Removing Redundancy from Lattices

Abstract

In this work we present a novel word lattice filtering algorithm which removes redundancy from lattices. We use the filtering algorithm to analyze lattices obtained from dynamic network and transducer-based LVCSR decoders from several sites regarding size, coverage, and redundancy. We show that our filtering algorithm reduces the size of lattices by 30 to 90% without degrading the oracle word error rate.

Index Terms: LVCSR, decoding, lattice, redundancy, filter

1. Introduction

In the most common interpretation, a word lattice is a weighted labeled directed acyclic graph, representing alternative word sequences. Usually corresponding word probabilities and timing information are preserved [1], and the language model (LM) and acoustic model (AM) probabilities are stored separately, depending on the intended use of the lattice. Word lattices are useful for many different applications. Some of the most common uses are LM rescoring, discriminative training, confidence estimation and -adaptation, confusion network generation, and keyword spotting.

There are lattice generation algorithms for lexical-tree based decoders and FST-based decoders. In this paper we consider lattice generation for lexical-tree based decoders. We formulate the lattice generation problem in the same way as [2], in that we aim to get retain only the best path for each distinct word sequence. The standard lattice generation algorithm in lexical-tree based decoders is based on word pair approximation [1]. Word pair approximation produces less redundant lattices than exact lattice generation methods [3], because the dynamic programming search optimizes over the word boundaries; nevertheless, even the word pair approximation method may generate redundant multiple paths for the same word sequence.

In order to generate a non-redundant lattice we need an algorithm that removes redundant paths from the lattice. This paper presents such an algorithm.

A weighted finite state transducer (WFST) based algorithm with a similar goal was introduced in [2] and [4]. That algorithm is difficult to apply when not using a WFST based speech recognition framework, and it is stretched to its efficiency limits when dealing with large lattices. Our novel algorithm can be implemented straightforwardly, and works efficiently even with huge lattices.

2. Redundancy Removal

The goal of our algorithm is to remove all arcs from the lattice which are not on the best path for at least one covered word sequence. In abstract, our algorithm should do the following:

1. For every possible word sequence \(w_1, w_2, \ldots\), find the best corresponding path through the lattice, and select all arcs on the path.
2. Remove all arcs from the lattice which were not selected.

2.1. Closure Form

We only want to consider non-noise tokens as distinct words (by noise we mean everything which we do not consider an actual word, eg. noises, silence, epsilon, etc.). Thus, for efficiency and simplicity reasons, we will define our algorithm based on an equivalent closure form of the lattice, which skips all noise arcs. Each arc in the closure form corresponds to any number of noise arcs followed by one word arc in the original form, thus each arc of the closure form has a non-noise word label. We directly filter away redundant noise paths, by storing only the best noise path leading to a word arc. In case there are multiple final states, we append an artificial final state before computing the closure, and insert new arcs from the old final states to the new one with a special final label; thereby we force all paths to meet at latest on the final state. Figure 1 shows a typical lattice, in its original form (a) and in closure form (b). It contains 4 different alignment paths for the word sequence how are you.

On such a lattice, redundancy removal corresponds to a search for the single-best path; we will nevertheless use this lattice to showcase our algorithm, because it exhibits all necessary features.

2.2. Naive Redundancy Removal Algorithm

After the closure mentioned above, we have a lattice where each arc has a non-noise word label (or the special final label). Our goal now is to remove those arcs that are not on the best path for some word sequence.

Let \(s\) be a state in the closure form lattice, \(A(s) = \{a_1, a_2, \ldots\}\) the outgoing arcs, \(t(a)\) the target state of arc \(a\) (or of the last arc of an arc sequence), \(l(a)\) its word label, \(s = 1\) the initial state, and \(S\) the total number of states.

Algorithm 1 enumerates all word sequences represented by the lattice. It manages hypotheses \(h_n\) for increasing word sequence lengths \(n\), by appending encountered successor words to sequences of length \(n - 1\) (line 6). It selects the best arc...
Algorithm 1: Naive redundancy removal algorithm.

1 \( h_0 \leftarrow \{((), (), 1)\} \) // hypotheses: (words, trace, state)
2 for \( n \in \{1, 2, \ldots \} \) do
3 \( h_E \leftarrow \{\} \)
4 foreach hyp. \((w_1, w_2, \ldots, w_n) = (l(a_1), l(a_2), \ldots, l(a_n))\) in \( h_{n-1} \) do
5 \( h_E \leftarrow add \left( (w_n, t(a)), (a_{n-1}, a), t(a) \right) \)
6 \( h_n \leftarrow \{\} \)
7 forall traces \( T = \{(w_n, s)\} \) in \( h_E \) with
8 equal words \((w_n)\) and equal state \( s'\) do
9 if \( s' \) is final then
10 select best \((T)\)
11 else
12 \( h_n \leftarrow add \left( (w_n), best(T), s' \right) \)

Algorithm 2: Efficient redundancy removal algorithm.

1 \( R(1) \leftarrow \{\} \)
2 \( R(2, \ldots, S) \leftarrow \{\} \)
3 foreach origin \( s \in \{1, \ldots, S\} \) in topological order do
4 \( h_0 \leftarrow \{((), s)\} \) hypotheses: (words, trace, state)
5 for \( n \in \{1, 2, 3, \ldots\} \) do
6 \( h_E \leftarrow \{\} \)
7 foreach hyp. \((w_1, w_2, \ldots, w_n, s)\) in \( h_{n-1} \) do
8 foreach arc \( a \) in \( A(s') \) do
9 if \( R(s) \) prefix-matches \((w_n, l(a))\) then
10 \( h_E \leftarrow add \left( (w_n, l(a)), (w_{n-1}, a), t(a) \right) \)
11 \( h_R \leftarrow \{\} \)
12 forall traces \( T = \{(w_n, s)\} \) in \( h_E \) with
13 equal words \((w_n)\) and equal state \( s'\) do
14 \( h_R \leftarrow add \left( (w_n), best(T), s' \right) \)
15 \( h_n \leftarrow \{\} \)
16 forall traces \( T = \{(w_n)\} \) in \( h_R \) with
17 equal words \((w_n, s)\) do
18 \( r \leftarrow \text{common prefix}(T) \)
19 if \( R(s) \) matches \((w_n)\) and \(|r| \geq 1\) then
20 \( R(t(r)) \leftarrow add \left( w_{|r|+1} \ldots \right) \)
21 select arcs \( r \)
22 else
23 \( h_n \leftarrow add \) all hyps in \( h_R \) with \((w_n)\)

Figure 1: Lattice in original form (a) and closure form (b). Each arc is annotated with its probability.

2.3. Efficient Redundancy Removal Algorithm

The fundamental idea of our efficient algorithm is to exploit the inherent path recombination [5] to break the expansion of full word sequences up into shorter sub-sequences. The path recombination assumption means that, for any two search hypotheses, all successor paths will merge after a specific number of equal words were recognized. It is important to note that the path recombination heuristic makes the algorithm efficient, but the correctness of the result is guaranteed unconditionally. The path recombination assumption is usually satisfied, because it is what allows us to prune in LVCSR in the first place [5], and it is a prerequisite of the word pair approximation commonly used to generate lattices [1].

Algorithm 2 efficiently selects the best path for each representend word sequence. It expands partial word- and arc sequences of increasing length \( n \), and recombines arc sequence hypotheses as soon as possible. As soon as it determines that some prefix arcs are definitely on the best path for a specific partial word sequence, it selects those arcs, and delays the following expansion. Follow-up word sequences are later expanded behind the ending state of the prefix arc sequence.

In worst case, if no common prefix would be found in line 18 before reaching the final state, then the algorithm would expand all word sequences from the initial origin state \( s = 1 \) up to the final state, and thus would be equivalent to the naive algorithm from the previous section. In practice however, most paths are recombined after few equal words, and only short partial word sequences are expanded behind each origin state \( s \) (line 3).

A trace (line 4, 12, and 16) is a sequence \((a_1, a_2, \ldots, a_n)\) of \( n \) arcs which originates in the current origin state \( s \). A hypothesis \((w_1, a_1), (a_1, a_2), \ldots, (a_n, s')\) (line 4) is a combination of a word-sequence \((w_1, \ldots, w_n)\), a trace \((a_1, \ldots, a_n)\), and an ending state \( s' \). The trace \((a_1, \ldots, a_n)\) of a hypothesis always originates in state \( s \), ends in state \( s' \), and is consistent with the corresponding word sequence \((w_1, \ldots, w_n)\) (eg. \((w_1, w_2, \ldots, w_n) = ((l(a_1)), l(a_2), \ldots, l(a_n))\)).
quences starting at state \( s \), for which we know that the state is on the best path. \( R(s) \) matches all word sequences starting with any of the prefixes contained in \( R(s) \) (line 19). \( R(s) \) prefix-matches all word sequences which are either a prefix of any of the prefixes in \( R(s) \), or which are matched as by the previous sentence (line 9).

The function \( \text{best}(T) \) takes a list of traces \( T \), and returns the one with the best overall score; common_prefix\((T)\) returns the longest common prefix (eg. those arcs which all given traces are starting with).

2.4. Summary

First, we initialize the prefixes \( R(1) \) for the initial state 1 with the empty prefix () which matches all word sequences (line 1). This means that the initial state is on the best path for all possible word sequences. We initially select nothing (line 2) for all other origin states. Then we process all origin states \( s \) in topological order, expanding all following word sequences which are consistent with the selected prefixes \( R(s) \), until all hypotheses for each expanded word sequence share a non-empty prefix arc sequence. As soon as the traces of all hypotheses for a specific word sequence share some common prefix arcs \( r \) (line 18), we know that these arcs must be selected, because they are definitely on the best path for that word sequence and all its extensions. At that point, all active hypotheses of that word sequence are reachable from the state \( t(r) \) where the common prefix ends; thus, instead of further expanding this already long word sequence relative to origin \( s \), we can re-expand equivalent hypotheses later by re-starting expansion behind a new origin state \( t(r) \). We delay the expansion by adding the prefix required to expand the same successor word sequences into \( R(t(r)) \) (line 20), and stopping further expansion relative to the current origin state. The re-expansion will later lead to the same arc selections (line 21) as if we continued expansion now, but eventually we can share some effort with other word sequences intersecting at the same state \( t(r) \).

In Algorithm 3 we apply the redundancy removal Algorithm 2 to the example lattice from Figure 1. We label arcs as their concatenated origin and target state (eg. arc 12 is the arc leading from state 1 to state 2). The line number labels correspond to the line numbers from original Algorithm 2, and operations without effect are skipped.

2.5. Practical Efficiency

It is not guaranteed that the expansion of word sequences actually terminates before reaching the final state, and the algorithm may try to expand all word sequences representable by the lattice; therefore, we have integrated two abortion points into our implementation: Either if the expanded sequence length \( n \) exceeds 10, or if the number of hypotheses \( h_n \) exceeds 100k, then we abort the expansion behind the current origin state \( s \), by simply selecting the currently best path for each hypothesized word sequence. This abortion usually never triggers, unless we’re dealing with very large lattices beyond 30k arcs per second.

2.6. Other Applications

Our algorithm preserves arc identities. This allows to use the algorithm flexibly, for example to solve the part-of-speech(POS) tagging problem described in [4]: The label of each arc can be mapped to the corresponding POS-tag, redundancy removal can be applied based on the POS tags, and afterwards the label can be mapped back based on the arc identity. Only those arcs which are on the single-best path of at least one POS sequence will remain.

3. Experimental Results

We evaluate our redundancy removal algorithm on lattices generated with 3 different decoders: The dynamic network decoder from IBM’s Attila speech recognition toolkit [6], its static WFST based decoder presented in [7, 8], and the RWTH Aachen decoder [9, 10]. Additionally, we perform a cross-site comparison between lattices generated with the mentioned decoders, plus lattices generated using Kaldi based on the WFST determinization method presented in [2]. All of these recognizers have different approximative integrated approaches to reduce lattice redundancy, specifically regarding noise tokens. Both dynamic network decoders rely on the word pair approximation method for lattice generation [1].

As test task we use the Babel Vietnamese development corpus, using the reference segmentation supplied by the project. For the cross-site comparison, independently developed ASR systems for Babel Vietnamese are used, all based on the same training data, all achieving a similar single-best precision, but trained using different methods. IBM’s Vietnamese system operates along the lines described in [11], and RWTH’s system was described in [12]. The Kaldi based system is very similar to the Vietnamese system described in [13], extended by the
Table 1: Effect of the redundancy removal algorithm on lattices from the RWTH decoder, RWTH decoder without word-end pruning, Attila dynamic decoder, and Attila static WFST decoder. We vary the lattice posterior pruning thresholds, and measure the oracle WER, the number of lattice arcs per second, the number of lattice arcs per second after redundancy removal, and the reduction ratio.

<table>
<thead>
<tr>
<th>Lat.-Pruning</th>
<th>Oracle WER [%]</th>
<th>Arcs / Second</th>
<th>Filt.</th>
<th>Reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>RWTH Baseline</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>40.26%</td>
<td>9.5</td>
<td>4.2</td>
<td>56%</td>
</tr>
<tr>
<td>8</td>
<td>24.49%</td>
<td>79</td>
<td>47</td>
<td>40%</td>
</tr>
<tr>
<td>12</td>
<td>17.53%</td>
<td>464</td>
<td>314</td>
<td>23%</td>
</tr>
<tr>
<td>16</td>
<td>15.95%</td>
<td>1.3k</td>
<td>0.8k</td>
<td>32%</td>
</tr>
<tr>
<td>20</td>
<td>15.66%</td>
<td>2k</td>
<td>1.3k</td>
<td>23%</td>
</tr>
<tr>
<td><strong>RWTH Unpruned</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>47.62%</td>
<td>19.23</td>
<td>2.6</td>
<td>86%</td>
</tr>
<tr>
<td>8</td>
<td>26.96%</td>
<td>350</td>
<td>35</td>
<td>90%</td>
</tr>
<tr>
<td>12</td>
<td>17.64%</td>
<td>2.7k</td>
<td>367</td>
<td>86%</td>
</tr>
<tr>
<td>16</td>
<td>14.86%</td>
<td>8.1k</td>
<td>1.6k</td>
<td>80%</td>
</tr>
<tr>
<td>20</td>
<td>14.45%</td>
<td>11.6k</td>
<td>2.6k</td>
<td>78%</td>
</tr>
<tr>
<td><strong>Attila Dynamic</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>43.0%</td>
<td>18</td>
<td>13</td>
<td>28%</td>
</tr>
<tr>
<td>8</td>
<td>27.5%</td>
<td>384</td>
<td>282</td>
<td>27%</td>
</tr>
<tr>
<td>12</td>
<td>22.71%</td>
<td>7.3k</td>
<td>3.9k</td>
<td>47%</td>
</tr>
<tr>
<td>16</td>
<td>22.61%</td>
<td>15.7k</td>
<td>6.8k</td>
<td>57%</td>
</tr>
<tr>
<td>20</td>
<td>22.59%</td>
<td>18.5k</td>
<td>7.5k</td>
<td>60%</td>
</tr>
<tr>
<td><strong>Attila Static</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>48.32%</td>
<td>49</td>
<td>11</td>
<td>76%</td>
</tr>
<tr>
<td>8</td>
<td>35.39%</td>
<td>283</td>
<td>74</td>
<td>74%</td>
</tr>
<tr>
<td>12</td>
<td>27.83%</td>
<td>1.1k</td>
<td>348</td>
<td>68%</td>
</tr>
<tr>
<td>16</td>
<td>23.46%</td>
<td>3k</td>
<td>1.1k</td>
<td>63%</td>
</tr>
<tr>
<td>20</td>
<td>21.04%</td>
<td>6.1k</td>
<td>2.4k</td>
<td>60%</td>
</tr>
</tbody>
</table>

Table 1: Effect of the redundancy removal algorithm on lattices from the RWTH decoder, RWTH decoder without word-end pruning, Attila dynamic decoder, and Attila static WFST decoder. We vary the lattice posterior pruning thresholds, and measure the oracle WER, the number of lattice arcs per second, the number of lattice arcs per second after redundancy removal, and the reduction ratio.

improved pitch extraction described in [14]. All systems use a 3-gram LM.

In Table 1 we see the effect of redundancy removal on lattices generated by the RWTH and Attila decoders. We first decode with a large beam, then we apply posterior-probability based lattice arc pruning, and afterwards we apply redundancy removal. The RWTH decoder generates lattices based on word-pair-approximation [1]. It basically records the recombination at word ends, and builds a lattice representing the pruned search space on-the-fly. It applies a noise closure filter already during decoding, thus the shown reduction is not related to redundant noise paths. We achieve a reduction of 23 to 56% in lattice density at equal oracle WER on the baseline RWTH decoder lattices. When we generate lattices without pruning word ends separately (eg. the RWTH Unpruned line), then the generated lattices are much more redundant, and we achieve a reduction of 78 to 90% through redundancy removal. The Attila dynamic decoder [8] generates lattices by storing a bigram-lattice-like index during decoding (also using word pair approximation), and rescoring it in a second pass. We achieve a reduction of the lattice size by 28 to 60% at equal oracle WER on these lattices. The Attila static WFST decoder generates lattices using the n-best method described in [7, 8], and redundancy removal reduces those lattices by 60 to 74%.

3.1. Cross-Site Comparison

In Figure 2 we compare the best results generated using the RWTH decoder, IBM’s Attila decoder, IBM’s Attila static

WFST decoder, and the Kaldi WFST decoder. The curves were generated by decoding with a large beam, applying different posterior-based lattice pruning thresholds, and then applying redundancy-removal. All systems achieve a similar single-best WER. These results need to be taken with a grain of salt, because the systems from different sites were created using very different methods. The Kaldi results could not be generated for larger pruning thresholds, because the WFST determination algorithm [2] was becoming problematic for such lattice sizes. This indicates that our novel redundancy removal is more efficient than the WFST based method, since we didn’t encounter efficiency issues for these lattice sizes.

4. Conclusions

We have introduced a novel redundancy removal algorithm, which efficiently reduces the size of lattices by 30 to 90% at equal oracle WER, over common redundancy-reducing methods like the word pair approximation. The algorithm is efficient even for large lattices, and can be implemented straightforwardly. It preserves arc identities, and can thus be used to filter the lattice based on arbitrarily mapped arc labels. Redundancy removal is specifically useful when using lattices as a persistent index, because disk space consumption can be reduced considerably.

5. Acknowledgements

Supported by the Intelligence Advanced Research Projects Activity (IARPA) via Department of Defense U.S. Army Research Laboratory (DoD / ARL) contract number W911NF-12-C-0012.1

1The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. Disclaimer: The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of IARPA, DoD/ARL, or the U.S. Government.
6. References


