Kaldi FasterDecoder 代码解析+逐步实例

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

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

FasterDecoder中的主要变化

1. 增加了max_active, min_active, beam_delta这几个和剪枝有关的变量；
2. 增加了GetCutoff方法，GetCutoff用于计算剪枝的阈值以及自适应的改变下一帧剪枝用到的beam参数
3. 不再使用SimpleDecoder中的unordered_map来存储prev_toks_和cur_toks_，而是使用HashList进行存储
4. ProcessEmitting方法中，剪枝的地方发生了变化

```
struct FasterDecoderOptions {
  BaseFloat beam;
  int32 max_active;
  int32 min_active;
  BaseFloat beam_delta;
  BaseFloat hash_ratio;
  FasterDecoderOptions(): beam(16.0),
                          max_active(std::numeric_limit
                          min_active(20), // This decod
                                          // alignment,
                          beam_delta(0.5),
                          hash_ratio(2.0) { }
```

```
  /// Gets the weight cutoff.  Also counts the active tokens.
  double GetCutoff(Elem *list_head, size_t *tok_count,
                   BaseFloat *adaptive_beam, Elem **best_elem);

  void PossiblyResizeHash(size_t num_toks);

  // ProcessEmitting returns the likelihood cutoff used.
  // It decodes the frame num_frames_decoded_ of the decodable object
  // and then increments num_frames_decoded_
  double ProcessEmitting(DecodableInterface *decodable);

  // TODO: first time we go through this, could avoid using the queue.
  void ProcessNonemitting(double cutoff);

  // HashList defined in ../util/hash-list.h.  It actually allows us to maintain
  // more than one list (e.g. for current and previous frames), but only one of
  // them at a time can be indexed by StateId.
  HashList<StateId, Token*> toks_;
  const fst::Fst<fst::StdArc> &fst_;
  FasterDecoderOptions config_;
  std::vector<const Elem* > queue_;  // temp variable used in ProcessNonemitting,
  std::vector<BaseFloat> tmp_array_;  // used in GetCutoff.
  // make it class member to avoid internal new/delete.
```

```
// ProcessEmitting returns the likelihood cutoff used.
double FasterDecoder::ProcessEmitting(DecodableInterface *decodable) {
  int32 frame = num_frames_decoded_;
  Elem *last_toks = toks_.Clear();
  size_t tok_cnt;
  BaseFloat adaptive_beam;
  Elem *best_elem = NULL;
  double weight_cutoff = GetCutoff(last_toks, &tok_cnt,
                                   &adaptive_beam, &best_elem);
  KALDI_VLOG(3) << tok_cnt << " tokens active.";
  PossiblyResizeHash(tok_cnt);  // This makes sure the hash is always big enough.

  // This is the cutoff we use after adding in the log-likes (i.e.
  // for the next frame).  This is a bound on the cutoff we will use
  // on the next frame.
  double next_weight_cutoff = std::numeric_limits<double>::infinity();

  // First process the best token to get a hopefully
  // reasonably tight bound on the next cutoff.
  if (best_elem) {
    StateId state = best_elem->key;
    Token *tok = best_elem->val;
    for (fst::ArcIterator<fst::Fst<Arc> > aiter(fst_, state);
         !aiter.Done();
         aiter.Next()) {
      const Arc &arc = aiter.Value();
      if (arc.ilabel != 0) {  // we'd propagate..
        BaseFloat ac_cost = - decodable->LogLikelihood(frame, arc.ilabel);
        double new_weight = arc.weight.Value() + tok->cost_ + ac_cost;
        if (new_weight + adaptive_beam < next_weight_cutoff)
          next_weight_cutoff = new_weight + adaptive_beam;
      }
```

ProcessEmitting方法

```
221 double FasterDecoder::ProcessEmitting(DecodableInterface *decodable) {
222   int32 frame = num_frames_decoded_;
223   Elem *last_toks = toks_.Clear();
224   size_t tok_cnt;
225   BaseFloat adaptive_beam;
226   Elem *best_elem = NULL;
227   double weight_cutoff = GetCutoff(last_toks, &tok_cnt,
228                                    &adaptive_beam, &best_elem);
229   KALDI_VLOG(3) << tok_cnt << " tokens active.";
230   PossiblyResizeHash(tok_cnt);  // This makes sure the hash is always big enough.
231
232   // This is the cutoff we use after adding in the log-likes (i.e.
233   // for the next frame).  This is a bound on the cutoff we will use
234   // on the next frame.
235   double next_weight_cutoff = std::numeric_limits<double>::infinity();
236
237   // First process the best token to get a hopefully
238   // reasonably tight bound on the next cutoff.
239   if (best_elem) {
240     StateId state = best_elem->key;
241     Token *tok = best_elem->val;
242     for (fst::ArcIterator<fst::Fst<Arc> > aiter(fst_, state);
243          !aiter.Done();
244          aiter.Next()) {
245       const Arc &arc = aiter.Value();
246       if (arc.ilabel != 0) {  // we'd propagate..
247         BaseFloat ac_cost = - decodable->LogLikelihood(frame, arc.ilabel);
248         double new_weight = arc.weight.Value() + tok->cost_ + ac_cost;
249         if (new_weight + adaptive_beam < next_weight_cutoff)
250           next_weight_cutoff = new_weight + adaptive_beam;
251       }
252     }
253   }
```

首先调用HashList的Clear方法，返回上一帧的所有token的链表，清空哈希表

weight_cutoff是上一帧token的剪枝阈值，best_elem是上一帧最好的token，adaptive_beam自适应剪枝参数

用于新的一帧产生的Token的剪枝

遍历上一帧最优Token的所有非空发射弧，更新next_weight_cutoff为上一帧最优Token出发的最优转移的得分+adaptive_beam，这实际上是一个预估一个下一帧的cutoff

```
257   // the tokens are now owned here, in last_toks, and the hash is empty.
258   // 'owned' is a complex thing here; the point is we need to call TokenDelete
259   // on each elem 'e' to let toks_ know we're done with them.
260   for (Elem *e = last_toks, *e_tail; e != NULL; e = e_tail) {  // loop this way
261     // n++;
262     // because we delete "e" as we go.
263     StateId state = e->key;
264     Token *tok = e->val;
265     if (tok->cost_ < weight_cutoff) {  // not pruned.
266       // np++;
267       KALDI_ASSERT(state == tok->arc_.nextstate);
268       for (fst::ArcIterator<fst::Fst<Arc> > aiter(fst_, state);
269            !aiter.Done();
270            aiter.Next()) {
271         Arc arc = aiter.Value();
272         if (arc.ilabel != 0) {  // propagate..
273           BaseFloat ac_cost =  - decodable->LogLikelihood(frame, arc.ilabel);
274           double new_weight = arc.weight.Value() + tok->cost_ + ac_cost;
275           if (new_weight < next_weight_cutoff) {  //
276             Token *new_tok = new Token(arc, ac_cost, tok);
277             Elem *e_found = toks_.Insert(arc.nextstate, new_tok);
278             if (new_weight + adaptive_beam < next_weight_cutoff)
279               next_weight_cutoff = new_weight + adaptive_beam;
280             if (e_found->val != new_tok) {
281               if (*(e_found->val) < *new_tok) {
282                 Token::TokenDelete(e_found->val);
283                 e_found->val = new_tok;
284               } else {
285                 Token::TokenDelete(new_tok);
286               }
287             }
288           }
289         }
290       }
291     }
292     e_tail = e->tail;
293     Token::TokenDelete(e->val);
294     toks_.Delete(e);
295   }
296   num_frames_decoded_++;
297   return next_weight_cutoff;
298 }
```

遍历上一帧所有的Elem，每个Elem存储着Token

如果tok的cost_大于weight_cutoff，就不会继续传递到下一帧

遍历从这个tok的state出发的所有发射弧

新的分数在next_weight_cutoff范围内新建token并插入到hashlist中

更新next_weight_cutoff为更严格的阈值

第277行调用Insert操作，如果state对应的token已经存在，那么Elem中的token是在外部更新的，如果不存在，在内部就会更新，详见Insert方法。

调用Detele方法，删除已经处理的e，参考HashList的Delete方法，这里并不会删除e的空间

返回当前帧的剪枝阈值，处理非发射转移时也会使用此阈值剪枝

GetCutoff方法

```
147  // Gets the weight cutoff.  Also counts the active tokens.
148  double FasterDecoder::GetCutoff(Elem *list_head, size_t *tok_count,
149                                  BaseFloat *adaptive_beam, Elem **best_elem) {
150    double best_cost = std::numeric_limits<double>::infinity();
151    size_t count = 0;
152    if (config_.max_active == std::numeric_limits<int32>::max() &&
153        config_.min_active == 0) {
154      for (Elem *e = list_head; e != NULL; e = e->tail, count++) {
155        double w = e->val->cost_;
156        if (w < best_cost) {
157          best_cost = w;
158          if (best_elem) *best_elem = e;
159        }
160      }
161      if (tok_count != NULL) *tok_count = count;
162      if (adaptive_beam != NULL) *adaptive_beam = config_.beam;
163      return best_cost + config_.beam;
164    } else {
165      tmp_array_.clear();
166      for (Elem *e = list_head; e != NULL; e = e->tail, count++) {
167        double w = e->val->cost_;
168        tmp_array_.push_back(w);
169        if (w < best_cost) {
170          best_cost = w;
171          if (best_elem) *best_elem = e;
172        }
173      }
174      if (tok_count != NULL) *tok_count = count;
175      double beam_cutoff = best_cost + config_.beam,
176          min_active_cutoff = std::numeric_limits<double>::infinity(),
177          max_active_cutoff = std::numeric_limits<double>::infinity();
178
179      if (tmp_array_.size() > static_cast<size_t>(config_.max_active)) {
180        std::nth_element(tmp_array_.begin(),
181                         tmp_array_.begin() + config_.max_active,
182                         tmp_array_.end());
183        max_active_cutoff = tmp_array_[config_.max_active];
184      }
```

GetCutoff代码大致流程

1. 152-163行，如果没有设置max_active和min_active，直接遍历上一帧的所有token，获得最优的cost，best_cost，然后剪枝阈值就是best_cost+beam
2. 如果设置了max或者min_active，那么在165-209行，会根据需要保留的max和min的token的来计算更加严格的剪枝阈值
3. 165-173，首先遍历上一帧所有token并获得best_cost，同时把所有token的分数都保存在tmp_array_中
4. 第175-177行，计算beam_cutoff，初始化min_active_cutoff和max_active_cutoff
5. 179行，如果活跃token的总数大于预设的max_active，使用nth_element函数来对tmp_array_进行排序，tmp_array_中前max_active个元素都比第max_active处的元素小，max_active位置之后的元素都比第max_active处大，因此第183行获得了token排在第max_active处的得分。

GetCutoff方法

```
185     if (max_active_cutoff < beam_cutoff) { // max_active is tighter than beam.
186       if (adaptive_beam)
187         *adaptive_beam = max_active_cutoff - best_cost + config_.beam_delta;
188       return max_active_cutoff;
189     }
190     if (tmp_array_.size() > static_cast<size_t>(config_.min_active)) {
191       if (config_.min_active == 0) min_active_cutoff = best_cost;
192       else {
193         std::nth_element(tmp_array_.begin(),
194                          tmp_array_.begin() + config_.min_active,
195                          tmp_array_.size() > static_cast<size_t>(config_.max_active) ?
196                          tmp_array_.begin() + config_.max_active :
197                          tmp_array_.end());
198         min_active_cutoff = tmp_array_[config_.min_active];
199       }
200     }
201     if (min_active_cutoff > beam_cutoff) { // min_active is looser than beam.
202       if (adaptive_beam)
203         *adaptive_beam = min_active_cutoff - best_cost + config_.beam_delta;
204       return min_active_cutoff;
205     } else {
206       *adaptive_beam = config_.beam;
207       return beam_cutoff;
208     }
209   }
210 }
```

beam_delta是一个可配置的常数

6. 185-189，如果使用max_active获得的cutoff比之前获得的beam_cutoff小，也就是更严格，188行则返回max_active_cutoff作为上一帧token的剪枝阈值，不再处理min_active；187行则是重新计算了一个较小的自适应beam，在当前帧的token剪枝的时候替代固定的beam，如图1
7. 第190行开始处理上一帧活跃token总数大于min_active的情况。如果设置了min_active>0，193-198获得cost为第min_active小，如果min_active_cutoff>beam_cutoff，说明此时beam cutoff的值太严格，不能保证有min_active个token，因此第203行更新beam为一个更大的数值，204行返回较为宽松的cutoff，如图2

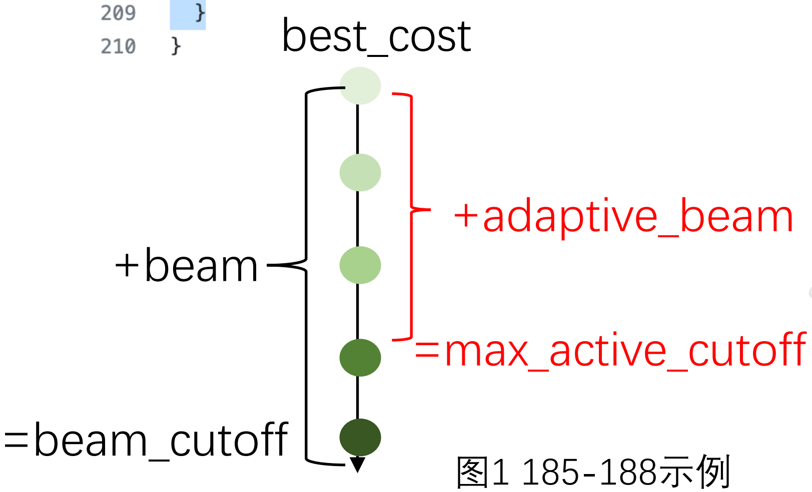

图1 185-188示例

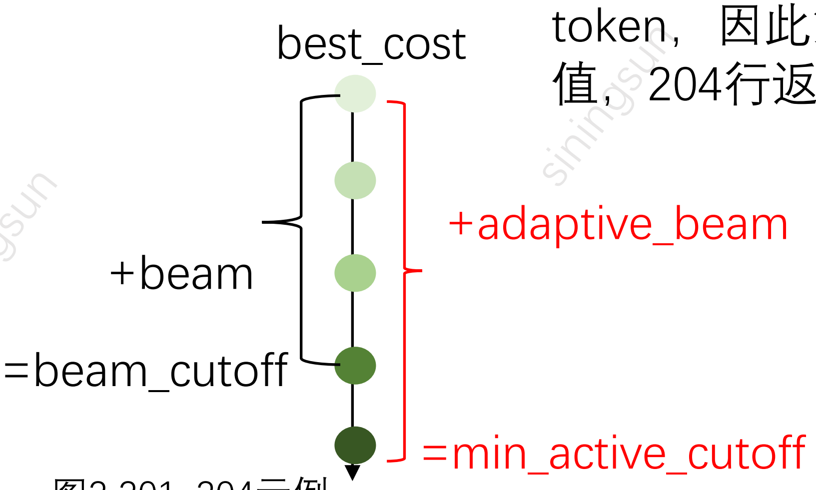

图2 201-204示例

siningsun

```cpp
167 void SimpleDecoder::ProcessEmitting(DecodableInterface *decodable) {
168   int32 frame = num_frames_decoded_;
169   // Processes emitting arcs for one frame.  Propagates from
170   // prev_toks_ to cur_toks_.
171   double cutoff = std::numeric_limits<BaseFloat>::infinity();
172   for (unordered_map<StateId, Token*>::iterator iter = prev_toks_.begin();
173        iter != prev_toks_.end();
174        ++iter) {
175     StateId state = iter->first;
176     Token *tok = iter->second;
177     KALDI_ASSERT(state == tok->arc_.nextstate);
178     for (fst::ArcIterator<fst::Fst<StdArc> > aiter(fst_, state);
179          !aiter.Done();
180          aiter.Next()) {
181       const StdArc &arc = aiter.Value();
182       if (arc.ilabel != 0) {  // propagate..
183         BaseFloat acoustic_cost = -decodable->LogLikelihood(frame, arc.ilabel);
184         double total_cost = tok->cost_ + arc.weight.Value() + acoustic_cost;
185 
186         if (total_cost >= cutoff) continue;
187         if (total_cost + beam_  < cutoff)
188           cutoff = total_cost + beam_;
189         Token *new_tok = new Token(arc, acoustic_cost, tok);
```

```cpp
267 void SimpleDecoder::PruneToks(BaseFloat beam, unordered_map<StateId, Token*> *toks) {
268   if (toks->empty()) {
269     KALDI_VLOG(2) <<  "No tokens to prune.\n";
270     return;
271   }
272   double best_cost = std::numeric_limits<double>::infinity();
273   for (unordered_map<StateId, Token*>::iterator iter = toks->begin();
274        iter != toks->end(); ++iter)
275     best_cost = std::min(best_cost, iter->second->cost_);
276   std::vector<StateId> retained;
277   double cutoff = best_cost + beam;
278   for (unordered_map<StateId, Token*>::iterator iter = toks->begin();
279        iter != toks->end(); ++iter) {
280     if (iter->second->cost_ < cutoff)
281       retained.push_back(iter->first);
282     else
283       Token::TokenDelete(iter->second);
284   }
285   unordered_map<StateId, Token*> tmp;
286   for (size_t i = 0; i < retained.size(); i++) {
287     tmp[retained[i]] = (*toks)[retained[i]];
288   }
289   KALDI_VLOG(2) <<  "Pruned to " << (retained.size()) << " toks.\n";
290   tmp.swap(*toks);
291 }
```

和SimpleDecoder的剪枝的对比

1. SimpleDecoder里面，188行进行了cutoff的更新，在cutoff没有达到最优之前，cutoff的阈值是比较宽松的，因此会有更多的token加入到解码过程中，而这些token会在PruneToks里面再被剪掉，从而造成了冗余计算。
2. FasterDecoder里面会预估下一帧的最优的剪枝cutoff，而SimpleDecoder里面没有
3. FasterDecoder里面会通过max-active和min-activel来不断的调整beam，而SimpleDecoder里面的beam是固定的。

总结来说，了解SimpleDecoder和HashList之后，FasterDecoder就非常简单了！

End