An Automata Approach to Some Problems on Context-Free Grammars

J. Esparza and P. Rossmanith

Institut für Informatik, Technische Universität München
Arcisstr. 21, D-80290 München, Germany

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. We hope that Wilfried Brauer will consider these results for inclusion in the next edition of his textbook on automata theory [5].

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; 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, 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 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 [6] 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* $\widehat{\delta}\colon (Q \times \Sigma^*) \to 2^Q$ by:

- $\widehat{\delta}(q, \epsilon) = \{q\}$,
- $\widehat{\delta}(q, a) = \delta(q, a)$, 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 $\stackrel{*}{\Rightarrow}$ 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 $\stackrel{*}{\Rightarrow}$ is the reflexive and transitive closure of $\Rightarrow$. If $\alpha \stackrel{*}{\Rightarrow} \beta$, 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,

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

3 Computation of *pre**

Let $G = (V, T, P, S)$ be a fixed context-free grammar. Given an NFA M recognizing a regular set $L(M) \subseteq \Sigma^*$, we wish to construct another NFA 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.

Algorithm 1

Input: an NFA $M = (Q, \Sigma, \delta, q_0, F)$
Output: an NFA $M' = (Q, \Sigma, \delta', q_0, F)$ with $L(M') = pre^*(L(M))$

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

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 heavier 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.

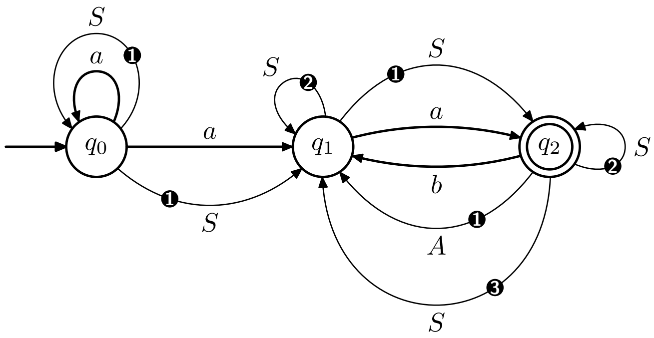


Fig. 1. Illustration of Algorithm 1

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

Lemma 1. $pre^*(L(M)) \subseteq L(M')$.

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 $L(M_i) \subseteq L(M')$, 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 $\alpha = \alpha_1 A \alpha_2 \in L(M_{i+1})$. $\square$

Lemma 2. $L(M') \subseteq pre^*(L(M))$.

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 $L(M')$ 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

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

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

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

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 $\alpha \in pre^*(L(M))$. $\square$

The running time of Algorithm 1 in the size of the input automaton is easy to estimate.[1] Let $n = |Q|$ be the number of states of the input automaton M. Since δ' contains at most $O(n^2)$ elements, the **repeat-until** loop is executed $O(n^2)$ times. The **for** loop is executed $\Theta(n^2)$ times. Checking whether $q' \in \widehat{\delta'}(q, \beta)$ holds can be done by simulating the NFA $(Q, \Sigma, \delta', q, F)$ on input β, which requires $O(n^2)$ time (see [1], pp. 327–329). Adding an element to δ' takes $O(1)$ time (assume for instance that δ' is stored as a bit matrix). So the running time is $O(n^6)$.

4 Improving the complexity

Algorithm 1 is very simple, but not efficient. In this section we present a new algorithm, Algorithm 2, with a running time of $O(n^4)$. It works for grammars with productions of the form $A \to BC$, $A \to a$, or $A \to \epsilon$, i.e., grammars in Chomsky normal form extended with ϵ-productions. Observe that every context-free grammar can be efficiently transformed into one in this form. The check $q' \in \widehat{\delta}(q, \beta)$ is now easier, since β has length at most 2.

We first observe that productions of the form $A \to a$ or $A \to \epsilon$ 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 initialization phase. It remains to deal properly with productions of the form $A \to BC$. In each iteration of the **repeat-until** loop, Algorithm 1 goes over all pairs of states

[1] It is also interesting to examine the complexity in the size of the grammar, but this is out of the scope of this little note.

(q, q'), checks if $q' \in \widehat{\delta}(q, BC)$, and if so adds the triple (q, A, q') to δ'. The procedure takes $\Theta(n^4)$ time. Algorithm 2 adds exactly the same transitions, but more efficiently: it goes through all states q'', and it computes for each of them the sets $L(B, q'') = \{q \in Q \mid q \xrightarrow{B} q''\}$ and $R(q'', C) = \{q' \in Q \mid q'' \xrightarrow{C} q'\}$; the whole procedure takes $\Theta(n^2)$ time. Then, it adds to δ' the union over all q'' of the triples $L(B, q'') \times \{A\} \times R(q'', C)$.

Actually, one last refinement is needed in order to achieve $O(n^4)$ running time: Algorithm 2 uses two sets of states $L(X, q)$, $L'(X, q)$ (and two analogous sets $R(q, X)$ and $R'(q, X)$). $L'(X, q)$ is reinitialised to the empty set in each iteration of the **repeat-until** loop; it stores the states q' for which a transition $q' \xrightarrow{X} q$ has been added *during the current iteration*. $L(X, q)$ is initialised only once before the execution of the **repeat-until** loop; it stores all the states q' for which a transition $q' \xrightarrow{X} q$ has been added so far. So the new triples that have to be added to δ' after each iteration are

$$(L'(B, q) \times \{A\} \times R(q, C)) \,\cup\, (L(B, q) \times \{A\} \times R'(q, C))$$

Algorithm 2

Input: an NFA $M = (Q, \Sigma, \delta, q_0, F)$
Output: an NFA $M' = (Q, \Sigma, \delta', q_0, F)$ with $L(M') = \mathit{pre}^*(L(M))$

$\delta' \leftarrow \delta$;
for $q \in Q,\ A \to \epsilon \in P$ **do** $\delta' \leftarrow \delta' \cup \{(q, A, q)\}$ **od**;
for $q, q' \in Q,\ A \to a \in P$ **do**
  if $(q, a, q') \in \delta'$ **then** $\delta' \leftarrow \delta' \cup \{(q, A, q')\}$ **fi**
od;
for $q \in Q,\ X \in V$ **do** $L(X, q) \leftarrow \emptyset$; $R(q, X) \leftarrow \emptyset$ **od**;
repeat
  for $q \in Q,\ X \in V$ **do** $L'(X, q) \leftarrow \emptyset$; $R'(q, X) \leftarrow \emptyset$ **od**;
  for $q, q' \in Q,\ A \to BC \in P$ **do**
    if $(q', B, q) \in \delta' \wedge q' \notin L(B, q)$ **then**
      $L(B, q) \leftarrow L(B, q) \cup \{q'\}$; $L'(B, q) \leftarrow L'(B, q) \cup \{q'\}$
    fi;
    if $(q, C, q') \in \delta' \wedge q \notin R(q, C)$ **then**
      $R(q, C) \leftarrow R(q, C) \cup \{q'\}$; $R'(q, C) \leftarrow R'(q, C) \cup \{q'\}$
    fi;
  od;
  for $q \in Q,\ A \to BC \in P$ **do**
    $\delta' \leftarrow \delta' \cup\ (L'(B, q) \times \{A\} \times R(q, C)) \,\cup\, (L(B, q) \times \{A\} \times R'(q, C))$
  od
until δ' does not change any more

The correctness of the algorithm is an immediate consequence of the fact that both algorithms have added exactly the same new transitions after each iteration of the **repeat-until** loop. More precisely: for every $k \geq 1$, after k

iterations of the **repeat-until** loop the variable δ' has exactly the same value in both Algorithm 1 and Algorithm 2.

Let us now examine the running time.

Lemma 3. *If the* **repeat-until** *loop is executed k times, then Algorithm 2 terminates in $O(kn^2) + O(n^3)$ time.*

Proof. The algorithm uses states (q and q'), sets of states (Q, $L(X,q)$, $L'(X,q)$, $R(q,X)$, $R'(q,X)$) and transition tables (δ and δ') as its basic data structures. We assume that states are implemented as numbers in $\{1,\dots,n\}$, while sets of states and transition tables are implemented as bit vectors of length n, respectively length $n^2|V|$.

With this implementation the time complexity of all operations is as follows:

Operation	*Time complexity*
$\delta' \leftarrow \delta$	$O(n^2)$
$(q,a,q') \in \delta'$	$O(1)$
$\delta \leftarrow \delta' \cup \{(q,A,q')\}$	$O(1)$
$L(X,q) \leftarrow \emptyset$	$O(n)$
$q' \in L(B,q)$	$O(1)$
$L'(B,q) \leftarrow L'(B,q) \cup \{q'\}$	$O(1)$
$\delta' \leftarrow \delta' \cup L'(B,q) \times \{A\} \times R(q,C)$	$O(n) + O(n \cdot \lvert L'(B,q)\rvert)$

Only the last line needs some explanation. It works by reading the bit vector of $L'(B,q)$ ignoring empty entries (ignoring an entry takes $O(1)$ time) and performing $\delta' \leftarrow \delta' \cup \{(s,A,s')\}$ for all $s \in R(q,C)$ when finding an entry s in $L'(B,q)$ ($O(n)$ steps).

Let us assume the **repeat-until** loop is executed exactly k times. Using the above table we easily see that everything before the **repeat-until** loop runs in time $O(n^2)$. The first **for** loop in the body of the **repeat-until** loop runs in time $O(n^2)$, the second **for** loop also in time $O(n^2)$. Not counting the last **for** loop, the overall time requirement is therefore $O(n^2) + k \cdot O(n^2) = O(kn^2)$.

Let $L'_i(B,q)$, $L_i(B,q)$, $R'_i(q,C)$, and $R_i(q,C)$ denote the values of $L'(B,q)$, $L(B,q)$, $R'(q,C)$, and $R(q,C)$ after the ith iteration of the **repeat-until** loop. The last **for** loop then requires

$$T(i) = \sum_{\substack{q \in Q \\ A \to BC \in P}} \Big(O(n) + O(n \cdot |L'_i(B,q)|) + O(n \cdot |R'_i(q,C)|) \Big)$$
$$= O(n^2) + O(n \cdot \sum_{\substack{q \in Q \\ A \to BC \in P}} |L'_i(B,q)|) + O(n \cdot \sum_{\substack{q \in Q \\ A \to BC \in P}} |R'_i(q,C)|)$$

steps during the ith iteration of the **repeat-until** loop. The total running time of the algorithm is therefore $O(kn^2) + T(1) + T(2) + \cdots + T(k)$.

Since the $L'_i(B,q)$'s for $i = 1, \ldots, k$ as well as the $R'_i(q,C)$'s are disjoint, we have

$$\sum_{i=1}^{k} |L'_i(B,q)| \leq n \quad \text{and} \quad \sum_{i=1}^{k} |R'_i(q,C)| \leq n.$$

So the sum $T(1) + T(2) + \cdots + T(k)$ yields

$$\sum_{i=1}^{k} O(n^2) + \sum_{\substack{q \in Q \\ A \to BC \in P}} \big(O(n \cdot \sum_{i=1}^{k} |L'_i(B,q)|) + O(n \cdot \sum_{i=1}^{k} |R'_i(q,C)|) \big) = O(kn^2) + O(n^3)$$

The overall running time is therefore $O(kn^2 + n^3)$. ☐

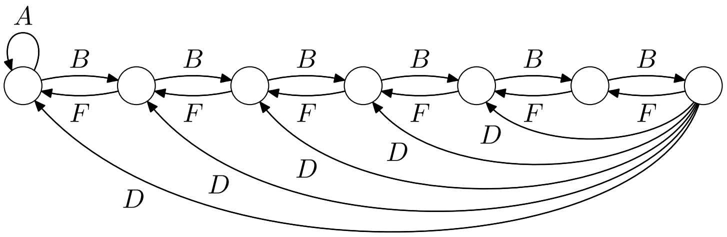
We immediately get:

Theorem 1. *Algorithm 2 runs in $O(n^4)$ time.*

Proof. Since M' has $O(n^2)$ transitions, the **repeat-until** loop is executed $O(n^2)$ times. Use now Lemma 3. ☐

It requires a bit of thought to find a grammar and a family of NFAs for which the **repeat-until** loop is executed $\Theta(n^2)$ times and Algorithm 2 runs in $\Theta(n^4)$ time. The next figure shows an example (more precisely, the figure shows the grammar and a member of the family):

$H \to DA$
$C \to AB$
$C \to CB$
$E \to CH$
$G \to EB$
$A \to FG$



5 A special case

In this section, we show that Algorithm 2 needs only $O(n^3)$ time for linear NFAs, a special class of inputs relevant for the next section. An NFA is *linear* if there is a bijection $l: Q \to \{1, \ldots, n\}$ such that $l(q) \leq l(q')$ if and only if $q \xrightarrow{\alpha} q'$ for some word α.[2]

Lemma 4. *If the input NFA M of Algorithm 2 is linear, then the* **repeat-until** *loop is executed at most $O(n)$ times.*

[2] Observe that all circuits of a linear NFA are of the form $q \xrightarrow{A} q$. We call them *self-loops*.

Proof. Observe first that any new transition $q \xrightarrow{A} q'$ added by the algorithm to a linear input M satisfies $l(q) \neq l(q')$. Therefore, if the input M is linear, so is the output M'.

Let the *width* of a transition $q \xrightarrow{A} q'$ be $l(q') - l(q)$. We show that transitions $q \xrightarrow{A} q'$ of width i are added after at most $(i+1)|V|$th iterations of the **repeat-until** loop.

We proceed by induction on i. The base of the induction is the case $i = 0$. We then have $q = q'$ and so the transition $q \xrightarrow{A} q'$ is in fact the self-loop $q \xrightarrow{A} q$. If $q \xrightarrow{A} q$ is added by one of the two initial **for** loops, then it has been added after 0 iterations of the **repeat-until** loop, and we are done. So assume that $q \xrightarrow{B} q', q' \xrightarrow{C} q$ and a rule $A \to BC$ yield together $q \xrightarrow{A} q$. Since the current automaton is linear, we have $q = q'$, and so both $q \xrightarrow{B} q$ and $q \xrightarrow{C} q$ are self-loops. Since each state can have at most $|V|$ self-loops labelled with variables, and in each iteration of the **repeat-until** loop at least one of them is added, $q \xrightarrow{A} q$ is added during the first $|V|$ iterations.

Now, let the width of $q \xrightarrow{A} q'$ be $i > 0$. Again, if $q \xrightarrow{A} q'$ is added by one of the two initial **for** loops, then we are done as before. So assume that there is a state q'' with $q \xrightarrow{B} q''$ and $q'' \xrightarrow{C} q'$ and a production $A \to BC \in P$. Clearly, the widths of $q \xrightarrow{B} q''$ and $q'' \xrightarrow{C} q'$ are at most i. If these two widths are smaller than i, then by the induction hypothesis $q \xrightarrow{B} q''$ and $q'' \xrightarrow{C} q'$ are added after at most $(i-1)|V|$ iterations. So $q \xrightarrow{A} q'$ is added after at most $(i-1)|V| + 1 \leq i|V|$ iterations.

Let us now assume that the width of $q \xrightarrow{B} q''$ is i or the width of $q'' \xrightarrow{C} q'$ is i. Then $q = q''$ or $q' = q''$. If $q = q''$ we say $q \xrightarrow{A} q'$ *depends directly* on $q \xrightarrow{C} q'$. If $q' = q''$ we say $q \xrightarrow{A} q'$ *depends directly* on $q \xrightarrow{B} q'$.

In general we have direct dependency chains $q \xrightarrow{A} q'$, $q \xrightarrow{A'} q'$, $q \xrightarrow{A''} q'$,…, $q \xrightarrow{A^{(k)}} q'$, where $q \xrightarrow{A^{(t)}} q'$ depends directly on $q \xrightarrow{A^{(t+1)}} q'$ for $t = 0, \ldots, k-1$. Since no two transitions of the chain can be identical and there are only $|V|$ variables, we have $k \leq |V|$. The last transition $q \xrightarrow{A^{(k)}} q'$ of a maximal chain does not depend directly on a transition, and so it is added because of transitions with width smaller than i. By induction hypothesis this occurs after at most $(i-1)|V|$ iterations. Then $q \xrightarrow{A^{(t)}} q'$ is added after at most $(i-1)|V| + k - t$ iterations, and $q \xrightarrow{A} q'$ after $(i-1)|V| + k \leq i|V|$ iterations.

Since the width of all transitions is at most $n = |Q|$, all transitions are added after at most $n|V|$ iterations of the **repeat-until** loop. So δ' does not change any more during the $n|V|+1$ iteration, and the loop is executed $O(n)$ times. □

It follows from Lemma 5 and Lemma 3 that Algorithm 2 runs in $O(n^3)$ time for linear NFAs.

6 Applications

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

We fix a context-free grammar $G = (V, T, P, S)$ for the rest of this section.

In order to avoid redundant symbols in G it is convenient to compute the set of *useless* variables ([6], p.88). Recall that $X \in V$ is *useful* if there is a derivation $S \overset{*}{\Rightarrow} \alpha X \beta \overset{*}{\Rightarrow} w$ 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 $X \in pre^*(T^*)$. Compute the automata accepting $pre^*(T^* X T^*)$ and $pre^*(T^*)$ using Algorithm 2, and check if they accept S and X, respectively.

Nullable variables have to be identified when eliminating ϵ-productions ([6], p. 90). A variable X is *nullable* if $X \overset{*}{\Rightarrow} \epsilon$. To decide the nullability of a variable observe that X is nullable if and only if $X \in pre^*(\{\epsilon\})$.

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 in constant time if it accepts S. Since there is a linear automaton with $n+1$ states recognizing $\{w\}$, the complexity of the algorithm is $O(n^3)$. This is also the complexity of the CYK-algorithm usually taught to undergraduates [7].

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 n states, compute a deterministic automaton for $\overline{L}$ in $O(n)$ time, and check $S \notin pre^*(\overline{L})$ in $O(n^4)$.

Similarly, to decide if $L(G)$ and L are disjoint, check whether $S \notin pre^*(L)$. In the example of Figure 1 the languages are disjoint because there is no transition $q_0 \xrightarrow{S} q_2$.

To decide if $L(G)$ is empty, check whether $L(G)$ is contained in the empty language, which is regular. In this case the automaton for $\overline{L}$ has just one state.

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 [6], 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 $X \overset{*}{\Rightarrow} \alpha X \beta$. This is the case if and only if $X \in pre^*(\Sigma^+ X \Sigma^* \cup \Sigma^* X \Sigma^+)$.

7 Conclusions

In our opinion, our adaptation of Book and Otto's technique 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.

Acknowledgements We are very grateful to Ahmed Bouajjani and Oded Maler, who drew our attention to Book and Otto's result, and applied it, together with the first author, to the analysis of pushdown automata [3, 4]. Many thanks to an anonymous referee for pointing out an important mistake in a former version of the paper, and for very helpful suggestions.

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 INFINITY '96, tech, rep. MIP-9614, Univ. Passau, 1996.
4. A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Application to model checking. To appear in CONCUR '97.
5. W. Brauer. *Automatentheorie.* Teubner, 1984.
6. J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, 1979.
7. D. H. Younger. Recognition and parsing of context-free languages in time n^3. *Information and Control*, 10:189–208, 1967.