WFST解码图构建

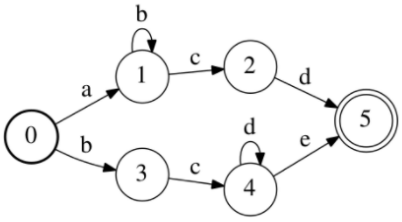
当有了声学模型，语言模型以及lexicon之后，下一步就是对输入的语音做识别解码的问题了，WFST（weighted finite-state transducer）由Mohri在2008年提出，是目前大词汇量连续语音识别（LVSSR）系统最常用也最高效的解码算法。

1. 有限自动机(Finite-Automata)

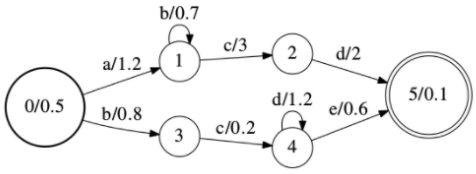
1.1 直观理解

WFST是一种有限自动机(FA)。一个有限自动机有一个有限的状态集合以及状态之间的跳转，其中每个跳转至少有一个标签(label)。最基本的FA是有限状态接收机(finite state acceptor/FSA)。给定一个输入符号序列，FSA返回"接受"或者"不接受"，它的判断条件是：如果存在一条从初始状态到终止状态的路径，使得路径上的标签序列正好等于输入符号序列，那么就是"接受"，否则就是"不接受"。

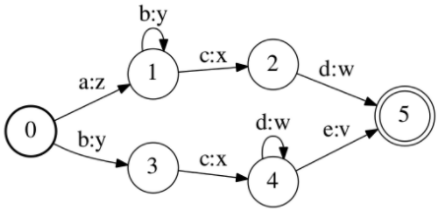
下图(a)是一个FSA的例子，图中的节点和边分布代表状态和状态之间的跳转。比如这个FSA接受在符号序列"a,b,c,d"，因为状态跳转序列"0,1,1,2,5"是从初始状态到最终状态的路径，它的边对应的符号序列正好是"a,b,c,d"。但是它不能接受"a,b,d"，因为无法找到满足条件的状态序列。因此，一个FSA代表了它能接受的符号序列的集合。符号序列也被成为字符串。如果没有特殊说明，一般用加粗的圆圈表示初始状态，用两个圈表示终止状态。



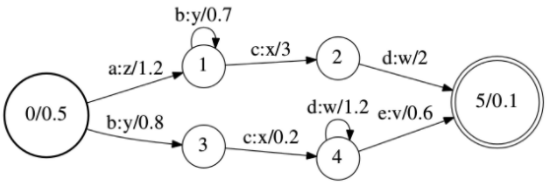
(a) Finite-State Acceptor (FSA)



(c) Weighted Finite-State Acceptor (WFSA)



(b) Finite-State Transducer (FST)



(d) Weighted Finite-State Transducer (WFST)

除了FSA之外，另外有好几种FSA的扩展，例如有限状态转换机(Finite-State Transducer/FST)，加权有限状态接收机(Weighted Finite-State Acceptor/WFSA 和加权有限状态转换机(Weighted Finite-State Transducer/WFST)。这些FA继承了FSA的基本特性，但是它们的输出不只是一个二值的"接受/不接受"，而是会输出另一个符号序列(FST)，一个权值(WFSA)或者同时输出一个新的符号序列和权值(WFST)。

FST的每个跳转(边)上都有一个输出符号，因此它的边上是一个输入标签和输出标签的pair。上图(b)中就是一个FST的例子。在图中用"输入符号:输出符号"来表示。通过这个扩展，FST描述了把一个输入符号序列转换为另一个输出符号的转换规则。例子中的FST可以把输入符号序列"a,b,c,d"转换成"z,y,x,w"。

WFSA的每个跳转除了输入符号还有一个weight，此外初始状态和终止状态也有对应的初始weight和终止weight。weight通常代表跳转的概率或者代价，不同的路径上的weight会通过"乘法"来累计。因此，WFSA提供了一种比较不同路径weight的度量。 如果weight代表跳转的概率且从每个状态输出的转移概率之和为1，则称这个WFSA是随机的(stochastic) ，上图(c)是一个WFSA的例子。在上图中，每条边

为”输入标签/weight”，并且初始状态表示为”初始状态ID/weight”；终止状态表示为”终止状态ID/weight”。对于这个WFSA，它接受序列”a,b,c,d”并且累计的weight是0.252：路径是0,1,1,2,5，weight为0.5 × 1.2 × 0.7 × 3 × 2 × 0.1。

WFST的跳转上同时包括输出标签和weight，因此WFST可以认为是FST和WFSA的组合。上图(d)是一个WFST的例子，图中的边上为”输入符号:输出符号/weight”。初始和终止的weight也在对应的状态上标识出来。这这个WFST里，它可以把输入符号序列”a,b,c,d”变成”z,y,x,w”，并且weight是0.252。

1.2 定义

一个WFST由下面的8元组$(\Sigma, \Delta, Q, I, F, E, \lambda, \rho)$ 来定义：

- Σ 是一个有限的输入符号集合
- Δ 是一个有限的输出符号集合
- Q 是一个有限的状态集合
- $I \subseteq Q$ 是初始状态集合
- $F \subseteq Q$ 是终止状态集合
- $E \subseteq Q \times (\Sigma \cup \epsilon) \times (\Delta \cup \epsilon) \times \mathbb{K} \times Q$ 是状态转移的集合。
- $\lambda : I \to \mathbb{K}$ 是初始状态weight的函数
- $\rho : F \to \mathbb{K}$ 是终止状态weight的函数

ϵ是一个特殊的(输入和输出)符号，它代表空，没有输入/输出。上图(d)的WFST即可定义为：

符号	图例
Σ	$\{a, b, c, d, e\}$
Δ	$\{v, x, y, w, z\}$
Q	$\{0, 1, 2, 3, 4, 5\}$
I	$\{0\}$
F	$\{5\}$
E	$\{(0, a, z, 1.2, 1), (0, b, y, 0.8, 3), (1, b, y, 0.7, 1), (1, c, x, 3, 2), (2, d, w, 2, 5), (3, c, x, 0.2, 4), (4, d, w, 1.2, 4), (4, e, v, 0.6, 5)\}$
λ	$\lambda(0) = 0.5$
ρ	$\rho(5) = 0.1$

E 中的每一个跳转为(源状态, 输入符号, 输出符号, weight, 目标状态)。其它的FA，包括FSA、FST和WFSA，都可以看成WFST的特殊情况。

1.2.1 半环理论

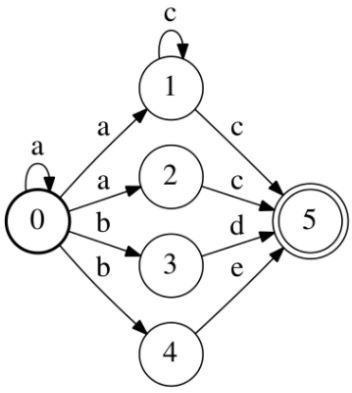
对于加权的(weighted) FA，weight以及其上的二元运算”加法”和”乘法”需要更加形式化的定义用于使得FA和相关的算法更加一般化。在理论上，WFST的weight和其上的运算是使用半环来定义的，这是一种抽象代数的代数结构。这意味着任何类型的weight都可以用FA的算法处理，前提是需要用这个weight的集合来定义一个半环。

1.3 基本性质及运算

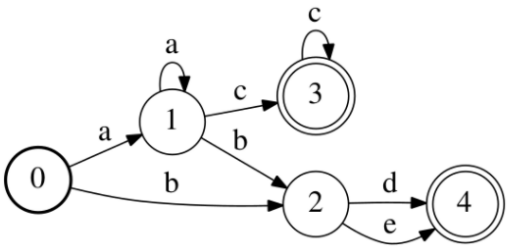
1.3.1 确定化(Determinization)

FA的一个重要性质是它是确定的(deterministic)还是非确定的(non-deterministic)，并且所有的FSA都是可以确定化的。一个确定的FA(DFA)只有一个初始状态，并且对于每个状态如果给定了一个输入符号，最多有一条边。因此如果某个输入符号序列是被它接受的，也只有一条对应的从初始状态到终止状态的路径。DFA的优点是给定输入符号序列，判断它是否被接受的计算是比较快的(相对于非确定的NFA)。如果使用二分查找来获得一个输入符号的边的话，DFA的计算复杂度是$O(Llog_2\hat{D})$。这里L是输入符号序列的长度，是从一个状态跳出的边的最大数量。这个计算复杂度是和长度L成线性比例关系的，但是和$\hat{D}$ 呈对数的关系，也就是说$\hat{D}$的成倍增长不会引起复杂度的成倍增长。

而NFA给定一个状态和一个输入符号，它可以有多条跳转的边，因此，我们需要考虑多种可能的路径。虽然NFA的计算复杂度依赖于它的结构，但在最坏的情况下的复杂度是$O(L\times|Q|\times|E|)$。不过存在一个标准的确定化(determinization)算法把NFA转换成与之等价的DFA。确定化之后，NFA的功能可以由与之等价的计算量更少的DFA来实现。下图的NFA和DFA是等价的。注意虽然所有的FSA都可以确定化，但是对于其它的FA，比如FST、WFSA和WFST不一定存在与之等价的确定化的FA。kaldi中的消歧符号就是为了保证生成的WFST是可确定化的。



(a) NFA

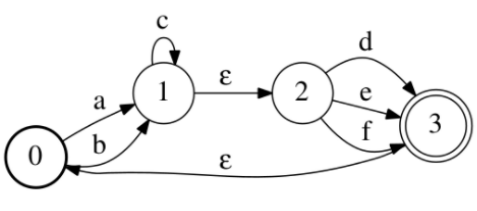


(b) DFA

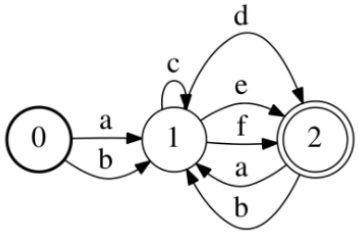
1.3.2 ε-消除

FA的另外一个重要属性就是它的输入标签里是否有特殊的ε。输入符号为ε的跳转叫做ε-跳转，这个状态的跳转不需要任何输入符号就可以进行。下图(a)是一个包含ε-跳转的FSA。这个FSA首先在状态0通过输入符号"a"或者"b"跳转到状态1。因为状态1和2之间有一个ε-跳转，因此它可以在不读入任何输入符号的条件下跳转到状态2，当然在没有任何输入的时候它也可以还是呆在状态1。因此这FSA可以同时呆在状态1和2。有ε-跳转的FSA是非确定的，我们把它叫做ε-NFA。

现已存在一个经典算法(ε-消除算法)把一个ε-NFA转换成与之等价的没有ε-跳转的NFA。下图(b)是与(a)等价的没有ε-跳转的NFA。



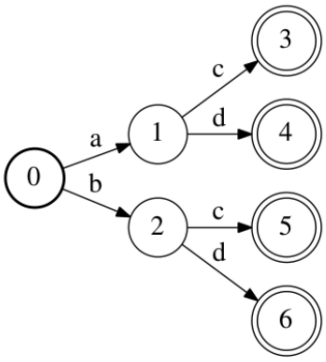
(a) FSA with ε-transitions



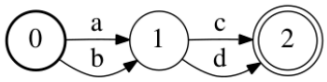
(b) ε-removed FSA

1.3.4 最小化(Minimization)

给定一个FA，我们把所有与之等价的DFA组成的集合里状态数最小的DFA叫做最小DFA。下图(a)是一个DFA，图(b)是与之等价的最小DFA。最小DFA也可以用于检查不同的FA是否等价，因为两个FA通过消除ε跳转、确定化和最小化之后，如果它们是等价的，则经过上述3个操作之后得到的最小DFA是完全一样的。



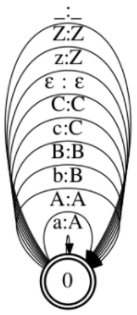
(a) DFA



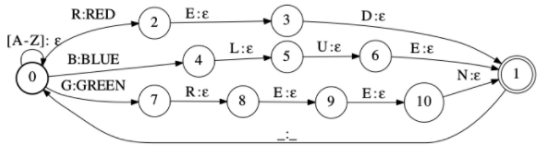
(b) minimal DFA

1.3.5 合并(Composition)

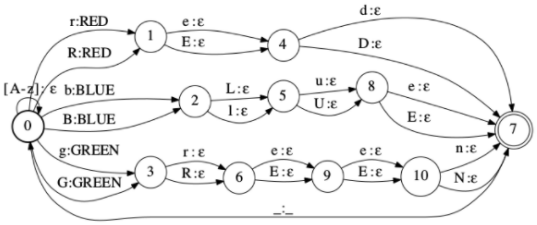
有的时候设计一个大的FST是比较复杂的，但是通过把它分解为多个FST的复合会变得容易得多。例如下图(a)是一个转换机，它的作用是把输入字符串所有的字符都变成大写；图(b)只接受"RED"、"BLUE"和"GREEN"三个词(字符串)。它们复合后的结果如图(c)所示，这个复合后的FST的作用是识别各种大小写组合的这3个词(比如"REd"、"BluE")并将其转换为大写。



(a) Letter transducer: transitions for some letters are omitted because of space limitations.



(b) Word recognition transducer that picks up words RED, BLUE, and GREEN.

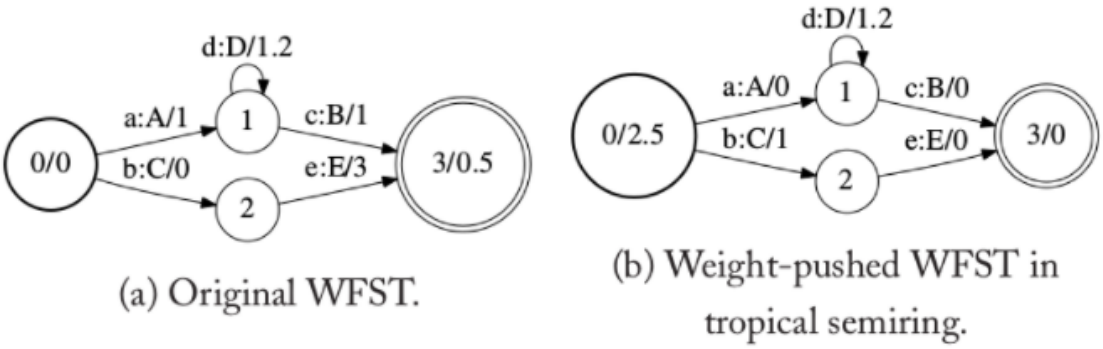


(c) Composition of (a) and (b).

FST的合并运算可以实现很多有用的操作.

1.3.5 Weight Pushing

Weight Pushing运算的作用是把一个WFST所有路径的weight分布往初始状态push，但是不改变任何成功路径的weight。在许多序列识别和转换问题里，寻找最可能或者最小代价是WFST需要解决的最重要的问题。当我们使用一个weighted的FA时，这个问题就变成FA搜索weight最大或者最小的路径的问题。Weight Pushing因为把路径的weight推到前面，因此对于那种look-ahead的算法(比如Beam-Search)，它能更快的滤掉早期不太有希望的路径，从而减少搜索时间。下图是一个 Weight Pushing运算的例子：



2. Kaldi 实现

Kaldi中用于语言识别的WFST解码器由 H、C、L、G 四部分组成，每一部分都是一个转换器，它们的作用分别如表所示：

组成	转换器	输入序列	输出序列
H	HMM	transition-id	mono-phone/tri-phone
C	音素上下文	mono-phone/tri-phone	mono-phone
L	lexicon	mono-phone	word
G	语言模型	word	word

HCLG的构图顺序为 $G \rightarrow L \rightarrow C \rightarrow H$，构图过程为 $G \rightarrow LG \rightarrow CLG \rightarrow HCLG$, 具体过程如下：

$$HCLG = als(min(rds(det(H^{'} \ o \ min(det(C \ o \ min(det(L \ o \ G)))))))))$$

als 表示添加自环，rds 表示去除消歧符号，$H^{'}$ 表示不带自环的HMM，o 代表合并操作，det 代表确定化操作，min 代表最小化操作。

2.1 mkgraph 代码

kaldi中的mkgraph.sh实现了整个构建WFST图的操作：

```
#decode using the tri6b model
utils/mkgraph.sh data/lang_test_tgsmall \
               exp/tri6b exp/tri6b/graph_tgsmall
```

```
Usage: utils/mkgraph.sh [options] <lang-dir> <model-dir> <graphdir>
e.g.: utils/mkgraph.sh data/lang_test exp/tri1/ exp/tri1/graph
 Options:
 --remove-oov       #  If true, any paths containing the OOV symbol (obtained
from oov.int
                    #  in the lang directory) are removed from the G.fst during
compilation.
 --transition-scale #  Scaling factor on transition probabilities.
 --self-loop-scale  #  Please see: http://kaldi-asr.org/doc/hmm.html#hmm_scale.
```

脚本中的细节：

```
# 定义转移概率尺度及自转移概率尺度
tscale=1.0
loopscale=0.1

#合并L.fst 及 G.fst，进行确定化（fstdeterminizestar）和最小化（fstminimizeencoded），得到LG.fst，并确保结果stochastic，即从每个状态输出的转移概率之和为1
if [[ ! -s $lang/tmp/LG.fst || $lang/tmp/LG.fst -ot $lang/G.fst || \
      $lang/tmp/LG.fst -ot $lang/L_disambig.fst ]]; then
  fsttablecompose $lang/L_disambig.fst $lang/G.fst | fstdeterminizestar --use-
log=true | \
    fstminimizeencoded | fstpushspecial > $lang/tmp/LG.fst.$$ || exit 1;
  mv $lang/tmp/LG.fst.$$ $lang/tmp/LG.fst
  fstisstochastic $lang/tmp/LG.fst || echo "[info]: LG not stochastic."
fi

#合并生成CLG.fst
clg=$lang/tmp/CLG_${N}_${P}.fst
clg_tmp=$clg.$$
ilabels=$lang/tmp/ilabels_${N}_${P}
ilabels_tmp=$ilabels.$$
trap "rm -f $clg_tmp $ilabels_tmp" EXIT HUP INT PIPE TERM
if [[ ! -s $clg || $clg -ot $lang/tmp/LG.fst \
    || ! -s $ilabels || $ilabels -ot $lang/tmp/LG.fst ]]; then
  fstcomposecontext $nonterm_opt --context-size=$N --central-position=$P \
   --read-disambig-syms=$lang/phones/disambig.int \
   --write-disambig-syms=$lang/tmp/disambig_ilabels_${N}_${P}.int \
    $ilabels_tmp $lang/tmp/LG.fst |\
    fstarcsort --sort_type=ilabel > $clg_tmp
  mv $clg_tmp $clg
  mv $ilabels_tmp $ilabels
  fstisstochastic $clg || echo "[info]: CLG not stochastic."
fi

#基于HMM拓扑结构、转移概率和决策树，构建不带自转移的声学模型Ha.fst（make-h-transducer），每个转移的输入标签为“trans-id”
if [[ ! -s $dir/Ha.fst || $dir/Ha.fst -ot $model  \
    || $dir/Ha.fst -ot $lang/tmp/ilabels_${N}_${P} ]]; then
  make-h-transducer $nonterm_opt --disambig-syms-out=$dir/disambig_tid.int \
    --transition-scale=$tscale $lang/tmp/ilabels_${N}_${P} $tree $model \
     > $dir/Ha.fst.$$  || exit 1;
  mv $dir/Ha.fst.$$ $dir/Ha.fst
fi
```

```
#将不带自转移的声学模型Ha.fst和CLG.fst组合（fsttablecompose），然后进行确定化
（fstdeterminizestar），去除消歧符号，去除空转移，然后进行最小化（fstminimizeencoded），得
到HCLGa.fst
if [[ ! -s $dir/HCLGa.fst || $dir/HCLGa.fst -ot $dir/Ha.fst || \
      $dir/HCLGa.fst -ot $clg ]]; then
  if $remove_oov; then
    [ ! -f $lang/oov.int ] && \
      echo "$0: --remove-oov option: no file $lang/oov.int" && exit 1;
    clg="fstrmsymbols --remove-arcs=true --apply-to-output=true $lang/oov.int
$clg|"
  fi
  fsttablecompose $dir/Ha.fst "$clg" | fstdeterminizestar --use-log=true \
    | fstrmsymbols $dir/disambig_tid.int | fstrmepslocal | \
     fstminimizeencoded > $dir/HCLGa.fst.$$ || exit 1;
  mv $dir/HCLGa.fst.$$ $dir/HCLGa.fst
  fstisstochastic $dir/HCLGa.fst || echo "HCLGa is not stochastic"
fi

#添加自环，增加每个HMM状态的自转移，从HCLGa.fst得到HCLG.fst
if [[ ! -s $dir/HCLG.fst || $dir/HCLG.fst -ot $dir/HCLGa.fst ]]; then
  add-self-loops --self-loop-scale=$loopscale --reorder=true $model
$dir/HCLGa.fst | \
    $prepare_grammar_command | \
    fstconvert --fst_type=const > $dir/HCLG.fst.$$ || exit 1;
  mv $dir/HCLG.fst.$$ $dir/HCLG.fst
  if [ $tscale == 1.0 -a $loopscale == 1.0 ]; then
    # No point doing this test if transition-scale not 1, as it is bound to
fail.
    fstisstochastic $dir/HCLG.fst || echo "[info]: final HCLG is not
stochastic."
  fi
fi
```

2.2 FST可视化

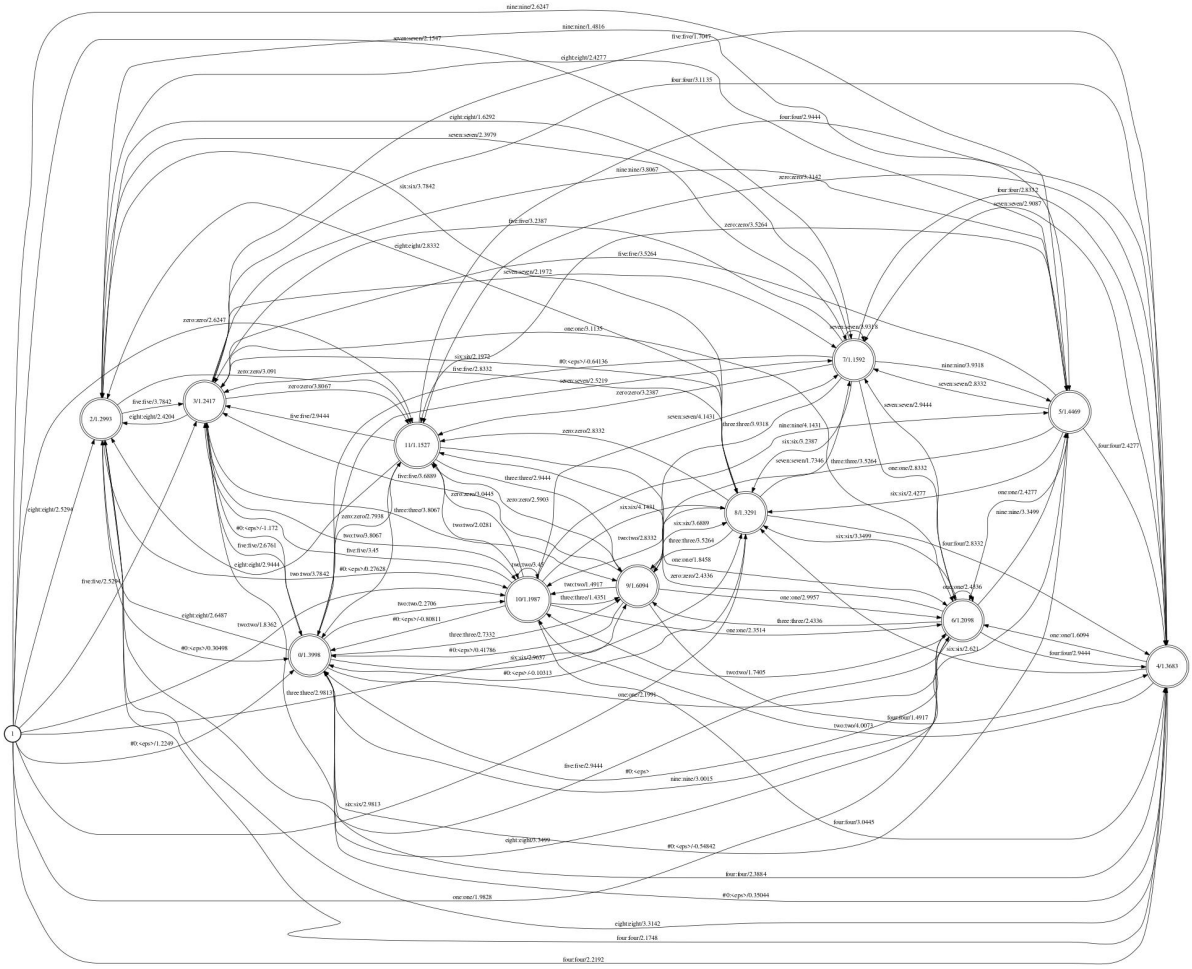
这里我们给出一些可视化的例子

语言模型 G.fst

```
# 将arpa格式LM转换为fst格式。
arpa2fst --disambig-symbol=#0 --read-symbol-table=words.txt ngram.arpa G-
ngram.fst
# 将fst文件输出为dot格式文件
fstdraw --isymbols=words.txt --osymbols=words.txt G-ngram.fst > G-ngram.dot
# 用dot文件生成jpg图片文件
dot -Tjpg G-ngram.dot > G-ngram.jpg
```

1-gram

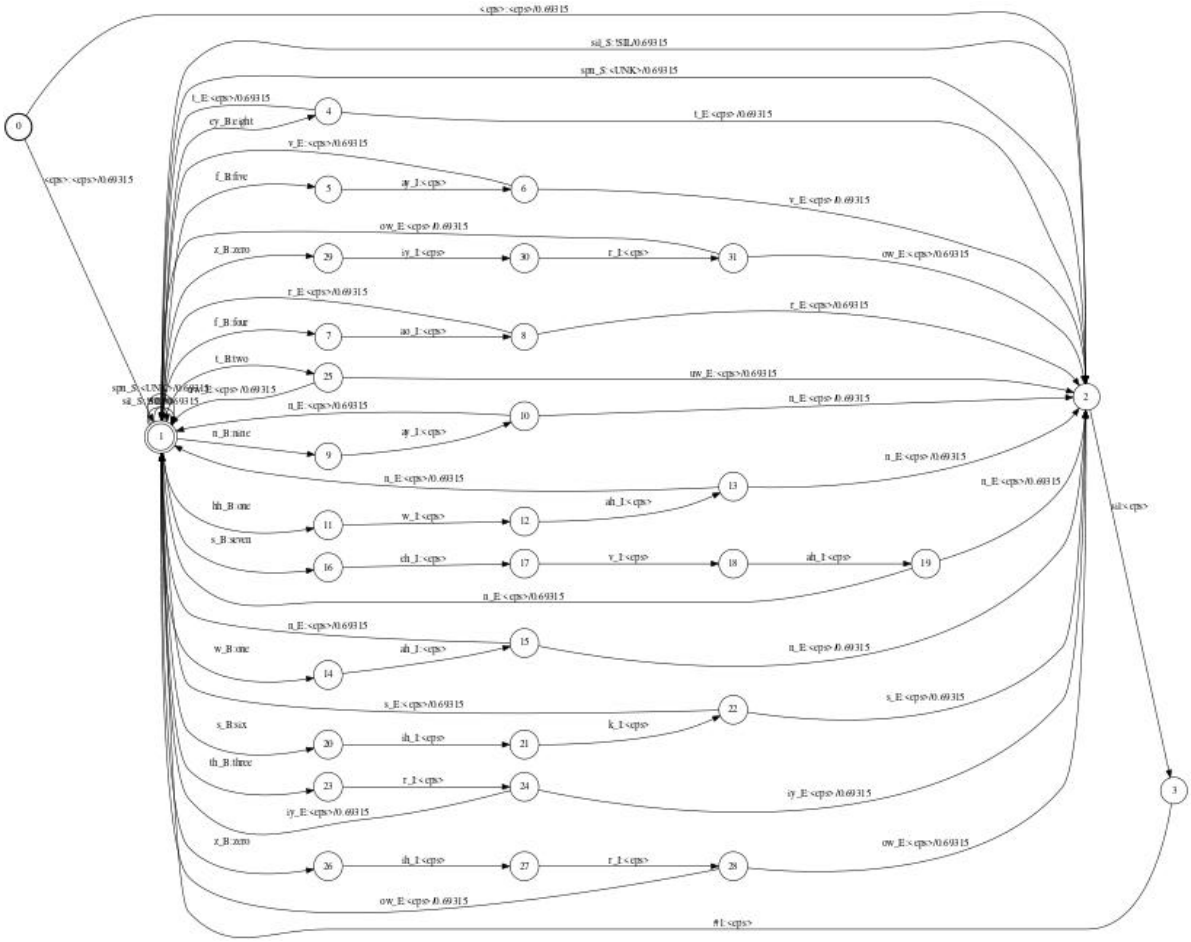
2-gram



L.fst

L.fst 文件的生成在utils/prepare_lang.sh 脚本中实现，执行如下命令将发音词典模型 fst 输出为 dot 格式文件：

```
fstdraw --isymbols=phones.txt --osymbols=words.txt L_disambig.fst >
L_disambig.dot
```



fst文本化

```
fstprint --isymbols=words.txt  --osymbols=words.txt  G.fst G.txt
fstprint --isymbols=phones.txt --osymbols=words.txt L.fst L.txt
```

```
1       2       eight   eight   2.52935743
1       3       five    five    2.52935743
1       4       four    four    2.21920371
1       5       nine    nine    2.62466836
1       6       one     one     1.98281479
1       7       seven   seven   2.15466499
1       8       six     six     2.98134327
1       9       three   three   2.98134327
1       10      two     two     1.8362112
1       11      zero    zero    2.62466836
1       0       #0      <eps>   1.22489929
0       2       eight   eight   2.64866376
0       3       five    five    2.67606211
0       4       four    four    2.38838172
```

参考文献

https://www.openfst.org/twiki/pub/FST/FstBackground/hbka.pdf