Lecture 7: Finite State Transducers, Language Modeling, and Speech Recognition Search

Lecturer: Mark Hasegawa-Johnson (jhasegaw@uiuc.edu)
TA: Sarah Borys (sborys@uiuc.edu)
Web Page: http://www.ifp.uiuc.edu/speech/courses/minicourse/

June 7, 2005

1 Finite State Machines

Download the AT&T FSM Library (http://www.research.att.com/sw/tools/fsm/) and the AT&T FSM-based speech recognition decoder (http://www.research.att.com/projects/mohri/dcd/). Read a few of the key manual pages (by calling "man" with a fully specified relative path to the file of interest, e.g.

```
man ~/src/dcd/dcd-2.0/ammodel.5
```

Key manual pages include `ammodel.5`, `drecog.1`, `fsmintro.1`, `fsm.5`, and `fsm.1`. Many of the ideas in this lecture are based on [5, 2].

1.1 Speech Understanding Grammar

1.1.1 Grammar Design

Consider the problem of developing a simple whois/whereis application for real people and real places. The system can accept two kinds of questions: "who is ¡person¿" and "where is ¡place¿." The goal of the grammar is to recognize any applicable query, and to output a segmentation that tags the beginning (the time stamp, in the speech waveform) of the query word (who or where), and the beginning of the person or place about whom the query is being submitted.

The basic structure of the application suggests that we need four sub-grammars: WHO, WHERE, PERSON, and PLACE. The WHO sub-grammar will recognize phrases such as "who is" and "who was;" upon recognizing either of these phrases, the grammar should output the symbol WHO in order to tell us that it has detected a WHO-phrase. The following grammar will do the trick; this could be stored in a file called `who.fst.txt`:

```
0 1 who WHO 0.0
1 2 is -  0.693
1 2 was - 0.693
2 0.0
```

This file specifies a grammar in which state 0 is the starting state, and state 2 is the ending state. In between, the grammar either observes the sequence "who is" or "who was;" in either case, it outputs the symbol "WHO" followed by the empty string (denoted "-"). The fifth column is the cost, which you may interpret as the negative log probability. The probability of the path "who is" is $e^{-0.0}e^{-0.693}e^{-0.0} = 0.5$. If cost is not specified, it is assumed to be 0.0.

In order to do anything with this grammar, it is necessary to compile it. The FSM tools don't store any strings internally, so it is necessary to create two vocabulary files. The output vocabulary contains the phrase-recognition symbols "WHO, WHERE, PERSON, PLACE." For example, the following file could be stored in `qtop.voc`:

```
- 0
WHO 1
WHERE 2
PERSON 3
PLACE 4
```

The first line in the vocabulary file is always reserved for the empty string, denoted "-". The input vocabulary contains all of the words that will be used in this application, e.g.,

```
- 0
who 1
where 2
is 3
was 4
Tom 5
Charles 6
Gregory 7
Martel 8
the 9
Great 10
Bald 11
Aachen 12
Rome 13
Urbana 14
```

Given the vocabulary files, the FSM can be compiled into binary format using the command

```
fsmcompile -t -i qwords.voc -o qtop.voc who.fst.txt > who.fst
```

The `-t` option specifies that this is a transducer, meaning that each edge in the FSM has both an input string and an output string—in this case, the input string and output string are identical. The content of the binary file can be printed again using `fsmprint -i qwords.voc -o qtop.voc who.fst`. Notice that the vocabulary must be specified, because it is not stored in the binary file. Finally, in order to get summary statistics about an FSM (e.g., number of states, number of edges), use `fsminfo -n who.fst`.

The subgrammars `where.fst.txt`, `person.fst.txt`, and `place.fst.txt` can be written similarly. For example, `person.fst.txt` could contain

```
0 1 Charles PERSON
0 1 Gregory PERSON
0 2 Tom PERSON
1 2 Martel -
1 3 the -
3 2 Great -
3 2 Bald -
2
```

This version will recognize the following PERSONs: Tom, Charles Martel, Charles the Bald, Charles the Great, Gregory the Great, Gregory Martel, and Gregory the Bald. The system has over-generalized; "Gregory the Bald" is not the name of any historical figure. Fictional PERSONs can be eliminated with a more complicated FSM, if it turns out to be important to do so.

Our goal is to recognize either a "who is PERSON" query or a "where is PLACE" query, but not queries of the form "where is PERSON." In other words, we want to combine the subgrammars according to the regular expression `(WHO PERSON|WHERE PLACE)`. This can be done by concatenating then unioning the various subgrammars:

```
fsmconcat who.fst person.fst | fsmrmepsilon > whoperson.fst
fsmconcat where.fst place.fst | fsmrmepsilon > whereplace.fst
fsmunion whoperson.fst whereplace.fst > query.fst
```

After the commands above, the whole grammar can be printed using

```
fsmprint -i qwords.voc -o qtop.voc query.fst
```

resulting in:

```
0       1       who     WHO
0       6       where   WHERE
1       2       is      -       0.693000019
1       2       was     -       0.693000019
2       5       Tom     PERSON
2       3       Charles PERSON
2       3       Gregory PERSON
3       5       Martel  -
3       4       the     -
4       5       Great   -
4       5       Bald    -
5
6       7       is      -       0.693000019
6       7       was     -       0.693000019
7       8       Aachen  PLACE
7       8       Rome    PLACE
7       8       Urbana  PLACE
8
```

1.1.2 Query Understanding

The task of "understanding" an input sentence is equivalent to the task of "composing" two finite state machines. The composition of two FSTs is an FSM that contains all paths that correspond to both a legal output string of the first FSM, and a legal input string of the second FSM.

For example, suppose that automatic speech recognition has resulted in some ambiguity. The first word might be either "where" or "who," with acoustic costs (negative log probability of the HMM) given by 0.32 and 0.5. Likewise, the last word might be either "Tom" or "Rome." Store this word lattice in the file `test.fst.txt`, and compile it to `test.fst`:

```
0 1 where 0.32
0 1 who 0.5
1 2 is 1.856
2 3 Rome 0.33
2 3 Tom 0.1
3
```

In order to perform speech understanding, type

```
fsmcompose test.fst query.fst | fsmrmepsilon > output.fst
```

We can see all possible speech understanding outputs by typing

```
fsmprint -i qwords.voc -o qtop.voc output.fst
```

resulting in

```
0       1       where   WHERE   0.319999993
0       4       who     WHO     0.5
1       2       is      -       2.54900002
2       3       Rome    PLACE   0.330000013
3
4       5       is      -       2.54900002
5       6       Tom     PERSON  0.100000001
6
```

In order to see the most probable speech understanding output, type

```
fsmbestpath output.fst | fsmprint -i qwords.voc -o qtop.voc
```

resulting in

```
0       1       who     WHO     0.5
1       2       is      -       2.54900002
2       3       Tom     PERSON  0.100000001
3
```

Therefore the most probable speech understanding output will interpret this sentence as a "WHO PERSON" query, specifically "Who is Tom?"

1.2 N-Grams

An N-gram defines the probability of word w_i to depend only on the $N-1$ previous words:

$$p(w_i|\text{history}) \equiv p(w_i|w_{i-N+1}, \cdots, w_{i-1}) \tag{1}$$

$$\sum_w p(w|w_{i-N+1}, \cdots, w_{i-1}) = 1 \tag{2}$$

With no further simplifications, an N-gram over a vocabulary containing V words defines a finite state machine with V^{N-1} states, and V^N edges [3]. For example, consider a trigram defined over the following two-word vocabulary, contained in the file `words.voc`:

```
- 0
spam 1
eggs 2
```

There may be as many as four states, labeled by the corresponding bigram sequences; the corresponding vocabulary file might be contained in `states.voc`:

```
spam_spam 0
spam_eggs 1
eggs_spam 2
eggs_eggs 3
```

The transitions among these states are labeled with the negative log probabilities of the N-gram model, so the FSM file `trigram.fst.txt` might look like:

```
eggs_spam spam_spam spam 0.105
eggs_spam spam_eggs eggs 2.302
spam_spam spam_eggs eggs 0.693
spam_spam spam_spam spam 0.693
spam_eggs eggs_eggs eggs 1.609
spam_eggs eggs_spam spam 0.223
eggs_eggs eggs_eggs eggs 1.203
eggs_eggs eggs_spam spam 0.357
spam_spam
spam_eggs
eggs_spam
eggs_eggs
```

This file is unlike those in the previous section, because it uses words for the states, instead of numbers. It can be compiled using

```
fsmcompile -S states.voc -i words.voc trigram.fst.txt > trigram.fst
```

The first line of `trigram.fst.txt` specifies that if the most recent two words are "eggs_spam," and we observe the word "spam," then the history is updated to "spam_spam." The probability of this particular history transition is $p(\text{spam}|\text{eggs}, \text{spam}) = e^{-0.105} = 0.9$. The last four lines in the file specify that every state is a valid end state.

Obviously, a fully-specified N-gram model is not practical if either V or N is large (in speech recognition, V is typically 20000-100000; recent systems use $N \approx 4$). For $N > 2$, it is necessary to depend on backoff [6]. For example, suppose that we are using a backoff approximation under which it is possible to write

$$p(w_i|w_{i-2}, w_{i-1}) = p_3(w_i|w_{i-2}, w_{i-1}) + b_2(w_{i-2}, w_{i-1})p_2(w_i|w_{i-1}) + b_1(w_{i-1})p_1(w_i) \tag{3}$$

This can be accomplished by inserting backoff states, `B2` and `B1`. The transition costs are

$$\begin{aligned}
c(w_{i-2}w_{i-1} \rightarrow w_{i-1}w_i) &= -\log p_3(w_i|w_{i-2}, w_{i-1}) && (4)\\
c(w_{i-2}w_{i-1} \rightarrow B2) &= -\log b_2(w_{i-2}, w_{i-1}) && (5)\\
c(B2 \rightarrow w_{i-1}w_i) &= -\log p_2(w_i|w_{i-1}) && (6)\\
c(w_{i-2}w_{i-1} \rightarrow B1) &= -\log b_1(w_{i-1}) && (7)\\
c(B1 \rightarrow w_i) &= -\log p_1(w_i) && (8)\\
& && (9)
\end{aligned}$$

It is possible to combine variable-length N-grams together with a speech understanding task grammar [7].

1.3 Dictionary

Finite state transducers give us a particularly flexible way of representing a dictionary. Each word in the dictionary may have one pronunciation or many. For example, the words "these" and "those" has only one common pronunciation, given in the files `those.fst.txt`: and `those.fst.txt`:

```
0 1 dh -
1 2 ow -
2 3 z -
3 4 - those
4
```

```
0 1 dh -
1 2 ow -
2 3 z -
3 4 - these
4
```

where the input vocabulary is `phones.voc`:

```
- 0
dh 1
iy 2
ow 3
z 4
ax 5
```

The word "the," on the other hand, has two common pronunciations, given in the file `the.fst.txt`:

```
0 1 dh -
1 2 iy -
1 2 ax -
2 3 - the
3
```

All of the different entries in the dictionary may be combined using the `fsmunion` command:

```
fsmunion dict/*.fst | fsmrmepsilon > dict.fst
```

By default, the resulting dictionary contains as many edges as the sum of all of the input lexical entries. That is very wasteful! It is useful to reduce the size of the dictionary by recoding it in the form of a tree — i.e., all words that start with the phoneme /dh/ should share their first transition. By combining entries in this way, we reduce both storage space and search complexity.

A dictionary can be automatically compiled into a tree as follows. First, encode the transducer into the form of a finite state acceptor:

```
fsmencode -cl dict.fst key.fst > dict_enc.fst
```

Then it must be "determinized," then "minimized:"

```
fsmdeterminize dict_enc.fst | fsmminimize > dict_min_enc.fst
```

Then decode it, bringing back the edge costs and output labels:

```
fsmencode -dcl dict_min_enc.fst key.fst > dict_min.fst
```

If all works as planned, the dictionary has been re-encoded into the form of a tree:

```
0       1       dh      -
1       2       iy      -
1       3       ax      -
1       4       ow      -
2       5       -       the
2       6       z       -
3       5       -       the
4       7       z       -
5
6       5       -       these
7       5       -       those
```

The dictionary shown here observes a sequence of phonemes. After observing the sequence of phonemes corresponding to a particular word, it outputs the word label. Suppose we wanted to output the word label at the beginning of the observed word, rather than at the end. One way to do this would be: (1) create lexical entries that start with the null→word transition, rather than ending with it, (2) fsmunion them, (3) reverse the resulting FSM (`fsmreverse`), (4) perform determinization and minimizing as shown above, (5) reverse the FSM again, in order to get an "inverse tree."

In order to use this dictionary in speech recognition, it is necessary to allow word sequences, not just words. Word sequences are allowed by computing "Kleene closure" of the dictionary:

```
fsmclosure dict_min.fst | fsmrmepsilon > dict_clo.fst
```

The transducer `dict_clo.fst` allows any sequence of words, with equal probability. It may be useful to limit the allowable word sequence, or to impose word sequence probabilities, by composing `dict_clo.fst` with a grammar (N-gram or speech understanding grammar). Suppose `gram.fst` contains a grammar FSM, i.e., an FSM that inputs a word string, and outputs a syntactic phrase label, e.g.,

```
0 1 the -
0 1 these -
0 1 those -
1 2 dogs -
2 3 - NP
3
```

It is possible to compose `dict_clo.fst` and `gram.fst` in order to create an FST that inputs a phone string, and outputs a syntactic phrase label:

```
fsmcompose dict_clo.fst gram.fst > recog.fst
```

resulting in

```
0       1       dh      -
1       2       iy      -
1       9       ax      -
1       10      ow      -
2       3       z       -
2       4       d       -
3       4       d       -
4       5       ao      -
5       6       g       -
6       7       z       -
7       8       -       NP
8
9       4       d       -
10      11      z       -
11      4       d       -
```

1.4 Triphones

Triphone expansion is computed using a finite state transducer that takes phone sequences as input, and computes triphone sequences as output. This FST looks very much like the N-gram FST. States are listed in the form of biphones; with P different phones, there are P^2 different states. Given a biphone, if another monophone is observed, we output a triphone:

```
d_ao ao_g g d-ao+g
ao_g g_z z ao-g+z
...
```

Notice that there is a delay between input and output: when the /g/ is input, we output the triphone corresponding to the /ao/. This requires some extra book-keeping at word boundaries.

The FST as defined above takes monophone inputs, and outputs triphones. In order to compose this with the dictionary, we need to have monophone *outputs*. The input-output relationship can be switched using `fsminverse` (alternatively, we could use perl to just swap the order of the third and fourth columns).

1.5 States and Mixtures

In automatic speech recognition, a triphone is composed of states, and a state is composed of Gaussian mixture elements. Each of these can be handled by the acoustic model definition (see `ammodel.5`), but for extra flexibility, they can also be handled directly by the finite state transducers. For example, the transducer mapping from triphones to states might look like

```
# d-ao+g_fst1 d-ao+g_hmm1 -
d-ao+g_fst1 d-ao+g_fst2 d-ao+g_hmm2 -
d-ao+g_fst2 # d-ao+g_hmm3 d-ao+g
...
```

The transducer shown above maps from an HMM state sequence (`d-ao+g_hmm1`, `d-ao+g_hmm2`, ...) to a triphone (`d-ao+g`). The FST states are numbered as `d-ao+g_fst1`, in order to avoid confusion between the FST states and the HMM states.

Each triphone starts and ends with a transition from or to the inter-phone symbol, #. The resulting transducer can be turned into a loop using `fsmclosure`, and then composed with the dictionary and the grammar.

If desired, it is even possible to create a map between HMM states and individual Gaussian mixture components. This mapping can be completely context-dependent, so all transitions in the finite state transducer can be self-loop transitions of the zero state. For example, suppose that HMM state d-ao+g_hmm1 is equally likely to draw from two different Gaussian mixtures. In the global recognizer, these mixtures are mixture #4980, and mixture #4981. The following transducer specifies that the probability of each of those two mixtures is 0.5:

```
0 0  mixture4980 d-ao+g_hmm1 0.693
0 0  mixture4981 d-ao+g_hmm1 0.693
```

If mixture elements are shared among different states, then the methods in this section make sense. If mixture elements are not shared, then there is no good reason to explicitly compose the mixtures transducer, the states transducer, and the higher-level transducers. Specifically, there is no extra redundancy that can be removed using the `fsmdeterminize` and `fsmminimize` commands. Since there is no extra redundancy that we can remove, it is better (unless mixtures are shared between states) to keep the mixture and state definitions separate from the higher-level transducers until the last minute. For this reason, `drecog` allows mixtures and states to be specified using the `ammodel` file format.

2 Speech Recognition Search

The AT&T decoder program, `drecog`, does on-the-fly composition of two search graphs: (1) an FST search graph (usually created by composing the task grammar or N-gram, the dictionary, and the monophone-to-triphone FST), and (2) a set of acoustic models. There should be one acoustic model per state in the FST.

The acoustic models are defined using the `ammodel.5` acoustic model file format. This file contains similar information to the HTK master model file (MMF), but not quite the same. It is possible to use `perl` or `python` to read in a text-format HTK MMF, and to write out a text-format `ammodel` file, or vice versa. A text-format `ammodel` file must be compiled into a binary file (using `amcompile`) before it can be used in speech recognition.

An `ammodel` parameters file is just a list, specifying the component filenames. There are several component files:

1. HMM file. The first column is an index specifying the ID number of the HMM. Remaining columns specify the ID numbers of states composing the HMM.

2. Durations file. The first column is an index specifying the ID number of the HMM. Remaining columns specify the mean and variance of the HMM duration. `drecog` implements an explicit-duration HMM, using a gamma PDF [4]. The shape of the gamma PDF is completely specified by knowledge of its mean and variance. `drecog` can also implement explicit duration PDF for each state individually, rather than for the entire HMM.

3. States file. The first column is an index specifying the ID number of the state. Remaining columns specify the ID numbers of the mixture elements composing the state, and the mixture weight of each.

4. Means file. The first column is an index specifying the ID number of the mixture. Remaining columns specify the mean vector.

5. Variances file. The first column is an index specifying the ID number of the mixture. Remaining columns specify the variance vector (main diagonal of the covariance matrix).

There are two fundamental differences between the `ammodel` file and the HTK MMF. First, `ammodel` by default uses explicit duration PDFs (to turn off this capability, you need to set the selfloop flag); HTK does not, but extensions available at [1] will implement explicit duration PDFs for HTK. Second, `ammodel` can only implement strictly left-to-right HMMs, with no skipped states. If you would like to use a more complicated model structure, then you need to represent the model structure explicitly in the FST; the `ammodel` file would then create trivial "HMMs" in which each state is its own HMM.

References

[1] Ken Chen and Mark Hasegawa-Johnson. HDK: Extensions of HTK for explicit-duration hidden markov modeling. Software available at http://www.ifp.uiuc.edu/speech/software, 2003.

[2] Eric Fosler-Lussier. FSM tutorial. Lecture Notes for the 2004 ACL Summer Workshop on Speech and Language, 2004.

[3] Frederick Jelinek. *Statistical Methods for Speech Recognition*. MIT Press, Cambridge, MA, 1997.

[4] S. E. Levinson. Continuously variable duration hidden Markov models for automatic speech recognition. *Computer Speech and Language*, 1:29–45, 1986.

[5] Mehryar Mohri, Fernando Pereira, and Michael Riley. Weighted finite-state transducers in speech recognition. *Computer Speech and Language*, 16:69–88, 2002.

[6] Guiseppe Riccardi, Roberto Pieraccini, and Enrico Bocchieri. Stochastic automata for language modeling. *Computer Speech and Language*, 10:265–293, 1996.

[7] Manhung Siu and Mari Ostendorf. Variable n-grams and extensions for conversational speech language modeling. *IEEE Trans. Speech and Audio Processing*, 8(1):63–75, 2000.