A Practical Algorithm for Intersecting Weighted Context-free Grammars with Finite-State Automata

Thomas Hanneforth
Linguistics Dept.
University of Potsdam, Germany
Thomas.Hanneforth@uni-potsdam.de

Abstract

It is well known that context-free parsing can be seen as the intersection of a context-free language with a regular language (or, equivalently, the intersection of a context-free grammar with a finite-state automaton). The present article provides a practical efficient way to compute this intersection by converting the grammar into a special finite-state automaton (the GLR(0)-automaton) which is subsequently intersected with the given finite-state automaton. As a byproduct, we present a generalisation of Tomita's algorithm to recognize several inputs simultaneously.

1 Introduction

At the least since the paper of Billot and Lang (1989) which defined parsing as the intersection of a context-free language (given by a grammar) with a regular language (the "input", which can be seen as a simple finite-state automaton) the importance of the notion of intersection for parsing purposes became apparent. Intersection is first of all defined on the *language level*: Intersect the (possibly infinite) set of strings which the grammar generates with the (possibly infinite) set of strings the finite-state automaton accepts. Since this may not be done effectively due to the infiniteness of the involved sets, the operation has to be lifted to the more compact level of the devices generating the sets.

In principle, there are at least two ways to intersect a context-free grammar G with a finite-state automaton A:

1. Convert G into a suitable pushdown automaton (PDA) M and intersect A with M to get M'. Then extract the result grammar G' from M'.
2. Intersect G and A directly.

With respect to 1., it is well known that the class of pushdown automata is closed under intersection with finite-state automata (henceforth FSA) (cf. Hopcroft and Ullman (1979)). The standard construction assumes a deterministic FSA (see next section) and operates both automata in parallel: Whenever the PDA makes a move on a certain input symbol a, this move is combined with a corresponding move of the FSA on a (see Hopcroft and Ullman (1979) for details of the construction).

Method 1 works most efficiently in case of grammars in *Greibach normal form* (GNF). Remember that a context-free grammar is in GNF if each of its rules conforms to the format $A \rightarrow aB_1 \ldots B_k$ where a is an alphabet symbol, A is a nonterminal (phrase) symbol and $B_1 \ldots B_k$ is a possibly empty sequence of nonterminals. Given a grammar in GNF, it is very easy to construct a pushdown automaton from it: when the PDA has A on its stack and next reads an a, it replaces A by $B_1 \ldots B_k$. Every context-free grammar can be converted into a weakly equivalent grammar in GNF (cf. Hopcroft and Ullman (1979)), but this changes the trees generated by the grammar and may lead to a substiantial increase in grammar size.

Method 2 was presented in Bar-Hillel et al. (1964) and works in the following way: Consider a grammar rule $X_0 \rightarrow X_1 X_2 \ldots X_k$. Then choose an arbitrary state sequence $p_0 \ldots p_k$ from the state set Q of the FSA and create a new grammar rule

$$\begin{aligned} &\langle p_0, X_0, p_k \rangle \rightarrow \\ &\langle p_0, X_1, p_1 \rangle \, \langle p_1, X_2, p_2 \rangle \, \ldots \, \langle p_{k-1}, X_k, p_k \rangle \; . \end{aligned} \tag{1}$$

Proceedings of the 9th International Workshop on Finite State Methods and Natural Language Processing, pages 57–64,
Blois (France), July 12-15, 2011.

Leaving the grammar fixed, the time complexity of this method is in $\mathcal{O}(|Q|^{r+1})$ where r is the length of the longest right-hand side of a grammar rule. The drawback of the method is that it creates a great amount of useless nonterminals and rules. Nederhof and Satta (2008) improved the algorithm by adding simultaneous top-down and bottom-up filtering mechanisms to prevent the creation of useless symbols which of course does not change its worst-case complexity. This upper bound can be reduced to $\mathcal{O}(|Q|^3)$ by binarising the grammar rules (which naturally changes the generated trees). However, it is not clear how the algorithm behaves in practice on grammars with several thousands of rules ubiquitous in natural language processing.

In the present paper, we propose another algorithm based on the creation of a *GLR(0) automaton* which is subsumed by method 1 above. The rest of the paper is organised as follows: Section 2 briefly defines the relevant notions. Section 3 defines GLR(0) automata and presents an algorithm how to intersect them with FSAs. In Section 4 we report some experiments conducted with a big grammar extracted from a treebank.

2 Preliminaries

An alphabet Σ is a finite set of symbols. A string $x = a_1 \dots a_n$ over Σ is a finite concatenation of symbols a_i taken from Σ. The length of a string $x = a_1 \dots a_n$ – symbolically $|x|$ – is n. The empty string is denoted by ε and has length zero. Let Σ^* denote the set of all finite-length strings (including ε) over Σ.

A *finite-state automaton* (FSA) A is a 5-tuple $\langle Q, \Sigma, q_0, \delta, F\rangle$ with Q being a finite set of states; Σ, an *alphabet*, $q_0 \in Q$, the start state; $\delta : Q \times \Sigma \mapsto 2^Q$, the transition function; and $F \subseteq Q$, the set of final states.

Given two states $p, q \in Q$, a *path* from p to q in A – symbolically $p \underset{A}{\rightsquigarrow} q$ – is a sequence $s_0 s_1 \dots s_k$ of states, such that $s_0 = p$, $s_k = q$, and for all $1 \leq i \leq k$: $\exists a \in \Sigma : s_i \in \delta(s_{i-1}, a)$. Given a path $\pi = p \underset{A}{\rightsquigarrow} q$, define $labels(\pi)$ as the concatenation of the symbols labeling the transitions along π. For an empty path $\pi = s_0$, $labels(\pi) = \varepsilon$. The length of path $\pi = s_0 s_1 \dots s_k$ – symbolically $|\pi|$ – is k. We use $p \overset{k}{\underset{A}{\rightsquigarrow}} q$ to denote a path from p to q of length k in A.

An FSA is called *deterministic* if for all symbol-state pairs q, a, $|\delta(q, a)| \leq 1$. For a deterministic FSA, we may modify δ to be a partial function $Q \times \Sigma \mapsto Q$. When appropriate, we use δ as a total function by adding a special element $\perp$ to Q denoting failure. For deterministic FSA, we use sometimes the notation $p \xrightarrow{a} q$ to denote transitions: $p \xrightarrow{a} q$ if $\delta(p, a) = q$.

Define $\delta^* : Q \times \Sigma \mapsto G$ as the reflexive and transitive closure of δ: $\forall q \in Q, \delta^*(q, \varepsilon) = q$ and $\forall q \in Q, a \in \Sigma, w \in \Sigma^* : \delta^*(q, aw) = \delta^*(\delta(q, a), w)$.

Given a (deterministic) FSA $A = \langle Q, \Sigma, q_0, \delta, F\rangle$, the *language* $L(A)$ of A is defined as: $L(A) = \{x \in \Sigma^* \mid \delta^*(q_0, x) \in F\}$.

A *semiring* $\mathcal{K}$ is a 5-tuple $(W, \oplus, \otimes, \overline{0}, \overline{1})$ such that 1. W is a non-empty set, the *carrier set* of the semiring, 2. $(W, \oplus, \overline{0})$ is a commutative monoid, 3. $(W, \otimes, \overline{1})$ is a monoid, 4. $\otimes$ distributes over $\oplus$, and 5. $\overline{0}$ is an annihilator for $\otimes$: $\forall x \in W : x \otimes \overline{0} = \overline{0} \otimes x = \overline{0}$. In the following, we will identify a semiring $\mathcal{K}$ with its carrier set W. Common semirings are the tropical semiring $\mathcal{T} = \langle \mathbb{R}, min, +, \infty, 0\rangle$ and the probabilistic semiring $\mathcal{P} = \langle \mathbb{R}, +, \cdot, 0, 1\rangle$.

A *weighted context-free grammar* (WCFG) G over a semiring $\mathcal{K}$ is a 4-tuple $\langle N, \Sigma, S, P\rangle$: N is a finite set, the non-terminals, Σ is an alphabet, $S \in N$ the start symbol, and P a finite set of pairs $\langle A \rightarrow \beta, c\rangle \in (N \times (\Sigma \cup N)^*) \times \mathcal{K}$, the set of weighted rules. A WCFG without rule weights is called a *context-free grammar* (CFG).

In particular, if $\mathcal{K}$ is the probabilistic semiring and if we define an additional condition on σ:

$$\forall A \in N : \sum_{\langle A \rightarrow \beta, c\rangle \in P} c = 1 \ , \qquad (2)$$

then G is called a *probabilistic context-free grammar* (PCFG) (see also Nederhof and Satta (2008)). Fig. 1 shows a toy PCFG.

1: *S* → *NP VP* / 1.0	**2**: *NP* → *DET N* / 0.6
3: *NP* → *NE* / 0.3	**4**: *NP* → *NP PP* / 0.1
5: *PP* → *P NP* / 1.0	**6**: *VP* → *V* / 0.5
7: *VP* → *V NP* / 0.4	**8**: *VP* → *VP PP* / 0.1

Figure 1: *A toy PCFG with numbered rules. Probabilities are stated after /.*

Let $G = \langle \Sigma, N, S, P \rangle$ be a context-free grammar, and let V be $N \cup \Sigma$. Define a relation $\Rightarrow \subseteq V^* \times V^*$ as follows: $\alpha \Rightarrow \beta$ if $\alpha = xA\gamma, \beta = x\psi\gamma, x \in \Sigma^*, \gamma \in V^*$ and $A \to \psi \in P$.

Let $\overset{*}{\Rightarrow}$ be equal to $\bigcup_{k \geq 0} \overset{k}{\Rightarrow}$. A *leftmost derivation* of a string $w \in \Sigma^*$ is a sequence of elements from $\Rightarrow$ such that $S \Rightarrow X_1 \ldots X_k \Rightarrow \ldots \Rightarrow w$. We abbreviate this to $S \overset{*}{\Rightarrow} w$. The *language* of a CFG G – symbollically $L(G)$ – is defined as $L(G) = \{w \in \Sigma^* \mid S \overset{*}{\Rightarrow} w\}$. Finally, given a CFG G and an FSA A, define the intersection of G and A as follows: $G_\cap = G \cap A$ if $L(G) \cap L(A) = L(G_\cap)$.

The notion of a derivation and the language of a CFG carry over to WCFGs, as well as the notion of intersection. See Nederhof and Satta (2008) for details.

3 The Intersection Algorithm

The main idea to compute the intersection of a weighted context-free grammar G with a finite-state automaton A is stated in Algorithm 1.

Algorithm 1: INTERSECTION OF A WCFG AND AN FSA

Input: WCFG $G = \langle N, \Sigma, S, P \rangle$ over semiring $\mathcal{K}$
Input: FSA $A = \langle Q, \Sigma, q_0, \delta, F \rangle$
Output: WCFG $G_\cap = \langle N_\cap, \Sigma, S_\cap, P_\cap \rangle$ over semiring $\mathcal{K}$ with $N_\cap \subseteq N \times Q$ and $P_\cap \subseteq (N_\cap \times (N_\cap \cup \Sigma)^*) \times \mathcal{K}$
1 Construct GLR(0) automaton M for G
2 Compute $M_\cap$, the intersection of M and A
3 Extract $G_\cap$ from $M_\cap$

In line 1, the WCFG is converted into a GLR(0) automaton. Given a WCFG $G = \langle N, \Sigma, S, P \rangle$ over $\mathcal{K}$, a *GLR(0) automaton* (for *Generalised LR*) $M = \langle Q, \Delta, q_0, F, \delta, \tau \rangle$ over $\mathcal{K}$ is a finite-state automaton with Q, F and q_0 defined as for FSAs; Δ is $N \cup \Sigma$. Since M is required to be deterministic, $\delta : Q \times \Delta \mapsto Q$ is a partial transition function which maps – when defined – a state q and a symbol $a \in \Delta$ to a follow state. $\tau : Q \mapsto 2^P$ is a mapping from states to subsets of grammar rules (indices).[1]

GLR(0) automata are computed from grammars by an algorithm adapted from a standard algorithm (cf. Aho et al. (1986, p. 216ff.)) which will be explained in greater detail in Section 3.1.

Fig. 2 gives a example GLR(0) automaton for the grammar from Fig. 1. A GLR(0) recognizer is controlled by a GLR(0) automaton M. Given some input string w, it creates another GLR(0) automaton M' as a result by repeatedly applying its two main operations:[2]

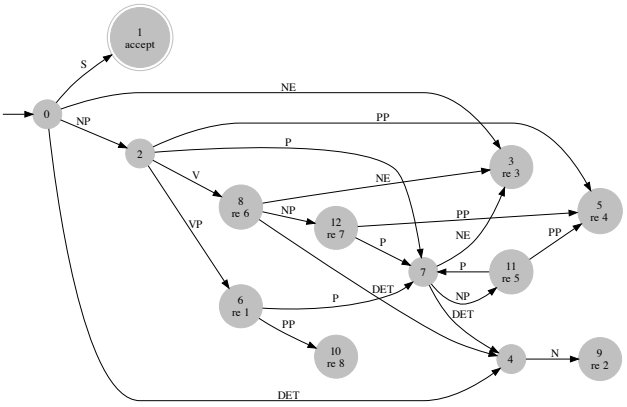


Figure 2: *The GLR(0) automaton for the grammar in Fig. 1. "re* n*"* means: "reduce with rule *n*". A transition from state p to state q corresponds to a (terminal or nonterminal) shift operation.

- **Shift**: When the recognizer reads an input symbol $a \in \Sigma$ in state q, it adds a transition $q \xrightarrow{a} \delta(q, a)$ to M'.

- **Reduce**: When while processing an input string, the parser reaches a state q for which $\tau(q)$ is defined, for each rule $A \to \alpha \in \tau(q)$, find all predecessor states p such that $\mathit{labels}(p \rightsquigarrow q) = \alpha$.[3] Then add a transition $p \xrightarrow{A} \delta(p, A)$ to M' (which may be interpreted as a nonterminal shift). $\delta(p, A)$ is also called a GOTO state.

The GLR(0) recognizer starts in state q_0 and scans the input string from left to right. It applies the operations *shift* and *reduce* until either an accepting state

[1] In the original definition of LR(k) automata, τ is a partial function $Q \mapsto P$. The presence of multiple reduce actions would indicate an ambiguity (a *reduce/reduce conflict*) which entails that the language of the underlying grammar G is not a LR(k)-language. See Aho and Ullman (1972).

[2] Actually, the LR(k) method was invented for parsing deterministic languages like programming languages. In these cases it is not necessary to create an output automaton. Instead, a simple stack is used on which state symbols are pushed and from which they a popped during a reduction.

[3] We momentarily disregard the rule weight here.

$f \in F$ is reached (in which case f is also marked as final in M') or the GLR(0) automaton blocks which causes an error to be signaled.

Continuing with Algorithm 1, line 2 intersects the GLR(0) automaton M with the FSA A, resulting in a GLR(0) automaton $M_\cap$.

Finally, in line 3, the WCFG $G_\cap$ generating $L(G) \cap L(A)$ is extracted from $M_\cap$.
The following subsections explain all three steps in greater detail.

3.1 Efficient Construction of GLR(0) Automata

The construction of a GLR(0) automaton M is based on the computation of the *collection of LR(0) item sets*. Here, the crucial notion is that one of a *dotted rule*. A *dotted rule* is a WCFG rule with a dot somewhere in its right-hand side. This dot indicates which part of the rule was already successfully applied and which part has yet to be matched. An example with respect to the grammar in Fig. 1 is: *NP* $\rightarrow$ *NP* $\bullet$ *PP*. A dotted rule is also called a *LR(0) item*. To compute M, we start with a set containing only the LR(0) item $S' \rightarrow \bullet S$ where S' is a new super start symbol. Then the main operation of the algorithm – *closure* – is applied to it. Basically, given a grammar $G = \langle N, \Sigma, S, P \rangle$, *closure*$(\{A \rightarrow \alpha \bullet B\beta\})$ (with $A, B \in N$ and $\alpha, \beta \in (N \cup \Sigma)^*$) computes on the basis of G the symbols which are expected next given that the automaton's expectation is to read a B.[4]
In our example case,

$$q_0 = \textit{closure}(\{S' \rightarrow \bullet\, S\}) = \left\{ \begin{array}{l} S' \rightarrow \bullet\, S \\ S \rightarrow \bullet\, \textit{NP VP} \\ \textit{NP} \rightarrow \bullet\, \textit{DET N} \\ \textit{NP} \rightarrow \bullet\, \textit{NE} \end{array} \right\} \quad (3)$$

The δ-function of M is computed as follows:

$$\delta(q, B) = \textit{closure}(\{A \rightarrow \alpha B \bullet \beta \mid A \rightarrow \alpha \bullet B\beta \in q\}) \ . \quad (4)$$

For example, $\delta(q_0, \text{NE}) = \textit{closure}(\{\text{NP} \rightarrow \text{NE} \bullet\}) = \{\text{NP} \rightarrow \text{NE} \bullet\}$.

Let the state set Q of M be the set of all LR(0) item sets that can be reached by recursively applying δ and *closure* to q_0 and all item sets originating from it.

If a state q contains an item $A \rightarrow \alpha \bullet$, the rule $A \rightarrow \alpha$ is added to $\tau(q)$. M reaches an accepting state f if the item set contains the LR(0) item $S' \rightarrow S \bullet$.

For grammars not having the LR(0) property (for example, ambiguous grammars), the construction introduces *conflicts*, see Aho and Ullman (1972). Nevertheless, the algorithm leads to deterministic GLR(0) automata for all grammars.

The naive approach representing LR(0) states as sets of dotted rules leads to increased computation times for bigger grammars, for example those extracted from treebanks. For example, the grammar extracted from the TiGer treebank (Brants et al. (2002)) has over 14,300 rules.

A better approach is replacing the dotted rule by a pair consisting of the rule index and the current position of the dot. But even then quite big item sets may result since treebank grammars often have several thousand rules for expanding a single nonterminal symbol.[5] Since the right hand sides of grammar rules expanding a given nonterminal symbol often share common prefixes, left-factoring the grammar (cf. Aho et al. (1986)) is an option when the structure of the parse tree is not of concern. In general, this is not the case in using (weighted) grammars for parsing natural languages.

Instead of altering the grammar, we prefer a more sophisticated representation of the dotted rules. All right hand sides α_i of a set of rules $\{B \rightarrow \alpha_i\}$ expanding B can be combined into a disjunctive regular expression $r = \alpha_1 + \alpha_2 + \ldots + \alpha_k$. r can be converted into a deterministic weighted finite-state machine by standard techniques (cf. Hopcroft and Ullman (1979)). The result is a trie-like left-factored automaton A_B representing the right hand sides of the rules for B. For the final states of A_B, we define a function $\rho : Q \rightarrow I$, where I is the set of rule indices of the WCFG. For a given final state q of A_B (representing a fully found right hand side α of a rule expanding B), $\rho(q) = i$ if $\exists c \in \mathcal{K} : \langle B \rightarrow \alpha, c \rangle$ is the i^{th} rule of the grammar.

Fig. 3 shows the rule FSA A_{VP} for the toy grammar shown in Fig. 1.
An LR(0) item is then represented as a pair $\langle B, q \rangle$

[4] Due to space limitations, we cannot state the definition of the closure algorithm. Please refer to Aho et al. (1986, p. 223) for the details.

[5] For example, the TiGer grammar used in Section 4 contains 2,378 rules for prepositional phrases and 3,475 rules for verbal phrases.

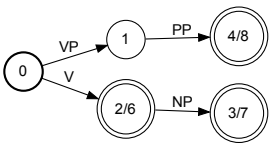


Figure 3: *The rule trie containing the right hand sides of VP of the grammar in Fig. 1. Double circles indicate final states and state the rule index after /.*

where B is a nonterminal and q a state in the FSA A_B associated with B. When during the closure operation a pair $\langle B, q\rangle$ is processed, the transitions $q \xrightarrow{X_i} p$ leaving q in A_B are enumerated and new items $\langle X_i, q_{0_{X_i}}\rangle$ (if $X_i \in N$) are added to the closure items set.

3.2 Intersecting LR(0) Automata with Finite-state Automata

Algorithm 2 computes the intersection of M and A. The algorithm maintains a breadth-first queue L of state pairs $\in Q_M \times Q_A$. In line 4, a pair consisting of the two start states is inserted into L. In the **while**-loop between lines 5 and 31, a state pair $\langle q_M, q_A\rangle$ is removed from L. Then, two types of actions are applied to the current state pair $\langle q_M, q_A\rangle$: Reductions and shifts. Line 9 checks whether M defines reductions for state q_M. If true, the **for**-loop between lines 10 and 23 considers each rule $B \rightarrow \alpha$ with weight c. In line 11, the set of all ancestor nodes for current state $\langle q_M, q_A\rangle$ is computed for which there are paths π of length $|\alpha|$ to $\langle q_M, q_A\rangle$ (simultaneously, we also record the labels of the paths between an ancestor state and $\langle q_M, q_A\rangle$). By definition of the construction of a GLR(0) automaton, *labels*(π) equals α (disregarding the indices).

Then, the **for**-loop between lines 12 and 23 operates over each ancestor state $\langle q, q'\rangle$ of $\langle q_M, q_A\rangle$ and constructs new states and transitions which correspond to the GOTO-actions of the GLR(0) recognizer for nonterminal symbols. Before doing that, a rule $\langle B_{q_A} \rightarrow \alpha', c\rangle$ is added to the reduce actions of state $\langle q_M, q_A\rangle$ in line 13. Note that α' differs from the original α in rule $B \rightarrow \alpha$ in the indices carried by the nonterminals.[6] The subsequent steps are:

- In line 14, a new state $\langle \delta_M(q, B), q_A\rangle$ is created and checked whether it is present in the state set (and inserted if it is not). Here, $\delta_M(q, B)$ denotes the GOTO-state M defines for nonterminal B. Note that the second component q_A of $\langle \delta_M(q, B), q_A\rangle$ is copied from the current state pair $\langle q_M, q_A\rangle$ (the input index does not "move" on after a reduction, so to speak).

[6] We will discuss the necessity of indexed nonterminals B_{q_A} below.

Algorithm 2: INTERSECTION OF A GLR(0) AUTOMATON AND AN FSA

Input: GLR(0) automaton $M = \langle Q_M, \Delta_M, q_{0_M}, F_M, \delta_M, \tau_M\rangle$ over a semiring $\mathcal{K}$
Input: Deterministic FSA $A = \langle Q_A, \Sigma, q_{0_A}, \delta_A, F_A\rangle$
Output: GLR(0) automaton $M' = \langle Q_{M'}, \Delta_{M'}, q_{0_{M'}}, F_{M'}, \delta_{M'}, \tau_{M'}\rangle$ over $\mathcal{K}$

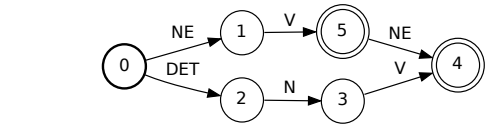
```
1  q_{0_{M'}} ← ⟨q_{0_M}, q_{0_A}⟩
2  Q_{M'} ← {q_{0_{M'}}}
3  F_{M'} ← ∅
4  Enqueue(q_{0_{M'}}, L)
5  while L ≠ ∅ do
6      ⟨q_M, q_A⟩ ← Dequeue(L)
7      if q_M ∈ F_M ∧ q_A ∈ F_A then
8          F_{M'} ← F_{M'} ∪ {⟨q_M, q_A⟩}
9      if τ_M(q_M) ≠ ⊥ then
           // Perform reductions
10         for ⟨B → α, c⟩ ∈ τ_M(q_M) do
11             V_α ← {⟨q, q', labels(π)⟩ | ⟨q, q'⟩ ∈
               Q_{M'} ∧ π = ⟨q, q'⟩ ⇝^{|α|}_{M'} ⟨q_M, q_A⟩}
12             for ⟨q, q', α'⟩ ∈ V_α do
13                 τ_{M'}(⟨q_M, q_A⟩) ←
                   τ_{M'}(⟨q_M, q_A⟩) ∪ {⟨B_{q_A} → α', c⟩}
14                 if ⟨δ_M(q, B), q_A⟩ ∉ Q_{M'} then
15                     Q_{M'} ← Q_{M'} ∪ {⟨δ_M(q, B), q_A⟩}
16                     δ_{M'}(⟨q, q'⟩, B_{q_A}) ←
                       ⟨δ_M(q, B), q_A⟩
17                     Δ_{M'} ← Δ_{M'} ∪ {B_{q_A}}
18                     Enqueue(⟨δ_M(q, B), q_A⟩, L)
19                 else
20                     if δ_{M'}(⟨q, q'⟩, B_{q_A}) = ⊥ then
21                         δ_{M'}(⟨q, q'⟩, B_{q_A}) ←
                           ⟨δ_M(q, B), q_A⟩
22                         if ⟨δ_M(q, B), q_A⟩ ∉ L then
23                             Enqueue(⟨δ_M(q, B), q_A⟩, L)
           // Perform shifts
24         for a ∈ Σ do
25             if δ_M(q_M, a) ≠ ⊥ ∧ δ_A(q_A, a) ≠ ⊥ then
26                 Δ_{M'} ← Δ_{M'} ∪ {a}
27                 p ← ⟨δ_M(q_M, a), δ_A(q_A, a)⟩
28                 δ_{M'}(⟨q_M, q_A⟩, a) ← p
29                 if p ∉ Q_{M'} then
30                     Q_{M'} ← Q_{M'} ∪ {p}
31                     Enqueue(p, L)
32 return M'
```

- In line 20, it is checked whether a transition leaving ancestor state $\langle q, q' \rangle$ with symbol B_{q_A} already exists. Here, there are two subcases to consider:
  1. $\langle \delta_M(q, B), q_A \rangle$ is a new state which entails that the transition $\langle q, q' \rangle \xrightarrow{B_{q_A}} \langle \delta_M(q, B), q_A \rangle$ also does not exist (lines 15–18). This happens when $\langle \delta_M(q, B), q_A \rangle$ is encountered for the first time. $\langle \delta_M(q, B), q_A \rangle$ is added to the state set (line 15) and a new transition leading to it is added to $\delta_{M'}$ (line 16). Finally, B_{q_A} is added to the alphabet of M' (line 17) and the new state $\langle \delta_M(q, B), q_A \rangle$ is inserted into the queue (line 18).
  2. $\langle \delta_M(q, B), q_A \rangle$ already existed, but not $\langle q, q' \rangle \xrightarrow{B_{q_A}} \langle \delta_M(q, B), q_A \rangle$ (lines 21–23). Here, we repeatedly encountered $\langle \delta_M(q, B), q_A \rangle$ during processing. This happens in case of local ambiguities where there exist multiple trees for some subpart of A headed by the same nonterminal. In terms of the graph structure of M', we create a *reentrant* node $\langle \delta_M(q, B), q_A \rangle$ with more than one incoming transition (line 21). Since $\langle \delta_M(q, B), q_A \rangle$ may trigger further reductions (its ancestor set was changed), it is reinserted into the queue (when not already present).

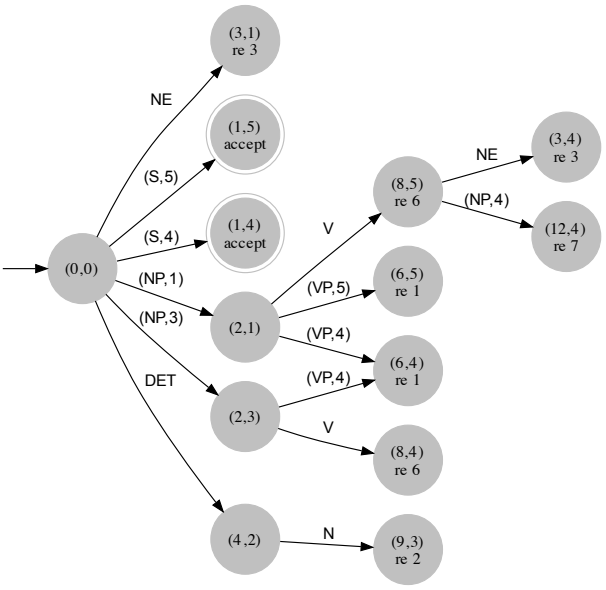
The shift operations performed in the **for**-loop between lines 24 and 31 are similar to the case of the intersection of two FSAs: For every transition leaving q_A labeled a it is tried to find a corresponding transition leaving q_M. If this transition exists, a new state pair $\langle \delta_M(q_M, a), \delta_A(q_A, a) \rangle$ is added to $Q_{M'}$ and L, if not already present (line 30). In addition, a transition $\langle q_M, q_A \rangle \xrightarrow{a} \langle \delta_M(q_A, a), \delta_A(q_A, a) \rangle$ is created (line 28).

Unsurprisingly, Algorithm 2 is simply a generalisation of Tomita's GLR algorithm (cf. Tomita and Ng (1991)) to the recognition of the language of an FSA instead of the recognition of a single sentence (which can be seen as a simple, linear FSA with a single final state). In the general case treated here, A may have several final states and may contain an infinite number of paths. Because of that, the nonterminal symbols of M' are indexed with the current state q_A of A. In that way, M' keeps track of the different reductions made for different paths in A.

Fig. 4(b) shows the result of applying Algorithm 2 to the GLR(0) automaton of Fig. 2 and the FSA of Fig. 4(a).



(a)



(b)

Figure 4: (a) *A deterministic FSA representing three sentences of the PCFG of Fig. 1.* (b) *The result of Algorithm 2 applied to the automata in Fig. 2 and Fig. 4(a). States are labeled with pairs* $\langle q_M, q_A \rangle$.

3.2.1 Complexity.

The outer **while**-loop of Algorithm 2 is bounded by $\mathcal{O}(|Q_M \times Q_A|)$. Since M and A are both deterministic, only a small subset of $Q_M \times Q_A$ will be actually created in the average case. Concerning the *shift*-actions (**for**-loop lines 24–31), each pair is inserted exactly once into the queue. Looking at the *reduce*-actions (lines 9–23), a state pair may be reinserted into the queue in case a new incoming transition is added for $\langle \delta_M(q, B), q_A \rangle$ (line 23). The number of reinsertions is bounded by the number of pos-

S' → S_4 / 1	S' → S_5 / 1
S_4 → *NP*$_1$ *VP*$_4$ / 1	S_4 → *NP*$_3$ *VP*$_4$ / 1
S_5 → *NP*$_1$ *VP*$_5$ / 1	*NP*$_1$ → *NE* / 0.3
NP$_3$ → *DET N* / 0.6	*NP*$_4$ → *NE* / 0.3
VP$_4$ → *V NP*$_4$ / 0.4	*VP*$_4$ → *V* / 0.5
VP$_5$ → *V* / 0.5	

Figure 5: Rule set of the grammar extracted from Fig. 4(b).

sible reductions taking place at $\langle q_M, q_A \rangle$ which is in turn bounded by $|N|$, the number of nonterminals of G. The most expensive step is found in line 11. Let r be the length of the longest right-hand side of a rule in G. The number of state pairs $\langle q, q' \rangle$ created in line 11 and subsequently considered in the **for**-loop lines 12–23 is bounded by $\mathcal{O}(|Q_M \times Q_A|^r)$ (the number of ancestors increases exponentially with respect to the distance r). By applying the memoisation techniques proposed in Kipps (1991), this bound can be strengthend to $\mathcal{O}(|Q_M \times Q_A|^2)$. Putting everything together, the overall complexity of Algorithm 2 is in $\mathcal{O}(|Q_M \times Q_A|^3)$.

3.3 Extracting Grammar Rules

The last step, the extraction of $G_\cap = G \cap A$ is easy and stated in Algorithm 3.

Algorithm 3: EXTRACTING $G_\cap$.

Input: A GLR(0) automaton
$M = \langle Q_M, \Delta_M, q_{0_M}, F_M, \delta_M, \tau_M \rangle$ over $\mathcal{K}$
Output: A WCFG $G_\cap = \langle N, \Delta_M \setminus N, S', P \rangle$ over $\mathcal{K}$

1 $N \leftarrow \{X_i \in \Delta_M \mid i \in \mathbb{N}\} \cup \{S'\}$
2

$$P \leftarrow \{\langle S' \rightarrow X_i, \overline{1} \rangle \mid \exists X_i \in N : \delta_M(q_{0_M}, X_i) \in F_M\} \cup \bigcup_{q \in Q_M \wedge \tau_M(q) \neq \perp} \tau_M(q)$$

Algorithm 3 simply extracts the grammar rules from the states q for which $\tau_M(q)$ is defined. Additionally, a new start symbol S' is introduced and trivially weighted rules $S' \rightarrow X_i$ are added to the rule set such that X_i is labeling a transition from q_{0_M} to a final state. Fig. 5 shows the grammar extracted from the automaton shown in Fig. 4(b).

Automaton	$\mathbf{A_1}$	$\mathbf{A_2}$	$\mathbf{A_3}$
$\lvert\mathbf{Q}\rvert$	8	56	173
$\lvert\delta\rvert$	9	64	259

Table 1: Sizes of the input FSA.

Phase	**Operation**	**Time (ms)**	**Avg. time per sent**	$\lvert\mathbf{Q}\rvert$	$\lvert\delta\rvert$
1	Constr. of M	3,520	-	17,279	1,226,776
2	$M \cap A_1$	47	47	1,841	14,005
2	$M \cap A_2$	3,198	320	18,355	389,140
2	$M \cap A_3$	17,847	178	50,332	1,097,768

Table 2: Results of the experiments with the TiGer grammar.

4 Experiments

We implemented the algorithm from the last section in the C++ programming language within the *fsm2* framework (see Hanneforth (2009)). The grammar used for the experiments has been extracted from the TiGer treebank (cf. Brants et al. (2002)). It contains 14,379 rules[7], and the sizes of the alphabet and the nonterminal sets are 51 and 25, resp. The length of the longest right-hand side of a rule is 17. For the FSA operand of the intersection algorithm, we created three minimal FSA accepting 1, 10 and 100 sentences (tag sequences) randomly taken from the TiGer corpus. The sizes of the automata are summarised in Table 1.

Table 2 shows the results of the experiments which were carried out on a 2.8 GHz CPU. The columns $|Q|$ and $|\delta|$ contain the sizes of the automata resulting from the operation mentioned before.

Since treebank grammars tend to avoid recursive rules and therefore assign flat structures to input strings, they create a lot of readings with spurious ambiguities. Johnson (1998) reports that approximately 9% of the rules of the Penn treebank are never used in a maximum likelihood setting since these rules are subsumed by combinations of other rules with a higher combined probability. We expect even better intersection timings in the face of more linguistically realistic grammars.

[7] Disregarding discontinous constructions like verb–particle rules which are not directly representable in the context-free grammar format.

5 Conclusion

Above, we presented a theoretical and practical algorithm to intersect weighted grammars with FSAs which can be used for parsing or language model training purposes (cf. Nederhof (2005)). No grammar transformation (for example, binarisation) is necessary to achieve optimal cubic complexity. Instead, the binarisation is implicit by using the dotted rule technique. However, the algorithm may suffer from a big grammar dependent constant for artificial grammars (see Johnson (1991) for details). This is due to the implicit subset construction present in the construction of M's δ-function in Eq. (4). An option to investigate for these artificial grammars would be considering the construction of *non-deterministic* GLR(0) automata.

References

Alfred V. Aho and Jeffrey D. Ullman. 1972. *The Theory of Parsing, Translation, and Compiling*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

A. V. Aho, R. Sethi, and J. D. Ullman. 1986. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, Mass.

Y. Bar-Hillel, M. Perles, and E. Shamir. 1964. On Formal Properties of Simple Phrase Structure Grammars. In Y. Bar-Hillel, editor, *Language and Information: Selected Essays on their Theory and Application*, pages 116–150. Addison-Wesley, Reading, Massachusetts.

S. Billot and B. Lang. 1989. The Structure of Shared Forests in Ambiguous Parsing. In *Proceedings of the Twenty-Seventh Annual Meeting of the Association for Computational Linguistics*, pages 143–151, Vancouver (British Columbia). Association for Computational Linguistics (ACL).

Sabine Brants, Stefanie Dipper, Silvia Hansen, Wolfgang Lezius, and George Smith. 2002. The TIGER Treebank. In *Proceedings of the Workshop on Treebanks and Linguistic Theories*.

Thomas Hanneforth. 2009. *fsm2* - A Scripting Language Interpreter for Manipulating Weighted Finite-state Automata. In *Anssi Yli-Jyrä et al. (eds): Finite-State Methods and Natural Language Processing, 8th International Workshop*, pages 13–30, Berlin. Springer.

John E. Hopcroft and Jeffrey D. Ullman. 1979. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley.

M. Johnson. 1991. The Computational Complexity of GLR Parsing. In M. Tomita, editor, *Generalized LR Parsing*, pages 35–42. Kluwer, Boston.

Mark Johnson. 1998. PCFG Models of Linguistic Tree Representations. *Computational Linguistics*, 24(4):613–632.

J. R. Kipps. 1991. GLR Parsing in Time $\mathcal{O}(n^3)$. In M. Tomita, editor, *Generalized LR Parsing*, pages 43–60. Kluwer, Boston.

Mark-Jan Nederhof and Giorgio Satta. 2008. Probabilistic Parsing. In G. Bel-Enguix, M. Dolores Jimenez-Lopez, and C. Martin-Vide, editors, *New Developments in Formal Languages and Applications*, pages 229–258. Springer.

Mark-Jan Nederhof. 2005. A General Technique to Train Language Models on Language Models. *Computational Linguistics*, 31:173–186, June.

M. Tomita and S.-K. Ng. 1991. The Generalized LR Parsing Algorithm. In M. Tomita, editor, *Generalized LR Parsing*, pages 1–16. Kluwer, Boston.