

State complexity of operations on input-driven pushdown automata

Alexander Okhotin^{1,2} * and Kai Salomaa³

¹ Department of Mathematics, University of Turku, 20014 Turku, Finland,
`alexander.okhotin@utu.fi`

² Academy of Finland

³ School of Computing, Queen's University, Kingston, Ontario K7L 3N6, Canada,
`ksalomaa@cs.queensu.ca`

Abstract. The family of deterministic input-driven pushdown automata (IDPDA; a.k.a. visibly pushdown automata, a.k.a. nested word automata) is known to be closed under reversal, concatenation and Kleene star. As shown by Alur and Madhusudan (“Visibly pushdown languages”, STOC 2004), the reversal and the Kleene star of an n -state IDPDA can be represented by an IDPDA with $2^{O(n^2)}$ states, while concatenation of an m -state and an n -state IDPDA is represented by an IDPDA with $2^{O((m+n)^2)}$ states. This paper presents more efficient constructions for the reversal and for the Kleene star, which yield $2^{\Theta(n \log n)}$ states, as well as an $m2^{\Theta(n \log n)}$ -state construction for the concatenation. These constructions are optimal due to the previously known matching lower bounds.

1 Introduction

A subclass of deterministic pushdown automata called *input-driven pushdown automata* (IDPDA), in which the input letter determines whether the automaton should push a symbol, pop a symbol or leave the stack untouched, was introduced by Mehlhorn [13] in 1980s and had only one follow-up paper at the time [5]. Their systematic investigation was initiated more than twenty years later by Alur and Madhusudan [2], who renamed the model to *visibly pushdown automata*, proved that its deterministic and nondeterministic cases are equal in power, and established its closure under all basic operations. These results inspired an ongoing stream of research on the properties of this model [1,3,6,7,9,15]. Part of the literature has adopted yet another name for the same model: *nested word automata*.

Though deterministic and nondeterministic IDPDAs define the same class of languages, they differ in terms of succinctness of description. A pushdown automaton uses states of two types—internal states and pushdown symbols. The natural succinctness measure is the sum of the number of these states. The determinization blowup of nonterministic IDPDAs was assessed by Alur and

* Supported by the Academy of Finland under grant 134860.

Madhusudan [2,3], who proved that representing an n -state nondeterministic IDPDA in the worst case requires $2^{\Theta(n^2)}$ states in an equivalent deterministic IDPDA. Recently, the authors [16] defined an intermediate family of *unambiguous* IDPDAs and showed that transforming a nondeterministic automaton to an unambiguous one, as well as an unambiguous automaton to deterministic, requires $2^{\Theta(n^2)}$ states in each case.

The closure of the language family defined by input-driven pushdown automata under all basic language-theoretic operations, established by Alur and Madhusudan [2], leaves related succinctness questions. According to Alur and Madhusudan [2], the Kleene star of an n -state IDPDA can be represented by an IDPDA with $2^{O(n^2)}$ states, and the concatenation of an m -state and an n -state IDPDA is representable by an IDPDA with $2^{O((m+n)^2)}$ states. In each case, the construction proceeds by representing the result of the operation by a nondeterministic IDPDA, and then by determinizing the latter. For the reversal of an n -state IDPDA, Piao and Salomaa [17] presented a similar $2^{O(n^2)}$ -state construction, that relies on constructing a nondeterministic IDPDA and determinizing it. Furthermore, Piao and Salomaa [17] gave an improved construction for the concatenation of two IDPDAs, which uses $m \cdot 2^{O(n^2)}$ states. At the same time, Piao and Salomaa [17] demonstrated lower bounds on the number of states required to represent these operations: the reversal and the Kleene star requires $2^{\Omega(n \log n)}$ states, and the concatenation requires $2^{\Omega(n \log n)}$ states as well.

This paper presents more efficient constructions for these three operations, and thus determines the asymptotically optimal number of states needed to represent them. It is shown that both the reversal and the star of an n -state IDPDA can be represented using only $2^{O(n \log n)}$ states, which coincides with the lower bound given by Piao and Salomaa [17]. For the concatenation, the proposed construction yields an IDPDA with $m2^{O(n \log n)}$ states. This result is accompanied by an $m2^{\Omega(n \log n)}$ -state lower bound, which refines the lower bound by Piao and Salomaa [17].

2 Definitions

A (*deterministic*) *input-driven pushdown automaton* (IDPDA) [13,2] is a special case of a deterministic pushdown automaton, in which the input alphabet is split into three classes, Σ_{+1} , Σ_{-1} and Σ_0 , and the type of the input symbol determines the type of the operation with the stack. For an input symbol in Σ_{+1} , the automaton always pushes one symbol onto the stack. If the input symbol is in Σ_{-1} , the automaton pops one symbol. Finally, for a symbol in Σ_0 , the automaton may not use the stack: that is, neither modify it, nor even examine its contents.

Unless otherwise mentioned, the acronym IDPDA shall refer to a *deterministic* input-driven pushdown automaton, and when the computation is allowed to use nondeterminism, the model shall be referred to as a *nondeterministic IDPDA*.

Different names and different notation for these automata has been used in the literature. Most of the time, they are regarded as pushdown automata, and recent literature often refers to them under an alternative name of *visibly pushdown automata* [1,2,3,6,7]. An alternative outlook at essentially the same definition regards this model as an automaton operating on nested words—a *nested word automaton* [3,9,17]. Under the latter outlook, internal states are called “linear” or “horizontal” states, while pushdown symbols are regarded as “hierarchical” or “vertical” states. This paper uses the terminology of pushdown automata.

An IDPDA is formally defined over an *action alphabet*, which is a triple $\tilde{\Sigma} = (\Sigma_{+1}, \Sigma_{-1}, \Sigma_0)$,

in which the components Σ_{+1} , Σ_{-1} and Σ_0 are finite disjoint sets. In the following, unless otherwise mentioned, Σ_{+1} , Σ_{-1} and Σ_0 always refer to components of an action alphabet, and their union is denoted by Σ . A string over $\tilde{\Sigma}$ is an ordinary string over Σ , where each symbol is assigned a “type” depending on the component it belongs to.

Let Q denote the set of (internal) states of the automaton, with a subset of accepting states $F \subseteq Q$, let Γ be its pushdown alphabet, and let $\perp \in \Gamma$ be the initial pushdown symbol. For each input symbol $a \in \Sigma_{+1}$, the behaviour of the automaton is described by partial functions $\delta_a : Q \rightarrow Q$ and $\gamma_a : Q \rightarrow (\Gamma \setminus \{\perp\})$, which provide the next state and the symbol to be pushed onto the stack, respectively. For every $b \in \Sigma_{-1}$, there is a partial function $\delta_b : Q \times \Gamma \rightarrow Q$ specifying the next state, assuming that the given stack symbol is popped from the stack. For $c \in \Sigma_0$, the state change is described by a partial function $\delta_c : Q \rightarrow Q$. There is an additional condition that whenever the stack contains only one symbol (which shall be \perp), any attempts to pop this symbol will result in checking that it is there, but not actually removing it. Once the automaton processes the last symbol of the input, it accepts if the current state is in F , regardless of whether the stack is empty or not.

Most of the constructions in this paper manipulate functions, mainly functions from Q to Q , and some related notation ought to be introduced. For any sets X and Y , denote the set of functions from X to Y by Y^X . Such functions can be applied to subsets of X : if $f : X \rightarrow Y$ is a function, then $f(S) = \{f(x) \mid x \in S\}$ for any set $S \subseteq X$; similarly, the set of pre-images of a set $T \subseteq Y$ is denoted by $f^{-1}(T) = \{x \mid x \in X : f(x) \in T\}$. If $f : X \rightarrow Y$ and $g : Y \rightarrow Z$ are two functions, then their *composition* $g \circ f$ is a function from X to Z defined by $(g \circ f)(x) = g(f(x))$.

The set of *well-nested strings* over an alphabet $(\Sigma_{+1}, \Sigma_{-1}, \Sigma_0)$, denoted by L_{Dyck} , is defined by the following context-free grammar:

$$\begin{aligned} S &\rightarrow \langle S \rangle && (\langle \in \Sigma_{+1}, \rangle \in \Sigma_{-1}) \\ S &\rightarrow SS \\ S &\rightarrow c && (c \in \Sigma_0) \\ S &\rightarrow \varepsilon \end{aligned}$$

Computations of IDPDAs on well-nested strings have the most straight form. The automaton finishes reading such a string with the same stack contents as in the beginning, and it never attempts to pop any symbols underneath. The behaviour of an automaton on a well-nested string w can thus be characterized by a function $f_w: Q \rightarrow Q$, which maps the initial state of the automaton to its state after processing the string. Given an IDPDA A , one can construct another IDPDA B with the set of states Q^Q , that calculates the behaviour of A on the last well-nested substring it reads; this fact will be essentially used in all constructions in this paper.

The measure of succinctness of an IDPDA adopted in the literature is the combined number $|Q| + |\Gamma|$ of internal states and pushdown symbols, which may be regarded as two kinds of states. The *state complexity of a language* is the least value of $|Q| + |\Gamma|$ among the IDPDAs recognizing this language. The *state complexity of an operation* is a function mapping the state complexities of its arguments to the worst-case state complexity of the result.

Generally, state complexity lower bounds are established using more or less *ad hoc* methods [3,19]. Various lower bound criteria for the size of IDPDAs were given by Piao and Salomaa [17]. The following variant of those criteria is tailored for the specific purpose of proving the results in this paper.

Lemma 1. *Let A be an IDPDA over action alphabet $(\Sigma_{+1}, \Sigma_{-1}, \Sigma_0)$ and let S be a set of strings over Σ_0 . Suppose that there exists a word $w \in \Sigma^*$ such that*

- (i) *For each $u \in S$, wu is a prefix of some word of $L(A)$, and,*
- (ii) *for any $u_1, u_2 \in S$, $u_1 \neq u_2$, there exists $v \in \Sigma^*$ such that $wu_1v \in L(A)$ if and only if $wu_2v \notin L(A)$.*

Then the number of states of A is at least $|S|$.

Proof. By (i) for each $u \in S$, A reaches the end of wu in some state q_u . Since the strings of S contain only symbols of Σ_0 , condition (ii) implies that for any $u_1 \neq u_2$ the states q_{u_1} and q_{u_2} need to be distinct. \square

3 Reversal

The reversal of a string $w = a_1a_2 \dots a_\ell$ over an action alphabet $\tilde{\Sigma} = (\Sigma_{+1}, \Sigma_{-1}, \Sigma_0)$, with $\ell \geq 0$ and $a_i \in \Sigma$, is the string $w^R = a_\ell \dots a_2a_1$ over the *inverted alphabet* $\tilde{\Sigma}^R = (\Sigma_{-1}, \Sigma_{+1}, \Sigma_0)$, in which the symbol types “+1” and “-1” are interchanged. The reversal of a language L over $\tilde{\Sigma}$ is the language $L^R = \{w^R \mid w \in L\}$, viewed as a language over the inverted alphabet $\tilde{\Sigma}^R$.

Consider a nondeterministic input-driven pushdown automaton recognizing a language L . Then another nondeterministic IDPDA recognizing the language L^R can be obtained by reversing all transitions and exchanging the sets of initial and accepting states [17]. However, the construction for determinizing a nondeterministic IDPDA implies only an $2^{O(n^2)}$ upper bound on the number of states of a deterministic IDPDA recognizing the reversal. A more efficient construction for reversing a given deterministic IDPDA is given in this section.

The construction is first presented in a simplified form applicable to an IDPDA A operating on well-nested strings. Under this assumption, an IDPDA for $L(A)^R$ can separately calculate the behaviour of A as a function from Q to Q on each level of nesting of brackets, transfer the behaviour through the stack upon reading a closing bracket, and combine behaviours on substrings upon reading an opening bracket.

Lemma 2. *Let A be an IDPDA over an alphabet $(\Sigma_{+1}, \Sigma_{-1}, \Sigma_0)$ that accepts only well-nested strings, let Q be its set of states and let Γ be its pushdown alphabet. Then there exists an IDPDA C with the set of states Q^Q and the pushdown alphabet $Q^Q \times \Sigma_{-1}$ that recognizes the language $L(A)^R$ (under the assumption that C rejects upon reaching the end of the input with nonempty stack).*

Proof. The goal of the construction is to simulate the behaviour of A on a well-nested string w , upon reading the string w^R . Note that the input alphabet of C is now the inverted alphabet $(\Sigma_{-1}, \Sigma_{+1}, \Sigma_0)$. A state of C represents the behaviour of A on the reversal of the longest well-nested suffix of the string read so far.

The initial state of C is $q'_0 = id$, the identity function on Q . The transition on each symbol from Σ_0 is defined as a composition of the transition function of A on this symbol with the calculated behaviour of A on the previously read suffix:

$$\delta'_c(f) = f \circ \delta_c.$$

If $f: Q \rightarrow Q$ is the behaviour of A on the suffix u , then this transition computes the behaviour of A on cu .

Whenever C reads a symbol from Σ_{-1} , on which A pops, C must push. What it does is to push the calculated behaviour $f: Q \rightarrow Q$ on the suffix and the current symbol $> \in \Sigma_{-1}$ to the stack, and begin calculating a new behaviour on the deeper level of nesting:

$$\begin{aligned} \delta'_>(f) &= id, \\ \gamma'_>(f) &= (f, >). \end{aligned}$$

By the time C reaches the matching bracket $< \in \Sigma_{+1}$, it will have the behaviour of A on the inner level calculated in its internal state $g: Q \rightarrow Q$. It pops from the stack the pair $(f, >)$, where f is the behaviour on the suffix and $>$ is the other previously read bracket. Let u denote the well-nested string between these brackets, the behaviour on which is g , and let v be the suffix, on which the behaviour is f . Now B has all the data to calculate the behaviour of A on $\langle u \rangle v$ as follows:

$$\delta'_<(g, (f, >)) = f \circ h,$$

where the function $h: Q \rightarrow Q$, defined by

$$h(q) = \delta'_>(g(\delta'_<(q)), \gamma'_<(q)),$$

represents the behaviour of A on $\langle u \rangle$.

Claim. For every string w over $\tilde{\Sigma}$, let f be the behaviour of A on the longest well-nested prefix of w . Then the automaton C , executed on w^R with the initial state id , finishes its computation in the state f .

The claim is proved by an induction on the length of w . The basis is $w = \varepsilon$, with the empty string as the longest well-nested prefix, and the behaviour of A on it is the identity function, which is the initial state of C . For the induction step, the proof is split into three cases, depending on the form of w :

- If w begins with a symbol $c \in \Sigma_0$, let $w = cuv$, where u is longest well-nested prefix of the rest of w . The first portion of the computation of C on $w^R = v^R u^R c^R$ is a computation on the shorter string $v^R u^R$. By the induction hypothesis for uv , the automaton C computes the behaviour of A on u . Let f be this behaviour. Then cu is the longest well-nested prefix of w , and the transition of C by c correctly computes the behaviour of A on cu as a composition $f \circ \delta_c$.
- Let the first symbol of w be $< \in \Sigma_{+1}$. Then the longest well-nested prefix of w is a string of the form $\langle u \rangle v$, where u and v are well-nested and $> \in \Sigma_{-1}$. Let $w = \langle u \rangle vx$ and note that the longest well-nested prefix of vx is v . The computation of C on $w^R = x^R v^R \rangle u^R <$ begins with the computation on $x^R v^R$ and, by the induction hypothesis, calculates the behaviour of A on v . Denote this behaviour by f . Next, C reads $>$ and pushes the pair $(f, >)$ to the stack, and afterwards, C processes the well-nested substring u^R . Applying the induction hypothesis to the ensuing computation of C —that is, to the string $u \rangle vx$ with the longest well-nested prefix u —shows that C calculates the behaviour A on u . Finally, C reads $<$, pops the pair $(f, >)$ from the stack, uses the behaviour of A on u stored in the internal state to calculate the behaviour h of A on $\langle u \rangle$, and finally combines it with the behaviour of A on v as $f \circ h$. This is the behaviour of A on $\langle u \rangle v$, which is the longest well-nested prefix of w .
- Assume that w begins with a symbol $> \in \Sigma_{-1}$. Then the longest well-nested prefix of w is ε , and the computation of C on w^R ends with transition by $>$, which sets the internal state to id .

It is left to define the set of accepting states of C as

$$F' = \{ f: Q \rightarrow Q \mid f(q_0) \in F \},$$

where q_0 is the initial state of A and F is the set of accepting states of A . Then, for every well-nested string w , the automaton C , executed on w^R computes the behaviour f of A on w , and accepts if and only if $f(q_0) \in F$, that is, if and only if A accepts w . If the input string is not well-nested, then C either reaches an undefined transition by the bottom stack symbol \perp , or finishes reading the input with nonempty stack. \square

In the general case of IDPDAs operating on not necessarily well-nested strings, the construction is slightly extended.

Lemma 3. For every IDPDA A over an alphabet $(\Sigma_{+1}, \Sigma_{-1}, \Sigma_0)$ with a set of states Q and a pushdown alphabet Γ , there exists an IDPDA C over the inverted alphabet $(\Sigma_{-1}, \Sigma_{+1}, \Sigma_0)$ with the set of states $Q^Q \times 2^Q$ and the pushdown alphabet $Q^Q \times 2^Q \times \Sigma_{-1}$ that recognizes the language $L(A)^R$.

Proof. Given a string uv , where u is its longest well-nested prefix, the automaton C is given a string $v^R u^R$ and should simulate the computation of A on uv . The goal of the construction is that C calculates (i) the behaviour of A on u , and (ii) the set states, beginning from which A would accept the string uv .

The initial state of C is $q'_0 = (id, F)$.

On any symbol $c \in \Sigma_0$, its transitions are:

$$\delta'_c((f, S)) = (f \circ \delta_c, \delta_c^{-1}(S)).$$

Every symbol $> \in \Sigma_{-1}$, which is a closing bracket for A , is therefore an opening bracket for C , and its transitions are defined as follows:

$$\begin{aligned} \delta'_>((f, S)) &= (id, \{q \mid \delta_>(q, \perp) \in S\}), \\ \gamma'_>((f, S)) &= (f, S, >). \end{aligned}$$

Each A 's opening bracket $< \in \Sigma_{+1}$ is regarded by C as a closing bracket, and is processed as follows:

$$\delta'_<((g, T), (f, S, >)) = (f \circ h, h^{-1}(S)),$$

where $h : Q \rightarrow Q$ is defined by

$$h(q) = \delta_>(g(\delta_<(q)), \gamma_<(q)).$$

And if the stack is empty (that is, $<$ has no matching right bracket):

$$\delta'_<((g, T), \perp) = (id, \delta_<^{-1}(T))$$

(p. 13)

Claim. For every string w , the automaton C , after reading w^R , reaches a state (f, S) , where f is the behaviour of A on the longest well-nested prefix of w , and S is the set of all such states q , that A , having begun a computation in the state q with the empty stack, accepts after reading w .

Finally, the set of accepting states of C is defined as

$$F' = \{(f, S) \mid q_0 \in S\},$$

where q_0 is the initial state of A . □

A matching lower bound for reversal is already known.

Proposition 1 (Piao and Salomaa [17]). Let $\Sigma_{+1} = \{<\}$, $\Sigma_{-1} = \{>\}$, $\Sigma_0 = \{a, b, c\}$. For $n \geq 1$, the language

$$L_n = \bigcup_{u \in \{a, b\}^{\lceil \log n \rceil}} u \langle (\{a, b\}^* c)^{n-1} u c (\{a, b\}^* c)^* \rangle$$

has an IDPDA with $O(n)$ states, while any IDPDA for its reversal requires at least $2^{\Omega(n \log n)}$ states.

This shows that the construction in Lemma 3 is asymptotically optimal, and thus the state complexity of reversal has been determined as follows.

Theorem 1. *The state complexity of reversal of IDPDAs is $2^{\Theta(n \log n)}$.*

4 Concatenation

Given an m -state IDPDA and an n -state IDPDA B , one can represent their concatenation by a nondeterministic IDPDA with $m+n$ states, and then determinize it to obtain $2^{O((m+n)^2)}$ states [2]. An improved construction given below directly yields a deterministic IDPDA, and this IDPDA contains only $m2^{O(n \log n)}$ states. Like in the previous section, this construction is first presented in an idealized form, in which all strings are well-nested.

Lemma 4. *Let A and B be IDPDAs over an alphabet $(\Sigma_{+1}, \Sigma_{-1}, \Sigma_0)$ that accept only well-nested strings, let P and Q be their respective sets of states, let Γ and Ω be their pushdown alphabets. Then there exists an IDPDA C with the set of states $P \times (2^Q \cup Q^Q)$ and the pushdown alphabet $\Gamma \times (2^Q \cup Q^Q) \times \Sigma_{+1}$ that recognizes the language $L(A) \cdot L(B)$.*

Proof. The first component of all states of C is used to simulate the operation of A . The states from $P \times 2^Q$ are used for well-nested prefixes of the input. After reading a well-nested prefix w , the automaton C should reach a state with a second component $S \subseteq Q$, containing all states of B reached on strings in $L(A)^{-1}w$. After reading a prefix w that is not well-nested, C should calculate a state with the second component $f: Q \rightarrow Q$ representing the behaviour of B on the longest well-nested suffix of w .

Let π and δ be the transition functions of A and B , respectively. Let μ and ν be their push functions on Σ_{+1} , and let F_A and F_B be their sets of accepting states.

The initial state of C is $(p_0, \{q_0 \mid \text{if } p_0 \in F_A\})$.

Its transitions on the bottom level, for $p \in P$, $S \subseteq Q$ and $c \in \Sigma_0$, are:

$$\delta'_c((p, S)) = (\pi_c(p), \delta_c(S) \cup \{q_0 \mid \text{if } \pi_c(p) \in F_A\}).$$

When C enters the first level of brackets by a symbol $< \in \Sigma_{+1}$, it continues simulating A in the first component, and switches to computing the behaviour of B in the second component:

$$\begin{aligned} \delta'_<((p, S)) &= (\pi_<(p), id), \\ \gamma'_<((p, S)) &= (\mu_<(p), S, <), \end{aligned}$$

where $id: Q \rightarrow Q$ is the identity function. The state S and the symbol $<$ are stored in the stack, along with the stack symbol of the simulated automaton A .

When C returns from the first level of brackets by a symbol $> \in \Sigma_{+1}$, it has the behaviour of B on the substring inside the brackets computed, and can

combine it with the behaviour on the brackets $<$ (popped from the stack) and $>$ (read from the input) to form a function $h : Q \rightarrow Q$, defined by

$$h(q) = \delta_{>}(g(\delta_{<}(q)), \nu_{<}(q)).$$

Then C can apply this function to the set S popped from the stack as follows:

$$\delta'_{>}((p, g), (s, S, <)) = (\pi_{>}(p, s), h(S)),$$

Transitions inside the brackets, on $c \in \Sigma_0$.

$$\delta'_c((p, f)) = (\pi_c(p), \delta_c \circ f)$$

Going into the next level of brackets: for $p \in P$, $f : Q \rightarrow Q$ and $< \in \Sigma_{+1}$:

$$\begin{aligned} \delta'_{<}((p, f)) &= (\pi_{<}(p), id), \\ \gamma'_{<}((p, f)) &= (\mu_{<}(p), f, <). \end{aligned}$$

Returning into the previous first level of brackets: for $> \in \Sigma_{-1}$, $p \in P$, $f : Q \rightarrow Q$, $s \in \Gamma$, $g : Q \rightarrow Q$ and $< \in \Sigma_{+1}$:

$$\delta'_{>}((p, g), (s, f, <)) = (\pi_{>}(p, s), g \circ f),$$

where $h : Q \rightarrow Q$ is defined by

$$h(q) = \delta_{>}(g(\delta_{<}(q)), \nu_{<}(q)).$$

Claim. Upon reading a well-nested prefix w of an input string, the automaton C enters a state (p, S) , where p is the state reached by A on w , and S is the set of all such states $q \in Q$ that $w = uv$ for some $u \in L(A)$ and $v \in \Sigma^*$ with $\delta(q_0, v) = q$.

Upon reading a not well-nested prefix w , the automaton C enters a state (p, f) , where p is the state reached by A on w , and f is the behaviour of B on the longest well-nested suffix of w .

Let all states $(p, S) \in P \times 2^Q$ with $S \cap F_B \neq \emptyset$ be the accepting states of C . Then, by the above claim, C accepts a string w if and only if $w = uv$ for some $u \in L(A)$ and $v \in L(B)$, and therefore $L(C) = L(A)L(B)$. Unlike the automaton constructed in Lemma 2, here the automaton C is also able to reject strings other than well-nested. \square

The next lemma presents the construction for concatenation of IDPDAs of the general form.

Lemma 5. *Let A and B be any IDPDAs over an alphabet $(\Sigma_{+1}, \Sigma_{-1}, \Sigma_0)$. let P and Q be their respective sets of states, let Γ and Ω be their pushdown alphabets. Then there exists a IDPDA C with the set of states $P \times 2^Q \times 2^Q \times Q^Q$ and the pushdown alphabet $\Gamma \times 2^Q \times 2^Q \times Q^Q \times \Sigma_{+1}$ that recognizes the language $L(A) \cdot L(B)$.*

The automaton C simulates A in the first component of its state, and calculates the behaviour of B on the last well-nested suffix of the input in the fourth component. The second and the third components are both sets of such states $q \in Q$, that the string read so far is a concatenation of a string in $L(A)$ with a string, on which B goes from q_0 to q . The third component handles all such factorizations, where the suffix processed by B is a concatenation of well-nested strings and closing brackets from Σ_{-1} . The second component refers to factorizations with the suffix of any other form. The details of the construction are omitted due to space constraints.

A $2^{\Omega(n \log n)}$ -state lower bound on the state complexity of concatenation of an m -state IDPDA and an n -state IDPDA was given by Piao and Salomaa [17]. This lower bound does not provide any dependence of the state complexity of concatenation on the complexity of the first language. The following more elaborate version refines the result of Piao and Salomaa [17] to reflect this dependence.

(p. 15)

Lemma 6. *Let $\Sigma_0 = \{a, b, \#\}$, $\Sigma_{+1} = \{<\}$ and $\Sigma_{-1} = \{>\}$. Then, for every $m, n \geq 1$, the language*

$$K_m = \{w \in \{a, b, \#, <, >\}^* \mid |w|_b \equiv 0 \pmod{m}\}$$

has a DFA with m states, and the language

$$L_n = \bigcup_{k, \ell \in \{1, \dots, n\}} a^k b^* \# (a^* b^* \#)^* < (a^* b^* \#)^k a^\ell b^* \# (a^* b^* \#)^* a^\ell > a^k$$

has an IDPDA with $O(n)$ states and n pushdown symbols, while any IDPDA for their concatenation $K_m L_n$ requires at least mn^n states.

Theorem 2. *The state complexity of concatenation of IDPDAs is $m \cdot 2^{\Theta(n \log n)}$.*

Using a variant of the languages L_n from Lemma 6, we get a lower bound for the state complexity of square.

(p. 16)

Lemma 7. *Let $\Sigma_0 = \{a, \#\}$, $\Sigma_{+1} = \{<\}$ and $\Sigma_{-1} = \{>\}$. Then, for every $n \geq 1$, the language*

$$L_n = (a^* \#)^* \cup \bigcup_{k, \ell \in \{1, \dots, n\}} a^k \# (a^* \#)^* < (a^* \#)^k a^\ell \# (a^* \#)^* a^\ell > a^k$$

has an IDPDA with $O(n)$ states and n pushdown symbols, while any IDPDA for the language $L_n \cdot L_n$ requires at least n^n states.

Theorem 3. *The state complexity of square of IDPDAs is $2^{\Theta(n \log n)}$.*

5 Kleene star

The concatenation $L(A)L(B)$ was recognized by simulating A as it is, along with keeping track of all possible computations of B following a prefix in $L(A)$. Every

time the simulated A would accept, one more computation of B was added to the set.

The below construction for the star is derived from the one for the concatenation. There is no automaton A this time, but multiple computations of B are traced in the same way as before. Whenever one of them would accept, the set is augmented with another computation of B .

Lemma 8. *Let B be a IDPDAs over an alphabet $(\Sigma_{+1}, \Sigma_{-1}, \Sigma_0)$ that accepts only well-nested strings, let Q be its set of states. Then there exists a IDPDA C with the set of states $2^Q \cup Q^Q$ and the pushdown alphabet $(2^Q \cup Q^Q) \times \Sigma_{+1}$ that recognizes the language $L(A)^*$.*

(p. 17)

The construction is extended to the full case of IDPDAs not restricted to well-nested strings as follows.

Lemma 9. *Let B be any IDPDA over an alphabet $(\Sigma_{+1}, \Sigma_{-1}, \Sigma_0)$, let Q be its set of states, and let Γ be its pushdown alphabet. Then there exists a IDPDA C with the set of states $2^Q \times 2^Q \times Q^Q$ and the pushdown alphabet $\Gamma \times 2^Q \times 2^Q \times Q^Q \times \Sigma_{+1}$ that recognizes the language $L(A)^*$.*

(p. 18)

The construction is similar to the one for the concatenation, and is not included in this extended abstract.

This establishes a $2^{O(n \log n)}$ -state upper bound on the state complexity of the star for IDPDAs. A matching lower bound is already known from Salomaa [19], who presented an IDPDA A with $O(n)$ states and stack symbols, such that the total number of states and stack symbols in any IDPDA recognizing $L(A)^*$ is at least $2^{n \cdot \log n}$. These results are combined to the following asymptotic estimation.

Theorem 4. *The state complexity of Kleene star of IDPDAs is $2^{\Theta(n \log n)}$.*

6 Conclusion

The complexity of all basic operations on both deterministic (IDPDA) and non-deterministic (NIDPDA) input-driven pushdown automata has now been determined. In the following table, it is compared to the similar results on two basic types of finite automata recognizing regular languages.

	DFA	NFA	IDPDA	NIDPDA
\cup	mn [12]	$m + n + 1$ [10]	$\Theta(mn)$ [17]	$m + n + O(1)$ [2]
\cap	mn [12]	mn [10]	$\Theta(mn)$ [17]	$\Theta(mn)$ [9]
\sim	n	2^n [4]	n	$2^{\Theta(n^2)}$ [16]
\cdot	$m \cdot 2^n - 2^{n-1}$ [12]	$m + n$ [10]	$m2^{\Theta(n \log n)}$	$m + n + O(1)$ [2]
2	$n \cdot 2^n - 2^{n-1}$ [18]	$2n$ [8]	$2^{\Theta(n \log n)}$	$n + O(1)$ [2]
$*$	$\frac{3}{4}2^n$ [12]	$n + 1$ [10]	$2^{\Theta(n \log n)}$	$n + O(1)$ [2]
R	2^n [11]	$n + 1$ [10]	$2^{\Theta(n \log n)}$	$n + O(1)$ [2]

Investigating the complexity of operations on *unambiguous IDPDAs* [16] is suggested for future work. Since descriptonal complexity questions are already difficult for unambiguous finite automata [14], this task might be nontrivial as well.

References

1. R. Alur, V. Kumar, P. Madhusudan, M. Viswanathan, “Congruences for visibly pushdown languages”, *Automata, Languages and Programming* (ICALP 2005, Lisbon, Portugal, 11–15 July 2005), LNCS 3580, 1102–1114.
2. R. Alur, P. Madhusudan, “Visibly pushdown languages”, *ACM Symposium on Theory of Computing* (STOC 2004, Chicago, USA, 13–16 June 2004), 202–211.
3. R. Alur, P. Madhusudan, “Adding nesting structure to words”, *Journal of the ACM*, 56:3 (2009).
4. J. C. Birget, “Partial orders on words, minimal elements of regular languages, and state complexity”, *Theoretical Computer Science*, 119 (1993) 267–291.
5. B. von Braunmühl, R. Verbeek, “Input-driven languages are recognized in $\log n$ space”, FCT 1983, LNCS 158, 40–51.
6. P. Chervet, I. Walukiewicz, “Minimizing variants of visibly pushdown automata”, *Mathematical Foundations of Computer Science* (MFCS 2007, Český Krumlov, Czech Republic, 26–31 August 2007), LNCS 4708, 135–146.
7. S. Crespi-Reghizzi, D. Mandrioli, “Operator precedence and the visibly pushdown property”, *Language and Automata Theory and Applications* (LATA 2010, Trier, Germany, May 24–28, 2010) LNCS 6031, 214–226.
8. M. Domaratzki, A. Okhotin, “State complexity of power”, *Theoretical Computer Science*, 410:24–25 (2009), 2377–2392.
9. Y.-S. Han, K. Salomaa, “Nondeterministic state complexity of nested word automata”, *Theoretical Computer Science*, 410 (2009), 2961–2971.
10. M. Holzer, M. Kutrib, “Nondeterministic descriptive complexity of regular languages”, *International Journal of Foundations of Computer Science*, 14 (2003), 1087–1102.
11. E. L. Leiss, “Succinct representation of regular languages by Boolean automata”, *Theoretical Computer Science*, 13 (1981), 323–330.
12. A. N. Maslov, “Estimates of the number of states of finite automata”, *Soviet Mathematics Doklady*, 11 (1970), 1373–1375.
13. 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.
14. A. Okhotin, “Unambiguous finite automata over a unary alphabet”, *Mathematical Foundations of Computer Science* (MFCS 2010, Brno, Czech Republic, 23–27 August 2010), LNCS 6281, 556–567.
15. A. Okhotin, “Comparing linear conjunctive languages to subfamilies of the context-free languages”, *SOFSEM 2011: Theory and Practice of Computer Science* (Nový Smokovec, Slovakia, 22–28 January 2011), LNCS 6543, 431–443.
16. A. Okhotin, K. Salomaa, “Descriptive complexity of unambiguous nested word automata”, *Language and Automata Theory and Applications* (LATA 2011, Taragona, Spain, 26–31 May 2011), LNCS 6638, to appear.
17. X. Piao, K. Salomaa, “Operational state complexity of nested word automata”, *Theoretical Computer Science*, 410 (2009), 3290–3302.
18. N. Rampersad, “The state complexity of L^2 and L^k ”, *Information Processing Letters*, 98 (2006), 231–234.
19. K. Salomaa, “Limitations of lower bound methods for deterministic nested word automata”, *Information and Computation*, 209 (2011), 580–589.

Appendix

The following table shows the location of the results presented in this paper.

	reversal	concatenation	star
idealized construction	Section 3	Section 4	Appendix D
its proof	Section 3	omitted	omitted
general construction	Section 3	Appendix B	Appendix D
its proof	Appendix A	omitted	omitted

A Omitted proof for reversal

The claim in Lemma 3. *For every string w , the automaton C , after reading w^R , reaches a state (f, S) , where f is the behaviour of A on the longest well-nested prefix of w , and S is the set of all such states q , that A , having begun a computation in the state q with the empty stack, accepts after reading w .* (p. 7)

Proof. The claim is proved by an induction on the length of w . The basis is $w = \varepsilon$, with the empty string as the longest well-nested prefix. The behaviour of A on ε is the identity function, while the set of states of A , from which it accepts upon reading ε , is exactly F : this is what is given in the initial state of C . For the induction step, the proof is split into four cases, depending on the form of w :

- If w begins with a symbol $c \in \Sigma_0$, let $w = cuv$, where u is longest well-nested prefix of the rest of w . The first portion of the computation of C on $w^R = v^R u^R c^R$ is a computation on the shorter string $v^R u^R$. By the induction hypothesis for uv , the automaton C computes a pair (f, S) , where f is the behaviour of A on u , and S is the set of states, from which A accepts uv . Then cu is the longest well-nested prefix of w , and the transition of C by c correctly computes the behaviour of A on cu as a composition $f \circ \delta_c$, and the set of states from which A accepts cuv as a pre-image $\delta_c^{-1}(S)$.
- Let the first symbol of w be $< \in \Sigma_{+1}$ and assume that w contains is a matching closing bracket, that is, that the longest well-nested prefix of w is a string of the form $<u>v$, where $u, v \in \Sigma^*$ are well-nested and $> \in \Sigma_{-1}$. Let $w = <u>vx$ and note that the longest well-nested prefix of vx is v . The computation of C on $w^R = x^R v^R >^R u^R <$ begins with the computation on $x^R v^R$ and, by the induction hypothesis, calculates a pair (f, S) , where f is the behaviour of A on v , and S is the set of states, from which A accepts vx . Next, C reads $>$ and pushes the triple $(f, S, >)$ to the stack, and afterwards, C processes the well-nested substring u^R . Applying the induction hypothesis to the ensuing computation of C —that is, to the string $u^R >$ with the longest well-nested prefix u^R —shows that C calculates the behaviour A on u , and the set of states, from which A accepts $u^R >$. Finally, C reads $<$, pops the triple $(f, S, >)$ from the stack, uses the behaviour of A on u stored in the internal state to calculate the behaviour h of A on $<u>$. Combining the latter with

the behaviour of A on v as $f \circ h$ yields the behaviour of A on $\langle u \rangle v$, which is the longest well-nested prefix of w . The set of states, from which A accepts $\langle u \rangle vx$, is obtained as the pre-image $h^{-1}(S)$.

- If the first symbol of w is $\langle \in \Sigma_{+1}$, but this symbol does not have a matching closing bracket, then the longest well-nested prefix of w is ε . In this case, the computation of A on w begins with pushing a symbol that will never get popped, while the computation of C on w^R symmetrically ends with trying to pop a symbol that has never been pushed.

Let $w = \langle x$. By the induction hypothesis, C , executed on x^R , computes (in the second component of its state) the set of states T , from which A accepts x . Then A accepts x from the set of states $\delta_{<}^{-1}(T)$, and id is the behaviour of A on ε . These are the values computed by C by a transition $\delta'_{<}((g, T), \perp)$.

- Assume that w begins with a symbol $\rangle \in \Sigma_{-1}$. Then the longest well-nested prefix of w is ε , on which the behaviour of A is id . The computation of A on w begins with an attempt to pop a symbol from the empty stack, while C will end its computation on w^R by pushing a symbol onto the stack.

Let $w = \rangle x$. By the induction hypothesis, the computation of C on x^R yields, in particular, the set of states S , from which A accepts x . The computation of C on $x^R \rangle$ continues by a transition by \rangle , and produces the set of states $\delta_{>}^{-1}(S)$, from which A accepts $\rangle x$, as well as the behaviour of A on ε . \square

B The general construction for concatenation

(p. 9)

Lemma 5. *Let A and B be any IDPDAs over an alphabet $(\Sigma_{+1}, \Sigma_{-1}, \Sigma_0)$, let P and Q be their respective sets of states, let Γ and Ω be their pushdown alphabets. Then there exists a IDPDA C with the set of states $P \times 2^Q \times 2^Q \times Q^Q$ and the pushdown alphabet $\Gamma \times 2^Q \times 2^Q \times Q^Q \times \Sigma_{+1}$ that recognizes the language $L(A) \cdot L(B)$.*

Proof. As in Lemma 4, let π and δ be the transition functions of A and B , let μ and ν be their push functions on Σ_{+1} , and let F_A and F_B be their sets of accepting states.

Initial state: $(p_0, \emptyset, \{q_0 \mid \text{if } p_0 \in F_A\}, id)$.

Transitions for $c \in \Sigma_0$:

$$\delta'_c((p, S, S', f)) = (\pi_c(p), \delta_c(S), \delta_c(S') \cup \{q_0 \mid \text{if } \pi_c(p) \in F_A\}, \delta_c \circ f)$$

For $\langle \in \Sigma_{+1}$:

$$\delta'_{<}((p, S, S', f)) = (\pi_{<}(p), \delta_{<}(S \cup S'), \{q_0 \mid \text{if } \pi_{<}(p) \in F_A\}, id),$$

$$\gamma'_{<}((p, S, S', f)) = (\mu_{<}(p), S, S', f, \langle).$$

For $\rangle \in \Sigma_{-1}$:

$$\begin{aligned} & \delta'_{>}((p, T, T', g), (s, S, S', f, \langle)) = \\ & = (\pi_{>}(p, s), h_{\langle g \rangle}(S), h_{\langle g \rangle}(S') \cup \delta_{>}(T', \perp) \cup \{q_0 \mid \text{if } \pi_{>}(p, s) \in F_A\}, h_{\langle g \rangle} \circ f), \end{aligned}$$

where the function $h_{\langle g \rangle} : Q \rightarrow Q$ is defined by

$$h_{\langle g \rangle}(q) = \delta_{\rangle}(g(\delta_{\langle}(q)), \nu_{\langle}(q)).$$

And if C 's stack is empty:

$$\delta'_{\rangle}((p, T, T', g), \perp) = (\pi_{\rangle}(p, s), \delta_{\rangle}(T, \perp), \delta_{\rangle}(T', \perp) \cup \{q_0 \mid \text{if } \pi_{\rangle}(p, s) \in F_A\}, id).$$

The correctness of the construction is formally claimed as follows:

Claim. Upon reading a prefix w of an input string, the automaton C should enter a state (p, S, S', f) , where

- p is the state reached by A on w ;
- S is the set of all such states $q \in Q$ that $w = uv$ for some $u \in L(A)$ and $v \notin (L_{\text{Dyck}} \cdot \Sigma_{-1})^*$ with $\delta(q_0, v) = q$;
- S' is the set of all such states $q \in Q$ that $w = uv$ for some $u \in L(A)$ and $v \in (L_{\text{Dyck}} \cup \Sigma_{-1})^*$ with $\delta(q_0, v) = q$;
- f is the behaviour of B on the longest well-nested suffix of w .

The set of accepting states of C contains all states (p, S, S', f) with $(S \cup S') \cap F_B \neq \emptyset$. Since, by the above claim, $S \cup S'$ is the set of all such states $q \in Q$ that $w = uv$ for some $u \in L(A)$ and $v \in \Sigma^*$ with $\delta(q_0, v) = q$, the automaton C will accept exactly the strings in $L(A)L(B)$. \square

C Lower bound proofs in Section 4

Lemma 6. *Let $\Sigma_0 = \{a, b, \#\}$, $\Sigma_{+1} = \{\langle\}$ and $\Sigma_{-1} = \{\rangle\}$. Then, for every $m, n \geq 1$, the language* (p. 10)

$$K_m = \{w \in \{a, b, \#, \langle, \rangle\}^* \mid |w|_b \equiv 0 \pmod{m}\}$$

has a DFA with m states, and the language

$$L_n = \bigcup_{k, \ell \in \{1, \dots, n\}} a^k b^* \# (a^* b^* \#)^* \langle (a^* b^* \#)^k a^\ell b^* \# (a^* b^* \#)^* a^\ell \rangle a^k$$

has an IDPDA with $O(n)$ states and n pushdown symbols, while any IDPDA for their concatenation $K_m L_n$ requires at least mn^n states.

Proof. We describe a construction of an IDPDA A for the language L_n . The following discussion assumes that the input is in $(a^* b^* \#)^* \langle (a^* b^* \#)^* a^* \rangle a^*$. This check can easily be made by multiplying the number of states of A by a constant.

The IDPDA A counts the number $1 \leq k \leq n$ of symbols a occurring in a prefix of the input and stores it in a state q_k . (If the prefix has more than n symbols a , the computation rejects.) After this A skips input symbols until the element $\langle \in \Sigma_{+1}$ where the computation pushes to the stack an encoding of k . After bypassing \langle , the computation uses the state as a counter to skip exactly

k substrings in $a^*b^*\#$ and counts the number of a following the k 'th substring. This number ℓ is stored in the state. When processing the remainder of the input, A remembers ℓ and counts each consecutive sequence of symbols a . If A encounters a substring of ℓ symbols a followed by $>\in \Sigma_{-1}$, the computation "retrieves" the number k from the stack symbol and checks that the remaining suffix consists of exactly k symbols a .

The IDPDA A uses exactly n pushdown symbols. The computation needs to count up to n and compare numbers of size at most n and this can be done easily with $O(n)$ states.

It remains to prove the lower bound. Let \mathcal{F}_n denote the set of all functions $\{1, \dots, n\} \rightarrow \{1, \dots, n\}$. For $f \in \mathcal{F}_n$ and $0 \leq j < m$ we define the following string over Σ_0 :

$$u_{f,j} = b^j \# a^{f(1)} \# a^{f(2)} \# \dots \# a^{f(n)}.$$

Using the notations of Lemma 1, we write

$$S = \{u_{f,j} \mid f \in \mathcal{F}_n, 0 \leq j < m\}, \quad \text{and,}$$

$$w = a \# a^2 \# a^3 \# \dots \# a^n \# <$$

Clearly each string $wu_{f,j}$ is a prefix of a string of K_m , and hence of $K_m L_n$, and condition (i) of Lemma 1 is satisfied. To verify condition (ii), consider $f_1, f_2 \in \mathcal{F}_n$ and $j_1, j_2 \in \{0, \dots, m-1\}$, where $(f_1, j_1) \neq (f_2, j_2)$.

First, if $j_1 \neq j_2$ choose $v = b^{m-j_1} \# > a \# < \# a \# a > a$. Now we can write $wu_{f_1, j_1} v = \alpha_1 \cdot \alpha_2$ where $\alpha_2 = a \# < \# a \# a > a \in L_n$ and $|\alpha_1|_b = m$, and consequently $\alpha_1 \in K_m$. On the other hand, consider the string $\beta = wu_{f_2, j_2} v$. The only suffix of β in L_n is $a \# < \# a \# a > a$ (because strings of L_n contain exactly one pair of symbols $<$ and $>$ and at least one a as a prefix). Now we note that $|wu_{f_2, j_2} b^{m-j_1} \# >|_b \equiv j_2 - j_1 \pmod{m}$ and hence β cannot be written as a concatenation of a string of K_m and of L_n .

Second, assume that $f_1 \neq f_2$ and choose $x \in \{1, \dots, n\}$ such that $f_1(x) \neq f_2(x)$. Now we choose $v = \# a^{f_1(x)} > a^x$. We can write

$$wu_{f_1, j_1} v = (a \# a^2 \# \dots \# a^{x-1} \#) \cdot (a^x \# \dots \# a^n \# < b^j \# a^{f_1(1)} \# a^{f_1(2)} \# \dots \# a^{f_1(n)} \# a^{f_1(x)} > a^x)$$

where the string inside the first (respectively, the second) pair of parentheses is in K_m (respectively, in L_n). On the other hand, since $f_1(x) \neq f_2(x)$ the string

$$< b^j \# a^{f_2(1)} \# a^{f_2(2)} \# \dots \# a^{f_2(n)} \# a^{f_1(x)} > a^x$$

cannot be a suffix of any string of L_n , and this implies that $wu_{f_2, j_2} v \notin K_m L_n$.

Now Lemma 1 implies that any IDPDA recognizing $K_m L_n$ has at least $|S| = m \cdot n^n$ states. \square

(p. 10)

Lemma 7. *Let $\Sigma_0 = \{a, \#\}$, $\Sigma_{+1} = \{<\}$ and $\Sigma_{-1} = \{>\}$. Then, for every $n \geq 1$, the language*

$$L_n = (a^* \#)^* \cup \bigcup_{k, \ell \in \{1, \dots, n\}} a^k \# (a^* \#)^* < (a^* \#)^k a^\ell \# (a^* \#)^* a^\ell > a^k$$

has an IDPDA with $O(n)$ states and n pushdown symbols, while any IDPDA for the language $L_n \cdot L_n$ requires at least n^n states.

Proof. An IDPDA of the required size is constructed for L_n completely analogously as in the proof of Lemma 6.

For the lower bound, let \mathcal{F}_n again denote the set of functions $\{1, \dots, n\} \rightarrow \{1, \dots, n\}$ and for $f \in \mathcal{F}_n$ define $u_f = \#a^{f(1)}\#a^{f(2)}\#\dots\#a^{f(n)} \in \Sigma_0^*$. Choose S to consist of all strings u_f , $f \in \mathcal{F}_n$ and $w = a\#a^2\#\dots\#a^n <$. Similarly as in the proof of Lemma 6 we can verify that for $f \in \mathcal{F}_n$ and $k, \ell \in \{1, \dots, n\}$:

$$wu_f\#a^\ell > a^k \in L_n \cdot L_n \text{ iff } f(k) = \ell.$$

This means that S and w satisfy the conditions of Lemma 1 for $L_n \cdot L_n$ and, consequently, any IDPDA recognizing $L_n \cdot L_n$ has at least $|S| = n^n$ states. \square

D Construction for the Kleene star

Lemma 8. *Let B be a IDPDAs over an alphabet $(\Sigma_{+1}, \Sigma_{-1}, \Sigma_0)$ that accepts (p. 11) only well-nested strings, let Q be its set of states. Then there exists a IDPDA C with the set of states $2^Q \cup Q^Q$ and the pushdown alphabet $(2^Q \cup Q^Q) \times \Sigma_{+1}$ that recognizes the language $L(A)^*$.*

Proof. The construction is very similar to the one for the concatenation, given in Lemma 4. After reading a well-nested prefix w of the input, the automaton C should calculate the set $S \subseteq Q$ of all states reached by B on all strings in $(L(A)^*)^{-1}w$. For a string w that is not well-nested, C should calculate the behaviour of B on the longest well-nested suffix of w .

Let Γ be the pushdown alphabet of B , let δ be its transition function, let γ be its push functions on Σ_{+1} , let F be its set of accepting states.

Initial state: $\{q_0\}$. Transitions on the bottom level, for $S \subseteq Q$ and $c \in \Sigma_0$:

$$\delta'_c(S) = \{ \delta_c(q) \mid q \in S \} \cup \{ q_0 \mid \text{if } \delta_c(q) \in F \text{ for some } q \in S \}$$

Entering the first level of brackets, for $S \subseteq Q$ and $< \in \Sigma_{+1}$:

$$\begin{aligned} \delta'_<(S) &= id, \\ \gamma'_<(S) &= (S, <), \end{aligned}$$

where $id : Q \rightarrow Q$ is the identity function.

Returning from the first level of brackets: for $> \in \Sigma_{-1}$, $g : Q \rightarrow Q$, $s \in \Gamma$, $S \subseteq Q$ and $< \in \Sigma_{+1}$:

$$\delta'_>((p, g), (s, S, <)) = (\pi_{>}(p, s), \{ h(q) \mid q \in S \}),$$

where $h : Q \rightarrow Q$ is defined by

$$h(q) = \delta_{>}(g(\delta_{<}(q)), \nu_{<}(q)).$$

Inside the brackets: the standard calculation of the behaviour of A . On $c \in \Sigma_0$:

$$\delta'_c(f) = \delta_c \circ f.$$

Going one level deeper:

$$\begin{aligned}\delta'_<(f) &= id, \\ \gamma'_<(f) &= (f, <).\end{aligned}$$

Returning back: for $> \in \Sigma_{-1}$, $f, g : Q \rightarrow Q$ and $< \in \Sigma_{+1}$:

$$\delta'_>(g, (f, <)) = h \circ f,$$

where $h : Q \rightarrow Q$ is defined by

$$h(q) = \delta_>(g(\delta_<(q)), \nu_<(q)).$$

Accepting states: S with $S \cap F \neq \emptyset$. □

(p. 11)

Lemma 9. *Let B be any IDPDA over an alphabet $(\Sigma_{+1}, \Sigma_{-1}, \Sigma_0)$, let Q be its set of states, and let Γ be its pushdown alphabet. Then there exists a IDPDA C with the set of states $2^Q \times 2^Q \times Q^Q$ and the pushdown alphabet $\Gamma \times 2^Q \times 2^Q \times Q^Q \times \Sigma_{+1}$ that recognizes the language $L(A)^*$.*

Proof. Initial state: $(\emptyset, \{q_0 \mid \text{if } p_0 \in F_A\}, id)$.

Transitions for $c \in \Sigma_0$:

$$\delta'_c((S, S', f)) = (\delta_c(S), \delta_c(S') \cup \{q_0 \mid \text{if } \delta_c(S \cup S') \cap F \neq \emptyset\}, \delta_c \circ f)$$

For $< \in \Sigma_{+1}$:

$$\begin{aligned}\delta'_<((S, S', f)) &= (\delta_<(S \cup S'), \{q_0 \mid \text{if } \delta_<(S \cup S') \cap F \neq \emptyset\}, id), \\ \gamma'_<((S, S', f)) &= (S, S', f, <).\end{aligned}$$

For $> \in \Sigma_{-1}$:

$$\begin{aligned}\delta'_>((T, T', g), (S, S', f, <)) &= \\ &= (h_{<g>}(S), h_{<g>}(S') \cup \delta_>(T', \perp) \cup \{q_0 \mid \text{if } h_{<g>}(S \cup S') \cap F \neq \emptyset\}, h_{<g>} \circ f),\end{aligned}$$

where the function $h_{<g>} : Q \rightarrow Q$ is defined by

$$h_{<g>}(q) = \delta_>(g(\delta_<(q)), \nu_<(q)).$$

And if C 's stack is empty:

$$\delta'_>((T, T', g), \perp) = (\delta_>(T, \perp), \delta_>(T', \perp) \cup \{q_0 \mid \text{if } \delta_>(T \cup T', \perp) \cap F \neq \emptyset\}, id).$$

The correctness of the construction is formally claimed as follows:

Claim. Upon reading a prefix w of an input string, the automaton C should enter a state (p, S, S', f) , where

- S is the set of all such states $q \in Q$ that $w = uv$ for some $u \in L(B)^*$ and $v \notin (L_{\text{Dyck}} \cdot \Sigma_{-1})^*$ with $\delta(q_0, v) = q$;
- S' is the set of all such states $q \in Q$ that $w = uv$ for some $u \in L(B)^*$ and $v \in (L_{\text{Dyck}} \cup \Sigma_{-1})^*$ with $\delta(q_0, v) = q$;
- f is the behaviour of B on the longest well-nested suffix of w .

The set of accepting states of C contains all states (S, S', f) with $(S \cup S') \cap F \neq \emptyset$. Since, by the above claim, $S \cup S'$ is the set of all such states $q \in Q$ that $w = uv$ for some $u \in L(B)^*$ and $v \in \Sigma^*$ with $\delta(q_0, v) = q$, the automaton B will accept exactly the strings in $L(B)^*$.