Our graph creation recipe during training time is simpler than during test time, mainly because we do not need the disambiguation symbols.

We will assume that you have already read the **test-time** version" of this recipe.
In training time we use the same HCLG formalism as in test time, except that G consists of just a linear acceptor corresponding to the training transcript (of course, this setup is easy to extend to cases where there is uncertainty in the transcripts).

Command-line programs involved in decoding-graph creation

Here we explain what happens at the script level in a typical training script; below we will dig into what happens inside the programs. Suppose we have built a tree and model. The following command creates an archive (c.f. **The Table concept**) that contains the graph HCLG corresponding to each of the training transcripts.

```
compile-train-graphs $dir/tree $dir/1.mdl data/L.fst ark:data/train.tra \
   ark:$dir/graphs.fsts
```

The input file train.tra is a file containing integer versions of the training transcripts, e.g. a typical line might look like:

```
011c0201 110906 96419 79214 110906 52026 55810 82385 79214 51250 106907 111943
99519 79220
```

(the first token is the utterance id). The output of this program is the archive graphs.fsts; it contains an FST (in binary form) for each utterance id in train.tra. The FSTs in this archive correspond to HCLG, except that there are no transition probabilities (by default, compile-train-graphs has –self-loop-scale=0 and –transition-scale=0). This is because these graphs will be used for multiple stages of training and the transition probabilities will change, so we add them in later. But the FSTs on the archive will have probabilities that arise from silence probabilities (these probabilities are encoded into L.fst), and if we were using pronunciation probabilities these would be present too.
A command that reads these archives and decodes the training data, producing state-level alignments (c.f. **Alignments in Kaldi**) is as follows; we will briefly review this command, although our main focus on this page is the graph creation itself.

```
gmm-align-compiled \
  --transition-scale=1.0 --acoustic-scale=0.1 --self-loop-scale=0.1 \
  --beam=8 --retry-beam=40 \
   $dir/$x.mdl ark:$dir/graphs.fsts \
   "ark:add-deltas --print-args=false scp:data/train.scp ark:- |" \
   ark:$dir/cur.ali
```

The first three argument are probability scales (c.f. **Scaling of transition and acoustic probabilities**). The transition-scale and self-loop-scale options are there because this program adds the transition probabilities to the graph before decoding (c.f. **Adding transition probabilities to FSTs**). The next options are the beams; we use an initial beam, and then if alignment fails to reach the end state we use another beam. Since we apply a scale of 0.1 the acoustics, we would have to multiply these beams by 10 to get figures comparable to acoustic likelihoods. The program requires the model; $x.mdl would expand to 1.mdl or 2.mdl, according to the iteration. It reads the graphs from the archive (graphs.fsts). The argument in quotes is evaluated as a pipe (minus the "ark:" and "|"), and is interpreted as an archive indexed by utterance id, containing features. The output goes to cur.ali, and if written in text form it would look like the .tra file described above, although the integers now correspond not to word-ids but transition-ids (c.f. **Integer identifiers used by TransitionModel**).
We note that these two stages (graph creation and decoding) can be done by a single command-line program, gmm-align, which compiles the graphs as they are needed. Because graph creation takes a fairly significant amount of time, any time the graphs will be needed more than once we write them to disk.

Internals of graph creation

The graph creation done in training time, whether from the program compile-train-graphs or from gmm-align, is done by the same code: the class **TrainingGraphCompiler**. Its behaviour when compiling graphs one by one is as follows:
In initialization:

- **Add the subsequential loop** (c.f. **Making the context transducer**) **to the lexicon L.fst, make sure it is sorted on its output label, and store it.**

When compiling a transcript, it does the following:

- Make a linear acceptor corresponding to the word sequence (G)
- Compose it (using TableCompose) with the lexicon to get an FST (LG) with phones on the input and words on the output; this FST contains pronunciation alternatives and optional silence.
- Create a new context FST "C" that will be expanded on demand.
- Compose C with LG to get CLG; the composition uses a special Matcher that expands C on demand.
- Call the function GetHTransducer to get the transducer H (this covers just the context-dependent phones that were seen).
- Compose H with CLG to get HCLG (this will have transition-ids on the input and words on the output, but no self-loops)
- Determinize HCLG with the function DeterminizeStarInLog; this casts to the log semiring (to preserve stochasticity) before determinizing with epsilon removal.
- Minimize HCLG with MinimizeEncoded (this does transducer minimization without weight-pushing, to preserve stochasticity).
- Add self-loops.

The **TrainingGraphCompiler** class has a function CompileGraphs() that will combine a number of graphs in a batch. This is used in the tool **compile-train-graphs** to speed up the graph compilation. The main reason it helps to do it in a batch is that when we create the H transducer, we only have to process each seen context-window once, even if it was used in many of the instances of CLG.

The reason that we can determinize without first adding disambiguation symbols is that HCLG in this case is functional (since any accepted input-label sequence is transuced to the same string) and acyclic; any acyclic FST has the twins property, and any functional FST with the twins property is determinizable.