Memory efficient approximative lattice generation for grammar based decoding

Miroslav Novak

IBM T.J. Watson Research Center
P. O. Box 218, Yorktown Heights, NY 10598, USA
miroslav@us.ibm.com

Abstract

ASR decoders are often required to produce a word lattice in addition to the best scoring path. While exact lattice generation is expensive (in the context of a Viterbi decoder), approximative algorithms are available to produce high quality lattices at much lower cost. Ideally, we would like to have an algorithm which does not require additional resources (either memory of CPU) in comparison to the best-path only decoder. We will present a lattice generation technique which uses a relatively strong approximation in comparison to other published techniques but requires little memory overhead (in comparison to a decoder optimized for best path only). We will show that the technique is suitable for tasks where a grammar is used as language model with little impact on the lattice quality (evaluated as n-best coverage).

1. Introduction

Word lattices produced by an ASR decoder in addition to the most likely word sequence have many practical uses. The lattice represents in a compact form the part of the search space with non-negligible likelihood. Lattices can be utilized by:

- techniques for confidence measure computation,
- discriminative training methods such as Maximum Mutual Information or Minimum Phone Error training,
- multi-pass decoding algorithms where a simpler and less expensive model (either acoustic, language or both) is used to produce a word lattice in the first pass and the more complex model is then used in the subsequent passes,
- efficient n-best generation, for example to apply additional (possibly complex) source of information to each alternative to determine the correct path in the n-best list.

Most of the state of the art decoders today use the Viterbi algorithm to find the most likely state sequence in the HMM network. While this algorithm is popular for its simplicity, it is not optimal for lattice generation. For an exact lattice generation, all possible word sequences and time alignments of boundaries between the words would have to be explored. Such algorithm would be prohibitively expensive and some approximation is always used. Previously reported lattice generation techniques [1],[2],[3] rely on a word-pair approximation, which assumes that the time boundary between two words depends on those words only and not on any previous words and requires that in the decoding graph all predecessors of any word are identical. Such condition is automatically satisfied for bi-gram and higher order language models (except for the backoff state), but would cause a significant increase in the graph size when finite state grammars are used. As an example consider recognition of alphanumeric strings. For a string of two symbols from an alphabet of 46 symbols, the grammar has $2 \times 46$ word arcs. To satisfy the bi-gram approximation, such a grammar would have $46^2$ arcs instead.

In previously reported lattice generation methods utilizing a “lexicon tree” minimization of the decoding graph is precluded. A method based on a phone-pair assumption has been reported [4], which eliminates these restrictions, so lattices can be produced in the context of decoding with a state minimized decoding network.

The rest of the article is organized as follows. First we will review previously published lattice generation techniques used in connection with the Viterbi search. Then we present our reasoning for the new algorithm and show how it can save the traceback memory. Then we will describe all steps of our algorithm and discuss its use in connection with garbage collection. Finally we present our results by comparing the memory needs of our method to the previously published ones.

2. Lattice generation

Before describing the algorithm, we will first analyze the memory needs of the previously reported algorithms. The amount of memory required to recover the best path depends on the required level of detail. Suppose we are looking for the best path as a sequence of arcs in the search graph $\alpha = [a_1, a_2, \ldots, a_n]$, aligned to a sequence of feature vectors $\sigma = [o_1, o_2, \ldots, O_T]$. The Viterbi algorithm finds a sequence which maximizes:

$$a' = \arg \max_{a} \prod_{i=1}^{n} P(a_{(t-1),t})|a_i| P(a_i), \quad (1)$$

where $x_{(t-1),t} \in \sigma$ is the part of the feature vector sequence aligned to the arc $a_i$ and $P(a_i)$ is the language model probability of that arc. The arc represents a sequence of HMM states, corresponding for example to a unique word or a particular pronunciation of a word.

For all incomplete arc sequences merging in a particular state $s$ a Viterbi decision is made which selects the path with the highest likelihood reaching the state $s$ at the time $t$. The likelihood of this arc is given by:

$$P(a_k, t', t) = P(a_{(t,t')}, a_k) P(a_k) = \frac{P([a_1, \ldots, a_k], 0, t)}{P([a_1, \ldots, a_k-1], 0, t')} \quad (2)$$

The best starting time $t'$ of the arc $a_k$ is determined by the Viterbi algorithm applied to the HMM states representing the...
Every time an ending state $s$ of an arc $a_k$ is reached, a traceback structure is created which records the time and likelihood of the path, identity of the state $s$ and a pointer to the previous state. The Viterbi decision performed at the state $s$ then selects the best arc from all arcs ending at the state $s$. A pointer $tr$ to the structure associated with the winning arc is then propagated forward to all arcs which leave the state $s$. When the final state of the decoding graph is reached at the end of the utterance, the best sequence is recovered by tracing the pointers from the final state back to the initial state.

If the best path is required at a higher level of detail, e.g. phones, the same algorithm can be used with arcs representing the corresponding HMM state sequences for each phone. This has a direct impact on the amount of memory needed - the traceback structures need to be created at the end of each phone.

The traceback structures can be used to generate a lattice. Starting from the final state, the traceback pointer determines the end of the previous arc $t'$ and the corresponding state $s'$. All traceback structures ending at the time $t'$ and state $s'$ contain enough information to insert a new arc into the lattice. In general, for a given traceback structure $[t, s, P, tr]$ a new arc is added to the lattice, from a lattice state $(tr \rightarrow t, tr \rightarrow s)$ to a lattice state $(t, s)$ with likelihood $P/(tr \rightarrow P)$.

From the memory use point of view, the most important observation we can make is that for lattice construction, one record needs to be created for each arc coming to a merge state (i.e. there is more than one arc leading to the state), unless we propagate all pointers till the end of the word (and increasing the cost of the search). The record is created at every time the likelihood of that arc is within some distance from the best path at that time.

In the lattice construction method [3] which uses tree lexicons, the merge state can only be the leaves of the tree, thus one record is created for each word end. When HMM state minimization is performed on the search network, the word internal states can become merge states as well. We need to emphasize here that when we apply the minimization, we do not allow the word labels to shift relatively to phone labels, so they always appear at the word ends. For proper lattice construction, traceback records need to be created for all internal merge states. While the minimization significantly improves the search of the best path by removing duplicated word internal states, it has no effect on the amount of memory needed for lattice construction. Can we exploit the effect of minimization for the lattice construction as well? The minimized graph has the property that no two arcs leading to any merge state at the boundary between words have the same label (the reversed network is deterministic). This property would potentially lead to a significant memory saving, if we were able to construct the lattice from records associated with those word label arcs only. We can estimate the potential for memory savings by analyzing the amount of internal states we can expect in a typical network.

They are two reasons for the appearance of internal merge states:

- internal states of two word arcs from two different LM states can be merged due to LM recombination,
- internal states of two word arcs from the same LM state but with different cross word contexts can be merged because the contexts became equivalent.

Consider the alphanumerics task again. The search network will contain only $2 \times 46^2$ arcs leading to internal merge nodes.

We can perform lattice construction from word end records only by further relaxing the word-pair assumption (in fact the phone-pair one as well) by allowing one word to have various predecessors. We need to find a way to recover lattice arcs lost due to merges in internal states. Before we describe the algorithm, we need to clarify that as unique word we in fact understand a unique pronunciation of a word, so for each word label there exists only one HMM state sequence.

### 3. Approximative algorithm

To explain the algorithm, we will consider a fragment of a grammar in figure 1 with word labels and log probabilities assigned to each arc. The corresponding HMM state network is shown in figure 2. This network contains non-emitting word end states (empty circles) and word internal emitting states (filled circles). As a result of the graph minimization procedure applied to the decoding graph, paths can merge in internal states. Suppose we would create traceback records only at non-emitting word end states, for each arc entering the state. The lattice arcs we would be able to create from such records are shown in figure 3 as solid thick lines. Clearly they don’t represent the complete lattice, there will be missing arcs shown as dashed thick lines. What can we infer about those arcs from the existing ones? Consider the arc between the states 2 and 3. To place an arc between states 2 and 6, 3, we need to find out:

- can the arc occur between the states? - that information can be recovered from the original word grammar,
- what is the arc likelihood? - we can approximate it by the likelihood of the existing arc between the states 2 and 3.

The likelihood of a lattice arc $l$ is a weighted combination of a language model and an acoustic model likelihoods.

$$
\log P(l) = \alpha_{ac} \times \log P_{as}(l) + \log P_{LM}(l).
$$

If we make an assumption that the HMM state sequence between states 2 and 3 and states 6 and 3 representing the same word will be similar (not necessarily equivalent due to the cross word context), than we can expect the acoustic score to be similar as well. To compute the likelihood of this arc $l'$, we need to compensate for the difference in the LM likelihoods:

$$
\log P(l') \approx \log P(l) - \log P_{LM}(l) + \log P_{LM}(l').
$$

There are two approximations involved in this approach:

- we assume that the boundary between two words does not depend on the identity of the previous word,
- we assume that the identity of the second word does not depend on the identity of the previous word.

We have experimentally verified the effect of this approximation and we will discuss it in the results section.

We need the original word grammar information in order to compute the estimate (4). We transform the word grammar into an auxiliary grammar in such a way that the arcs represent the unique word pronunciations rather than words and state numbers correspond to the word ends in the decoding graph. We use left cross-word context in our acoustic model, in that case this auxiliary grammar is deterministic. The left cross-word context constraints also creates several word end states for each state in the original grammar. The auxiliary grammar is stored in a
factored form, only one set of arcs is created for each original grammar state and is referenced by the context dependent state in the auxiliary grammar.

The size of the auxiliary grammar clearly presents an overhead in the memory use. But we do not consider this memory as critical, it can be considered as part of the decoding graph and requires read only access. This fact can be exploited in the actual deployment, it can be either put into ROM or can be shared among multiple threads or CPUs in a shared memory segment.

2. For each complete path with likelihood greater than $P_{\text{min}}$, insert a new arc into the lattice. Create a new lattice state if needed. Each lattice state is associated with time $t$ and traceback state id $s$ for which it was created.

3. For each inserted arc, update the backward score $P_b(q)$ of its starting lattice state $q$. The backward score is likelihood of the best path from this state to the lattice end.

4. Starting from the end of the utterance, for each time frame $t$:

   5. If there are no lattice states associated with this time, skip this time frame. Otherwise:

   6. For each path $p$ ending in the word end state which is the same as the state associated with the lattice state, add a new lattice arc if the product of the path score and backward score of that state is above the threshold $P_{\text{min}}$.

7. For each lattice arc starting at the time $t$, determine its word label and original word end state. Search all traceback records for time $t$, look in the auxiliary grammar for an arc with the same word label and ending state. If such an arc exists, insert a new arc into the lattice with a score compensated according to 4.

8. Repeat steps 5 to 7 till the time $t = 0$ is reached.

4. Garbage collection

Relatively few traceback records created during the search are used in the final resulting lattice. It is the purpose of garbage collection to reduce the amount of memory needed during the search by recycling the memory of records no longer needed. Garbage collection is run periodically to locate those records which can be recycled. If only the best path was required, then relatively simple algorithm could be used. First all records are marked as unused. Then at a particular time at which the collection is performed, all active paths are traced towards the beginning of the utterance and all visited records are marked as used. After all active path has been traversed, all records which are not marked as used can be recycled, and all records marked as used are reset as unused for the next garbage collection cycle.

To expand the garbage collection algorithm for lattice generation, we need to build the lattice incrementally, as part of the garbage collection cycle. The pruning algorithm used for lattice building is based on pruning away all paths with likelihood lower than a threshold relative to the best path. The most efficient pruning is achieved once the likelihood of the best complete path is known at the end of the utterance. When constructing the lattice in the middle of the utterance, we want to guarantee that all pruned records would not occur in the final lattice. That is not always possible to achieve if a grammar is used as a LM. The problem is that the path recombination may not take place until the very end of the utterance, the grammar may contain completely independent word sequences. A path with a very high likelihood at certain time $t$ can eventually drop bellow the pruning threshold if there is no good acoustic match in the rest of the utterance. We need to locate the recombination points explicitly. What we are looking for is the longest path which represents a common prefix for all active path. Then we can build a lattice for this common path using the likelihood of that path to set the pruning threshold - whatever happens to the active paths in the future will have no effect on the relative differences between the best common path and its alternatives in the lattice.
To locate a common path, we need to slightly modify the previous algorithm. First we initialize the common path end time to the current time of the search. Then during the first phase the garbage collection, every time we arrive at a record already marked as used, we take a note of the time associated with this record. If that time is less then the common path end time, it becomes a new common path end time. Before the recycling phase of the garbage collection procedure is performed, we can build a part of the lattice corresponding to the common path. For that part of the utterance, we can in fact recycle all records.

5. Results

For testing of our algorithm we have modified the EARS decoder [5]. As the baseline, we used a version which traces only the word best path and built our algorithm on top of it. We used a telephony acoustic model. Cepstral coefficients were generated at a 15ms frame rate with overlapping 25ms frames. Nine frames are spliced together and linearly-transformed and projected using LDA+MLLT into a 39 dimensional feature vector. A cross-word left-context pentaphone acoustic HMM model is built with 1080 states and 160000 Gaussians.

We have tested our algorithm on a variety of tasks, to test the performance with grammars of various vocabulary sizes and complexities. We measure the quality of the lattice by generating an n-best list from it and then computing the oracle string error rate on the n-best list of various sizes (sentence is considered as correctly decoded if it appears in the n-best list). We have found out that our approximation has very little negative effect on the n-best error rate across all tasks we have tested. In figure 4 we show a comparison between the lattice generation with full traceback record collection at all merge states and our approximative method, combined for all tasks.

Table 1 shows comparison in the amount of traceback records needed by the implementation for best path only, full traceback collection at merge states and the approximative method. The numbers shown it the table represent the number of allocated traceback records. We have measured these numbers in connection with the described garbage collection technique, the numbers show the allocated records which need to be kept between two garbage collection cycles (both average and maximum values shown for each task). The table shows clearly that the full method requires several times more memory than the approximative method. The approximative method itself needs only slightly more traceback memory than is needed to find the best path only.

In terms of additional CPU cost, our method is very efficient. The impact on the search is practically negligible, since the amount of word end traceback records is only slightly higher. The actual lattice building algorithm is very fast and depending on the task, it represents only few percent increase in the total decoding time.

6. Conclusion

We have presented an approximative method for lattice generation which requires little additional resources in the decoder in comparison to the task of finding the best-path only. Instead of increased memory needs for the traceback, we increase the amount of static memory associated with the decoding graph. In many situations, the need of the static memory is less critical. We have verified that the approximation we use causes very little degradation in the quality of n-best lists generated from the lattices.

7. Acknowledgements

The author would like to thank to Sreeram Balakrishnan, George Saon, Stanley Chen and Daniel Povey for useful discussions about lattice generation.

8. References