In *Information Processing Letters, 74(5–6):221–227, 2000.*

An Efficient Automata Approach to Some Problems on Context-Free Grammars

Ahmed Bouajjani[1], Javier Esparza[2], Alain Finkel[3], Oded Maler[4], Peter Rossmanith[2], Bernard Willems[5], and Pierre Wolper[5]

[1] LIAFA, 2 Place Jussieu, 75251 Paris Cedex 5, France.
e-mail: Ahmed.Bouajjani@liafa.jussieu.fr
[2] Institut für Informatik, Technische Universität München, Arcisstr. 21, 80290 München, Germany
e-mail: {esparza,rossmani}@in.tum.de
[3] LSV, ENS de Cachan, 94235 Cachan, France
e-mail: finkel@lsv.ens-cachan.fr
[4] Verimag, Centre Equation, 2 avenue de Vignate, 38610 Gieres, France.
e-mail: Oded.Maler@imag.fr
[5] Université de Liège, Institute Montefiore, B28, 4000 Liège, Belgium
e-mail: {willems,pw}@montefiore.ulg.ac.be

Abstract. In Chapter 4 of [2], Book and Otto solve a number of word problems for monadic string-rewriting systems using an elegant automata-based technique. In this note we observe that the technique is also very interesting from a pedagogical point of view, since it provides a uniform solution to several elementary problems on context-free languages.

1 Introduction

In Chapter 4 of their book "String-Rewriting Systems" [2], Book and Otto study so-called *monadic string rewriting systems*. These are sets of rewriting rules of the form $\alpha \to \beta$, where $\alpha, \beta \in \Sigma^*$ for some finite alphabet Σ, satisfying $|\alpha| > |\beta|$ and $|\beta| \leq 1$. The rule $\alpha \to \beta$ allows to rewrite α into β.

Among other results, Book and Otto show that the set of *descendants* of a regular set L of strings – i.e., the set of strings that can be derived from the elements of L through repeated application of the rewriting rules – is also regular[1]; moreover, they provide an elegant algorithm to compute it. The input to the algorithm is a nondeterministic finite automaton (NFA) accepting L, and the output is another NFA accepting the descendants of L.

There is a tight relationship between monadic string rewriting systems and context-free grammars. Given a context-free grammar $G = (V, T, P, S)$ without ϵ-productions and unit productions, the set $R = \{\alpha \to A \mid (A \to \alpha) \in P\}$ is a monadic string rewriting system over the alphabet $V \cup T$. Loosely speaking, R is obtained by "reversing" the productions of G. The set of descendants of a language $L \subseteq (V \cup T)^*$ in R is the set of *predecessors* of L in G, i.e., the set of

[1] This result had also been obtained by Büchi in a different framework. See Section 6.

strings from which some word of L is derivable through repeated application of the productions.

The similarity between monadic string rewriting systems and context-free grammars was already observed by Book and Otto in [2]. In particular, they remark that the algorithm for the computation of descendants could be applied to problems on context-free grammars, but do not elaborate on this point. The purpose of this note is to show that the algorithm indeed leads to elegant and uniform solutions for the membership, emptiness and finiteness problems of context-free grammars, among others.

2 Preliminaries

We use the notations of [11] for finite automata and context-free grammars. Given an NFA $M = (Q, \Sigma, \delta, q_0, F)$, where $\delta \subseteq Q \times \Sigma \times Q$, we define the *transition relation* $\widehat{\delta}: (Q \times \Sigma^*) \to 2^Q$ by:

- $\widehat{\delta}(q, \epsilon) = \{q\}$,
- $\widehat{\delta}(q, a) = \{q' \mid (q, a, q') \in \delta\}$, and
- $\widehat{\delta}(q, wa) = \{\, p \mid p \in \widehat{\delta}(r, a)$ for some state $r \in \widehat{\delta}(q, w) \,\}$

We often denote $q' \in \widehat{\delta}(q, \alpha)$ by $q \xrightarrow{\alpha} q'$.

Given a context-free grammar $G = (V, T, P, S)$, we denote $\Sigma = V \cup T$. We define two relations $\Rightarrow$ and $\overset{*}{\Rightarrow}$ between strings in Σ^*. If $A \to \beta$ is a production of P and α and γ are any strings in Σ^*, then $\alpha A \gamma \Rightarrow \alpha\beta\gamma$. The string $\alpha A \gamma$ is an *immediate predecessor* of $\alpha\beta\gamma$. The relation $\overset{*}{\Rightarrow}$ is the reflexive and transitive closure of $\Rightarrow$. If $\alpha \overset{*}{\Rightarrow} \beta$, then α is a *predecessor* of β. Given $L \subseteq \Sigma^*$, we define

$$pre(L) = \{\, \alpha \in \Sigma^* \mid \exists \beta \in L \text{ with } \alpha \Rightarrow \beta \,\}$$

$pre^i(L)$ is inductively defined by $pre^0(L) = L$ and $pre^{i+1}(L) = pre(pre^i(L))$. Finally, we define $pre^*(L) = \bigcup_{i \geq 0} pre^i(L)$, or, equivalently,

$$pre^*(L) = \{\, \alpha \in \Sigma^* \mid \exists \beta \in L \text{ with } \alpha \overset{*}{\Rightarrow} \beta \,\}$$

3 Computation of *pre**

Let $G = (V, T, P, S)$ be a context-free grammar, and let M be an NFA recognizing a regular set $L(M) \subseteq \Sigma^*$. We wish to construct another NFA M_{pre^*} recognizing $pre^*(L(M))$. Book and Otto's idea (translated into context-free grammars) is to exhaustively perform the following operation, starting with M as current NFA: If $A \to \alpha$ is a production, and in the current NFA we have $q \xrightarrow{\alpha} q'$, then we add a new transition $q \xrightarrow{A} q'$. The algorithm terminates, because the number of states of the NFA remains constant, and there is an upper bound to the number of transitions of an NFA with a fixed number of states and a fixed alphabet. Observe that the new NFA M_{pre^*} has the same states, initial state and final states as M; it differs from M only in its transition relation, which we denote by δ_{pre^*}.

Algorithm 1

Input: $G = (V, T, P, S)$, $M = (Q, \Sigma, \delta, q_0, F)$
Output: δ_{pre^*}

$rel \leftarrow \delta$;
repeat
  for $q, q' \in Q$, $A \to \beta \in P$ **do**
    if $q' \in \widehat{rel}(q, \beta)$ **then** $rel \leftarrow rel \cup \{(q, A, q')\}$ **fi**
  od
until rel does not change any more;
return rel

We apply the algorithm to an example. Consider the context-free grammar $S \to AS \mid SA \mid a$, $A \to b$ and the NFA of Figure 1 having only the transitions drawn with thicker lines. Assume that for each pair of states (q, q') the **for** loop examines all productions of the grammar in the order above. Then the transitions labeled by 1 in Figure 1 are added in the first iteration of the **repeat-until** loop. The second iteration adds the transitions $q_1 \xrightarrow{S} q_1$, derived from $q_1 \xrightarrow{S} q_2 \xrightarrow{A} q_1$, and $q_2 \xrightarrow{S} q_2$, derived from $q_2 \xrightarrow{A} q_1 \xrightarrow{S} q_2$. They are labeled by 2 in Figure 1. The third iteration adds $q_2 \xrightarrow{S} q_1$, derived from $q_2 \xrightarrow{A} q_1 \xrightarrow{S} q_1$ and labeled by 3 in the figure. Nothing is added in the fourth iteration, and the algorithm terminates.

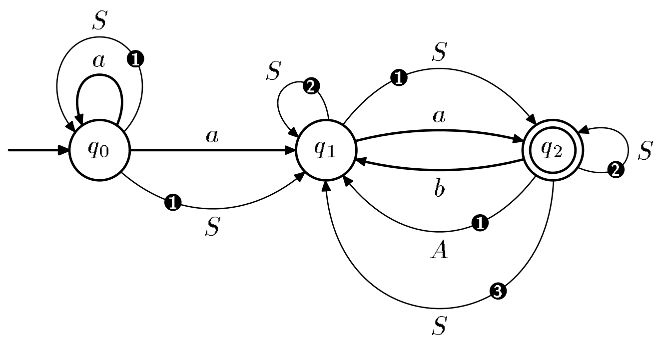

Fig. 1. Illustration of Algorithm 1

The correctness of Algorithm 1 follows immediately from the following two lemmata:

Lemma 1. $pre^*(L(M)) \subseteq L(M_{pre^*})$.

Proof. Let M_i be the NFA computed by the algorithm after i executions of the **repeat-until** loop ($M_0 = M$), and let $\xrightarrow[i]{}$ be the transition relation of M_i. Since $L(M_i) \subseteq L(M_{pre^*})$, it suffices to prove $pre^i(L(M)) \subseteq L(M_i)$ for every $i \geq 0$.

We proceed by induction on i. The case $i = 0$ is trivial because $L(M) \subseteq L(M_0)$ and $pre^0(L(M)) = L(M)$. For the step from i to $i+1$, let α be an arbitrary word of $pre^{i+1}(L(M))$. By the definition of pre, there exist words α_1, α_2 and a production $A \to \beta$ such that $\alpha = \alpha_1 A \alpha_2$ and $\alpha_1 \beta \alpha_2 \in pre^i(L(M))$. By induction hypothesis, $\alpha_1 \beta \alpha_2 \in L(M_i)$. Therefore, there exist states q, q' such that

$$q_0 \xrightarrow[i]{\alpha_1} q \xrightarrow[i]{\beta} q' \xrightarrow[i]{\alpha_2} q_f$$

for some final state q_f. So we have

$$q_0 \xrightarrow[i+1]{\alpha_1} q \xrightarrow[i+1]{A} q' \xrightarrow[i+1]{\alpha_2} q_f$$

which implies $\alpha = \alpha_1 A \alpha_2 \in L(M_{i+1})$. □

Lemma 2. *$L(M_{pre^*}) \subseteq pre^*(L(M))$.*

Proof. For all $j \geq 0$, let N_j be the NFA obtained after the algorithm has added j transitions to the input automaton M, and let $\xrightarrow[j]{}$ denote the transition relation of N_j. Since $L(M_{pre^*})$ is the union of all the sets $L(N_j)$, it suffices to prove $L(N_j) \subseteq pre^*(L(M))$ for every $j \geq 0$.

We proceed by induction on j. The case $j = 0$ is trivial because $N_0 = M$. For the step from j to $j+1$, assume that N_{j+1} is obtained from N_j through the addition of a new transition $q_1 \xrightarrow{A} q_2$. Let α be an arbitrary word of $L(N_{j+1})$. If α is accepted by N_j, then, by the induction hypothesis, $\alpha \in pre^*(L(M))$. If α is not accepted by N_j, then we have $\alpha = \alpha_1 A \alpha_2 A \ldots A \alpha_n$ and

$$q_0 \xrightarrow[j]{\alpha_1} q_1 \xrightarrow[j+1]{A} q_2 \xrightarrow[j]{\alpha_2} q_1 \xrightarrow[j+1]{A} q_2 \ldots q_1 \xrightarrow[j+1]{A} q_2 \xrightarrow[j]{\alpha_n} q_f$$

for some final state q_f. Since there exists a production $A \to \beta$ such that $q_1 \xrightarrow[j]{\beta} q_2$, we have

$$q_0 \xrightarrow[j]{\alpha_1} q_1 \xrightarrow[j]{\beta} q_2 \xrightarrow[j]{\alpha_2} q_1 \xrightarrow[j]{\beta} q_2 \ldots q_1 \xrightarrow[j]{\beta} q_2 \xrightarrow[j]{\alpha_n} q_f$$

and therefore N_j accepts $\alpha' = \alpha_1 \beta \alpha_2 \beta \ldots \beta \alpha_n$. By the induction hypothesis, $\alpha' \in pre^*(L(M))$. Since $\alpha \stackrel{*}{\Rightarrow} \alpha'$, we have $\alpha \in pre^*(L(M))$. □

The running time of Algorithm 1 is easy to estimate. Let p be the number of productions of G, let l be the maximal length of the right hand side of a production, and let s be the number of states of M. Since the transitions added by the algorithm to δ_{pre^*} are labelled by variables appearing on the left hand side of a production, δ_{pre^*} contains at most $O(p \cdot s^2)$ elements. So the **repeat-until** loop is executed $O(p \cdot s^2)$ times. The **for** loop is executed $\Theta(p \cdot s^2)$ times. Checking whether $q' \in \widehat{rel}(q, \beta)$ holds can be done by simulating the NFA (Q, Σ, rel, q, F) on input β, which requires $O(l \cdot s^2)$ time (see [1], pp. 327–329). Adding an element to rel takes $O(1)$ time (assume for instance that rel is stored as a bit matrix). So the running time is $O(l \cdot p^2 \cdot s^6)$.

Algorithm 2

Input: $G = (V, T, P, S)$, $M = (Q, T, \delta, q_0, F)$
Output: δ_{pre^*} as defined above

```
rel ← ∅; trans ← δ;
for every A → ε ∈ P, q ∈ Q
      add (q, A, q) to trans;
for every A → a ∈ P, q, q' ∈ Q
      if (q, a, q') ∈ δ then add (q, A, q') to trans;
for every q, q' ∈ Q, A ∈ V
      direct(q, A, q') ← ∅; impl(q, A, q') ← ∅;
for every A → B ∈ P, q, q' ∈ Q
      add (q, A, q') to direct(q, B, q')
for every A → BC ∈ P, q, q', q'' ∈ Q
      add (q', C, q'') → (q, A, q'') to impl(q, B, q');
      add (q', B, q'') → (q, A, q'') to impl(q, C, q')
while trans ≠ ∅
      pop t from trans; add t to rel;
      append direct(t) to trans;
      while impl(t) ≠ ∅
            pop t' → t'' from impl(t);
            if t' ∈ rel then add t'' to trans
            else add t'' to direct(t')
return rel
```

4 Improving the complexity

Algorithm 1 is very simple, but very inefficient. In this section we present a new algorithm, Algorithm 2, with a running time of $O(p \cdot s^3)$. It works for grammars whose productions are of the form $A \rightarrow BC$, $A \rightarrow B$, $A \rightarrow a$, or $A \rightarrow \epsilon$, i.e., grammars in Chomsky normal form extended with unit productions and ϵ-productions. Observe that any grammar can be transformed into an equivalent grammar of this form. The number of productions of the new grammar is linear in the size of the old grammar, and the transformation can be performed in linear time[2].

We first observe that productions of the form $A \rightarrow a$ or $A \rightarrow \epsilon$ can only contribute new transitions to the input NFA during the first iteration of the **repeat-until** loop. In Algorithm 2 they are processed in an initialisation phase. It remains to deal properly with productions of the form $A \rightarrow B$ and $A \rightarrow BC$.

The main idea is to avoid recomputing information. If, say, we have $A \rightarrow BC$ and we find out during the execution of the algorithm that (q, B, q') belongs to δ_{pre^*}, then we store the following information: If (q', C, q'') belongs to δ_{pre^*}, then

[2] For these grammars the check $q' \in \widehat{rel}(q, \beta)$ is easier, since β has length at most 2, and so Algorithm 1 requires $O(p^2 \cdot s^6)$ time.

so does (q, A, q''). This implication can be used at a later point if it turns out that (q', C, q'') indeed belongs to δ_{pre^*}.

A transition t that is known to belong to δ_{pre^*} is first stored in a FIFO-list *trans*. When, at a later point, t is taken from *trans*, the information that can be extracted from the fact that t belongs to δ_{pre^*} is computed, and then t is added to the final result *rel*. A transition may be added several times to *trans*, but this does no harm.

For each triple $t = (q, A, q')$ two FIFO-lists are used. The first one, called *direct*(t) contains transitions that are known to belong to δ_{pre^*} in case t belongs to δ_{pre^*}. The second one, called *impl*(t), contains *implications* of the form $t_1 \to t_2$ that are known to hold in case t belongs to δ_{pre^*}. An implication $t_1 \to t_2$ is read 'if t_1 belongs to δ_{pre^*} then so does t_2'.

The correctness of the algorithm follows from the following facts, of which we sketch the proof:

(1) After termination $rel \subseteq \delta_{pre^*}$ holds.
This follows from the fact that the conjunction of the following statements is an invariant:
- The transitions of *trans* and *rel* belong to δ_{pre^*}.
- If $t \in direct(t')$ and $t' \in \delta_{pre^*}$, then $t \in \delta_{pre^*}$.
- If $t' \to t'' \in impl(t)$ and $t, t' \in \delta_{pre^*}$, then $t'' \in \delta_{pre^*}$.

Let us show that this conjunction, call it I, is invariant under the execution of line 19, the proofs for the other lines being similar or simpler. It suffices to show that if $t' \in \delta_{pre^*}$, then $t'' \in \delta_{pre^*}$. By I and line 14, $t \in \delta_{pre^*}$. By I and line 17, if $t' \in \delta_{pre^*}$ then $t'' \in \delta_{pre^*}$.

(2) After termination $\delta_{pre^*} \subseteq rel$ holds.
We recall that δ_{pre^*} is defined as the smallest relation containing δ that is closed under the rule for addition of new transitions. So $\delta_{pre^*} \subseteq rel$ follows from the following facts:
- If $t \in \delta$, then t is eventually added to *rel*.
See lines 1, 14.
- If $A \to \epsilon \in P$ then (q, A, q) is eventually added to *rel* for every state q.
See lines 3, 14.
- If $A \to a \in P$ and $(q, a, q') \in \delta$, then (q, A, q') is eventually added to *rel*.
See lines 5, 14.
- If $A \to B \in P$ and $(q, B, q') \in rel$, then (q, A, q') is eventually added to *rel*.
If $(q, B, q') \in rel$, then (q, B, q') has been popped from *trans* at some point (line 14). Since $(q, A, q') \in direct(q, B, q')$ (line 9), (q, A, q') is eventually added to *trans* (line 16), and to *rel* (line 14).
- If $A \to BC \in P$ and (q, B, q') and (q', C, q'') belong to *rel*, then (q, A, q'') is eventually added to *rel*.
Assume (q, B, q') is added to *rel* after (q', C, q'') (the other case is analogous). (q, B, q') is popped from *trans* and added to *rel* at some point (line 14). Since $(q', C, q'') \to (q, A, q'') \in impl(q, B, q')$ (line 11), the implication $(q', C, q'') \to (q, A, q'')$ is popped as well (line 17); since at this

point (q', C, q'') has already been added to *rel*, (q, A, q'') is eventually added to *trans* (line 18), and later to *rel* (line 14).

(3) The algorithm terminates.

We make the following observations:

- During the execution of the outer **while**-loop the *impl* lists only *lose* elements (line 17).
- Transitions flow from the *impl* lists to *trans* either directly (line 18) or via the *direct* lists (lines 19 and 15). No other transitions are added to *trans*, or to the *direct* lists.

This implies that only finitely many transitions can be added to *trans*, and so the outer **while**-loop can only be executed finitely many times.

In order to determine the time and space complexity, let p and s be the number of productions of G and states of M, respectively. We assume that each variable and terminal of G appears in at least one production (otherwise they can be removed in linear time), and so G has size $O(p)$.

The add, pop, and append operations can be performed in constant time. All the steps before the outer **while**-loop can be executed in time $O(p \cdot s^3)$. In order to estimate the running time of the outer **while**-loop, we add one further observation to those of part (3) above:

- Right before the execution of the outer **while**-loop the *trans*, *direct*, and *impl* lists contain together $O(p \cdot s^3)$ elements.

It follows from (3) and this new observation that at most $O(p \cdot s^3)$ elements flow through *trans* during the execution of the algorithm[3], which implies that lines 14 and 15 are executed $O(p \cdot s^3)$ times. Since the *impl* lists only lose elements, lines 17, 18, 19 are also executed $O(p \cdot s^3)$ times. So each line in the body of the outer **while**-loop is executed $O(p \cdot s^3)$ times. Since each line takes constant time, the total running time is $O(p \cdot s^3)$.

We finish the section with a small remark. Algorithm 2 remains correct — and has the same asymptotic time complexity—after removing anyone (but only one!) of the lines 11, 12, or 19. The proof is left to the interested reader.

5 Applications

We show that several standard problems on context-free grammars, for which textbooks often give independent algorithms, can be solved using Algorithm 2.

Let $G = (V, T, P, S)$ be a context-free grammar of size g. In order to avoid redundant symbols in G it is convenient to compute the set of *useless* variables ([11], p. 88). Recall that $X \in V$ is *useful* if there is a derivation $S \stackrel{*}{\Rightarrow} \alpha X \beta \stackrel{*}{\Rightarrow} w$ for some α, β and w, where w is in T^*. Otherwise it is *useless*. To decide if X is useless, observe that X is useful if and only if $S \in pre^*(T^* X T^*)$ and $X \in$

[3] An element may flow through *trans* more than once. We count them as different elements.

$pre^*(T^*)$. Compute the automata accepting $pre^*(T^*XT^*)$ and $pre^*(T^*)$ using Algorithm 2, and check if they accept S and X, respectively. Since the automata for T^*XT^* and T^* have a constant number of states, it can be determined in linear time if X is useful.

Nullable variables have to be identified when eliminating ϵ-productions ([11], p. 90). A variable X is *nullable* if $X \stackrel{*}{\Rightarrow} \epsilon$. To decide the nullability of a variable observe that X is nullable if and only if $X \in pre^*(\{\epsilon\})$, which leads to a linear algorithm.

Consider now the membership problem: given a word $w \in T^*$ of length n, is w generated by G? To solve it, compute the automaton accepting $pre^*(\{w\})$, and check if it accepts S. Since there is an automaton with $n+1$ states recognizing $\{w\}$, the complexity of the algorithm is $O(n^3)$ (for a fixed grammar). This is also the complexity of the CYK-algorithm usually taught to undergraduates [11, 12].

To decide if $L(G)$ is contained in a given regular language L, observe that $L(G) \subseteq L$ is equivalent to $L(G) \cap \overline{L} = \emptyset$, which is equivalent to $S \notin pre^*(\overline{L})$. If L is presented as a deterministic finite automaton with s states, $L(G) \subseteq L$ can be decided by computing a deterministic automaton for $\overline{L}$ in $O(s)$ time, and checking whether $S \notin pre^*(\overline{L})$ in $O(p \cdot s^3)$ time.

Similarly, to decide if $L(G)$ and L are disjoint, check whether $S \notin pre^*(L)$ in $O(p \cdot s^3)$ time.

To decide if $L(G)$ is empty, check whether $L(G)$ is contained in the empty language, which is regular. The automaton for $\overline{L}$ has one state, and so the algorithm is linear.

To decide if $L(G)$ is infinite, assume that G has no useless symbols (otherwise apply the algorithm above), and use the following characterization (see for instance [11], Theorem 6.6): $L(G)$ is infinite if and only if there exists a variable X and strings $\alpha, \beta \in \Sigma^*$ with $\alpha\beta \neq \epsilon$ such that $X \stackrel{*}{\Rightarrow} \alpha X \beta$. This is the case if and only if $X \in pre^*(\Sigma^+ X \Sigma^* \cup \Sigma^* X \Sigma^+)$. This condition can be checked in linear time for each variable X, and so infiniteness can be checked in quadratic time.

6 History

The history of this paper may be interesting to the reader. The regularity of $pre^*(L)$ for a regular language L seems to have been first observed by Büchi in his work on regular canonical systems (see [5] and Chapter 5 of [6]), and has been rediscovered many times in sligtly different contexts, for instance by Caucal in [7] and by Book and Otto in [2].

In 1996, Bouajjani and Maler addressed the model checking problem for pushdown automata [3] and provided a solution based on the regularity of $pre^*(L)$, which they had learnt about in Book and Otto's text [2]. Esparza and Rossmanith, who learnt about the regularity of $pre^*(L)$ from Bouajjani, read [2]. Their attention was attracted to a remark stating that the regularity of $pre^*(L)$ could be applied to problems on context-free grammars. They observed in [9] that this was indeed the case: a number of classical problems (those considered in the previous section) could be reduced to the computation of the set of predecessors of suitable regular languages. However, they were only able to provide an $O(p \cdot s^4)$ algorithm for the computation of $pre^*(L)$.

Independently of Bouajjani and Maler, Finkel, Willems and Wolper also studied in 1996 the model checking problem for pushdown automata. They provided a model checking procedure based on a saturation algorithm [10]. When Esparza read this paper, he observed that the saturation algorithm could be adapted to the computation of $pre^*(L)$, leading to an improved complexity bound of $O(p \cdot s^3)$.

Finally, an anonymous referee has pointed out that an algorithm running in $O(p \cdot s^3)$ time can also be obtained by reducing the problem of computing $pre^*(L)$ to HORNSAT, the satisfiability problem for propositional Horn clauses. The reduction proceeds as follows. Define a set of boolean variables $B = \{X^A_{qq'} \mid q, q' \in Q, A \in V\}$. For each production of the grammar construct a set of Horn clauses with the following intended meaning: in the least model of the set of clauses, $X^A_{qq'}$ is true if and only if $(q, A, q') \in \delta_{pre^*}$. For instance, for a production $A \to BC$ we construct the set

$$\{\neg X^B_{qq'} \vee \neg X^C_{q'q''} \vee X^A_{qq''} \mid q, q', q'' \in Q\}$$

Other productions are dealt with similarly. The construction yields a total of $O(p \cdot s^3)$ clauses. Dowling and Gallier present in [8] two algorithms that compute the minimal model in linear time, and so we obtain an $O(p \cdot s^3)$ time algorithm.

Interestingly, in [8] HORNSAT is solved by reduction to the emptiness problem for context-free grammars, which is one of the applications of our algorithm. So Algorithm 2 can also be used to solve HORNSAT, and in fact it does so in linear time.

7 Conclusions

In our opinion, our adaptation of Büchi's and Book and Otto's result has a number of pedagogical merits that make it very suitable for an undergraduate course on formal languages and automata theory: it is appealing and easy to understand, its correctness proof is simple, it applies the theory of finite automata to the study of context-free languages, and it provides a unified view of several standard algorithms.

Acknowledgments

It is a pleasure to thank the two anonymous referees of this paper. One of them pointed out the connection to HORNSAT, and the other spotted a mistake in a former version of Algorithm 2.

References

1. A. V. Aho, J. E. Hopcroft and J. D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1976.
2. R. F. Book and F. Otto. *String-Rewriting Systems.* Texts and Monographs in Computer Science. Springer, 1993.
3. A. Bouajjani and O. Maler. Reachability Analysis of Pushdown Automata. Proceedings of the INFINITY '96 Workshop, Technical Report MIP-9614, Faculty of Mathematics and Computer Science, University of Passau (1996).
4. A. Bouajjani, J. Esparza, and O. Maler. Reachability Analysis of Pushdown Automata: Application to Model-Checking. Proceedings of CONCUR '97, A. Mazurkiewicz and J. Winkowski (eds.), LNCS 1243, 135–150 (1997).
5. J. R. Büchi. Regular Canonical Systems and Finite Automata. Arch. Math. Logik Grundlagenforschung 6, 91–111 (1964).
6. J. R. Büchi. *Finite Automata, Their Algebras and Grammars*, D. Siefkes (ed.). Springer, (1988).
7. D. Caucal. On the Regular Structure of Prefix Rewriting. Theoretical Computer Science 106(1), 61–86 (1992).
8. W.F. Dowling and J.H. Gallier. Linear-time Algorithms for Testing the Satisfiability of Propositional Horn Formulae. Journal of Logic Programming 3, 267–284 (1984).
9. J. Esparza and P. Rossmanith. An Automata Approach to Some Problems on Context-free Grammars. Foundations of Computer Science: Potential, Theory, Cognition, C. Freksa, M. Jantzen, R. Valk (eds.), LNCS 1337, 143–152 (1997).
10. A. Finkel, B. Willems, and P. Wolper. A Direct Symbolic Approach to Model-checking Pushdown Systems. Electronic Notes in Theoretical Computer Science 9, Proceedings of the INFINITY '97 Workshop (1997).
11. J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, 1979.
12. D. H. Younger. Recognition and Parsing of Context-free Languages in time n^3. *Information and Control*, 10:189–208, 1967.