Accepted Manuscript

Further closure properties of input-driven pushdown automata

Alexander Okhotin, Kai Salomaa

PII: S0304-3975(19)30283-X
DOI: <https://doi.org/10.1016/j.tcs.2019.04.006>
Reference: TCS 11988

To appear in: *Theoretical Computer Science*

Received date: 6 November 2018
Revised date: 9 April 2019
Accepted date: 16 April 2019

Please cite this article in press as: A. Okhotin, K. Salomaa, Further closure properties of input-driven pushdown automata, *Theoret. Comput. Sci.* (2019), <https://doi.org/10.1016/j.tcs.2019.04.006>

This is a PDF file of an unedited manuscript that has been accepted for publication. As a service to our customers we are providing this early version of the manuscript. The manuscript will undergo copyediting, typesetting, and review of the resulting proof before it is published in its final form. Please note that during the production process errors may be discovered which could affect the content, and all legal disclaimers that apply to the journal pertain.

Further closure properties of input-driven pushdown automata[☆]

Alexander Okhotin^a, Kai Salomaa^b

^a*St. Petersburg State University, 7/9 Universitetskaya nab., Saint Petersburg 199034, Russia*

^b*School of Computing, Queen's University, Kingston, Ontario K7L 2N8, Canada*

Abstract

The paper investigates the closure of the language family defined by input-driven pushdown automata (IDPDA) under the following operations: insertion $\text{ins}(L, K) = \{xyz \mid xz \in L, y \in K\}$, deletion $\text{del}(L, K) = \{xz \mid xyz \in L, y \in K\}$, square root $\sqrt{L} = \{w \mid ww \in L\}$, the first half $\frac{1}{2}L = \{u \mid \exists v : |u| = |v|, uv \in L\}$ and cyclic shift $\text{SHIFT}(L) = \{vu \mid uv \in L\}$. For K and L recognized by nondeterministic IDPDA, with m and n states, respectively, insertion requires exactly $mn + 2m$ states, as long as K is well-nested; deletion requires exactly $2n$ states, for well-nested K ; square root requires $n^3 - O(n^2)$ states, for well-nested L ; the well-nested subset of the first half is representable with $2^{O(n^2)}$ states; the well-nested subset of the cyclic shift requires exactly $2n^2$ states. Without the well-nestedness constraints, non-closure is established in each case.

Keywords: Input-driven automata, visibly pushdown automata, insertion, deletion, square root, proportional removals, cyclic shift

1. Introduction

Input-driven pushdown automata (IDPDA), also known under the name of *visibly pushdown automata*, are an important special class of pushdown automata, introduced by Mehlhorn [19] and later studied, in particular, by Alur and Madhusudan [1, 2]. In these automata, the input symbol determines whether the automaton should push a stack symbol, pop a stack symbol or leave the stack untouched. These symbols are called *left brackets*, *right brackets* and *neutral symbols*, and the symbol pushed at each left bracket is always popped when reading the corresponding right bracket. Input-driven automata are important as a model of hierarchically structured data, such as XML documents or computation traces for recursive procedure calls.

Input-driven automata exist in deterministic (DIDPDA) and nondeterministic (NIDPDA) variants, which, as shown by von Braunmühl and Verbeek [4], are equivalent in power. Input-driven automata are also notable for their appealing closure properties, which almost rival those of finite automata. For instance, they are closed under all Boolean operations, concatenation $K \cdot L$, Kleene star L^* , reversal L^R with left and right brackets interchanged, homomorphisms $h(L)$ that respect nesting [1, 24], quotient $K \cdot L^{-1}$ [26], and edit distance ℓ -neighbourhood E_ℓ [25]; for binary operations it is assumed that the argument automata share the same partition of the alphabet into classes. Besides the closure results, the descriptive complexity of these operations has also been estimated in the literature. For instance, the concatenation of an m -state and an n -state DIDPDA in the worst case requires a DIDPDA with $m2^{\Theta(n \log n)}$ states [24, 15], whereas both for the union and for the intersection its state complexity is $\Theta(mn)$ [28]. For NIDPDA, the state complexity of concatenation and of union is $m + n + O(1)$ [1], and for the intersection it is $\Theta(mn)$ [11]. Further state complexity results were established for an intermediate family of *unambiguous input-driven*

[☆]A preliminary version of this paper [27] was presented at DCFS 2018 held in Halifax, Canada, July 25–27, 2018. Supported by Russian Science Foundation, project 18-11-00100.

Email addresses: alexander.okhotin@spbu.ru (Alexander Okhotin), ksalomaa@cs.queensu.ca (Kai Salomaa)

	general case		well-nested case	
	DIDPDA	NIDPDA	DIDPDA	NIDPDA
$K \cup L$	$\Theta(mn)$ [28]	$m + n$ [11]	$\Theta(mn)$ [28]	$m + n$ [11]
$K \cap L$	$\Theta(mn)$ [28]	$\Theta(mn)$ [11]	$\Theta(mn)$ [28]	$\Theta(mn)$ [11]
$\bar{L}$	n	$2^{\Theta(n^2)}$ [23] (*)	n	$2^{\Theta(n^2)}$ [23] (*)
$K \cdot L$	$m \cdot n^n \cdot (O(1))^n$ [24, 15]	$\Theta(m + n)$	$(1 - o(1))mn^n$ [24, 15]	$\Theta(m + n)$
L^*	$n^n \cdot (O(1))^n$ [24, 15]	$\Theta(n)$	$n^n + O(2^n)$ [24, 15]	$\Theta(n)$
L^2	$n^n \cdot (O(1))^n$ [24, 15]	$\Theta(n)$	$n^{n+O(1)}$ [24, 15]	$\Theta(n)$
L^R	$(1 - o(1))n^{n+1}$ [24, 15]	$n + O(1)$	n^n [24, 15]	$n + O(1)$
$h(L)$ (**)	$2^{\Theta(n^2)}$ [24]	$\Theta(n)$ [1]	$2^{\Theta(n^2)}$ [24]	$\Theta(n)$ [1]
$K \cdot L^{-1}$	?	m^2n [26]	?	m^2n [26]
$E_\ell(L)$	$\Theta(n^{\ell+1})$ [25]	?	$\Theta(n^{\ell+1})$ [25]	?
$\text{ins}(L, K)$	non-closure		?	$mn + 2m$
$\text{del}(L, K)$	non-closure		?	$2n$
$\sqrt{L}$	non-closure		n^n	$n^3 - O(n^2)$
$\frac{1}{2}(L)$	non-closure		?	$\leq 2^{O(n^2)}$
$\text{SHIFT}(L)$	non-closure		?	$2n^2$

Table 1: Estimation of the number of states in an NIDPDA that is sufficient and in the worst case necessary to represent various operations on languages K , recognized by an m -state NIDPDA, and L , recognized by an n -state NIDPDA. (*) For the complementation on NIDPDA, an estimation of the number of states *plus* the number of stack symbols is given; no estimation of the number of states is known. (**) Only homomorphisms that respect nesting are considered.

automata (UIDPDA) [23]. For more details on input-driven automata and their complexity, the readers are directed to a fairly recent survey [22].

The purpose of this paper is to investigate the closure and the state complexity of input-driven pushdown automata with respect to several further operations on languages, which are fairly standard in theoretical research on formal languages—yet their application to IDPDA has not yet been considered.

The first operation, investigated in Section 3, is **insertion**, $\text{ins}(L, K) = \{xyz \mid xz \in L, y \in K\}$. For finite automata, the closure under this operation is folklore, and its precise state complexity has recently been determined by Han et al. [9]. For input-driven automata, the closure is currently known only for singleton K [25]. This paper demonstrates the closure under the assumption that K consists only of well-nested strings, and determines the worst-case number of states in an NIDPDA recognizing $\text{ins}(L, K)$ as $mn + 2m$, where m is the number of states in the NIDPDA for K , while the NIDPDA for L has n states. For unrestricted K , the non-closure is established.

Another related operation is **deletion**, $\text{del}(L, K) = \{xz \mid \exists y \in K : xyz \in L\}$. It is well-known that if L is regular, then the result is regular for an arbitrary K . The state complexity of this operation for deterministic finite automata (DFA) has recently been studied by Han et al. [10]. As shown in Section 4, when L has an NIDPDA with n states, $\text{del}(L, K)$ is representable with $2n$ states, as long as K consists only of well-nested strings; without this assumption, there is a non-closure result. Section 4 gives also a $2n$ lower bound for the number of states for an NIDPDA for $\text{del}(L, K)$ and the same lower bound applies for the nondeterministic state complexity of deletion on regular languages. The nondeterministic state complexity of deletion was not considered by Han et al. [10].

The third operation, studied in Section 5, is the **square root**, $\sqrt{L} = \{w \mid ww \in L\}$. Assuming that L consists of only well-nested strings and is recognized by an n -state DIDPDA, its square root can be represented by a DIDPDA with n^n states, and this bound is tight; this proof uses the *behaviour functions* of

Figure 1: A part of a computation of an IDPDA on a well-nested substring.

DIDPDA [21, 24] in the same way as in the similar result for deterministic finite automata (DFA) [18]. For NIDPDA, as well as for nondeterministic finite automata (NFA), it is shown that this operation requires at most n^3 states and at least $(n-1)(n-2)(n-3)$ states in the worst case. If L is not restricted to well-nested strings, a non-closure is established.

Section 6 considers the operation of taking the **first half** of all even-length strings in a language, $\frac{1}{2}L = \{u \mid \exists v : |u| = |v|, uv \in L\}$. This is a particular case of *proportional removals*, studied by Maslov [18] and by Domaratzki [5] for DFA, and by Goč et al. [7] for NFA. If L is recognized by an NIDPDA with n states, then the set of all well-nested strings in $\frac{1}{2}L$, referred to as *the well-nested subset of $\frac{1}{2}L$* , is recognized by an NIDPDA with $2^{O(n^2)}$ states. The whole language $\frac{1}{2}L$ need not be representable, hence there is again a non-closure.

The operation investigated in Section 7 is the **cyclic shift**, $\text{SHIFT}(L) = \{vu \mid uv \in L\}$: its state complexity for finite automata was studied by Maslov [18] and by Jirásková and Okhotin [14]. In the case of input-driven automata, for L recognized by an n -state NIDPDA, the well-nested subset of $\text{SHIFT}(L)$ is recognized by an NIDPDA with $2n^2$ states. Without the well-nestedness restriction, a non-closure is established.

The state complexity of taking **scattered substrings** for IDPDA is briefly discussed in Section 8.

2. Input-driven automata

A *deterministic input-driven pushdown automaton* (DIDPDA) [1, 19] is a special case of a deterministic pushdown automaton, in which the input alphabet Σ 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 angled brackets, respectively ($<$, $>$), whereas lower-case Latin letters from the beginning of the alphabet ($a, b, c, \dots$) shall be used for symbols from Σ_0 .

A *nondeterministic input-driven pushdown automaton* (NIDPDA) [1, 4] is a similarly restricted case of a nondeterministic pushdown automaton, in which the type of the action on the stack is determined by the input symbol, whereas the actual next state and the symbol pushed onto the stack may be selected nondeterministically.

Input-driven automata are often restricted to 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 exactly 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, as illustrated in Figure 1.

Alur and Madhusudan [1] also considered input-driven automata operating on potentially ill-nested input strings. 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. 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.

Definition 1 (von Braunmühl and Verbeek [4]; Alur and Madhusudan [1]). A nondeterministic input-driven pushdown automaton (NIDPDA) over an alphabet $\tilde{\Sigma} = (\Sigma_{+1}, \Sigma_{-1}, \Sigma_0)$ 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 $\perp \notin \Gamma$ for the empty stack;
- for a neutral symbol $c \in \Sigma_0$, a transition function $\delta_c: Q \rightarrow 2^Q$ 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 $\delta_{<}: Q \rightarrow 2^{Q \times \Gamma}$, which, for a given current state, provides a set of pairs (q, γ) , with $q \in Q$ and $\gamma \in \Gamma$, where each pair means that the automaton enters the state q and pushes γ onto the stack;
- for every right bracket symbol $> \in \Sigma_{-1}$, there is a function $\delta_{>}: Q \times (\Gamma \cup \{\perp\}) \rightarrow 2^Q$ 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.

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

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 $x \in \Gamma^*$ (for well-nested inputs, the stack is empty at this point).

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

Von Braunmühl and Verbeek [4] proved that every n -state NIDPDA operating on well-nested strings can be transformed to a 2^{n^2} -state DIDPDA. Alur and Madhusudan [1] extended this construction to allow ill-nested inputs, so that a DIDPDA has 2^{2n^2} states; in the worst case, $2^{\Omega(n^2)}$ states are necessary.

The known closure results under Boolean operations, concatenation, star, reversal, quotient and edit distance neighbourhood are all valid for input-driven automata operating on ill-nested strings.

3. Insertion

The insertion operation is a binary operation on languages: $\text{ins}(L, K)$ is the set of all strings obtained by taking any string from L and any position in this string, and inserting any string from K at that position.

$$\text{ins}(L, K) = \{xyz \mid xz \in L, y \in K\}$$

The question is: when both K and L are recognized by IDPDA, shall $\text{ins}(L, K)$ always be recognized by an IDPDA? Provided that K does not contain any ill-nested strings, the answer is positive, established by the following effective construction.

Lemma 1. Let L be a language recognized by an NIDPDA $\mathcal{B}$ with the set of states Q and with a stack alphabet Γ . Let K be a set of well-nested strings, which is recognized by an NIDPDA $\mathcal{A}$ with the set of states P and with a stack alphabet Θ . Then, the language $\text{ins}(L, K)$ is recognized by an NIDPDA with the set of states $Q \cup \tilde{Q} \cup (P \times Q)$, where $\tilde{Q} = \{\tilde{q} \mid q \in Q\}$, and with the stack alphabet $\Gamma \cup \Theta$.

Figure 2: Construction for insertion in Lemma 1: (left) a well-nested string accepted by $\mathcal{A}$ inserted into a string accepted by $\mathcal{B}$; (right) an accepting computation of $\mathcal{C}$ on the resulting string.

Proof. Let $P_0 \subseteq P$ and $Q_0 \subseteq Q$ be the sets of initial states of $\mathcal{A}$ and $\mathcal{B}$, respectively. Their transition functions are: $\eta_c: P \rightarrow 2^P$ and $\delta_c: Q \rightarrow 2^Q$ for each neutral symbol $c \in \Sigma_0$; $\eta_{<}: P \rightarrow 2^{P \times \Theta}$ and $\delta_{<}: Q \rightarrow 2^{Q \times \Gamma}$, for each left bracket $< \in \Sigma_{+1}$; $\eta_{>}: P \times \Gamma \rightarrow 2^P$ and $\delta_{>}: Q \times \Omega \rightarrow 2^Q$ for each right bracket $> \in \Sigma_{-1}$. Let $E \subseteq P$ and $F \subseteq Q$ be the sets of accepting states of these automata.

The new automaton $\mathcal{C}$ begins in any state $\tilde{q}_0$, where $q_0 \in Q_0$ is any initial state of $\mathcal{A}$, and proceeds with simulating the computation of $\mathcal{B}$ in the states $\tilde{Q}$. Its transition functions, denoted by δ'_a , with $a \in \Sigma_{+1} \cup \Sigma_0 \cup \Sigma_{-1}$, are defined below.

At any moment, while in a state $\tilde{q} \in \tilde{Q}$, it may nondeterministically guess that a substring accepted by $\mathcal{A}$ begins at the next symbol. If the next symbol is $c \in \Sigma_0$, the new automaton may make a transition from $\tilde{q}$ by c to a pair (p, q) , where p is any state, to which $\mathcal{A}$ may go from any of its initial states by reading c .

$$\delta'_c(\tilde{q}) = \{ \tilde{r} \mid r \in \delta_c(q) \} \cup \{ (p, q) \mid p \in \eta_c(p_0) \text{ for some } p_0 \in P_0 \}$$

If the next symbol is a left bracket $< \in \Sigma_{+1}$, the new automaton may choose to go from $\tilde{q}$ by the left bracket $<$ to a state $(p, q) \in P \times Q$, pushing s , where $\mathcal{A}$ may go from p_0 to p by this bracket ($<$), pushing s .

$$\delta'_{<}(\tilde{q}) = \{ (\tilde{r}, t) \mid (r, t) \in \delta_{<}(q) \} \cup \{ ((p, q), s) \mid (p, s) \in \eta_{<}(p_0) \text{ for some } p_0 \in P_0 \}$$

When the new automaton reads a right bracket $> \in \Sigma_{-1}$, since no string in K begins with this symbol, the inserted substring may not begin at this point, and thus the new automaton continues simulating $\mathcal{B}$.

$$\delta'_{>}(\tilde{q}, t) = \{ \tilde{r} \mid r \in \delta_{>}(q, t) \}$$

Once the simulation of $\mathcal{A}$ on a substring begins, it proceeds in the states from $P \times Q$, with the simulation carried out in the first component, whereas the state of $\mathcal{B}$ remembered in the second component remains unchanged. Whenever the simulating automaton is in a state (p, q) with p accepting, it may decide that the well-nested substring being inserted has now ended, and accordingly simulate $\mathcal{B}$'s transition from the state q on the next input symbol.

$$\begin{aligned} \delta'_c((p, q)) &= \{ (p', q) \mid p' \in \eta_c(p) \} \cup \{ q' \mid p \in E, q' \in \delta_c(q) \} \\ \delta'_{<}((p, q)) &= \{ ((p', q), s) \mid (p', s) \in \eta_{<}(p) \} \cup \{ (q', t) \mid p \in E, (q', t) \in \delta_{<}(q) \} \end{aligned}$$

If the symbol being read is a right bracket, then the decision to continue simulating the computation of $\mathcal{A}$ is determined by the type of the stack symbol—that is, whether it is $\mathcal{A}$'s or $\mathcal{B}$'s.

$$\delta'_{>}(p, q, s) = \begin{cases} \{ (p', q) \mid p' \in \eta_{>}(p, s) \}, & \text{if } s \in \Theta \\ \{ q' \mid p \in E, q' \in \delta_{>}(q, s) \}, & \text{if } s \in \Gamma \end{cases}$$

Once the new automaton is in a state from Q , the simulation of $\mathcal{B}$ continues in these states. The computation is illustrated in Figure 2.

The accepting states of the new automaton are the same as in $\mathcal{B}$. If $\mathcal{A}$ accepts the empty string, then all states $\tilde{q}$, with q accepting in $\mathcal{B}$, are also marked as accepting, thus handling the case of inserting ε . $\square$

This construction is optimal in the worst case, which follows from the recent result by Han et al. [9] for NFA. Han et al. [9] constructed a pair of witness languages, K and L , recognized by an m -state and by an n -state NFA, respectively, and proved that every NFA recognizing the language $\text{ins}(L, K)$ must have at least $mn + 2m$ states. Assuming that the partition of the alphabet is fixed, with all symbols in Σ_0 , and with no brackets, an NIDPDA cannot do anything more than an NFA, and for that reason, the lower bound by Han et al. [9] also applies to NIDPDA. This yields the following result.

Theorem 1. *The state complexity of inserting a language recognized by an m -state NIDPDA into a language of well-nested strings recognized by an n -state NIDPDA is exactly $mn + 2m$.*

On the other hand, if the strings being inserted are not required to be well-nested, then the closure property no longer holds, as shown in the next example.

Example 1. The following language is recognized by an input-driven automaton.

$$L = \{ \langle^n \rangle^n \mid n \geq 1 \}$$

However, no input-driven automaton recognizes the language $\text{ins}(L, \langle \rangle^* \langle \rangle^*)$.

Proof. If $\text{ins}(L, \langle \rangle^* \langle \rangle^*)$ is recognized by some input-driven automaton, then so is its intersection with the regular language $\langle^+ \rangle^+ \langle^+ \rangle^+$. The intersection ensures that the insertion is made exactly in the middle of a string $\langle^n \rangle^n$.

$$\text{ins}(L, \langle \rangle^* \langle \rangle^*) \cap \langle^+ \rangle^+ \langle^+ \rangle^+ = \{ \langle^{n+1} \rangle^i \langle^j \rangle^{n+1} \mid n, i, j \geq 1 \}$$

Let the latter intersection be recognized by a DIDPDA with a set of states Q . Let $n = |Q|^2 + 1$, and consider the accepting computation on the string $\langle^n \rangle^n \langle^n \rangle^n$. In the first half of this computation, that is, on the prefix $\langle^n \rangle^n$, for every $i \in \{1, \dots, n\}$, let p_i and q_i be the states reached after reading $\langle^i$ and $\rangle^{n-i}$, respectively. When the automaton reads the $(i+1)$ -th left bracket in the state p_i , it pushes the stack symbol $s_i = \gamma_{<}(p_i)$, which is popped when reading the $(n-i)$ -th right bracket in the state q_{j+1} , so that $q_j = \delta_{>}(q_{j+1}, s_i)$. There are $|Q|^2$ different pairs (p_i, q_i) , and therefore, for some i and j , these pairs coincide: $(p_i, q_i) = (p_j, q_j)$, with $i < j$. Then the two segments of computation between p_{i+1} and p_j and between q_j and q_{i+1} , along with the stack symbols linking them to each other, can be cut, obtaining an accepting computation on the string $\langle^{n+j-i} \rangle^{n+j-i} \langle^n \rangle^n$, which does not belong to the language. $\square$

4. Deletion

The deletion operation is a binary operation on languages: $\text{del}(L, K)$ is the set of all strings obtained by taking any string from L and removing any substring belonging to K from that string.

$$\text{del}(L, K) = \{ xz \mid \exists y \in K : xyz \in L \}$$

If the set of strings being deleted consists of well-nested strings, there is the following closure result.

Figure 3: Construction for deletion in Lemma 2: (left) a string accepted by $\mathcal{B}$ containing a well-nested substring accepted by $\mathcal{A}$; (right) an accepting computation of $\mathcal{C}$ on the string with the substring deleted.

Lemma 2. Let a language L be recognized by an NIDPDA $\mathcal{B}$ with states Q and stack alphabet Γ . Let K be a language containing only well-nested strings, recognized by an NIDPDA $\mathcal{A}$. Then, the language $\text{del}(L, K)$ is recognized by an NIDPDA $\mathcal{C}$ with the set of states $Q \cup \tilde{Q}$, where $\tilde{Q} = \{\tilde{q} \mid q \in Q\}$, and with the stack alphabet Γ .

Proof. The automaton $\mathcal{C}$ recognizing $\text{del}(L, K)$ shall simulate the computation of $\mathcal{B}$ on xyz while reading an input string xz . After reading x , it guesses that a string y accepted by $\mathcal{A}$ has been deleted beginning from the next position, and adjusts its ongoing simulation of $\mathcal{B}$ to take the effect of this imaginary string into account.

In order to do this, some information needs to be calculated beforehand. For every pair of states $q, q' \in Q$, let $\mathcal{B}_{q,q'}$ be an NIDPDA obtained from $\mathcal{B}$ by setting q as the only initial state and q' as the only accepting state. Next, one can construct an NIDPDA $\mathcal{D}_{q,q'}$ recognizing the intersection $L(\mathcal{A}) \cap L(\mathcal{B}_{q,q'})$: this is the set of all such strings $y_{q,q'}$, that if the simulation of $\mathcal{B}$ decides to skip a deleted substring $y_{q,q'}$ while in a state q , then it can instantly move to the state q' . The form of these strings is inessential: it is only important to know, for each pair (q, q') , whether such a string $y_{q,q'}$ exists, so that the possibility to jump from q to q' could be embedded in the automaton $\mathcal{C}$. The existence of such a string is effectively determined by applying the known emptiness test [2] to $\mathcal{D}_{q,q'}$.

Let $Q_0 \subseteq Q$ be the set of initial states of $\mathcal{B}$, and let its transition functions be $\delta_c: Q \rightarrow 2^Q$ for neutral symbols $c \in \Sigma_0$, $\delta_<: Q \rightarrow 2^{Q \times \Gamma}$ for left brackets $< \in \Sigma_{+1}$, and $\delta_>: Q \times \Omega \rightarrow 2^Q$ for right brackets $> \in \Sigma_{-1}$. Let $F \subseteq Q$ be its set of accepting states.

The computation of the new automaton $\mathcal{C}$, illustrated in Figure 3, proceeds as follows. In the beginning, it simulates $\mathcal{B}$ in the states from $\tilde{Q}$. All states $\tilde{q}_0$, with $q_0 \in Q_0$, are marked as initial. At some point, while in some state $\tilde{q}$, it may guess that the erased substring began at this point. Let $q' \in Q$ be any such state of $\mathcal{B}$, that there exists a well-nested string $y_{q,q'}$, which is accepted by $\mathcal{A}$, whereas $\mathcal{B}$ may read $y_{q,q'}$ beginning from the state q , and finish reading it in the state q' . Then the simulating automaton may guess such a state q' and continue its simulation, as if it is currently in the state q' .

$$\delta'_c(\tilde{q}) = \{\tilde{r} \mid r \in \delta_c(q)\} \cup \{r \mid r \in \delta_c(q'), \text{ for some } q' \in Q \text{ with } L(\mathcal{A}) \cap L(\mathcal{B}_{q,q'}) \neq \emptyset\}$$

$$\delta'_<(\tilde{q}) = \{(\tilde{r}, t) \mid (r, t) \in \delta_<(q)\} \cup \{(r, t) \mid (r, t) \in \delta_<(q'), \text{ for some } q' \in Q \text{ with } L(\mathcal{A}) \cap L(\mathcal{B}_{q,q'}) \neq \emptyset\}$$

$$\delta'_>((\tilde{q}, t)) = \{\tilde{r} \mid r \in \delta_>(q, t)\} \cup \{r \mid r \in \delta_>(q', t), \text{ for some } q' \in Q \text{ with } L(\mathcal{A}) \cap L(\mathcal{B}_{q,q'}) \neq \emptyset\}$$

All accepting states of $\mathcal{B}$ are accepting in the new automaton. Furthermore, a state $\tilde{q}$ is accepting, if there is a string accepted by $\mathcal{A}$, as well as by $\mathcal{B}$ from the state q .

$$F' = F \cup \{\tilde{q} \mid \exists q' \in F \text{ with } L(\mathcal{A}) \cap L(\mathcal{B}_{q,q'}) \neq \emptyset\}$$

□

Ordinary NFAs recognize well-nested languages and NFAs provide a lower bound construction matching the upper bound of Lemma 2. The proof of the lower bound is based on the standard fooling set method [3]. Recall that a set of pairs of strings $\{(u_1, v_1), \dots, (u_m, v_m)\}$ is a *fooling set* for a language L if (i) for all $1 \leq i \leq m$, $u_i v_i \in L$, and, (ii) for all $1 \leq i, j \leq m$, $i \neq j$ implies $u_i v_j \notin L$ or $u_j v_i \notin L$. The size of a fooling set for a regular language L gives a lower bound for the number of states of any NFA recognizing L .

Lemma 3. *For $n \in \mathbb{N}$ there exists an NFA $\mathcal{B}$ with n states over a binary alphabet $\Sigma = \{a, b\}$ and a finite language $K \subseteq \Sigma^*$ such that any NFA for $\text{del}(L(\mathcal{B}), K)$ needs at least $2n$ states.*

Proof. The proof is given for the case where $n = 2k$ is even. The case of odd n is similar.

Consider $L = ((ab)^k)^*$ and $K = \{a, b\}$. The language L has an NFA with $2k = n$ states. Define

$$\mathcal{F} = \mathcal{F}_1 \cup \mathcal{F}_2 \cup \mathcal{F}_3 \cup \mathcal{F}_4,$$

where

$$\begin{aligned} \mathcal{F}_1 &= \{ ((ab)^i, (ab)^{2k-i-1}b) \mid 0 \leq i < k \}, \\ \mathcal{F}_2 &= \{ ((ab)^i a, b(ab)^{2k-i-2}a) \mid 0 \leq i < k \}, \\ \mathcal{F}_3 &= \{ (b(ab)^{i-1}a, b(ab)^{2k-i-1}) \mid 0 \leq i < k \}, \\ \mathcal{F}_4 &= \{ (a(ab)^{i+1}, (ab)^{2k-i-2}) \mid 0 \leq i < k \}. \end{aligned}$$

The claim is that $\mathcal{F}$ is a fooling set for $\text{del}(L, K)$. For each pair of strings $(u, v) \in \mathcal{F}$, their concatenation uv is a string of length $4k - 1$, which can be obtained by deleting one a or b from the string $(ab)^{2k}$; and, hence, $uv \in \text{del}(L, K)$.

Next for any two distinct pairs (u_1, v_1) and (u_2, v_2) in $\mathcal{F}$, it shall be proved that at least one of the mismatched concatenations $u_1 v_2$ and $u_2 v_1$ is substantially different from the strings in $(ab)^*$, which rules out obtaining it from any string in L by deleting a single symbol. Assume that $(u_1, v_1) \in \mathcal{F}_\ell$ and $(u_2, v_2) \in \mathcal{F}_h$, with $1 \leq \ell \leq h \leq 4$.

1. If $\ell = h$, then the length of $u_1 v_2$ (or of $u_2 v_1$) is not congruent to -1 modulo m , and hence $u_1 v_2, u_2 v_1 \notin \text{del}(L, K)$.
2. If $\ell = 1$ and $h \in \{2, 4\}$, then the string $u_1 v_2$ (as well as $u_2 v_1$) has even length and is not in $\text{del}(L, K)$.
3. Suppose that $\ell = 1$ and $h = 3$. Now the string $u_2 v_1$ begins with a b and ends with bb , and $u_2 v_1$ is not obtained by deleting one symbol from a string of L .
4. If $\ell = 2$ and $h = 3$, then the string $u_1 v_2$ has even length and is not in $\text{del}(L, K)$.
5. If $\ell = 2$ and $h = 4$, then the string $u_2 v_1$ has a substring aa and a substring bb , and cannot be obtained from a string of L by deleting one symbol.
6. If $\ell = 3$ and $h = 4$, then the string $u_1 v_2$ begins with a b and has a substring aa , which again is impossible.

This confirms that $\mathcal{F}$ is a fooling set for $\text{del}(L, K)$, and therefore any NFA for $\text{del}(L, K)$ needs at least $|\mathcal{F}| = 4k$ states. □

Lemma 2 and Lemma 3 yield the following result. The proof of Lemma 3 implies also that the non-deterministic state complexity of deletion on regular languages is exactly $2n$.

Theorem 2. *The state complexity of deleting a well-nested language K from a language recognized by an n -state NIDPDA $\mathcal{B}$ is exactly $2n$. The lower bound is reached using a fixed language K consisting of two strings. If K is recognized by an NIDPDA, then the construction of an NIDPDA for the language $\text{del}(L(\mathcal{B}), K)$ is effective.*

If ill-nested strings may be deleted, then, as in the case of insertion, the family is no longer closed under this operation.

Example 2. The following language is recognized by an input-driven automaton.

$$L = \{ \langle^m \ll^n \gg^n \rangle^m \mid m, n \geq 1 \}$$

However, no input-driven automaton recognizes the language $\text{del}(L, \ll^*)$.

Proof. If the language $\text{del}(L, \ll^*)$ is recognized by some input-driven automaton, then so is its intersection with the regular language $\langle^+ \gg^+ \rangle^+$. This intersection ensures that *all* double left brackets ($\ll$) are erased.

$$\text{del}(L, \ll^*) \cap \langle^+ \gg^+ \rangle^+ = \{ \langle^m \gg^i \rangle^m \mid m, i \geq 1 \}$$

If the latter intersection is recognized by a DIDPDA with a set of states Q , then let $m = |Q|$. Then, in the computation on $\langle^m \gg^m \rangle^m$, the stack must be empty after reading the prefix $\langle^m \gg^m$. In the last part of the computation, while reading the suffix $\rangle^m$, the automaton passes through a sequence of states $p_0, \dots, p_m$, behaving like a DFA, with every next state determined by a transition with the empty stack as $p_{i+1} = \delta_{\triangleright}(p_i, \perp)$. Since $m + 1 > |Q|$, two of these states must coincide: $p_i = p_j$, with $0 \leq i < j \leq m$. Then this segment of the computation can be cut out without affecting the acceptance, and the automaton accepts the string $\langle^m \gg^m \rangle^{m-(j-i)}$, which is not in the language. $\square$

5. Square root

For a string of the form ww , the string w is its *square root*, denoted by $\sqrt{ww} = w$. The square root of a language is defined as the set of square roots of all its applicable elements.

$$\sqrt{L} = \{ w \mid ww \in L \}$$

The regular languages are closed under this operation. Indeed, for every n -state DFA recognizing L , one can construct a DFA with n^n states that computes the *behaviour function* of the original DFA on the string w : this is a function $f_w: Q \rightarrow Q$ that maps each state $q \in Q$ to the state reached by the DFA after reading w beginning from the state q . Denote by Q^Q the set of all such functions. Then, the set of states of the constructed DFA is Q^Q .

This construction generalizes to input-driven automata, under the assumption that L contains only well-nested strings. Notably, in such a case, the language $\sqrt{L}$ also contains only well-nested strings.

Lemma 4. *If a language L contains only well-nested strings, and is recognized by a DIDPDA with the set of states Q , then the language $\sqrt{L}$ is recognized by a DIDPDA with the set of states Q^Q and with the stack alphabet $\Sigma_{+1} \times Q^Q$.*

The new DIDPDA processing w constructs the *behaviour function* [21, 24] of the original DIDPDA on w . This is one of the basic constructions for DIDPDA, based on the following observation: when a DIDPDA with a set of states Q processes a well-nested string w and begins in a state q , it finishes reading that string in some state $f_w(q)$, where $f_w: Q \rightarrow Q$ is its *behaviour function on w* , and the stack is left untouched. Thus, f completely characterizes the behaviour of a DIDPDA on w . For any given DIDPDA $\mathcal{A}$, it is possible to construct an n^n -state DIDPDA, where $n = |Q|$, that reaches the end of an input w in a state representing the behaviour of $\mathcal{A}$ on the longest well-nested suffix of w . This construction is necessary for optimal constructions representing operations on DIDPDA [24].

Proof of Lemma 4. This is a known DIDPDA construction for the behaviour function $f_w: Q \rightarrow Q$ of $\mathcal{A}$ on the input string w . Then the function $f_w \circ f_w$ is its behaviour of $\mathcal{A}$ on ww , and the simulating automaton accepts in a state $f \in Q^Q$ whenever $(f \circ f)(q_0) \in F$. $\square$

A matching lower bound for the state complexity of the square root on DFA is known. It immediately applies to DIDPDA, showing that n^n is the exact complexity of the square root for this model as well.

Lemma 5 (Maslov [18]). *For every $n \geq 1$, there exists such an n -state DFA over a 3-symbol alphabet (equivalently, an n -state DIDPDA over the alphabet $\Sigma_{+1} = \Sigma_{-1} = \emptyset$, $\Sigma_0 = \{a, b, c\}$) that every DFA (as well as a DIDPDA) recognizing the language $\sqrt{L}$ requires at least n^n states.*

For NIDPDA, the state complexity of the square root is quite different. As to the authors' knowledge, the number of states in an NFA recognizing the square root of an n -state NFA is not yet known, and it is natural to establish it first.

Theorem 3. *Square root of an n -state NFA is recognized by an n^3 -state NFA.*

For every n , there is a language L_n over a three-symbol alphabet, which is recognized by an n -state DFA, whereas every NFA recognizing $\sqrt{L_n}$ must have at least $(n-1)(n-2)(n-3)$ states.

Proof. Let $\mathcal{A} = (\Sigma, Q, Q_0, \delta, F)$ be any n -state NFA. The NFA $\mathcal{B}$ recognizing $\sqrt{L(\mathcal{A})}$ uses the set of triples $Q \times Q \times Q$ as its states. In the beginning of its computation on a string w , it guesses the state p reached by $\mathcal{A}$ after reading w , remembers this state as the last component on the triple, and begins simulating two computations of $\mathcal{A}$, one beginning from an initial state and the other beginning from p . Accordingly, its set of initial states is $\{(q_0, p, p) \mid q_0 \in Q_0, p \in Q\}$, and it uses the following transitions.

$$\delta'((q, r, p), a) = \{(q', r', p) \mid q' \in \delta(q, a), r' \in \delta(r, a)\}$$

When $\mathcal{B}$ finishes reading w in a state (q, r, p) , it has verified that $\mathcal{A}$, upon reading w , can move from q_0 to q , as well as from p to r . If $q = p$, this confirms that $\mathcal{A}$ can move from q_0 to r upon reading w , and if r is accepting, then $\mathcal{B}$ should accept its input string w . Accordingly, the set of accepting states of $\mathcal{B}$ is $\{(p, r, p) \mid p \in Q, r \in F\}$.

Turning to the lower bound, the language L_n is the standard “universal” witness language, as in Lemma 5. It is recognized by a DFA with the states $\{0, \dots, n-1\}$, and has the following property: for every function $f: \{0, \dots, n-1\} \rightarrow \{0, \dots, n-1\}$, there is such a string x_f , that, upon reading x_f beginning from a state i , the DFA finishes reading it in the state $f(i)$. The state 0 is initial and $n-2$ is accepting.

Now the lower bound is proved using the standard *fooling set* method. Let (i, j, k) be a triple of states, with $i, k \in \{0, \dots, n-2\}$, $j \in \{1, \dots, n-3\}$ and $i \neq k$. Define the string $u_{i,j,k} = x_f$, where f is a function defined by $f(0) = i$, $f(j) = k$ and $f(t) = n-1$ for all remaining arguments. The other string is $v_{i,j,k} = x_g$, where $g(i) = j$, $g(k) = n-2$ and $g(t) = n-1$ for the rest of the arguments. Then, the concatenation $u_{i,j,k}v_{i,j,k}$ maps the initial state to j , and j to the accepting state. This means that the square $(u_{i,j,k}v_{i,j,k})^2$ maps the initial state to the accepting state, and therefore $u_{i,j,k}v_{i,j,k}$ belongs to $\sqrt{L(\mathcal{A})}$.

On the other hand, for any two different pairs on this list, $(u_{i,j,k}, v_{i,j,k})$ and $(u_{i',j',k'}, v_{i',j',k'})$, at least one of the mismatched concatenations $u_{i,j,k}v_{i',j',k'}$ and $u_{i',j',k'}v_{i,j,k}$ is not in $\sqrt{L(\mathcal{A})}$. It turns out that neither concatenation is actually in $\sqrt{L(\mathcal{A})}$. Thus, the goal is to prove that $w = u_{i,j,k}v_{i',j',k'}u_{i,j,k}v_{i',j',k'}$ is not in $L(\mathcal{A})$. This is established by the following case study.

Case I: $i \neq i'$ and $i \neq k'$. Then $\mathcal{A}$ rejects the string w as follows: it begins in the state 0, after reading $u_{i,j,k}$ it is in the state i , and then, upon reading $v_{i',j',k'}$, it enters the dead state $n-1$, in which it remains until the end of the computation.

Case II: $i \neq i'$ and $i = k'$. In this case, $\mathcal{A}$ also rejects the string w , this time in the following computation: from 0 to i by $u_{i,j,k}$, then to $n-2$ by $v_{i',j',k'}$, and next, to $n-1$ by $u_{i,j,k}$, where it remains until the end.

Case III: $i = i'$ and $j \neq j'$. Then $\mathcal{A}$ rejects the string w as follows: from 0 to i by $u_{i,j,k}$, then to j' by $v_{i',j',k'}$, from whence it proceeds to $n-1$ by $u_{i,j,k}$, because $j' \notin \{0, j\}$.

Case IV: $i = i'$, $j = j'$ and $k \neq k'$. Once again the string w is rejected, as follows: from 0 to i by $u_{i,j,k}$, then to j by $v_{i',j',k'}$, to k by $u_{i,j,k}$. Finally, since $k \neq k'$, upon reading the last substring $v_{i',j',k'}$, the automaton proceeds from k either to j or to $n-1$. Since neither state is $n-2$, the string is rejected.

Then, every NFA recognizing the square root must have at least as many states as there are pairs in this fooling set, and there are $(n-1)(n-2)(n-3)$ such pairs. $\square$

The construction for NFA easily generalizes to the input-driven case: a simulating NIDPDA guesses a middle state and then simulates two computations of the original NIDPDA on the same well-nested input w . Each simulated computation uses its own stack symbols, which are pushed in pairs.

The lower bound in Theorem 3 applies to NIDPDA as it is.

Theorem 4. *Assume that a language L contains only well-nested strings, and let it be recognized by an NIDPDA with set of states Q , and with stack alphabet Γ . Then the language $\sqrt{L}$ is recognized by an NIDPDA with the set of states Q^3 and with the stack alphabet Γ^2 .*

For every n , there is a language L_n over an alphabet that consists of four neutral symbols, which is recognized by an n -state DFA, whereas every NIDPDA recognizing $\sqrt{L_n}$ must have at least $(n-1)(n-2)(n-3)$ states.

Thus, the state complexity of the square root has been established as $n^3 - O(n^2)$ both for NFA and for NIDPDA. No tight bound is known in either case. For the case of NIDPDA, the construction relies on all strings' being well-nested.

In the general case without the well-nestedness assumption, input-driven automata again demonstrate a non-closure.

Example 3. The following language is recognized by an input-driven automaton.

$$L = \{ >^i <^n >^n <^j \mid i, n, j \geq 1 \} \cup \{ <^m >^m <^n >^n \mid m, n \geq 1 \}$$

Its square root has the following form.

$$\sqrt{L} = \{ >^n <^n \mid n \geq 1 \} \cup \{ <^n >^n \mid n \geq 1 \}$$

It is not recognized by any input-driven automaton, for any partition of the alphabet.

6. Propotional removals

For a string of even length, $w = a_1 \dots a_{2\ell}$, its *first half*, denoted by $\frac{1}{2}w$, is the string $a_1 \dots a_\ell$, obtained by discarding the second half of w . This operation is extended to languages element-wise, as follows.

$$\frac{1}{2}(L) = \{ \frac{1}{2}w \mid w \in L, |w| \text{ is even} \}$$

As reported by Seiferas and McNaughton [30], Yamada, Stearns and Hartmanis were the first to prove that the regular languages are closed under this operation. Maslov [18] determined its state complexity for DFA as $2^{\Theta(\sqrt{n} \log n)}$, relying on the known determinization of unary NFAs using $e^{(1+o(1))\sqrt{n} \ln n}$ states. Later Domaratzki [5] further investigated its state complexity, and Goč et al. [7] proved that for NFA, the state complexity of “one half” is $\Theta(n^2)$.

For input-driven automata, there is a construction somewhat similar to those used for finite automata. Unfortunately, it cannot anymore rely on determinizing unary NFA, and for that reason is much less efficient with respect to the number of states.

Lemma 6. *For an n -state NIDPDA recognizing a language L , the well-nested subset of the language $\frac{1}{2}(L)$ is recognized by an NIDPDA with $2^{O(n^2)}$ states.*

This time no assumptions are made on the well-nestedness of strings in L , but the construction produces an automaton that recognizes the intersection of $\frac{1}{2}L$ with the set of well-nested strings.

Proof. Let $\mathcal{A}$ be the automaton that recognizes L , with the set of states Q , initial states $Q_0 \subseteq Q$, transition functions be $\delta_c: Q \rightarrow 2^Q$ (for $c \in \Sigma_0$), $\delta_<: Q \rightarrow 2^{Q \times \Gamma}$ (for $< \in \Sigma_{+1}$) and $\delta_>: Q \times \Omega \rightarrow 2^Q$ (for $> \in \Sigma_{-1}$), and accepting states $F \subseteq Q$.

Given an input string, the new automaton $\mathcal{B}$ has to check whether it is the first half of some string accepted by $\mathcal{A}$. It uses states of the form $(q, \hat{q}, p)$, where $q \in Q$ is the state of the ongoing simulation of $\mathcal{A}$

on the input string, $\hat{q} \in Q$ is the guessed state of $\mathcal{A}$ after reading the input string, whereas p is a state of a certain finite automaton defined below.

For each state $\hat{q} \in Q$ of $\mathcal{A}$, there exists an NFA $\mathcal{C}_{\hat{q}}$ that accepts a string a^ℓ if and only if there exists any string of length ℓ accepted by the input-driven automaton $\mathcal{A}$, beginning in the state $\hat{q}$ with the empty stack. Such a finite automaton exists by Parikh's theorem, and is effectively obtained as follows. First, the NIDPDA is transformed to a grammar of size $O(n^2)$. Then, the efficient construction for Parikh's theorem by Esparza et al. [6] is applied to that grammar, producing an NFA of size $2^{O(n^2)}$. Let $\eta_{\hat{q}}$ be the transition function of $\mathcal{C}_{\hat{q}}$.

Given the well-nested first half u of some string, the automaton $\mathcal{B}$ begins its computation in any state of the form $(q_0, \hat{q}, p_0^{(\hat{q})})$, where $q_0 \in Q_0$ is any initial state of the original automaton $\mathcal{A}$, $\hat{q} \in Q$ can be any of its states, and $p_0^{(\hat{q})} \in P$ is the initial state of $\mathcal{C}_{\hat{q}}$. Then, $\mathcal{B}$ simulates the behaviour of $\mathcal{A}$ on u , along with running $\mathcal{C}_{\hat{q}}$ on the unary string $a^{|u|}$, and preserves the second component $\hat{q}$ unchanged.

$$\begin{aligned}\delta_c((q, \hat{q}, p)) &= \{ (q', \hat{q}, p') \mid q' \in \delta_c(q), p' \in \eta_{\hat{q}}(p, a) \} \\ \delta_<((q, \hat{q}, p)) &= \{ ((q', \hat{q}, p'), s) \mid (q', s) \in \delta_<(q), p' \in \eta_{\hat{q}}(p, a) \} \\ \delta_>((q, \hat{q}, p), s) &= \{ (q', \hat{q}, p') \mid q' \in \delta_>(q, s), p' \in \eta_{\hat{q}}(p, a) \}\end{aligned}$$

In the end, $\mathcal{B}$ accepts if its first component reaches the state $\hat{q}$ guessed in the beginning, while the simulated $\mathcal{C}_{\hat{q}}$ accepts the string.

$$F' = \{ (q, q, p) \mid q \in Q, p \text{ is accepting} \}$$

Indeed, this guarantees that, on the one hand, $\mathcal{A}$ finishes reading the string u in the state q with the empty stack, and on the other hand, that $\mathcal{A}$ accepts some string v of length $|u|$ beginning in the state q with the empty stack. Since $uv \in L$ and $u = \frac{1}{2}uv$, this confirms that u is in $\frac{1}{2}L$. $\square$

The construction in Lemma 6 should be taken as a rough upper bound on the state complexity of the *one half* operation. Improving this construction is proposed as an open problem.

If the exact value of $\frac{1}{2}L$ is required, that is, without the intersection with the set of well-nested strings, then there is yet another non-closure result.

Example 4. The following language consists only of well-nested strings and is recognized by an input-driven automaton.

$$L = \{ <^m >^m <^n >^n (cc)^n cc \mid m, n \geq 1 \}$$

However, the language $\frac{1}{2}L$ is not recognized by any input-driven automaton, for any partition of the alphabet.

Proof. Each string is of length $2m + 4n + 2$, and its first half is of length $m + 2n + 1$. This first half belongs to the set $<^* >^* <^* c$ if and only if $m = n$. For this reason, the set of first halves of strings in L , under intersection with the regular language $<^* >^* <^* c$, has the following form.

$$\frac{1}{2}L \cap <^* >^* <^* c = \{ <^n >^n <^n c \mid n \geq 1 \}$$

The latter language is certainly not recognized by any input-driven automaton, and therefore neither is $\frac{1}{2}L$. $\square$

7. Cyclic shift

For every partition of a string w as $w = uv$, the string vu is a *cyclic shift* of w . The cyclic shift of a language L is the set of all cyclic shifts of all strings in L .

$$\text{SHIFT}(L) = \{ vu \mid uv \in L \}$$

The state complexity of this operation was first studied by Maslov [18], who proved that it requires $2^{n^2+n \log n - O(1)}$ states in the worst case, with witness languages defined over a growing alphabet. Later Jirásková and Okhotin [14] proved the same lower bound using a fixed 4-symbol alphabet, and also established the state complexity of the cyclic shift for NFA: it is exactly $2n^2$.

In the case of input-driven automata, it is possible to construct an automaton recognizing all well-nested strings in the cyclic shift.

Lemma 7. *Let L be recognized by an n -state NIDPDA with k stack symbols. Then, the well-nested subset of the language $\text{SHIFT}(L)$ is recognized by an NIDPDA with $2n^2$ states and $2k$ stack symbols.*

Proof. Let $\mathcal{A}$ be the original NIDPDA, with a set of states Q and with a stack alphabet Γ . The new NIDPDA $\mathcal{C}$ recognizing the cyclic shift uses the set of states $Q \times Q \times \{1, 2\}$, and the stack alphabet $\Gamma \times \{1, 2\}$.

In general, $\mathcal{C}$ works as follows. Let vu be a well-nested string, for which uv is in L . Then, $\mathcal{C}$ should read v in the states from $Q \times Q \times \{1\}$, simulating the computation of $\mathcal{A}$ on v , and then read u in the states from $Q \times Q \times \{2\}$, simulating $\mathcal{A}$ on u . If both u and v happen to be well-nested, then these two parts of the computation cannot interact through the stack, and everything proceeds as in the known construction in the case of NFA.

Indeed, let the accepting computation of $\mathcal{A}$ on uv begin in an initial state q_0 , then pass through a state $\hat{q}$ after reading u , and finally accept in a state q_{acc} . Then, $\mathcal{C}$ begins reading vu in a state $(\hat{q}, \hat{q}, 1)$, and simulates $\mathcal{A}$ in the first component, while keeping the second and the third components intact. At some point $\mathcal{C}$ is in a state $(q_{acc}, \hat{q}, 1)$, and it guesses that it has finished reading v . Then it enters the state $(q_0, \hat{q}, 2)$, and proceeds with simulating $\mathcal{A}$ on the first component, again preserving the second and the third components unchanged. It accepts in a state $(\hat{q}, \hat{q}, 2)$.

The actual construction has some further details necessary to handle potentially ill-nested u and v . The situation when u has an unmatched left bracket ($<$) or v has an unmatched right bracket ($>$) is impossible, because any such bracket would be unmatched in vu , which is impossible by assumption. However, the converse situation cannot be ruled out: v may have some unmatched left brackets ($<$), and then u has the same number of unmatched right brackets ($>$), so that these brackets match each other in the shifted string vu . Since both brackets are mismatched in uv , the new automaton $\mathcal{C}$ should treat them as mismatched in order to simulate $\mathcal{A}$ correctly. For that reason, whenever a simulated $\mathcal{A}$ pushes a stack symbol, $\mathcal{C}$ pushes the same stack symbol *labelled with the number of the phase*, and when this stack symbol is popped, it will know whether it was pushed in the current or in the previous phase.

Transitions by neutral symbols are not affected by these considerations: the simulation in the first component goes on, and the automaton may also decide to switch from the first phase to the second phase.

$$\begin{aligned} \delta'_c((q, \hat{q}, 1)) &= \{ (q', \hat{q}, 1) \mid q' \in \delta_c(q) \} \cup \underbrace{\{ (q', \hat{q}, 2) \mid q' \in \delta_c(q_0) \text{ for some } q_0 \in Q_0 \}}_{\text{if } q \in F} \\ \delta'_c((q, \hat{q}, 2)) &= \{ (q', \hat{q}, 2) \mid q' \in \delta_c(q) \} \end{aligned}$$

When a left bracket ($< \in \Sigma_{+1}$) is read, the number of the phase is pushed along with the stack symbol.

$$\begin{aligned} \delta'_{<}((q, \hat{q}, 1)) &= \{ ((q', \hat{q}, 1), (s, 1)) \mid (q', s) \in \delta_{<}(q) \} \cup \underbrace{\{ ((q', \hat{q}, 2), (s, 2)) \mid (q', s) \in \delta_{<}(q_0) \text{ for some } q_0 \in Q_0 \}}_{\text{if } q \in F} \\ \delta'_{<}((q, \hat{q}, 2)) &= \{ ((q', \hat{q}, 2), (s, 2)) \mid (q', s) \in \delta_{<}(q) \} \end{aligned}$$

Reading a right bracket ($> \in \Sigma_{-1}$) along with popping a stack symbol splits into several cases. In the first phase, only first-phase stack symbols can be popped; if the transition between phases occurs at this point, then the bracket should be treated as mismatched.

$$\delta'_{>}((q, \hat{q}, 1), (s, 1)) = \{ (q', \hat{q}, 1) \mid q' \in \delta_{>}(q, s) \} \cup \underbrace{\{ (q', \hat{q}, 2) \mid q' \in \delta_{>}(q_0, \perp) \text{ for some } q_0 \in Q_0 \}}_{\text{if } q \in F}$$

If a second-phase stack symbol is popped in the second phase, the simulation continues normally.

$$\delta'_>((q, \hat{q}, 2), (s, 2)) = \{ (q', \hat{q}, 2) \mid q' \in \delta_>(q, s) \}$$

Finally, a first-phase stack symbol popped in the second phase indicates that the bracketed is mismatched in the original string.

$$\delta'_>((q, \hat{q}, 2), (s, 1)) = \{ (q', \hat{q}, 2) \mid q' \in \delta_>(q, \perp) \}$$

□

A matching lower bound was given for NFA by Jirásková and Okhotin [14, Lem. 13] using a 2-symbol alphabet. Altogether, the following result has been established.

Theorem 5. *The state complexity of the cyclic shift on languages of well-nested strings recognized by n -state NIDPDA is exactly $2n^2$.*

As in the earlier results, an input-driven automaton is guaranteed to exist only for the well-nested subset of $\text{SHIFT}(L)$, whereas the whole language $\text{SHIFT}(L)$ need not be representable. This is illustrated in the following example.

Example 5. The following language consists only of well-nested strings and is recognized by an input-driven automaton.

$$L = \{ \langle^n \rangle^n \mid n \geq 1 \}$$

Its cyclic shift is not recognized by any input-driven automaton, under any partition of the alphabet, because its intersection with a regular language $\langle^* \rangle^* \cup \rangle^* \langle^*$ has the following form.

$$\text{SHIFT}(L) \cap (\langle^* \rangle^* \cup \rangle^* \langle^*) = \{ \langle^n \rangle^n \mid n \geq 1 \} \cup \{ \rangle^n \langle^n \mid n \geq 1 \}$$

8. Scattered substrings

For each string $w = a_1 \dots a_n$, any string $a_{i_1} \dots a_{i_\ell}$, with $1 \leq i_1 < \dots < i_\ell \leq n$, is its *scattered substring*. Denote by $\text{sub}(w)$ the set of all scattered substrings of w , and let $\text{sub}(L) = \bigcup_{w \in L} \text{sub}(w)$ for a language L .

Theorem 6. *The family of languages recognized by input-driven automata is closed under taking scattered substrings.*

Proof. By the Higman–Haines theorem [12, 8], the language $\text{sub}(L)$ is regular for an arbitrary language L . Therefore, it can be recognized by an IDPDA that pays no attention to the bracket structure and simulates a finite automaton recognizing $\text{sub}(L)$ while pushing and popping dummy stack symbols on the brackets. □

For regular languages, the state complexity of scattered substrings is $2^{\Theta(n)}$ [13, 20], with recent further results by Karandikar et al. [16].

What is the state complexity of this operation for IDPDA? As proved by van Leeuwen [17], for L given by a grammar, the language $\text{sub}(L)$ is *effectively* regular. However, van Leeuwen's [17] constructive proof does not include any estimation of the size of the regular language; it uses the finiteness of the basis to prove that the construction terminates. For that reason, no upper bound on the state complexity of scattered substrings for IDPDA is known.

9. Conclusion

The prospects of applying an operation on languages to input-driven automata mainly depend on how this operation affects the bracket structure of the strings. Boolean operations do not affect it at all, and are implemented in input-driven automata without any difficulties. Concatenation and Kleene star leave the strings as they are, and even though they may form new matchings between unmatched brackets in the strings being concatenated, they can still be implemented on input-driven automata [1, 24]. The recently studied edit distance neighbourhood operation affects a bounded number of symbols, and thus disrupts the bracket structure fairly lightly, so that the family of input-driven languages is closed under that operation.

In contrast, the operations studied in this paper can generally destroy the bracket structure entirely, leading to non-closure results. The contribution of this paper is that, under various assumptions on partial preservation of bracket structure, these operations can be implemented on input-driven automata, and the implementations are efficient in terms of state complexity. The results obtained can be listed among the positive closure properties of this language family.

A few theoretical questions are left open. First of all, for all operations except the square root, their state complexity for deterministic input-driven automata was not investigated; determining it precisely is a question for further research. In the case of scattered substrings, no state complexity results could be obtained at all; it would be interesting to establish any definite upper bounds on the complexity of this known construction.

- [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] M. Domaratzki, “State complexity of proportional removals”, *Journal of Automata, Languages and Combinatorics*, 7:4 (2002), 455–468.
- [6] J. Esparza, P. Ganty, S. Kiefer, M. Luttenberger, “Parikh’s theorem: A simple and direct automaton construction”, *Information Processing Letters*, 111:12 (2011), 614–619.
- [7] D. Goč, A. Palioudakis, K. Salomaa, “Nondeterministic state complexity of proportional removals”, *International Journal of Foundations of Computer Science*, 25:7 (2014), 823–836.
- [8] L. H. Haines, “On free monoids partially ordered by embedding”, *Journal of Combinatorial Theory*, 6 (1969), 94–98.
- [9] Y.-S. Han, S.-K. Ko, T. Ng, K. Salomaa, “State complexity of insertion”, *International Journal of Foundations of Computer Science*, 27:7 (2016), 863–878.
- [10] Y.-S. Han, S.-K. Ko, K. Salomaa, “State complexity of deletion and bipolar deletion”, *Acta Informatica*, 53:1 (2016), 67–85.
- [11] Y.-S. Han, K. Salomaa, “Nondeterministic state complexity of nested word automata”, *Theoretical Computer Science*, 410 (2009), 2961–2971.
- [12] G. Higman, “Ordering by divisibility in abstract algebras”, *Proceedings of London Mathematical Society*, s3-2:1 (1952), 326–336.
- [13] H. Gruber, M. Holzer, M. Kutrib, “More on the size of Higman–Haines sets: effective constructions”, *Fundamenta Informaticae*, 91:1 (2009), 105–121.
- [14] G. Jirásková, A. Okhotin, “State complexity of cyclic shift”, *RAIRO Informatique Théorique et Applications*, 42:2 (2008), 335–360.
- [15] G. Jirásková, A. Okhotin, “Towards exact state complexity bounds for input-driven pushdown automata”, *Developments in Language Theory (DLT 2018, Tokyo, Japan, 10–14 September 2018)*, LNCS 11088, 441–452.
- [16] P. Karandikar, M. Niewerth, Ph. Schnoebelen, “On the state complexity of closures and interiors of regular languages with subwords and superwords”, *Theoretical Computer Science*, 610:A (2016), 91–107.
- [17] J. van Leeuwen, “Effective construction in well-partially-ordered free monoids”, *Discrete Mathematics*, 21:3 (1978), 237–252.
- [18] A. N. Maslov, “Estimates of the number of states of finite automata”, *Soviet Mathematics Doklady*, 11 (1970), 1373–1375.
- [19] K. Mehlhorn, “Pebbling mountain ranges and its application to DCFL-recognition”, *Automata, Languages and Programming (ICALP 1980, Noordwijkerhout, The Netherlands, 14–18 July 1980)*, LNCS 85, 422–435.
- [20] A. Okhotin, “On the state complexity of scattered substrings and superstrings”, *Fundamenta Informaticae*, 99:3 (2010), 325–338.
- [21] A. Okhotin, “Input-driven languages are linear conjunctive”, *Theoretical Computer Science*, 618 (2016), 52–71.
- [22] A. Okhotin, K. Salomaa, “Complexity of input-driven pushdown automata”, *SIGACT News*, 45:2 (2014), 47–67.
- [23] A. Okhotin, K. Salomaa, “Descriptive complexity of unambiguous input-driven pushdown automata”, *Theoretical Computer Science*, 566 (2015), 1–11.

- [24] A. Okhotin, K. Salomaa, “State complexity of operations on input-driven pushdown automata”, *Journal of Computer and System Sciences*, 86 (2017), 207–228.
- [25] 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, 260–272.
- [26] A. Okhotin, K. Salomaa, “The quotient operation on input-driven pushdown automata”, *Descriptive Complexity of Formal Systems* (DCFS 2017, Milan, Italy, 3–5 July 2017), LNCS 10316, 299–310.
- [27] A. Okhotin, K. Salomaa “Further closure properties of input-driven pushdown automata”, *Descriptive Complexity of Formal Systems* (DCFS 2018, Halifax, Canada, 25–27 July 2018), LNCS 10952, 224–236.
- [28] X. Piao, K. Salomaa, “Operational state complexity of nested word automata”, *Theoretical Computer Science*, 410 (2009), 3290–3302.
- [29] K. Salomaa, “Limitations of lower bound methods for deterministic nested word automata”, *Information and Computation*, 209 (2011), 580–589.
- [30] J. I. Seiferas, R. McNaughton, “Regularity-preserving relations”, *Theoretical Computer Science*, 2:2 (1976), 147–154.