Kaldi Decoding Analysis
(online2-wav-nnet3-latgen-faster-force)

Based on

commit e89280576107fcac7ad4d1b95eb8eaf8164bdccd

src/online2bin/online2-wav-nnet3-latgen-faster-force.cc

1

```
nnet3::DecodableNnetSimpleLoopedInfo decodable_info
  (decodable_opts,&am_nnet);
```

src/nnet3/decodable-simple-looped.cc

a

```
DecodableNnetSimpleLoopedInfo::DecodableNnetSimpleLoopedInfo
  (const NnetSimpleLoopedComputationOptions &opts, Nnet *nnet):
    opts(opts), nnet(*nnet) {
    Init(opts, nnet); }
void DecodableNnetSimpleLoopedInfo::Init
  (const NnetSimpleLoopedComputationOptions &opts, Nnet *nnet)
```

src/online2bin/online2-wav-nnet3-latgen-faster-force.cc

2

```
SingleUtteranceNnet3Decoder decoder
  (decoder_opts, trans_model,decodable_info,*decode_fst, &feature_pipeline);
```

src/online2/online-nnet3-decoding.h

a

```
SingleUtteranceNnet3Decoder
  (const LatticeFasterDecoderConfig &decoder_opts,
   const TransitionModel &trans_model,
   const nnet3::DecodableNnetSimpleLoopedInfo &info,
   const fst::Fst<fst::StdArc> &fst,
   OnlineNnet2FeaturePipeline *features);
nnet3::DecodableAmNnetLoopedOnline decodable_;
LatticeFasterOnlineDecoder decoder_;
```

src/nnet3/decodable-online-looped.h

c

```
class DecodableNnetLoopedOnline:
  public DecodableNnetLoopedOnlineBase {
   public:
    DecodableNnetLoopedOnline(
      const DecodableNnetSimpleLoopedInfo &info,
      OnlineFeatureInterface *input_features,
      OnlineFeatureInterface *ivector_features):
        DecodableNnetLoopedOnlineBase
          (info, input_features, ivector_features) { }
```

src/online2/online-nnet3-decoding.cc

b

```
SingleUtteranceNnet3Decoder::SingleUtteranceNnet3Decoder(
    const LatticeFasterDecoderConfig &decoder_opts,
    const TransitionModel &trans_model,
    const nnet3::DecodableNnetSimpleLoopedInfo &info,
    const fst::Fst<fst::StdArc> &fst,
    OnlineNnet2FeaturePipeline *features):
    decoder_opts_(decoder_opts),
    input_feature_frame_shift_in_seconds_(features->FrameShiftInSeconds()),
    trans_model_(trans_model),
    decodable_(trans_model_, info,
               features->InputFeature(), features->IvectorFeature()),
    decoder_(fst, decoder_opts_) {
  decoder_.InitDecoding();
}
```

src/decoder/lattice-faster-online-decoder.cc

d

```
LatticeFasterOnlineDecoder::LatticeFasterOnlineDecoder(
    const fst::Fst<fst::StdArc> &fst,
    const LatticeFasterDecoderConfig &config):
    fst_(fst), delete_fst_(false), config_(config), num_toks_(0) {
  config.Check();
  toks_.SetSize(1000);}
```

e

```
void LatticeFasterOnlineDecoder::InitDecoding() {
  // clean up from last time:
  DeleteElems(toks_.Clear());
  cost_offsets_.clear();
  ClearActiveTokens();
  warned_ = false;
  num_toks_ = 0;
  decoding_finalized_ = false;
  final_costs_.clear();
  StateId start_state = fst_.Start();
  KALDI_ASSERT(start_state != fst::kNoStateId);
  active_toks_.resize(1);
  Token *start_tok = new Token(0.0, 0.0, NULL, NULL, NULL);
  active_toks_[0].toks = start_tok;
  toks_.Insert(start_state, start_tok);
  num_toks_++;
  ProcessNonemitting(config_.beam);
}
```

3 ***src/online2bin/online2-wav-nnet3-latgen-faster-force.cc***

```
silence_weighting.ComputeCurrentTraceback(decoder.Decoder());
```

4 ***src/online2bin/online2-wav-nnet3-latgen-faster-force.cc***

```
while (samp_offset < data.Dim()) {...
    decoder.AdvanceDecoding();
...}
```

src/online2/online-nnet3-decoding.cc

a

```
void SingleUtteranceNnet3Decoder::AdvanceDecoding() {
  decoder_.AdvanceDecoding(&decodable_);
}
```

src/itf/decodable-itf.h

c

```
class DecodableInterface {
 public:
virtual BaseFloat LogLikelihood(int32 frame, int32 index) = 0;
```

src/nnet3/decodable-online-looped.cc

e

```
BaseFloat DecodableAmNnetLoopedOnline::LogLikelihood
  (int32 subsampled_frame, int32 index) {
    EnsureFrameIsComputed(subsampled_frame);
    return current_log_post_(
      subsampled_frame - current_log_post_subsampled_offset_,
      trans_model_.TransitionIdToPdf(index));
}
```

src/nnet3/decodable-online-looped.h

f

```
inline void EnsureFrameIsComputed(int32 subsampled_frame) {
  while (subsampled_frame >= current_log_post_subsampled_offset_ +
         current_log_post_.NumRows())
    AdvanceChunk();
}
...
  NnetComputer computer_;
...
  Matrix<BaseFloat> current_log_post_;
```

src/online2/online-nnet3-decoding.h

a

```
const LatticeFasterOnlineDecoder &Decoder() const { return decoder_; }
```

src/decoder/lattice-faster-online-decoder.cc

b

```
void LatticeFasterOnlineDecoder::AdvanceDecoding(DecodableInterface *decodable,int32 max_num_frames) {
...
  while (NumFramesDecoded() < target_frames_decoded) {
    if (NumFramesDecoded() % config_.prune_interval == 0) {
      PruneActiveTokens(config_.lattice_beam * config_.prune_scale);
    }
    // note: ProcessEmitting() increments NumFramesDecoded().
    BaseFloat cost_cutoff = ProcessEmitting(decodable);
    ProcessNonemitting(cost_cutoff);
  }
}
```

d

```
BaseFloat LatticeFasterOnlineDecoder::ProcessEmitting(DecodableInterface *decodable) {
...
  Elem *final_toks = toks_.Clear(); // analogous to swapping prev_toks_ / cur_toks_
...
  BaseFloat cur_cutoff = GetCutoff(final_toks, &tok_cnt, &adaptive_beam, &best_elem); ...
  for (Elem *e = final_toks, *e_tail; e != NULL; e = e_tail) {
    StateId state = e->key; Token *tok = e->val;
    if (tok->tot_cost <=  cur_cutoff) {
      for (fst::ArcIterator<fst::Fst<Arc> > aiter(fst_, state); !aiter.Done(); aiter.Next()) {
        const Arc &arc = aiter.Value();
        if (arc.ilabel != 0) {  // propagate..
          BaseFloat ac_cost = cost_offset - decodable->LogLikelihood(frame, arc.ilabel);
          graph_cost = arc.weight.Value(), cur_cost = tok->tot_cost, tot_cost = cur_cost + ac_cost + graph_cost;
          if (tot_cost > next_cutoff) continue;
          else if (tot_cost + adaptive_beam < next_cutoff)
            next_cutoff = tot_cost + adaptive_beam; // prune by best current token
          Token *next_tok = FindOrAddToken(arc.nextstate, frame + 1, tot_cost, tok, NULL);
          tok->links = new ForwardLink(next_tok, arc.ilabel, arc.olabel, graph_cost, ac_cost, tok->links);
        }
      } // for all arcs
    }
    e_tail = e->tail; toks_.Delete(e); // delete Elem
  }
  return next_cutoff;
}
```

src/nnet3/decodable-online-looped.cc

g

```
void DecodableNnetLoopedOnlineBase::AdvanceChunk() {
...
  CuMatrix<BaseFloat> feats_chunk;
  { // this block sets 'feats_chunk'.
    for (int32 i = begin_input_frame; i < end_input_frame; i++) {
...
      input_features_->GetFrame(input_frame, &this_row);
    }
    feats_chunk.Swap(&this_feats);
  }
  computer_.AcceptInput("input", &feats_chunk);
  if (info_.has_ivectors) {
...
    if (num_ivector_frames_ready > 0) {
...
      ivector_features_->GetFrame(ivector_frame_to_use, &ivector);
    }
...
    CuMatrix<BaseFloat> cu_ivectors;
    cu_ivectors.Swap(&ivectors);
    computer_.AcceptInput("ivector", &cu_ivectors);
  }
  computer_.Run();
  {
    CuMatrix<BaseFloat> output;
    computer_.GetOutputDestructive("output", &output);

    if (info_.log_priors.Dim() != 0) {
      // subtract log-prior (divide by prior)
      output.AddVecToRows(-1.0, info_.log_priors);
    }
    // apply the acoustic scale
    output.Scale(info_.opts.acoustic_scale);
    current_log_post_.Resize(0, 0);
    current_log_post_.Swap(&output);
  }
...
}
```

My model gives:
current_log_post_.NumRows(): 7
current_log_post_.NumCols(): 9470

src/nnet3/nnet-compute.cc

h

```
void NnetComputer::AcceptInput(const std::string &node_name, CuMatrix<BaseFloat> *input) {...}
void NnetComputer::GetOutputDestructive(const std::string &node_name, CuMatrix<BaseFloat> *output) {...}
void NnetComputer::Run() {
...
  for (; program_counter_ < num_commands; program_counter_++) {
...
    ExecuteCommand();
...
  }
}
void NnetComputer::ExecuteCommand() {
  const NnetComputation::Command &c = computation_.commands[program_counter_];
  int32 m1, m2;
  try {
    switch (c.command_type) {
...
      case kPropagate: {
        const Component *component = nnet_.GetComponent(c.arg1);
        ComponentPrecomputedIndexes *indexes =
            computation_.component_precomputed_indexes[c.arg2].data;
        const CuSubMatrix<BaseFloat> input(GetSubMatrix(c.arg3));
        CuSubMatrix<BaseFloat> output(GetSubMatrix(c.arg4));
        void *memo = component->Propagate(indexes, input, &output);
        if (c.arg6) {  // need to store stats.
          Component *upd_component = nnet_to_update_->GetComponent(c.arg1);
          bool was_in_place = (c.arg3 == c.arg4);
          const CuSubMatrix<BaseFloat> maybe_input(
              GetSubMatrix(was_in_place ? 0 : c.arg3));
          upd_component->StoreStats(maybe_input, output, memo);
        }
        SaveMemo(c.arg5, *component, memo);
        break;
      }
...
}
```

i *src/decoder/lattice-faster-online-decoder.cc*

```
inline LatticeFasterOnlineDecoder::Token *LatticeFasterOnlineDecoder::FindOrAddToken(
    StateId state, int32 frame_plus_one, BaseFloat tot_cost,
    Token *backpointer, bool *changed) {
  // Returns the Token pointer.  Sets "changed" (if non-NULL) to true
  // if the token was newly created or the cost changed.
  KALDI_ASSERT(frame_plus_one < active_toks_.size());
  Token *&toks = active_toks_[frame_plus_one].toks;
  Elem *e_found = toks_.Find(state);
  if (e_found == NULL) {  // no such token presently.
    const BaseFloat extra_cost = 0.0;
    Token *new_tok = new Token (tot_cost, extra_cost, NULL, toks, backpointer);
    toks = new_tok; num_toks_++;
    toks_.Insert(state, new_tok);
    if (changed) *changed = true;
    return new_tok;
  } else {
    Token *tok = e_found->val;  // There is an existing Token for this state.
    if (tok->tot_cost > tot_cost) {  // replace old token
      tok->tot_cost = tot_cost; tok->backpointer = backpointer;
      if (changed) *changed = true;
    } else {
      if (changed) *changed = false;
    }
    return tok;
  }
}
```

j *src/decoder/lattice-faster-online-decoder.h*

```
struct ForwardLink {
  Token *next_tok; // the next token [or NULL if represents final-state]
  Label ilabel; // ilabel on link.
  Label olabel; // olabel on link.
  BaseFloat graph_cost; // graph cost of traversing link (contains LM, etc.)
  BaseFloat acoustic_cost; // acoustic cost (pre-scaled) of traversing link
  ForwardLink *next; // next in singly-linked list of forward links from a
                     // token.
  inline ForwardLink(Token *next_tok, Label ilabel, Label olabel,
                     BaseFloat graph_cost, BaseFloat acoustic_cost,
                     ForwardLink *next):
      next_tok(next_tok), ilabel(ilabel), olabel(olabel),
      graph_cost(graph_cost), acoustic_cost(acoustic_cost),
      next(next) { }
};

struct Token {
  BaseFloat tot_cost; // would equal weight.Value()... cost up to this point.
  BaseFloat extra_cost; // >= 0.  After calling PruneForwardLinks, this equals

  ForwardLink *links; // Head of singly linked list of ForwardLinks
  Token *next; // Next in list of tokens for this frame.
  Token *backpointer;

  inline Token(BaseFloat tot_cost, BaseFloat extra_cost, ForwardLink *links,
               Token *next, Token *backpointer):
      tot_cost(tot_cost), extra_cost(extra_cost), links(links), next(next),
      backpointer(backpointer) { }
  inline void DeleteForwardLinks() {
    ForwardLink *l = links, *m;
    while (l != NULL) {
      m = l->next;
      delete l;
      l = m;
    }
    links = NULL;
  }
};
```

5 ***src/online2bin/online2-wav-nnet3-latgen-faster-force.cc***

```
while (samp_offset < data.Dim()) {...
    if (do_endpointing && decoder.EndpointDetected(endpoint_opts))
        break;
...}
```

a ***src/online2/online-nnet3-decoding.cc***

```
bool SingleUtteranceNnet3Decoder::EndpointDetected(
    const OnlineEndpointConfig &config) {
  BaseFloat output_frame_shift =
      input_feature_frame_shift_in_seconds_ * decodable_.FrameSubsamplingFactor();
  return kaldi::EndpointDetected(config, trans_model_, output_frame_shift, decoder_);
}
```

6 ***src/online2bin/online2-wav-nnet3-latgen-faster-force.cc***

```
decoder.FinalizeDecoding();
```

a ***src/online2/online-nnet3-decoding.cc***

```
void SingleUtteranceNnet3Decoder::FinalizeDecoding() {
  decoder_.FinalizeDecoding();
}
```

b ***src/decoder/lattice-faster-online-decoder.cc***

```
void LatticeFasterOnlineDecoder::FinalizeDecoding() {
  int32 final_frame_plus_one = NumFramesDecoded();
  int32 num_toks_begin = num_toks_;
  // PruneForwardLinksFinal() prunes final frame (with final-probs), and
  // sets decoding_finalized_.
  PruneForwardLinksFinal();
  for (int32 f = final_frame_plus_one - 1; f >= 0; f--) {
    bool b1, b2; // values not used.
    BaseFloat dontcare = 0.0; // delta of zero means we must always update
    PruneForwardLinks(f, &b1, &b2, dontcare);
    PruneTokensForFrame(f + 1);
  }
  PruneTokensForFrame(0);
  KALDI_VLOG(4) << "pruned tokens from " << num_toks_begin
                << " to " << num_toks_;
}
```

7 ***src/online2bin/online2-wav-nnet3-latgen-faster-force.cc***

```
CompactLattice clat;
bool end_of_utterance = true;
decoder.GetLattice(end_of_utterance, &clat);
```

a ***src/lat/kaldi-lattice.h***

```
typedef fst::VectorFst<LatticeArc> Lattice;
typedef fst::VectorFst<CompactLatticeArc> CompactLattice;
```

b ***src/online2/online-nnet3-decoding.cc***

```
void SingleUtteranceNnet3Decoder::GetLattice(bool end_of_utterance, CompactLattice *clat) const {
  Lattice raw_lat;
  decoder_.GetRawLattice(&raw_lat, end_of_utterance);

  BaseFloat lat_beam = decoder_opts_.lattice_beam;
  DeterminizeLatticePhonePrunedWrapper(trans_model_, &raw_lat, lat_beam, clat, decoder_opts_.det_opts);
}
```

src/decoder/lattice-faster-online-decoder.cc

```
bool LatticeFasterOnlineDecoder::GetRawLattice(Lattice *ofst, bool use_final_probs) const {
  typedef LatticeArc Arc, typedef Arc::StateId StateId, typedef Arc::Weight Weight, typedef Arc::Label Label;
  unordered_map<Token*, BaseFloat> final_costs_local;
  const unordered_map<Token*, BaseFloat> &final_costs = (decoding_finalized_ ? final_costs_ : final_costs_local);
  if (!decoding_finalized_ && use_final_probs) ComputeFinalCosts(&final_costs_local, NULL, NULL);
  ofst->DeleteStates();
  int32 num_frames = active_toks_.size() - 1;
  const int32 bucket_count = num_toks_/2 + 3;
  unordered_map<Token*, StateId> tok_map(bucket_count);
  // First create all states.
  std::vector<Token*> token_list;
  for (int32 f = 0; f <= num_frames; f++) {
    if (active_toks_[f].toks == NULL) {
      KALDI_WARN << "GetRawLattice: no tokens active on frame " << f << ": not producing lattice.₩n";       return false;     }
    TopSortTokens(active_toks_[f].toks, &token_list); for (size_t i = 0; i < token_list.size(); i++) if (token_list[i] != NULL) tok_map[token_list[i]] = ofst->AddState();
  }
  // The next statement sets the start state of the output FST.  Because we topologically sorted the tokens, state zero must be the start-state.
  ofst->SetStart(0);
  KALDI_VLOG(4) << "init:" << num_toks_/2 + 3 << " buckets:" << tok_map.bucket_count() << " load:" << tok_map.load_factor() << " max:" << tok_map.max_load_factor();
  // Now create all arcs.
  for (int32 f = 0; f <= num_frames; f++) {
    for (Token *tok = active_toks_[f].toks; tok != NULL; tok = tok->next) {
      StateId cur_state = tok_map[tok];
      for (ForwardLink *l = tok->links; l != NULL; l = l->next) {
        unordered_map<Token*, StateId>::const_iterator iter = tok_map.find(l->next_tok);
        StateId nextstate = iter->second; KALDI_ASSERT(iter != tok_map.end()); BaseFloat cost_offset = 0.0;
        if (l->ilabel != 0) {  // emitting..
          KALDI_ASSERT(f >= 0 && f < cost_offsets_.size()); cost_offset = cost_offsets_[f];
        }
        Arc arc(l->ilabel, l->olabel, Weight(l->graph_cost, l->acoustic_cost - cost_offset), nextstate);
        ofst->AddArc(cur_state, arc);
      }
      if (f == num_frames) {
        if (use_final_probs && !final_costs.empty()) {
          unordered_map<Token*, BaseFloat>::const_iterator iter = final_costs.find(tok);
          if (iter != final_costs.end()) ofst->SetFinal(cur_state, LatticeWeight(iter->second, 0));
        } else {
          ofst->SetFinal(cur_state, LatticeWeight::One());
        }       }     } }  return (ofst->NumStates() > 0);}
```

src/decoder/lattice-faster-online-decoder.cc

```
void LatticeFasterOnlineDecoder::TopSortTokens(Token *tok_list,
                                               std::vector<Token*> *topsorted_list) {
  unordered_map<Token*, int32> token2pos;
  typedef unordered_map<Token*, int32>::iterator IterType;
  int32 num_toks = 0;
  for (Token *tok = tok_list; tok != NULL; tok = tok->next)
    num_toks++;
  int32 cur_pos = 0;
  // We assign the tokens numbers num_toks - 1, ... , 2, 1, 0.
  // This is likely to be in closer to topological order than
  // if we had given them ascending order, because of the way
  // new tokens are put at the front of the list.
  for (Token *tok = tok_list; tok != NULL; tok = tok->next)
    token2pos[tok] = num_toks - ++cur_pos;

  unordered_set<Token*> reprocess;

  for (IterType iter = token2pos.begin(); iter != token2pos.end(); ++iter) {
    Token *tok = iter->first;
    int32 pos = iter->second;
    for (ForwardLink *link = tok->links; link != NULL; link = link->next) {
      if (link->ilabel == 0) {
        // We only need to consider epsilon links, since non-epsilon links
        // transition between frames and this function only needs to sort a list
        // of tokens from a single frame.
        IterType following_iter = token2pos.find(link->next_tok);
        if (following_iter != token2pos.end()) { // another token on this frame,
                                                 // so must consider it.
          int32 next_pos = following_iter->second;
          if (next_pos < pos) { // reassign the position of the next Token.
            following_iter->second = cur_pos++;
            reprocess.insert(link->next_tok);
          }
        }
      }
    }
    // In case we had previously assigned this token to be reprocessed, we can
    // erase it from that set because it's "happy now" (we just processed it).
    reprocess.erase(tok);
  }
```

```
  size_t max_loop = 1000000, loop_count; // max_loop is to detect epsilon cycles.
  for (loop_count = 0;
       !reprocess.empty() && loop_count < max_loop; ++loop_count) {
    std::vector<Token*> reprocess_vec;
    for (unordered_set<Token*>::iterator iter = reprocess.begin();
         iter != reprocess.end(); ++iter)
      reprocess_vec.push_back(*iter);
    reprocess.clear();
    for (std::vector<Token*>::iterator iter = reprocess_vec.begin();
         iter != reprocess_vec.end(); ++iter) {
      Token *tok = *iter;
      int32 pos = token2pos[tok];
      // Repeat the processing we did above (for comments, see above).
      for (ForwardLink *link = tok->links; link != NULL; link = link->next) {
        if (link->ilabel == 0) {
          IterType following_iter = token2pos.find(link->next_tok);
          if (following_iter != token2pos.end()) {
            int32 next_pos = following_iter->second;
            if (next_pos < pos) {
              following_iter->second = cur_pos++;
              reprocess.insert(link->next_tok);
            }
          }
        }
      }
    }
  }
  KALDI_ASSERT(loop_count < max_loop && "Epsilon loops exist in your decoding "
               "graph (this is not allowed!)");

  topsorted_list->clear();
  topsorted_list->resize(cur_pos, NULL);  // create a list with NULLs in between.
  for (IterType iter = token2pos.begin(); iter != token2pos.end(); ++iter)
    (*topsorted_list)[iter->second] = iter->first;
}
```

src/lat/determinize-lattice-pruned.h

e

```
bool DeterminizeLatticePhonePrunedWrapper(
    const kaldi::TransitionModel &trans_model,
    MutableFst<kaldi::LatticeArc> *ifst,
    double prune,
    MutableFst<kaldi::CompactLatticeArc> *ofst,
    DeterminizeLatticePhonePrunedOptions opts
      = DeterminizeLatticePhonePrunedOptions());

struct DeterminizeLatticePhonePrunedOptions {
  // delta: a small offset used to measure equality of weights.
  float delta;
  // max_mem: if > 0, determinization will fail and return false when the
  // algorithm's (approximate) memory consumption crosses this threshold.
  int max_mem;
  // phone_determinize: if true, do a first pass determinization on both phones
  // and words.
  bool phone_determinize;
  // word_determinize: if true, do a second pass determinization on words only.
  bool word_determinize;
  // minimize: if true, push and minimize after determinization.
  bool minimize;
  DeterminizeLatticePhonePrunedOptions(): delta(kDelta),
                                          max_mem(50000000),
                                          phone_determinize(true),
                                          word_determinize(true),
                                          minimize(false) {}
  void Register (kaldi::OptionsItf *opts) {
    opts->Register("delta", &delta, "Tolerance used in determinization");
    opts->Register("max-mem", &max_mem, "Maximum approximate memory usage in "
                   "determinization (real usage might be many times this).");
    opts->Register("phone-determinize", &phone_determinize, "If true, do an "
                   "initial pass of determinization on both phones and words (see"
                   " also --word-determinize)");
    opts->Register("word-determinize", &word_determinize, "If true, do a second "
                   "pass of determinization on words only (see also "
                   "--phone-determinize)");
    opts->Register("minimize", &minimize, "If true, push and minimize after "
                   "determinization.");
  }
};
```

8 ***src/online2bin/online2-wav-nnet3-latgen-faster-force.cc***

```
fst::ScaleLattice(fst::LatticeScale(lm_scale, acoustic_scale), &clat);
kaldi::TopSortCompactLatticeIfNeeded(&clat);
```

src/fstext/lattice-utils.h

a

```
inline vector<vector<double> > LatticeScale(double lmwt, double acwt) {
  vector<vector<double> > ans(2);
  ans[0].resize(2, 0.0);
  ans[1].resize(2, 0.0);
  ans[0][0] = lmwt;
  ans[1][1] = acwt;
  return ans;
}
```

src/fstext/lattice-utils-inl.h

b

```
template<class Weight, class ScaleFloat>
void ScaleLattice(
    const vector<vector<ScaleFloat> > &scale,
    MutableFst<ArcTpl<Weight> > *fst) {
  assert(scale.size() == 2 && scale[0].size() == 2 && scale[1].size() == 2);
  if (scale == DefaultLatticeScale()) // nothing to do.
    return;
  typedef ArcTpl<Weight> Arc;
  typedef MutableFst<Arc> Fst;
  typedef typename Arc::StateId StateId;
  StateId num_states = fst->NumStates();
  for (StateId s = 0; s < num_states; s++) {
    for (MutableArcIterator<Fst> aiter(fst, s);
         !aiter.Done();
         aiter.Next()) {
      Arc arc = aiter.Value();
      arc.weight = Weight(ScaleTupleWeight(arc.weight, scale));
      aiter.SetValue(arc);
    }
    Weight final_weight = fst->Final(s);
    if (final_weight != Weight::Zero())
      fst->SetFinal(s, Weight(ScaleTupleWeight(final_weight, scale)));
  }
}
```

src/lat/lattice-functions.cc

c

```
void TopSortCompactLatticeIfNeeded(CompactLattice *clat) {
  if (clat->Properties(fst::kTopSorted, true) == 0) {
    if (fst::TopSort(clat) == false) {
      KALDI_ERR << "Topological sorting failed";
    }
  }
}
```