Dynamic compilation of weighted context-free grammars

Mehryar Mohri and **Fernando C. N. Pereira**
AT&T Labs – Research
180 Park Avenue
Florham Park, NJ 07932, USA
{mohri,pereira}@research.att.com

Abstract

Weighted context-free grammars are a convenient formalism for representing grammatical constructions and their likelihoods in a variety of language-processing applications. In particular, speech understanding applications require appropriate grammars both to constrain speech recognition and to help extract the meaning of utterances. In many of those applications, the actual *languages* described are regular, but context-free representations are much more concise and easier to create. We describe an efficient algorithm for compiling into weighted finite automata an interesting class of weighted context-free grammars that represent regular languages. The resulting automata can then be combined with other speech recognition components. Our method allows the recognizer to dynamically activate or deactivate grammar rules and substitute a new regular language for some terminal symbols, depending on previously recognized inputs, all without recompilation. We also report experimental results showing the practicality of the approach.

1. Motivation

Context-free grammars (CFGs) are widely used in language processing systems. In many applications, in particular in speech recognition, in addition to recognizing grammatical sequences it is necessary to provide some measure of the probability of those sequences. It is then natural to use *weighted* CFGs, in which each rule is given a weight from an appropriate weight algebra (Salomaa and Soittola, 1978). Weights can encode probabilities, for instance by setting a rule's weight to the negative logarithm of the probability of the rule. Rule probabilities can be estimated in a variety of ways, which we will not discuss further in this paper.

Since speech recognizers cannot be fully certain about the correct transcription of a spoken utterance, they instead generate a range of alternative hypotheses with associated probabilities. An essential function of the grammar is then to rank those hypotheses according to the probability that they would be actually uttered. The grammar is thus used together with other information sources – pronunciation dictionary, phonemic context-dependency model, acoustic model (Bahl et al., 1983; Rabiner and Juang, 1993) – to generate an overall set of transcription hypotheses with corresponding probabilities.

General CFGs are computationally too demanding for real-time speech recognition systems, since the amount of work required to expand a recognition hypothesis in the way just described would in general be unbounded for an unrestricted grammar. Therefore, CFGs used in spoken-dialogue applications often represent regular languages (Church, 1983; Brown and Buntschuh, 1994), either by construction or as a result of a finite-state approximation of a more general CFG (Pereira and Wright, 1997).[1] Assuming that the grammar can be efficiently converted into a finite automaton, appropriate techniques can then be used to combine it with other finite-state recognition models for use in real-time recognition (Mohri et al., 1998b).

There is no general algorithm that would map an arbitrary CFG generating a regular language into a corresponding finite-state automaton (Ullian, 1967). However, we will describe a useful class of grammars that can be so transformed, and a transformation algorithm that avoids some of the potential for combinatorial

[1] Grammars representing regular languages have also been used successfully in other areas of computational linguistics (Karlsson et al., 1995).

explosion in the process.

Spoken dialogue systems require grammars or language models to change as the dialogue proceeds, because previous interactions set the context for interpreting new utterances. For instance, a previous request for a date might activate the date grammar and lexicon and inactivate the location grammar and lexicon in an automated reservations task. Without such *dynamic grammars*, efficiency and accuracy would be compromised because many irrelevant words and constructions would be available when evaluating recognition hypotheses. We consider two dynamic grammar mechanisms: activation and deactivation of grammar rules, and the substitution of a new regular language for a terminal symbol when recognizing the next utterance.

We describe a new algorithm for compiling weighted CFGs, based on representing the grammar as a weighted transducer. This representation provides opportunities for optimization, including optimizations involving weights, which are not possible for general CFGs. The algorithm also supports dynamic grammar changes without recompilation. Furthermore, the algorithm can be executed on demand: states and transitions of the automaton are expanded only as needed for the recognition of the actual input utterances. Moreover, our lazy compilation algorithm is optimal in the sense that the construction requires work linear in the size of the input grammar, which is the best one can expect given that any algorithm needs to inspect the whole input grammar. It is however possible to speed-up grammar compilation further by applying pre-compilation optimizations to the grammar, as we will see later. The class of grammars to which our algorithm applies includes right-linear grammars, left-linear grammars and certain combinations thereof.

The algorithm has been fully implemented and evaluated experimentally, demonstrating its effectiveness.

2. Algorithm

We will start by giving a precise definition of dynamic grammars. We will then explain each stage of grammar compilation. Grammar compilation takes as input a weighted CFG represented as a weighted transducer (Salomaa and Soittola, 1978), which may have been optimized prior to compilation (*preoptimized*). The weighted transducer is analyzed by the compilation algorithm, and the analysis, if successful, outputs a collection of weighted automata that are combined at runtime according to the current dynamic grammar configuration and the strings being recognized. Since not all CFGs can be compiled into weighted automata, the compilation algorithm may reject an input grammar. The class of allowed grammars will be defined later.

2.1. Dynamic grammars

The following notation will be used in the rest of the paper. A weighted CFG $G = (V, P)$ over the alphabet Σ, with real-number weights consists of a finite alphabet V of variables or nonterminals disjoint from Σ, and a finite set $P \subset V \times \mathbb{R} \times (V \cup \Sigma)^*$ of productions or derivation rules (Autebert et al., 1997). Given strings $u, v \in (V \cup \Sigma)^*$, and real numbers c and c', we write $(u, c) \xrightarrow{*} (v, c')$ when there is a derivation from u with weight c to v with weight c'. We denote by $L_G(X)$ the weighted language generated by a nonterminal X:

$$L_G(X) = \{(w, c) \in \Sigma^* \times \mathbb{R} : (X, 0) \xrightarrow{*} (w, c)\}$$

We can now define the two grammar-changing operations that we use.

Dynamic activation or deactivation of rules [2] We augment the grammar with a set of *active nonterminals*, which are those available as start symbols for derivations. More precisely, let $A \subseteq V$ be the set of active nonterminals. The language generated by G is then $L_G = \bigcup_{X \in A} L_G(X)$. Note that inactive nonterminals, and the rules involving them, are available for use in derivations; they are just not available as start symbols. Dynamic rule activation or deactivation is just the dynamic redefinition of the set A in successive uses of the grammar.

Dynamic substitution Let σ be a weighted rational transduction of Σ^* to $\Delta^* \times R$, $\Sigma \subseteq \Delta$, that is a regular weighted substitution (Berstel, 1979). σ is a monoid morphism verifying:

[2] This is the terminology used in this area, though a more appropriate expression would be dynamic activation or deactivation of *nonterminal symbols*.

$$\forall x \in \Sigma, \sigma(x) \subset Reg(\Delta^* \times \mathbb{R})$$

where $Reg(\Delta^* \times \mathbb{R})$ denotes the set of weighted regular languages over the alphabet Δ. Thus σ simply substitutes for each symbol $a \in \Sigma$ a weighted regular expression $\sigma(a)$. A dynamic substitution consists of the application of the substitution σ to Σ, during the process of recognition of a word sequence. Thus, after substitution, the language generated by the new grammar G' is:[3]

$$L_{G'} = \sigma(L_G)$$

Our algorithm allows for both of those dynamic grammar changes without recompiling the grammar.

2.2. Preprocessing

Our compilation algorithm operates on a weighted transducer $\tau(G)$ encoding a factored representation of the weighted CFG G, which is generated from G by a separate preprocessor. This preprocessor is not strictly needed, since we could use a version of the main algorithm that works directly on G. However, preprocessing can improve dramatically the time and space needed for the main compilation step, since the preprocessor uses determinization and minimization algorithms for weighted transducers (Mohri, 1997) to increase the sharing — *factoring* — among grammar rules that start or end the same way.

The preprocessing step builds a weighted transducer in which each path corresponds to a grammar rule. Rule $X\alpha \to Y_1 \ldots Y_n$ has a corresponding path that maps X to the sequence $Y_1 \ldots Y_n$ with weight α. For example, the small CFG in Figure 1 is preprocessed into the compacted transducer shown in Figure 2.

$$\begin{aligned} Z\ .1 &\to XY \\ X\ .2 &\to aY \\ Y\ .3 &\to bX \\ Y\ .4 &\to c \end{aligned} \tag{1}$$

Figure 1: Grammar G_1.

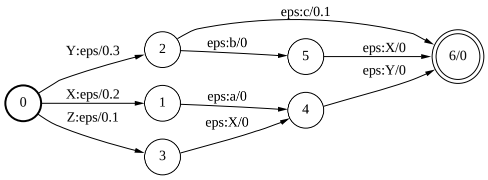

Figure 2: Weighted transducer $\tau(G_1)$.

2.3. Compilation

The compilation of weighted left-linear or right-linear grammars into weighted automata is straightforward (Aho and Ullman, 1973). In the right-linear case, for instance, the states of the automaton are the grammar nonterminals together with a new final state F. There is a transition labeled with $a \in \Sigma$ and weight $\alpha \in R$ from $X \in V$ to $Y \in V$ iff the grammar contains the rule $X\alpha \to aY$. There is a transition from X to F labeled with a and weight α iff $X\alpha \to a$ is a rule of the grammar. The initial states are the states corresponding to the active nonterminals. For example, Figure 3 shows the weighted automaton for grammar G_2 consisting of the last three rules of G_1 with start symbol X.

However, the standard methods for left- and right-linear grammars cannot be used for grammars such as G_1 that generate regular sets but have rules that are neither left- nor right-linear. But we can use the methods for left- and right-linear grammars as subroutines if the grammar can be decomposed into left-linear and right-linear components that do not call each other recursively (Pereira and Wright, 1997). More precisely, define a *dependency graph* D_G for G's nonterminals and examine the set of its strongly-connected components (SCCs).[4] The nodes of D_G are G's nonterminals, and there is a directed edge from X to Y if Y appears in the right-hand side of a rule with left-hand side X, that is, if the definition of X depends on Y. Each SCC S of D_G has a corresponding subgrammar of G consisting of those rules with

[3] σ can be extended as usual to map $\Sigma^* \times \mathbb{R}$ to $Reg(\Delta^* \times \mathbb{R})$.

[4] Recall that the strongly connected components of a directed graph are the equivalence classes of graph nodes under the relation R defined by: $q\ R\ q'$ if q' can be reached from q and q from q'.

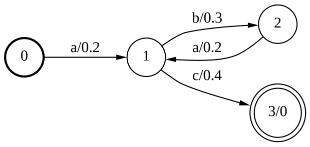

Figure 3: Compilation of G_2.

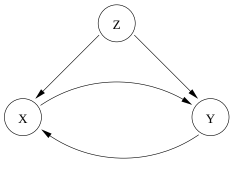

Figure 4: Dependency graph D_{G_1} for grammar G_1.

left-hand nonterminals in S, with nonterminals not in S treated as terminal symbols. If each of these subgrammars is either left-linear or right-linear, we shall see that compilation into a single finite automaton is possible.

The dependency graph D_G can be obtained easily from the transducer $\tau(G)$. For example, Figure 4 shows the dependency graph for our example grammar G_1, with SCCs $\{Z\}$ and $\{X, Y\}$. It is clear that G_1 satisfies our condition, and Figure 5 shows the result of compiling G_1 with $A = \{Z\}$.

The SCCs of D_G can be obtained in time linear in the size of G (Aho et al., 1974). Before starting the compilation, we check that each subgrammar is left-linear or right-linear (as noted above, nonterminals not in the SCC of a subgrammar are treated as terminals). For example, if $\{X_1, X_2\}$ is an SCC, then the subgrammar

$$\begin{aligned} X_1 &\rightarrow aY_1bY_2X_1 \\ X_1 &\rightarrow bY_2aY_1X_2 \\ X_2 &\rightarrow bbY_1abX_1 \end{aligned} \tag{2}$$

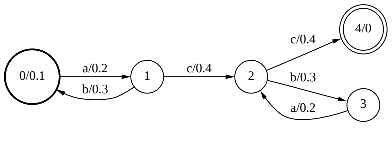

Figure 5: Compilation of G_1 with start symbol Z.

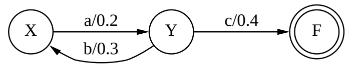

Figure 6: Weighted automaton $K(\{X, Y\})$ corresponding to the strongly connected component $\{X, Y\}$ of G_1.

with $X_1, X_2, Y_1, Y_2 \in V$ and $a, b \in \Sigma$ is right-linear, since expressions such as aY_1bY_2 can be treated as elements of the terminal alphabet of the subgrammar.

When the compilation condition holds, for each SCC S we can build a weighted automaton $K(S)$ representing the language of S's subgrammar using the standard methods. Since some nonterminals of G are treated as terminal symbols within a subgrammar, the transitions of an automaton $K(S)$ may be labeled with nonterminals not in S.[5] The nonterminals not in S can then be *replaced* by their corresponding automata. The replacement operation is *lazy*, that is, the states and transitions of the replacing automata are only expanded when needed for a given input string. Another interesting characteristic of our algorithm is that the weighted automata $K(S)$ can be made smaller by determinization and minimization, leading to improvements in runtime performance.

The automaton $M(X)$ that represents the language generated by nonterminal symbol X can be defined using $K(S)$, where S is the strongly connected component containing X, $X \in S$. For instance, when the subgrammar of S is right-linear, $M(X)$ is the automaton that has the same states, transitions, and final states as $K(S)$ and has the state corresponding to X as initial state. For example, Figure 6 shows $K(\{X, Y\})$ for G_1. $M(X)$ is then obtained from $K(\{X, Y\})$ by taking X as initial state. The left-linear case can be treated in a similar way. Thus, $M(X)$ can always be defined in constant time and space by *editing* the automaton $K(S)$. We use a lazy implementation of this editing operation for the definition

[5] More precisely, they can only be part of other strongly connected components that come before S in a reverse topological sort of the components. This guarantees the convergence of the replacement of the nonterminals by the corresponding automata.

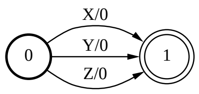

Figure 7: Automaton M_G with activated nonterminals: $A = \{X, Y, Z\}$.

of the automata $M(X)$: the states and transitions of $M(X)$ are determined using $K(S)$ only when necessary for the given input string. This allows us to save both space and time by avoiding a copy of $K(S)$ for each $X \in S$.

Once the automaton representing the language generated by each nonterminal is created, we can define the language generated by G by building an automaton M_G with one initial state and one final state, and transitions labeled with active nonterminals from the initial to the final state. Figure 7 illustrates this in the case where $A = \{X, Y, Z\}$.

Given this construction, the dynamic activation or deactivation of nonterminals can be done by modifying the automaton M_G. This operation does not require any recompilation, since it does not affect the automaton $M(X)$ built for each nonterminal X.

All the steps in building the automata $M(X)$ — construction of D_G, finding the SCCs, and computing for $K(S)$ for each SCC S — require linear time and space with respect to the size of G. In fact, since we first convert G into a compact weighted transducer $\tau(G)$, the total work required is linear in the size of $\tau(G)$.[6] This leads to significant gains as shown by our experiments.

In summary, the compilation algorithm has the following steps:

1. Build the dependency graph D_G of the grammar G.
2. Compute the SCCs of D_G.[7]
3. For each SCC S, construct the automaton $K(S)$. For each $X \in S$, build $M(X)$ from $K(X)$. [8]
4. Create a simple automaton M_G accepting exactly the set of active nonterminals A.
5. The automaton is then expanded on-the-fly for each input string using lazy replacement and editing.

The dynamic substitution of a terminal symbol a by a weighted automaton[9] σ_a is done by *replacing* the symbol a by the automaton σ_a, using the replacement operation discussed earlier. This replacement is also done on demand, with only the necessary part of σ_a being expanded for a given input string. In practice, the automaton σ_a can be large, a list of city or person names for example. Thus a lazy implementation is crucial for dynamic substitutions.

3. Optimizations, Experiments and Results

We have a full implementation of the compilation algorithm presented in the previous section, including the lazy representations that are crucial in reducing the space requirements of speech recognition applications. Our implementation of the compilation algorithm is part of a general set of grammar tools, the GRM Library (Mohri, 1998b), currently used in speech processing projects at AT&T Labs. The GRM Library also includes an efficient compilation tool for weighted context-dependent rewrite rules (Mohri and Sproat, 1996) that is used in text-to-speech projects at Lucent Bell Laboratories. Since the GRM library is compatible with the FSM general-purpose finite-state machine library (Mohri et al., 1998a), we were able to use the tools provided in FSM library to optimize the input weighted transducers $\tau(G)$ and the weighted automata in the compilation output.

We did several experiments that show the efficiency of our compilation method. A key feature of our grammar compilation method is the representation of the grammar by a weighted transducer that can then be preoptimized using weighted transducer determinization and minimization (Mohri, 1997; Mohri, 1998a). To show

[6] Applying the algorithm to a compacted weighted transducer $\tau(G)$ involves various subtleties that we omit for simplicity.

[7] We order the SCCs in reverse topological order, but this is not necessary for the correctness of the algorithm.

[8] For any X, this is a constant time operation. For instance, if $K(S)$ is right-linear, we just need to pick out the state associated to X in $K(X)$.

[9] In fact, our implementation allows more generally dynamic substitutions by weighted transducers.

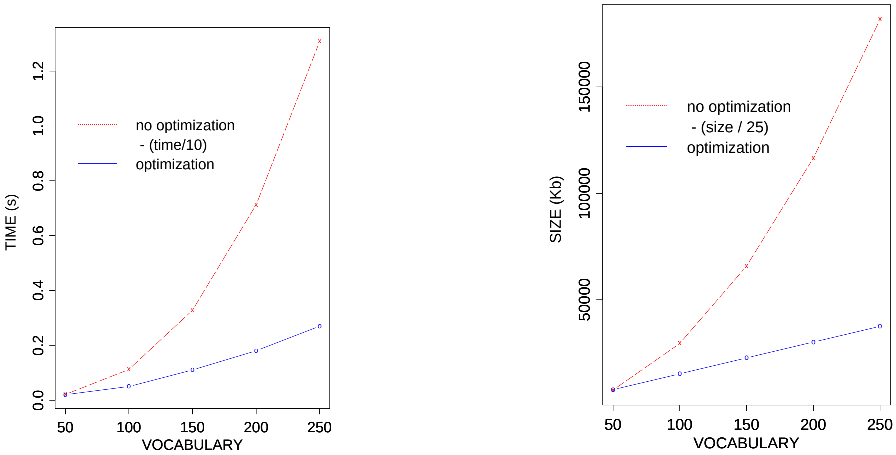

Figure 8: Advantage of transducer representation combined with preoptimization: time and space.

the benefits of this representation, we compared the compilation time and the size of the resulting lazy automata with and without preoptimization. The advantage of preoptimization would be even greater if the compilation output were fully expanded rather than on-demand.

We did experiments with full bigram models with various vocabulary sizes, and with two unweighted grammars derived by feature instantiation from hand-built feature-based grammars (Pereira and Wright, 1997). Figure 8 shows the compilation times of full bigram models with and without preoptimization, demonstrating the importance of the optimization allowed by using a transducer representation of the grammar. For a 250-word vocabulary model, the compilation time is about 50 times faster with the preoptimized representation.[10] Figure 8 also shows the sizes of the resulting lazy automata in the two cases. While in the preoptimized case time and space grow linearly with vocabulary size ($O(\sqrt{|G|})$), they grow quadratically in the unoptimized case ($O(|G|)$).

The bigram examples also show the advantages of lazy replacement and editing over the full expansion used in previous work (Pereira and Wright, 1997). Indeed, the size of the fully-expanded automaton for the preoptimized case grows quadratically with the vocabulary size ($O(|G|)$), while it grows with the cube of the vocabulary size in the unoptimized case ($O(|G|^{3/2})$). For example, compilation is about 700 times faster in the optimized case for a fully expanded automaton even for a 40-word vocabulary model, and the result about 39 times smaller.

Our experiments with a small and a medium-sized CFG obtained from feature-based grammars confirm these observations (Table 1).

Table 1: Feature-based grammars.

$\|G\|$	optim.	time (s)	expanded states	expanded transitions
431	no	.04	9675	11470
431	yes	.02	1535	2002
12657	no	9.76	274614	321615
12657	yes	2.02	112795	144083

If dynamic grammars and lazy expansion are not needed, we can expand the result fully and then apply weighted determinization and minimization algorithms. Additional experiments show that this can yield dramatic reductions in automata size.

4. Conclusion

A new weighted CFG compilation algorithm has been presented. It can be used to compile effi-

[10]For convenience, the compilation time for the unoptimized case in Figure 8 was divided by 10, and the size of the result by 25.

ciently an interesting class of grammars representing weighted regular languages and allows for dynamic modifications that are crucial in many speech recognition applications.

While we focused here on CFGs with real number weights, which are especially relevant in speech recognition, weighted CFGs can be defined more generally over an arbitrary *semiring* (Salomaa and Soittola, 1978). Our compilation algorithm applies to general semirings without change. Both the grammar compilation algorithms (GRM library) and our automata optimization tools (FSM library) work in the most general case.

Acknowledgements

We thank Bruce Buntschuh and Ted Roycraft for their help with defining the dynamic grammar features and for their comments on this work.

References

Alfred V. Aho and Jeffrey D. Ullman. 1973. *The Theory of Parsing, Translation and Compiling.* Prentice-Hall.

Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. 1974. *The design and analysis of computer algorithms.* Addison Wesley: Reading, MA.

Jean-Michel Autebert, Jean Berstel, and Luc Boasson. 1997. Context-free languages and pushdown automata. In Grzegorz Rozenberg and Arto Salomaa, editors, *Handbook of Formal Languages*, volume 1, pages 111–172. Springer.

Lalit R. Bahl, Fred Jelinek, and Robert Mercer. 1983. A maximum likelihood approach to continuous speech recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence (PAMI)*, 5(2):179–190.

Jean Berstel. 1979. *Transductions and Context-Free Languages.* Teubner Studienbucher: Stuttgart.

Michael K. Brown and Bruce M. Buntschuh. 1994. A context-free grammar compiler for speech understanding systems. In *Proceedings of the International Conference on Spoken Language Processing (ICSLP '94)*, pages 21–24, Yokohama, Japan.

Kenneth W. Church. 1983. A finite-state parser for use in speech recognition. In 21^{st} *Meeting of the Association for Computational Linguistics (ACL '83), Proceedings of the Conference.* ACL.

Fred Karlsson, Atro Voutilainen, Juha Heikkila, and Atro Anttila. 1995. *Constraint Grammar, A language-Independent System for Parsing Unrestricted Text.* Mouton de Gruyter.

Mehryar Mohri and Richard Sproat. 1996. An efficient compiler for weighted rewrite rules. In *34th Meeting of the Association for Computational Linguistics (ACL 96), Proceedings of the Conference, Santa Cruz, California.* ACL.

Mehryar Mohri, Fernando C. N. Pereira, and Michael Riley. 1998a. A rational design for a weighted finite-state transducer library. *Lecture Notes in Computer Science*, to appear.

Mehryar Mohri, Michael Riley, Don Hindle, Andrej Ljolje, and Fernando C. N. Pereira. 1998b. Full expansion of context-dependent networks in large vocabulary speech recognition. In *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing (ICASSP '98)*, Seattle, Washington.

Mehryar Mohri. 1997. Finite-state transducers in language and speech processing. *Computational Linguistics*, 23:2.

Mehryar Mohri. 1998a. Minimization algorithms for sequential transducers. *Theoretical Computer Science*, to appear.

Mehryar Mohri. 1998b. Weighted Grammar Tools: the GRM Library. In preparation.

Fernando C. N. Pereira and Rebecca N. Wright. 1997. Finite-state approximation of phrase-structure grammars. In Emmanuel Roche and Yves Schabes, editors, *Finite-State Language Processing*, pages 149–173. MIT Press, Cambridge, Massachusetts.

Lawrence Rabiner and Biing-Hwang Juang. 1993. *Fundamentals of Speech Recognition.* Prentice-Hall, Englewood Cliffs, NJ.

Arto Salomaa and Matti Soittola. 1978. *Automata-Theoretic Aspects of Formal Power Series.* Springer-Verlag: New York.

Joseph S. Ullian. 1967. Partial algorithm problems for context free languages. *Information and Control*, 11:90–101.