Lab 4 — Large Vocabulary Decoding: A Love Story

ELEN E6884/COMS 86884: Speech Recognition

Due: November 21, 2005 at 12:01am

0 Overview

By far the sexiest piece of software associated with ASR is the large-vocabulary decoder. Since this course is nothing if not about being sexy, this assignment will deal with various aspects of large-vocabulary decoding. In the first portions of this lab, we will investigate the various steps involved in building static decoding graphs as will be needed by our decoder. In the second half of the lab, you will implement a complete real-time large-vocabulary decoder from scratch. Just kidding! While such decoders are not that complex when working within the static graph paradigm, they are still beyond the scope of a two-week lab. Instead, we provide an almost complete decoder, and you will be required to add various features to it, namely word traceback a.k.a. *token passing*, skip arc support, and beam and rank pruning.

The goal of this assignment is for you, the student, to gain a better understanding of the various steps involved in constructing a static decoding graph for LVCSR and of the various algorithms used in large-vocabulary decoding. The lab consists of the following parts, all of which are required:

- **Part 1: Playing with FSM's and the IBM FSM toolkit**
- **Part 2: Investigating the static graph expansion process**
- **Part 3: Add support for word traceback and skip arcs to a large-vocabulary decoder**
- **Part 4: Add support for beam and rank pruning to a large-vocabulary decoder**
- **Part 5: Evaluate the performance of various models on WSJ test data**

All of the files needed for the lab can be found in the directory `~stanchen/e6884/lab4/`. Before starting the lab, please read the file `lab4.txt`; this includes all of the questions you will have to answer while doing the lab. Questions about the lab can be posted on Courseworks (`https://courseworks.columbia.edu/`); a discussion topic will be created for each lab. **Note:** The hyperlinks in this document are enclosed in square brackets; you need an online version of this document to find out where they point to.

1 Part 1: Playing With FSM's and the IBM FSM Toolkit

In this part, we introduce the IBM FSM toolkit as a first step in learning more about the static graph expansion process. In particular, you will be using the program FSMOP, which is a utility that can perform a variety of finite-state operations on weighted FSA's and FST's. The program FSMOP is like a calculator that operates on FSM's rather than real values, where arguments are input using [reverse Polish notation] as is used in some [HP calculators]. For example, to produce the composition of an FSA held in the file `foo.fsm` and an FST held in `bar.fsm`, you would use the command

```
FsmOp foo.fsm bar.fsm -compose > result.fsm
```

Operations begin with the "`-`" character, and include `-compose`, `-determinize`, and `-minimize` among many others. By default, the resulting FSM is written to standard output, but if the last argument supplied is a filename (rather than an operation), the resulting FSM will be written to that file instead. Thus, the command

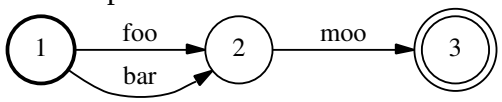
```
FsmOp foo.fsm bar.fsm -compose result.fsm
```

has the same effect as the last example.

We explain the default FSA format through an example:

```
1   2   foo
1   2   bar
2   3   moo
3
```

Each line with three fields describes an arc in the FSM; the format is: *src-state dst-state label*. States need not be numbered starting from 1; the label <`epsilon`> is used to represent the empty label. Each line with a single field lists a final state. The first state mentioned in the file is the start state. Thus, the above FSA file corresponds to:



We drew the above Postscript diagram using the following command:

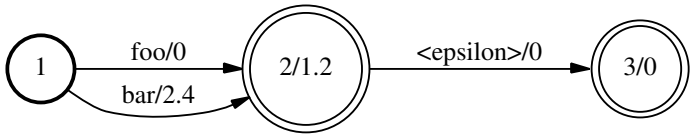
```
FsmOp foo.fsm -draw | dot -Tps > foo.ps
```

You can use similar commands to help you visualize FSM's.

For weighted FSA's, each line can optionally be followed with a *cost*, or negative log probability base 10. If a cost is omitted on a line, it is taken to be zero. For example, here is a weighted FSA:

```
1   2   foo
1   2   bar 2.4
2   3   <epsilon>
2   1.2
3
```

corresponding to



Finite-state transducers have a similar format, except lines representing arcs have an extra field:

src-state dst-state in-label out-label [*optional-cost*]

In addition, to signal that a file holds an FST rather than an FSA, the following line should be included at the start of the file:

```
# transducer: true
```

Here is an example FST:

```
# transducer: true
1   2   ax  AX
1   2   bar BAR 1.0
2   3   moo <epsilon>
2   1.2
3
```

For this part of the lab, you will have to create various FSM's and perform operations on them, as described in `lab4.txt`. Here are some hints:

- Don't forget to add final states to your FSM's! Without these, your FSM's will be equivalent to empty FSM's.
- For transducers, don't forget the line "`# transducer: true`"!
- For a list of all of the operations that FSMOP can perform, run FSMOP with no arguments.

To prepare for this part, create the relevant subdirectory and copy over the needed files:

```
mkdir -p ~/e6884/lab4/
cd ~/e6884/lab4/
cp ~stanchen/e6884/lab4/copy/* .
cp ~stanchen/e6884/lab4/.mk_chain .
```

This will also copy over all files needed in later parts of the lab.

2 Part 2: Investigating the Static Graph Expansion Process

In this part of the lab, we will look at the static graph expansion process used to create the decoding graphs that we will need for our decoder. First, we introduce the models that we are using in this lab, and then we go step-by-step through the graph expansion process.

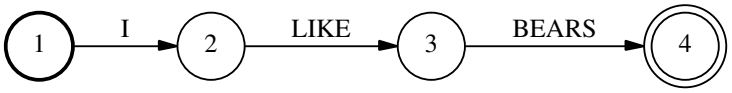
2.1 The Model

For this lab, we will be working with the Wall Street Journal corpus. This corpus was created many years ago for a program sponsored by DARPA to spur development of basic LVCSR technology, and this corpus was designed to be about as easy as an LVCSR corpus could be for ASR. The data consists of people reading Wall Street Journal articles into a close-talking microphone in clean conditions; *i.e.*, there is little noise. Since the data is read text, there are few conversational artifacts (such as filled pauses like UH) and it is easy to find relevant language model training data.

We trained an acoustic model on about 10 hours of Wall Street Journal audio data and a smoothed trigram language model on 24M words of WSJ text. The acoustic model is context-dependent; phonetic decision trees were built for each state position (*i.e.*, start, middle, and end) for each of the ~ 50 phones, yielding about 3×50=150 trees. The trees have a total of about 3000 leaves; *i.e.*, the model contains 3000 GMM's, one for each context-dependent variant of each state for each phone. Each GMM consists of eight Gaussians, so the model contains about 3000×8=24000 Gaussians or *prototypes*. The model was trained by first training a (1 Gaussian per phone) context-independent phonetic model, using this to seed a (1 Gaussian per leaf) CD phonetic model, and then doing Gaussian splitting until we achieved 8 Gaussians per leaf or GMM. For the front end, we used 13-dimensional MFCC's with delta's and delta-delta's, yielding 3×13=39 dimensional feature vectors. The vocabulary was taken to be the 5000 most frequent words in the LM training data. If this paragraph meant nothing to you, you should probably cut down on the brewski's.

2.2 Graph Expansion Steps

We explain the steps in graph expansion by going through an example. For full-scale decoding, we would start with a word FSM representing a pruned trigram LM; in this example, we start with a word graph `wd.fsm` containing just a single word sequence:

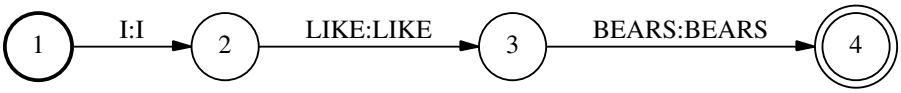


(All example files in this section can be found in `~stanchen/e6884/lab4/`.)

The first thing we do is convert the FSA into an FST:

```
FsmOp wd.fsm -make-transducer wd2.fsm
```

resulting in

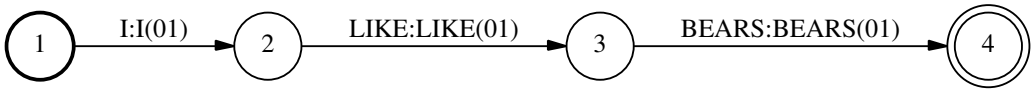


The reason why we do this is left as an exercise.

Then, we compose the FST `wd2lx.fsm` to convert from words to pronunciation variants, or *lexemes* in IBM terminology:

```
FsmOp wd2.fsm wd2lx.fsm -compose lx.fsm
```

In this case, each word has only a single pronunciation variant (*e.g.*, I(01) denotes the first pronunciation variant of the word I), but in general there may be words with multiple pronunciations. This yields

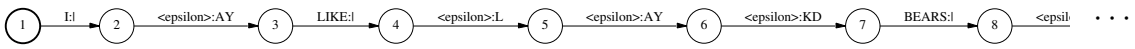


Notice that `wd2lx.fsm` only contains entries for the words in `wd.fsm`; in general, this FST would contain entries for all words in some large vocabulary.

Next, we convert from lexemes to phones:

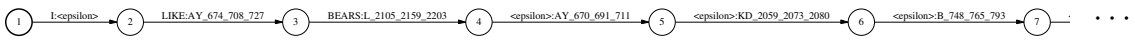
```
FsmOp lx.fsm lx2pn.fsm -compose pn.fsm
```

yielding



(For this and following FSM's, we do not display the whole machine due to space constraints.) Notice that each pronunciation begins with the marker "|"; this is used to encode the location of word boundaries. This marker is used during phonetic decision tree expansion, but does not expand to an HMM itself.

Next, we convert from phones to the *model* level (held in the file `md.fsm`); this involves applying the phonetic decision trees to find the context-dependent variation at each state position of each phone. We do not explain how we do this since it involves *virtual* FST's and we don't want to get into this, but the result is



To explain the notation, this model has a total of 3416 leaves/GMM's, or context-dependent state variants, which are numbered from 0 to 3415. The notation `AY_674_708_727` denotes that the phone `AY` in this context expands to the leaves/GMM's 674, 708, and 727, respectively, for the first, middle, and last states of its 3-state HMM. Notice that the word labels are no longer necessarily aligned with the models they expand to. This is because the identity of the leaves in a model may not be known until phones to the right (since the decision tree may ask about phones to the right), so the model tokens are shifted later relative to the word tokens.

The contents of each decision tree can be found in the file `tree.txt`. Here is an excerpt of the tree for the feneme `AY_1`, *i.e.*, the first state of the phone `AY`:

```
Tree for feneme AY_1:
node   0: quest-P 111[-1] --> true: node   1, false: node   2
  quest: |
node   1: quest-P  36[-2] --> true: node   3, false: node   4
  quest: D$ X
node   2: quest-P  66[-1] --> true: node   5, false: node   6
  quest: AO AXR ER IY L M N NG OW OY R UH UW W Y
node   3: leaf  661
node   4: quest-P  15[-2] --> true: node   7, false: node   8
  quest: AXR ER L OW R UW W
```

Node 0 is the root of the tree. At node 0, question 111 is asked of the phone in position -1 (*i.e.*, the phone to the left); if the question is true, go to node 1, else go to node 2. Question 111 asks whether the given phone belongs to the set {"|"}. By convention, the beginning and ending of utterances are padded with the "`-1`" phone, or unknown phone. (Recall that the "|" phone is the word boundary phone and is placed at the beginning of each word pronunciation.)

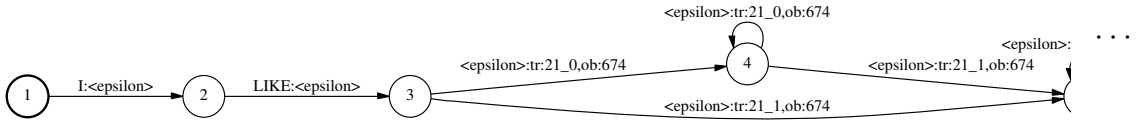
Let us go through a partial example of calculating the leaf number for the first state in the first `AY` phone in our example. Here, the phone to the left of the first `AY` is the "|" phone, so the question at the root node is true and we go to node 1. At node 1, we ask about the phone at position -2, or two phones to the left. This is beyond the beginning of the utterance, so this means we use the `-1` phone. Since this phone does not belong to the set {`D$`, `X`}, the question is false and we go to node 4. The remainder of this example is left as an exercise. Some notes:

- For `MERGE` nodes, this means that two nodes in the tree were merged into one because they had sufficiently similar output distributions, and you should just go directly to the named node.
- The decision tree used in this example is a *left-context* decision tree, which means it is only allowed to ask questions across a word boundary for words to the *left*. (This is kind of halfway between a word-internal and true cross-word decision tree.) That is, if a question is ever asked about a phone position that is beyond the end of the current word, you should pretend the `-1` phone is in that position.

Anyway, back to our graph expansion example. In the last step, we expand the FSM to the final HMM, rewriting each model token by the HMM that represents it:

```
FsmOp md.fsm md2obtr.fsm -compose obtr.fsm
```

This HMM is too large to display in its entirety, but here is an excerpt:



Notice that we pack two different tokens into the output token for each arc, separated by a "`,`". The notation `ob:674` states that the GMM corresponding to leaf 674 should be used as the output distribution for that arc. The notation `tr:21_0` encodes which transition probability should be used on this arc; transition probability markers will be translated into arc costs within the decoder. Why transition probabilities are encoded in tokens rather than as arc costs is left as an exercise. While the topology of the above HMM does not look exactly like what we've been presenting in class, it is actually equivalent.

The actual transition probabilities are kept in a separate file, like the file `example.tmd`. In this file, the first line contains the probabilities corresponding to the tokens `tr:0_0` and `tr:0_1`, respectively, and the last line corresponds to `tr:155_0` and `tr:155_1`. The Gaussian parameters are kept in a file like `example.omd`, which contains 2-component Gaussian mixtures. In the first part of the file, each line corresponds to a leaf. For each leaf, the index of each Gaussian in the mixture and the mixture weight for that Gaussian is listed. In the latter part of the file, each line corresponds to a Gaussian; the Gaussian indices listed for each leaf are indices into this list of Gaussians. For each Gaussian, the mean and variance for each dimension is listed.

So, this pretty much describes all the data that we need to feed into our decoder. As mentioned before, in real life we begin with a word graph representing an n-gram language model. The LM word graph we used in this lab can be found in `lm.fsm.gz`; it is a trigram model that has been pruned to about 200k bigrams and 125k trigrams. To look at it, you can use ZCAT (as it's compressed), *e.g.*,

```
zcat lm.fsm.gz | more
```

The final expanded decoding graph can be found in `decode.fsm.gz`. It contains about 1.5M states and 4M arcs.

For this part of the lab, you will have to edit some of the FSM's used in our toy example to handle a new word in the vocabulary. In addition, you will need to do some manual decision-tree expansion; see `lab4.txt` for directions.

3 Part 3: Add Support for Word Traceback and Skip Arcs to a Large-Vocabulary Decoder

In this part of the lab, we provide a nearly complete large vocabulary decoder, and you will need to add support for recording word traceback information as well as support for handling skip arcs, *i.e.*, arcs with no output. However, we have done most of the work; these changes should involve adding/modifying only a few lines of code.

Now, we outline the basic functioning of the code. Recall the Viterbi algorithm we implemented for Lab 2:

```
C[0, start].vProb = 1
for t in [0...(T-1)]:
  for s_src in [1...S]:
    for a in outArcs(s_src):
      s_dst = dest(a)
      curProb = C[t, s_src].vProb × arcProb(a, t)
      if curProb > C[t+1, s_dst].vProb:
        C[t+1, s_dst].vProb = curProb
        C[t+1, s_dst].trace = a
```

Now, look at the main function `ViterbiLab4G` in `Lab4_DP.C`, which executes the Viterbi algorithm for a single utterance. As mentioned in [Lecture 8], it is not generally feasible to store the whole dynamic programming chart $C[t, s]$ in large-vocabulary decoding. Instead, we store only the active states for the last and current frames. For each frame, we store the active states in a `FrameData` structure. We allocate two of these structures, and initially set the variable `lastFrame` to point to one and `nextFrame` to point to the other. For each iteration in our outer loop over frames:

```
for (int frm = 0; frm <= numFrames; ++frm)
```

the variable `lastFrame` holds the active states for frame `frm` and `nextFrame` holds the active states that we are creating for frame `frm+1`.[1] At the end of the iteration, we swap `lastFrame` and `nextFrame`, and at the beginning of the next iteration, we clear `nextFrame`, thus setting us up correctly for the next iteration.

Now, in Lab 2, we iterated over all states, but for LVCSR we can only iterate over active states. In addition, we must iterate over active states in topological order (relative to skip arcs) in order for the dynamic programming calculation to be correct, as mentioned in class. We handle this for you, using a data structure known as a [heap]. A heap holds a list of elements such

[1] You may notice that the loop over frames goes to `numFrames` inclusive, which seems one frame too far. We iterate over this last frame to update skip arcs only, and skip over emitting arcs for this frame.

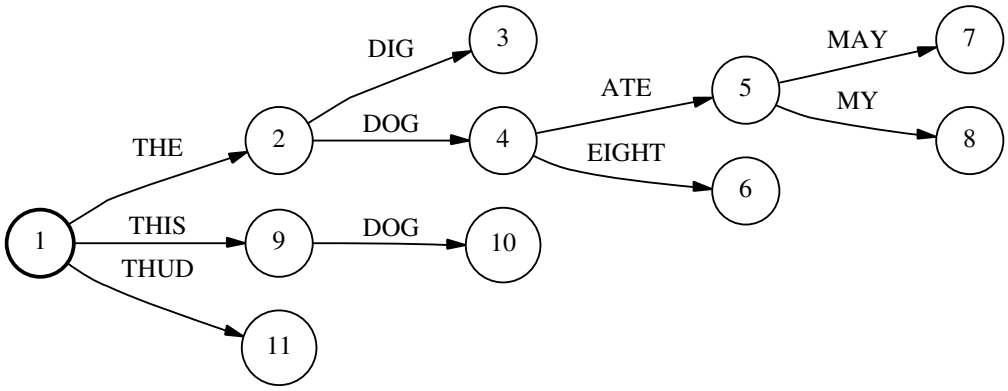


Figure 1: A Backtrace Word Tree

that the top of the heap always holds the first element according to some ordering, in our case, the topological ordering. In the code, the loop over active states is implemented by first calling `lastFrame->init_heap()` to initialize the heap for active states in the last frame, and then doing

```
while ((curState = lastFrame->pop_heap()))
```

to pop states off the top of the heap until the heap is empty. In our decoder, we assume that the static decoding graph has already been topologically sorted relative to skip arcs, so topological state ordering will be the same as numeric order.

To create/lookup a DP cell for a state, we use the method `insert_cell()`. That is, we no longer preallocate all cells $C[t, s]$ in the DP chart for a frame t, and instead create cells on the fly as new active states are created. For example, see the line

```
ChartCell& dstCell = nextFrame->insert_cell(dstState);
```

If you don't understand the "&" notation, you can do the following instead:

```
ChartCell* dstCellPtr = &nextFrame->insert_cell(dstState);
```

If a cell for state `dstState` already exists in `nextFrame`, this cell is returned; otherwise, a new cell is created (and its logprob is initialized to `zeroLogProbS`). Then, this covers the main differences between the Viterbi implementations for Lab 2 and Lab 4.

Now, the first task for Part 3 is to add backtrace updating; the code as it stands only calculates the Viterbi probability but not the best word sequence. Unlike in previous labs, we do not provide `BEGIN_LAB` and `END_LAB` markers; you need to figure out where to make the changes by yourself. Backtrace information is stored in the `wordTreeNodeM` field of the `ChartCell` structure. That

is, in contrast to [Lab 2] where we store the best incoming arc for each cell in the chart, we now store the word sequence labeling the best incoming path for each cell in the dynamic programming chart. As described in class, we encode the best word sequence as a single integer index that specifies a node in what we call a *word tree*. An example word tree is depicted in Figure 1. The word sequence associated with a node is the sequence of words labeling the path from the root node to that node. For example, with the word graph in Figure 1, if a cell has a `wordTreeNodeM` value of 5, this signals that the best path to this cell is labeled with the word sequence THE DOG ATE.

For this part of the lab, you will need to set the `wordTreeNodeM` field for all cells iterated over and to update the word tree as you go, creating new nodes as needed. The word tree is stored in a variable `wordTree` of type `WordTree`. (Note: In the code, the root has index 0 rather than 1 as in Figure 1.) Initially, the word tree will consist of a single node, the root node. To complete this task, you will need to access the following method of `WordTree`: `wordTree.insert_node(prevNode, nextWord)`. This returns the index of the node you arrive at if you extend the node `prevNode` with the word `nextWord` (represented as an integer index), creating the node if it doesn't already exist. (Recall from Lab 3 that we can represent word identities as integer indices.) For example, `wordTree.insert_node(5, nextWord)` would return the value 8 in the above example if `nextWord` corresponds to the word MY. To find the word label residing on an arc `arcPtr`, call the method `arcPtr->get_label()`. This returns the integer index of the word residing on the arc, or 0 if there is no word label. Note: most arcs do not have word labels on them.

To complete this task, you should need only write a few lines of code. Hint: if the best path into `dstCell` comes from `curCell` and the arc between them has no word label, how does the `wordTreeNodeM` value of `dstCell` compare to that of `curCell`? What if the arc between them does have a word label?

Your code will be compiled into the program DCDLAB4, which is almost identical to the program DCDLAB2 from [Lab 2], except that it links in the Lab 4 Viterbi implementation instead of the Lab 2 Viterbi implementation. To compile this program with your code, type

```
smk DcdLab4
```

To run this program on a small test set of Wall Street Journal utterances using the models described in Section 2.1, run

```
lab4p3a.sh
```

In this part, for speed's sake we are not using our big static decoding graph. Instead, we create an HMM consisting of all "word" sequences consistent with the reference transcript as described in slide 45 in [Lecture 7]. This HMM can be used to produce an "extended" transcript for an utterance, one that includes silences and pronunciation variant information. In this scenario, the decoded output will always contain the same "words" as the reference transcript.

The target output can be found in the file `p3a.out` in `~stanchen/e6884/lab4/`; you should match the decoded word sequence printed for each utterance. As usual, the `-debug` flag can be used to run the script in the debugger. The instructions in `lab4.txt` will ask you to run the script `lab4p3b.sh`, which does the same thing as `lab4p3a.sh` except on a different test set.

For the second task of this part, you need to add support for skip arcs, *i.e.*, arcs with no output. That is, the code as is will not work correctly if skip arcs are present, and you need to find the fix that will make the code work correctly. Note that in the loop over arcs in the code, the variable `hasOutput` will be `false` for skip arcs. For this part, you need only change a single line of code; the code is designed so that the obvious change will result in the correct behavior. Again, you need to recompile DCDLAB4 to test your change:

```
smk DcdLab4
```

To run this program on a small sample test set, run

```
lab4p3c.sh
```

This uses exactly the same models and test set as `lab4p3a.sh`, except that we modified the graph expansion process to introduce unnecessary skip arcs in some places. Target output can be found in the file `p3c.out` in `~stanchen/e6884/lab4/`. You should try to match the logprob for each utterance exactly. The instructions in `lab4.txt` will ask you to run the script `lab4p3d.sh`, which does the same thing as `lab4p3c.sh` except on a different test set.

4 Part 4: Add Support for Beam and Rank Pruning to a Large-Vocabulary Decoder

In the first task for this part, you need to implement beam pruning. For each frame, the variable `threshLogProb` should be set to $b -$ `logProbBeam`, where b is the highest logprob of a cell in the last frame (*i.e.*, `lastFrame`). However, it is unacceptable to just loop through all cells in `lastFrame` right before setting `threshLogProb`, as we are concerned about efficiency and it is possible to solve this task without adding any new loops.

Once you have solved this, don't forget to recompile DCDLAB4. To run this program on a small sample test set, run

```
lab4p4a.sh
```

In this part, we use the same models as before, but we use slightly bigger decoding graphs than in the last part. (Again, for speed's sake, we do not use the full big static decoding graph.) Instead of using a graph that contains just the reference word sequence (more or less), we made a graph

that contains all 245 utterances in our full test set. (Thus, the decoding task is to decide which of the 245 test utterances the current utterance is, rather than true unconstrained word decoding.) The target output can be found in the file `p4a.out` in `~stanchen/e6884/lab4/`. You should try to come close to matching the number of active states reported. The instructions in `lab4.txt` will ask you to run the script `lab4p4b.sh`, which does the same thing as `lab4p4a.sh` except on a different test set.

In the second task for this part, you need to implement rank pruning. Look in the code to see where the rank pruning code should be placed. Note that we provide an example of how to loop through all active cells. (For rank pruning, unlike in beam pruning, it is unclear there is a more efficient way of doing this other than to at least loop through all cells.) In rank pruning, we want to keep the `maxRank` cells with the highest logprobs. However, it is OK if you keep slightly more than `maxRank` cells (*e.g.*, if you do a bucket sort). For this part, set the variable `rankThreshLogProb` to the logprob value such that there are (about) `maxRank` cells in the last frame with at least this logprob. Notice that the rank pruning code will only be called if there are indeed at least `maxRank` active cells in the last frame.

Once you have solved this, you again need to recompile DCDLAB4. To run this program on the same test set as before, run

```
lab4p4c.sh
```

Sample output can be found in the file `p4c.out` in `~stanchen/e6884/lab4/`. The maximum number of active states should be in the ballpark of 1000, which is what `maxRank` is set to in this script, though it is OK if it is slightly larger. You need not match the target output exactly in terms of active states, as your implementation may be completely different from our sample implementation. To give a ballpark figure for how fast a pretty efficient implementation is, our sample implementation runs `lab4p4c.sh` in about 0.42 "times real time" (xRT); *i.e.*, on average, it takes $0.42 \times t$ seconds to process an utterance that is t seconds long. The program DCDLAB4 outputs the real time factor at the end of each run. (It uses CPU time to compute this rather than elapsed time, so it should be fine if someone else is concurrently running a job on the same machine as you.) You don't want your implementation to be too slow, or Part 5 of the lab will involve an excessive amount of waiting.

5 Part 5: Evaluate the Performance of Various Models on WSJ Test Data

In this section, we will be doing runs using our big static decoding graph (yay!) on a small 10-utterance WSJ test set. Before doing this part, you should recompile your code with optimization on so things will run faster. Enter the following commands:

```
smk -all -O2 -DNDEBUG Lab4_DP.o
smk DcdLab4
```

First, let us look into the impact of various modeling decisions. Our baseline system contains 3k 8-component GMM's running on MFCC's with delta's and delta-delta's. Let's see what happens if we reduce the number of Gaussians per mixture; run the following scripts:

```
lab4p5.gmm8.sh
lab4p5.gmm4.sh
lab4p5.gmm2.sh
lab4p5.gmm1.sh
```

These scripts correspond to 8-component, 4-component, 2-component, and single Gaussian GMM's, respectively. (We saved the intermediate models as we did mixture splitting to go from 1 component to 8 components per GMM.) Remember to note the error rate of each run.

Now, let's see how much delta's and delta-delta's help. Run the following scripts:

```
lab4p5.mfccdd.sh
lab4p5.mfccd.sh
lab4p5.mfcc.sh
```

The first run has MFCC with delta's and delta-delta's, the second has MFCC with only delta's, and the last is only MFCC. (Instead of retraining each model from scratch, we truncated the extra dimensions from each Gaussian.) These and all following runs use 8-component GMM's.

Finally, let's see how pruning affects performance. Run the following scripts:

```
lab4p5.10.none.sh
lab4p5.5.none.sh
lab4p5.2.none.sh
lab4p5.none.10k.sh
lab4p5.none.5k.sh
lab4p5.none.2k.sh
lab4p5.10.10k.sh
lab4p5.5.5k.sh
```

The first value in each script name is the beam used in beam pruning; the second value is the rank threshold used in rank pruning. For this part, note both the real-time factor as well as the WER.

6 What is to be handed in

Make a copy of the ASCII file `lab4.txt` from the directory `~stanchen/e6884/lab4/`. Fill in all of the fields in this file and E-mail the contents of the file to `stanchen@watson.ibm.com`. (Please paste this file into the main body of the E-mail; *i.e.*, don't include this file as an attachment.)