Kaldi LatticeFasterDecoder 代码解析+逐步实例

https://github.com/kaldi-asr/kaldi/blob/master/src/decoder/lattice-faster-decoder.cc

https://github.com/kaldi-asr/kaldi/blob/master/src/decoder/lattice-faster-decoder.h

LatticeFasterDecoder是一个模版类

1. LatticeFasterDecoder是一个模版类，具有更强的通用性和扩展性，支持不同的FST和Token类型的组合。ForwarLink也是使用模版定义的，支持不同类型的Token。
2. 代码最后给出了所有可能的FST类型和Token类型的组合，用于不同场景的解码
3. 代码里使用了FST的内存管理，使用Memory Pool来新建和管理内存

```
231     The decoder is templated on the FST type and the token type.  The token type
232     will normally be StdToken, but also may be BackpointerToken which is to support
233     quick lookup of the current best path (see lattice-faster-online-decoder.h)
234
235     The FST you invoke this decoder which is expected to equal
236     Fst::Fst<fst::StdArc>, a.k.a. StdFst, or GrammarFst.  If you invoke it with
237     FST == StdFst and it notices that the actual FST type is
238     fst::VectorFst<fst::StdArc> or fst::ConstFst<fst::StdArc>, the decoder object
239     will internally cast itself to one that is templated on those more specific
240     types; this is an optimization for speed.
241    */
242   template <typename FST, typename Token = decoder::StdToken>
243   class LatticeFasterDecoderTpl {
244    public:
```

```
116  // ForwardLinks are the links from a token to a token on the next frame.
117  // or sometimes on the current frame (for input-epsilon links).
118  template <typename Token>
119  struct ForwardLink {
120    using Label = fst::StdArc::Label;
121
122    Token *next_tok;  // the next token [or NULL if represents final-state]
123    Label ilabel;  // ilabel on arc
124    Label olabel;  // olabel on arc
125    BaseFloat graph_cost;  // graph cost of traversing arc (contains LM, etc.)
126    BaseFloat acoustic_cost;  // acoustic cost (pre-scaled) of traversing arc
127    ForwardLink *next;  // next in singly-linked list of forward arcs (arcs
128                        // in the state-level lattice) from a token.
129    inline ForwardLink(Token *next_tok, Label ilabel, Label olabel,
130                       BaseFloat graph_cost, BaseFloat acoustic_cost,
```

```
1004  // Instantiate the template for the combination of token types and FST types
1005  // that we'll need.
1006  template class LatticeFasterDecoderTpl<fst::Fst<fst::StdArc>, decoder::StdToken>;
1007  template class LatticeFasterDecoderTpl<fst::VectorFst<fst::StdArc>, decoder::StdToken >;
1008  template class LatticeFasterDecoderTpl<fst::ConstFst<fst::StdArc>, decoder::StdToken >;
1009
1010  template class LatticeFasterDecoderTpl<fst::ConstGrammarFst, decoder::StdToken>;
1011  template class LatticeFasterDecoderTpl<fst::VectorGrammarFst, decoder::StdToken>;
1012
1013  template class LatticeFasterDecoderTpl<fst::Fst<fst::StdArc> , decoder::BackpointerToken>;
1014  template class LatticeFasterDecoderTpl<fst::VectorFst<fst::StdArc>, decoder::BackpointerToken >;
1015  template class LatticeFasterDecoderTpl<fst::ConstFst<fst::StdArc>, decoder::BackpointerToken >;
1016  template class LatticeFasterDecoderTpl<fst::ConstGrammarFst, decoder::BackpointerToken>;
1017  template class LatticeFasterDecoderTpl<fst::VectorGrammarFst, decoder::BackpointerToken>;
1018
```

LatticeFasterDecoder的整体流程

1. 整体流程和LatticeSimpleDecoder是非常相似的，只不过核心的解码过程在AdvanceDecoding函数中进行的
2. Lattice剪枝的算法，和LatticeSimpleDecoder是相同的，Token剪枝的算法，则和FasterDecoder中的自适应剪枝策略相同，调用GetCutoff计算剪枝相关的阈值，并且自适应的改变beam参数。

```
588 template <typename FST, typename Token>
589 void LatticeFasterDecoderTpl<FST, Token>::AdvanceDecoding(DecodableInterface *decodable,
590                                                           int32 max_num_frames) {
591   if (std::is_same<FST, fst::Fst<fst::StdArc> >::value) {
592     // if the type 'FST' is the FST base-class, then see if the FST type of fst_
593     // is actually VectorFst or ConstFst.  If so, call the AdvanceDecoding()
594     // function after casting *this to the more specific type.
595     if (fst_->Type() == "const") {
596       LatticeFasterDecoderTpl<fst::ConstFst<fst::StdArc>, Token> *this_cast =
597           reinterpret_cast<LatticeFasterDecoderTpl<fst::ConstFst<fst::StdArc>, Token>* >(this);
598       this_cast->AdvanceDecoding(decodable, max_num_frames);
599       return;
600     } else if (fst_->Type() == "vector") {
601       LatticeFasterDecoderTpl<fst::VectorFst<fst::StdArc>, Token> *this_cast =
602           reinterpret_cast<LatticeFasterDecoderTpl<fst::VectorFst<fst::StdArc>, Token>* >(this);
603       this_cast->AdvanceDecoding(decodable, max_num_frames);
604       return;
605     }
606   }
607
608
609   KALDI_ASSERT(!active_toks_.empty() && !decoding_finalized_ &&
610                "You must call InitDecoding() before AdvanceDecoding");
611   int32 num_frames_ready = decodable->NumFramesReady();
612   // num_frames_ready must be >= num_frames_decoded, or else
613   // the number of frames ready must have decreased (which doesn't
614   // make sense) or the decodable object changed between calls
615   // (which isn't allowed).
616   KALDI_ASSERT(num_frames_ready >= NumFramesDecoded());
617   int32 target_frames_decoded = num_frames_ready;
618   if (max_num_frames >= 0)
619     target_frames_decoded = std::min(target_frames_decoded,
620                                      NumFramesDecoded() + max_num_frames);
621   while (NumFramesDecoded() < target_frames_decoded) {
622     if (NumFramesDecoded() % config_.prune_interval == 0) {
623       PruneActiveTokens(config_.lattice_beam * config_.prune_scale);
624     }
625     BaseFloat cost_cutoff = ProcessEmitting(decodable);
626     ProcessNonemitting(cost_cutoff);
627   }
628 }
```

首先根据FST的类型，使用reinterpret_cast重新诠释转型之后再调用AdvanceDecoding方法，然后执行608之后的代码

```
83  // Returns true if any kind of traceback is available (not necessarily from
84  // a final state).  It should only very rarely return false; this indicates
85  // an unusual search error.
86  template <typename FST, typename Token>
87  bool LatticeFasterDecoderTpl<FST, Token>::Decode(DecodableInterface *decodable) {
88    InitDecoding();
89    // We use 1-based indexing for frames in this decoder (if you view it in
90    // terms of features), but note that the decodable object uses zero-based
91    // numbering, which we have to correct for when we call it.
92    AdvanceDecoding(decodable);
93    FinalizeDecoding();
94
95    // Returns true if we have any kind of traceback available (not necessarily
96    // to the end state; query ReachedFinal() for that).
97    return !active_toks_.empty() && active_toks_.back().toks != NULL;
98  }
```

ProcessEmitting方法的细节变化

```
723  BaseFloat LatticeFasterDecoderTpl<FST, Token>::ProcessEmitting(
724      DecodableInterface *decodable) {
725    KALDI_ASSERT(active_toks_.size() > 0);
726    int32 frame = active_toks_.size() - 1; // frame is the frame-index
727                                           // (zero-based) used to get likelihoods
728                                           // from the decodable object.
729    active_toks_.resize(active_toks_.size() + 1);
730
731    Elem *final_toks = toks_.Clear(); // analogous to swapping prev_toks_ / cur_toks_
732                                      // in simple-decoder.h.   Removes the Elems from
733                                      // being indexed in the hash in toks_.
734    Elem *best_elem = NULL;
735    BaseFloat adaptive_beam;
736    size_t tok_cnt;
737    BaseFloat cur_cutoff = GetCutoff(final_toks, &tok_cnt, &adaptive_beam, &best_elem);
738    KALDI_VLOG(6) << "Adaptive beam on frame " << NumFramesDecoded() << " is "
739                  << adaptive_beam;
740
741    PossiblyResizeHash(tok_cnt);  // This makes sure the hash is always big enough.
742
743    BaseFloat next_cutoff = std::numeric_limits<BaseFloat>::infinity();
744    // pruning "online" before having seen all tokens
745
746    BaseFloat cost_offset = 0.0; // Used to keep probabilities in a good
747                                 // dynamic range.
748
749
750    // First process the best token to get a hopefully
751    // reasonably tight bound on the next cutoff.  The only
752    // products of the next block are "next_cutoff" and "cost_offset".
753    if (best_elem) {
754      StateId state = best_elem->key;
755      Token *tok = best_elem->val;
756      cost_offset = - tok->tot_cost;
757      for (fst::ArcIterator<FST> aiter(*fst_, state);
758           !aiter.Done();
759           aiter.Next()) {
760        const Arc &arc = aiter.Value();
761        if (arc.ilabel != 0) {  // propagate..
762          BaseFloat new_weight = arc.weight.Value() + cost_offset -
763              decodable->LogLikelihood(frame, arc.ilabel) + tok->tot_cost;
764          if (new_weight + adaptive_beam < next_cutoff)
765            next_cutoff = new_weight + adaptive_beam;
766        }
767      }
```

增加了一个costs_offset变量，它等于每一帧最优token损失的相反数，770行注释对其进行了解释，它作于声学得分(789,799),每一帧的cost_offset都会存下，在生成Lattice的时候再减去

```
770    // Store the offset on the acoustic likelihoods that we're applying.
771    // Could just do cost_offsets_.push_back(cost_offset), but we
772    // do it this way as it's more robust to future code changes.
773    cost_offsets_.resize(frame + 1, 0.0);
774    cost_offsets_[frame] = cost_offset;
775
776    // the tokens are now owned here, in final_toks, and the hash is empty.
777    // 'owned' is a complex thing here; the point is we need to call DeleteElem
778    // on each elem 'e' to let toks_ know we're done with them.
779    for (Elem *e = final_toks, *e_tail; e != NULL; e = e_tail) {
780      // loop this way because we delete "e" as we go.
781      StateId state = e->key;
782      Token *tok = e->val;
783      if (tok->tot_cost <= cur_cutoff) {
784        for (fst::ArcIterator<FST> aiter(*fst_, state);
785             !aiter.Done();
786             aiter.Next()) {
787          const Arc &arc = aiter.Value();
788          if (arc.ilabel != 0) {  // propagate..
789            BaseFloat ac_cost = cost_offset -
790                decodable->LogLikelihood(frame, arc.ilabel),
791                graph_cost = arc.weight.Value(),
792                cur_cost = tok->tot_cost,
793                tot_cost = cur_cost + ac_cost + graph_cost;
794            if (tot_cost >= next_cutoff) continue;
795            else if (tot_cost + adaptive_beam < next_cutoff)
796              next_cutoff = tot_cost + adaptive_beam; // prune by best current token
797            // Note: the frame indexes into active_toks_ are one-based,
798            // hence the + 1.
799            Elem *e_next = FindOrAddToken(arc.nextstate,
800                                          frame + 1, tot_cost, tok, NULL);
801            // NULL: no change indicator needed
802
803            // Add ForwardLink from tok to next_tok (put on head of list tok->links)
804            tok->links = new (forward_link_pool_.Allocate())
805                ForwardLinkT(e_next->val, arc.ilabel, arc.olabel, graph_cost,
806                             ac_cost, tok->links);
807          }
808        } // for all arcs
809      }
810      e_tail = e->tail;
811      toks_.Delete(e); // delete Elem
812    }
813    return next_cutoff;
```

使用Allocate（）方法进行内存开辟管理了

PruneActiveTokens方法

```
514  template <typename FST, typename Token>
515  void LatticeFasterDecoderTpl<FST, Token>::PruneActiveTokens(BaseFloat delta) {
516    int32 cur_frame_plus_one = NumFramesDecoded();
517    int32 num_toks_begin = num_toks_;
518    // The index "f" below represents a "frame plus one", i.e. you'd have to subtract
519    // one to get the corresponding index for the decodable object.
520    for (int32 f = cur_frame_plus_one - 1; f >= 0; f--) {
521      // Reason why we need to prune forward links in this situation:
522      // (1) we have never pruned them (new TokenList)
523      // (2) we have not yet pruned the forward links to the next f,
524      // after any of those tokens have changed their extra_cost.
525      if (active_toks_[f].must_prune_forward_links) {
526        bool extra_costs_changed = false, links_pruned = false;
527        PruneForwardLinks(f, &extra_costs_changed, &links_pruned, delta);
528        if (extra_costs_changed && f > 0) // any token has changed extra_cost
529          active_toks_[f-1].must_prune_forward_links = true;
530        if (links_pruned) // any link was pruned
531          active_toks_[f].must_prune_tokens = true;
532        active_toks_[f].must_prune_forward_links = false; // job done
533      }
534      if (f+1 < cur_frame_plus_one &&      // except for last f (no forward links)
535          active_toks_[f+1].must_prune_tokens) {
536        PruneTokensForFrame(f+1);
537        active_toks_[f+1].must_prune_tokens = false;
538      }
539    }
540    KALDI_VLOG(4) << "PruneActiveTokens: pruned tokens from " << num_toks_begin
541                  << " to " << num_toks_;
542  }
```

解码过程中，Lattice核心的剪枝过程和LatticeSimpleDecoder相同，都是周期性的调用PruneActiveTokens方法进行Lattice的剪枝：

1. 首先调用PruneForwardLinks对前向弧剪枝
2. 然后调用哦PruneTokensForFrame进行Token的剪枝；

少有不同的也是一些细节：

```
487  template <typename FST, typename Token>
488  void LatticeFasterDecoderTpl<FST, Token>::PruneTokensForFrame(int32 frame_plus_one) {
489    KALDI_ASSERT(frame_plus_one >= 0 && frame_plus_one < active_toks_.size());
490    Token *&toks = active_toks_[frame_plus_one].toks;
491    if (toks == NULL)
492      KALDI_WARN << "No tokens alive [doing pruning]";
493    Token *tok, *next_tok, *prev_tok = NULL;
494    for (tok = toks; tok != NULL; tok = next_tok) {
495      next_tok = tok->next;
496      if (tok->extra_cost == std::numeric_limits<BaseFloat>::infinity()) {
497        // token is unreachable from end of graph; (no forward links survived)
498        // excise tok from list and delete tok.
499        if (prev_tok != NULL) prev_tok->next = tok->next;
500        else toks = tok->next;
501        token_pool_.Free(tok);
502        num_toks_--;
503      } else {  // fetch next Token
504        prev_tok = tok;
505      }
506    }
507  }
```

```
348          if (link_extra_cost > config_.lattice_beam) {  // excise link
349            ForwardLinkT *next_link = link->next;
350            if (prev_link != NULL) prev_link->next = next_link;
351            else tok->links = next_link;
352            forward_link_pool_.Free(link);
353            link = next_link;  // advance link but leave prev_link the same.
354            *links_pruned = true;
355          } else {   // keep the link and update the tok_extra_cost if needed.
```

一些数据清理和内存清理方法的不同

1. 在最后一帧的前向弧剪枝中，使用DeleteElems方法，把当前帧的Elem标志为未被使用的状态。
2. ClearActiveTokens里面，调用DeteteForwardLinks来删除每个Token的ForwarLink，在LatticeSimpleDecoder代码里，前向弧删除是写在Token结构题里面的方法

```
384  template <typename FST, typename Token>
385  void LatticeFasterDecoderTpl<FST, Token>::PruneForwardLinksFinal() {
386    KALDI_ASSERT(!active_toks_.empty());
387    int32 frame_plus_one = active_toks_.size() - 1;
388
389    if (active_toks_[frame_plus_one].toks == NULL)  // empty list; should not happen.
390      KALDI_WARN << "No tokens alive at end of file";
391
392    typedef typename unordered_map<Token*, BaseFloat>::const_iterator IterType;
393    ComputeFinalCosts(&final_costs_, &final_relative_cost_, &final_best_cost_);
394    decoding_finalized_ = true;
395    // We call DeleteElems() as a nicety, not because it's really necessary;
396    // otherwise there would be a time, after calling PruneTokensForFrame() on the
397    // final frame, when toks_.GetList() or toks_.Clear() would contain pointers
398    // to nonexistent tokens.
399    DeleteElems(toks_.Clear());
400
401    // Now go through tokens on this frame, pruning forward links...  may have to
```

```
818  void LatticeFasterDecoderTpl<FST, Token>::DeleteForwardLinks(Token *tok) {
819    ForwardLinkT *l = tok->links, *m;
820    while (l != NULL) {
821      m = l->next;
822      forward_link_pool_.Free(l);
823      l = m;
824    }
825    tok->links = NULL;
826  }
```

```
900  template <typename FST, typename Token>
901  void LatticeFasterDecoderTpl<FST, Token>::DeleteElems(Elem *list) {
902    for (Elem *e = list, *e_tail; e != NULL; e = e_tail) {
903      e_tail = e->tail;
904      toks_.Delete(e);
905    }
906  }
907
```

调用HashList的Delete方法，将这么Elem标志为未被使用的状态

```
908  template <typename FST, typename Token>
909  void LatticeFasterDecoderTpl<FST, Token>::ClearActiveTokens() { // a cleanup routine, at utt end/begin
910    for (size_t i = 0; i < active_toks_.size(); i++) {
911      // Delete all tokens alive on this frame, and any forward
912      // links they may have.
913      for (Token *tok = active_toks_[i].toks; tok != NULL; ) {
914        DeleteForwardLinks(tok);
915        Token *next_tok = tok->next;
916        token_pool_.Free(tok);
917        num_toks_--;
918        tok = next_tok;
919      }
920    }
921    active_toks_.clear();
922    KALDI_ASSERT(num_toks_ == 0);
923  }
```

一些数据清理和内存清理方法的不同

1. 在最后一帧的前向弧剪枝中，使用DeleteElems方法，把当前帧的Elem标志为未被使用的状态。
2. ClearActiveTokens里面，调用DeteteForwardLinks来删除每个Token的ForwarLink，在LatticeSimpleDecoder代码里，前向弧删除是写在Token结构题里面的方法

```
384  template <typename FST, typename Token>
385  void LatticeFasterDecoderTpl<FST, Token>::PruneForwardLinksFinal() {
386    KALDI_ASSERT(!active_toks_.empty());
387    int32 frame_plus_one = active_toks_.size() - 1;
388
389    if (active_toks_[frame_plus_one].toks == NULL)  // empty list; should not happen.
390      KALDI_WARN << "No tokens alive at end of file";
391
392    typedef typename unordered_map<Token*, BaseFloat>::const_iterator IterType;
393    ComputeFinalCosts(&final_costs_, &final_relative_cost_, &final_best_cost_);
394    decoding_finalized_ = true;
395    // We call DeleteElems() as a nicety, not because it's really necessary;
396    // otherwise there would be a time, after calling PruneTokensForFrame() on the
397    // final frame, when toks_.GetList() or toks_.Clear() would contain pointers
398    // to nonexistent tokens.
399    DeleteElems(toks_.Clear());
400
401    // Now go through tokens on this frame, pruning forward links...  may have to
```

```
818  void LatticeFasterDecoderTpl<FST, Token>::DeleteForwardLinks(Token *tok) {
819    ForwardLinkT *l = tok->links, *m;
820    while (l != NULL) {
821      m = l->next;
822      forward_link_pool_.Free(l);
823      l = m;
824    }
825    tok->links = NULL;
826  }
```

```
900  template <typename FST, typename Token>
901  void LatticeFasterDecoderTpl<FST, Token>::DeleteElems(Elem *list) {
902    for (Elem *e = list, *e_tail; e != NULL; e = e_tail) {
903      e_tail = e->tail;
904      toks_.Delete(e);
905    }
906  }
907
```

调用HashList的Delete方法，将这么Elem标志为未被使用的状态

```
908  template <typename FST, typename Token>
909  void LatticeFasterDecoderTpl<FST, Token>::ClearActiveTokens() { // a cleanup routine, at utt end/begin
910    for (size_t i = 0; i < active_toks_.size(); i++) {
911      // Delete all tokens alive on this frame, and any forward
912      // links they may have.
913      for (Token *tok = active_toks_[i].toks; tok != NULL; ) {
914        DeleteForwardLinks(tok);
915        Token *next_tok = tok->next;
916        token_pool_.Free(tok);
917        num_toks_--;
918        tok = next_tok;
919      }
920    }
921    active_toks_.clear();
922    KALDI_ASSERT(num_toks_ == 0);
923  }
```

GetRawLattice方法的细节变化

主要的变化是新调用了TopSortTokens函数，用来检测是不是有环路

```
114 bool LatticeFasterDecoderTpl<FST, Token>::GetRawLattice(
115     Lattice *ofst,
116     bool use_final_probs) const {
117   typedef LatticeArc Arc;
118   typedef Arc::StateId StateId;
119   typedef Arc::Weight Weight;
120   typedef Arc::Label Label;
121
122   // Note: you can't use the old interface (Decode()) if you want to
123   // get the lattice with use_final_probs = false.  You'd have to do
124   // InitDecoding() and then AdvanceDecoding().
125   if (decoding_finalized_ && !use_final_probs)
126     KALDI_ERR << "You cannot call FinalizeDecoding() and then call "
127               << "GetRawLattice() with use_final_probs == false";
128
129   unordered_map<Token*, BaseFloat> final_costs_local;
130
131   const unordered_map<Token*, BaseFloat> &final_costs =
132       (decoding_finalized_ ? final_costs_ : final_costs_local);
133   if (!decoding_finalized_ && use_final_probs)
134     ComputeFinalCosts(&final_costs_local, NULL, NULL);
135
136   ofst->DeleteStates();
137   // num-frames plus one (since frames are one-based, and we have
138   // an extra frame for the start-state).
139   int32 num_frames = active_toks_.size() - 1;
140   KALDI_ASSERT(num_frames > 0);
141   const int32 bucket_count = num_toks_/2 + 3;
142   unordered_map<Token*, StateId> tok_map(bucket_count);
143   // First create all states.
144   std::vector<Token*> token_list;
145   for (int32 f = 0; f <= num_frames; f++) {
146     if (active_toks_[f].toks == NULL) {
147       KALDI_WARN << "GetRawLattice: no tokens active on frame " << f
148                  << ": not producing lattice.\n";
149       return false;
150     }
151     TopSortTokens(active_toks_[f].toks, &token_list);
152     for (size_t i = 0; i < token_list.size(); i++)
153       if (token_list[i] != NULL)
154         tok_map[token_list[i]] = ofst->AddState();
155   }
```

每一帧的活跃token都会调用TopSortTokens给Token排序，先加入到token list中的token对应的位置越靠前，因此158行直接设置Lattice FST的开始状态为0。

对比LatticeSimple版本的第118-119行

```
156   // The next statement sets the start state of the output FST.  Because we
157   // topologically sorted the tokens, state zero must be the start-state.
158   ofst->SetStart(0);
159
160   KALDI_VLOG(4) << "init:" << num_toks_/2 + 3 << " buckets:"
161                 << tok_map.bucket_count() << " load:" << tok_map.load_factor()
162                 << " max:" << tok_map.max_load_factor();
163   // Now create all arcs.
164   for (int32 f = 0; f <= num_frames; f++) {
165     for (Token *tok = active_toks_[f].toks; tok != NULL; tok = tok->next) {
166       StateId cur_state = tok_map[tok];
167       for (ForwardLinkT *l = tok->links;
168            l != NULL;
169            l = l->next) {
170         typename unordered_map<Token*, StateId>::const_iterator
171             iter = tok_map.find(l->next_tok);
172         StateId nextstate = iter->second;
173         KALDI_ASSERT(iter != tok_map.end());
174         BaseFloat cost_offset = 0.0;
175         if (l->ilabel != 0) {  // emitting..
176           KALDI_ASSERT(f >= 0 && f < cost_offsets_.size());
177           cost_offset = cost_offsets_[f];
178         }
179         Arc arc(l->ilabel, l->olabel,
180                 Weight(l->graph_cost, l->acoustic_cost - cost_offset),
181                 nextstate);
182         ofst->AddArc(cur_state, arc);
183       }
184       if (f == num_frames) {
185         if (use_final_probs && !final_costs.empty()) {
186           typename unordered_map<Token*, BaseFloat>::const_iterator
187               iter = final_costs.find(tok);
188           if (iter != final_costs.end())
189             ofst->SetFinal(cur_state, LatticeWeight(iter->second, 0));
190         } else {
191           ofst->SetFinal(cur_state, LatticeWeight::One());
192         }
193       }
194     }
195   }
196   return (ofst->NumStates() > 0);
197 }
```

减去cost_offset得到绝对的AM得分

```
926  template <typename FST, typename Token>
927  void LatticeFasterDecoderTpl<FST, Token>::TopSortTokens(
928      Token *tok_list, std::vector<Token*> *topsorted_list) {
929    unordered_map<Token*, int32> token2pos;
930    typedef typename unordered_map<Token*, int32>::iterator IterType;
931    int32 num_toks = 0;
932    for (Token *tok = tok_list; tok != NULL; tok = tok->next)
933      num_toks++;
934    int32 cur_pos = 0;
935    // We assign the tokens numbers num_toks - 1, ... , 2, 1, 0.
936    // This is likely to be in closer to topological order than
937    // if we had given them ascending order, because of the way
938    // new tokens are put at the front of the list.
939    for (Token *tok = tok_list; tok != NULL; tok = tok->next)
940      token2pos[tok] = num_toks - ++cur_pos;
941
942    unordered_set<Token*> reprocess;
943
944    for (IterType iter = token2pos.begin(); iter != token2pos.end(); ++iter) {
945      Token *tok = iter->first;
946      int32 pos = iter->second;
947      for (ForwardLinkT *link = tok->links; link != NULL; link = link->next) {
948        if (link->ilabel == 0) {
949          // We only need to consider epsilon links, since non-epsilon links
950          // transition between frames and this function only needs to sort a list
951          // of tokens from a single frame.
952          IterType following_iter = token2pos.find(link->next_tok);
953          if (following_iter != token2pos.end()) { // another token on this frame,
954                                                   // so must consider it.
955            int32 next_pos = following_iter->second;
956            if (next_pos < pos) { // reassign the position of the next Token.
957              following_iter->second = cur_pos++;
958              reprocess.insert(link->next_tok);
959            }
960          }
961        }
962      }
963      // In case we had previously assigned this token to be reprocessed, we can
964      // erase it from that set because it's "happy now" (we just processed it).
965      reprocess.erase(tok);
966    }
967
```

如果没有环路，956行不会命中

TopSortTokens方法

TopSortTokens的方法有两个作用：

1. 对每一帧的所有的活跃token计算一个token到链表中位置的映射，检查空转移中，转移的目标状态的位置一定是在发射状态的后面；比如图1，tn3由tn2经过空转移到达，那么tn3的位置序号3就小于tn1的位置1.
2. 如果图上有空跳转的环路，比如a-b-c-d..-a。图2为例，当处理tn3的时候(pos=3)，其link指向的next token为tn1(next_pos=1), 满足了956的条件，tn1的位置序号就会增大，同时把tn1放入reprocess中，968行之后的代码进行环路判断。

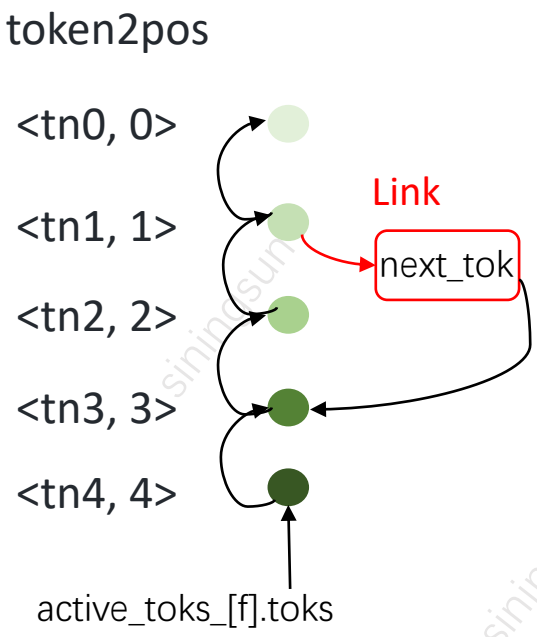

图1 无环路情况

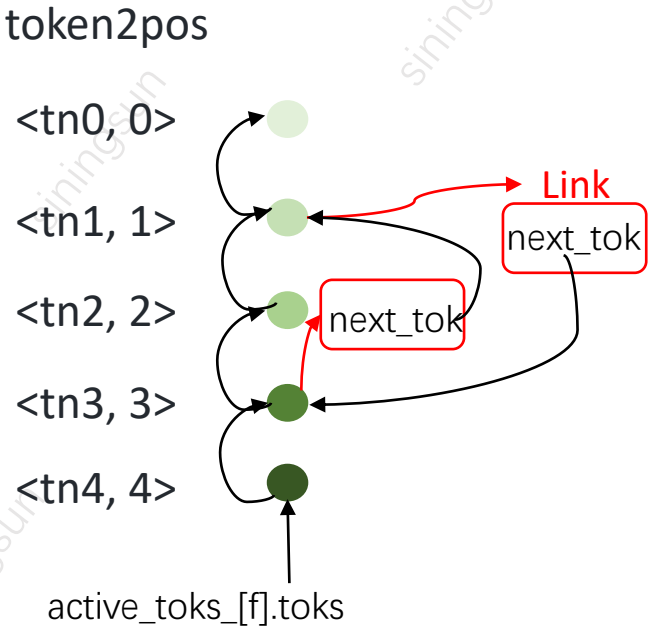

图2 有环路的情况

设置了一个比较大的数，避免存在过大的环路或者过多的空转移，大部分时候不会命中最坏情况

```
968   size_t max_loop = 1000000, loop_count; // max_loop is to detect epsilon cycles.
969   for (loop_count = 0;
970        !reprocess.empty() && loop_count < max_loop; ++loop_count) {
971     std::vector<Token*> reprocess_vec;
972     for (typename unordered_set<Token*>::iterator iter = reprocess.begin();
973          iter != reprocess.end(); ++iter)
974       reprocess_vec.push_back(*iter);
975     reprocess.clear();
976     for (typename std::vector<Token*>::iterator iter = reprocess_vec.begin();
977          iter != reprocess_vec.end(); ++iter) {
978       Token *tok = *iter;
979       int32 pos = token2pos[tok];
980       // Repeat the processing we did above (for comments, see above).
981       for (ForwardLinkT *link = tok->links; link != NULL; link = link->next) {
982         if (link->ilabel == 0) {
983           IterType following_iter = token2pos.find(link->next_tok);
984           if (following_iter != token2pos.end()) {
985             int32 next_pos = following_iter->second;
986             if (next_pos < pos) {
987               following_iter->second = cur_pos++;
988               reprocess.insert(link->next_tok);
989             }
990           }
991         }
992       }
993     }
994   }
995   KALDI_ASSERT(loop_count < max_loop && "Epsilon loops exist in your decoding "
996                "graph (this is not allowed!)");
997
998   topsorted_list->clear();
999   topsorted_list->resize(cur_pos, NULL);  // create a list with NULLs in between.
1000  for (IterType iter = token2pos.begin(); iter != token2pos.end(); ++iter)
1001    (*topsorted_list)[iter->second] = iter->first;
1002  }
```

将token按照插入顺序，或者保证后继性的顺序输出

TopSortTokens方法

TopSortTokens的方法有两个作用：

1. 对每一帧的所有的活跃token计算一个token到链表中位置的映射，检查空转移中，转移的目标状态的位置一定是在发射状态的后面；比如图1，tn3由tn2经过空转移到达，那么tn3的位置序号3就小于tn1的位置1.
2. 如果图上有空跳转的环路，比如a-b-c-d..-a。图2为例，当处理tn3的时候(pos=3)，其link指向的next token为tn1(next_pos=1), 满足了956的条件，tn1的位置序号就会增大，同时把tn1放入reprocess中，968行之后的代码进行环路判断。

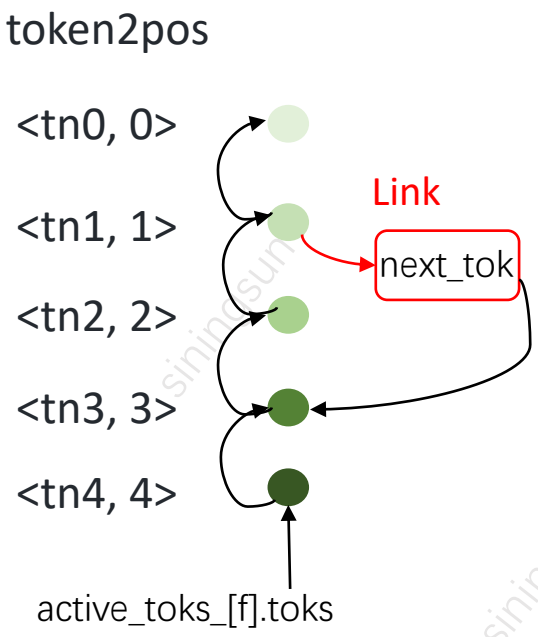

图1 无环路情况

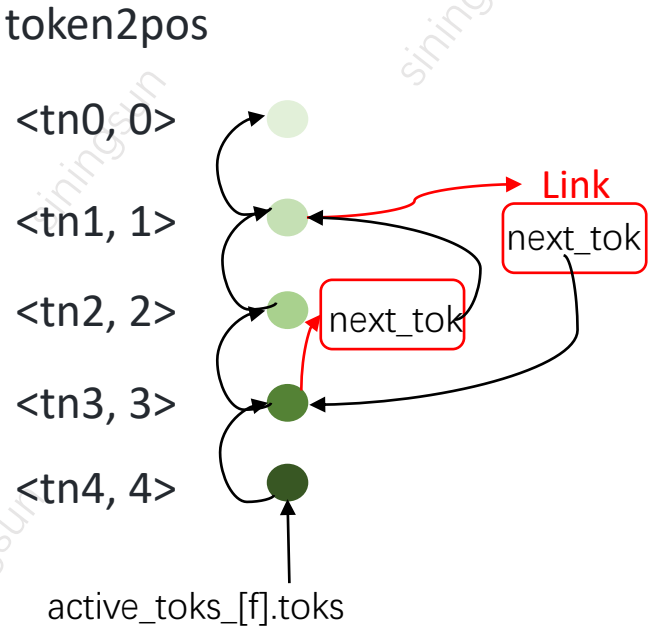

图2 有环路的情况