The Quotient Operation on Input-Driven Pushdown Automata

Alexander Okhotin, Kai Salomaa

The quotient operation on input-driven pushdown automata

Alexander Okhotin[1] and Kai Salomaa[2]

[1] St. Petersburg State University, 14th Line V.O., 29B, Saint Petersburg 199178, Russia alexander.okhotin@spbu.ru
[2] School of Computing, Queen's University, Kingston, Ontario K7L 2N8, Canada, ksalomaa@cs.queensu.ca

Abstract. The quotient of a formal language K by another language L is the set of all strings obtained by taking a string from K that ends with a suffix from L, and removing that suffix. The quotient of a regular language by any language is always regular, whereas the context-free languages and many of their subfamilies, such as the linear and the deterministic languages, are not closed under the quotient operation. This paper establishes the closure of the family of input-driven pushdown automata (IDPDA), also known as visibly pushdown automata, under the quotient operation. A construction of automata representing the result of the operation is given, and its state complexity with respect to nondeterministic IDPDA is shown to be $m^2n + O(m)$, where m and n is the number of states in the automata recognizing K and L, respectively.

1 Introduction

Let K and L be formal languages over some alphabet Σ. Then, the right-quotient of K by L is the following formal language, denoted by $K \cdot L^{-1}$.

$$K \cdot L^{-1} = \{ u \mid \exists v \in L : \ uv \in K \}$$

The left-quotient operation is defined symmetrically.

$$L^{-1} \cdot K = \{ v \mid \exists u \in L : \ uv \in K \}$$

The family of regular languages is closed under quotient with *any* language: as shown by Ginsburg and Spanier [6], if K is a regular language, then the languages $K \cdot L^{-1}$ and $L^{-1} \cdot K$ are both regular, regardless of L. For formal grammars, Ginsburg and Spanier [6] showed that for every context-free language K and a regular language L, their quotients are again context-free. On the other hand, if both arguments can be any context-free languages, then their quotient need not be context-free: indeed, for $K = a\{ b^\ell a^\ell \mid \ell \geqslant 1 \}^*$ and $L = \{ a^m b^{2m} \mid m \geqslant 1 \}^*$, their quotient satisfies $K^{-1}L \cap b^* = \{ b^{2^n} \mid n \geqslant 1 \}$. Besides just the non-closure, it is known that every recursively enumerable set is representable as a quotient of two context-free languages [8].

For an important subfamily of grammars, the *LR(k) grammars*, which are equivalently defined by *deterministic pushdown automata* (DPDA)—it is known that they are closed under right-quotient with regular languages, but not closed under left-quotient with finite languages [5]. Another classical subfamily of *LL(k) grammars* is not closed under both right- and left-quotient with regular languages [18]. On the other hand, the family of languages recognized by pushdown automata with one stack symbol (the *one-counter languages*) is surprisingly closed under quotient [9].

This paper investigates the quotient operation for one of the most important subclasses of pushdown automata: the *input-driven pushdown automata* (IDPDA). These automata were introduced in the work of Mehlhorn [10] and of von Braunmühl and Verbeek [4], and are characterized by the following restriction: their input alphabet is split into three disjoint classes of symbols, on which the automaton must push one symbol onto the stack (left brackets), or must pop one symbol off the stack (right brackets) or may not touch the stack (neutral symbols). The model defined by Mehlhorn [10] was deterministic (DIDPDA); von Braunmühl and Verbeek [4] introduced its nondeterministic variant (NIDPDA) and presented a novel determinization construction. Furthermore, Mehlhorn [10] and von Braunmühl and Verbeek [4] presented efficient algorithms for simulating these automata.

Later, Alur and Madhusudan [1] reintroduced IDPDA under the name of *visibly pushdown automata* and pointed out their applications to verification; their work revived the interest in the model. One of the theoretical contributions of Alur and Madhusudan [1] is the study of the succinctness of description by input-driven automata. In particular, they proved that determinizing an n-state NIDPDA requires $2^{\Theta(n^2)}$ states in the worst case, and initiated a systematic study of their closure properties.

In the follow-up work, the state complexity of the main language-theoretic operations on IDPDA was determined. The precise number of states necessary to represent concatenation, Kleene star and reversal by deterministic IDPDA (DIDPDA) was later determined by the authors [14]. For Boolean operations, the state complexity results were obtained by Han and Salomaa [7] and by Piao and Salomaa [16]. Recently, the authors [15] established the closure of IDPDA under the edit distance operation. For more details on the descriptional complexity of input-driven automata, an interested reader is directed to a fairly recent survey paper [12].

This paper investigates the quotient operation on IDPDA. The main result is that the family of languages recognized by IDPDA is closed under the quotient. If both argument languages consist only of well-nested strings, then so does their quotient, and the construction of an IDPDA for that quotient is straightforward. In the general case, without the well-nestendness condition, the closure is established by a more involved construction: given a pair of NIDPDA with m and n states, a construction of a $(3m+m^2n)$-state NIDPDA recognizing their quotient is described in Section 3.

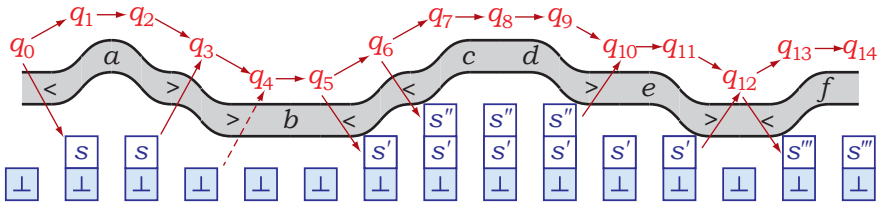

Fig. 1. A sample computation of an IDPDA on an ill-nested string.

The rest of the paper establishes a close lower bound to this construction. The general plan of the lower bound argument, explained in Section 4, is to construct witness languages of a special form, so that the task of constructing them is basically a problem of finding witness NFA (nondeterministic finite automata) for the state complexity of a certain unconventional operation on languages. This operation has been named *palindromic quotient*, and the NFA state complexity problem for it is solved in Section 5. The results are adapted to NIDPDA in the final Section 6

2 Input-driven automata

The input alphabet of an *input-driven pushdown automaton* (IDPDA) [1,2,10] is split into three disjoint sets of *left brackets* Σ_{+1}, *right brackets* Σ_{-1} and *neutral symbols* Σ_0. If the input symbol is a left bracket from Σ_{+1}, then the automaton always pushes one symbol onto the stack. For a right bracket from Σ_{-1}, the automaton must pop one symbol. Finally, for a neutral symbol in Σ_0, the automaton may not use the stack. In this paper, symbols from Σ_{+1} and Σ_{-1} shall be denoted by left and right angle brackets, respectively ($<$, $>$), whereas lower-case Latin letters from the beginning of the alphabet ($a, b, c, \ldots$) shall be used for symbols from Σ_0. Input-driven automata may be deterministic (DIDPDA) and nondeterministic (NIDPDA).

Under the original definition used by Mehlhorn [10] and by von Braunmühl and Verbeek [4], input-driven automata operate on input strings, in which the brackets are *well-nested*. When an input-driven automaton reads a left bracket ($< \in \Sigma_{+1}$), it pushes a symbol onto the stack. This symbol is popped at the exact moment when the automaton encounters the matching right bracket ($> \in \Sigma_{-1}$). Thus, a computation of an input-driven automaton on any well-nested substring leaves the stack contents untouched.

For instance, in Figure 1, the fragment of the computation beginning in the state q_4 and ending in the state q_{12} processes a well-nested substring $b{<}{<}cd{>}e{>}$, and therefore ends with the same stack contents as in which it began (in this case, the empty stack).

The more general definition of input-driven automata proposed by Alur and Madhusudan [1] also allows *ill-nested* input strings, such as the whole string ${<}a{>}{>}b{<}{<}cd{>}e{>}{<}f$ in Figure 1. For every unmatched left bracket, the symbol

pushed to the stack when reading this bracket is never popped, and remains in the stack to the end of the computation; in the figure, this is the case with the symbol s''' pushed in the state q_{12}. An unmatched right bracket is read with an empty stack: instead of popping a stack symbol, the automaton merely detects that the stack is empty and makes a special transition, which leaves the stack empty. The latter happens in the state q_3 in the figure, where the special transition upon an unmatched right bracket leads the automaton to the state q_4.

Definition 1 (von Braunmühl and Verbeek [4]; Alur and Madhusudan [1]). *A nondeterministic input-driven pushdown automaton (NIDPDA) over an alphabet $\widetilde{\Sigma} = (\Sigma_{+1}, \Sigma_{-1}, \Sigma_0)$ consists of*

- *a finite set Q of states, with set of initial states $Q_0 \subseteq Q$ and accepting states $F \subseteq Q$;*
- *a finite stack alphabet Γ, and a special symbol $\bot \notin \Gamma$ for the empty stack;*
- *for a neutral symbol $c \in \Sigma_0$, a transition function $\delta_c \colon Q \to 2^Q$ gives the set of possible next states;*
- *for each left bracket symbol $< \in \Sigma_{+1}$, the behaviour of the automaton is described by a function $\delta_< \colon Q \to 2^{Q \times \Gamma}$, which, for a given current state, provides a set of pairs (q, s), with $q \in Q$ and $s \in \Gamma$, where each pair means that the automaton enters the state q and pushes s onto the stack;*
- *for every right bracket symbol $> \in \Sigma_{-1}$, there is a function $\delta_> \colon Q \times (\Gamma \cup \{\bot\}) \to 2^Q$ specifying possible next states, assuming that the given stack symbol is popped from the stack (or that the stack is empty).*

A configuration is a triple (q, w, x), with the current state $q \in Q$, remaining input $w \in \Sigma^*$ and stack contents $x \in \Gamma^*$. Possible next configurations are defined as follows.

$$
\begin{aligned}
(q, cw, x) &\vdash_A (q', w, x), && c \in \Sigma_0,\ q \in Q,\ q' \in \delta_c(q)\\
(q, <w, x) &\vdash_A (q', w, sx), && < \in \Sigma_{+1},\ q \in Q,\ (q', s) \in \delta_<(q)\\
(q, >w, sx) &\vdash_A (q', w, x), && > \in \Sigma_{-1},\ q \in Q,\ s \in \Gamma,\ q' \in \delta_>(q, s)\\
(q, >w, \varepsilon) &\vdash_A (q', w, \varepsilon), && > \in \Sigma_{-1},\ q' \in \delta_>(q, \bot)
\end{aligned}
$$

The language recognized by A is the set of all strings $w \in \Sigma^*$, on which the automaton, having begun its computation in the configuration (q_0, w, ε), eventually reaches a configuration of the form (q, ε, x), with $q \in F$ and with any stack contents $x \in \Gamma^*$.

An NIDPDA is deterministic (DIDPDA), if there is a unique initial state and every transition provides exactly one action.

As shown by von Braunmühl and Verbeek [4], every n-state NIDPDA operating on well-nested strings can be transformed to a 2^{n^2}-state DIDPDA. Alur and Madhusudan [1] proved that $2^{\Omega(n^2)}$ states are necessary in the worst case, and also extended the transformation to handle ill-nested inputs, with the resulting DIDPDA using 2^{2n^2} states.

For more details on input-driven automata and their complexity, the readers are directed to a recent survey [12].

3 Closure under the quotient

In this section, it is proved that the language family defined by input-driven automata is closed under the quotient operation.

For the class of regular languages, it is well-known that they are closed under quotient *with any language*. Indeed, if K is recognized by a deterministic finite automaton (DFA), then, from each state q of this DFA, it is the case or not the case that the DFA accepts some string from L beginning from q. Depending on this, q is relabelled as accepting or rejecting, and the resulting DFA recognizes exactly the quotient $K \cdot L^{-1}$.

Turning to input-driven automata, as long as all strings in L are well-nested, the same property still holds. That is, an n-state DIDPDA recognizing K can be transformed to an n-state DIDPDA recognizing the quotient $K \cdot L^{-1}$, simply by relabelling its states.

Given an arbitrary pair of NIDPDA, $\mathcal{A}$ and $\mathcal{B}$, the goal is to construct a new NIDPDA $\mathcal{C}$ that recognizes their quotient, $L(\mathcal{A}) \cdot L(\mathcal{B})^{-1}$. Whenever the automaton $\mathcal{A}$ accepts a string uv, and the other automaton $\mathcal{B}$ accepts the string v, the simulating automaton should therefore accept u. If none of the brackets in the u-part of uv match any brackets in the v-part, then the simulation proceeds like in the case of finite automata, without using any extra states. In the general case, the string u may have unmatched left brackets, v may have unmatched right brackets, and these brackets *match each other* in uv; thus, the computation of $\mathcal{A}$ may rely on the data transferred from u to v in the stack symbols. The simulating automaton $\mathcal{C}$ is given only u, with its unmatched left brackets, and while doing so, it has to guess the string v and imagine the computations of both $\mathcal{A}$ and $\mathcal{B}$ on this guessed v.

In the computation of $\mathcal{C}$ on u, these imaginary computations on v are traced *backwards*, so that whenever a left bracket ($<$) in u matches a right bracket ($>$) in v, the simulating automaton $\mathcal{C}$, upon reading u up to that left bracket, tracks the imaginary computations of $\mathcal{A}$ and $\mathcal{B}$ *that begin from the matching right bracket ($>$) in v and accept in the end of v*. As $\mathcal{C}$ finishes reading the string u, its imaginary computations on v are backtracked to their beginning at the boundary between u and v. Then, at this point $\mathcal{C}$ ensures that $\mathcal{B}$'s computation is in its initial configuration, whereas the actual simulated computation of $\mathcal{A}$ on u smoothly continues into the imaginary computation of $\mathcal{A}$ on v. Thus, $\mathcal{C}$ finally verifies that a string v and a computation on it that it has been guessing actually do exist; and accordingly $\mathcal{C}$ accepts u.

This idea is implemented in the following construction.

Lemma 1. *Let K be a language recognized by an NIDPDA $\mathcal{A}$ with the set of states P and with the pushdown alphabet Γ, and let L be another language recognized by an NIDPDA $\mathcal{B}$ with the set of states Q and with the pushdown alphabet Ω. Then, the quotient $K \cdot L^{-1}$ is recognized by an NIDPDA $\mathcal{C}$ with the set of states $(P \times \{0,1\}) \cup P \cup (P \times P \times Q)$ and with the pushdown alphabet $(\Gamma \times \{0,1\}) \cup \Gamma \cup \{\#\} \cup (\Gamma \times P \times Q)$.*

Proof (a sketch). At the **first phase** of the computation of $\mathcal{C}$ on an input string u, the simulation of the computations of $\mathcal{A}$ and $\mathcal{B}$ on its imaginary continuation v has not yet been started. This means that $\mathcal{C}$ assumes that all left brackets read so far are either going to have a matching right bracket in u, or are unmatched both in u and in v.

Thus, at the first phase, $\mathcal{C}$ simply simulates the operation of $\mathcal{A}$ on a prefix of u, while maintaining a single extra bit of data: whether the stack is empty or not. This is represented in states of the form (p, d), where $p \in P$ is the state of $\mathcal{A}$, and $d \in \{0, 1\}$, with $d = 0$ representing stack emptiness. While in these states, $\mathcal{C}$ uses stack symbols of the form (s, d), with $s \in \Gamma$ and $d \in \{0, 1\}$, which also carry the information on whether this stack symbol is at the bottom of the stack ($d = 0$). This allows the simulating automaton to enter a state of the form $(p, 0)$ upon popping the last symbol from the stack, and thus always be aware of its stack's emptiness.

Every time $\mathcal{C}$ reads a left bracket ($<$), it nondeterministically guesses whether this bracket has a matching right bracket ($>$) in v. If $\mathcal{C}$ guesses that this is not the case, it pushes the same stack symbol as $\mathcal{A}$ would push (that is, $\mathcal{C}$ pushes $(s, 0)$ or $(s, 1)$, if $\mathcal{A}$ would push s), and continues its computation in a state of the form $(p, 0)$ or $(p, 1)$. If later, while still at the first phase, $\mathcal{C}$ encounters a matching right bracket and pops that symbol, it again behaves as $\mathcal{A}$ would do, remaining in a state from $P \times \{0, 1\}$.

At some point, $\mathcal{C}$ may read a left bracket ($<$) and decide that it has a matching right bracket in v, so that $\mathcal{A}$ operating on uv would transfer some stack symbol s from the left bracket ($<$) to the right bracket ($>$). If this guess is correct, then this left bracket is unmatched in u, and thus $\mathcal{C}$ will never have a chance to pop the stack symbol it pushes at this moment; for that reason, it pushes a special stack symbol (#) that will cause immediate rejection if it is ever popped. At the same time, $\mathcal{C}$ guesses the computations of $\mathcal{A}$ and $\mathcal{B}$ on a suffix of v containing the matching right bracket ($>$) and the neighbouring well-nested substrings, and enters the second phase of the simulation in a state from $P \times P \times Q$.

In the **second phase**, $\mathcal{C}$ uses triples of the form $(p, \widetilde{p}, \widetilde{q})$ as states, and, while reading the input string u from left to right, it also guesses an imaginary string v from right to left, along with the computations of $\mathcal{A}$ and of $\mathcal{B}$ on that imaginary string. According to this plan, the first component of each triple, $p \in P$, is the state of the ongoing simulation of $\mathcal{A}$ on the prefix of u read so far. The other two components are the states of $\mathcal{A}$ and $\mathcal{B}$ processing v. To be precise, the second component, $\widetilde{p} \in P$, is a state, beginning from which $\mathcal{A}$ accepts a suffix of v guessed in the course of this simulation, whereas $\widetilde{q} \in Q$ is a state of $\mathcal{B}$, beginning from which it accepts the same guessed suffix of v.

When $\mathcal{C}$ nondeterministically decides to move to the second phase along with reading a left bracket ($<$), it guesses $\mathcal{A}$'s and $\mathcal{B}$'s computations on the last suffix of the imaginary second part of the string. If $\mathcal{C}$'s stack is empty—that is, if $\mathcal{C}$ is in a state $(p, 0)$—then the last suffix of v is of the form $x{>}y$, where x is a well-nested string, the right bracket ($>$) following x is the one that matches the current left bracket ($<$) in u, and y is a concatenation of a *descending string* and

an *ascending string* (that is, a concatenation of well-nested strings and right brackets, followed by well-nested strings and left brackets). All right brackets in y are then unmatched both in u and in the earlier part of v, and accordingly, $\mathcal{C}$ may enter any state $(p', \widetilde{p}, \widetilde{q})$ satisfying the following conditions:

1. upon reading this left bracket ($<$) in the state p, $\mathcal{A}$ pushes some stack symbol $s \in \Gamma$ and enters the state p';
2. the automaton $\mathcal{A}$, having begun its computation on $x{>}y$ in the state $\widetilde{p}$ and with s on the stack, accepts;
3. the other automaton $\mathcal{B}$, having begun its computation on $x{>}y$ in the state $\widetilde{q}$ and with the empty stack, accepts as well.

In the other case, if $\mathcal{C}$'s stack is not empty, and it is therefore in a state $(p, 1)$, the suffix of v is of the form $x{>}y$, where both x and y are well-nested, and the above three conditions remain the same.

Transitions of $\mathcal{C}$ in a state $(p, \widetilde{p}, \widetilde{q})$ are defined as follows. A right bracket ($>$) cannot be read in this state, and if it is encountered, $\mathcal{C}$ rejects.

Upon reading a neutral symbol $c \in \Sigma_0$, the simulation of $\mathcal{A}$ in the first component continues, while the last two components stay unchanged.

When reading a left bracket ($<$), the automaton $\mathcal{C}$ again has to guess whether this bracket has a matching right bracket ($>$) in v. In case it does, $\mathcal{C}$ pushes the stack symbol (#) that will cause rejection if popped, and advances the simulation in all three components of the state in the same way as it did when entering the second phase. On the other hand, if $\mathcal{C}$ nondeterministically guesses that this left bracket ($<$) has a matching bracket in u, it suspends the simulation of $\mathcal{A}$ and $\mathcal{B}$ on the imaginary suffix v, pushing a triple $(s, \widetilde{p}, \widetilde{q})$ onto the stack, where s is the stack symbol in the ongoing simulation of $\mathcal{A}$ on u. Then, $\mathcal{C}$ enters a state $p' \in P$ and begins processing the current well-nested substring of u in the state from P, simulating only $\mathcal{A}$.

When this well-nested substring ends, $\mathcal{C}$ reads the matching right bracket ($>$) in u and pops the triple $(s, \widetilde{p}, \widetilde{q})$ from the stack. Then, it resumes the second phase of the simulation in the state $(p'', \widetilde{p}, \widetilde{q})$, where p'' is the next state in the ongoing simulation of $\mathcal{A}$ on u.

The precise correctness statement of the construction takes the following form. When the simulating NIDPDA, after having read a string $t{<_1}u_1 \ldots {<_h}u_h \in \Sigma^*$, where t is any string, $u_1, \ldots, u_h$ are well-nested strings and $<_1, \ldots, <_h$ are unmatched left brackets in this string, is in a state $(p, \widetilde{p}, \widetilde{q})$ and has stack contents $(s_h, p_h, q_h) \ldots (s_1, p_1, q_1)$, this means that, **first**, there exists a computation of $\mathcal{A}$ on the string $t{<_1}u_1 \ldots {<_h}u_h$ that pushes each symbol s_i on the corresponding left bracket $<_i$, and reaches the state p after reading $t{<_1}u_1 \ldots {<_h}u_h$, and **second**, there exists a string of the form $v = v_h{>_h} \ldots v_1{>_1}w$, where $v_1, \ldots, v_h$ are well-nested strings, $>_1, \ldots, >_h$ are right brackets and $w \in \Sigma^*$ is any string that has no matching right brackets ($>$) to any left brackets ($<$) in t, so that $\mathcal{A}$, having begun its computation on v in the state $\widetilde{p}$, with the stack contents $s_h \ldots s_1$, after popping each right bracket $>_i$ will be in the corresponding state p_i, and will accept in the end, whereas $\mathcal{B}$, having begun its computation on the same string

v in the state $\widetilde{q}$ and with the empty stack, will be in the state q_i after each right bracket $>_i$, and will accept the string as well.

The correctness statement could be proved by induction on the length of the computation.

Finally, accepting states are of the form (p, p, q_0), that is, $\mathcal{A}$ finishes reading u in the state p, and $\mathcal{A}$ accepts v beginning in the state p, and also $\mathcal{B}$ accepts v beginning in the state q_0. Then, $\mathcal{C}$ recognizes exactly the desired quotient. □

This proves the closure under right-quotient. Since the family of languages recognized by input-driven automata is closed under reversal (where, in the reversed string, left brackets become right brackets and vice versa [2]) the closure result also extends to the left-quotient operation.

Theorem 1. *The family of languages recognized by input-driven pushdown automata is closed under right-quotient and left-quotient.*

4 Plan for a lower bound argument

The construction given in the previous section uses $3m+m^2n$ states to represent the quotient, and it turns out that it cannot be much improved upon. A lower bound on the state complexity of the quotient of NIDPDA shall be proved using witness languages of the following general form.

Fix an alphabet of labels, Γ. The first language contains nested sequences of brackets with the matching brackets having identical labels; it is a subset of the following base language.

$$K_0 = \{ <_{a_1} \dots <_{a_m} >_{a_m} \dots >_{a_1} \mid m \geqslant 0,\ a_1, \dots, a_m \in \Gamma \}$$

All strings in the second language consist of right brackets ($>$), which are to be erased by the quotient operation. Thus, the second language is a subset of the following language.

$$L_0 = \{ >_{a_m} \dots >_{a_1} \mid m \geqslant 0,\ a_1, \dots, a_m \in \Gamma \}$$

An automaton $\mathcal{A}$ recognizing a subset $K \subseteq K_0$ performs two tasks. First, upon reading each bracket $<_a$, it pushes the symbol a to stack, and upon reading a bracket $>_a$ it ensures that the symbol being popped is a; doing this task does not require any states. Second, it operates on the string as a DFA, ensuring that it belongs to a certain regular language.

The second automaton $\mathcal{B}$ recognizes a subset $L \subseteq L_0$ essentially as a DFA.

Then, the quotient $K \cdot L^{-1}$ contains a string of the form $<_{a_1} \dots <_{a_m}$ if the whole string $<_{a_1} \dots <_{a_m} >_{a_m} \dots >_{a_1}$ is in K, whereas its second half $>_{a_m} \dots >_{a_1}$ belongs to L.

In order to construct efficient witness languages of this form, it is convenient to reformulate them in terms of finite automata, and to consider a related state complexity problem for finite automata. Let every left bracket ($<_a$) labelled with

a symbol $a \in \Gamma$ be regarded as a symbol a, and let every right bracket $(>_a)$ be regarded as $\widetilde{a}$, from a marked copy of the alphabet $\widetilde{\Gamma} = \{ \widetilde{a} \mid a \in \Gamma \}$. Then the associated state complexity problem for finite automata over $\Gamma \cup \widetilde{\Gamma} \cup \{\#\}$ is concerned with the complexity of the following *palindromic quotient* operation on languages with respect to NFAs.

$$\mathrm{PQ}(K, L) = \{ a_1 \dots a_m \mid a_1 \dots a_m \# \widetilde{a}_m \dots \widetilde{a}_1 \in K,\, \widetilde{a}_m \dots \widetilde{a}_1 \in L \}$$

Lemma 2. *Let $K \subseteq \Gamma^* \# \widetilde{\Gamma}^*$ and $L \subseteq \widetilde{\Gamma}^*$ be any languages, and define the corresponding languages over the alphabet of brackets as follows.*

$$K' = \{ <_{a_1} \dots <_{a_m} >_{a_m} \dots >_{a_1} \mid a_1 \dots a_m \# \widetilde{a}_m \dots \widetilde{a}_1 \in K \}$$
$$L' = \{ >_{a_m} \dots >_{a_1} \mid \widetilde{a}_m \dots \widetilde{a}_1 \in L \}$$

Then:

1. *if K is recognized by an m-state NFA, then K' is recognized by an m-state NIDPDA;*
2. *if L is recognized by an n-state NFA, then L' is recognized by an n-state NIDPDA;*
3. *if $K' \cdot (L')^{-1}$ is recognized by an N-state NIDPDA, then $\mathrm{PQ}(K, L)$ is recognized by an N-state NFA.*

In particular, to prove the third part, one can directly transform an IDPDA recognizing the quotient $K' \cdot (L')^{-1}$ to an NFA recogizing the palindromic quotient $PQ(K, L)$ by eliminating all transitions by right brackets and by ignoring all symbols pushed to the stack upon reading left brackets.

5 The lower bound for NFA

In order to apply Lemma 2, the task is now to determine the state complexity of the *palindromic quotient* operation with respect to NFAs. The tools for doing this are well-known.

Definition 2 (Birget [3]). *Let $L \subseteq \Sigma^*$ and $S = \{(x_1, y_1), \dots, (x_m, y_m)\}$, $x_i, y_i \in \Sigma^*$, $i = 1, \dots, m$. The set S is a* fooling set *for L, if*

1. *$x_i y_i \in L$ for all $1 \leqslant i \leqslant m$,*
2. *$x_i y_j \notin L$ or $x_j y_i \notin L$ for all $1 \leqslant i < j \leqslant m$.*

The *nondeterministic state complexity* of a regular languate L, $\mathrm{nsc}(L)$, is the minimal number of states of any NFA recognizing L.

Lemma 3 (Fooling set lemma [3]). *If a regular language L has a fooling set of cardinality k, then $\mathrm{nsc}(L) \geqslant k$.*

For an alphabet Σ define $\widetilde{\Sigma} = \{\widetilde{a} \mid a \in \Sigma\}$. For a string $w = a_1 \cdots a_k$, $a_i \in \Sigma$, $1 \leqslant i \leqslant k$, let $\widetilde{w} = \widetilde{a}_1 \cdots \widetilde{a}_k$.

Consider an alphabet $\Omega = \Sigma \cup \widetilde{\Sigma} \cup \{\#\}$, where $\# \notin \Sigma \cup \widetilde{\Sigma}$. For $K, L \subseteq \Omega^*$ define

$$\mathrm{PQ}(K, L) = \{w \in \Sigma^* \mid w\#\widetilde{w}^R \in K,\ \widetilde{w}^R \in L\}.$$

The lower bound for the state complexity of the operation $\mathrm{PQ}(\cdot, \cdot)$ will be used for obtaining a lower bound for the state complexity of quotient of input driven languages. For this reason the alphabet is partitioned into sets Σ, $\widetilde{\Sigma}$ and $\{\#\}$ which play the roles of left brackets, right brackets and neutral symbols, respectively.

Lemma 4. *If A is an NFA with n states and B an NFA with m states, the language $\mathrm{PQ}(L(A), L(B))$ has an NFA with $n^2 \cdot m$ states.*

Proof. The language $\mathrm{PQ}(L(A), L(B))$ can be recognized by an NFA C operating as follows. On input $w \in \Sigma^*$, C simulates in parallel (i) a computation of A from a start state to a state q_1, (ii) a computation of A in reverse starting from a final state on the string $\widetilde{w}$, ending in a state q_2, and (iii) a computation of B from a final state in reverse on the string $\widetilde{w}$ ending in a state p. Thus, the states of C are triples (q_1, q_2, p) where q_1, q_2 are states of A and p is a state of B. A state (q_1, q_2, p) is accepting if A has a transition on $\#$ from q_1 to q_2 and p is a start state of B.

Now C has $n^2 \cdot m$ states and, by the choice of the final states, it is clear that $L(C) = \mathrm{PQ}(L(A), L(B))$. □

Lemma 5. *Let $\Sigma = \{a, b, c\}$ and $\Omega = \Sigma \cup \widetilde{\Sigma} \cup \{\#\}$. For $n, m \in \mathbb{N}$ there exist regular languages K and L over the alphabet Ω with $\mathrm{nsc}(K) = n$ and $\mathrm{nsc}(L) = m$ such that*

$$\mathrm{nsc}(\mathrm{PQ}(K, L)) \geqslant n^2 \cdot m.$$

Proof. Define

$$K = \{u_1\#u_2\#\cdots\#u_\ell \mid u_i \in \Omega^*,\ |u_i|_a + |u_i|_{\widetilde{b}} \equiv 0 \pmod{n},\ i = 1, \ldots, \ell\},$$
$$L = \{v \in \widetilde{\Sigma}^* \mid |v|_{\widetilde{c}} \equiv 0 \pmod{m}\}.$$

Note that the definition allows some of the substrings u_i to be empty which means that the strings of K may begin or end with $\#$ and have consecutive occurrences of $\#$.

The language K is recognized by an NFA $A = (\Omega, Q, 0, 0, \delta)$ where $Q = \{0, 1, \ldots, n-1\}$ and the transitions of δ are defined by setting

1. $\delta(i, a) = \delta(i, \widetilde{b}) = i + 1$ for $i = 0, \ldots, n-2$, and $\delta(n-1, a) = \delta(n-1, \widetilde{b}) = 0$,
2. $\delta(i, b) = \delta(i, c) = \delta(i, \widetilde{a}) = \delta(i, \widetilde{c}) = i$ for $i = 0, \ldots, n-1$,
3. $\delta(0, \#) = 0$ and $\delta(i, \#)$ is undefined for $i = 1, \ldots, n-1$.

The automaton A is, in fact, an incomplete DFA having a cycle of length n where the cycle counts the sum of the numbers of symbols a and $\widetilde{b}$ modulo n. Transitions on # are defined only when the current sum has a value divisible by n. This means that A checks that in the substring between two occurrences of # the sum of the numbers of occurrences of a and $\widetilde{b}$ must be divisible by n and A recognizes exactly the language K.

It is clear that L has an NFA with a cycle of length m that simply verifies that the input is in $\{\widetilde{a}, \widetilde{b}, \widetilde{c}\}^*$ and counts the number of occurrences of symbols $\widetilde{c}$ modulo m.

For establishing the lower bound for the nondeterministic state complexity of $\mathrm{PQ}(K, L)$ we define

$$S = \{(a^i b^j c^k, a^{n-i} b^{n-j} c^{m-k}) \mid 0 \leqslant i, j \leqslant n-1,\ 0 \leqslant k \leqslant m-1\}.$$

The set S has cardinality $n^2 \cdot m$ and to prove the claim, by Lemma 3, it is sufficient to verify that S is a fooling set for $\mathrm{PQ}(K, L)$.

For any pair $(a^i b^j c^k, a^{n-i} b^{n-j} c^{m-k})$ of S we have $a^i b^j c^k \cdot a^{n-i} b^{n-j} c^{m-k} \in \mathrm{PQ}(K, L)$ because with $w = a^i b^j c^k a^{n-i} b^{n-j} c^{m-k}$ we have $w\#\widetilde{w}^R \in K$ and $\widetilde{w}^R \in L$ due to the observations that $|w|_a + |w|_{\widetilde{b}} = n$, $|\widetilde{w}|_a + |\widetilde{w}|_{\widetilde{b}} = n$ and $|\widetilde{w}|_{\widetilde{c}} = m$.

Next consider two distinct elements of S, $(a^i b^j c^k, a^{n-i} b^{n-j} c^{m-k})$ and $(a^r b^s c^t, a^{n-r} b^{n-s} c^{m-t})$, where $(i, j, k) \neq (r, s, t)$. Denote $w = a^i b^j c^k \cdot a^{n-r} b^{n-s} c^{m-t}$. If $k \neq t$, $w \notin \mathrm{PQ}(K, L)$ because $|\widetilde{w}|_{\widetilde{c}} \not\equiv 0 \mod m$. If $i \neq r$ then $|w|_a + |w|_{\widetilde{b}} = i + n - r \not\equiv 0 \mod n$ and, consequently, $w\#\widetilde{w}^R \notin K$ and $w \notin \mathrm{PQ}(K, L)$. Similarly, if $j \neq s$ then $|\widetilde{w}|_a + |\widetilde{w}|_{\widetilde{b}} = j + n - s \not\equiv 0 \mod n$ and again $w\#\widetilde{w}^R \notin K$. □

6 The state complexity of the quotient

The results on the number of states in NIDPDA needed to represent the quotient are put together in the following theorem.

Theorem 2. *In order to represent the quotient of an m-state NIDPDA by an n-state NIDPDA, it is sufficient to use an NIDPDA with $3m + m^2 n$ states. In the worst case, it is necessary to use at least $m^2 n$ states.*

This gives the state complexity of $m^2 n + O(m)$.

If the goal is to construct a deterministic automaton, one possible solution is to determinize the constructed NIDPDA. However, that would produce as many as $2^{\Theta(m^4 n^2)}$ states. Previously, for some operations, such as the concatenation, a much more succinct direct construction of a DIDPDA was defined [14] using the idea of computing *behaviour functions* of the given DIDPDA [11]. Investigating whether there is a significantly better construction of a DIDPDA for a quotient of two DIDPDAs is left as an open problem. A possible starting point is the DFA state complexity of the *palindromic quotient* operation defined in this paper.

Another open problem concerns the state complexity of the quotient for the intermediate *unambiguous IDPDA* model [13].

Acknowledgement. The authors are grateful to the anonymous reviewers for many pertinent comments and suggestions; the implementation of some of them is deferred until the full version of this paper.

References

1. R. Alur, P. Madhusudan, "Visibly pushdown languages", *ACM Symposium on Theory of Computing* (STOC 2004, Chicago, USA, 13–16 June 2004), 202–211.
2. R. Alur, P. Madhusudan, "Adding nesting structure to words", *Journal of the ACM*, 56:3 (2009).
3. J.-C. Birget, "Intersection and union of regular languages and state complexity", *Information Processing Letters*, 43 (1992), 185–190.
4. B. von Braunmühl, R. Verbeek, "Input driven languages are recognized in $\log n$ space", *Annals of Discrete Mathematics*, 24 (1985), 1–20.
5. S. Ginsburg, S. A. Greibach, "Deterministic context-free languages", *Information and Control*, 9:6 (1966), 620–648.
6. S. Ginsburg, E. H. Spanier, "Quotients of context-free languages", *Journal of the ACM*, 10:4 (1963), 487–492.
7. Y.-S. Han, K. Salomaa, "Nondeterministic state complexity of nested word automata", *Theoretical Computer Science*, 410 (2009), 2961–2971.
8. J. Hartmanis, "Context-free languages and Turing machine computations", *Proceedings of Symposia in Applied Mathematics*, Vol. 19, AMS, 1967, 42–51.
9. M. Latteux, B. Leguy, B. Ratoandromanana, "The family of one-counter languages is closed under quotient", *Acta Informatica*, 22:5 (1985), 579–588.
10. K. Mehlhorn, "Pebbling mountain ranges and its application to DCFL-recognition", *Automata, Languages and Programming* (ICALP 1980, Noordweijkerhout, The Netherlands, 14–18 July 1980), LNCS 85, 422-435.
11. A. Okhotin, "Input-driven languages are linear conjunctive", *Theoretical Computer Science*, 618 (2016), 52–71.
12. A. Okhotin, K. Salomaa, "Complexity of input-driven pushdown automata", *SIGACT News*, 45:2 (2014), 47–67.
13. A. Okhotin, K. Salomaa, "Descriptional complexity of unambiguous input-driven pushdown automata", *Theoretical Computer Science*, 566 (2015), 1–11.
14. A. Okhotin, K. Salomaa, "State complexity of operations on input-driven pushdown automata", *Journal of Computer and System Sciences*, 86 (2017), 207–228.
15. A. Okhotin, K. Salomaa, "Edit distance neighbourhoods of input-driven pushdown automata", *Computer Science in Russia* (CSR 2017, Kazan, Russia, 8–12 June 2017), LNCS 10304, to appear.
16. X. Piao, K. Salomaa, "Operational state complexity of nested word automata", *Theoretical Computer Science*, 410 (2009), 3290–3302.
17. K. Salomaa, "Limitations of lower bound methods for deterministic nested word automata", *Information and Computation*, 209 (2011), 580–589.
18. D. Wood, "A further note on top-down deterministic languages", *Computer Journal*, 14:4 (1971), 396–403.