

Compiling Large-Context Phonetic Decision Trees into Finite-State Transducers

Stanley F. Chen

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

Abstract

Recent work has shown that the use of finite-state transducers (FST's) has many advantages in large vocabulary speech recognition. Most past work has focused on the use of triphone phonetic decision trees. However, numerous applications use decision trees that condition on wider contexts; for example, many systems at IBM use 11-phone phonetic decision trees. Alas, large-context phonetic decision trees cannot be compiled straightforwardly into FST's due to memory constraints. In this work, we discuss memory-efficient techniques for manipulating large-context phonetic decision trees in the FST framework. First, we describe a lazy expansion technique that is applicable when expanding small word graphs. For general applications, we discuss how to construct large-context transducers via a sequence of simple, efficient finite-state operations; we also introduce a memory-efficient implementation of determinization.

1. Introduction

The use of finite-state transducers (FST's) in speech recognition has been shown to yield many benefits. Most notably, the task of decoding can be framed as a search on a finite-state machine (FSM) created by the composition of several finite-state transducers [1]. Specifically, if we take G to be an FSM encoding a grammar or language model, L to be an FST encoding a pronunciation lexicon, and C to be an FST encoding the expansion of context-independent phones to context-dependent units, then the composition $G \circ L \circ C$ yields an FST mapping word sequences to their corresponding sequences of context-dependent units. The resulting FSM, after minimization, can be used directly in a speech recognition decoder; such decoders have been shown to yield excellent performance [2].

In this paper, we focus on the construction of the transducer C for large-context phonetic decision trees. That is, most speech recognition systems use decision trees to map from context-independent phones to context-dependent units. For example, for a triphone decision tree, the context-dependent units corresponding to a phone are determined by using a decision tree that can ask questions about three phones of context, *i.e.*, the current phone and one phone to the left and right. (We shall hereafter refer to context-dependent units as *leaves*, as they correspond to leaves in a decision tree.) Thus, a triphone decision tree represents a mapping from triphones to leaves, which suggests a straightforward representation as an FST containing p^2 states and p^3 arcs, where p is the size of the phone vocabulary. More generally, an n -phone decision tree can be expressed with an FST containing p^{n-1} states and p^n arcs [1].

While several papers have discussed the construction of C corresponding to a triphone decision tree, there has been lim-

ited discussion of the use of wider-context decision trees, which have been shown to yield improved accuracies over triphones. With a phone set of size 50, a straightforward conversion of a quinphone decision tree to an FST would contain $50^5 \approx 3 \times 10^8$ arcs; at 16 bytes per arc, this translates to about 5GB of memory, which is more than what today's prevalent 32-bit computing systems can handle. However, Mohri *et al.* [1] note that it should be possible to shrink such transducers a great deal through minimization, and consequently, it should be possible to construct such transducers using current hardware. In this work, we describe memory-efficient algorithms for computing C for large-context phonetic decision trees.

First, we discuss a technique that we refer to as *virtual* finite-state machines, applicable when composing a phonetic decision tree with a graph of modest size as in lattice rescoring. Similar in spirit to *lazy* finite-state operations [3] and to the decision tree FST outlined by Sproat [4], only the portions of an FSM that are used are expanded in memory. However, this method is not helpful in unconstrained decoding, as much of the transducer will be expanded and little savings will be attained.

For more general applications, we discuss how to compile phonetic decision trees into fully-realized, minimal finite-state transducers. We use similar ideas as described by Sproat and Riley [5] for expressing decision trees as the intersection of many small transducers. However, unlike in that work, we discuss techniques for making this process as memory-efficient as possible. As the memory bottleneck in this process (as in many FSM expansion tasks) is the determinization operation, we also describe a memory-efficient implementation of determinization.

2. Virtual finite-state machines

The basic idea behind *virtual* FSM's is that there exist some FSM's that are extremely large if explicitly expressed using a generic FSM representation, but which can be expressed compactly using special-purpose code. For example, FST's encoding phonetic decision trees have a regular structure: an n -phone decision tree can be expressed with p^{n-1} states, each state encoding the identity of the last $n-1$ phones $p_{-m} \cdots p_{m-1}$ accepted where $m = \frac{n-1}{2}$. For a state corresponding to the history $p_{-m} \cdots p_{m-1}$, the output for the outgoing arc with input p_m is determined by applying the decision tree to the context $p_{-m} \cdots p_{m-1} p_m$. Thus, in theory, we can satisfy the *interface* for a large FSM with a decision-tree module that uses very little memory; however, in practice, some extra glue is required.

Most FSM toolkits express the identity of a state as a 32-bit integer (*e.g.*, [6]); to integrate virtual FSM's in existing toolkits, we need to also satisfy this requirement. However, phonetic decision tree FSM's can contain more than 2^{32} states. To solve

```

class VirtualFsm {
  // map from long state descriptions to short
  map<vector<int>, int> l2s;
  // map from short state descriptions to long
  vector<vector<int> > s2l;
public:
  init() { add start state to l2s, s2l; }
  // given short state s, return outgoing arcs
  get_out_arcs(int s) {
    l = s2l[s];
    compute outgoing arcs of l;
    add newly-reached states to l2s, s2l;
    map arc dst. states to short descriptions via l2s;
    return arcs;
  }
};

```

Figure 1: Pseudocode for virtual FSM class.

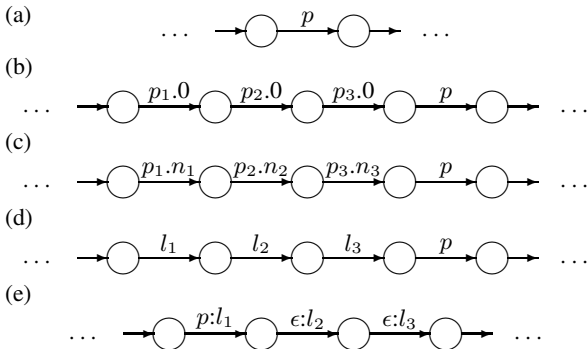


Figure 2: Outline of decision tree expansion.

this dilemma, we note that in most applications, fewer than 2^{32} states in the FSM will be accessed, and we need only number the states that are actually used. We can dynamically construct a table that maps from the full description of a state (*e.g.*, a vector of integers encoding the last $n - 1$ phones) to unique 32-bit integers. We present C++-like pseudocode for a virtual FSM class in Figure 1. The *short* description of a state is its 32-bit integer identity; the *long* description is its full vector representation.

Note that for each state accessed in the virtual FSM, concomitant space in the `l2s` and `s2l` structures must be allocated. As a result, this technique is only applicable when applying the FSM to a graph of modest size, such as in lattice rescoring. For unconstrained word or phone decoding, a large portion of the virtual FSM will be expanded, thus making the memory requirements of this technique excessively high.

3. Decomposing tree expansion

In this section, we describe how to convert a phonetic decision tree to a minimal FST via a sequence of simple finite-state operations. We begin by giving an overview of how the process will work. We start with a *phone loop*, an acceptor with a single state containing a self-loop arc labeled with p for every phone p , *i.e.*, an FSM that accepts any phone sequence. We depict a single arc of this machine in Figure 2(a).

Then, we expand this graph as in Figure 2(b): Before each phone, we insert a marker for each leaf that the phone will expand to; in most speech recognition systems, phones expand to

```

G = graph to be expanded (e.g., a phone loop)
while (G not completely expanded) do
  for each question  $q_i$ 
    apply  $q_i$  to  $G$  for phones to left;
    determinize and minimize  $G$ ;
    reverse  $G$  and determinize and minimize;
  for each question  $q_i$ 
    apply  $q_i$  to  $G$  for phones to right;
    determinize and minimize  $G$ ;
    reverse  $G$  and determinize and minimize;

```

Figure 3: Pseudocode for question-by-question expansion.

```

class Determinize: public VirtualFsm {
  // how many states done expanding
  int done;
public:
  init() { done = 0; }
  determinize() {
    while done < s2l.size() {
      A = get_out_arcs(done);
      write out arcs A of current state;
      done += 1; }
  }
};

```

Figure 4: Pseudocode for determinization.

a sequence of three leaves. For each leaf position p_i that the phone p expands to, we track our current position in the corresponding decision tree. Initially, we begin at node 0, representing the root node of the associated tree.

Next, we apply a sequence of FST's that expand the current graph *one question at a time*. Each such FST updates only the leaf markers containing decision tree nodes that ask that question and leaves all other arcs unchanged. The FST rewrites the arc label of each affected arc so that it points to the appropriate child of the original node. By repeatedly applying FST's for each question, we will update the source FSM from the root nodes of trees (Figure 2(b)) to internal nodes n_i (Figure 2(c)) and eventually to leaves l_i (Figure 2(d)). (We defer the explanation of how to build these question FST's until Section 3.2.)

Finally, we can convert the acceptor in Figure 2(d) to the transducer in Figure 2(e). Unlike the previous steps, we cannot express this as a composition operation, but it still straightforward to do. We delay the conversion from an acceptor to a transducer to the final step because operations on acceptors tend to be more efficient. It is clear that this process will accomplish the desired compilation of a decision-tree to an FST; we now discuss how to execute this procedure efficiently.

3.1. Efficient question expansion

We strive to use as little memory as possible over what is required to store the final FSM, as this is the limiting factor determining which trees we can successfully compile. One natural strategy is to perform determinization and minimization after applying each question FST; then, we need only try to limit the amount of expansion produced when composing with each question FST. (Indeed, the reason we decompose expansion into individual question expansions is to make each composition require as little memory as possible.)

For memory efficiency, operations with nondeterminized

context	states	arcs	time (sec.)
triphone	1971	8334	~30
quinphone	26974	181514	~300
7-phone	266930	1892140	~13000
11-phone	n/a	n/a	n/a

Table 1: Expansion of 1000-leaf phonetic decision trees to FSM’s for different context sizes.

automata are the enemy, because such operations expand many paths which will ultimately be pruned. Unfortunately, decision tree expansion is naturally nondeterministic because the expansion of a phone depends on phones both to the left and right. We address this issue by applying each question in two steps. We first apply a question only to those nodes which ask that question of a phone to the left; the FST implementing this expansion will be deterministic. Then, we reverse the current graph (and determinize and minimize), and then apply the FST expanding the question for all phones to the right; this FST will also be deterministic as we are applying it on the reversed graph. Finally, we reverse the current graph again (and minimize).

Note that we need not reverse the graph for every question; we can apply every question in the forward direction, then reverse the graph, and then apply every question in the reverse direction. We summarize the complete expansion algorithm in Figure 3. In the worst case, we need to execute the outer loop as many times as the maximum depth of the decision tree. With a maximum tree depth of 10 and a question set of size 100, this translates to about $10 \times 100 \times 2 = 2000$ compositions, determinizations, and minimizations. However, in practice, there will be many times when a question q_i is not applicable to any node in G and thus can be skipped. In addition, as G is determinized after every operation and the question FST’s are also deterministic, the result of this composition is usually deterministic as well and thus determinization can often be skipped.

3.2. Constructing question expansion transducers

Consider constructing the FST for expanding a question q_i in the forward direction (*i.e.*, asking about phones to the left) for an n -phone decision tree with $m = \frac{n-1}{2}$, and consider a decision tree node that asks about the identity of a phone the maximum m positions to the left. To perform this expansion correctly, the FST states must encode the answer to q_i for each of the m phones to the left. (Note that we need not encode phone identities, just binary question answers.) An FST containing 2^m states is sufficient for this purpose, one state for each possible set of answers for the last m phones. For each of these states, we will have outgoing arcs for each phone, mapping that phone to itself and advancing to the appropriate next state. For all other tokens, there will be self-loops at each state, either mapping a node to the appropriate child if the question applies, or mapping a token to itself otherwise. Creating FST’s for the reverse direction is similar, except that we need to remember $m + 1$ answers rather than m , as we need to also encode the answer for the current phone (since we place phone identities *after* the corresponding leaf markers in the graph).

4. Memory-efficient determinization

In this section, we describe a memory-efficient implementation of determinization. Here, we discuss only unweighted acceptors, but the algorithm can straightforwardly be extended to weighted transducers as well. Recall that determinization in-

volves finding sets of states in the source machine that can all be reached with a given prefix string; each state in the resulting FSM corresponds to one of these state sets. In fact, the description of a virtual FSM given in Figure 1 partially implements determinization; the *long* descriptions are the state sets corresponding to each result state. To complete the algorithm, we add a little code in a derived class, as shown in Figure 4.

Now, consider the determinization operation performed after each reversal in the algorithm in Figure 3. After reversal, the graph G will be extremely nondeterministic; *e.g.*, the state set corresponding to each result state may contain thousands of states on average, as will be shown later. Thus, the structures $\mathbb{L}\mathbb{S}$ and $\mathbb{S}\mathbb{L}$ can grow to be very large in size.

To reduce the size of $\mathbb{L}\mathbb{S}$, instead of remembering entire state sets (which may be thousands of bytes each), we can just store a short hash signature for each set. There will then be a small chance that the algorithm returns an incorrect answer, in the case that two distinct state sets hash to the same value and the algorithm merges the two states into one. However, we can minimize the chance of error by using a sufficiently large hash signature. For example, with a hash function that is perfectly uncorrelated across state sets, the chance of a hash conflict using a 96-bit signature for a computation that results in a machine with 50M states is less than $50M^2/2^{96} < 10^{-12}$. Additionally, we can run the algorithm multiple times and perform voting to further reduce the chance of error.

However, we cannot use the same technique to compress $\mathbb{S}\mathbb{L}$, as we need to know the full state set identity in order to correctly compute the outgoing arcs of the corresponding result state. Instead, we note that $\mathbb{S}\mathbb{L}$ is accessed in a completely first-in first-out manner; *i.e.*, writes to $\mathbb{S}\mathbb{L}$ occur exactly in order from beginning to end, as are reads. Thus, $\mathbb{S}\mathbb{L}$ can be stored reasonably efficiently on disk. Furthermore, when storing $\mathbb{S}\mathbb{L}$ on disk, it is straightforward to compress this structure using a program such as `gzip`; this makes it possible to take advantage of subsets common to many state sets for compression. In our actual implementation, we store $\mathbb{S}\mathbb{L}$ partially in memory, writing to disk only when memory is depleted. On disk, we spread $\mathbb{S}\mathbb{L}$ over multiple files, so we can delete portions of $\mathbb{S}\mathbb{L}$ when they are no longer needed.

5. Optimizing composition

When composing two machines S_1 and S_2 , the resulting machine may have as many as $|S_1| \times |S_2|$ states, one state (s_1, s_2) for each $s_1 \in S_1$ and $s_2 \in S_2$ [3]. Thus, when we apply a question FSM to the graph G (as in Figure 3) for an 11-phone tree, say, the resulting G may be up to $2^{(11-1)/2} = 32$ times larger, which may be unacceptable for large G . In this section, we describe how to make this composition more memory-efficient.

We note that different parts of G may need to be expanded different amounts. For example, there may be portions of G that do not include any nodes where the given question is asked, so these portions need not be expanded at all. In other words, for states g in these areas, instead of creating states (g, q) for all states q in the question FSM Q , we need only create the state (g, q_0) for a single $q_0 \in Q$. More generally, for each state $g \in G$, we can compute exactly which phone positions in the past will be asked about in future tokens; we can do this via dynamic programming starting from all arcs that will be rewritten by the current question. Then, during composition, whenever we encounter a state (g, q) , we find which phone positions the state g does not care about, and map q to a canonical state q' (*e.g.*, the lowest-numbered such state) that is equivalent to q

given the positions that are irrelevant. Using this technique, we find in practice that for most question FSM's, the graph G is only expanded by a small fraction after composition.

6. Results

Here, we present results of applying the methods described earlier to converting wide-context phonetic decision trees into finite-state transducers. All experiments were conducted using a phone set containing 53 phones including a word boundary phone. We used a 1.26GHz Pentium III CPU with 4GB.

For the first set of experiments, we used decision trees containing about 1000 leaves trained from several hundred hours of telephony data. We trained triphone, quinphone, 7-phone, and 11-phone decision trees. We attempted to convert each of these trees to its corresponding finite-state acceptor (as shown in Figure 2(d)) using the algorithm described in Section 3; the results are given in Table 1. We were unable to complete the 11-phone run due to resource constraints; however, extrapolating from the table indicates that the final FSM for this tree may be exceedingly large. We discuss methods for dealing with this issue later in this section. On the other hand, we were able to construct the 7-phone transducer in a few hours, and as was hoped, the final minimized FSM is indeed many orders of magnitude smaller than the naive FSM containing 53^6 states and 53^7 arcs.

An example of the efficacy of our memory-efficient determinization implementation can be found during our aborted 11-phone run. One such determinization resulted in an FSM containing about 1M states, each of which corresponded, on average, to a set of 6600 states from the original FSM. A typical determinization implementation would require at least $1M \times 6600 \times 4$ bytes ≈ 26 GB of memory; our run used a total of 1.7GB of main memory and 2.7GB of disk space.

In our next set of experiments, we used larger phonetic decision trees trained from ~ 200 hours of Switchboard data: a triphone tree with 3978 leaves and an 11-phone tree with 3416. More precisely, the triphone tree was actually a quinphone tree where the second and fourth positions take on binary values signaling the presence or absence of a word boundary. Expanding a phone loop with the triphone tree resulted in an FSM with 15319 states and 73881 arcs and took about 500 seconds. Again, we were unable to expand the 11-phone tree.

However, we note that for many applications, it is not necessary to build the complete FSM corresponding to a phonetic decision tree because we have some constraints on the type of graphs that it will be applied to. For example, the most common applications involve *word* decoding, where we need only construct the portions of the FSM that correspond to valid word sequences. For this scenario, instead of expanding the tree FSM by starting with a phone loop, we start with a word loop over some vocabulary and expand this loop to the phone level using a pronunciation lexicon. We expanded a Switchboard vocabulary with 40k baseforms with our 11-phone decision tree, yielding an FSM with 185k states and 303k arcs.

While being able to expand a phone loop with an 11-phone tree would allow us to do unconstrained phone decoding with such a model, it is not clear that this is even feasible with current hardware as the size of the final FSM may be unmanageable. Instead, we consider a constrained phone decoding scenario: we construct a phone graph that generates all phone sequences composed entirely of triphone sequences that occur in a given word vocabulary. Thus, decoding with such a graph would never consider an unseen triphone sequence, but this may be adequate for many phone decoding applications. Us-

ing the same Switchboard vocabulary as above, we expanded a triphone graph with the 11-phone tree, yielding an FSM with 1005k states and 2917k arcs in about 17 hours.

7. Discussion

In this work, we describe efficient algorithms for expanding wide-context decision trees into finite-state transducers. For applications that require the online expansion of small graphs as in lattice rescoring, virtual FSM's can be effective. For tasks that require trees to be statically compiled into an FST beforehand, the methods described in Section 3 are applicable. While we only discussed phonetic decision trees here, the ideas in Section 3 can be extended in a straightforward manner to a large class of decision trees; the main requirement is that questions must be expressible via FSM's.

Though the basic ideas behind converting decision trees to FST's have been laid out by Sproat and Riley [5], the algorithms described in that work are not very efficient. They report compiling a phonetic decision tree with 291 leaves into a set of transducers (one for each phone) totaling millions of arcs. On the other hand, we have expanded trees with thousands of leaves using up to 11-phone context into a single transducer usually containing hundreds of thousands of arcs or less.

Zweig *et al.* [7] also discuss methods for expanding graphs using wide-context decision trees. However, they restrict their discussion to expanding word graphs and attempt to build graphs with minimal numbers of arcs rather than states. Using the method described in Section 6 of expanding a word loop rather than a phone loop, we can also achieve the cross-word 11-phone expansion of word graphs reported in that work.

Finally, we note that the finite-state transducers corresponding to Figure 2(e) may be very nondeterministic, as the leaf identities for a phone cannot be determined unambiguously until some number of phones in the future. Consequently, applying these FST's to a graph may be inefficient. In practice, one can determinize the tree FST, or alternatively, one can simply apply the methods from Section 3 directly to the graph in question, rather than expanding a phone loop.

8. References

- [1] M. Mohri, F. Pereira, and M. Riley, "Weighted finite-state transducers in speech recognition," *Computer Speech and Language*, vol. 16, pp. 69–88, 2002.
- [2] S. Kanthak, H. Ney, M. Riley, and M. Mohri, "A comparison of two LVR search optimization techniques," in *Proceedings of ICSLP*, 2002.
- [3] M. Mohri, F. Pereira, and M. Riley, "The design principles of a weighted finite-state transducer library," *Theoretical Computer Science*, vol. 231, no. 1, pp. 17–32, 2000.
- [4] R. Sproat, "*Pmtools*: A pronunciation modeling toolkit," in *Proceedings of the 4th ISCA Tutorial and Research Workshop on Speech Synthesis*, Scotland, 2001.
- [5] R. Sproat and M. Riley, "Compilation of weighted finite-state transducers from decision trees," in *Proceedings of ACL*, Santa Cruz, California, June 1996.
- [6] M. Mohri, F. Pereira, and M. Riley, "General-purpose finite-state machine software tools," <http://www.research.att.com/sw/tools/fsm>, 1998.
- [7] G. Zweig, G. Saon, and F. Yvon, "Arc minimization in finite state decoding graphs with cross-word acoustic context," in *Proceedings of ICSLP*, 2002.