Memory efficient decoding graph compilation with wide cross-word acoustic context

Miroslav Novak  
IBM T.J. Watson Research Center  
Yorktown Heights, USA  
miroslav@us.ibm.com

Vladimir Bergl  
IBM Voice Technologies and Systems  
Prague, Czech Republic  
vladimir.bergl@us.ibm.com

Abstract

We present an efficient technique for the compilation of static decoding graphs which can utilize full word of left cross-word context. We put an emphasis on memory efficiency, in particular to be able to deploy this technique on platforms with limited resources. The incremental application of the composition process efficiently produces a weighted finite state acceptor which is globally deterministic and minimized with the maximum memory need during the composition essentially the same as that needed for the final graph.

1. Introduction

The use of static HMM state networks (search graphs) is considered one of the most speed efficient approaches to implementing synchronous (Viterbi) decoder. The speed efficiency comes not only from the elimination of the graph construction overhead during the search, but also from the fact that global determinization and minimization provides the smallest possible search space.

The use of finite state transducers has become popular in the speech community [1]. It provides a solid theoretical framework for the operations needed for search graph construction. A search graph is the result of a composition

\[ C \circ L \circ G \]

where \( G \) represents a language model, \( L \) represents a pronunciation dictionary and \( C \) converts the context independent phones to context dependent HMMs. The main problem with direct application of the composition step is that it can produce a non-deterministic transducer, possibly much larger than its optimized equivalent. The amount of memory needed for the intermediate expansion may be prohibitively large given the targeted platform.

Many techniques proposed for efficient search graph composition restrict the phone context to triphones [2], since the complexity of the task grows significantly with the size of the phonetic context used to build the acoustic model, particularly when cross-word context is considered. For large cross word contexts, [3] introduces auxiliary null states using a bipartite graph partitioning scheme. In the suggested approximative partitioning method the most computationally expensive part is vocabulary dependent. Determinization and minimization is applied to the graph in the subsequent steps.

Another technique [4] builds the phone to state transducer \( C \) by incremental application of tree questions one at a time. The tree can be built effectively only up to a certain context size, unless it is built for a fixed vocabulary. This method still relies on explicit determinization and minimization steps in the process of the composition of the search graph.

When we started to look for a method suitable for practical applications, we have set the following objectives:

- vocabulary independence
- maximal memory efficiency
- ability to trade speed for complexity

By vocabulary independence we understand the requirement that the vocabulary can be changed without significantly affecting the efficiency of the algorithm. In some situations the grammar \( G \) needs to be constructed before the recognition starts, defining the vocabulary. For example, in dialog systems the grammars are composed dynamically in each dialog state. In another case, the user is allowed to customize the application by adding new words. For this reason, we could not use algorithms which require expensive vocabulary specific preprocessing step.

Our main objective is to maximize the memory efficiency. We found this property more important than speed, since it is the limiting factor on the size of the model we are able to process. Though we had paid attention to the speed as well, it is a secondary factor, particularly in a statically compiled graph scenario. In such a case, a more complex model can be used, e.g. wider cross-word context. If speed is required as well, one can use a model with reduced context size to meet the requirements.

For our applications we found the use of the left cross-word context sufficient. We found very little evidence which would justify the increased complexity of
right context cross-word modeling. IBM acoustic models are typically built with 11-phone context (including the word boundary symbol), which means that within the word the context is ±5 phones wide in each direction and the left cross-word context is at most 4 phones wide.

We will first explain the basic algorithm for the word-internal context modeling case. Then we will show how it can be extended to left-only cross-word context models.

2. Incremental composition

Deterministic acyclic finite state automata can be built with high memory efficiency using an incremental approach. This has been shown on a task of building a dictionary [5]. Our graph is not necessarily acyclic (certainly not if it is an n-gram model), but we will show that cyclic graph minimization is not needed assuming that the grammar $G$ is provided in its minimal form.

The distinct feature of our algorithm is that the amount of memory needed to store the graph at any point will not exceed the amount of memory needed for the final graph. It should be understood that the actual graph representation during the composition requires more memory per state than the final representation during the decoding, but it is fair to say that the memory need is $O(S + A)$, where $S$ is the number of states and $A$ is the number of arcs of the final graph.

We have achieved this efficiency by using a somewhat simpler framework, using acceptors rather than transducers, which makes the operations such as determinization and minimization less complex. The main concept of our scheme is the combination of all steps (composition, determinization, minimization and weight pushing) into a single one.

We first construct a deterministic prefix tree $T$ which maps HMM state sequences to pronunciation variants of words (lexemes) in $G$. Each unique arc sequence representing an HMM state sequence is terminated by an arc labeled with the corresponding lexeme. We then replace all arcs leaving a particular state of $G$ by a subtree of $T$ with the proper scores assigned to the subtree leaves. We denote the operation which performs this replacement on all states of $G$ as $R_T(G)$.

The resulting FSA is deterministic. We will further show the we can incorporate the minimization (including weight pushing) step into the subtree selection algorithm so the resulting FSA is a minimized.

This minimization is done locally - it means that its extent is limited to the replaced subtrees and already existing subtrees leading to same target states of $G$. This is due to the fact that the algorithm preserves states and arcs of $G$. Precisely if $a$ and $b$ are two different states of $G$,

$$a \neq b \rightarrow \mathcal{L}(G_a) \neq \mathcal{L}(G_b)$$

$$\rightarrow \mathcal{L}(R_T(G_a)) \neq \mathcal{L}(R_T(G_b)),$$

(2)

Where $G(a)$ is maximal connected sub-automaton of $G$ with start state $a$, $\mathcal{L}(G_a)$ is the language generated by $G_a$. In another words, if $G$ is minimized, the algorithm cannot produce a graph which would allow the original states to merge. This has an important implication. To minimize the composed graph, we only need to perform local minimization, i.e. any two states of the composed graph need to be considered for merge only if they lead to the same sets of states of $G$. This minimization is acyclic and thus very efficient (there exist algorithms with complexity $O(N + A)$). The subtree selection algorithm is applied incrementally to each state of $G$. As the states of the subtree are processed, they are immediately merged with the final graph in a way which preserves the final graph minimal.

We need to mention that the minimized FSA may be suboptimal in comparison to its equivalent FST form, since the transducer minimization allows input and output labels to move. While this minimization can still be performed on the graph we have constructed, it is avoided for practical reasons as we prefer placement of the lexeme labels at the word ends.

The algorithm uses a post-order tree traversal. Starting at the leaves, each state is visited after all of its children had been visited. When the state is visited, the minimization step is performed, i.e. the state is checked for equivalence with other states which are already a part of the final graph. Two states are equivalent if they have the same number of arcs and the arcs are pair-wise equivalent, i.e. they have the same label, cost and destination state. If no equivalent state is found, than the state is added to the final graph. Otherwise the equivalent state is used. A hash table is used to perform the equivalence test efficiently.

Our implementation of the postorder processing algorithm needs to take into the account that only a subset of the tree, defined by the selected leaves corresponding to the active lexemes, needs to be traversed. The node numbering follows pre-order traversal. The index of each leaf corresponds to one lexeme. Each node also carries information about its distance to the root (tree level).

One aspect of the minimization we have not mentioned so far is weight pushing. It fits naturally in the postorder processing framework. The cost are initially assigned to the selected leaves. As the state of the prefix tree are visited, the cost in pushed towards the root using the algorithm described in [6].

We will now explain the subtree selection algorithm on an example shown in figure 1. At any step of the algorithm, each state can be in one of the three conditions: not visited (circle), visited but marked as waiting to be tested for equivalence (hexagon) or merged with the final graph (doublecircle). At any time, only one state in each level can be marked as waiting. In the step (a) active leaves of the tree are marked and assigned their LM scores (log-likelihoods). These leaves can be immediately marked as
merged, since they are part of $G$ and will appear in the final graph. Starting with the top leaf, the tree is traversed towards the root and all states along the path are marked as waiting. When the second leaf is processed, its parent state (3) is already marked as waiting, so the traversal towards the root stops there. In step (b), we look at the level of the parent state of the third leaf (6) (the state (7) is not visited because it does represent an active leaf). There already exists a marked state at that level, which is not a parent of this leaf. This means that all children of the marked state (3) have already been merged with the final graph, so this state can be added to the graph as well. The scores of all children states are pushed towards this state and the appropriate scores of all arcs are computed. After this state has been merged with the graph, the state (6) is marked as waiting at this tree level. The process is repeated for every parent until either the tree root or a waiting parent state is reached.

The same process is performed in step (c) as the last active leaf is processed. Finally in step (d), after all active leaves have been processed, all remaining waiting states are merged with the graph. It can be clearly seen that only those states which became a part of the final graph were visited.

The upper bound on the amount of memory needed to traverse the tree is proportional to the depth of the tree times maximum number of arcs leaving one state. The memory in which the tree is stored does not need write access and neither the memory nor the computational cost of the selection depends directly on the size of the whole tree. In situations where the vocabulary does not change or when a large vocabulary can be created to guarantee the coverage in all situations, the tree can be precompiled and stored in ROM or can be shared among clients through shared memory access.

3. Left context

In left cross-word context modeling, instead of one prefix tree we need to build a new tree for each unique context. The number of unique contexts theoretically grows exponentially with the number of phones across the word boundary. In practice, this is limited by the vocabulary. The number of phones inside a word which can be affected by the left context does not have a significant effect on the complexity of the algorithm.

The graph for left context can be build using the same incremental technique as was used for the word internal context by applying the whole algorithm for each unique context. The complete algorithm has the following steps:

1. Construct a set of all left context classes given all active lexemes and the cross-word context size. Create a map $C(l)$ which assigns a context class to each lexeme.

2. For each context class $c$ build the tree $T_c$ and apply the subtree selection algorithm to each state $s$ of $G$ affected by this context. Insert the root of each subtree into a map $M(c, s)$.

3. For each arc in the final graph with a lexeme label $l$, change its destination from $s$ to $M(C(l), s)$.

The set of left context classes is constructed by simply enumerating all phone $k$-tuples observed in all lexemes. This is an upper bound as some phone sequences will have the same left context effect. As the graph is built, those classes with a truly unique context will be automatically found by the minimization step. For this reason it is important to perform the connection of each lexeme arc to its corresponding unique tree root in a separate final step, after all trees for all contexts have applied to the graph.
Table 1: Compilation time (in seconds) for various left context sizes

<table>
<thead>
<tr>
<th>context-size</th>
<th>grammar</th>
<th>n-gram</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>34</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td>51</td>
</tr>
<tr>
<td>2</td>
<td>57</td>
<td>216</td>
</tr>
<tr>
<td>3</td>
<td>314</td>
<td>1306</td>
</tr>
<tr>
<td>4</td>
<td>767</td>
<td>3560</td>
</tr>
</tbody>
</table>

Table 2: Comparison between the upper limit and the actual number of unique contexts in a vocabulary constrained system

<table>
<thead>
<tr>
<th>context size</th>
<th>grammar</th>
<th>n-gram</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>limit</td>
<td>effective</td>
</tr>
<tr>
<td>1</td>
<td>51</td>
<td>49</td>
</tr>
<tr>
<td>2</td>
<td>966</td>
<td>872</td>
</tr>
<tr>
<td>3</td>
<td>5196</td>
<td>3911</td>
</tr>
<tr>
<td>4</td>
<td>12028</td>
<td>6069</td>
</tr>
</tbody>
</table>

4. State Equivalence test

For state equivalence testing performed during the incremental build, we use a hash table. The state is represented by a set of arcs, each arc is represented by a triple \((\text{destination state}, \text{label}, \text{cost})\). To minimize the amount of memory used by the hash table, we have implemented the hash as an integral part of the algorithm. In a typical stand-alone hash implementation the key value must be stored in the hash table for conflict resolution. This would effectively double the amount of memory needed to store the graph. Our memory structure for the graph state representation contains records related to the hashing, i.e. a pointer for the link list construction and the hash lookup value (the graph state id). This way, the hashing adds only 8 bytes to each graph state.

5. Experimental results

We will illustrate the effect of the context size on the compilation speed for two tasks. The first task is a grammar (list of stock names) with 8335 states and 22078 arcs. The acoustic vocabulary has 8k words and 24k lexemes. The second task in a n-gram language model (switchboard task) with 1.7M of 2-grams, 1.2M of 3-grams 86k of 4-grams, with a vocabulary of 30k words and 32.5k lexemes. The compilation time was measured on a linux workstation with 3GHz Pentium 4 CPUs and 2.0 GB of memory and is shown in table 1.

It is clear the the efficiency of the algorithm suffers when the context size increases. One way to speed up the computation for large context is to relax the vocabulary independence requirement and precompute the effective number of unique contexts. Given a fixed vocabulary, the number of contexts is limited by the number of unique combinations of last \(n\) phones of all words. But some of the contexts will have the same cross word context effect. For those contexts only one context prefix tree needs to be build. Table 2 compares the limit and effective values of context classes on both tasks. The effective value can be found as the number of tree roots in an expansion of a unigram model. This expansion is in fact a part of any backoff n-gram graph compilation and represents the most time consuming part of the expansion.

We used a much larger n-gram model to test the memory needs of the algorithm. While keeping the total memory use below 2GB, we were able to compile a language model into a graph with 35M states 85M arcs.

6. Conclusion

We have presented a method for memory efficient decoding graph construction. By eliminating the intermediate steps, the memory need of the algorithm is proportional to the number of states and arcs of the final minimal graph. This is very computationally efficient for short left crossword contexts (and unlimited intra-word context size), but it can also be used to compile graphs for a wide left cross-word context without sacrificing the memory efficiency.

7. References


