FAST HIERARCHICAL GRAMMAR OPTIMIZATION
ALGORITHM TOWARD TIME AND SPACE EFFICIENCY

Jing Zheng, Horacio Franco

Speech Technology and Research Laboratory, SRI International
333 Ravenswood Avenue, Menlo Park, CA 94025, USA
(zj,hef)@speech.sri.com

ABSTRACT

We present an algorithm for hierarchical grammar optimization achieving time and space efficiency in Automatic Speech Recognition (ASR). The algorithm includes two parts: a selective grammar expansion algorithm and a graph reduction algorithm. The latter algorithm includes a node-merging procedure and an arc-sharing procedure. The algorithm is general so as to handle all types of grammar used in ASR; it is efficient so as to process dynamically generated grammars online; it is flexible and accepts several parameters controlling tradeoff between time and space complexity in favor of various different application requirements. We used this algorithm to optimize a grammar representing a class-based trigram language model, and obtained significant grammar size reduction and a 140% recognition speedup.

1 INTRODUCTION

Hierarchical grammar structure, in which a parent grammar can reference to subgrammars recursively, provides great flexibility and space efficiency in Automatic Speech Recognition (ASR). In SRI’s DynaSpeak™ [1] small-footprint continuous speech recognition system, this grammar structure is supported in a way that allows subgrammars to be expanded on demand during search, so that only activated grammars are kept in memory to achieve space efficiency. However, for time-critical applications, this search strategy has two disadvantages. First, maintaining dynamic search space brings computation overhead. Second, since grammars are expanded dynamically, there is no way to perform intergrammar optimization that merges duplicated search paths for achieving faster recognition. Generally speaking, using hierarchical grammars with dynamic expansion is slower than using fully expanded grammars, while it usually has a lesser memory requirement.

To obtain a flexible tradeoff between time and space efficiency for satisfying different application requirements, we developed an algorithm for optimizing hierarchical grammars. The design goal of this algorithm is to process generic grammars, to allow a user to select the desired speed and space compromise, and to be time efficient for applications with a real-time requirement. The algorithm has two parts: an expansion algorithm that allows users to selectively expand subgrammars with certain criteria and a graph reduction algorithm that compacts grammars’ sizes through merging graph nodes and using NULL nodes to share common arcs under certain control.

The paper is organized as follows. In Section 2, we describe the grammar representation format. We describe the two parts of the optimization algorithms in Section 3, with illustrations. In Section 4 we give an application example. Finally, a summary is given in Section 5.

2 GRAMMAR REPRESENTATION

Inside DynaSpeak™ [1], a grammar is represented as a weighted directed graph. In such a graph, nodes are divided into two categories: real nodes and NULL nodes. Each real node is labeled a word or a subgrammar, while a NULL node is empty of content. Each arc is labeled a weight that takes the form of a negative logarithm of transition probability. Every sentence accepted by the grammar corresponds to at least one path from the initial node to the final node. This grammar representation is very general, and supports almost all types of grammars used in ASR. In addition, it is mutually convertible with another popular form of representation, the Weighted Finite State Automata (WFSA), in which symbol outputs are associated with arcs instead of nodes [2].

3 ALGORITHM DESCRIPTION

3.1 Selective grammar expansion algorithm

Dynamic subgrammar expansion brings overhead in the search and impedes effective optimization of the search space. To improve search efficiency, one solution is to remove the hierarchy of the grammar structure by iteratively expanding grammar nodes, until the whole grammar becomes a flat word graph.

This solution has some limitations. First, not all the hierarchical grammars can be flattened down to word graphs; for example, grammars with self-reference cannot be flattened. Second, the resulting word graph could be very large so as to cause memory inefficiency. To address these problems and to allow more flexibility in choosing a tradeoff point between speed and space, we propose a selective expansion approach, in which only those subgrammars satisfying certain criteria are expanded. The criteria we use are that:

- The number of nodes of subgrammar must be less than a specified threshold $TH_{1}$ to avoid severe memory inefficiency in expansion.
- The subgrammar must not have self-reference or involve loop reference.
The subgrammar is not in a specified protection list, which is set up to reserve some special subgrammars from being expanded for application purposes. This selective expansion algorithm usually does not completely remove hierarchy from grammars, but can effectively make room for further search space optimization. Reserving a certain degree of hierarchy helps reduce memory requirements. The parameter $TH_{1}$ gives the user flexibility to select different tradeoffs between computation and memory complexity. Note that this algorithm should apply to all those subgrammars being reserved during the expansion procedure.

3.2 Graph reduction algorithm

After grammar expansion, we want to find compact representations for each of the reserved grammars to improve speed and space efficiency. A well-known approach is to view a grammar as a WFSA, and perform a determinization/minimization algorithm [2]. A deterministic WFSA has a speed advantage given the input symbol string, generally with a computation complexity linear to the input size, and it has a unique minimal equivalent form, which can be found by an efficient minimization algorithm [3]. However, we did not use this approach because of the following reasons: First, in the context of ASR, the speed advantage of deterministic grammars is less obvious. Since the underlying word string is unknown, we generally need to test all possible word strings allowed by the grammar against the speech observations via the Viterbi algorithm, whose complexity relates to a grammar’s determinism as well as a grammar’s size. As a matter of fact, the minimal deterministic WFSA is usually not the minimal equivalent WFSA. Second, not all WFSAs are determinizable, and for a determinizable WFSA, the complexity of the determinization process is exponential in general. Finally, our grammar representation is not in WFSA format; bidirectional conversions are needed in order to apply the WFSA algorithms.

In consideration of our goal for a time-efficient grammar optimizer, we propose an algorithm that aims at reducing grammar size, though not necessarily to the minimal size, with light computational load.

3.2.1 Node merging

Under certain condition, some nodes with identical labels can be merged without changing the function of a grammar that maps a string into a weight. The merged node from this operation has the union set of the original nodes’ predecessors and successors, and the new arcs from and to the merged node take minimal possible weights. In [4], an algorithm based on a similar principle is presented to compact word lattices (directed acyclic word graphs). Here we propose an algorithm that can deal with general directed graphs.

It is easy to prove that nodes with identical labels and satisfying one of the following conditions can be merged into one node without changing a grammar’s function:

1. With same set of successor/predecessor and corresponding transition weights;
2. With same self-transition weight, and same remaining set of successor/predecessor and corresponding transition weights excluding the self-transition.

The basic idea is to reduce the size of a grammar through admissible node merging. To simplify the implementation, we divide the algorithm into two symmetrical passes: The forward pass considers outgoing arcs only; the backward pass considers incoming arcs only. For simplicity, the following description covers only the forward pass.

The major computation load of this algorithm comes from finding nodes to merge. If simply comparing every pair of nodes, the complexity will be $O(n^2)$, where $n$ is the number of nodes in the grammar. And even worse, since each merging could generate a new node that can be merged with other nodes, this procedure has to iterate repeatedly until no pair of nodes is found mergeable. This simple approach is clearly not feasible for large grammars.

To achieve high-speed efficiency, we have the following solutions. First, we use sorting to speed up finding mergeable nodes. Considering the forward-pass case, when we test Condition 1 on the graph nodes, we associate a node $i$ with the following feature string:

$$F_{1i} = (L_i, N_i, s_{i1}, s_{i2}, \ldots s_{ij}, w_{i1}, \ldots w_{ij})$$

where $L_i$ is the label of node $i$ (we map labels to integers), $N_i$ is the node’s out-degree, $s_j$ is the index of the $j$-th successor, and $w_j$ is the weight of the $j$-th arc.

For each node, we sort its outgoing arcs by indexes of their destination nodes. Thus, each node has a unique feature string $F_{1i}$. Note that the nodes with identical labels satisfy Condition 1 if and only if they have identical feature strings $F_{1i}$. We can sort all nodes in the lexicographic order of their feature strings; then, all mergeable nodes are automatically grouped together. In this way, the complexity in comparison computation can be substantially reduced.

We used a very efficient algorithm based on a revised bucket sort algorithm [5] for lexicographic sort, which can find all groups of mergeable nodes with computation complexity linear to grammar size. This algorithm is very similar to the one in [6], which was used for acyclic deterministic automata minimization.

The following feature string is defined for Condition 2:

$$F_{2j} = (L_j, N_j, w_0, s_1, w_1, \ldots s_{j-1}, w_{j-1}, w_j)$$

where $w_0$ is the self-transition weight node $i$, if the node does not have a self-transition arc, $w_0$ is assigned to –1. $N_j$, $s_j$, $w_j$
have the same definition as in (1), except that they are for the successors other than the node itself.

By performing the lexicographic sort on the whole node set of a grammar, we can pick out all groups of mergeable nodes and merge them. However, this procedure could not end at this stage, since the newly generated nodes could lead to new possible merging, and thus this algorithm has to iterate until no nodes are found mergeable. The topological order of nodes can be used to avoid iteration for acyclic graphs [4][6]; however, that order does not exist for cyclic graphs. We use the following fact to reduce unnecessary computations: After merging of all suitable nodes in the current graph, only the predecessors of those nodes being merged could contain new mergeable nodes (this is for the forward pass; for the backward pass, the procedure is the same, but for the successors). Therefore, the next round of lexicographic sort can be performed only on the specific node subset, which is usually much smaller than the whole node set.

After iterative node merging based on Conditions 1 and 2 in both forward and backward passes, we can guarantee obtaining a more compact (or at least not larger) equivalent representation of the original grammar. The following two procedures can make this algorithm more effective for search efficiency. The first one is pushing, which redistributes weights on the graphs as close to the initial node as possible, while keeping the total weights for any path from the initial node to the final node unchanged. Pushing will not change a grammar’s function, but will make the pruning of the search algorithm more efficient. Another important effect of pushing is that it could make more nodes mergeable, and further improve the compacting rate. We used the pushing algorithm in [2], which was originally proposed for weighted deterministic FSA minimization.

The second procedure is called NULL removal. As we know, real nodes in a grammar, which represent words or subgrams, correspond to major computation load during recognition. However, the existence of NULL nodes could prevent real nodes from being effectively merged, and NULL nodes themselves could bring extra overhead in the search. So for the sake of speed efficiency, we remove NULL nodes based on the following facts: Any arc toward a NULL node can be replaced by the arcs transiting from its predecessor and the NULL node’s successors, without changing the function of the grammar. However, NULL removal could significantly increase the grammar’s size, lead to heavy memory load, and slow down the node-merging procedure under certain situations. To control this effect, we introduce another threshold $TH_2$, and remove a NULL node only when its out degree is less than $TH_2$. The parameter $TH_2$ is important for handling certain types of grammars, in which NULL nodes have an important role in reducing grammar size.

Considering the symmetry of operations between the forward pass and the backward pass, we apply a single-pass algorithm to graphs in the original and the reverse direction to fulfill the two-pass node merging. The computation for graph reversal is trivial since we maintain incoming and outgoing arc tables all the time.

**Single-pass node merging algorithm:**

1. **Pushing** (for weighted grammars only)
2. Merge nodes based on feature string $F1$
3. Merge nodes based on feature string $F2$ (for grammars with self-transitions only)
4. If merging happened, repeat step 2 on predecessors of nodes being merged

**Complete node merging algorithm:**

1. Selective NULL removal; (controlled by $TH_2$)
2. Reverse graph
3. Single-pass node merging (the backward pass)
4. Reverse graph
5. Single-pass node merging (the forward pass)

The above node-merging algorithm generally guarantees reducing the number of nodes, but not the number of arcs because of the NULL removal procedure. If memory is the most important factor to be considered, using $TH_2 = 0$ to disable NULL removal can also guarantee arc reduction.

**3.2.2 Arc sharing**

People often use NULL nodes to reduce the number of transitions in grammar design. In the node merging procedure, NULL nodes are selectively removed in favor of merging real nodes. It is natural to consider reintroducing some NULL nodes to reduce grammar size after the node-merging procedure. The principle of this approach is explained as follows: If $n$ nodes have a common subset of $m$ successors and transition weights, then we can introduce a single NULL node, with $m$ outgoing arcs transiting to the $m$ successors. Then each of these $n$ nodes uses one 0-weighted arc toward the newly introduced NULL node to replace its original $m$ arcs. In this way we can save $(n-1)-(m-1)–1$ arcs in a row at the cost of one extra NULL node. If $m$ and $n$ have big values, the number of reduced arcs can be significant. We call this operation arc sharing. Introducing NULL nodes could bring overhead in the search. To limit this disadvantage and simplify the algorithm, we confine this operation to those nodes having a common set of successors, with an extra condition that one introduced NULL node must save at least $TH_3$ arcs, where $TH_3$ is a threshold set by the user to control the procedure. To find the nodes that can share their arcs, we use the same lexicographic sort algorithm as earlier, with the feature string defined as

$$F3_j = (N_j, \tau_1, w_1, \tau_2, s_j, w_j, \cdots \tau_{n_j}, w_{n_j})$$

The only difference between (3) and (1) is the missing item $L_j$. For each group of nodes found by the sort algorithm, we perform arc sharing only if the resulted arc reduction is larger than or equal to $TH_3$. Like other parameters, tuning $TH_3$ can achieve a flexible compromise between speed and size. Unlike in node merging, the lexicographic sort here needs to be performed only once, on the grammars from node merging.

**3.2.3 Illustrations**

Figure 2 illustrates the effects of main steps in the proposed graph reduction algorithm. For simplicity, only the forward-pass node-merging step is exhibited. Note that the arc-sharing step is just for exhibition, since it reduces only one arc, but brings in a NULL node. In practice, the extra NULL should not be introduced. The whole procedure reduces a graph with 3 real nodes, 3 NULL nodes and 14 arcs, to a graph with 2 real nodes, 3 NULL nodes and 7 arcs.
4 APPLICATION EXAMPLE
The presented algorithm has been used in many tasks; here we introduce one using a class-based trigram back-off language model [7] with 16 classes and a vocabulary of 2,290 words. Each of these classes contains a certain number of weighted word strings. The largest two classes each have more than 1,700 word strings.

We first used the SRILM toolkit [8], compiling this language model (LM) into the PFSG format, and then converted it to the DynaSpeak hierarchical grammar format. In the initial grammar, each word in the LM corresponds to a subgrammar, which contains the word itself and a following pause filler grammar representing an optional self-looped pause word. Each class is naturally represented as a subgrammar, with parallel paths for different word strings. The trigram LM is represented by a single grammar, in which each real node represents a class or a word. NULL nodes are used in this grammar for efficient representation of back-off transitions.

Clearly, this grammar is not efficient for search because of too much unnecessary hierarchy and redundancy in the search space. We used the proposed algorithm to optimize this grammar with appropriate parameter settings: $TH_1 = 5$, $TH_2 = 5$, $TH_3 = 128$. In addition, we deliberately reserved all grammars for classes, since they could be updated online. The whole algorithm finished within 1.5 seconds in a 1 GHz Pentium-3 PC with the Sun/Solaris operation system.

We tested both the initial and the optimized grammars using the DynaSpeak recognizer on a test set with 318 sentences and 3,042 words in the same computer. Perplexity of the test set is 40; OOV rate is 2%. As Table 1 shows, optimization brings a 140% speedup and a small reduction of word error rate (WER). After optimization, only 18 out of 2,309 grammars need to be kept. The node reduction rate is about 66%. The arc reduction rate, about 13%, is less impressive at first sight, but considering the big reduction of hierarchy it is actually considerable.

Table 1: Comparison between grammar before and after optimization.

<table>
<thead>
<tr>
<th></th>
<th>#gram.</th>
<th>#nodes</th>
<th>#arcs.</th>
<th>x RT</th>
<th>WER</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial</td>
<td>2,309</td>
<td>21,219</td>
<td>31,143</td>
<td>2.4</td>
<td>30.0%</td>
</tr>
<tr>
<td>Optimized</td>
<td>18</td>
<td>7,412</td>
<td>26,999</td>
<td>1.0</td>
<td>29.5%</td>
</tr>
</tbody>
</table>

5 SUMMARY
We have described in detail an algorithm for hierarchical grammar optimization. The algorithm first performs a selective expansion procedure to reduce the hierarchy of an input grammar, and then a graph reduction procedure to achieve a compact representation of the grammar without changing its function. The algorithm accepts several parameters to offer a user flexibility to select the desired speed and memory tradeoff according to specific application’s requirements. The algorithm is designed to be very time efficient, allowing to perform on-the-fly optimization of complex dynamic grammars. The algorithm resulted in a significant recognition speed improvement in a real application.

6 REFERENCES