MPhil in Advanced Computer Science
Module L102: Statistical Machine Translation

Practical Handout 2
Weighted Finite-State Transducers for Language Processing

1 Introduction

This second practical focuses on the use of Weighted Finite-State Transducers (WFSTs) for language processing. For this purpose, we will use the OpenFst Library, an open-source project developed by contributors from Google Research and NYU's Courant Institute. If you have interest, much more information about OpenFST, including examples, tutorials and software, can be found at `www.openfst.org`.
Important: Run the following command in order to setup the variables needed to run the practical.

```
source /usr/groups/acs-software/L102/setup.sh
```

1.1 Preliminary tutorial

This is a brief tutorial to get acquainted with OpenFST (how to create fsts, print them draw them, etc...), based on material from the project's website.

Example FST

Figure 1 depicts an example finite state transducer:

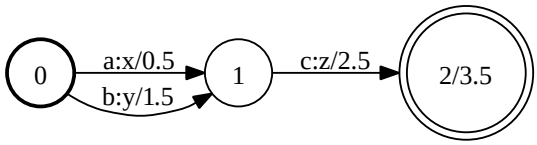

Figure 1: Example FST

The initial state is label 0. There can only be one initial state. The final state is 2 with final weight of 3.5. Any state with non-infinite final weight is a final state, and is drawn with a double circle. There is an arc (or transition) from state 0 to 1 with input label 'a', output label 'x', and weight 0.5. This FST transduces, for instance, the string 'ac' to 'xz' with weight 6.5 (the sum of the arc and final weights).

Creating FSTs Using Text Files from the Shell

We can create the text FST file for our example as follows:

```
# arc format: src dest ilabel olabel [weight]
# final state format: state [weight]
# lines may occur in any order except initial state must be first line
# unspecified weights default to 0.0 (for the library-default Weight type)
$ cat >text.fst <<EOF
0 1 a x .5
0 1 b y 1.5
1 2 c z 2.5
2 3.5
EOF
```

The internal representation of an arc label is an integer. We must provide the mapping from symbols to integers explicitly with a symbol table file, in the following format:

```
$ cat >isyms.txt <<EOF
<eps> 0
a 1
b 2
c 3
EOF

$ cat >osyms.txt <<EOF
<eps> 0
x 1
y 2
z 3
EOF
```

You may use any string for a label; you may use any non-negative integer for a label ID. The zero label ID is reserved for the epsilon label, which is the empty string. We have included 0 in our table, even though it is not used in our example. Since subsequent FST operations might add epsilons, it is good practice to include a symbol for it.

This text FST must be converted into a binary FST file before it can be used by the OpenFst library.

```
# Creates binary Fst from text file.
# The symbolic labels will be converted into integers using the symbol table files.
$ fstcompile --isymbols=isyms.txt --osymbols=osyms.txt text.fst binary.fst
```

Accessing FSTs: Printing, Drawing, Summarizing

As FSTs are encoded in binary files, a set of special commands must be used to access the information they contain. The following command will **print out** an FST in text format:

```
# Print FST using symbol table files.
$ fstprint --isymbols=isyms.txt --osymbols=osyms.txt binary.fst text.fst
```

If the symbol table files are omitted, the FST will be printed with numeric labels. The following command will **draw** an FST using Graphviz dot format:

```
# Draw FST using symbol table files and Graphviz dot:
$ fstdraw --isymbols=isyms.txt --osymbols=osyms.txt binary.fst binary.dot
$ dot -Tps binary.dot >binary.ps
```

Summary information about an FST can be obtained with:

```
$ fstinfo binary.fst
fst type                          vector
arc type                          standard
input symbol table                isyms.txt
output symbol table               osyms.txt
# of states                       3
# of arcs                         3
initial state                     0
# of final states                 1
# of input/output epsilons        0
# of input epsilons               0
# of output epsilons              0
# of accessible states            3
# of coaccessible states          3
# of connected states             3
# of connected components         1
# of strongly conn components     3
...
```

FST Operations

The FST operations can be invoked from shell-level commands, which typically read one or more input binary FST files and then write an output binary FST file. If the output file is omitted, standard output is used. If the input file is also omitted (unary case) or is "-", then standard input is used. Specifically, they have the form:

- Unary Operations

```
fstunaryop in.fst out.fst
fstunaryop < in.fst > out.fst
```

- Binary Operations

```
fstbinaryop in1.fst in2.fst out.fst
fstbinaryop - in2.fst < in1.fst > out.fst
```

Available include (among others):

- concatenating and unioning two FSTs (`fstconcat, fstunion`)
- composing two FSTs (`fstcompose`)
- obtaining the intersection or the difference between two FSTs (`fstintersect, fstdifference`)
- finding equivalent FSTs with less states/arcs by removing empty ϵ-arcs, determinizing and minimizing (`fstrmepsilon, fstdeterminize, fstminimize`)
- computing shortest distance, N-best paths and pruning (`fstshortestdistance, fstshortestpath, fstprune`)
- creating an acceptor from a transducer by projecting input or output labels (`fstproject`)
- sorting arcs (`fstarcsort, fsttopsort`)

- reverse the direction (`fstreverse`) or swap input-output symbols (`fstinvert`)
- connect final states with the initial state via epsilon arcs so that (`fstclosure`)
- other: `fstpush, fstrelabel, fstreplace, etc`

Details of each operation can be found at `www.openfst.org`.

Example: Obtaining N-best Lists and Composing two FSTs

The following command will return the **N shortest paths** in the FST, ranked by cost:

```
# Print the N shortest paths (-n 10), showing their cost (-c)
# and the strings they encode in the input labels (-1 -i isym.txt)
$ cat binary.fst | fstprintstrings -n 10 -c -1 -i isyms.txt
a c  6.5
b c  7.5

# Print the N shortest paths (-n 10), showing their cost (-c)
# and the strings they encode in the output labels (-2 -i osym.txt)
$ cat binary.fst | fstprintstrings -n 10 -c -2 -i osyms.txt
x z  6.5
y z  7.5
```

In this example, only two possible paths exist in the transducer, representing two distinct hypotheses[1]. However, an FST can contain multiple paths representing the same string, in which case repeated strings would be listed, each with its associated cost. This can be avoided using the '-u' switch, so that only the least cost instance of each string is shown.
One of the most useful finite-state operations is composition, which produces the relational composition of two transductions (see lecture notes 7 for more details). It can be used, for example, to apply a transduction to some input according to a particular model. Example:

```
# Create an input transducer in text format:
# When input and output labels are equal, it is an acceptor
$ cat > input.txt <<EOF
0 1 a a
1 2 b b
2 3 c c
3 4 a a 0.4
2 5 a a 0.2
5 6 a a
2 1.0
3
4
6 0.1
EOF

# Compile as binary file (using same input/output table):
$ fstcompile --isymbols=isyms.txt --osymbols=isyms.txt input.txt input.fst
```

[1]Please note that this command returns one empty line after all valid strings. This line should not be counted when calculating how many paths or strings are contained in an FST.

```
# Create a weighted transducer in text format
$ cat > model.txt <<EOF
0 1 a x 0.2
0 1 b y 1.2
0 1 c z 3
1 2 b y 0.5
1 4 b z 0.1
2 3 c x
2 4 a <eps> 0.5
4 4 a x 0.1
3
4
EOF

# Compile as binary file (using same input/output table):
$ fstcompile --isymbols=isyms.txt --osymbols=osyms.txt model.txt model.fst
```

Composition is obtained as follows:

```
# Creates the composed FST.
$ fstcompose input.fst model.fst comp.fst

# Just keeps the output label
$ fstproject --project_output comp.fst result.fst
```

Note that the FSTs must be sorted along the dimensions they will be composed (in fact, only one needs to be so sorted). In case your FST is not sorted, you can sort it by doing the following:

```
# Compose with a previous arc sorting step:
$ fstarcsort --sort_type=olabel input.fst input_sorted.fst
$ fstarcsort --sort_type=ilabel model.fst model_sorted.fst
$ fstcompose input_sorted.fst model_sorted.fst comp.fst

# Do it all in a single command line.
$ fstarcsort --sort_type=ilabel model.fst | fstcompose input.fst - |\
    fstproject --project_output result.fst
```

Preliminary Question: Draw the input and model FSTs generated above. Then compose them and project the result to keep only output symbols, and draw the resulting FST (result.fst). Note that when drawing acceptors (such as 'input'), you can optionally use '`fstdraw --acceptor`' to avoid showing the redundant output symbols (only showing input ones). Answer the following questions:

(i) How many alternative paths exist in the input FST? How many distinct strings do these encode? Which ones of these strings can be accepted by the model transducer?

(ii) How many alternative paths exist in the result FST? How many distinct strings do these encode?

(iii) Now remove epsilon arcs, determinize and minimize the result FST. How many alternative paths exist now? How many distinct strings do these encode? Why?

(iv) How many alternative strings are accepted by the model FST? Refer to the drawing to support your answer.

2 Practical Exercise

The following sections describe a set of exercises to be completed in this practical. In your report you should state how you have achieved a solution, including description or drawings of FSTs, and listings of FST operations. Please seek help from the demonstrator should you encounter problems with this part of the practical. All relevant files can be found in:

```
DIR=/usr/groups/acs-software/L102/practical-2/files
```

1. Given the alphabet $L = \{a, b, ...z, A, B, ...Z, \langle space \rangle\}$, for which a symbol table file can be found in `$DIR/table1.txt`, create an automaton that:

   (a) Accepts a letter in L (including space).

   (b) Accepts a single space.

   (c) Accepts a capitalized word (where a word is a string of letters in L excluding space and a capitalized word has its initial letter uppercase and remaining letters lowercase)

   (d) Accepts a word containing the letter a.

2. Using the automata in Question 1 as the building blocks, use appropriate FST operations on them to create an automaton that:

   (a) Accepts zero or more capitalized words followed by spaces.

   (b) Accepts a word beginning or ending in a capitalized letter.

   (c) Accepts a word that is capitalized and contains the letter a.

   (d) Accepts a word that is capitalized or does not contain an a.

   (e) Accepts a word that is capitalized or does not contain an a without using `fstunion`.

   For each case, give the number of states and arcs before and after applying epsilon removal, determinization and minimization to the resulting automata.

3. Given the alphabet $L = \{0, 1, ...9\}$, create a transducer that maps numbers (in the range 00000 to 99999) represented as strings of 5 digits to their English read form, e.g.

   0 0 0 0 1 → one

   0 0 8 0 7 → eight hundred seven

   1 3 2 5 5 → thirteen thousand two hundred fifty five

   A complete symbol table file has already been created for you in `$DIR/table3.txt`. Please try to define basic transducers as building blocks and use as many FST operations as possible to create the final transducer.

4. Given the alphabet $L = \{a, b, ...z, \langle space \rangle, ., ,\}$ (includes period and comma), for which a symbol table file can be found in `$DIR/table4.txt`:

   (a) Create a transducer that implements the *rot13* cipher: $a \rightarrow n$, $b \rightarrow o$, ... , $m \rightarrow z$, $n \rightarrow a$, $o \rightarrow b$, ... , $z \rightarrow m$.

   (b) Encode and decode the message '`my secret message`' (assume $\langle space \rangle \rightarrow \langle space \rangle$, . → . and , →,).

(c) We wish to decipher the message that can be found in the file

`$DIR/4.encoded1.fst`

knowing that in order to do so, we must simultaneously allow two transductions, namely *rot13* and *rot16* ($a \rightarrow q$, $b \rightarrow r$, ...), so that a can either encode an original n or q. Build a suitable decoding transducer, apply it to the encoded message and examine the resulting FST (after projecting onto the output symbols). How many states and arcs does it contain? How many states and arcs after removing epsilons, determinizing and minimizing it? How many distinct strings does it represent?

(d) We now know that the original text belongs to Charles Dickens' *David Copperfield* novel. Accordingly, a language model has been implemented as an unweighted automaton that accepts any sentence contained in this novel. This can be found in

`$DIR/4.lm.fst`

Compose your resulting FST from question (c) with this language model. What was the original text?

(e) Similarly, a second text belonging to the same novel has been encoded in the file

`$DIR/4.encoded2.fst`

by applying the *rot13* cipher and allowing some pairs of consecutive letters to be swapped (excluding $\langle space \rangle$ or punctuation). Build a suitable decoding transducer[2], apply it to the encoded message and compose your answer with the language model. What was the original text?

[2] You will need to allow any possible pair of letters to be swapped in the input.