WFST Compression for Automatic Speech Recognition

Diamantino Caseiro

AT&T Labs-Research, 180 Park Avenue, Florham Park, NJ 07932, USA

ABSTRACT

The large size of Weighted Finite-State Transducers (WFSTs) used in Automatic Speech Recognition (ASR), such as the language model or integrated networks, is an important problem for many ASR applications. To address this problem, we present a general purpose compression technique for WFSTs that is specially designed for the finite-state machines most commonly used in ASR. Experiments run on two large tasks show the method to be very effective, typically reducing memory and disk requirements to less than 35%. By combining it with “on-the-fly” composition, the memory requirements are further reduced to below 14%. These reductions show no negative impact on recognition speed.

Index Terms— Weighted Finite-state transducers, language model, compression, ASR

1. INTRODUCTION

Finite-state machines, and in particular Weighted Finite-State Transducers have become a fundamental tool in speech and language processing. In automatic speech recognition, they allow an elegant integration of knowledge sources such as context dependency, the pronunciation lexicon and the language model in a precompiled and very efficient search network [1].

One side effect of precompiling a static search network is that its size can be very large, thus requiring large amounts of memory at run time. Adding memory may be a minor problem in a research environment, however it may be expensive or not at all possible for small devices. Most importantly, recent distributed ASR applications, such as [2, 3, 4], require multigigabyte search networks that are multiplied across large server farms to handle high numbers of clients. These systems are very expensive to upgrade.

The WFST size problem has been approached from various angles. One is the compilation method itself that uses weighted determination and minimization algorithms to remove redundancy from the network and reduce its size [1]. The main contributor to the size of the integrated network is the language model. It is natural that many approaches focus on reducing its size, either by pruning it of $n$-grams with little impact on perplexity [5, 6], by block compression [8] or lossy methods [9]. Among the approaches that directly address the problem of WFST compression, we can include [10] that uses a variable length arc representation, [11] that represents the transducer as an acceptor and [12] that combines a constant length packed arc representation with weight quantization. The approach described in this paper is similar to [10] in that it uses a variable length arc representation, but it is more general and extensible. Its main innovation is that it searches through an extensible collection of alternative field transformations and packing methods for the combination that results in the most compact representation of a particular WFST.

The approach is described in detail in the next section. Section 3 presents various compression and recognition experiments. Finally, in section 4 the main results are summarized and we point to future research.

2. PROPOSED METHOD

The proposed compression method was designed with some constraints in mind. First, it should be effective for WFSTs with very large number of arcs and/or states, and with vocabulary sizes of millions of words. Second, incremental expansion of random states must be possible and very efficient. Third, the format can relabel the states of the WFST, but must preserve the order of arcs leaving each state. Finally, the file format must be memory mappable to reduce loading latency.

A weighted finite-state transducer $T = (\Sigma, \Delta, Q, i, F, E, \rho)$ over a semiring $K$, can be formalized as a tuple where $\Sigma$ and $\Delta$ are respectively input and output alphabets, $Q$ is a finite set of states, $i \in Q$ is the initial state, $F \subseteq Q$ is the set of final states, $E \subseteq Q \times \Sigma \times \Delta \times K \times Q$ is a finite set of arcs, and $\rho : F \rightarrow K$ is the final weight function associating a weight with each final state. Unweighted transducers and acceptors can be similarly defined by omitting weights and/or output labels. Some algorithms require a potential weight $\tau : Q \rightarrow K$ associated with each state. A semiring $K = (\oplus, \otimes, 0, 1)$ is a ring that may lack negation, and has two associative operators $\oplus$ and $\otimes$ such that $\otimes$ distributes over $\oplus$, $0$ and $1$ are the identity elements of $\oplus$ and $\otimes$, and $0$ is an annihilator. In speech recognition, weights often represent probabilities or their logarithm. In this paper we are only concerned with real value weights, and the tropical and log semirings [1].

2.1. Baseline WFST representation

The size of the WFST is often dominated by the set of arcs $E$. Most WFST systems implement an arc as a record with 4 fields: destination state, input and output labels, and a cost. The first 3 fields are 32 bit integers and the last is a 32 bit float1. The origin state is usually implicit. Frequently states are stored in an array of records (or in parallel arrays) that are indexed using the state id. Each state record contains: the number of arcs leaving the state; a pointer to a vector of arcs leaving the state; it’s final weight $\rho$ (states $s$ such that $\rho(s) = 0$ are considered not final); and a state potential weight. Using this representation, a WFST with $n$ states and $m$ arcs occupies around $16 \times m + 12 \times n$ or $16 \times m + 16 \times n$ bytes, depending on whether 64 bit pointers are used.

1Some libraries such as openFst [13] allow for the specification of different field sizes and types during c++ compilation.
## 2.2. State encoding

### 2.2.1. Arc indexing

To allow access to the arcs that follow a random state, the baseline representation requires 8 or 12 bytes per state. To reduce this value, we propose a paged indexing method. We group states into pages containing all the arcs of k consecutive states (typically 128 or 64). The address of the first arc leaving state $s$ is easily computed from the address of the arcs of state $s + 1$. When an offset is larger than $2^{15}$, it is stored in a secondary page $[s/k]+offset64[-offset[s%k]]$. If, during compression, $offset64$ grows above $2^{15}$ entries, the page size $k$ is halved, and the index is rebuilt. In ASR applications, the size of $offset64$ is small, and, in practice, this scheme reduces memory requirements to less than 3 bytes per state.

### 2.2.2. Final states and Potentials

It is assumed that only a small fraction of states are final (i.e. $\rho(s) \neq 0$). The final state information is encoded as a vector of pairs ($state, cost$) sorted by state id, and binary search is used to determine if a state is final.

In the proposed format, the state potential is represented as a weight vector indexed by the state. Since potentials are often not used, this vector is only allocated if any state has a potential different from the default value 0.

## 2.3. Arc encoding

Our arc encoding method is based on 2 steps. In the first one, a field specific transformation function is applied to reduce the average magnitude of each field. The goal is to represent each field by the smallest possible absolute value.

In the second step, the field is converted to a variable length sequence of bytes using a packing function. Like [10], our method is based on variable length arc encoding, in which the number of bytes used to represent a given field can vary from arc to arc. Any such variable length representation, requires the implicit or explicit allocation of some control bits to represent the size of each instance of the field. The optimal balance between the number of bits used the represent the field size and the field itself depends on the distribution of field values. We allow 2 alternative packing functions. The first one, byte code, encodes each 7-bits of the original value in one byte, using the remaining bit as an end of sequence indicator. The second one uses an explicit bit field to control the size of the sequence. Both functions use the least significant bit of the sequence to store the sign of the value.

Arcs are stored as a byte sequence where the first byte is a control byte containing 5 bit fields: the first 4 encode the number of bytes used by each field, while the 5th is filled with the least significant bits of the destination states. The size of the first 4 fields can vary from 0 to 2 bits, and it is optimized for the particular WFST. A variable number of bytes representing each field follows the control byte.

### 2.3.1. Destination state

One of the most effective compact graph representations consists of representing the destination of each arc either as the difference from its origin or from the destination of the previous arc with the same origin [10, 14]. State ids can be permuted to minimize these differences, however, the problem of finding the permutation that globally minimizes the difference is np-complete [15]. Our current implementation simply renumbers states in the order of a depth-first graph traversal. One effect of state renumbering is that it increases memory locality, because states with close ids are stored closely. This may have a positive effect on model loading reducing disk access to a memory-mapped file, and on run time processor cache usage.

Our destination transformation functions include:

- $abs$, the unchanged state id,
- $delta$ orig, difference from the origin state,
- $delta$ sib, difference from the destination of the previous arc.

### 2.3.2. Input and output label

For indexing purposes, the vector of arcs following a given state is normally sorted either by input or output label. Encoding each label as the difference from the previous arc can be a very effective way of reducing its dimension. Another effective encoding consists of resorting the vocabulary by frequency and using a look up table. Our label transformation functions include:

- $abs$, the unchanged absolute value is stored,
- $delta$, the difference from the previous label is stored,
- $table$, a look up table is used,
- $acceptor$, the output label is the same as the input label.

### 2.3.3. Arc cost

For many ASR applications, weights can be significantly quantized with little or no accuracy impact. Language models, in particular, can be quantized to as little as 4 bits [7, 12, 8]. Our weight quantization defaults to 16 bits, and to maximize its effectiveness, quantization is performed between the maximum (non $\bar{0}$) and minimum costs of theWFST: $quant(c, bits) = \lfloor 2^{bits}(max - c)/(max - min) \rfloor$

- $abs$, the unchanged value is stored,
- 8 bits, quantized to 8 bits,
- 16 bits, quantized to 16 bits.

## 2.4. Arc encoding method selection

The algorithm that selects the best combination of transformation and packing functions does 2 passes through the input WFST. The first pass is a depth first pass that assigns a new integer id to each state in order of discovery. The goal is reducing the numeric difference between the origin and destination of each arc. During this pass, some global statistics such as the minimum and maximum arc weights, and label frequencies are collected. These statistics are required by some transformation functions, and are used in the second pass.

The second pass collects statistics that will enable computing the final size of the WFST for a given combination of transformation and packing functions. In detail, this pass counts how many instances of a field can be encoded with a particular number of bytes (0 to 4) for a given combination of transformation and packing functions.

1494
The arcs of sequences to words, finally, is the optimized composition \( \text{CLG} \) of maps HMMs to context-independent phonemes. In Tables 1 and 3, we see that the proposed method is very effective, reducing very large WFSTs to around 1/3 of their original size. The compression ratios are comparable to gzip -9, which cannot be used in run time\(^2\).

Tables 2 and 4, show the average size of each field including control bits, and the transformation functions selected. We see that the method can easily exploit missing or constant fields. It is also doing well encoding monotonic and sparse fields, such as input and output labels. The delta transformations are most often chosen, except for small vocabularies or numbers of states. We also observe that the best packing function and number of control bits assigned to each field varies with each WFST.

### 3. EXPERIMENTS

#### 3.1. Compression

The effectiveness of the proposed method was evaluated on two large-vocabulary ASR applications. The first one is a voice-search ASR application \(2\) with a vocabulary of 212k words and with an acoustic models containing 44k HMMs, the second one is a questions-answering application \(3\) with a 515k word vocabulary and the same acoustic model. In both tasks, the 3-gram language model was pruned \(5\) due to memory concerns and in both significantly better accuracy is obtained with larger models.

The size of various WFSTs built for these applications are shown in Tables 1 and 3. These particular WFSTs were selected because of their different properties: \( G \) is the language model, and is a weighted acceptor. \( L \) is the pronunciation dictionary, and it is an unweighted transducer mapping sequences of context-independent phonemes (210 different labels) to words. \( C \) is an unweighted transducer that maps HMMs to context-independent phonemes. \( \text{CLG} = \text{opt}(L \circ G) \) is the optimized composition \(1\) of \( L \) and \( G \) and maps phoneme sequences to words, finally, \( \text{CLG} = C \circ \text{CLG} \) maps HMMs to words. The arcs of \( \text{CLG}, L \) and \( G \) are sorted by input label, and those of \( L \) and \( C \) by output label. Of these transducers, only \( \text{CLG} \) is required in memory during first-pass decoding. In Tables 1 and 3, we see that the proposed method is very effective, reducing very large WFSTs to around 1/3 of their original size. The compression ratios are comparable to gzip -9, which cannot be used in run time\(^2\).

#### 3.2. “On-the-fly” Composition

In tables 1 and 3 one can see that \( LG \) is 2.5 to 3x smaller than \( \text{CLG} \), these are typical ratios that we observe across many applications. Furthermore, composition is an online operation that can accept a lazy or “on-the-fly” implementation which generates states and arcs as they are needed by a client program such as the ASR decoder. With this in mind, we developed a “delayed composition” file format that stores 2 WFSTs in the same file, and builds their lazy composition when loaded. This format can be used to store \( C \) and \( LG \) and effectively represents \( \text{CLG} \) when loaded by the decoder. The advantage is that disk and memory requirements are much smaller, but decoding may be slower because \( C \) and \( LG \) are composed at recognition time. “On-the-fly” composition can thus be used to reduce voice-search memory requirements from 1,950 MB (for uncompressed \( \text{CLG} \)) to 278 MB (for the delayed composition of compressed \( C \) and \( LG \)). In the question-answering application, we obtain a 9.37x reduction from 5,969 MB to 637 MB.

\(^2\)All WFSTs were quantized to 16 bits before gzip compression.

---

**Table 1.** Size of WFSTs for question-answering application (in millions of arcs/states and megabytes).

<table>
<thead>
<tr>
<th>FSM</th>
<th>states / arcs</th>
<th>Standard size</th>
<th>gzip size</th>
<th>Packed size</th>
<th>Ratio (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{CLG} )</td>
<td>69 / 304</td>
<td>5,969</td>
<td>1,972</td>
<td>2,085</td>
<td>34.9</td>
</tr>
<tr>
<td>( LG )</td>
<td>32 / 95</td>
<td>2,042</td>
<td>609</td>
<td>634</td>
<td>31.0</td>
</tr>
<tr>
<td>( G )</td>
<td>7 / 41</td>
<td>778</td>
<td>300</td>
<td>301</td>
<td>38.7</td>
</tr>
<tr>
<td>( L )</td>
<td>1.1 / 1.6</td>
<td>44</td>
<td>8</td>
<td>9</td>
<td>20.4</td>
</tr>
<tr>
<td>( C )</td>
<td>0.003 / 0.8</td>
<td>12.5</td>
<td>3</td>
<td>3</td>
<td>24.3</td>
</tr>
</tbody>
</table>

**Table 2.** Average number of bits per field, transformation function and number of control bits (BC refers to byte code encoding) in question-answering application.

<table>
<thead>
<tr>
<th>FSM</th>
<th>i label</th>
<th>o label</th>
<th>cost</th>
<th>destination</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{CLG} )</td>
<td>6.86 delta(2)</td>
<td>10.08 delta(1)</td>
<td>16</td>
<td>12.10 delta orig(2)</td>
</tr>
<tr>
<td>( LG )</td>
<td>8.76 delta(2)</td>
<td>8.52 delta(2)</td>
<td>16.00</td>
<td>11.92 delta orig(2)</td>
</tr>
<tr>
<td>( G )</td>
<td>15.14 delta(BC)</td>
<td>0 acceptor(0)</td>
<td>16.00</td>
<td>16.14 delta orig(2)</td>
</tr>
<tr>
<td>( L )</td>
<td>8.00 abs(0)</td>
<td>3.46 delta(1)</td>
<td>0</td>
<td>13.55 delta orig(2)</td>
</tr>
<tr>
<td>( C )</td>
<td>9.52 delta(2)</td>
<td>8.00 delta(BC)</td>
<td>0</td>
<td>8.24 abs(1)</td>
</tr>
</tbody>
</table>

**Table 3.** Size of WFSTs for voice-search application (in millions of arcs/states and megabytes).

<table>
<thead>
<tr>
<th>FSM</th>
<th>states / arcs</th>
<th>Standard size</th>
<th>gzip size</th>
<th>Packed size</th>
<th>Ratio (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{CLG} )</td>
<td>26 / 96</td>
<td>1,950</td>
<td>652</td>
<td>657</td>
<td>33.7</td>
</tr>
<tr>
<td>( LG )</td>
<td>11 / 38</td>
<td>795</td>
<td>273</td>
<td>275</td>
<td>34.6</td>
</tr>
<tr>
<td>( G )</td>
<td>1.7 / 18</td>
<td>317</td>
<td>153</td>
<td>122</td>
<td>38.5</td>
</tr>
<tr>
<td>( L )</td>
<td>0.3 / 0.6</td>
<td>14.5</td>
<td>3</td>
<td>3</td>
<td>20.7</td>
</tr>
<tr>
<td>( C )</td>
<td>0.003 / 0.8</td>
<td>12.5</td>
<td>3</td>
<td>3</td>
<td>24.3</td>
</tr>
</tbody>
</table>

**Table 4.** Average number of bits per field, transformation function and number of control bits (BC refers to byte code encoding) in voice-search application.
3.3. Recognition

We performed speech recognition experiments using an independent test set, with both the voice-search compressed and the uncompressed CLG WFSTs. All experiments were ran in an Intel core 2 Linux computer. Accuracy and real time results obtained at various beams are shown in Figure 1. We observe no loss of accuracy due to the compressed format. Furthermore, the compressed format leads to slightly faster recognition. This is a positive but not surprising result. [12] also obtained some speed improvements by using a compact format. In modern processors, the extra computation due to compression is more than compensated by better cache usage. In figure 2 we see that the performance of the “on-the-fly” composition is very similar to that of the original CLG network. The overhead of run time $C \circ LG$ composition is annulled by the speed advantage of the compressed format.

![Figure 1](image1.png)

**Fig. 1.** Accuracy vs. real time factor of compressed and uncompressed CLG WFSTs.

![Figure 2](image2.png)

**Fig. 2.** Accuracy vs. real time factor of uncompressed CLG vs “on-the-fly” compressed $C \circ LG$.

4. CONCLUSIONS AND FUTURE WORK

The presented WFST compression technique has proven very effective at reducing the memory requirements for large ASR tasks. However, there are several aspects that we plan to improve in future work. One is the state renumbering procedure, which may be improved by using methods developed for separable graphs [14]. Weight quantization can also be improved by studying the effect of stronger quantization on certain WFST classes such as language models. And, in general, more specific transformation and packing functions can be added. We believe that the technique is general enough that it will accommodate these and other improvements with no substantial changes.

5. REFERENCES