Comparison of Two Algorithms for Detecting Useless Transitions in Pushdown Automata

Evangelos Chatzikalymnios

Supervisor: Wan Fokkink

Vrije Universiteit Amsterdam
e.chatzikalymnios@student.vu.nl

Abstract. Context-free languages can be expressed by means of pushdown automata. Pushdown automata may contain transitions that are never used in any accepting run. Such transitions do not affect the languages the automata define, but they increase their memory requirements as well as their running time and thus it is desirable to remove them. The aim of this paper is to examine the performance of two different algorithms for detecting useless transitions in pushdown automata. We propose an optimization of the second algorithm with regard to performance and allowed input. Both versions of the second algorithm were implemented and their efficiency was compared to that of an existing implementation of the first algorithm. Performance testing shows that the first algorithm is more efficient than the second one and its optimized version.

1 Introduction

Automata theory is the theoretical foundation of various branches of computer science. In particular, pushdown automata (PDA) are often used to describe context-free languages and are therefore at the core of language design and specification. Moreover, PDAs are utilized in various top-down and bottom-up parsing algorithms used for syntax analysis in the context of compiler design [4]. Further examples include their use in the area of security in order to implement efficient intrusion detection systems [6] and in graphical programming for user interface specification and management [10].

A PDA may contain useless transitions, that is, transitions that are never used during an accepting run of the automaton. Such transitions do not affect the language of the PDA, but may increase its memory requirements and running time. Thus, it is desirable to remove them in order to obtain an equivalent, but more efficient PDA.

In this paper, we discuss two different approaches for detecting useless transitions in PDAs. Algorithm 1 was presented by Fokkink et al. [8], while Algorithm 2a was proposed by Javier Esparza and combines the efficient model-checking procedures, *pre** and *post** presented by Esparza et al. [5], which operate on pushdown systems (PDS) that are in normal form. Furthermore, we propose

extended versions of these model-checking procedures so that they may accept as input PDSs that are not in normal form, and utilize them in an optimized version of Algorithm 2a, Algorithm 2b, which admits a broader range of input. We also discuss two variations of the latter, Algorithms 2c and 2d. In addition, we have implemented *pre** and *post** (and their optimized versions), and Algorithms 2a-2d. Finally, we compare the performance of the implementations of Algorithms 2a-2d to that of an existing implementation of Algorithm 1.

The first algorithm's worst-case time complexity is $\mathcal{O}(Q^4T)$, where Q is the number of states and T the number of transitions of the input GPDA. In contrast, the worst-case time complexity of Algorithm 2a is $\mathcal{O}(S^3T^4)$, where S is the maximum stack string length found in any transition of the GPDA. Algorithms 2b and 2c exhibit the same worst-case time complexity, while Algorithm 2d has a worst-case time complexity of $\mathcal{O}(S^2T^4)$. Thus, Algorithm 1 was predicted to be more efficient in most cases, particularly when the number of transitions was much higher than the number of states.

For the purpose of comparing the performance of these algorithms, *PDAgen*, a tool built by Dick Grune, was used to generate input GPDAs of various sizes and topologies. The testing results support the claim that Algorithm 1 outperforms the other two. In addition, Algorithm 2b and its variations are found to outperform Algorithm 2a on average as they seem to avoid certain superfluous operations.

The paper is structured as follows. Section 2 discusses some previous work related to the current research. Section 3 provides basic definitions and required theory. Section 4 briefly describes Algorithm 1. Section 5 discusses Algorithm 2a and the model-checking procedures *pre** and *post**. Section 6 introduces Algortihm 2b and the optimized versions of *pre** and *post**. Section 7 presents two variations of Algorithm 2b. Section 8 describes some technical aspects and implementation details. Section 9 discusses the results of the performance tests. Finally, Section 10 provides a summary of the previous sections.

2 Related Work

The analogous process of eliminating useless symbols and productions in context-free grammars is simpler than detecting useless trasitions in PDAs and well-documented [9, Chapter 7.1.1]. In order to characterize a symbol X as useful, two properties have to hold. First, X has to be *generating*, i.e., there has to exist a derivation $X \Rightarrow^* w$, for some terminal string w. Second, X has to be *reachable* from the start variable. All symbols for which these properties do not hold are useless and may be removed along with all productions that contain them, without altering the language defined by the grammar.

In contrast, there does not exist a standard method of minimization or reduction of PDAs. Caralp et al. presented a procedure for trimming visibly pushdown automata, a subclass of PDAs, that amounts to removing states that are not reachable from an initial state or not co-reachable from a final state [2].

Chervet and Walukiewicz discussed the minimization of only certain variants of deterministic visibly pushdown automata [3].

Bouajjani et al. provided the algorithm *pre** that takes as input a set of configurations of a pushdown system (PDS) and returns the set of all predecessor configurations [1]. Finkel et al. contributed an algorithm for computing all configurations that are reachable from the initial configuration of a PDS [7]. Esparza et al. presented a different approach for capturing these configurations through the algorithm *post** and also included efficient pseudocode implementations for both *post** and the previously mentioned *pre** [5], which are utilized in this paper.

Finally, to the best of our knowledge only Fokkink et al. have directly addressed the problem of detecting useless transitions in PDAs [8]. The approach they provided consists of two parts: (1) capturing the set of transitions that are not reachable from the initial configuration, and (2) capturing the set of transitions that do not lead to an accepting state. A transition that is found in either of these two sets is useless and can safely be removed from the PDA.

3 Preliminaries

In this paper, (1) ε denotes the empty string, (2) a denotes a symbol in Σ, (3) γ (sometimes carrying an index) denotes a symbol in Γ, (4) α, β denote symbols in $(\Gamma \cup \{\varepsilon\})$, (5) ρ, σ, τ denote strings in Γ^*, and (6) p, q, r, s denote states in Q.

Definition 1. *A pushdown automaton (PDA) P consists of a finite set of states Q, a finite input alphabet Σ, a finite stack alphabet Γ, a transition function $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \to 2^{Q \times \Gamma^*}$, an initial state q_0, an initial stack symbol Z_0, a finite set of accepting (final) states F, and is specified as*

$$P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$$

In addition, we use *generalized pushdown automata*, which are PDAs that allow for zero or more symbols to be popped from the stack in one transition instead of exactly one. Formally, they are defined as follows.

Definition 2. *A generalized pushdown automaton (GPDA) P consists of a finite set of states Q, a finite input alphabet Σ, a finite stack alphabet Γ, a transition function $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma^* \to 2^{Q \times \Gamma^*}$, an initial state q_0, an initial stack string Z_0, a finite set of accepting (final) states F, and is specified as*

$$P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$$

A *configuration* of a GPDA P is a tuple $\langle q, \sigma \rangle$, where $q \in Q$ and $\sigma \in \Gamma^*$. The language accepted by P, denoted by $L(P)$, consists of all strings in Σ^* that result in a run from the initial configuration $\langle q_0, Z_0 \rangle$ to a configuration $\langle r, \tau \rangle$, where $r \in F$. Such a run is called *accepting*.

A transition $(r, \tau) \in \delta(q, a, \sigma)$ (also written as $q \xrightarrow{a[\sigma/\tau]} r$) is *useless* if it is not included in any accepting run of P, for any input string $u \in L(P)$. Therefore,

when determining whether a transition is useless, the input symbol $a \in \Sigma$ is irrelevant, since at any state $q \in Q$ we may assume that any symbol is available. Disregarding input strings gives rise to the following definition.

Definition 3. *A pushdown system (PDS) $\mathcal{P}$ consists of a finite set of control locations P, a finite stack alphabet Γ, a transition function $\Delta : P \times \Gamma^* \to 2^{P \times \Gamma^*}$, and is specified as*

$$\mathcal{P} = (P, \Gamma, \Delta)$$

In this paper, we are interested in the usefulness of the transitions in a GPDA P rather than the language it accepts. Therefore, it suffices to examine the PDS $\mathcal{P}$ that is derived from P by using Q as its set of control locations, Γ as its stack alphabet, and adding transitions to Δ as follows.

$$\langle q, \sigma \rangle \hookrightarrow \langle r, \tau \rangle \in \Delta \iff q \xrightarrow{a[\sigma/\tau]} r \in \delta$$

for some $a \in (\Sigma \cup \{\varepsilon\})$. Note that $\langle q, \sigma \rangle \hookrightarrow \langle r, \tau \rangle \in \Delta$ can be written as $\langle r, \tau \rangle \in \Delta(q, \sigma)$ or $q \xrightarrow{\sigma/\tau} r$, and that there exists a one-to-many mapping from Δ to δ.

The algorithms discussed in this research examine the possible configurations of a PDS $\mathcal{P}$ in order to determine which transitions in Δ (and therefore in δ) are useful. In order to represent the possibly infinite state-space of $\mathcal{P}$, a special case of *nondeterministic finite automata* are introduced.

Definition 4. *Given a PDS $\mathcal{P} = (P, \Gamma, \Delta)$, we describe a set of configurations with the use of a $\mathcal{P}$-automaton $\mathcal{A}$, which has a finite set of states Q, uses Γ as its input alphabet, has a transition function $\delta : Q \times (\Gamma \cup \{\varepsilon\}) \to 2^Q$, uses P as its set of initial states, has a finite set of accepting (final) states F, and is specified as*

$$\mathcal{A} = (Q, \Gamma, \delta, P, F)$$

The $\mathcal{P}$-automaton $\mathcal{A}$ accepts a configuration $\langle q, \sigma \rangle$, where $q \in P$ and $\sigma \in \Gamma^*$, if and only if there exists a path from q to some state $r \in F$ marked σ, i.e., $q \xrightarrow{\sigma} r$. $Conf(\mathcal{A})$ denotes the set of configurations accepted by $\mathcal{A}$.

4 Description of Algorithm 1

The first algorithm to detect useless transitions in GPDAs that is considered in this paper is the one presented by Fokkink et al. Only a brief description of the algorithm is presented here. For details the reader is referred to [8]. Given a GPDA $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$, the set of useless transitions in P is calculated as follows.

Step 1 The input GPDA P is transformed in such a way that it contains a single final state that has no outgoing transitions and can only be reached with an empty stack. To this end, three fresh states are added to Q, one of which becomes the new single final state, a fresh stack symbol is added to Γ and a number of transitions that ensure the stack is emptied at the final state are added to δ. The transformed GPDA is called P_0.

Step 2 A *nondeterministic finite automaton* (NFA) N that expresses all stack configurations that are possible in each state of P_0 is constructed. This is achieved using the *forward* procedure, which populates N with states and transitions by examining the transitions in P_0. After N has been fully constructed, the procedure outputs a set U_1, which contains all transitions in P_0 that are unreachable. These transitions are removed from P_0 and the resulting GPDA is called P_1.

Step 3 The NFA N constructed in the previous step is utilized to determine which transitions in P_1 cannot possibly lead to the final state. These transitions are captured using the *backward* procedure in the set U_2.

Step 4 Finally, all transitions in $\delta \cap (U_1 \cup U_2)$ are reported as useless and may be removed from δ without altering the language that P defines.

Time complexity Algorithm 1 has a worst-case time complexity of $\mathcal{O}(Q^4T)$, where Q is the number of states and T the number of transitions in the input GPDA.

5 Description of Algorithm 2a

The second approach to detect useless transitions in GPDAs utilizes the model-checking procedures *pre** and *post**, presented by Esparza et al. [5]. Both procedures compute sets of configurations of PDSs. Specifically, given a PDS $\mathcal{P}$ and a set of configurations C, *pre** computes the set of configurations of $\mathcal{P}$ that can lead to a configuration in C. Furthermore, given a PDS $\mathcal{P}$ and a set of configurations C, *post** computes the set of configurations of $\mathcal{P}$ that are reachable from a configuration in C. Sets of configurations are expressed by means of $\mathcal{P}$-automata.

One important note is that *pre** and *post** require as input a PDS that is derived from a PDA instead of a GPDA, i.e. exactly one stack symbol has to be popped in each transition of the PDS. Moreover, due to the nature of the problem that Esparza et al. addressed, a PDS used as input for these procedures may push at most two symbols onto the stack in each transition. In this paper we are interested in GPDAs and therefore a normalization procedure is required before *pre** and *post** can be applied.

However, the normalization procedure used in this research does not handle transitions that pop no symbols from the stack. Therefore, the number of transitions in a normalized PDS is in $\mathcal{O}(ST)$, where S is the maximum stack string

length and T is the number of transitions in the original PDS. Note that we can normalize a transition $e = q \xrightarrow{\varepsilon/\tau} r$ by adding a fresh stack symbol b_0 to Γ to signify the bottom of the stack and replacing e by transitions $q \xrightarrow{\alpha/\tau\alpha} r$ for all $\alpha \in \Gamma$. However, incorporating this in a normalization procedure would result in the number of transitions being in $\mathcal{O}((|\Gamma| + S)T)$, where $|\Gamma|$ is the size of the stack alphabet, and would decrease the overall efficiency of Algorithm 2a. Thus, Algorithm 2a only accepts GPDAs that only contain transitions that pop one or more symbols from the stack as input. A brief description of the *pre** and *post** procedures follows.

5.1 Computing *pre**

The input is a PDS $\mathcal{P} = (P, \Gamma, \Delta)$ in normal form and a $\mathcal{P}$-automaton $\mathcal{A}$ that accepts a set of configurations C. We compute $\mathcal{A}_{pre*}$, the $\mathcal{P}$-automaton that expresses the set of all predecessor configurations of C, by adding new transitions to $\mathcal{A}$ according to the following saturation procedure.

> If $\langle p, \gamma\rangle \hookrightarrow \langle p', \tau\rangle \in \Delta$ and $p' \xrightarrow{\tau} q$ in the current $\mathcal{P}$-automaton, add a transition (p, γ, q).

A naive implementation of this abstract procedure would be to continuously apply this rule until no further changes are possible. The implementation of *pre** in this research is based on a more efficient pseudocode implementation provided by Esparza et al. [5].

5.2 Computing *post**

The input is a PDS $\mathcal{P} = (P, \Gamma, \Delta)$ in normal form and a $\mathcal{P}$-automaton $\mathcal{A}$ that accepts a set of configurations C. We compute $\mathcal{A}_{post*}$, the $\mathcal{P}$-automaton that expresses the set of all successor configurations of C, by adding new states and transitions to $\mathcal{A}$ according to the following procedure.

> – Add to $\mathcal{A}$ a new state r for each transition rule $r \in \Delta$ of the form $\langle p, \gamma\rangle \hookrightarrow \langle p', \gamma'\gamma''\rangle$ and a transition (p', γ', r).
> – Add new transitions to $\mathcal{A}$ according to the following saturation rules:
>
> If $\langle p, \gamma\rangle \hookrightarrow \langle p', \varepsilon\rangle \in \Delta$ and $p \xrightarrow{\gamma} q$ in the current $\mathcal{P}$-automaton, add a transition (p', ε, q).
> If $\langle p, \gamma\rangle \hookrightarrow \langle p', \gamma'\rangle \in \Delta$ and $p \xrightarrow{\gamma} q$ in the current $\mathcal{P}$-automaton, add a transition (p', γ', q).
> If $r = \langle p, \gamma\rangle \hookrightarrow \langle p', \gamma'\gamma''\rangle \in \Delta$ and $p \xrightarrow{\gamma} q$ in the current $\mathcal{P}$-automaton, add a transition (r, γ'', q).

Similarly to *pre**, a naive implementation of this abstract procedure for *post** would be to continuously apply the above rules until no further changes are possible. The implementation of *post** in this research is based on a more efficient pseudocode implementation provided by Esparza et al. [5].

5.3 Outline of Algorithm 2a

Step 1 As explained previously, since we are only interested in the behavior of the automaton, the input symbols of the GPDA $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ may be ignored. This gives rise to the PDS $\mathcal{P} = (Q, \Gamma, \Delta)$. Notice that there is a one-to-many mapping from Δ to δ. Therefore, in the following steps the transitions in Δ are examined. Thus, if a transition in Δ is found to be useless in $\mathcal{P}$, then the transitions in δ that it maps to are useless in P.

Step 2 The PDS $\mathcal{P}$ is normalized according to the following procedure. Every transition of the form $q \xrightarrow{\gamma_1\gamma_2\cdots\gamma_n/\tau} r$ is substituted by extension transitions $q \xrightarrow{\gamma_1/\varepsilon} s_1, s_1 \xrightarrow{\gamma_2/\varepsilon} s_2, \ldots, s_{n-1} \xrightarrow{\gamma_n/\tau} r$, where $s_1, s_2, \ldots, s_{n-1}$ are fresh states. Notice that $q \xrightarrow{\gamma_1\gamma_2\cdots\gamma_n/\tau} r$ is useless if and only if $q \xrightarrow{\gamma_1/\varepsilon} s_1$ is useless. Therefore, in the following steps we only need to test the first extension transition for usefulness. Next, every transition of the form $q \xrightarrow{\gamma/\gamma_1\gamma_2\cdots\gamma_n} r$, with $n > 2$ is substituted by $q \xrightarrow{\gamma/\gamma_{n-1}\gamma_n} s_1, s_1 \xrightarrow{\gamma_{n-1}/\gamma_{n-2}\gamma_{n-1}} s_2, \cdots, s_{n-2} \xrightarrow{\gamma_2/\gamma_1\gamma_2} r$, where $s_1, s_2, \ldots, s_{n-2}$ are fresh states. Similarly, notice that the original transition is useless if and only if the first extension transition is useless, and thus only the first extension transition is examined in the following steps. The resulting PDS is called $\mathcal{P}_1$.

Step 3 As stated previously, we do not need to examine each transition in $\mathcal{P}_1$, only those that map to a transition in P. This mapping has been constructed in two stages in steps 1 and 2. Intuitively, a transition $\theta = q \xrightarrow{\gamma/\tau} r$ of $\mathcal{P}_1$ is useful if and only if (1) it is applicable, and (2) it leads to an accepting state. The first criterion holds if and only if $\langle q, \gamma\rho\rangle$, where ρ is an arbitrary stack string, is a successor of the initial configuration $\langle q_0, Z_0\rangle$. However, the second criterion does not necessarily hold if $\langle q, \gamma\rho\rangle$ is a predecessor of an arbitrary accepting configuration $\langle q_f, \sigma\rangle$. Notice that θ may not lead to an accepting state, but if a second transition $\theta' = q \xrightarrow{\gamma/\tau'} r'$ does, then $\langle q, \gamma\rho\rangle$ is in fact a predecessor of some accepting configuration. Hence, we need to isolate the effects of θ on the PDS in order to determine its usefulness. To this end, we carry out the following procedure.

For each transition θ, we introduce a fresh state q_θ and replace θ by two transitions $q \xrightarrow{\gamma/\gamma} q_\theta$ and $q_\theta \xrightarrow{\gamma/\tau} r$. Then, we pass the resulting PDS $\mathcal{P}_\theta$ as input to *pre** along with a $\mathcal{P}$-automaton that expresses all configurations $F \times \Gamma^*$. We also pass $\mathcal{P}_\theta$ as input to *post** along with a $\mathcal{P}$-automaton that expresses only

the initial configuration $\langle q_0, Z_0 \rangle$. Both functions return $\mathcal{P}$-automata, $\mathcal{A}_{pre*}$ and $\mathcal{A}_{post*}$. We take q_θ as the starting state in both automata and compute their intersection $\mathcal{A}_\theta$. Finally, the transition θ is useless if and only if no accepting state is reachable from the starting state in $\mathcal{A}_\theta$. In that case, the one or more transitions of the input GPDA P that correspond to θ are reported as useless.

Indeed, the introduction of q_θ and transitions $q \xrightarrow{\gamma/\gamma} q_\theta$ and $q_\theta \xrightarrow{\gamma/\tau} r$ is necessary as it ensures that if $\langle q_\theta, \gamma\rho \rangle$ is found to be a predecessor of an arbitrary accepting configuration, then it is because of θ and not some other transition.

Time complexity If T is the number of transitions in $\mathcal{P}$, then the number of transitions in $\mathcal{P}_1$ is in $\mathcal{O}(ST)$, where S is the maximum string length that occurs in all transitions. Computing $post*(C)$ can be done in $\mathcal{O}(S^3T^3)$ time and computing $pre*(C)$ can be done in $\mathcal{O}(S^2T^3)$ time. Both of these operations have to be done T times (once for each transition in $\mathcal{P}$), so the overall worst-case time complexity for this algorithm is $\mathcal{O}(S^3T^4)$.

5.4 Drawbacks

One drawback of the current algorithm, compared to Algorithm 1, is that it does not allow zero symbols to be popped in a transition of the input GPDA, i.e. only a subclass of all possible GPDAs can be used as input. It is also interesting to note that it provides less information about the useless transitions than the latter.

Moreover, Algorithm 1 makes a distinction between transitions that are never applicable (transitions in U_1) and transitions that are not used in an accepting run of the GPDA (transitions in U_2). This extra piece of information, while not a natural by-product of Algorithm 2a, may be computed based on the output of $pre*$ and $post*$ for each transition $\theta = q \xrightarrow{\gamma/\tau}$. Specifically, if $\langle q_\theta, \gamma\rho \rangle$, where ρ is an arbitrary stack string, is not a configuration in $\mathcal{A}_{post*}$, then θ is never applicable. Similarly, if $\langle q_\theta, \gamma\rho \rangle$ is not a configuration in $\mathcal{A}_{pre*}$, then θ is not used in any accepting run. However, this functionality has not been implemented in the current research.

6 Description of Algorithm 2b

The algorithm discussed in the previous section has two drawbacks. Specifically, it requires the intermediate PDS to be normalized and, more importantly, it cannot handle transitions that pop no symbols from the stack without a decrease in efficiency. To resolve these issues and retain the same overall worst-case time complexity of $\mathcal{O}(S^3T^4)$ that Algorithm 2a exhibits, $pre*$ and $post*$ have been redesigned to work with all PDSs, not only those in normal form. Intuitively, a transition in the input PDS is normalized on-the-fly as needed during the execution of these procedures. As a result, in Algorithm 2b, step 1 remains

unchanged, step 2 is dropped since no normalization is required and step 3 is altered slightly as follows.

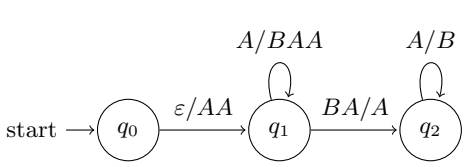
For each transition $\theta = q \xrightarrow{\sigma/\tau} r$ of $\mathcal{P}$, we introduce a fresh state q_θ and replace θ by two transitions $q \xrightarrow{\varepsilon/\varepsilon} q_\theta$ and $q_\theta \xrightarrow{\sigma/\tau} r$. Then, we pass the resulting PDS $\mathcal{P}_\theta$ as input to $pre*$ along with a $\mathcal{P}$-automaton that expresses all configurations $F \times \Gamma^*$. We also pass $\mathcal{P}_\theta$ as input to $post^*$ along with a $\mathcal{P}$-automaton that expresses only the initial configuration $\langle q_0, Z_0 \rangle$. Both functions return $\mathcal{P}$-automata, $\mathcal{A}_{pre*}$ and $\mathcal{A}_{post*}$. We take q_θ as the starting state in both automata and compute their intersection $\mathcal{A}_\theta$. Finally, the transition θ is useless if and only if no accepting state is reachable from the starting state in $\mathcal{A}_\theta$. The extended versions of pre^* and $post^*$ are presented below.

6.1 Generalized *pre**

The input is a PDS $\mathcal{P} = (P, \Gamma, \Delta)$ and a $\mathcal{P}$-automaton $\mathcal{A}$ that describes a set of configurations C. We compute $\mathcal{A}_{pre*}$, the $\mathcal{P}$-automaton that expresses the set of all predecessor configurations of C, by adding new states and transitions to $\mathcal{A}$ according to the following saturation procedure.

> If $\langle p, \gamma \rangle \hookrightarrow \langle p', \tau \rangle \in \Delta$ and $p' \xrightarrow{\tau} q$ in the current $\mathcal{P}$-automaton, add a transition (p, γ, q).
> If $\langle p, \varepsilon \rangle \hookrightarrow \langle p', \tau \rangle \in \Delta$ and $p' \xrightarrow{\tau} q$ in the current $\mathcal{P}$-automaton, add a transition (p, ε, q).
> If $\langle p, \gamma_1 \gamma_2 \cdots \gamma_n \rangle \hookrightarrow \langle p', \tau \rangle \in \Delta$ with $n \geq 2$ and $p' \xrightarrow{\tau} q$ in the current $\mathcal{P}$-automaton, add transitions $(p, \gamma_1, s_1), (s_1, \gamma_2, s_2), \ldots, (s_{n-1}, \gamma_n, q)$, where $s_1, s_2, \ldots, s_{n-1}$ are fresh states.

This procedure is best illustrated with an example. We use the PDS $\mathcal{P}_1 = (P, \Gamma, \Delta)$ with $P = \{q_0, q_1, q_2\}$ (Figure 1) and the $\mathcal{P}$-automaton $\mathcal{A}_1$ that defines the set $C_1 = \{\langle q_2, BAA \rangle\}$ (Figure 2, left).



$$\Delta = \{\langle q_0, \varepsilon \rangle \hookrightarrow \langle q_1, AA \rangle,$$
$$\langle q_1, A \rangle \hookrightarrow \langle q_1, BAA \rangle,$$
$$\langle q_1, BA \rangle \hookrightarrow \langle q_2, A \rangle,$$
$$\langle q_2, A \rangle \hookrightarrow \langle q_2, B \rangle\}$$

Fig. 1. The PDS $\mathcal{P}_1$ and its set of transitions Δ

Due to the first rule and transitions $\langle q_2, A \rangle \hookrightarrow \langle q_2, B \rangle$ and $q_2 \xrightarrow{B} r_1$, we add transition $q_2 \xrightarrow{A} r_1$ to $\mathcal{A}_1$. Using the third rule in conjunction with transi-

tions $\langle q_1, BA\rangle \hookrightarrow \langle q_2, A\rangle$ and $q_2 \xrightarrow{A} r_1$, we add a fresh state s_1 and transitions $q_1 \xrightarrow{B} s_1$ and $s_1 \xrightarrow{A} r_1$ to $\mathcal{A}_1$. Next, we apply the first rule with transition $\langle q_1, A\rangle \hookrightarrow \langle q_1, BAA\rangle$ and path $q_1 \xrightarrow{BAA} r_2$ in order to add transition $q_1 \xrightarrow{A} r_2$. Finally, because of the second rule and transition $\langle q_0, \varepsilon\rangle \hookrightarrow \langle q_1, AA\rangle$ and path $q_1 \xrightarrow{AA} q_f$, we add transition $q_0 \xrightarrow{\varepsilon} q_f$. The resulting $\mathcal{P}$-automaton is called $\mathcal{A}^1_{pre*}$ (Figure 2, right) and it expresses all configurations of $\mathcal{P}_1$ that are predecessors of configuration $\langle q_2, BAA\rangle$.

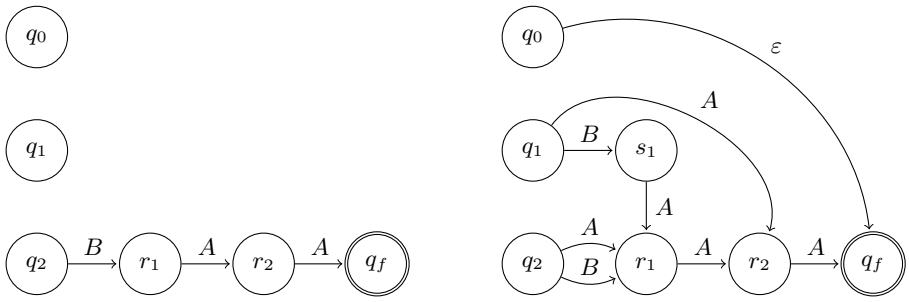


Fig. 2. The $\mathcal{P}$-automata $\mathcal{A}_1$ (left) and $\mathcal{A}^1_{pre*}$ (right)

A naive implementation of *pre** would be to continuously apply the three rules until no further changes can be made to the $\mathcal{P}$-automaton. However, the pseudocode implementation discussed in this paper is based on a more efficient implementation of the original *pre** algorithm presented by Esparza et al. [5], and maintains the same worst-case time complexity. The increased efficiency in this implementation is due to lines 16-23 of the pseudocode (discussed below), which eliminate the need for backward searching. Note that $\alpha, \beta \in (\Gamma \cup \{\varepsilon\})$ in this algorithm.

As in the original *pre** algorithm, we use sets *rel* and *trans* to hold transitions that belong to $\mathcal{A}_{pre*}$; *rel* contains the transitions that have already been examined, while *trans* contains the transitions that still need to be examined. In contrast to the original algorithm, transitions may be examined multiple times. A brief explanation of each part of the algorithm follows.

In line 1 of the algorithm, we initialize the required sets. As mentioned before, *rel* holds transitions that have been examined and thus is initially empty, while *trans* holds transitions that have to be examined and is therefore initialized with all the transitions in δ. Moreover, the algorithm works by evaluating PDS transitions, but it also creates new PDS transitions during its execution. Hence, we use a copy of Δ called Δ' so that we can add new PDS transitions without changing the input PDS $\mathcal{P}$. Similarly, the output $\mathcal{A}_{pre*}$ will contain all the states of the input $\mathcal{A}$, but we may also need to add fresh states. To this end, we make a copy of Q called Q' so that we can introduce new states without altering $\mathcal{A}$.

Generalized *pre

Input: a PDS $\mathcal{P} = (P, \Gamma, \Delta)$
a $\mathcal{P}$-automaton $\mathcal{A} = (Q, \Gamma, \delta, P, F)$
Output: the $\mathcal{P}$-automaton $\mathcal{A}_{pre^*}$

1: $rel \leftarrow \emptyset$; $trans \leftarrow \delta$; $\Delta' \leftarrow \Delta$; $Q' \leftarrow Q$;
2: **for all** $\langle p, \alpha \rangle \hookrightarrow \langle p', \varepsilon \rangle \in \Delta'$ **do**
3: $\quad trans \leftarrow trans \cup \{(p, \alpha, p')\}$;
4: **for all** $\langle p, \gamma_1\gamma_2 \cdots \gamma_n \rangle \hookrightarrow \langle p', \varepsilon \rangle \in \Delta'$ with $n \geq 2$ **do**
5: $\quad trans \leftarrow trans \cup \{(p, \gamma_1, s_1), (s_1, \gamma_2, s_2), \ldots, (s_{n-1}, \gamma_n, p')\}$;
6: $\quad Q' \leftarrow Q' \cup \{s_1, s_2, \ldots, s_{n-1}\}$;
7: **while** $trans \neq \emptyset$ **do**
8: $\quad$ pop $t = (q, \beta, q')$ from $trans$;
9: $\quad$ **if** $t \notin rel$ **then**
10: $\quad\quad rel \leftarrow rel \cup \{t\}$;
11: $\quad\quad$ **for all** $\langle p, \alpha \rangle \hookrightarrow \langle q, \beta \rangle \in \Delta'$ **do**
12: $\quad\quad\quad trans \leftarrow trans \cup \{(p, \alpha, q')\}$;
13: $\quad\quad$ **for all** $\langle p, \gamma_1\gamma_2 \cdots \gamma_n \rangle \hookrightarrow \langle q, \beta \rangle \in \Delta'$ with $n \geq 2$ **do**
14: $\quad\quad\quad trans \leftarrow trans \cup \{(p, \gamma_1, s_1), (s_1, \gamma_2, s_2), \ldots, (s_{n-1}, \gamma_n, q')\}$;
15: $\quad\quad\quad Q' \leftarrow Q' \cup \{s_1, s_2, \ldots, s_{n-1}\}$;
16: $\quad\quad$ **for all** $\langle p, \sigma \rangle \hookrightarrow \langle q, \beta\gamma\tau \rangle \in \Delta'$ with $|\beta\gamma\tau| \geq 2$ **do**
17: $\quad\quad\quad e \leftarrow \langle p, \sigma \rangle \hookrightarrow \langle q', \gamma\tau \rangle$;
18: $\quad\quad\quad$ **if** $e \notin \Delta'$ **then**
19: $\quad\quad\quad\quad \Delta' \leftarrow \Delta' \cup \{e\}$;
20: $\quad\quad\quad\quad$ **for all** $u = (q', \varepsilon, q'') \in rel$ **do**
21: $\quad\quad\quad\quad\quad rel \leftarrow rel \setminus \{u\}$; $trans \leftarrow trans \cup \{u\}$;
22: $\quad\quad\quad\quad$ **for all** $u = (q', \gamma, q'') \in rel$ **do**
23: $\quad\quad\quad\quad\quad rel \leftarrow rel \setminus \{u\}$; $trans \leftarrow trans \cup \{u\}$;
24: **return** (Q', Γ, rel, P, F);

In lines 2-6, we populate $\mathcal{A}_{pre^*}$ with states and transitions that are due to PDS transitions that push no symbols onto the stack. We operate under the assumption that $\langle p', \rho \rangle$ where ρ is an arbitrary stack string is a configuration in $pre^*(C)$, that is, there exists a path $p' \xrightarrow{\rho} q_f$ where q_f is an arbitrary final state. If there exists a transition $\langle p, \sigma \rangle \hookrightarrow \langle p', \varepsilon \rangle \in \Delta'$, then $\langle p, \sigma\rho \rangle$ is a predecessor of $\langle p', \rho \rangle$. Therefore, we add one or more transitions and any necessary states to construct a path $p \xrightarrow{\sigma} p'$. This forms a path $p \xrightarrow{\sigma\rho} q_f$, and thus $\langle p', \sigma\rho \rangle$ is now a configuration in $pre^*(C)$. Note that if the initial assumption does not hold, i.e. $\langle p', \rho \rangle \notin pre^*(C)$, then the path we constructed does not change $pre^*(C)$.

Within the while-loop, we examine each transition $t = (q, \beta, q') \in trans$. Similarly, we operate under the assumption that $\langle q', \rho \rangle \in pre^*(C)$. Intuitively, t tells us that $\langle q, \beta\rho \rangle$ is a predecessor of $\langle q', \rho \rangle$. In lines 11-15, we evaluate PDS transitions of the form $\langle p, \sigma \rangle \hookrightarrow \langle q, \beta \rangle$. If such a transition exists in Δ', then $\langle p, \sigma\rho \rangle$ is a predecessor of $\langle q, \beta\rho \rangle$ and consequently it is also a predecessor of $\langle q', \rho \rangle$. Thus, we create a path $p \xrightarrow{\sigma} q'$, which adds $\langle p, \sigma\rho \rangle$ to $pre^*(C)$.

In lines 16-23 of the algorithm, we deal with PDS transitions that push two or more symbols onto the stack. We work under the assumption that $\langle r, \rho \rangle \in pre^*(C)$ where r is an arbitrary state. We look out for a series of transitions $t_1, t_2, \ldots, t_m$ that forms a path $q \xrightarrow{\beta\gamma\tau} r$ with $|\beta\gamma\tau| \geq 2$. If such a path exists, then $\langle q, \beta\gamma\tau\rho \rangle$ is a predecessor of $\langle r, \rho \rangle$. If there exists a PDS transition of the form $\langle p, \sigma \rangle \hookrightarrow \langle q, \beta\gamma\tau \rangle$, then $\langle p, \sigma\rho \rangle$ is a predecessor of $\langle q, \beta\gamma\tau\rho \rangle$. Thus, the former is also a predecessor of $\langle r, \rho \rangle$. Hence, we want to add a path $p \xrightarrow{\sigma} r$ to the $\mathcal{P}$-automaton. During the execution of the algorithm, we do not know the order in which we examine the transitions $t_1, t_2, \ldots, t_m$. If we perform a backward search every time we see a transition t_k, where $2 \leq k \leq m$, to check the existence of transitions $t_1, t_2, \ldots, t_{k-1}$, then we might waste time with many negative checks. However, we know that if we examine $t_1 = (q, \beta, q')$, then all subsequent transitions $t_2 = (q', \gamma, q'')$ (or $t_2 = (q', \varepsilon, q'')$) may eventually be part of a path marked $\beta\gamma\tau$. Therefore, we introduce a new PDS transition (if one does not already exist) $\langle p, \sigma \rangle \hookrightarrow \langle q', \gamma\tau \rangle$ to take care of such cases. Moreover, if we indeed add this new PDS transition we check whether we have already examined t_2. If that is the case, then t_2 is moved from *rel* to *trans* to be examined again.

Finally, the output $\mathcal{P}$-automaton $\mathcal{A}_{pre^*}$ consists of the stack alphabet Γ, the set of initial states P and the set of final states F of $\mathcal{A}$. In addition, it consists of the set of states Q', which is a superset of Q, and of the set of transitions *rel*, which is a superset of δ.

To illustrate how this implementation works, we use the PDS $\mathcal{P}_2$ (Figure 3) and the $\mathcal{P}$-automaton $\mathcal{A}_2$ (Figure 4, left). $\mathcal{P}_2$ contains a single transition, and $\mathcal{A}_2$ expresses all configurations $\langle q_1, AB^* \rangle$. Clearly, the desired output of pre^* is a $\mathcal{P}$-automaton that represents all configurations $\langle q_1, AB^* \rangle$ and $\langle q_0, CB^* \rangle$.

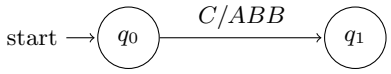


Fig. 3. The PDS $\mathcal{P}_2$

Consider the execution of pre^* with this input. Initially, $trans = \{(q_f, B, q_f), (q_1, A, q_f)\}$, $rel = \emptyset$, $\Delta' = \{\langle q_0, C \rangle \hookrightarrow \langle q_1, ABB \rangle\}$, and $Q' = \{q_0, q_1, q_f\}$ (line 1). Since Δ' does not contain transitions that push no symbols onto the stack, lines 2-6 are skipped. In the first iteration of the while-loop, when (q_1, A, q_f) is popped from *trans* (line 8) and found not to be contained in *rel* (line 9), it is added to *rel* (line 10). Since Δ' does not contain transitions that push at most one symbol onto the stack, lines 11-15 are skipped. Transition $\langle q_0, C \rangle \hookrightarrow \langle q_1, ABB \rangle$ is evaluated in line 16. This leads to the construction of transition $\langle q_0, C \rangle \hookrightarrow \langle q_f, BB \rangle$ (line 17). This transition is not contained in Δ', so it is added (lines 18-19). Since there exist no transitions that originate from q_f in *rel*, lines 20-23 are skipped. In the next iteration, (q_f, B, q_f) is popped from *trans* (line 8), found not to be in *rel* (line 9), and added to *rel* (line 10). As before, lines 11-15 are skipped. Transition

$\langle q_0, C\rangle \hookrightarrow \langle q_f, BB\rangle$ is examined at line 16, which leads to the construction of $\langle q_0, C\rangle \hookrightarrow \langle q_f, B\rangle$ (line 17). The latter is found not to be in Δ' (line 18) and is subsequently added (line 19). Since there is no transition of the form (q_f, ε, s) where s is an arbitrary state, lines 20-21 are skipped. The transition (q_f, B, q_f) is found to be contained in *rel* (line 22), and is moved from *rel* to *trans* (line 23). In the following iteration, (q_f, B, q_f) is popped from *trans* again (line 8), found not to be in *rel* (line 9), and added to *rel* (line 10). Transition $\langle q_0, C\rangle \hookrightarrow \langle q_f, B\rangle$ is evaluated and leads to the addition of (q_0, C, q_f) to *trans* (lines 11-12). Since no transition in Δ' matches the clause in line 13, lines 13-15 are skipped. Transition $\langle q_0, C\rangle \hookrightarrow \langle q_f, BB\rangle$ is evaluated at line 16, which leads to the construction of $\langle q_0, C\rangle \hookrightarrow \langle q_f, B\rangle$ (line 17). The latter is found to be contained in Δ' and thus lines 19-23 are skipped. In the final iteration, (q_0, C, q_f) is popped from *trans* (line 8), found not to be contained in *rel* (line 9), and added to *rel* (line 10). Since no transition in Δ' leads to a configuration $\langle q_0, \rho\rangle$ where ρ is an arbitrary stack string, lines 11-23 are skipped. Finally, the algorithm terminates by returning the $\mathcal{P}$-automaton $\mathcal{A}^2_{pre^*} = (Q', \Gamma, rel, P, F)$ (Figure 4, right).

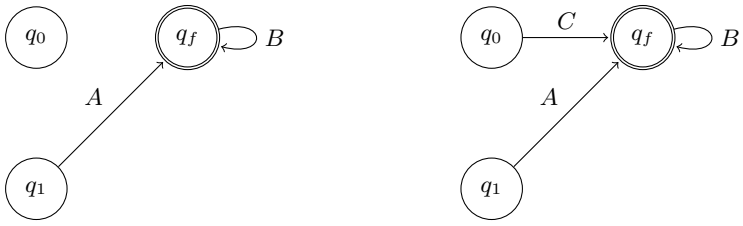


Fig. 4. The $\mathcal{P}$-automata $\mathcal{A}_2$ (left) and $\mathcal{A}^2_{pre^*}$ (right)

The above example clearly demonstrates that considering transitions multiple times is necessary when dealing with arbitrary stack string lengths. In fact, there exists an upper bound on the times that each transition may be examined, namely S, the maximum stack string length in $\mathcal{P}$.

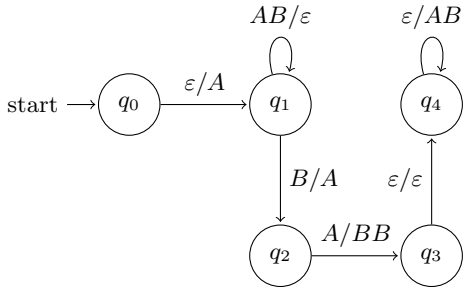
6.2 Generalized *post**

The input is a PDS $\mathcal{P} = (P, \Gamma, \Delta)$ and a $\mathcal{P}$-automaton $\mathcal{A}$ that describes a set of configurations C. We compute $\mathcal{A}_{post^*}$, the $\mathcal{P}$-automaton that expresses the set of all successor configurations of C, by adding new states and transitions to $\mathcal{A}$ according to the following procedure.

– Add to $\mathcal{A}$ new states $s_1, s_2, \ldots, s_{n-2}, r$ for each transition rule $r \in \Delta$ of the form $\langle p, \sigma\rangle \hookrightarrow \langle p', \gamma_1\gamma_2\cdots\gamma_n\rangle$ with $n \geq 2$, and transitions $(p', \gamma_1, s_1), (s_1, \gamma_2, s_2), \ldots, (s_{n-2}, \gamma_{n-1}, r)$.
– Add new transitions to $\mathcal{A}$ according to the following saturation rules:

If $\langle p, \sigma\rangle \hookrightarrow \langle p', \varepsilon\rangle \in \Delta$ and $p \xrightarrow{\sigma} q$ in the current $\mathcal{P}$-automaton, add a transition (p', ε, q).
If $\langle p, \sigma\rangle \hookrightarrow \langle p', \gamma'\rangle \in \Delta$ and $p \xrightarrow{\sigma} q$ in the current $\mathcal{P}$-automaton, add a transition (p', γ', q).
If $r = \langle p, \sigma\rangle \hookrightarrow \langle p', \gamma_1\gamma_2 \cdots \gamma_n\rangle \in \Delta$ with $n \geq 2$ and $p \xrightarrow{\sigma} q$ in the current $\mathcal{P}$-automaton, add a transition (r, γ_n, q).
If $\langle p, \varepsilon\rangle \hookrightarrow \langle p', \varepsilon\rangle \in \Delta$ and $p \xrightarrow{\gamma} q$ for some $\gamma \in \Gamma$ in the current $\mathcal{P}$-automaton, add a transition (p', γ, q).
If $\langle p, \varepsilon\rangle \hookrightarrow \langle p', \gamma'\rangle \in \Delta$ and $p \xrightarrow{\gamma} q$ for some $\gamma \in \Gamma$ in the current $\mathcal{P}$-automaton, add transitions $(p', \gamma', s), (s, \gamma, q)$ where s is a fresh state.
If $r = \langle p, \varepsilon\rangle \hookrightarrow \langle p', \gamma_1\gamma_2 \cdots \gamma_n\rangle \in \Delta$ with $n \geq 2$ and $p \xrightarrow{\gamma} q$ for some $\gamma \in \Gamma$ in the current $\mathcal{P}$-automaton, add transitions $(r, \gamma_n, s), (s, \gamma, q)$, where s is a fresh state.

We illustrate this procedure with an example. We use the PDS $\mathcal{P}_3 = (P, \Gamma, \Delta)$ with $P = \{q_0, q_1, q_2, q_3, q_4\}$ (Figure 5) and the $\mathcal{P}$-automaton $\mathcal{A}_3$ that accepts the set $C_3 = \{\langle q_0, BB\rangle\}$ (Figure 6, left).



$$\begin{aligned}\Delta = \{&\langle q_0, \varepsilon\rangle \hookrightarrow \langle q_1, A\rangle,\\ &\langle q_1, AB\rangle \hookrightarrow \langle q_1, \varepsilon\rangle,\\ &\langle q_1, B\rangle \hookrightarrow \langle q_2, A\rangle,\\ &\langle q_2, A\rangle \hookrightarrow \langle q_3, BB\rangle,\\ &\langle q_3, \varepsilon\rangle \hookrightarrow \langle q_4, \varepsilon\rangle,\\ &\langle q_4, \varepsilon\rangle \hookrightarrow \langle q_4, AB\rangle\}\end{aligned}$$

Fig. 5. The PDS $\mathcal{P}_3$ and its set of transitions Δ

In the first step of the procedure, transition $r_1 = \langle q_2, A\rangle \hookrightarrow \langle q_3, BB\rangle$ leads to the addition of state r_1 and transition $q_3 \xrightarrow{B} r_1$. Because of transition $r_2 = \langle q_4, \varepsilon\rangle \hookrightarrow \langle q_4, AB\rangle$, we add state r_2 and transition $q_4 \xrightarrow{A} r_2$. In the second step, the saturation rules are applied as follows. Due to the fifth rule and transitions $\langle q_0, \varepsilon\rangle \hookrightarrow \langle q_1, A\rangle$ and $q_0 \xrightarrow{B} p_1$, we add a fresh state s_1 and transitions $q_1 \xrightarrow{A} s_1$ and $s_1 \xrightarrow{B} p_1$. Next, we apply the fist rule with transition $\langle q_1, AB\rangle \hookrightarrow \langle q_1, \varepsilon\rangle$ and path $q_1 \xrightarrow{AB} p_1$ so as to add transition $q_1 \xrightarrow{\varepsilon} p_1$. Using the second rule, transition $\langle q_1, B\rangle \hookrightarrow \langle q_2, A\rangle$, and path $q_1 \xrightarrow{B} q_f$, we add transition $q_2 \xrightarrow{A} q_f$. Because of

the third rule and transitions r_1 and $q_2 \xrightarrow{A} q_f$, we introduce transition $r_1 \xrightarrow{B} q_f$. Due to the fourth rule and transitions $\langle q_3, \varepsilon \rangle \hookrightarrow \langle q_4, \varepsilon \rangle$ and $q_3 \xrightarrow{B} r_1$, we add transition $q_4 \xrightarrow{B} r_1$. The sixth rule and transitions r_2 and $q_4 \xrightarrow{B} r_1$ lead to the addition of a fresh state s_2 and transitions $r_2 \xrightarrow{B} s_2$ and $s_2 \xrightarrow{B} r_1$. Finally, we apply the sixth rule with transitions r_2 and $q_4 \xrightarrow{A} r_2$, in order to add a fresh state s_3 and transitions $r_2 \xrightarrow{B} s_3$ and $s_3 \xrightarrow{A} r_2$. The resulting $\mathcal{P}$-automaton is called $\mathcal{A}^3_{post*}$ (Figure 6, right) and it expresses all configurations of $\mathcal{P}_3$ that are reachable from the configuration $\langle q_0, BB \rangle$.

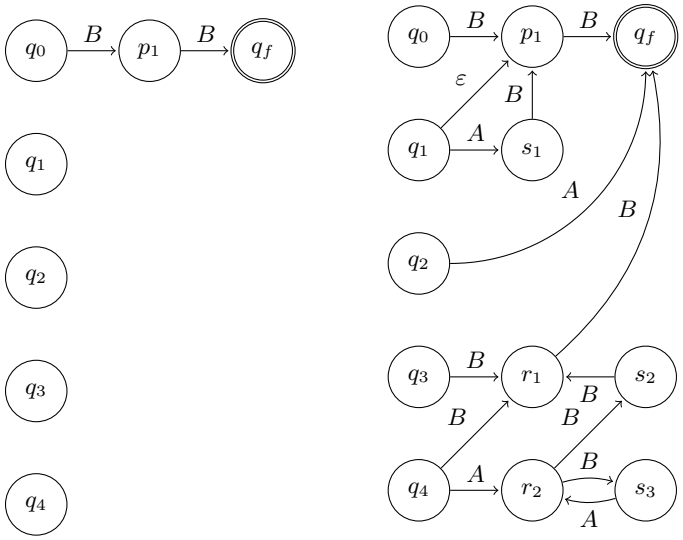


Fig. 6. The $\mathcal{P}$-automata $\mathcal{A}_3$ (left) and $\mathcal{A}^3_{post*}$ (right)

As was the case with *pre**, a naive implementation of *post** would be to continuously apply the saturation rules until no further changes to the $\mathcal{P}$-automaton are possible. However, the pseudocode implementation presented in this research is based on a more efficient implementation of the original *post** algorithm contributed by Esparza et al. [5], and exhibits the same worst-case time complexity. As before, this implementation is more efficient due to lines 25-34 of the pseudocode (discussed below), which help avoid backward searches. Note that $\alpha, \beta \in (\Gamma \cup \{\varepsilon\})$ in this algorithm.

Similarly to the pseudocode implementation for *pre**, the pseudocode implementation for *post** uses sets *trans* and *rel* to hold transitions of the $\mathcal{P}$-automaton, set Δ' to hold PDS transitions, and set Q' to hold states of the output $\mathcal{A}_{post*}$. Furthermore, the upper bound for the number of times each transition may be examined is S. A brief description of each block of the algorithm follows.

Generalized *post**

Input: a PDS $\mathcal{P} = (P, \Gamma, \Delta)$
a $\mathcal{P}$-automaton $\mathcal{A} = (Q, \Gamma, \delta, P, F)$
Output: the $\mathcal{P}$-automaton $\mathcal{A}_{post*}$

1: $trans \leftarrow \delta \cap (P \times (\Gamma \cup \{\varepsilon\}) \times Q)$;
2: $rel \leftarrow \delta \setminus trans$;
3: $Q' \leftarrow Q$; $\Delta' \leftarrow \Delta$; $map \leftarrow \emptyset$;
4: **for all** $r = \langle p, \sigma \rangle \hookrightarrow \langle p', \gamma_1 \gamma_2 \cdots \gamma_n \rangle \in \Delta'$ with $n \geq 2$ **do**
5: $\quad trans \leftarrow trans \cup \{(p', \gamma_1, s_1), (s_1, \gamma_2, s_2), \ldots, (s_{n-2}, \gamma_{n-1}, s_{n-1})\}$;
6: $\quad Q' \leftarrow Q' \cup \{s_1, s_2, \ldots, s_{n-2}, s_{n-1}\}$;
7: $\quad map[r] \leftarrow s_{n-1}$;
8: **while** $trans \neq \emptyset$ **do**
9: $\quad$ pop $t = (p, \beta, q)$ from $trans$
10: $\quad$ **if** $t \notin rel$ **then**
11: $\quad\quad rel \leftarrow rel \cup \{t\}$
12: $\quad\quad$ **if** $\beta \neq \varepsilon$ **then**
13: $\quad\quad\quad$ **for all** $\langle p, \varepsilon \rangle \hookrightarrow \langle p', \varepsilon \rangle \in \Delta'$ **do**
14: $\quad\quad\quad\quad trans \leftarrow trans \cup \{(p', \beta, q)\}$;
15: $\quad\quad\quad$ **for all** $\langle p, \varepsilon \rangle \hookrightarrow \langle p', \gamma_1 \rangle \in \Delta'$ **do**
16: $\quad\quad\quad\quad trans \leftarrow trans \cup \{(p', \gamma_1, s), (s, \beta, q)\}$;
17: $\quad\quad\quad\quad Q' \leftarrow Q' \cup \{s\}$;
18: $\quad\quad\quad$ **for all** $r = \langle p, \varepsilon \rangle \hookrightarrow \langle p', \gamma_1 \gamma_2 \cdots \gamma_n \rangle \in \Delta'$ with $n \geq 2$ **do**
19: $\quad\quad\quad\quad trans \leftarrow trans \cup \{(map[r], \gamma_n, s), (s, \beta, q)\}$;
20: $\quad\quad\quad\quad Q' \leftarrow Q' \cup \{s\}$;
21: $\quad\quad$ **for all** $\langle p, \beta \rangle \hookrightarrow \langle p', \alpha \rangle \in \Delta'$ **do**
22: $\quad\quad\quad trans \leftarrow trans \cup \{(p', \alpha, q)\}$;
23: $\quad\quad$ **for all** $r = \langle p, \beta \rangle \hookrightarrow \langle p', \gamma_1 \gamma_2 \cdots \gamma_n \rangle \in \Delta'$ with $n \geq 2$ **do**
24: $\quad\quad\quad trans \leftarrow trans \cup \{(map[r], \gamma_n, q)\}$;
25: $\quad\quad$ **for all** $r = \langle p, \beta\gamma\tau \rangle \hookrightarrow \langle p', \sigma \rangle \in \Delta'$ with $|\beta\gamma\tau| \geq 2$ **do**
26: $\quad\quad\quad e \leftarrow \langle q, \gamma\tau \rangle \hookrightarrow \langle p', \sigma \rangle$;
27: $\quad\quad\quad$ **if** $e \notin \Delta'$ **then**
28: $\quad\quad\quad\quad \Delta' \leftarrow \Delta' \cup \{e\}$;
29: $\quad\quad\quad\quad$ **if** $r \in keys(map)$ **then**
30: $\quad\quad\quad\quad\quad map[e] \leftarrow map[r]$;
31: $\quad\quad\quad\quad$ **for all** $u = (q, \varepsilon, q') \in rel$ **do**
32: $\quad\quad\quad\quad\quad rel \leftarrow rel \setminus \{u\}$; $trans \leftarrow trans \cup \{u\}$;
33: $\quad\quad\quad\quad$ **for all** $u = (q, \gamma, q') \in rel$ **do**
34: $\quad\quad\quad\quad\quad rel \leftarrow rel \setminus \{u\}$; $trans \leftarrow trans \cup \{u\}$;
35: **return** (Q', Γ, rel, P, F)

In lines 1-3 of the algorithm, we initialize the objects that are used. One notable difference with *pre** is that transitions originating from states outside of P are placed directly in *rel* because these states do not occur in transitions of $\mathcal{P}$, and thus examining them would yield no new transitions for $\mathcal{A}_{post*}$. As before, Q' and Δ' are initially just copies of Q and Δ respectively. Finally, *map* is used

to map PDS transitions to states and is initially empty. It is worth noting that this object is not explicitly mentioned in the pseudocode implementation for the original *post** algorithm in [5]. Its use is made explicit in this paper because the relation in the generalized algorithm is many-to-one instead of one-to-one.

In lines 4-7, we carry out the first step of the abstract procedure described previously. For each PDS transition of the form $\langle p, \sigma\rangle \hookrightarrow \langle p', \gamma_1\gamma_2\cdots\gamma_n\rangle$ with $n \geq 2$, which pushes two or more symbols onto the stack, a path of the form $p' \xrightarrow{\gamma_1\gamma_2\cdots\gamma_{n-1}} s_{n-1}$ where s_{n-1} is a fresh state is constructed. In addition, we map the PDS transition to s_{n-1}.

Within the while loop, each transition $t = (p, \beta, q)$ is examined. We work under the assumption that $\langle q, \rho\rangle \in post^*(C)$ where ρ is an arbitrary stack string, that is, there exists a path $q \xrightarrow{\rho} q_f$ where q_f is an arbitrary final state. Intuitively, t indicates that $\langle p, \beta\rho\rangle \in post^*(C)$. Lines 13-20 evaluate PDS transitions of the form $\langle p, \varepsilon\rangle \hookrightarrow \langle p', \tau\rangle$. If such a transition exists in Δ', then $\langle p', \tau\beta\rho\rangle$ is a successor of configuration $\langle p, \beta\rho\rangle$ and thus should also be added to $post^*(C)$. This is achieved by the construction of a path $p' \xrightarrow{\tau\beta} q$. Note that a part of this path will have been previously constructed in lines 4-7 if $|\tau| \geq 2$, and is completed with the help of *map*. Also note that if $\beta = \varepsilon$, lines 13-20 are skipped because in that case they have the same effect as lines 21-24, which are discussed below.

In lines 21-24, we examine PDS transitions of the form $\langle p, \beta\rangle \hookrightarrow \langle p', \tau\rangle$. If such a transition is found in Δ', then $\langle p', \tau\rho\rangle$ is a successor of configuration $\langle p, \beta\rho\rangle$ and thus should also be added to $post^*(C)$. Therefore, we form a path $p' \xrightarrow{\tau} q$ in the $\mathcal{P}$-automaton.

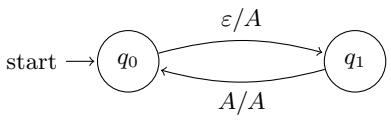
In lines 25-34, we deal with PDS transitions that pop two or more symbols from the stack. We work under the assumption that $\langle r, \rho\rangle \in post^*(C)$ where r is an arbitrary state. We look out for a series of transitions $t_1, t_2, \ldots, t_m$ that forms a path $p \xrightarrow{\beta\gamma\tau} r$ with $|\beta\gamma\tau| \geq 2$. If such a path exists, then $\langle p, \beta\gamma\tau\rho\rangle$ is also a configuration in $post^*(C)$. If there exists a PDS transition of the form $r = \langle p, \beta\gamma\tau\rangle \hookrightarrow \langle p', \sigma\rangle \in \Delta'$, then $\langle p', \sigma\rho\rangle$ is a successor of $\langle p, \beta\gamma\tau\rho\rangle$. Hence, we want to add a path $p' \xrightarrow{\sigma} r$ to the $\mathcal{P}$-automaton. As was the case with *pre**, we do not know the order in which we examine transitions $t_1, t_2, \ldots, t_m$. If we perform a backward search every time we see a transition t_k, where $2 \leq k \leq m$, to check the existence of transitions $t_1, t_2, \ldots, t_{k-1}$, then we might waste time with many negative checks. However, we know that if we examine $t_1 = (p, \beta, q)$, then all subsequent transitions $t_2 = (q, \gamma, q')$ (or $t_2 = (q, \varepsilon, q')$) may eventually be part of a path marked $\beta\gamma\tau$. Therefore, we introduce a new PDS transition (if one does not already exist) $e = \langle q, \gamma\tau\rangle \hookrightarrow \langle p', \sigma\rangle$ to take care of such cases. Moreover, if we indeed add this new PDS transition and r is mapped to a state s, i.e. r pushes two or more symbols onto the stack, then we map e to the same state s. In addition, if we add e to Δ', then we check whether we have already examined t_2, and if that is the case, we move t_2 from *rel* to *trans* to examine it again.

Finally, the output $\mathcal{P}$-automaton $\mathcal{A}_{post*}$ consists of the stack alphabet Γ, the set of initial states P and the set of final state F of $\mathcal{A}$. It also consists of the set of states Q', which is a superset of Q, and of the set of transitions *rel*, which is a superset of δ.

6.3 Drawbacks

Algorithm 2b eliminates the need for an input normalization step and facilitates the use of transitions that pop no symbols from the stack. However it presents a fairly significant drawback. Specifically, for a certain group of GPDAs the algorithm does not terminate. In some cases this is due to *post**, while in others it is due to *pre**.

One such example of non-termination becomes evident if we use the PDS $\mathcal{P}_4$ (Figure 7) and the $\mathcal{P}$-automaton $\mathcal{A}_4$ (Figure 8, left) as input to *post**. $\mathcal{P}_4$ contains only two transitions that form a cycle, and $\mathcal{A}_4$ expresses the configuration $\langle q_0, \varepsilon \rangle$. The desired output of *post** is the $\mathcal{P}$-automaton $\mathcal{A}_5$, which represents all configurations $\langle q_0, A^* \rangle$ and $\langle q_1, AA^* \rangle$ (Figure 8, right).



$$\Delta = \{\langle q_0, \varepsilon \rangle \hookrightarrow \langle q_1, A \rangle, \langle q_1, A \rangle \hookrightarrow \langle q_0, A \rangle\}$$

Fig. 7. The PDS $\mathcal{P}_4$ and its set of transitions Δ

Consider the execution of *post** with the above input. Initially, $trans = \{(q_0, \varepsilon, q_f)\}$, $rel = \emptyset$, $Q' = \{q_0, q_1, q_f\}$, $\Delta' = \{\langle q_0, \varepsilon \rangle \hookrightarrow \langle q_1, A \rangle, \langle q_1, A \rangle \hookrightarrow \langle q_0, A \rangle\}$, and $map = \emptyset$ (lines 1-3). Since Δ' contains no transitions that push two or more symbols onto the stack, lines 4-7 are skipped. In the first iteration of the while loop, (q_0, ε, q_f) is popped from *trans* (line 9), found not to be in *rel* (line 10), and added to *rel* (line 11). In this case $\beta = \varepsilon$, so lines 12-20 are skipped. PDS transition $\langle q_0, \varepsilon \rangle \hookrightarrow \langle q_1, A \rangle$ is evaluated in line 21, which leads to the addition of (q_1, A, q_f) to *trans* (line 22). No other PDS transitions that originate from q_0 exist, so lines 23-34 are skipped. In the following iteration, (q_1, A, q_f) is popped from *trans* (line 9), found not to be in *rel* (line 10), and added to *rel* (line 11). Because there exists no PDS transition in Δ' originating from $\langle q_1, \varepsilon \rangle$, lines 13-20 are skipped. PDS transition $\langle q_1, A \rangle \hookrightarrow \langle q_0, A \rangle$ is evaluated in line 21, and (q_0, A, q_f) is added to *trans* in the following line. Since there are no other PDS transitions originating from q_1, lines 23-34 are skipped. In the next iteration, (q_0, A, q_f) is popped from *trans* (line 9), found not to be in *rel* (line 10), and added to *rel* (line 11). In this case, the clause in line 15 holds because of $\langle q_0, \varepsilon \rangle \hookrightarrow \langle q_1, A \rangle \in \Delta'$. As a result, a fresh state s_1 is added to Q' and transitions (q_1, A, s_1) and (s_1, A, q_f) are added to *trans* (lines 16-17). There are

no other PDS transitions originating from q_0, so lines 18-34 are skipped. In the next iteration, (q_1, A, s_1) is popped from *trans* (line 9), found not to be in *rel* (line 10), and added to *rel* (line 11). When $\langle q_1, A\rangle \hookrightarrow \langle q_0, A\rangle$ is evaluated (line 21), it leads to the addition of (q_0, A, s_1) to *trans* (line 22). This process continues to form the infinite automaton of Figure 9 and clearly it never terminates. Analogous examples exist that demonstrate this issue in *pre**.

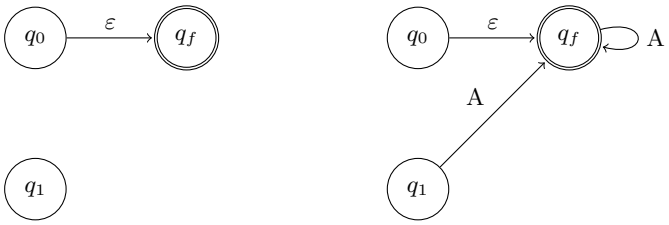


Fig. 8. The $\mathcal{P}$-automata $\mathcal{A}_4$ (left), and $\mathcal{A}_5$ (right)

The existence of similar cycles to the one presented above in GPDAs that are used as input for Algorithm 2b will result in an infinite loop. This problem does not occur in either of the other two algorithms.

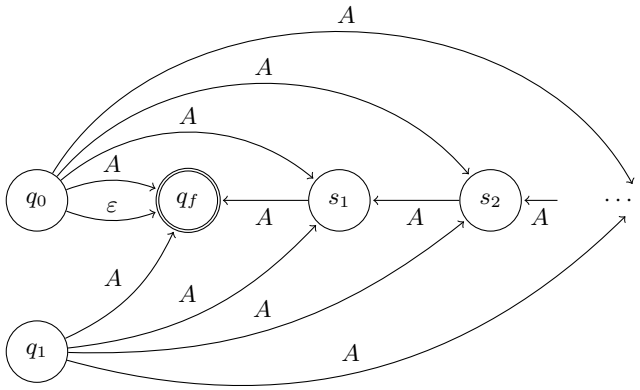


Fig. 9. The (infinite) output of $post^*(\mathcal{P}_4, \mathcal{A}_4)$

In addition, the current algorithm presents the same drawback as Algorithm 2a, that is, it does not differentiate between transitions that are never applicable and transitions that are not used in an accepting run of the automaton. However, this extra piece of information can be gained using an analogous process to the one described previously regarding Algorithm 2a.

7 Description of Algorithms 2c and 2d

Algorithm 2b operates in a generalized setting where words of arbitrary lengths may be popped from and pushed onto the stack. Therefore, *pre** and *post** are each other's dual.

Moreover, we define the *reverse* procedure, which operates on a PDS $\mathcal{P}$, and replaces every transition $q \xrightarrow{\sigma/\tau} r$ in $\mathcal{P}$ with a transition $r \xrightarrow{\tau/\sigma} q$. We call the resulting PDS $\mathcal{P}^R$. For an arbitrary set of configurations C, expressed by a $\mathcal{P}$-automaton $\mathcal{A}$ we have that $pre^*(\mathcal{P}, \mathcal{A}) = post^*(\mathcal{P}^R, \mathcal{A})$ and $post^*(\mathcal{P}, \mathcal{A}) = pre^*(\mathcal{P}^R, \mathcal{A})$. Therefore, we could implement the *pre** procedure as

$$pre^*(\mathcal{P}, \mathcal{A}) = post^*(reverse(\mathcal{P}), \mathcal{A})$$

and the *post** procedure as

$$post^*(\mathcal{P}, \mathcal{A}) = pre^*(reverse(\mathcal{P}), \mathcal{A})$$

These alternative implementations give rise to two variations of Algorithm 2b.

Algorithm 2c This algorithm replaces the *pre** implementation in Algorithm 2b with the alternative presented previously. It maintains the overall worst-case time complexity of Algorithm 2b, namely $\mathcal{O}(S^3T^4)$.

Algorithm 2d This algorithm replaces the *post** implementation in Algorithm 2b with the alternative described above. Hence, it exhibits an improved overall worst-case time complexity of $\mathcal{O}(S^2T^4)$.

Drawbacks Both of these variations have the same drawbacks as Algorithm 2b.

8 Implementations

All algorithms have been implemented in the same framework so as to ensure that the performance comparison is not affected by differences in programming environments. We have built on the work done by Dick Grune, who implemented Algorithm 1 and all necessary abstract data types (ADT) using the C programming language.

The algorithms rely largely on set manipulation. To this end, there are a number of ADT modules in the framework, namely (1) `Symbol.c` for (sets of) stack symbols and strings, (2) `State.c` for (sets of) GPDA and NFA states, (3) `Ptrans.c` for (sets of) GPDA and PDS transition, and (4) `Ntrans.c` for (sets of) NFA and $\mathcal{P}$-automaton transitions. Moreover, there are associative array modules that are utilized in the implementation of Algorithm 1, namely

`Sqs.c` and `StateVsd.c`, which have not been discussed in this paper. In addition, the framework contains modules `NPDA.c` and `sNPDA.c` for GPDAs and PDSs, and module `NFA.c` for NFAs and $\mathcal{P}$-automata.

Some new modules have been added for the purpose of implementing Algorithms 2a-2d. In particular, module `PPMap.c` has been constructed to map PDS transitions to other PDS transitions. Specifically, it is utilized in the implementation of Algorithm 2a during Step 2, in which transitions that pop two or more symbols from the stack are broken into multiple transitions that pop exactly one symbol from the stack. The module is used to map the original transition to the first extension transition. Furthermore, module `PSMap.c` has been created to map PDS transitions to states. This module performs the function of the *map* object presented in the pseudocode implementation for the generalized version of *post**.

The prototype implementations of the three algorithms are not optimal, but they serve their purpose well as proofs of concept and comparison tools. However, there are particular implementation details that cause certain shortcomings in the performance of the algorithms that need to be addressed.

Set size In the original framework, which was developed for the sole purpose of building a prototype implementation for Algorithm 1, sets are implemented to hold GPDA and NFA states and transitions. These are actually implemented by using arrays of a fixed maximum size. This is sufficient for the testing purposes of Algorithm 1, because the sizes of the various sets remain small in relation to the size of the input GPDA. In contrast, this limitation poses a problem for Algorithms 2a-2d. As noted in the previous section, these algorithms compute the intersection of two NFAs in order to determine whether a transition is useless. If one NFA contains m states, while the other contains n states, then their intersection may contain up to $m \times n$ states. As a result, the set of states overflows fairly quickly even if the input GPDA is relatively small. Therefore, the maximum size of the sets has to be manually adjusted for each individual input used during testing in order to prevent a crash. Consequently, initialization and operations on these sets become slower as their maximum size is increased. Obviously, this solution is by no means scalable, but it does allow us to compare the differences in speed of the different algorithms.

Memory usage This issue is closely related to the one mentioned above. In particular, since all sets have a large fixed size, they also have a large fixed memory footprint. This results in executions of the algorithms for small input automata to waste memory. Furthermore, when the maximum set size is increased to facilitate larger input automata, the maximum allowed stack size on the testing system has to be increased to prevent segmentation faults. As before, this issue has no bearing on the outcome of the comparison, but it is important to note that the implementations can be improved in this respect.

9 Performance

In order to compare the efficiency of the implementations of the five algorithms presented in the previous sections, an array of different input GPDAs was used. These automata were generated using the *PDAgen* tool developed by Dick Grune, and were manually adjusted as needed. The implementations were tested on a 64-bit Linux system with a 2.00 GHz processor.

The first round of testing compares the performance of all five algorithms on the same inputs. Thus the input GPDAs do not contain transitions that pop no symbols from the stack, as these are not valid input for Algorithm 2a. Moreover, the key goal of the comparison is to determine which algorithm scales better when the input GPDAs become large. Therefore, the inputs were designed to contain different numbers of states, transitions and maximum string lengths. The results are presented in Table 1. Q denotes the number of states in the input GPDA, T the number of transitions, S the maximum stack string length, $\overline{s}$ the average stack string length, and *L_SIZE* the maximum set size. DNT denotes that the algorithm did not terminate with the provided input.

Input GPDA							**Algorithm (sec)**				
#	Q	T	S	$\overline{s}$	S^3T^4	*L_SIZE*	1	2a	2b	2c	2d
P_1	2	2	5	2	2000	1200	0.37	0.67	0.47	0.58	DNT
P_2	4	5	5	2	78125	1200	0.38	1.33	1.08	1.29	1.00
P_3	6	6	9	4	944784	2000	0.66	3.14	2.16	2.32	2.12
P_4	11	10	5	2	1250000	1200	0.32	2.82	2.13	2.55	1.89
P_5	7	10	6	2	2160000	2000	0.65	4.27	3.31	3.70	DNT
P_6	7	10	7	3	3430000	1200	0.38	3.59	2.36	2.56	2.18
P_7	8	10	8	3	5120000	2000	0.66	4.84	3.36	3.77	3.23
P_8	7	10	12	6	17280000	2800	1.02	16.86	11.57	6.29	11.17
P_9	16	20	6	3	34560000	2000	0.65	18.21	12.68	11.58	7.69
P_{10}	7	20	6	1	34560000	2000	0.65	9.18	6.70	DNT	6.37
P_{11}	6	10	17	7	49130000	5000	2.06	34.48	25.56	11.15	26.14
P_{12}	22	31	5	2	115440125	3000	1.04	26.35	22.86	26.61	19.20
P_{13}	9	20	10	4	160000000	6000	2.50	78.33	159.49	45.00	99.24
P_{14}	8	20	13	5	351520000	8000	3.83	156.01	108.35	90.24	116.95
P_{15}	8	20	13	6	351520000	10000	5.96	274.46	325.94	141.99	376.41

Table 1. First round of testing results

In the second round of testing, GPDAs that contain transitions that pop no symbols from the stack were generated. Therefore, that comparison is limited between Algorithm 1 and Algorithms 2b-2d. The performance of the algorithms can be seen in Table 2.

The results show that Algorithm 1 outperforms the other four. As noted in [8], its worst-case performance is highly unlikely and the algorithm exhibits a very good average performance. As a matter of fact, the algorithm's performance seems to be minimally affected by the increase in the number of states and

Input GPDA							Algorithm (sec)			
#	Q	T	S	$\overline{s}$	S^3T^4	*L_SIZE*	1	2b	2c	2d
P_{20}	3	5	5	1	390625	1200	0.37	1.09	1.33	DNT
P_{21}	6	6	5	2	810000	1200	0.37	1.28	1.59	1.22
P_{22}	4	10	3	1	810000	2000	0.64	3.64	4.53	DNT
P_{23}	5	10	5	2	6250000	1200	0.59	3.40	2.57	DNT
P_{24}	10	10	5	2	6250000	2000	0.67	3.41	3.87	3.34
P_{25}	3	11	5	2	9150625	2000	0.68	4.03	4.41	DNT
P_{26}	8	8	9	4	26873856	6000	2.48	10.40	8.52	11.10
P_{27}	7	10	8	3	40960000	4000	1.48	7.44	7.36	7.76
P_{28}	9	11	8	4	59969536	2000	1.10	10.25	4.85	DNT
P_{29}	7	10	9	4	65610000	6000	2.43	12.52	DNT	15.29
P_{30}	13	19	8	3	533794816	6000	2.75	67.16	31.26	DNT
P_{31}	11	20	13	5	4569760000	8000	3.44	122.40	72.77	DNT

Table 2. Second round of testing results

transitions. Rather its efficiency is decreased because of the increase of *L_SIZE*, which is done for the benefit of Algorithms 2a-2d. To support this claim, Table 3 presents the results of executing three GPDAs of different sizes for various values of *L_SIZE*. In addition, Table 4 shows the performance of Algorithm 1 on different-sized inputs when *L_SIZE* is kept constant.

Input GPDA				Algorithm (sec)
#	Q	T	*L_SIZE*	1
P_{14}	8	10	1200	0.45
			2000	0.76
			4000	1.68
			8000	4.13
P_{30}	13	19	1200	0.40
			2000	0.69
			4000	1.56
			8000	3.90
P_{46}	44	80	1200	0.68
			2000	1.00
			4000	1.97
			8000	4.47

Table 3. Effect of *L_SIZE* on performance of Algorithm 1

Furthermore, the time taken for Algorithms 2a and 2b to complete grows roughly at the same rate as the product S^3T^4. This observation is in line with the worst-case time complexities of the algorithms presented in the previous sections. Algorithm 2b performs slightly better than Algorithm 2a on average. This advantage can be attributed to the fact that the former performs normalization

on-the-fly only when needed, and therefore no unnecessary operations are carried out.

In addition, there are cases, such as automata P_{10} and P_8 (Table 1), where the product S^3T^4 is greater for the former than for the latter, and yet Algorithms 2a and 2b terminate earlier in the first case. This is explained by the fact that the average string length $\overline{s}$ in P_{10} is equal to 1, while in P_8 it is equal to 6. Therefore, the maximum string length $S = 6$ that P_{10} presents is an outlier that causes the overestimation of the time complexity. Hence, the average string length $\overline{s}$ is a reliable secondary indicator of the expected performance of Algorithms 2a and 2b.

Input GPDA				**Algorithm (sec)**
#	Q	T	*L_SIZE*	1
P_{40}	16	20	1200	0.38
P_{41}	23	30	1200	0.42
P_{42}	26	40	1200	0.45
P_{43}	30	50	1200	0.47
P_{44}	29	60	1200	0.50
P_{45}	33	70	1200	0.68
P_{46}	44	80	1200	0.68
P_{47}	60	120	1200	1.39
P_{48}	78	140	1200	1.12
P_{49}	86	140	1200	1.33

Table 4. Fourth round of testing results

Interestingly, neither of the two variations of Algorithm 2b clearly outperforms the other. Algorithm 2d, which has a better worst-case time complexity, seems to often be faster than Algorithms 2b and 2c for small input, while Algorithm 2c seems to be more efficient for larger input.

10 Conclusion

We have implemented and compared two different algorithms for detecting useless transitions in GPDAs, and modified the second one to eliminate an input normalization step and slightly increase its average efficiency. Testing showed that the problem discussed in this paper can be most efficiently solved using Algorithm 1 presented by Fokkink et al. [8]. This algorithm has a worst-case time complexity in $\mathcal{O}(Q^4T)$, but the worst-case is unlikely and thus it presents a good average performance.

The generalized versions of the model-checking procedures *pre** and *post** that were presented in this paper facilitate the use of GPDAs as input and retain the efficiency of the original algorithms discussed in [5]. However, both procedures contain an infinite loop when the input GPDA contains cycles with

certain characteristics. Future work can focus on mitigating this issue by means of a cycle detection mechanism.

In addition, Algorithms 2a-2d could be augmented with the procedure described in Section 5 regarding the distinction between transitions that are never applicable and transitions that are never used in any accepting run. Algorithm 2a could also be adjusted to make use of an alternative normalization procedure so that it may accept any GPDA as input, but this would come at the cost of a greater worst-case time complexity.

The prototype implementations of the algorithms have a number of drawbacks that primarily stem from the need to manually choose the maximum allowed set size. They were sufficient for the purposes of this research, but new implementations that make use of dynamically growing sets could be constructed. Such an implementation for Algorithm 1, which was found to be the most efficient, could be used as a part of large projects that involve PDAs.

References

1. Bouajjani, A., Esparza, J., Maler, O.: Reachability analysis of pushdown automata: Application to model-checking. In: CONCUR '97: Concurrency Theory, 8th International Conference, Warsaw, Poland, July 1-4, 1997. pp. 135–150 (1997)
2. Caralp, M., Reynier, P., Talbot, J.: Trimming visibly pushdown automata. Theor. Comput. Sci. 578, 13–29 (2015)
3. Chervet, P., Walukiewicz, I.: Minimizing variants of visibly pushdown automata. In: Mathematical Foundations of Computer Science 2007, 32nd International Symposium, MFCS 2007, Ceský Krumlov, Czech Republic, August 26-31, 2007. pp. 135–146 (2007)
4. Crespi Reghizzi, S., Breveglieri, L., Morzenti, A.: Formal Languages and Compilation, chap. Pushdown Automata and Parsing, pp. 141–291. Springer, London (2013)
5. Esparza, J., Hansel, D., Rossmanith, P., Schwoon, S.: Efficient algorithms for model checking pushdown systems. In: Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000. pp. 232–247 (2000)
6. Feng, H.H., Giffin, J.T., Huang, Y., Jha, S., Lee, W., Miller, B.P.: Formalizing sensitivity in static analysis for intrusion detection. In: 2004 IEEE Symposium on Security and Privacy (S&P 2004), 9-12 May 2004, Berkeley, CA, USA. pp. 194–208 (2004)
7. Finkel, A., Willems, B., Wolper, P.: A direct symbolic approach to model checking pushdown systems (extended abstract). In: Infinity'97, Second International Workshop on Verification of Infinite State Systems, Electronic Notes in Theoretical Computer Science. vol. 9, pp. 27–37. Elsevier BV (1997)
8. Fokkink, W., Grune, D., Hond, B., Rutgers, P.: Detecting useless transitions in pushdown automata. CoRR abs/1306.1947 (2013)
9. Hopcroft, J.E., Ullman, J.D.: Introduction to Automata Theory, Languages and Computation. Addison-Wesley (1979)
10. Olsen, D.R.: Pushdown automata for user interface management. ACM Trans. Graph. 3(3), 177–203 (1984)