A Uniform Framework for Problems on Context-Free Grammars

Javier Esparza Peter Rossmanith Stefan Schwoon

Technische Universität München, Germany
{esparza,rossmani,schwoon}@in.tum.de

Abstract

In [2], Bouajjani and others presented an automata-based approach to a number of elementary problems on context-free grammars. This approach is of pedagogical interest since it provides a uniform solution to decision procedures usually solved by independent algorithms in textbooks, e.g. [7]. This paper improves upon [2] in a number of ways. We present a new algorithm which not only has a better space complexity but is also (in our opinion) easier to read and understand. Moreover, a closer inspection reveals that the new algorithm is competitive to well-known solutions for most (but not all) standard problems.

1 Introduction

Textbooks on formal languages and automata (e.g. [7]) discuss solutions for certain standard problems on context-free grammars like tests for membership, emptiness, and finiteness. For many of these problems independent solutions are given, e.g. reductions to a graph-theoretic problem for finiteness or the well-known algorithm by Cocke, Younger and Kasami [8, 9] for the membership problem. Closer inspection reveals that several problems reduce to reachability questions between sentential forms.

It has been observed that finite automata can play a useful role in solving reachability questions. Book and Otto, in their work about string rewriting systems [1], established a connection to context-free grammars. More to the point, they discovered that for a regular set L of strings over variables and terminals, the predecessors of L – i.e. the strings from which some element of L can be derived by repeatedly applying the productions – also form a regular set. Very similar results were discovered independently by Büchi [3] and Caucal [4]. Book and Otto also pointed out the applicability of their algorithm to problems on context-free grammars without going into details.

Esparza and Rossmanith [6], attracted by this idea, showed that indeed the aforementioned problems, among others, could be solved by computing predecessors of suitable regular languages and provided an $O(ps^4)$ time algorithm for the task, where p is the size of the productions and s the number of states in an automaton for the regular language. Later on (in collaboration with other authors [2]), they improved the space and time complexity to $O(ps^3)$. It has been pointed out that a reduction to the satisfiability problem for Horn clauses solves the problem within the same time and space constraints.

In this paper, we provide a further improvement which reduces the space requirements to $O(ps^2)$ without affecting time complexity. This improvement makes the algorithm competitive with standard algorithms for some classical problems, e.g. the membership problem. The purpose of the paper is therefore to show how a number of different decision procedures for a number of problems can be replaced by a uniform framework, often without sacrificing efficiency. Since the algorithm is also relatively easy to understand, we hope that this result will be of value for educational purposes.

The rest of the paper is structured as follows: In section 2 we introduce some notations and other preliminaries. In sections 3 and 4 we present our algorithm. We discuss applications and give a comparison with other algorithms in section 5.

2 Notations

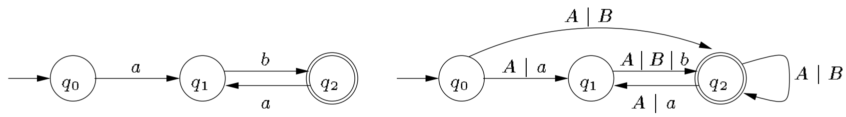
We use the notations of [7] for finite automata and context-free grammars.

Fix a context-free grammar $G = (V, T, P, S)$ for the rest of the section where V is the set of variables and T the set of terminals. Let $\Sigma = V \cup T$. The set of productions P generates reachability relations $\Rightarrow$ and $\stackrel{*}{\Rightarrow}$ between strings over Σ in the following sense: If $A \to \beta$ is a production of P and α and γ are arbitrary strings over Σ, then $\alpha A\gamma \Rightarrow \alpha\beta\gamma$ holds (we also say that $\alpha\beta\gamma$ is derived from $\alpha A\gamma$ by application of $A \to \beta$ or that $\alpha A\gamma$ is a direct predecessor of $\alpha\beta\gamma$). The relation $\stackrel{*}{\Rightarrow}$ is the reflexive and transitive closure of $\Rightarrow$. If $\alpha \stackrel{*}{\Rightarrow} \beta$ (for $\alpha, \beta \in \Sigma^*$) then we call α a predecessor of β. A string α is called a sentential form of G if $S \stackrel{*}{\Rightarrow} \alpha$. For a set $L \subseteq \Sigma^*$ we denote by $pre^*(L)$ the predecessors of elements of L:

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

We can represent regular subsets of Σ^* with finite automata. Given an automaton $A = (Q, \Sigma, \delta, q_0, F)$, the transition relation $\to \subseteq Q \times \Sigma^* \times Q$ is inductively defined as follows:

- $q \xrightarrow{\varepsilon} q$ for all $q \in Q$.
- if $(q, a, q') \in \delta$, then $q \xrightarrow{a} q'$.
- if $(q, a, q'') \in \delta$ and $q'' \xrightarrow{w} q'$ for some $q'' \in Q$, then $q \xrightarrow{aw} q'$.



$A \to a \mid BB$
$B \to AB \mid b$

Figure 1: Example automaton before (left) and after the algorithm (right).

Clearly, the language accepted by the automaton is

$$L(A) = \{\, w \in \Sigma^* \mid \exists q_f \in F \colon q_0 \xrightarrow{w} q_f \,\}.$$

3 Computing $pre^*(L)$

In [1], Book and Otto show that for a context-free grammar $G = (V, T, P, S)$ and a regular language $L \subseteq \Sigma^*$ the set $pre^*(L)$ is also regular. Moreover, they provide an algorithm for its computation.

We start from an automaton $A = (Q, \Sigma, \delta, q_0, F)$ accepting L. We obtain an automaton A_{pre^*} accepting $pre^*(L)$ by a saturation procedure. This procedure adds transitions to A according to the following saturation rule:

If $A \to \beta \in P$ and $q \xrightarrow{\beta} q'$ in the current automaton, add a transition (q, A, q').

Notice that no new states are added, and that all added transitions are labelled with variables. Consider an example where the grammar has rules $A \to a \mid BB$ and $B \to AB \mid b$. We apply the algorithm to the automaton in the left half of figure 1. The algorithm will add transitions (q_0, A, q_1), (q_1, B, q_2), and (q_2, A, q_1) because of the productions $A \to a$ and $B \to b$. Because of $B \to AB$, we now get (q_0, B, q_2) and (q_2, B, q_2) which in turn lets us apply the production $A \to BB$ to add (q_0, A, q_2), (q_1, A, q_2), and (q_2, A, q_2). The algorithm in the right half depicts the result.

The correctness of the procedure was shown in [2], but for the sake of completeness we repeat the proof here.

Termination: The algorithm terminates because no new states are added and so the number of possible transitions is finite.

Lemma 3.1 $L(A_{pre^*}) \subseteq pre^*(L(A))$

Proof: We show that the lemma holds initially and is kept invariant by the addition rule. Let $\xrightarrow[i]{}$ denote the transition relation of the automaton after the i-th addition. We proceed by induction on i, i.e. we show that the lemma holds after i additions.

Basis. $i = 0$. Here, $\xrightarrow[i]{}$ is the transition relation of A and clearly $L(A) \subseteq pre^*(L(A))$.

Step. $i > 0$. Let α be a word accepted by the automaton obtained after the i-th step, and let $q_0 \xrightarrow[i]{\alpha} q_f$ be the path by which it is accepted. Either $q_0 \xrightarrow[i-1]{\alpha} q_f$ holds, too, then by the induction hypothesis $\alpha \in pre^*(L(A))$. Otherwise, let (q, A, q') be the transition added in the i-th step. Then α can be written as $\alpha_0 A \alpha_1 \ldots \alpha_n$ where $n \geq 1$ is the number of occurences of the new transition, and the path has the form $q_0 \xrightarrow[i-1]{\alpha_0} q \xrightarrow[i]{A} q' \xrightarrow[i-1]{\alpha_1} q \ldots q' \xrightarrow[i-1]{\alpha_n} q_f$. The addition rule dictates that there is some production $A \rightarrow \beta$ and that $q \xrightarrow[i-1]{\beta} q'$. Then $\alpha' = \alpha_0 \beta \alpha_1 \ldots \alpha_n$ is a direct successor of α, and $q_0 \xrightarrow[i-1]{w'} q_f$. By the induction hypothesis, α' is in $pre^*(L(A))$ and such is α. $\square$

Lemma 3.2 $pre^*(L(A)) \subseteq L(A_{pre^*})$

Proof: Let α be an element of $pre^*(L(A))$, and select $\beta \in L(A)$ such that $\alpha \overset{*}{\Rightarrow} \beta$. Let i be the length of the derivation $\alpha \overset{*}{\Rightarrow} \beta$. We proceed by induction on i.

Basis. $i = 0$. Then $\alpha = \beta$, and α is accepted by both A and A_{pre^*}.

Step. $i > 0$. Then there exists γ such that $\alpha \Rightarrow \gamma$ and $\gamma \overset{*}{\Rightarrow} \beta$ in $i-1$ steps. Because of the former, there exist $\alpha_1, \alpha_2, \alpha_3 \in \Sigma^*$ and $A \in V$ such that $\alpha = \alpha_1 A \alpha_3$, $\gamma = \alpha_1 \alpha_2 \alpha_3$ and $A \rightarrow \alpha_2 \in P$. Because of the latter, by the induction hypothesis γ is accepted by A_{pre^*}, so there exists a path $q_0 \xrightarrow{\alpha_1} q \xrightarrow{\alpha_2} q' \xrightarrow{\alpha_3} q_f$ for some $q_f \in F$ in A_{pre^*}. By the saturation rule, (q, A, q') is a transition of A_{pre^*} and so α is accepted, too. $\square$

4 The Algorithm

We present an efficient implementation of the procedure from the previous section. The new algorithm works on grammars which are in Chomsky normal form extended with unit productions and ε-productions, i.e. we allow productions of the form $A \rightarrow BC$, $A \rightarrow a$, $A \rightarrow B$, and $A \rightarrow \varepsilon$. Notice that this is not a real restriction since such a normal form can be obtained from an arbitrary context-free grammar in linear time and with a linear growth in the size of the productions. Also, assume that every element of Σ occurs in at least one production; otherwise, a missing element can be removed in linear time. The algorithm is shown below.

Transitions which are known to belong to A_{pre^*} are collected in the sets δ' and η. The latter contains transitions which still need to be examined. No transition is examined more than once.

Input: a context-free grammar $G = (V, T, P, S)$ in extended CNF;
an automaton $A = (Q, \Sigma, \delta, q_0, F)$
Output: the automaton A_{pre^*}

1 $\delta' \leftarrow \emptyset$; $\eta \leftarrow \delta$;
2 **for all** $A \to a \in P$, $(q, a, q') \in \delta$ **do** $\eta \leftarrow \eta \cup \{(q, A, q')\}$;
3 **for all** $A \to \varepsilon \in P$, $q \in Q$ **do** $\eta \leftarrow \eta \cup \{(q, A, q)\}$;
4 **while** $\eta \neq \emptyset$ **do**
5   remove $t = (q, B, q')$ from η;
6   **if** $t \notin \delta'$ **then**
7     $\delta' \leftarrow \delta' \cup \{t\}$;
8   **for all** $A \to B \in P$ **do**
9       $\eta \leftarrow \eta \cup \{(q, A, q')\}$;
10   **for all** $A \to BC \in P$ **do**
11   **for all** q'' s.t. $(q', C, q'') \in \delta'$ **do**
12         $\eta \leftarrow \eta \cup \{(q, A, q'')\}$;
13   **for all** $A \to CB \in P$ **do**
14   **for all** q'' s.t. $(q'', C, q) \in \delta'$ **do**
15         $\eta \leftarrow \eta \cup \{(q'', A, q')\}$;
16
17 **return** $(Q, \Sigma, \delta', q_0, F)$

First, we observe that productions of the form $A \to a$ can be dealt with once at the beginning since the algorithm will not add transitions labelled by terminals. Similarly, the ε-productions are only regarded during initialisation. It remains to deal with productions of the form $A \to B$ and $A \to BC$ which the algorithm does in a fairly straightforward way.

We prove correctness of the new algorithm by showing that it is equivalent to the algorithm from section 3.

Termination: Initially, η contains finitely many transitions. Every iteration of the while-loop removes one of them. An iteration can add transitions only if the check in line 6 is positive which is only finitely often the case (since the size of δ' is bounded). Therefore η will eventually become empty, and the algorithm terminates.

It remains to show that upon termination δ' is δ_{pre^*}, i.e. the smallest set containing δ and closed under the saturation rule. Since every element of η eventually goes into δ', we examine additions to η. These include the additions of δ in line 1, and it is easy to see that all other additions fulfill the saturation rule. So $\delta' \subseteq \delta_{pre^*}$ holds. For the other direction, $\delta' \supseteq \delta_{pre^*}$, we already know that $\delta \subseteq \delta'$. Every other transition of δ_{pre^*} is due to the saturation rule. Assume that $A \to \beta \in P$ and $q \xrightarrow{\beta} q'$ in A_{pre^*}.

- if $\beta = \varepsilon$, then $q = q'$ and (q, A, q') is added in line 3.
- if $\beta = a$, then (q, a, q') must be a transition of δ (as already observed, the algorithm only adds transitions labelled with non-terminals). Then (q, A, q') is added in line 2.
- if $\beta = B$, then $t_1 = (q, B, q')$ must be a transition of A_{pre^*}. When t_1 is transferred from η into δ', (q, A, q') is added in line 9.
- if $\beta = BC$, let q'' be a state such that $t_1 = (q, B, q'')$ and $t_2 = (q'', C, q')$ are transitions in A_{pre^*}. Depending on the order in which t_1 and t_2 occur in η, (q, A, q') is added in line 12 or 15.

In [2] it was observed that a straightforward implementation of Book and Otto's procedure may lead to an $O(p^2 s^6)$ time algorithm if p is the number of productions and s the number of states in A. In [6] an $O(ps^4)$ time algorithm was proposed, and in [2] the bounds were improved to $O(ps^3)$ time and space. The algorithm above takes the same amount of time but only $O(ps^2)$ space. Moreover, we will see later that its time complexity becomes $O(ps^2)$ for unambiguous grammars.

Imagine that δ' is implemented by a bit-array with one bit for each element of $Q \times \Sigma \times Q$. Since $|\Sigma| = O(p)$ this takes $O(ps^2)$ space. Membership test and addition can then be performed in constant time. Let η be implemented as a stack such that finding an element, addition and removal take constant time. Moreover, if we use a second bit-array that keeps track of all additions to η ever, we can prevent duplicate entries in η while still keeping the operations constant in time. The space needed for these structures is $O(ps^2)$, too.

Line 1 takes at most $O(ps^2)$ time (the maximum size of δ), the same holds for line 2 (there are at most s^2 many transitions labelled with a). Line 3 takes $O(ps)$ time. To find out the complexity of the main loop, imagine that for every variable A, there exist three sets of productions, A_{chain}, A_{front}, and A_{back}. A production $A \to B$ would be put into B_{chain}; a production $A \to BC$ would occur in both B_{front} and C_{back}. These sets can be constructed prior to starting the algorithm proper in $O(p)$ time and space. Then the loops in lines 8, 10 and 13 need to traverse only the sets B_{chain}, B_{front} and B_{back}, i.e. those rules which are relevant for the selected transition t. Since no element is added to δ' twice, line 9 is executed at most once for every rule $A \to B$ and states q, q', i.e. $O(ps^2)$ times. Similarly, lines 12 and 15 are both executed at most once for every combination of productions $A \to BC$ and states q, q', q'', i.e. $O(ps^3)$ times. Line 5 is executed at most $O(ps^2)$ times.

From these observations it follows that the algorithm takes $O(ps^3)$ time and $O(ps^2)$ space. As an aside, notice that in lines 11 and 14 it is not necessary to iterate q'' over all elements of Q. Instead, one could maintain a set $Back(q', C)$ for every pair $q' \in Q$ and $C \in V$ such that $Back(q', C) \ni q''$ exactly if $(q', C, q'') \in \delta'$.

Line 11 then only needs to go through $Back(q', C)$. These sets can be updated in constant time whenever a transition is added to δ'. Since the combined size of all *Back* sets is no larger than $|\delta'|$, neither space nor time complexity are affected. Similarly, one can maintain sets $Front(C, q)$ for line 14.

5 Applications

In this section we show how several standard problems for context-free grammars can be solved using the pre^* algorithm. Let $G = (V, T, P, S)$ be a context-free grammar and let $p = |P|$.

Membership: Given a string $w \in T^*$ of length n, is $w \in L(G)$? To solve the question check whether $S \stackrel{*}{\Rightarrow} w$ holds, i.e. if $S \in pre^*(\{w\})$. An automaton accepting $\{w\}$ has $n + 1$ states. Hence, for a fixed grammar, the algorithm takes $O(n^3)$ time and $O(n^2)$ space which is also the complexity of the well-known CYK algorithm for the same problem.

However, there is more. Earley's algorithm [5] also takes cubic time in general, but only quadratic time for unambiguous grammars. Our algorithm has the same property. Assume that G has no unreachable variables. Then unambiguity implies for any pair of derivations $A \to BC \stackrel{*}{\Rightarrow} u_1u_2$ and $A \to DE \stackrel{*}{\Rightarrow} u_3u_4$ with $u = u_1u_2 = u_3u_4$ that $BC = DE$, $u_1 = u_3$ and $u_2 = u_4$ (otherwise there would be two different parse trees for a word containing u). If w consists of terminals $w_1 \dots w_n$, an automaton accepting $\{w\}$ has transitions of the form (q_{i-1}, w_i, q_i) for $1 \leq i \leq n$. When computing $pre^*(\{w\})$, a transition (q_i, A, q_j) implies that $A \stackrel{*}{\Rightarrow} w_{i+1} \dots w_j$. Recall from the analysis of the algorithm that the time complexity is dominated by the number of times lines 12 and 15 are executed, i.e. once for every production $A \to BC$ and states q, q', q'' such that (q, B, q'') and (q'', C, q') are transitions in the automaton. In the unambiguous case we can deduce that once q and q' are known, q'' is fixed. Hence, the lines are executed only $O(ps^2)$ times, and the algorithm takes quadratic time in the number of states.

The pre^* algorithm can be modified to solve the *parsing* problem: Given a word $w \in L(G)$, produce a derivation $S \stackrel{*}{\Rightarrow} w$. The modification consists of adding extra information to each transition produced by the saturation rule. Every time a transition is added, annotate it with the 'reason' for its addition, i.e. with the production and the states by which the saturation rule was fulfilled (notice that this information has constant length for each transition, so the space complexity is not affected by this change). After computing $pre^*(\{w\})$, take the transition (q_0, S, q_n) (if $n = |w|$) and repeatedly exploit the new information until w is derived.

Identifying useless variables: In order to rid a grammar of redundant symbols one computes the set of useless variables. A variable A is called useful if it occurs in the parse tree of some word $w \in L(G)$, i.e. if there is a derivation $S \stackrel{*}{\Rightarrow} w_1 A w_2 \stackrel{*}{\Rightarrow} w$ for some $w_1, w_2 \in T^*$. Otherwise A is useless. Checking if a variable is useful amounts to two tests: checking if A is productive (whether there is some $w' \in T^*$ such that $A \stackrel{*}{\Rightarrow} w'$) and checking if A is reachable, i.e. there exist $w_1, w_2 \in T^*$ such that $S \stackrel{*}{\Rightarrow} w_1 A w_2$. Productive variables can be identified by computing $pre^*(T^*)$. An automaton accepting T^* has one single state, therefore this takes time $O(p)$. Notice that the test yields the set of all productive variables. This time can also be achieved by a careful implementation of the procedure given in [7], Lemma 4.1.

To see whether a variable A is reachable, we check if $S \in pre^*(T^* A T^*)$. An automaton for this set has two states, so this check takes linear time in the size of the grammar, too. To find the set of all reachable variables, however, we need to repeat the check for each variable, so the complexity would become $O(p\,|V|)$. The procedure in [7] finds the set of all reachable variables in $O(p)$, so the pre^* algorithms performs worse in this case.

Emptiness: To check if the language generated by G is empty, test if S is unproductive, i.e. $S \notin pre^*(T^*)$. We could also reduce emptiness to the *containment* problem: Given a regular language L, is $L(G) \subseteq L$? Since $L(G) \subseteq L$ is equivalent to $L(G) \cap \bar{L} = \emptyset$ it is sufficient to check if $S \notin pre^*(\bar{L})$. If $\bar{L}$ can be represented by an automaton with s states, the procedure takes $O(ps^3)$ time and $O(ps^2)$ space. For the emptiness problem we take $L = \emptyset$; again, since $\bar{L} = T^*$ can be represented with one state, the check takes $O(p)$ time and space.

Finiteness: To decide if the language generated by G is finite, we first rid G of all useless symbols and recall Theorem 6.6 from [7]: $L(G)$ is infinite exactly if there is a variable $A \in V$ and strings $u, v \in T^*$ such that $A \stackrel{*}{\Rightarrow} uAv$ and $uv \neq \varepsilon$. Therefore, a solution consists of checking whether $A \in pre^*(T^+ A T^* \cup T^* A T^+)$ holds for some variable A. The whole procedure can be carried out in quadratic time and linear space in the size of the grammar. The algorithm given in [7] is quadratic in both time and space (since it requires conversion to CNF) but could probably be converted into a linear algorithm with a careful implementation.

Nullable variables: In [7] the problem of finding nullable variables is discussed in the context of eliminating ε-productions. A variable A is called nullable if $A \stackrel{*}{\Rightarrow} \varepsilon$ which is equivalent to $A \in pre^*(\{\varepsilon\})$. This condition can be checked in $O(p)$ time using our algorithm. In fact, computing $pre^*(\{\varepsilon\})$ yields the set of all nullable variables. The procedure given in [7] can also be carried out in linear time.

6 Conclusions

The material presented in this paper is mostly of educational value. The concept of computing pre^* is relatively easy to understand and provides a unified view of several standard problems. We think that it would be suitable for undergraduate courses on formal languages and automata theory where it can be used to replace the various independent algorithms usually taught for these problems.

As a decision procedure, the algorithm is easily comprehensible; its complexity analysis is more subtle but shows that the algorithm is equally efficient in comparison to the standard solutions in many cases. Of particular interest is the application to the membership problem where the algorithm is more flexible than CYK.

Similar arguments were put forward in [2]. We have built upon the work presented there and improved it by providing an algorithm with better space complexity and by a more detailed inspection of its applications. Moreover, we think that our algorithm is easier to understand than the one in [2] which contributes to its pedagogical merits.

References

[1] R. F. Book and F. Otto. *String-Rewriting Systems.* Texts and Monographs in Computer Science. Springer-Verlag, Heidelberg, 1993.

[2] A. Bouajjani, J. Esparza, A. Finkel, O. Maler, P. Rossmanith, B. Willems, and P. Wolper. An efficient automata approach to some problems on context-free grammars. *To appear in Information Processing Letters*, 2000.

[3] J. R. Büchi. Regular canonical systems and finite automata. *Arch. Math. Logik*, 6:91–111, 1964.

[4] D. Caucal. On the regular structure of prefix rewriting. *Theoretical Computer Science*, 106(1):61–86, 1992.

[5] J. Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102, 1970.

[6] J. Esparza and P. Rossmanith. An automata approach to some problems on context-free grammars. In *Foundations of Computer Science: Potential, Theory, Cognition*, volume 1337 of *LNCS*, pages 143–152, 1997.

[7] J. E. Hopcroft and J. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, 1979.

[8] T. Kasami. An efficient recognition and syntax algorithm for context-free languages. Technical Report AFCRL-65-758, Air Force Cambridge Research Lab, Bedford, 1965.

[9] D. H. Younger. Recognition and parsing of context-free languages in time n^3. *Information and Control*, 10(2):189–208, 1967.