INTERSPEECH 2009 BRIGHTON

Incremental composition of static decoding graphs

Miroslav Novák

T.J. Watson Research Center, IBM

miroslav@us.ibm.com

Abstract

A fast, scalable and memory-efficient method for static decoding graph construction is presented. As an alternative to the traditional transducer-based approach, it is based on incremental composition. Memory efficiency is achieved by combining composition, determinization and minimization into a single step, thus eliminating large intermediate graphs. We have previously reported the use of incremental composition limited to grammars and left cross-word context [1]. Here, this approach is extended to n-gram models with explicit ϵ arcs and right cross-word context.

Index Terms: speech recognition, static graph composition

1. Introduction

Weighted finite state transducers are commonly used in state-of-the-art speech recognition systems. As originally proposed by Mohri *at al* [2], they provide a solid theoretical framework for the operations needed for decoding graph construction. A decoding graph is the result of a composition

$$S = HC \circ L \circ G, \tag{1}$$

where G represents a language model, L represents a pronunciation dictionary and HC translates phone sequences to context dependent HMM states. After minimization, such a graph leads to the most efficient decoder implementations. Two main issues arise though; the complexity of the composition and the size of the final graph. On-the-fly composition has been proposed to deal with graph size at the price of additional computation [3], but the availability of 64-bit CPUs with a significantly larger amounts of memory has made the static graph approach popular again. The static graph method is also popular on low-resource platforms.

The complexity of the composition comes mainly from the need to model co-articulation. A decision tree is typically used to assign a GMM to each HMM state given its surrounding phonetic context. The complexity of graph construction grows significantly with the size of the modeled context, i.e., the number of phones the tree can see on both sides of the modeled phone. To reduce the complexity of graph construction, the context is often limited to triphones, left cross-word or even word internal. Direct application of the composition steps can be very memory intensive and determinization and minimization may be needed after each step. Methods for the efficient construction of deterministic and minimal HC have been proposed [4], [5] and show a significant improvement over the original method by Mohri, but the final composition can still create a graph substantially larger than its minimal form.

Determininistic acyclic finite state automata can be built with high memory efficiency using *incremental composition*[6]. For the composition $((HC \circ L) \circ G)$, we show not only that the incremental approach is applicable even in cases where G is cyclic (e.g., n-gram models) but that use of acyclic minimization is sufficient to produce a minimized graph.

Incremental composition has the following advantages:

- It eliminates the creation of large graphs produced by intermediate composition steps.
- It requires only acyclic minimization, which is significantly faster than general cyclic minimization (word internal and left cross-word context).
- Its time complexity is not dependent on the size of the context modeled within a word.

2. Word-internal context modeling

We slightly modify the definitions of G and L from [2]. Rather than words, arcs of G will represent individual pronunciations of words. L will transduce phonetic sequences to pronunciations, rather than words. In the examples shown it this paper, all word labels actually represent word pronunciations. For simplicity, all presented examples use single pronunciations only.

The basic idea of the incremental approach is to perform the composition locally for each state $q_i \in G$ of the grammar creating graph segments

$$S_i = ((HC \circ L) \circ G_i), \tag{2}$$

where G_i represents all arcs of G, leaving the state q_i. As it is created, each graph segment S_i is immediately combined with the final graph S by inserting each of its states. If S_i is deterministic, e.g. if it results from the composition of deterministic graphs, a minimization step can be applied immediately. Before each new state is inserted, an equivalence test is performed to find out if such a state already exists in the final graph or if a new state needs to be added. The equivalence test uses a memory efficient implementation of a hash table [1]. When all states q_i are processed, the resulting final graph is deterministic and minimal.

Let us illustrate the algorithm on an example. Consider the grammar G in Figure 1 being composed with a dictionary L in Figure 2. The dictionary is presented here as an acceptor rather than its usual transducer form. Instead of input and output labels, it has only one label on each arc from the union of the input and output alphabets. This acceptor can be converted to a transducer as long as it is possible to determine which of the original alphabets each label belongs to. Note that L is acyclic and it can only produce paths with a single output label. Working with acceptors is not as formally elegant as with transducers, but it simplifies the implementation and avoids complications such as non-determinism caused by homonyms. An important consequence of the acceptor approach is that the final graph is not strictly minimal in its transducer form, because it preserves the relative position of the input and output labels. In practice, this property is often desirable. The algorithm may be extended with output label pushing to achieve further minimization, but this work is still in progress.

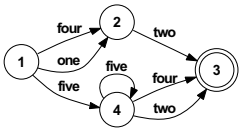

Figure 1: Example of a grammar G

Figure 3 shows results of the composition application to each state of G. After the first step (a), the set of arcs leaving

10.21437/Interspeech.2009-341

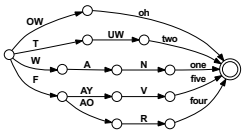

Figure 2: Pronunciation dictionary L

state 1 is replaced by a subset of L representing all pronunciations generated by these arcs. It is important to note that the original states of G are preserved. In the next step (b) state 2 is processed. When state 4 is processed in the final step (c), many of the new states already have their equivalents in the final graph, so only states and arcs shown in dashed red are created.

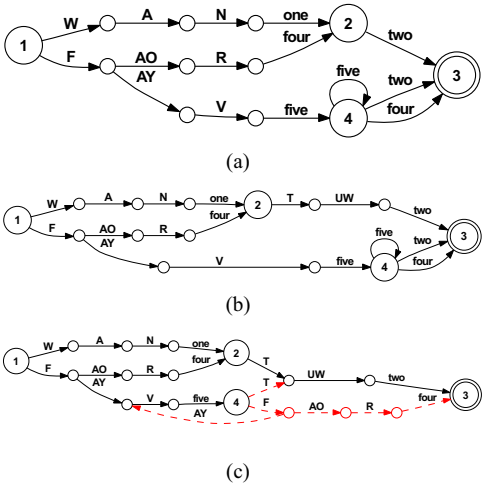

Figure 3: Steps of incremental composition

One of the great advantages of this algorithm is that the cost of minimization is reduced significantly. Global minimization on a cyclic graph is costly, $O(SA\log(S))$, where S is the number of states and A the number of arcs. In our case, we know that the original states of G are preserved and that the composition only adds new states. Under the assumption than G is minimal and that L produces all output labels from the same alphabet as labels of G (i.e. all of the output labels are reachable from the initial state of L), it is not possible that the composition would render any of the original states of G equivalent. This assumption is valid for the word-internal and left cross-word models (in the left-context case, L can be thought as having multiple initial states, one for each context). Furthermore, since graph segments being added always form a tree, only an acyclic minimization, applied locally to the new states added in each step, is required. Fast algorithms using a hash table ($O(S+A)$) are available. To visualize this situation, consider the machine G' in Figure 4 which is derived from G by splitting each state into a pair of states, one for arc beginnings and the other for arc ends. G' is clearly acyclic and composition with an acyclic L will be acyclic as well. After the composition, the final graph can be obtained by simply merging the pairs of states back together. We will use this approach even for the right-context case, though the assumptions above are no longer valid.

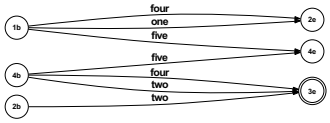

Figure 4: Transformation of G into an acyclic graph

Let us now explain in more detail how the composition and minimization process is applied for each state of $q_i \in G$ through another example. This process consists of the following subtasks: selection of the sub-tree of L matching the active

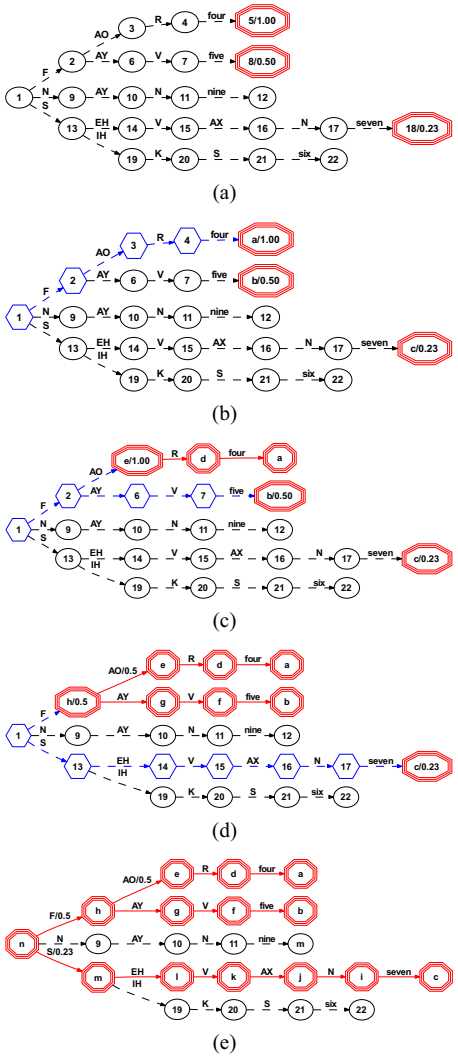

Figure 5: Sub-tree selection

pronunciations given q_i, weight pushing and state equivalence testing. In the presented algorithm, all three subtasks will be preformed simultaneously.

Consider an example of the machine L in Figure 5(a). Assume that the state q_i contains arcs labeled *four, five* and *seven*. States and arcs (each arc is tied to its destination state) of the tree are stored in memory in a topological order (see the numbering of states). The algorithm is based on a post-order traversal. As the states are visited, each state of this tree can be in one of the following three conditions: not visited (circle), visited but marked as waiting to be tested for equivalence (hexagon) or merged with the final graph (triple octagon). At any time, only one state in each level of the tree can be marked as waiting.

Two buffers are needed, both with size equal to the maximum tree depth; the first stores states marked as waiting and the second stores arcs prepared for insertion (in blue).

In the first step (a), the active leaves of the tree are marked to match the pronunciations of q_i and the corresponding costs are assigned to these leaves. The active leaves are then put into a queue sorted by their state number. They are immediately marked as merged, since they are part of G and will appear in the final graph, and the proper destination state index is assigned. As each state is inserted into the final graph, the new state index will be represented by a letter to distinguish it from the tree state index (this change is not shown in Figure (a) in order to show the full tree state numbering). Taking the first state

from the leaf queue the tree is traversed towards its root (b) and all states along the path are marked as waiting and inserted into the buffer of waiting states (indexed by their level in the tree).

When the next leaf is removed from the queue, its parent is compared to the state in the buffer at the same level. If these two states are not equal, then the state in the buffer is inserted into the graph and marked as merged. The costs of its children are pushed on this state, applying the costs on the newly created arcs according to weight pushing in the tropical semiring [7]. The index of the new state (reflecting its position in the final graph) will be used as the target state when its parent arc is added to the graph. The process is repeated with each parent state until either the root or an equal state is reached. If the states are equal (e.g., state 2 in (c)), the traversal towards the root stops. The next leaf is then removed from the queue and the same algorithm is repeated until the queue is empty (d). In the final step, all of the remaining waiting states are merged with the graph (e).

The upper bound on the amount of memory needed to traverse the tree is proportional to the depth of the tree times the maximum number of arcs leaving one state. The memory in which the tree is stored can be read-only and neither the memory required 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 coverage in all situations, the tree can be precompiled and stored in ROM or can be shared among clients through shared memory access.

The described algorithm works well when word-internal context models are used. In this case, the application of the described algorithm is straightforward. The state prefix tree $(HC \circ L)$ to be used in place of L is built simply by enumerating state sequences of all pronunciations in the dictionary and merging the common prefixes.

3. Cross-word context

To extend the algorithm to cross-word context models which affect the composition beyond the word boundaries, a two-pass approach is used. In the first pass, incremental composition is performed multiple times using a context dependent version of $(HC \circ L)$. The minimization step will guarantee that only those states which differ due to the context are added to the graph, when multiple compositions are applied to the same state of G.

As opposed to the word-internal case, not all of the new arcs can be connected to the correct state, because of the cross-word context conditions. These connections are finalized in the second pass.

Let us define:

$\mathcal{P}$	phone vocabulary
$\mathcal{L}$	pronunciation vocabulary
l_L	left context size
l_R	right context size
$C_L = \mathcal{P}^{l_L}$	set of all left contexts
$C_R = \mathcal{P}^{l_R}$	set of all right contexts
$P_{C_R} = P(C_R)$	power set of C_R

Given a fixed pronunciation vocabulary $\mathcal{L}$, we can easily find subsets $C_{L_\mathcal{L}} \subset C_L$ and $C_{R_\mathcal{L}} \subset C_R$ and there is a unique mapping $\lambda : \mathcal{L} \mapsto C_{L_\mathcal{L}}$. Given L in Figure 2, examples of single elements of these sets are: $\{AY, V\} \in C_{L_\mathcal{L}}$ for $l_L = 2$, $\{F, AY\} \in C_{R_\mathcal{L}}$ and $\{\{F, AY\}, \{W, A\}\} \in P_{C_R}$ for $l_R = 2$. In our further examples, we will use $l_L = l_R = 1$, though the algorithm is applicable to larger context sizes at the price of higher computational cost.

We can now construct a left-context-specific state prefix tree for each context $c_L \in C_{L_\mathcal{L}}$. During the tree build, for each pronunciation we need to consider a state sequence for each possible right context $c_R \in C_{R_\mathcal{L}}$. Since some of the state sequences will be equivalent, each leaf of the tree will represent a set of specific right contexts $p_{c_R} \in P_{C_R}$. We can assign an index to each unique p_{c_R} using a hash table. Though cardinality of P_{C_R} is $2^{(l_R)^{|\mathcal{P}|}}$, the actual size of the observed subset for $l_R = 1$ was below 200 even for a large vocabulary. The index of p_{c_R} can thus be encoded into the label of the leaf arc, together with the pronunciation index. The resulting tree is shown in Figure 6. The root state of the tree carries the information about the left context in which the tree was built, and each leaf state provides information about which context it can be used in. An actual GMM index is attached to each input label.

The incremental composition step can now be applied to each state of G as in the word-internal case but with several modifications. Each left-context-specific instance of the tree is applied only to those states which are affected by this context. All leaf states (right-context variants) corresponding to an active pronunciation are selected. This may lead to the creation of unreachable states, but this is a concern for grammar based Gs only. In back-off n-gram models, due to the back-off arcs, all right-context variants will be used. When leaf states are inserted into the graph at the beginning of sub-tree selection, the original destination state indices are used as in the word-internal context case, because their context-dependent instances may not exist yet. The correct destination will be determined in the second pass (with the exception of ϵ arcs). When the root state is reached at the end of sub-tree selection, its index will represent the left-context-dependent state index. This index is inserted into a mapping table $\rho : (q, c_L) \mapsto q_{c_L}$ to be used in the second pass.

A different treatment needs to be applied to ϵ arcs. They are used in n-gram models to represent back-off transitions. If they are considered as part of the pronunciation vocabulary, G is still deterministic. They don't create a context on their own; they only transfer it from one root state to another. The correct destination must already be known during the first pass, otherwise the equivalence test performed for the minimization would not be correct. The states of G are first topologically sorted with respect to ϵ transitions and the composition is applied in reversed topological order, so the final destination of each ϵ arc is always available. In the case of n-gram models, this means that the unigram back-off state is processed first, then the bigram back-off states etc.

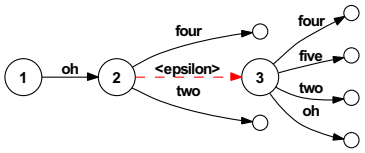

Figure 8: Segment of G

We will now explain in detail how the connections are finalized in the second pass on an example of a grammar segment in Figure 8. The result of the composition of this segment is shown in Figure 7. At the end of the first pass, arcs shown in green do not exist yet. All arcs and states in solid black do exist and are a part of the final graph. States in blue represent the roots of inserted left-context instances of the prefix tree for states 2 and 3. Arcs leaving those states (in dotted black) will not be a part of the final graph, but they will be used to construct new arcs with the correct right-context constraints. The following algorithm is applied to each arc with a pronunciation label. Consider the arc with label l = 'oh' directed to state $q = [2/\text{R}=\{\text{T,F}\}]$. As mentioned earlier, the information about the right context is actually encoded in the label as an index of $p_{c_R} \in P_{C_R}$, but in the figure it is shown on the state for better clarity.

1. Find the corresponding left-context root $q_{c_L} = \rho(\text{q=2}, \lambda(\text{l='oh'}))$, given the left context implied by the label and the destination state, using the map created in

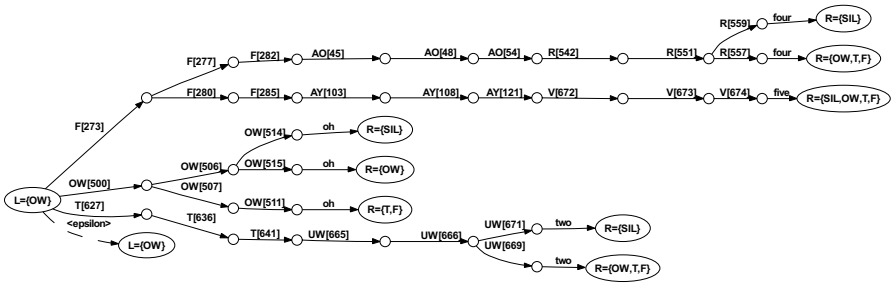

Figure 6: Context dependent state prefix tree

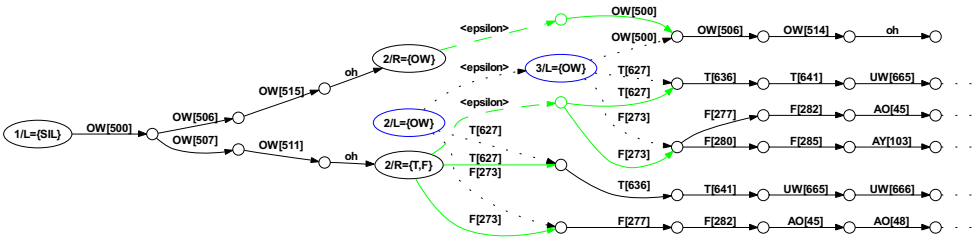

Figure 7: Cross word connections

the first run. It is the blue state $[2/\text{L}=\{\text{OW}\}]$ in the figure.

2. Insert the state q_{c_L} into a stack. If there is an ϵ arc leaving the state, insert its destination state into the stack. Repeat until a state without an ϵ arc is reached.
3. Remove the top state from the stack (blue state $[3/\text{L}=\{\text{OW}\}]$). Form a new state using only arcs which satisfy the right context condition (green arcs $T[627]$ and $F[273]$). Insert the state into the final graph (checking for equivalence).
4. Remove the next state from the stack. Repeat the previous step, using the index of the previously created state as the destination of the ϵ arc of this state (creates arcs $\langle epsilon \rangle$, $T[627]$ and $F[273]$).
5. Repeat until the stack is empty. Change the destination of the arc to the state created in the last step (imagine the state $[2/\text{R}=\{\text{T,F}\}]$ in the figure becoming green).

The algorithm can be sped up significantly by using a hash table which maps a pair (q_{c_L}, p_{c_R}) (i.e. the right-context dependent subset of arcs of the left-context state) to a state in the final graph, if it already exists.

In the next step, all unreachable states are removed. These are some of the left-context tree roots and right-context endings. While finding unreachable states in general requires a forward-backward pass, in this specific case only a forward pass is needed.

In the final step, the graph needs to globally minimized. Due to the right cross-word constraints, conections created during the second pass may create states for which the assumptions presented in section 2 are not valid. But the experiments showed that the graph is close to being minimal and the final minimization step is relatively fast.

4. Results

Table 1 shows the compilation times and final graph sizes for a grammar and various n-gram LMs and acoustic models. All models use one GMM for each state and context of 1 phone across the word boundaries and 2 or 3 phones on each side within words for the LMs and the grammar respectively. The number of pronunciations in the vocabulary is also shown. The resulting graph sizes are shown for both the composed and the globally minimized graphs. Times were measured on an IBM HS21 5 GHz 64 bit blade server with 8 GB of RAM. For comparison, the compilation times using Chen's method [4] are shown on the last line. Correctness of each graph was verified by reproducing the corresponding test results.

task/ descr.	Stock names	English LM	Arabic unvow.	Arabic vowelized
# of GMMs	2500	2196	5000	6000
LM size	8.3k	543k	573k	189k
states/arcs	25k	3.9M	6.2M	2.3M
vocab. size	23.7k	90k	920k	2.6M
composed	735k	30.9M	48M	78.8M
states/arcs	855k	60M	75.4M	119.8M
minimized	725k	25.3M	43.4M	77.8M
states/arcs	845k	44.7M	70.7M	118.5M
memory use	260MB	3.5GB	5.2GB	13GB
composition	17s	314s	27m	1h30m
minimization	4s	201s	270s	244s
baseline	373s	2h21m	9h	15h

Table 1: Composition timing

5. References

[1] M. Novak and V. Bergl, "Memory efficient decoding graph compilation with wide cross-word acoustic context", In *Proceedings of Interspeech 2004*, 281-284, Seul, South Korea, 2004.

[2] M. Mohri, F. Pereira and M. Riley, "Weighted finite-state transducers in speech recognition," *Computer Speech & Language*, vol. 16, no. 1, 69-88, 2002.

[3] D. Caseiro and I. Trancose, "A specialized on-the-fly algorithm for lexicon and language model composition", *IEEE Transactions on Audio, Speech and Language Processing*, 14(4):1281-1291, 2006.

[4] S. Chen, "Compiling large-context phonetic decision trees into finite-state transducers", In Proceedings of Eurospeech 2003,1169-1172, Geneva, Switzerland, 2003.

[5] E. Stoimenov and J. McDonough, "Modeling polyphone context with weighted finite-state transducers" In *Proceedings of ICASSP 2006*, Toulouse, France, 2006.

[6] J. Daciuk, S. Mihov, B. Watson and E. Watson, "Incremental construction of minimal acyclic finite-state automata", Computational Linguistics, 16(1):3-16, 2000.

[7] M. Mohri and M. Riley, "A weight pushing algorithm for large vocab vocabulary speech recognition", In *Proceedings of Eurospeech 2001*, 1603-1606, Aalborg, Denmark, 2001.