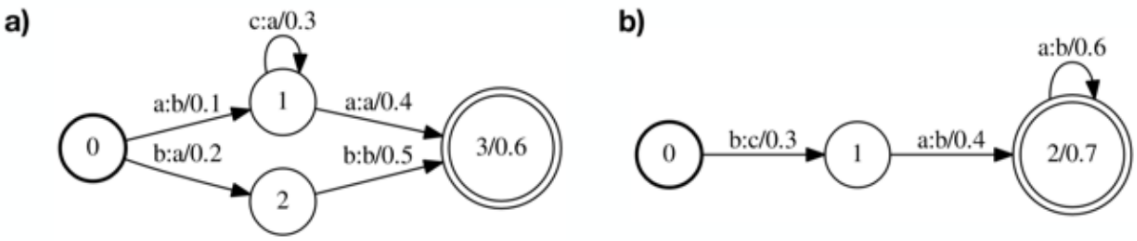
作业一



1. 将图示 a) 和 b) 两个 WFST 写成 text 格式 a.txt.fst 和 b.txt.fst。

```
$ cat > a.text.fst << EOF
0 1 a b 0.1
0 2 b a 0.2
1 1 c a 0.3
1 3 a a 0.4
2 3 b b 0.5
3 0.6
EOF
$ cat > b.text.fst << EOF
0 1 b c 0.3
1 2 a b 0.4
2 2 a b 0.6
2 0.7
EOF
```

2. 定义 input label 和 output label 的字符表 (即字符到数值的映射)。

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

3. 生成 a) 和 b) 对应的 binary 格式 WFST，记为 a.fst 和 b.fst。

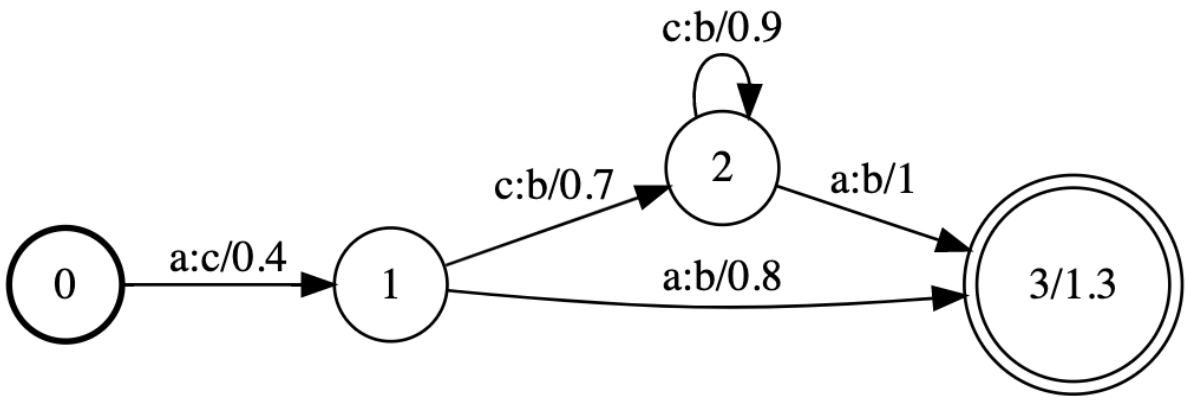
```
$ fstcompile --isymbols=isyms.txt --osymbols=osyms.txt a.text.fst a.fst
$ fstcompile --isymbols=isyms.txt --osymbols=osyms.txt b.text.fst b.fst
```

4. 进行 compse，并输出 out.fst。

```
$ fstcompose a.fst b.fst comp.fst
```

5. 打印输出的样子。

```
$ fstdraw --portrait --isymbols=isyms.txt --osymbols=osyms.txt comp.fst
comp.dot
$ dot -Tps comp.dot -o comp.ps
```



作业二

从 `kaldi/src/decoder/lattice-faster-decoder.cc` 中查找

1. histogram pruning 的代码段
   - 计算剪枝的阈值
     1. 不指定 `--max-active` 和 `--min-active`

        ```
        // 默认 config_.beam = 16.0
        cur_cutoff = best_weight + config_.beam;
        adaptive_beam = config_.beam;
        ```

     2. 指定 `--max-active` 和 `--min-active`

```
// max_active_cutoff < beam_cutoff
cur_cutoff = max_active_cutoff;
adaptive_beam = max_active_cutoff - best_weight +
config_.beam_delta;

// max_active_cutoff >= beam_cutoff && min_active_cutoff >
beam_cutoff
cur_cutoff = min_active_cutoff
adaptive_beam = min_active_cutoff - best_weight +
config_.beam_delta;

// max_active_cutoff >= beam_cutoff && min_active_cutoff <=
beam_cutoff
cur_cutoff = best_weight + config_.beam
adaptive_beam = config_.beam;
```

- 剪枝

```
if (tok->tot_cost <= cur_cutoff) {
  ...
}
```

2. beam pruning 的代码段
   - 计算剪枝的阈值

```
BaseFloat new_weight = arc.weight.Value() + cost_offset -
            decodable->LogLikelihood(frame, arc.ilabel) + tok-
>tot_cost;
// next_cutoff 初始化为上一时刻最优 token 前向传递产生的 cost
if (new_weight + adaptive_beam < next_cutoff)
  next_cutoff = new_weight + adaptive_beam;
```

   - 剪枝

```
cur_cost = tok->tot_cost,
tot_cost = cur_cost + ac_cost + graph_cost;
if (tot_cost >= next_cutoff) continue;
// 加上前向传递产生的 cost 后，tot_cost 上一时刻用最优 token 的 tot_cost
更小
else if (tot_cost + adaptive_beam < next_cutoff)
  next_cutoff = tot_cost + adaptive_beam; // prune by best current
token
```

作业三

运行 `kaldi/egs/mini_librispeech` 至少训练完 3 音素模型 tri1 (nosp: no silence phone)。

1. 此时你的 `data/lang_nosp_test_tglarge` 中无 G.fst 文件，将 `data/local/lm/lm_tglarge.arpa.gz` 转化为 `G.fst` 存于其中 (tglarge 原本是用在 `steps/lmrescore_const_arpa.sh` 中的，所以没有 G.fst 文件)。

```
$ utils/format_lm.sh data/lang_nosp_test_tglarge
data/local/lm/lm_tglarge.arpa.gz data/local/dict_nosp/lexicon.txt
data/lang_nosp_test_tglarge
```

2. 用 tri1 模型和 tgsmall 构建的 HCLG 图解码 dev_clean_2 集合的“1272-135031-0009”句，输出 Lattice 和 CompactLattice 的文本格式 (`utils/show_lattice.sh` 可以绘图)。

```
$ utils/mkgraph.sh data/lang_nosp_test_tgsmall exp/tri1
exp/tri1/graph_nosp_tgsmall
$ steps/decode.sh --nj 5 --cmd run.pl exp/tri1/graph_nosp_tgsmall
data/dev_clean_2 exp/tri1/decode_nosp_tgsmall_dev_clean_2

$ . ./path.sh
$ lattice-copy --write-compact=false ark:'gunzip -c
exp/tri1/decode_nosp_tgsmall_dev_clean_2/lat.1.gz |' 'scp,t,p:echo
1272-135031-0009 -|' | utils/int2sym.pl -f 4
data/lang_nosp_test_tgsmall/words.txt > lat.txt

$ lattice-copy ark:'gunzip -c
exp/tri1/decode_nosp_tgsmall_dev_clean_2/lat.1.gz |' 'scp,t,p:echo
1272-135031-0009 -|' | utils/int2sym.pl -f 3
data/lang_nosp_test_tgsmall/words.txt > clat.txt
```

   - Lattice (FST):

     ```
     utterance_ID
     start_state end_state transition_ID word weights
     (graph_cost,acoustic_cost)
     ```

   - CompactLattice (Acceptor):

     ```
     utterance_ID
     start_state end_state word weights
     (graph_cost,acoustic_cost,transition_IDs)
     ```

3. 使用 1 中生成的 tglarge 的 G.fst 和 `steps/lmrescore.sh` 对 `exp/tri1/decode_nosp_tgsmall_dev_clean_2` 中的 lattice 重打分，汇报 wer (`RESULTS` 脚本提供统计命令)。

```
$ steps/lmrescore.sh --cmd run.pl
data/lang_nosp_test_{tgsmall,tglarge} data/dev_clean_2
exp/tri1/decode_nosp_{tgsmall,tglarge}_dev_clean_2
$ grep WER exp/tri1/decode_nosp_tglarge_dev_clean_2/wer_* |
utils/best_wer.sh
```

当 `LMWT = 15` 和 `WIP = 0.0` 时可以得到最低的 WER：

```
%WER 20.98 [ 4224 / 20138, 457 ins, 591 del, 3176 sub ]
exp/tri1/decode_nosp_tglarge_dev_clean_2/wer_15_0.0
```