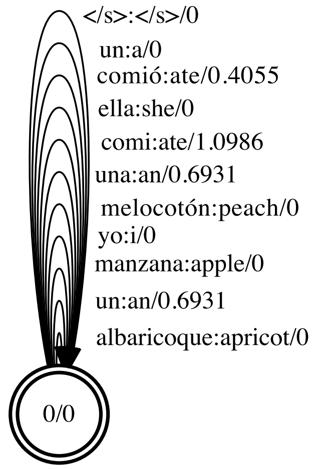

Figure 35: A graph of a word-to-word translation model where $P(F|E) = \prod_{t=1}^{|E|} P(f_t \mid e_t)$.

To express these translation probabilities as a WFST, we define a WFST where the input symbol g_x is the word f, the output symbol g_y is the word e, and the weight g_s is the negative log probability $-\log P(f \mid e)$. Because translation probability is independent of the others, we do not need to maintain the "state" of the translation model, and thus the we can just use a single input and output state for every edge. Figure 35 shows an example of a WFST representing this translation model.

12.4 Composing Multiple WFSTs

OK, so now we have a WFSTs calculating our language model probability $P(E)$ as shown in Figure 34, and translation model probability $P(F \mid E)$ as shown in Figure 35. But, as in all statistical translation models, what we really want to do is find the best translation:

$$\begin{aligned}\hat{E} &= \underset{E}{\operatorname{argmax}}\ P(E \mid F) \\ &= \underset{E}{\operatorname{argmax}}\ P(F \mid E)P(E).\end{aligned} \tag{124}$$

This requires combining together the scores $P(F \mid E)$ and $P(E)$, which are each expressed as separate WFSTs, so how do we do so?

Luckily, one of the benefits of the WFST framework is that it has general purpose algorithms that allow us to perform a number of common operations, independent of the underlying semantics of the WFSTs themselves. One of these operations is **composition**, which combines together two consecutive operations into a single one. In other words, if we have two WFSTs representing functions $T_1(X)$, and $T_2(x)$, if we perform composition we can create a new WFST

$$T_3(X) = T_2(T_1(X)). \tag{125}$$

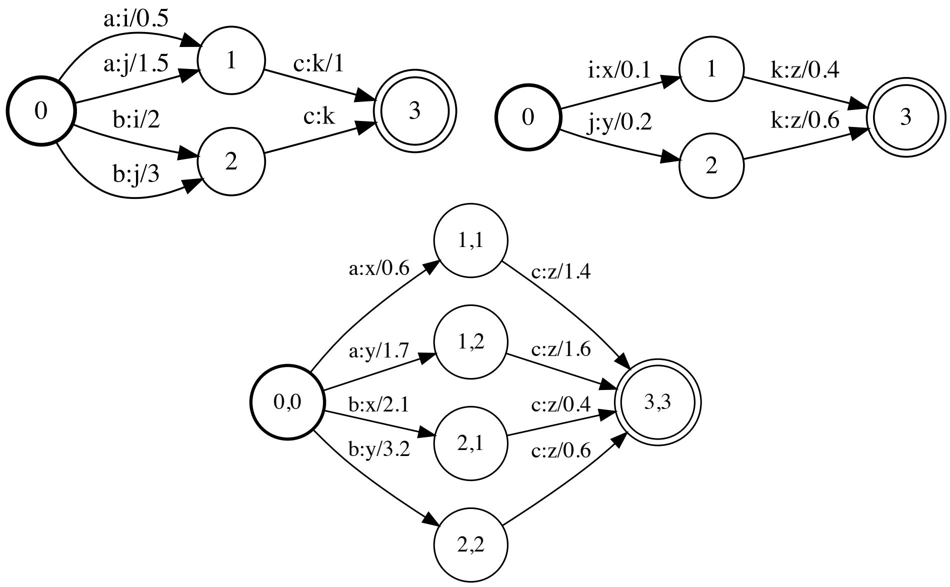

Figure 36: An example two simple transducers (top) composed into one (bottom).

This composition operation over transducers is generally expressed as $T_3 = T_1 \circ T_2$.

The full detail of the composition algorithm is beyond the scope of this chapter (and interested readers can reference [7]), but the general procedure is basically as follows:

1. Add a pair of initial nodes $\langle 0, 0\rangle$ to a stack S.
2. For each pair of nodes $\langle n_1, n_2\rangle$ in S that has not already been processed, in topological order:
   (a) Step over each pair of edges e_1 and e_2 where $e_{1,\mathrm{p}} = n_1$ and $e_{2,\mathrm{p}} = n_2$ respectively.
   (b) If $e_{1,\mathrm{y}} = e_{2,\mathrm{x}}$, add $\langle n_1, n_2\rangle$ to the stack S, and create a new edge in T_3 which has a previous state $e_{3,\mathrm{p}} = \langle e_{1,\mathrm{p}}, e_{2,\mathrm{p}}\rangle$, next state $e_{3,\mathrm{n}} = \langle e_{1,\mathrm{n}}, e_{2,\mathrm{n}}\rangle$, input symbol $e_{3,\mathrm{x}} = e_{1,\mathrm{x}}$, output symbol $e_{3,\mathrm{y}} = e_{2,\mathrm{y}}$, and score $e_{3,\mathrm{s}} = e_{1,\mathrm{s}} + e_{2,\mathrm{s}}$.

Figure 36 shows an example of the composition of two simple transducers. As an example of combining two edges in the two transducers into a single edge in the output transducer, we can see the edges $e_1 = \langle 0, 1, \mathrm{a}, \mathrm{j}, 1.5\rangle$ and $e_2 = \langle 0, 2, \mathrm{j}, \mathrm{y}, 0.2\rangle$ get composed into $e_3 = \langle\langle 0, 0\rangle, \langle 1, 2\rangle, \mathrm{a}, \mathrm{y}, 1.7\rangle$.

Next, as a more concrete example, I'll show an example from the translation model that we've been talking about so far. The WFST in Figure 35 can be viewed as a function $T_{P(F|E)}(F)$ that takes an input F, and outputs a graph encoding all possible E along with their negative log probabilities $P(F \mid E)$. The WFST in Figure 34 is a function $T_{P(E)}(E)$ that takes in E and returns the same E but with the addition of a score representing the negative log probability according to the bigram model. The composition of these two $T_{P(F|E)} \circ T_{P(E)}$, gives us a function that takes an input F and outputs candidate Es with scores that reflect $-\log P(F \mid E) - \log P(E)$, which we will call our $T_{P(E|F)}$. An example of this composed transducer is shown in Figure 37. As we can see, the general structure of the WFST basically

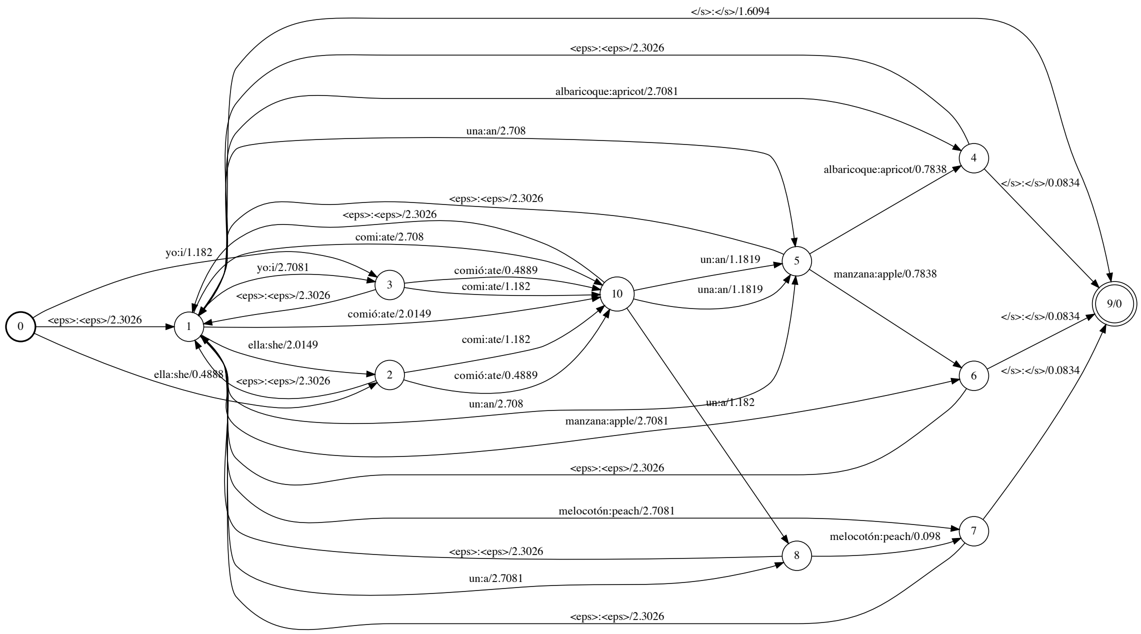

Figure 37: A WFST representing the negative log probability of E given F ($T_{P(E|F)}$), created by composing Figure 35 ($T_{P(F|E)}$) with Figure 34 ($T_{P(E)}$).

follows that in Figure 34, but with the addition of input representing Spanish words, and different arcs when different Spanish words can translate into a single English word.

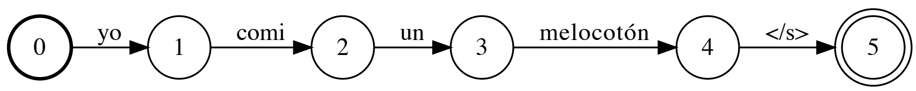

Figure 38: A WFST representing the input T_F.

Now, we can use this model to actually perform translations. The way we do so is by creating another WFSA to represent the input T_F; the WFSA in Figure 38 is an example for "yo comi un melocotón". This WFSA is then composed with $T_{P(E|F)}$ to obtain a search graph, as shown in Figure 39, which encodes all of our possible translations, and paths through the language model WFST. The actual graph we acquire through the composition process, on the top of the figure, includes many ϵ transitions for every time we might fall back to the unigram context in the bigram language model. To make it easier to read and more compact, we can also run an ϵ removal algorithms ([6]), which collapses the ϵ transitions to leave only the best-scoring path, and giving us the easy-to-read graph at the bottom of the figure. From this graph, we can see that we have two candidate translations "i ate a peach" and "i ate an peach". We can then run the Viterbi algorithm from Section 12.1 to obtain the best scoring translation, which in this case will be "i ate a peach", which was given much higher probability by the language model than its counterpart using "an peach".

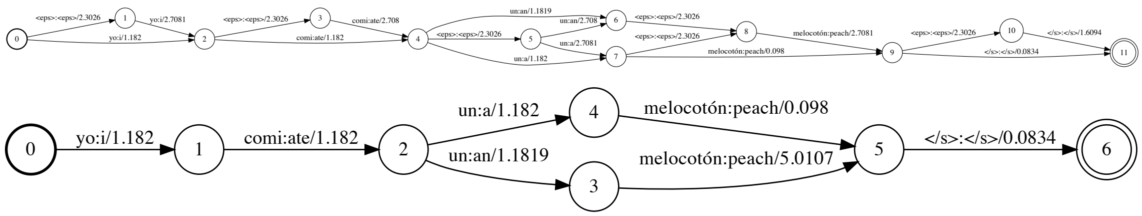

Figure 39: A WFST representing all of the candidate translations for input "yo comi un melocotón", with epsilon transitions (on top) and after these epsilon transitions have been removed (on the bottom).

12.5 Other WFST Models

One thing that we should stress at this point is that the *only* problem-specific knowledge encoded in the process above was used in the construction of our model transducers $T_{P(E|F)}$ and $T_{P(E)}$. The other pieces of the previous section, specifically the Viterbi search algorithm and the WFST composition algorithm, were both entirely general and equally applicable to other tasks.

Because of the elegant and extensible framework for processing symbol sequences provided by WFSTs, they have seen wide use in a number of sequence-to-sequence processing tasks. We will cover some examples from machine translation in the following chapter, and the following is a far-from-comprehensive list of some examples from other areas:

Speech processing: WFSTs are behind many standard speech recognizers [7]. In this case, it is common to create transducers such as an acoustic model T_A that converts speech features into phonemes, a pronunciation dictionary T_D that converts phonemes into words, and a language model T_L. The whole speech recognition process is represented as the composition of the input speech features T_X with the composed model $T_A \circ T_D \circ T_L$.

Down-stream tasks for speech: Because speech recognizers often are based on WFSTs, it is also common to express down-stream tasks that consume speech as input as WFSTs as well. Some examples include dialog management [4], which manages the flow of a dialog system, and transcript cleaning [8], which converts a transcript with colloquial expressions into a more clean and readable version for archival purposes.

Models of words: It is also common to use transducers to create models of the characters in words for various purposes. For example, it is possible to do pronunciation estimation [3], estimating the pronunciation of words from their spelling, or processing of the morphology of words in morphologically rich languages [2].

One thing that the observant reader will note is that we have not discussed any more complicated model for machine translation than the simple word substitution model introduced in Section 12.3. The reason for this is that, despite their desirable properties, WFSTs do have one relatively major weak point: it is not trivial to model problems that require reordering of the elements in the input and output. As machine translation problems do require this reordering, they will require slightly more involved methods for creating graphs, which we will cover in the next chapter.

12.6 Other Properties of WFSTs

While this chapter touched on a variety of interesting properties of WFSTs and demonstrated how they can be used to formalize models and search problems that we are interested in, it just scratched the surface of this very elegant and extensible formalism. For example, there is an interesting concept of **semi-rings**, which can be used to change the semantics of the weights in the WFST, allowing us to perform a number of operations using the same underlying formalism and algorithms. For example, we can perform Viterbi search (using the "tropical" semi-ring [7]), marginalization over all of the paths in the graph (using the "log" or "probability semi-rings" [7]), calculation of expectations of feature weights for discriminative training (usign the "expectation" semi-ring [5]), and calculation of edit distance between strings (using the "edit-distance" semi-ring [1]). There are also other algorithms over WFSTs in addition to the search, composition, and epsilon removal algorithms noted above. These can allow us to perform unions or intersections over the languages expressed by WFSTs, or efficiently compress complicated models consisting of the composition of multiple component WFSTs into the most efficient form possible, greatly improving processing speed [7].

12.7 Exercise

The exercise this time will be to combine the simple word-to-word translation model introduced in Section 12.3 with the 2-gram language model that was introduced in Section 3. The probabilities of the word-to-word translation model can be estimated with IBM Model 1, or whatever variant you implemented in the exercise Section 11. This whole implementation exercise will involve:

- Creating WFSTs to represent the translation model and the 2-gram language model, compiling them, and composing them together.
- Creating one WFSA for each input, composing it with the model WFST, and performing shortest path to find the best answer.

Models can be implemented in OpenFST (`http://openfst.org`) which should make things easier. OpenFST allows you to specify models in text format, compile them into binary, compose together WFSTs, perform shortest path search, and print the resulting output. This can be done either through a command line interface, C++ code, or Python bindings.

References

[1] Corinna Cortes, Patrick Haffner, and Mehryar Mohri. Rational kernels. In *Proceedings of the 15th International Conference on Neural Information Processing Systems*, pages 617–624. MIT Press, 2002.

[2] Markus Dreyer, Jason Smith, and Jason Eisner. Latent-variable modeling of string transductions with finite-state methods. In *Proceedings of the 2008 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1080–1089, 2008.

[3] Timothy J Hazen, I Lee Hetherington, Han Shu, and Karen Livescu. Pronunciation modeling using a finite-state transducer representation. *Speech Communication*, 46(2):189–203, 2005.

[4] Chiori Hori, Kiyonori Ohtake, Teruhisa Misu, Hideki Kashioka, and Satoshi Nakamura. Weighted finite state transducer based statistical dialog management. In *Automatic Speech Recognition & Understanding, 2009. ASRU 2009. IEEE Workshop on*, pages 490–495. IEEE, 2009.

[5] Zhifei Li and Jason Eisner. First- and second-order expectation semirings with applications to minimum-risk training on translation forests. In *Proceedings of the 2009 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 40–51, 2009.

[6] Mehryar Mohri. Generic epsilon-removal and input epsilon-normalization algorithms for weighted transducers. *International Journal of Foundations of Computer Science*, 13(01):129–143, 2002.

[7] Mehryar Mohri, Fernando Pereira, and Michael Riley. Speech recognition with weighted finite-state transducers. *Handbook on speech processing and speech communication, Part E: Speech recognition*, 2008.

[8] Graham Neubig, Shinsuke Mori, and Tatsuya Kawahara. A WFST-based log-linear framework for speaking-style transformation. In *Proceedings of the 10th Annual Conference of the International Speech Communication Association (InterSpeech)*, pages 1495–1498, 2009.

[9] Andrew Viterbi. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *IEEE transactions on Information Theory*, 13(2):260–269, 1967.