Complexity of Input-Driven Pushdown Automata[1]

Alexander Okhotin[2] ***Kai Salomaa***[3]

1 Introduction

In an input-driven pushdown automaton (IDPDA), the current input symbol determines whether the automaton performs a push operation, a pop operation, or does not touch the stack. Input-driven pushdown automata, also known under alternative names of visibly pushdown automata and of nested word automata, have been intensively studied because of their desirable features: for instance, the model allows determinization, and the associated family of languages retains many of the strong closure and decidability properties of regular languages. This paper reports on various aspects of the complexity of input-driven pushdown automata, such as their *descriptional complexity*, the *computational complexity* of their membership problem and of other decision problems for input-driven languages.

The research on IDPDAs has been associated to their complexity from the very beginning. When Mehlhorn [25] originally introduced the model, it was studied as a subclass of deterministic context-free languages with better space complexity. Further work on the model carried out in the 1980s [6, 12, 36] concentrated on improving the bounds on the complexity of the languages accepted by such automata, culminating in the proof of their containment in NC^1. In 2004, the model was reintroduced by Alur and Madhusudan [2] under the name of *visibly pushdown automata*, and among their most important contributions were the first results on the descriptional complexity of the model, such as upper and lower bounds on the number of states in automata representing some operations on languages. Also, Alur and Madhusudan [2] established the computational complexity of several decision problems for the model.

The paper by Alur and Madhusudan [2] sparked a renewed interest in IDPDAs, and inspired the research on various aspects of the model [1, 8, 10, 19, 39]. Alur and Madhusudan [3] also introduced an equivalent outlook on IDPDAs as automata operating on *nested words*, which provide a natural model for applications such as XML document processing, where data has a dual linear-hierarchical structure. Nested word automata have been studied in a number of recent papers [9, 11, 17, 35, 37]. Another equivalent outlook on IDPDAs is represented by *pushdown forest automata* [14], that are, roughly speaking, tree walking automata that traverse the tree in depth-first left-to-right order and are equipped with a synchronized pushdown.

[2]Department of Mathematics and Statistics, University of Turku, Turku FI-20014, Finland. E-mail: `alexander.okhotin@utu.fi`. Supported by the Academy of Finland under grant 257857.

[3]School of Computing, Queen's University, Kingston, Ontario K7L 3N6, Canada. E-mail: `ksalomaa@cs.queensu.ca`. Supported by NSERC under grant OGP0147224.

Although the name "visibly pushdown automaton" has been more common in recent literature, this paper sticks to the original name of *input-driven pushdown automata,* which the authors feel to be more descriptive for this machine model. When referring to the work on nested word automata, all terminology is translated to IDPDAs without separate mention, that is, we talk about *states* and *stack symbols* of an IDPDA, instead of linear and hierarchical states of a nested word automaton.

This survey begins with the definitions of deterministic and nondeterministic input-driven automata (DIDPDA, NIDPDA), which are given in Section 2. The complexity tradeoff between DIDPDAs and NIDPDAs, explained in Section 3, consists of the following two results. First, there is the fundamental determinization construction due to von Braunmühl and Verbeek [6], which relies on the fact that all nondeterministic computation paths of an NIDPDA must use exactly the same sequence of push, pop and neutral operations, and thus allows simulating an n-state NIDPDA by a deterministic machine with 2^{n^2} states. Second, there is a matching $2^{\Omega(n^2)}$-state lower bound on this blow-up given by Alur and Madhusudan [3]. Section 3 also reports on the succinctness of some intermediate models between DIDPDAs and NIDPDAs, such as the unambiguous IDPDAs.

Descriptional complexity of operations on input-driven automata is presented in the next Section 4. Efficient representations of concatenation, Kleene star and reversal of input-driven automata are all based on a common fundamental construction, which allows a deterministic IDPDA to calculate the behaviour of another deterministic IDPDA on the last well-nested substring, as a function mapping states to states [27, 32]. Using this construction, Kleene star of an n-state DIDPDA is recognized by another DIDPDA with $n^{\Theta(n)}$ states, etc.; matching lower bounds are also known [35].

Computational complexity of input-driven languages, which, back in 1980s, was the original motivation for investigating this family, is described in Section 5. Mehlhorn [25] was the first to show that input-driven languages can be recognized by a polynomial-time algorithm using space $O\big(\frac{\log^2 n}{\log\log n}\big)$, as compared to space $O(\log^2 n)$ for the whole class of deterministic context-free languages [5]. This bound was improved to $O(\log n)$ space by von Braunmühl and Verbeek [6]. Later, a simpler algorithm with the same complexity was given by Rytter [36]; that algorithm is presented in this survey.

Another complexity aspect is the hardness of testing various decision problems for input-driven automata, which is presented in Section 6. Alur and Madhusudan [3] showed that the inclusion and equivalence problems for are EXPTIME-complete for nondeterministic IDPDA, while for deterministic IDPDA, they are decidable in polynomial time. The emptiness problem is in P for nondeterministic IDPDA, and Lange [19] showed it to be P-hard already for deterministic IDPDA.

2 Definitions

In the following, Σ denotes a finite alphabet and Σ^* is the set of strings over Σ, ϵ is the empty string and the length of a string $w \in \Sigma^*$ is $|w|$. The set of natural numbers is $\mathbb{N}$ and for $n \in \mathbb{N}$, $[0,n] = \{0,1,\ldots,n\}$. For any sets S and T, the set of partial functions from S to T is denoted by T^S. The cardinality of a finite set S is $|S|$.

In a transition of an input-driven pushdown automaton, the type of the current input symbol determines whether the automaton pushes onto the stack, pops from the stack, or does not touch the stack. The input alphabet of an input-driven pushdown alphabet is split into three parts as $\Sigma = \Sigma_{+1} \cup \Sigma_{-1} \cup \Sigma_0$, where the components Σ_{+1}, Σ_{-1} and Σ_0 are finite disjoint sets. Elements of Σ_{+1} Σ_{-1} and Σ_0 are referred to as *left brackets*, *right brackets* and *neutral symbols*, respectively. A string over Σ is *well-matched,* if every left bracket has a matching right bracket and vice versa.

2.1 Deterministic input-driven automata

Definition 1. *A deterministic input-driven pushdown automaton (DIDPDA) is a 7-tuple* $A = (\Sigma, Q, \Gamma, q_0, \bot, [\delta_a]_{a\in\Sigma}, F)$*, where*

- $\Sigma = \Sigma_{+1} \cup \Sigma_{-1} \cup \Sigma_0$ *is an input alphabet split into three disjoint classes;*
- Q *is a finite set of (internal) states of the automaton, with an initial state* $q_0 \in Q$ *and with a subset of accepting states* $F \subseteq Q$*;*
- Γ *is a finite pushdown alphabet, and a special symbol* $\bot \notin \Gamma$ *denotes an empty stack;*
- *the transition function by each left bracket symbol* $< \in \Sigma_{+1}$ *is a partial function* $\delta_<\colon Q \to Q \times \Gamma$*, which, for a given current state, provides the next state and the symbol to be pushed onto the stack;*
- *for every right bracket symbol* $> \in \Sigma_{-1}$*, a partial function* $\delta_>\colon Q \times (\Gamma \cup \{\bot\}) \to Q$ *specifies the next state, assuming that the given stack symbol is popped from the stack or the stack is empty;*
- *for a neutral symbol* $c \in \Sigma_0$*, the state change is described by a partial function* $\delta_c\colon Q \to Q$*.*

A configuration of A *is a triple* (q, w, x)*, where* $q \in Q$ *is the state,* $w \in \Sigma^*$ *is the remaining input and* $x \in \Gamma^*$ *is the stack contents. The initial configuration on an input string* $w_0 \in \Sigma^*$ *is* (q_0, w_0, ϵ)*. For each configuration with at least one remaining input symbol, the next configuration is uniquely determined by a single step transition function defined as follows:*

- *for each left bracket* $< \in \Sigma_{+1}$*, let* $(q, <w, x) \vdash_A (q', w, \gamma x)$*, where* $\delta_<(q) = (q', \gamma)$*;*
- *for every right bracket* $> \in \Sigma_{-1}$*, let* $(q, >w, \gamma x) \vdash_A (\delta_>(q, \gamma), w, x)$*, and in case the stack is empty at this point, let* $(q, >w, \epsilon) \vdash_A (\delta_>(q, \bot), w, \epsilon)$*;*
- *for a neutral symbol* $c \in \Sigma_0$*, define* $(q, cw, x) \vdash_A (\delta_c(q), w, x)$*.*

Once the input string is exhausted, the last configuration (q, ϵ, u) *is accepting if* $q \in F$*, regardless of the stack contents. The language* $L(A)$ *recognized by the automaton is the set of all strings* $w \in \Sigma^*$*, on which the computation from* (q_0, w, ϵ) *is accepting.*

A DIDPDA can accept strings with unmatched left brackets, resulting the stack being non-empty at the end of the computation. Unmatched right brackets are handled by special transitions with the empty-stack marker $\bot$ used as the second argument in place of a stack symbol.

Note that the next computation step depends on the top-of-stack symbol only when the input symbol is a right bracket and the stack is popped. In other words, when reading a left bracket or a neutral symbol, the computation step does not depend on the stack contents. Naturally, the machine could be made to remember the top stack symbol in the state, however, this would change the descriptional complexity results associated with the model.

If the alphabet contains only neutral symbols, that is, if $\Sigma_{+1} = \Sigma_{-1} = \emptyset$, then a DIDPDA becomes a deterministic finite automaton (DFA).

Example 1. *Consider the alphabet* $\Sigma = \Sigma_{+1} \cup \Sigma_{-1} \cup \Sigma_0$, *with* $\Sigma_{+1} = \{a\}$, $\Sigma_{-1} = \{b, c\}$ *and* $\Sigma_0 = \emptyset$. *Then the language* $L = \{\, a^n b^n \mid n \geqslant 2 \,\} \cup \{\, a^n c^n \mid n \geqslant 2 \,\}$ *is recognized by a DIDPDA* $A = (\Sigma, Q, \Gamma, q_0, \bot, [\delta_s]_{s\in\Sigma}, F)$, *with the set of states* $Q = \{q_0, q_1, q_b, q_c, q_{acc}\}$ *and the pushdown alphabet* $\Gamma = \{\gamma_0, \gamma_1\}$. *In the initial state* q_0, *the automaton reads the first* a *and pushes* γ_0: $\delta_a(q_0) = (q_1, \gamma_0)$. *All subsequent symbols* a *are read in the state* q_1, *where the stack symbol pushed is* γ_1: $\delta_a(q_1) = (q_1, \gamma_1)$. *Once the automaton reads the first* b *or* c, *it enters a state, in which it will only read symbols of the same type:* $\delta_b(q_1, \gamma_1) = q_b$, $\delta_c(q_1, \gamma_1) = q_c$. *Each remaining symbols* b *or* c *is matched to the corresponding* a *by popping a stack symbol:* $\delta_b(q_b, \gamma_1) = q_b$, $\delta_c(q_c, \gamma_1) = q_c$. *Once the last matching* b *or* c *is read, the automaton knows that by the symbol* γ_0, *and accordingly enters the unique accepting state:* $\delta_b(q_b, \gamma_0) = \delta_c(q_c, \gamma_0) = q_{acc}$.

2.2 Nondeterministic input-driven automata

In nondeterministic automata, the transition function is multi-valued, and accordingly, there may be multiple computations on the same input. A string is considered accepted, if at least one of these computations is accepting.

Definition 2. *A nondeterministic input-driven pushdown automaton (NIDPDA) is defined as a 7-tuple* $B = (\Sigma, Q, \Gamma, Q_0, \bot, [\delta_a]_{a\in\Sigma}, F)$, *in which the input alphabet* $\Sigma = \Sigma_{+1} \cup \Sigma_{-1} \cup \Sigma_0$, *the set of states* Q, *the pushdown alphabet* Γ, *the symbol* $\bot \notin \Gamma$ *for the empty stack, and the set of accepting states* $F \subseteq Q$ *are as in Definition 1, and*

- *there is a set of initial states* $Q_0 \in Q$, *and a computation may begin from any of them;*
- *for each left bracket symbol* $< \in \Sigma_{+1}$, *the transition function* $\delta_<\colon Q \to 2^{Q\times\Gamma}$ *provides, for a given current state, a set of possible outcomes, which are pairs of the next state and the stack symbol to be pushed;*
- *for every right bracket symbol* $> \in \Sigma_{-1}$, *there is a function* $\delta_>\colon Q \times (\Gamma \cup \{\bot\}) \to 2^Q$ *that lists all possible next states, if the given stack symbol is popped from the stack;*
- *for a neutral symbol* $c \in \Sigma_0$, *there is a function* $\delta_c\colon Q \to 2^Q$.

A configuration of B *is again a triple* (q, w, x), *with* $q \in Q$, $w \in \Sigma^*$ *and* $x \in \Gamma^*$. *On an input string* $w_0 \in \Sigma^*$, *all configurations* (q_0, w_0, ϵ) *with* $q_0 \in Q_0$ *are initial. The transition relation is defined as follows:*

- *for each left bracket* $< \in \Sigma_{+1}$, *and for all pairs* (q', γ) *in* $\delta_<(q)$, *let* $(q, <w, x) \vdash_A (q', w, \gamma x)$;
- *for every right bracket* $> \in \Sigma_{-1}$, *and for all* $q' \in \delta_>(q, \gamma)$, *let* $(q, >w, \gamma x) \vdash_A (q', w, x)$, *and for the empty stack, for all* $q' \in \delta_>(q, \bot)$, *let* $(q, >w, \epsilon) \vdash_A (q', w, \epsilon)$;
- *for a neutral symbol* $c \in \Sigma_0$, *for every* $q' \in \delta_c(q)$, *let* $(q, cw, x) \vdash_A (q', w, x)$.

The last configuration (q, ϵ, u) *is accepting if* $q \in F$. *The language* $L(A)$ *recognized by the automaton is the set of all strings* $w \in \Sigma^*$, *on which at least one computation from* (q_0, w, ϵ) *is accepting.*

An NIDPDA B becomes a DIDPDA, if $|Q_0| = 1$ and its transition functions δ_a, $a \in \Sigma$, give at most one possible action for each input symbol, state and top stack symbol. If there are only neutral symbols in the alphabet ($\Sigma_{+1} = \Sigma_{-1} = \emptyset$), then an NIDPDA is a nondeterministic finite automaton (NFA).

2.3 Input-driven grammars

In the literature, input-driven automata originally emerged as a general case for a series of weaker models defined in terms of formal grammars. These grammars, which are all special cases of context-free grammars, are defined with a terminal alphabet $\Sigma = \Sigma_{+1} \cup \Sigma_{-1} \cup \Sigma_0$, and their rules observe the well-nestedness of brackets.

The first such model were the *parenthesis grammars* of McNaughton [23], in which the alphabet contains one left bracket ($\Sigma_{+1} = \{<\}$) and one right bracket ($\Sigma_{-1} = \{>\}$), and each rule must be either of the form $A \to <\alpha>$, where $\alpha \in (\Sigma_0 \cup N)^*$, or of the form $A \to w$, with $w \in \Sigma_0^*$. A similar model, called *bracketed grammars*, was defined by Ginsburg and Harrison [15]: in their model, each bracket in Σ_{+1} or in Σ_{-1} is marked with a rule r, and then r must be of the form $A \to <_r \alpha>_r$. Later, Berstel and Boasson [4] studied *balanced grammars*, in which there is a bijection between Σ_{+1} and Σ_{-1}, so that a rule $A \to <\alpha>$ must use a pair of corresponding brackets.

All these grammar models generate subclasses of the input-driven languages. Furthermore, input-driven pushdown automata have have a direct representation by grammars, which are isomorphic to NIDPDA up to some insignificant details.

Definition 3 ([2])**.** *An input-driven grammar is a quadruple $G = (\Sigma, N, R, S)$, where*

- $\Sigma = \Sigma_{+1} \cup \Sigma_0 \cup \Sigma_{-1}$ *is the alphabet, split into three disjoint classes;*
- N *is the set of nonterminal symbols;*
- R *is the set of rules, each of the form $A \to <B>C$, $A \to aC$ or $A \to \epsilon$, with $A, B, C \in N$, $< \in \Sigma_{+1}$, $> \in \Sigma_{-1}$ and $a \in \Sigma$;*
- $S \in N$ *is the initial symbol.*

Under this definition, input-driven grammars define only well-nested strings. Alur and Madhusudan [2] gave a slightly relaxed definition that also applies to ill-nested strings.

3 Succinctness tradeoffs between models

3.1 Fundamental construction I: determinization

One of the most important facts about input-driven automata is that their nondeterministic variant can be determinized. This possibility relies on the property that all nondeterministic computations on the same input string must use exactly the same sequence of push and pop operations. A simulating deterministic IDPDA follows the same sequence of stack operations, and can trace all possible computations of the nondeterministic IDPDA. Differing from the well-known *subset construction* for finite automata, this simulation keeps track of a *set of pairs of states* of the nondeterministic machine (rather than just a subset of the set of states), where a pair (p, q) refers to a computation on a substring that begins in state p and ends in state q.

Theorem 1 (von Braunmühl and Verbeek [6])**.** *An NIDPDA over an alphabet $\Sigma = \Sigma_{+1} \cup \Sigma_{-1} \cup \Sigma_0$ with n states and with any number of stack symbols can be simulated by a DIDPDA with 2^{n^2} states and $|\Sigma_{+1}| \cdot 2^{n^2}$ stack symbols.*

Notably, the number of stack symbols in the original NIDPDA does not affect the size of the simulating DIDPDA, because the latter never stores those stack symbols.

Proof. Let $A = (\Sigma, Q, \Gamma, Q_0, \bot, [\delta_a]_{a\in\Sigma}, F)$ be an NIDPDA. Construct a DIDPDA $B = (\Sigma, Q_B, \Gamma_B, q_B^0, \bot, [\tau_a]_{a\in\Sigma}, F_B)$, with the set of states $Q_B = 2^{Q\times Q}$, and with the stack alphabet $\Gamma_B = \Sigma_{+1} \times 2^{Q\times Q}$, as follows. Every state $P \subseteq Q \times Q$ of B contains pairs of states of A, each corresponding to the following situation: whenever $(p, q) \in P$, both p and q are states in one of the computations of A, where p was reached just after reading the most recent left bracket, whereas q is the current state of that computation.

The initial state of B, defined as $q_B^0 = \{(q, q) \mid q \in Q_0\}$, represents the behaviour of A on the empty string, which begins its computation in an initial state, and remains in the same state. The set of accepting states reflects all computations of A ending in an accepting state:

$$F_B = \{P \subseteq Q \times Q \mid \text{ there is a pair } (p, q) \in P \text{ with } q \in F\}.$$

The transition functions τ_a, with $a \in \Sigma$, are defined as follows.

- On a left bracket $< \in \Sigma_{+1}$, the transition in a state $P \in Q_B$ is $\tau_<(P) = (P', (<, P))$, where

  $$P' = \{(q', q') \mid (\exists (p, q) \in P)(\exists \gamma \in \Gamma) : (q', \gamma) \in \delta_<(q)\}.$$

  Thus, B pushes the current context of the simulation onto the stack, along with the current left bracket, and starts the simulation afresh at the next level of brackets, where it will trace the computations from all states q' reachable by A at this point.

- For a neutral symbol $c \in \Sigma_0$ and a state $P \in Q_B$, the transition $\tau_c(P) = \{(p, q') \mid \exists (p, q) \in P : q' \in \delta_c(q)\}$ directly simulates one step of A in all currently traced computations.

- For a right bracket $> \in \Sigma_{-1}$ and a state $P' \subseteq Q_B$, the automaton pops a stack symbol $(<, P) \in \Gamma_B$ containing a matching left bracket and the context of the previous simulation. Then, each computation in P is continued by simulating the transition by the left bracket, the behaviour inside the brackets stored in P', and the transition by the right bracket.

  $$\tau_>(P', (<, P)) = \{(p, q'') \mid (\exists (p, q) \in P)(\exists (p', q') \in P')(\exists \gamma \in \Gamma) : (p', \gamma) \in \delta_<(q), q'' \in \delta_>(q', \gamma)\}.$$

- For an unmatched right bracket $> \in \Sigma_{-1}$, the transition in a state $P \in Q_B$ advances all currently simulated computations of A in the same way as for a neutral symbol: $\tau_>(P, \bot) = \{(p, q') \mid \exists (p, q) \in P : q' \in \delta_>(q', \bot)\}$.

The correctness of the construction can be proved by induction on the bracket structure of an input string. □

3.2 Lower bound on determinization

The following lower bound on the size blow-up of determinization is precise up to a multiplicative constant in the exponent (as compared to the upper bound in Theorem 1).

Theorem 2 (Alur and Madhusudan [3])**.** *Let Σ be an alphabet, with $\Sigma_{+1} = \{<\}$, $\Sigma_{-1} = \{>\}$ and $\Sigma_0 = \{a, \#, \$\}$. Then, for each $n \geqslant 1$, there exists a language L_n over Σ, which is recognized by an NIDPDA with $8n + 1$ states and n stack symbols, while every DIDPDA for L_n needs at least 2^{n^2} states.*

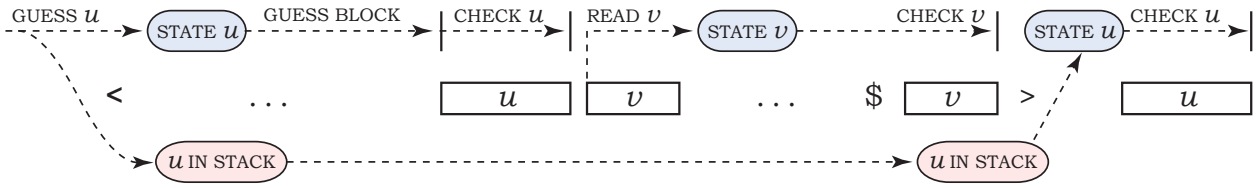

Figure 1: Recognizing the language L_n by an NIDPDA with $O(n)$ states.

Proof. For each $n \geqslant 1$, define the language L_n as

$$L_n = \big\{ <u_1\#v_1\#u_2\#v_2\#\ldots\#u_\ell\#v_\ell\$v>u \;\big|\; \ell \geqslant 1,\ u_i, v_i \in a^* \text{ for all } i \in \{1,\ldots,\ell\},$$
$$u, v \in a^*, 0 \leqslant |u|, |v| \leqslant n-1, \text{and there exists } t \in \{1,\ldots,\ell\} \text{ with } u = u_t,\ v = v_t\big\}.$$

An NIDPDA A with $O(n)$ states and n stack symbols recognizes the language L_n as follows. At the first step of the computation, upon reading the left bracket $<$, the automaton guesses the length of the unary string u and pushes its length $|u|$ onto the stack as a single symbol, as well as stores it in its internal state. Then the automaton nondeterministically decides to skip any even number of blocks, verifies that the next unary string is u, and then stores the following unary string v (after the #-marker) in its internal state. After skipping until the dollar sign, the automaton verifies that the last string inside the brackets is exactly v. Finally, upon reading the right bracket $>$, the automaton pops $|u|$ from the stack and compares the string $a^{|u|}$ to the remaining substring, to verify that they are the same. The data flow in such a computation is illustrated in Figure 1.

At each moment, the automaton needs to remember one number from 1 to n and the current stage of processing. As seen from Figure 1, there are six stages in total: holding u, comparing u to the current block, reading v, holding v, comparing v to the last block inside brackets, and checking u in the end. Two of them (holding u and holding v) also require counting the parity of the number of blocks. Hence, the construction uses $8n + 1$ states.

It remains to establish the lower bound on the number of states in any DIDPDA for L_n. For any relation $R \subseteq [0, n-1] \times [0, n-1]$, denote

$$w_R = u_1\#v_1\#u_2\#v_2\#\cdots u_{|R|}\#v_{|R|}, \qquad u_i, v_i \in a^*,\ 1 \leqslant i \leqslant |R|,$$

where the pairs $(|u_i|, |v_i|)$, with $i = 1, \ldots, |R|$, list the elements of R in an arbitrary order.

Let B be any DIDPDA recognizing L_n, and let γ be the symbol pushed by B upon reading the left bracket $<$ in the initial state. For any relation $R \subseteq [0, n-1] \times [0, n-1]$, consider the computation of B on the input $<w_R$. As $<w_R$ is a prefix of some strings accepted by B, the computation successfully reaches some state q_R, with γ in the stack.

It is claimed that all states q_R, with $R \subseteq [0, n-1] \times [0, n-1]$, are pairwise distinct. For the sake of contradiction, assume that $q_{R_1} = q_{R_2}$ for two different relations $R_1, R_2 \subseteq [0, n-1] \times [0, n-1]$. Let (r, s) be a pair belonging to one of these relations and not to the other. As the configurations of B after reading $<w_{R_1}$ and $<w_{R_2}$ are identical, B accepts $<w_{R_1}\$a^s>a^r$ if and only if it accepts $<w_{R_2}\$a^s>a^r$. However, one of these strings is in L_n, and the other is not in L_n. The contradiction establishes that B must have at least 2^{n^2} states. □

The upper bound in Theorem 1 and the lower bound in Theorem 2 are tight up to a multiplicative constant in the exponent. The lower bound constant (roughly, $\frac{1}{64}$) can be improved by

increasing the size of the alphabet, but it remains open whether the precise bound 2^{n^2} can be reached by languages defined over a fixed alphabet.

The languages L_n in Theorem 2 are defined in a way that prevents a DIDPDA from making any meaningful use of the stack: that is, L_n can, in fact, be recognized by a DFA with 2^{n^2} states. Okhotin et al. [30] presented more a refined argument on the size blow-up of determinization, that includes lower bounds on both the number of states and the number of stack symbols.

3.3 IDPDA with restricted nondeterminism

A nondeterministic input-driven pushdown automaton A is *unambiguous* (UIDPDA) if it has exactly one accepting computation on each string w it accepts, that is, a unique sequence of configurations $C_1, C_2, \ldots, C_m$ where C_1 is the initial configuration of A on w, $C_i \vdash_A C_{i+1}$ for $1 \leqslant i \leqslant m-1$, and C_m is of the form (q, ε, u) where $q \in F$, $u \in \Gamma^*$.

The same $2^{\Omega(n^2)}$ lower bound as for determinization holds also for converting an NIDPDA to a UIDPDA and for converting a UIDPDA to a deterministic machine [31]. The situation is analogous with the known descriptional complexity results for finite automata: simulating a general n-state NFA with an unambiguous finite automaton (UFA) requires, in the worst case, $2^n - 1$ states, and the UFA to DFA trade-off is 2^n [20].

Theorem 3 (Okhotin and Salomaa [31])**.** (a) *For every $n \geqslant 1$, there exists a language K_n over a fixed alphabet Σ, recognized by a UIDPDA with $O(n)$ states and n stack symbols, such that every DIDPDA for K_n needs at least $2^{n^2} - 1$ states.*

(b) *For every $n \geqslant 1$, there exists a language K'_n over a fixed alphabet Σ recognized by an NIDPDA with $5n + 1$ states and n stack symbols, such that if A is an arbitrary UIDPDA for K'_n, then the product of the number of states and the number of stack symbols of A is at least $2^{\lfloor \frac{n^2}{2} \rfloor}$.*

A different restriction on NIDPDAs considered in the literature limits the total number of nondeterministic paths in any computation. An analogous notion of limited nondeterminism was considered for NFAs under the name of "leaf size" [18] or "tree width" [34].

Let A be an NIDPDA. The computation tree of A on an input w, $T_{A,w}$, has its root labeled with the initial configuration of A on w, and a node labeled with a configuration C has finitely many children labeled with all configurations C', such that $C \vdash_A C'$. Then A is said to be a *k-path NIDPDA,* where $k \in \mathbb{N}$, if for every string w the computation tree $T_{A,w}$ has at most k leaves.

A k-path NIDPDA can be converted to a DIDPDA with only a polynomial size blow-up; however, converting an NIDPDA to a k-path NIDPDA entails the same worst-case size blow-up as determinization.

Theorem 4 (Okhotin and Salomaa [33])**.** *A k-path NIDPDA with n states and m stack symbols can be simulated by a DIDPDA with $\sum_{i=1}^{k} (n+1)^i \cdot i^i$ states and $\sum_{i=1}^{k} m^i$ stack symbols.*

For every $n \in \mathbb{N}$ there exists a language L_n recognized by an NIDPDA with $O(n)$ states and n stack symbols, such that any k-entry NIDPDA for L_n needs at least $2^{n^2} - 1$ states.

The construction needed for the first part of Theorem 4 is considerably more involved than in the case of finite automata [34]. When a DIDPDA B currently simulating $k_1 < k$ computations of a k-path NIDPDA A encounters a left bracket, it pushes k_1 stack symbols of A. Inside the brackets, the simulated computations of A may involve further nondeterministic choices, and then some k_2 states arriving at the matching right bracket have to be paired up with the k_1 symbols popped from the stack. The extra data needed for pairing accounts for the i^i factor in the number of states.

4 Descriptional complexity of operations

Descriptional complexity of language operations deals with questions of the type: "Given DIDPDAs A and B, what is the worst-case size of a DIDPDA for the concatenation of $L(A)$ and $L(B)$?" Such questions are addressed by presenting constructions applicable to all automata (which imply upper bounds on the worst-case size), and by finding such argument automata that every automaton for the result of the operation requires a certain number of states. This section presents the known bounds for several operations. For a few important operations (namely, for concatenation, Kleene star and reversal), the upper bounds are based on a single construction presented below.

4.1 Fundamental construction II: calculating the behaviour

The determinization construction in Theorem 1 works by tracing all possible computations of an NIDPDA on each level of brackets. On an input string $u{<}v$, where v is well-nested, the determinization requires remembering the set of such pairs (p, q), that the original NIDPDA can begin reading v in the state p and end in the state q. This is possible, because an input-driven automaton finishes reading a well-nested substring with the same stack contents as in the beginning, without ever consulting the original stack contents.

Consider calculating the same kind of data for a given *deterministic* IDPDA (rather than NIDPDA). Then, for a string $u{<}v$, where v is well-nested, and for each state p, the automaton has a single computation on v beginning with p, and hence, instead of remembering a *relation* on the set of states, it is sufficient to remember a *(partial) function* $f_v \colon Q \to Q$, which maps the original state of the automaton to its state after processing the string v. The following lemma shows how to transform any given n-state DIDPDA A into another DIDPDA C with n^n states and $n^n \cdot |\Sigma_{+1}|$ stack symbols, which calculates the behaviour function of A on each level of brackets.

Denote the set of all partial functions from Q to Q by Q^Q.

Lemma 1 (Okhotin and Salomaa [32])**.** *For every DIDPDA A with the set of states Q and the stack alphabet Γ, there exists a DIDPDA C with the set of states Q^Q and the stack alphabet $Q^Q \times \Sigma_{+1}$, that on an input w reaches a state $f \in Q^Q$ that is the behaviour of A on the longest well-nested suffix of w.*

Proof. Let $A = (\Sigma, Q, \Gamma, q_0, \bot, [\delta_a]_{a\in\Sigma}, F)$ be the given DIDPDA. The initial state of C is $q'_0 = id$, the identity function on Q. This is the behaviour on ϵ.

The transition function of C on input $a \in \Sigma$ is denoted as δ'_a. On a neutral symbol $c \in \Sigma_0$, the transition function calculates the composition of the behaviour on the previously read string with the behaviour on the current symbol: $\delta'_c(f) = \delta_c \circ f$.

Consider how C calculates the behaviour of A on a substring enclosed in brackets. Let u be a well-nested substring, followed by another well-nested substring $<v>$, as shown in Figure 2. Let $f \colon Q \to Q$ be the behaviour of A on u, let $g \colon Q \to Q$ be the behaviour of A on v, and finally denote by $h \colon Q \to Q$ the behaviour of A on $<v>$. In Figure 2(left), $f(q_1) = q$, $g(q_2) = q_3$ and $h(q) = q'$.

By the time C reaches the left bracket $<$, it should compute the behaviour on u and store it in its internal state. Upon reading the left bracket $<$, the automaton C stores the current context of the simulation in the stack and turns to calculating the behaviour on the inner level of brackets, beginning with the identity function on Q. To this end, the transition in the state $f \colon Q \to Q$ by the left bracket $< \in \Sigma_{+1}$, is defined as $\delta'_<(f) = (id, (f, <))$.

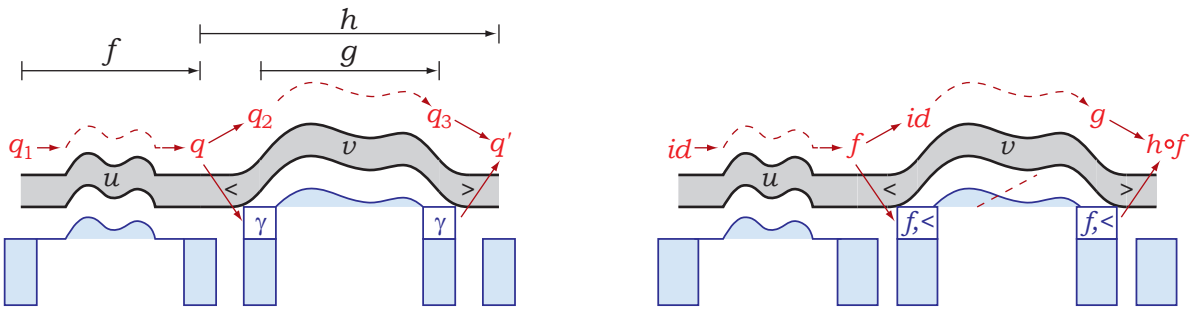

Figure 2: How the behaviour of an IDPDA A (left) on a well-nested string $u{<}v{>}$ is calculated by an IDPDA C (right).

Next, as illustrated in Figure 2, the automaton C reads the well-nested substring v and calculates A's behaviour on this substring, that is, the function g. When C reaches the right bracket $>$, it has g in the current state and it pops the behaviour on u from the stack. Then C can calculate the behaviour on the longer well-nested substring $<v>$ as a function $h\colon Q \to Q$, defined as follows: for each state $q \in Q$, let $\delta_<(q) = (q', \gamma)$ and set $h(q) = \delta_>\big(g(q'), \gamma\big)$. Then the desired behaviour of A on $u{<}v{>}$ is obtained by composing the behaviour on u with the behaviour on $<v>$, which is implemented by the transition $\delta'_>(g, (f, <)) = h \circ f$.

When the automaton C reads an unmatched right bracket $>$, it reaches a new lowest level of brackets, and knows this fact, because it sees a bottom marker $\perp$ instead of a stack symbol. Then it begins calculating a new behaviour from the identity function: $\delta'_>(g, \perp) = id$. □

The above construction did not specify the accepting states of C. If a state $f \in Q^Q$ is set to be accepting when $f(q_0) \in F$, then C recognizes $L(A)$. Thus, a deterministic automaton is transformed to another deterministic automaton, which, however, collects more data about possible computations beginning at different positions. These data shall be used in the constructions below.

4.2 Concatenation, Kleene star and reversal

The straightforward constructions establishing closure under concatenation, star and reversal proceed by creating an NIDPDA for the resulting language and then determinizing it. For the concatenation of DIDPDAs with m and n states, this yields a DIDPDA with $2^{(m+n)^2}$ states. However, this construction is non-optimal, and there is a more efficient construction based on calculating the behaviour of the DIDPDA, as in Lemma 1.

This construction is given here for the case when both automata are assumed to accept only well-nested strings.

Lemma 2 (Okhotin and Salomaa [32])**.** *Let A and B be DIDPDAs over an alphabet Σ that accept only well-nested strings, let P and Q be their respective sets of states, let Γ and Ω be their stack alphabets. Then there exists a DIDPDA C with the set of states $P \times (2^Q \cup Q^Q)$ and the stack alphabet $\Gamma \times (2^Q \cup Q^Q) \times \Sigma_{+1}$ that recognizes the language $L(A) \cdot L(B)$.*

Sketch of a proof. Each state of an IDPDA C recognizing the concatenation is comprised of two components. The first component is a state of the automaton A, which C continuously simulates, so that, after reading any prefix w, the first component of C's state is exactly the state A reaches

after reading w. Whenever A pushes or pops a stack symbol $\gamma \in \Gamma$, the new automaton C pushes or pops a triple with γ in the first component. Thus, C knows all that A knows.

Since A and B accept only well nested strings, for every concatenation uv, with $u \in L(A)$ and $v \in L(B)$, the splitting point between u and v is at the outer level of brackets; accordingly, C keeps track of whether the current level of brackets is the outer level. On the outer level of brackets—that is, after reading a well-nested prefix of the input—C uses states from $P \times 2^Q$, and *inside any brackets*—or, in other words, after reading an ill-nested prefix—its states are from $P \times Q^Q$.

Whenever C is inside any brackets, it calculates the behaviour of B on the current level, exactly as described in Lemma 1. At the same time, C maintains the ongoing simulation of A. This is done in states of the form (p, f), where $p \in P$ is the current state of A, and $f\colon Q \to Q$ is the behaviour of B on the longest well-nested string on the current level of brackets. There is no interaction between simulating A and tracing the behaviour of B.

On the outer level of brackets, C implements the subset construction, as in the NFA to DFA transformation. First of all, C monitors whether A accepts the prefix read so far. If A would accept it, then, at this point, C starts the simulation of a new instance of B. This is implemented in states of the form (p, S), where $p \in P$ is the current state of A, while the set $S \subseteq Q$ contains the states of all currently running instances of B. When C reads a neutral symbol $c \in \Sigma_0$, it applies B's transition function by c to every element of S. When C encounters a bracketed structure, it stores the current value of S in the stack and turns to calculating the behaviour of B on the inner level of brackets. Once the bracketed structure is read and the desired behaviour function is calculated, C restores S from the stack and applies that behaviour function to every element of S. Once the entire input string is read, the string belongs to the concatenation $L(A) \cdot L(B)$ if and only if one of the running instances of B accepts at this point. This is set as the acceptance condition of C. □

If A and B are allowed to accept also ill-nested strings, the construction incurs the following complications. Consider a concatenation of two ill-nested strings, $u \in L(A)$ and $v \in L(B)$. The first string u may contain unmatched left brackets, and if so, then the middle point between u and v in the concatenation uv occurs inside some brackets (in contrast to the well-nested case, where the middle point is always at the outer level of brackets). Furthermore, the second string v may contain unmatched right brackets, which will then match the unmatched left brackets in u.

In order to handle these possibilities, a DIDPDA recognizing the concatenation of $L(A)$ and $L(B)$ should expect the middle point to occur at every position in the input. Every time it considers a middle point inside matching brackets, these brackets are unmatched in the strings being concatenated. Thus, the simulating automaton should track the behaviour of A and B on unmatched brackets, while reading matching brackets. To this end, the simulating automaton uses a different set of states than in the proof of Lemma 2, but its size is not much larger.

Together with a matching lower bound result [32, 35], this implies that, roughly speaking, the descriptional complexity of the concatenation of two DIDPDAs of size m and n, is $m \cdot 2^{\Theta(n \cdot \log n)}$. The more precise statement is given in the theorem below.

Theorem 5 (Okhotin and Salomaa [32])**.** *Let A be a DIDPDA with m states and k stack symbols, and let B be a DIDPDA with n states and any number of stack symbols, both defined over the same alphabet Σ. Then the concatenation $L(A) \cdot L(B)$ is recognized by a DIDPDA with $m4^n(n+1)^n$ states and $|\Sigma_{+1}| \cdot k4^n(n+1)^n$ stack symbols.*

Conversely, for all $m, n \geqslant 1$, there exists a pair of languages K_m and L_n over a fixed alphabet Σ, where K_m is recognized by a DIDPDA with $O(m)$ states and m stack symbols, and L_n is recognized

by a DIDPDA with $O(n)$ states and n stack symbols, while every DIDPDA for their concatenation $K_m L_n$ requires at least $m \cdot n^n$ states.

The tight upper bound constructions for Kleene star and reversal are also based on Lemma 1. For simplicity, the result is stated below just in terms of the size of an IDPDA, defined as the sum of the number of states and of the number of stack symbols.

Theorem 6 (Okhotin and Salomaa [32], lower bounds from Piao and Salomaa [35])**.** *If A is a DIDPDA of size n, the Kleene star and the reversal of $L(A)$ can be recognized by a DIDPDA of size $2^{\Theta(n \cdot \log n)}$. Furthermore, for each operation, size $2^{\Theta(n \cdot \log n)}$ is necessary in the worst case, and the lower bound examples can be constructed over a fixed alphabet.*

4.3 Boolean operations

The input-driven languages are closed under all Boolean operations, and the constructions implementing these operations are similar to those used for finite automata.

Union and intersection of input-driven automata are implemented using the *direct product construction*. Consider the deterministic case (the same idea applies to NIDPDAs).

Lemma 3 (Alur and Madhusudan [2])**.** *Let A and B be DIDPDAs over a common alphabet, let P and Q be their sets of states, and let Γ and Ω be their stack alphabets. Then there exist DIDPDA C and C', each defined with the set of states $(P \cup \{-\}) \times (Q \cup \{-\})$ and with the stack alphabet $\Gamma \times \Omega$, that recognize the languages $L(A) \cup L(B)$ and $L(A) \cap L(B)$, respectively.*

Sketch of a proof. The initial states and transition functions are the same in C and in C'. These automata carry out two independent simulations of A and B on the given input string. Whenever the simulated A and B encounter a left bracket and have to push a symbol each, C pushes a pair, and then pops it at the matching right bracket. Once the string is consumed, C accepts if A or B is in an accepting configuration, and C' accepts if both A and B accept. □

An almost matching lower bound is known in the case of intersection.

Lemma 4 (Piao and Salomaa [35])**.** *Let $n_1, n_2, k_1, k_2 \geqslant 2$ be any integers, where each k_i divides n_i. Then there exists a pair of DIDPDAs A_1, A_2, where each A_i has n_i states and k_i stack symbols, such that every DIDPDA recognizing the intersection $L(A_1) \cap L(A_2)$ must have at least $n_1 n_2$ states and at least $k_1 k_2$ stack symbols.*

Piao and Salomaa [35] also gave a lower bound on the size of a DIDPDA recognizing the union of two languages, but it is not equally precise as the lower bound for intersection.

Turning to the complementation operation, here the situation with deterministic and nondeterministic automata is different. For a DIDPDA, just like for a DFA, it is sufficient to exchange its accepting and rejecting states to obtain an automaton recognizing the complement of the original language. However, this does not work for an NIDPDA (just like it does not work for an NFA), and complementing an NIDPDA involves, roughly speaking, the same size blow-up as determinization.

Theorem 7. *For every language recognized by an n-state NIDPDA its complement is recognized by an NIDPDA with 2^{n^2} states and $O(2^{n^2})$ stack symbols. At the same time, for every $n \geqslant 1$, there exists a language recognized by an NIDPDA with $O(n)$ states, such that for every NIDPDA recognizing its complement, the sum of the number of states and the number of stack symbols of this NIDPDA is at least $2^{\frac{n^2}{2}+1}$.*

	DFA	UFA	NFA	DIDPDA	NIDPDA
$\cup$	mn	?	$m+n+1$	$\Theta(mn)$ [35]	$m+n+O(1)$ [2]
$\cap$	mn	mn [28]	mn	$\Theta(mn)$ [35]	$\Theta(mn)$ [17]
$\sim$	n	$n^{2-o(1)} \leqslant \cdot \leqslant 2^n$ [28]	2^n	n	$2^{\Theta(n^2)}$ [31]
$\cdot$	$m \cdot 2^n - 2^{n-1}$	?	$m+n$	$m2^{\Theta(n \log n)}$ [32]	$m+n+O(1)$ [2]
^{2}	$n \cdot 2^n - 2^{n-1}$	?	$2n$	$2^{\Theta(n \log n)}$ [32]	$n+O(1)$ [2]
*	$\frac{3}{4}2^n$	$(n-1)^2 \leqslant \cdot \leqslant \frac{3}{4}2^n$ [28]	$n+1$	$2^{\Theta(n \log n)}$ [32]	$n+O(1)$ [2]
^{R}	2^n	n	$n+1$	$2^{\Theta(n \log n)}$ [32]	$n+O(1)$ [2]

Table 1: State complexity of union ($\cup$), intersection ($\cap$), complementation ($\sim$), concatenation ($\cdot$), squaring (^{2}), Kleene star (*) and reversal (^{R}) for finite automata (DFA, UFA, NFA) and for input-driven automata (DIDPDA, NIDPDA). For finite automata and for DIDPDA, the table gives the worst-case number of states, if m and n are the number of states in the argument automata. The column for NIDPDA is given with respect to the complexity measure $|Q| + |\Gamma|$. References for the results for DFA and NFA can be found in the survey article [13].

4.4 Summary of descriptional complexity of operations

The state complexity of all basic operations on both deterministic and nondeterministic IDPDA is now known. Table 1 summarizes the results and compares them with results for finite automata.

Investigating the complexity of operations on the family of *unambiguous IDPDAs* [31] is suggested for future work. Since descriptional complexity questions are already difficult for unambiguous finite automata (UFA), this task might be nontrivial as well.

5 Computational complexity of input-driven languages

It is well-known that every context-free language lies in the complexity class NC^2, that is, can be recognized by a uniform family of Boolean circuits of depth $O(\log^2 n)$ and with $poly(n)$ gates. The main subclasses of the context-free languages, such as the linear languages, the deterministic languages, etc., are important, in particular, for their lower computational complexity; these results are summarized in a recent survey paper [29, Sect. 8.1].

The notion of an input-driven automaton emerged in the study of computationally easy context-free languages, and was predated by efficient algorithms for more restricted language families. A logarithmic-space algorithm for recognizing Dyck languages, due to Hotz and Messerschmidt, is reported by Mehlhorn [24]. A stronger result that the language generated by every parenthesis grammar can be recognized in logarithmic space was independently discovered by Lynch [22] and by Mehlhorn [24]. Input-driven languages then emerged in a subsequent paper by Mehlhorn [25], who defined the notion of a DIDPDA and presented an algorithm for simulating it using space $O\big(\frac{(\log n)^2}{\log \log n}\big)$. An improved algorithm using space $O(\log n)$ was presented by von Braunmühl and Verbeek [6]. Later, Rytter [36] developed a simpler alternative algorithm for simulating input-driven automata using space $O(\log n)$, which was based upon Mehlhorn's [24] method for parenthesis grammars.

Theorem 8 (von Braunmühl and Verbeek [6])**.** *Every input-driven language is recognized by a logarithmic-space deterministic Turing machine.*

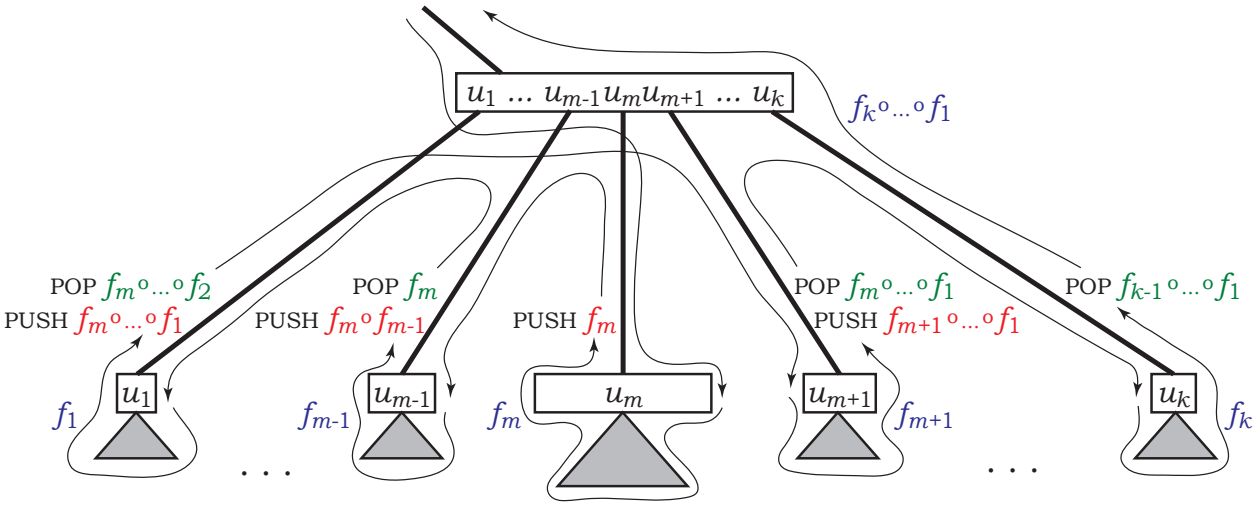

Figure 3: How the Mehlhorn–Rytter algorithm handles a concatenation $u_1 \ldots u_k$, where $k \geqslant 2$ and m is the least number with $|u_m| \geqslant |u_i|$ for all i.

The proof given here follows Rytter [36] and presents the Mehlhorn–Rytter algorithm.

Proof. Let $A = (\Sigma, Q, \Gamma, q_0, \bot, [\delta_a]_{a\in\Sigma}, F)$ be a deterministic input-driven automaton. For every well-nested string, consider the *behaviour* of A, which is a function $f\colon Q \to Q$ mapping states to states, defined in Section 4.1. Given an input string w, the algorithm calculates the behaviour of A on selected well-nested substrings of w, beginning with its shorter substrings and ending with the whole string. Once the behaviour f on the whole string is computed, the algorithm accepts if and only if $f(q_0) \in F$.

Consider the input string as a formula over the operations of concatenation and enclosure in brackets. In other words, w is represented as a tree, where every node corresponds to a well-nested substring of w as follows. The root is w. Every internal node labeled with a substring $<u>$, where u is well-nested, has one son, u. If a node is labeled with a concatenation $u_1 \ldots u_k$, where $k \geqslant 2$ and each substring u_i is well-nested, then this node has k sons labeled with u_1, …, u_k. Nodes labeled with a neutral symbol $c \in \Sigma_0$ or with the empty string are leaves of the tree.

The Mehlhorn–Rytter algorithm makes a single traversal of the tree, keeping the following data in its memory: the current node u (remembered as two pointers to the input string), the direction of motion (up or down), the behaviour of the automaton on u (when going up), and a stack containing the behaviour of A on at most $\log_2 n$ previously traversed subtrees. The computation begins at the root, in the direction down, and ends when the root is visited again, in the direction up.

Every time the algorithm reaches a node $u = <v>$ in the direction down, it directly proceeds further down to the node v. Eventually, once the subtree for v is traversed, the algorithm returns up to the node v. At this time, the algorithm knows the behaviour g on v. Then it calculates the behaviour f on $<v>$ by the formula

$$h(q) = \delta_>\big(g(q'), \gamma\big), \qquad \text{where } (q', \gamma) = \delta_<(q),$$

and goes up to the node $<v>$. The stack is not used at this node.

The key idea of the algorithm is in how it handles a concatenation node $u = u_1 \dots u_k$, which is illustrated in Figure 3. When this node is first visited on the way down, the algorithm determines *the longest substring* u_m, with $m \in \{1, \dots, k\}$, and continues down to u_m without pushing anything to the stack. If there are multiple longest substrings of the same length, then the first of them is considered the longest.

When the algorithm eventually returns from the subtree of u_m, it will have the corresponding behaviour function f_m calculated. By examining u and u_m, it determines that it has just returned from the longest substring of u, and hence has not yet visited any other substrings. Then it pushes the behaviour function f_m onto the stack and continues down to the neighbouring substring u_{m-1}.

Once the algorithm returns from u_{m-1}, having calculated the behaviour f_{m-1} on u_{m-1}, it again determines that u_m is the longest substring, while it has just returned from the directly preceding substring u_{m-1}. Therefore, it can pop the behaviour f_m from the stack and compose it with the behaviour f_{m-1}. Then the algorithm pushes the resulting behaviour $f_m \circ f_{m-1}$ on $u_{m-1}u_m$ to the stack and proceeds down to the next substring u_{m-2}.

The traversal of the subtrees is carried out in the following order: u_m, u_{m-1}, u_{m-2}, $\dots$, u_2, u_1, and then u_{m+1}, u_{m+2}, $\dots$, u_{k-1}, u_k. Thus, after processing every next substring, the algorithm can compose the behaviour on the *concatenation of all previously processed substrings* (popped from the stack) with the behaviour on the last substring (just computed). After the last son is traversed (this will be u_k, unless $m = k$, in which case the last son is u_1), the algorithm determines the behaviour $f_k \circ \dots \circ f_1$ on the whole substring $u_1 \dots u_k$, and travels from this node up to its father.

On every leaf, the algorithm uses the known behaviour of A on that leaf. For a neutral symbol $c \in \Sigma_0$, the behaviour function is $f = \delta_c$, and for the empty string, it is the identity function on Q.

The correctness of the algorithm is evident from the above explanations. To see that it uses logarithmic space, consider that the depth of the stack on an input of length n is bounded by $\log_2 n$, because the algorithm pushes a function to the stack only when visiting a *second largest son* or a son smaller than that, and hence, whenever the stack depth is increased by 1, the size of the subtree is at least halved. Every behaviour function requires constant space, and hence the stack uses $O(\log n)$ bits of memory. Besides the stack, the algorithm remembers a fixed number of pointers to the input (a few of them are needed to implement tree-walking primitives), which together use space $O(\log n)$. □

Though the Mehlhorn–Rytter algorithm settles the space complexity of input-driven languages, there are stronger complexity upper bounds in terms of shallow circuits. This line of research has a history of its own. Since every input-driven language is context-free, it is recognized by uniform circuits of depth $O(\log^2 n)$. For parenthesis languages, Gupta [16] improved the depth of circuits to $O(\log n \log\log n)$, and then Buss [7] further improved it to $O(\log n)$, thus establishing that every parenthesis language is in NC^1. Finally, Dymond [12] applied the method of Buss [7] to implement the Mehlhorn–Rytter algorithm [36] in NC^1.

6 Complexity of decision problems

This section reports on the computational complexity of testing various properties of input-driven automata. The simplest problem is whether a given IDPDA A accepts a given string w: the *(uniform) membership problem*. The complexity of this problem depends both on the size of A and the length of w, measured in bits.

One way of solving the membership problem is by using the Mehlhorn–Rytter algorithm, explained in Theorem 8. However, the Mehlhorn–Rytter algorithm is efficient only with respect to the length of the string. The dependence of both its time and its space complexity on the size of the automaton is at least linear, because already a single behaviour function of A requires $|A| \log |A|$ bits to store. Much lower complexity is achieved by another algorithm, which applies to a large class of deterministic pushdown automata (DPDA), including all DIDPDAs.

Theorem 9 (von Braunmühl, Cook, Mehlhorn, Verbeek [5])**.** *Consider DPDAs, which make at most n push operations on an input of length n. Then the uniform membership problem for such DPDAs is decidable in time polynomial in $|A|$ and $|w|$, using space $O\big((\log |w|)((\log |w| + \log |A|)\big)$.*

At any point, the von Braunmühl–Cook–Mehlhorn–Verbeek algorithm remembers a number of *computation snapshots* taken at several points of the DPDA's computation. Each snapshot represents partial information on the configuration of the DPDA at a certain time in its computation: that it was in a certain state observing a certain position in the input, and its stack had a certain top symbol and a certain height. At any time, the algorithm remembers $O(\log n)$ snapshots, while each snapshot requires $O\big(\log |w| + \log |A|\big)$ bits of memory, which gives the space complexity. With so few snapshots remembered at once, the algorithm has to recompute some of them again and again, but nevertheless stays within polynomial time.

In order to apply Theorem 9 to a given DPDA, one has to transform it to ensure that there are at most n push operations. However, for DIDPDAs, this condition holds by definition, which implies the following complexity upper bound.

Corollary 1. *The membership problem for DIDPDA is in $SC^2 = \mathrm{DTIMESPACE}(n^{O(1)}, \log^2 n)$.*

The next decision problem is the *emptiness problem*, that is, whether a given input-driven automaton recognizes the empty language. For context-free grammars, this problem is P-complete. As shown in the next theorem, it is actually P-complete already for DIDPDAs.

Theorem 10 (Lange [19])**.** *The emptiness problem for both deterministic and nondeterministic input-driven automata is P-complete.*

Proof. It is well known that emptiness of general nondeterministic pushdown automata can be decided in polynomial time.

The P-hardness of the emptiness problem for DIDPDAs is proved by reduction from the Monotone Circuit Value Problem (MCVP). Assume Boolean circuits consisting of the gates $C_1, \ldots, C_n$, where each gate C_i is of one of the following forms: a conjunction gate $C_i = C_j \wedge C_k$, with $j, k < i$, a disjunction gate $C_i = C_j \vee C_k$, where $j, k < i$, or a constant gate $C_i = 0$ or $C_i = 1$. The gate C_n is the output gate. Then MCVP is stated as "Given a Boolean circuit, determine whether the value computed at the output gate is 1".

The reduction function transforms a given circuit of this form to a DIDPDA over a fixed alphabet $\Sigma_{+1} = \{<\}$, $\Sigma_{-1} = \{>\}$, $\Sigma_0 = \{c\}$. If the circuit evaluates to 1, then the constructed DIDPDA accepts a unique string, and if the circuit evaluates to 0, then the DIDPDA rejects all inputs.

The set of states of the DIDPDA is $Q = \{q_1, \ldots, q_n, r_0, r_1\}$ and its stack alphabet is $\Gamma = \{\, \gamma_{i,\mathrm{op}} \mid i \in \{1, \ldots, n\}, \mathrm{op} \in \{\wedge, \vee\} \,\}$. The transitions are constructed, so that, from each state q_i, the automaton may read a unique non-empty well-nested string w_i. The bracket structure of this string corresponds to the Boolean formula expressing the value of the gate C_i, though the

string contains no values of any gates. After reading w_i, the automaton enters the state r_0 or r_1, depending on whether C_i evaluates to 0 or to 1.

If C_i is a constant ($C_i = 0$ or $C_i = 1$), then the corresponding string $w_i = c$ contains no brackets. The transition function by c is defined to move from q_i to r_0 (if $C_i = 0$) or to r_1 (if $C_i = 1$). No other transitions are allowed in q_i.

Let $C_i = C_j \wedge C_k$. Then the transition from q_i by a left bracket $<$ is defined to enter the state q_j and push the symbol $\gamma_{k,\wedge}$ onto the stack. Inside the brackets, the automaton may read only the string w_j corresponding to C_j, and finishes reading it in a state r_0 or r_1, depending on the value in C_j. Next, the automaton expects a right bracket $>$. If the value of C_j is 0, then the value of C_i is also 0, and there are no more computations to be done; thus, the transition by the right bracket $>$ from r_0 by the stack symbol $\gamma_{k,\wedge}$ leads to the state r_0, and $w_i = <w_j>$. On the other hand, if the value computed in C_j is 1, then the value of C_i depends on the value of C_k; accordingly, the transition in the state r_1 by the right bracket $>$ and by the stack symbol $\gamma_{k,\wedge}$ leads to the state q_k. In this case, the string corresponding to C_i is $w_i = <w_j>w_k$.

A disjunction gate $C_i = C_j \vee C_k$ is handled similarly, using a stack symbol $\gamma_{k,\vee}$.

Finally, let q_n be the initial state of the automaton, and let r_1 be its only accepting state. Then the automaton recognizes the singleton language $\{w_n\}$ if the circuit evaluates to 1, and accepts no strings otherwise, which proves the correctness of the reduction. □

Consider the more general *equivalence problem* (whether two given automata recognize the same language) and *inclusion problem* (whether the language recognized by one given automaton is a subset of the language recognized by the other). For context-free grammars, both problems are undecidable, whereas for DPDAs, equivalence is decidable and inclusion is not. For deterministic IDPDAs, both problems have polynomial-time algorithms.

Theorem 11. *Both the equivalence and the inclusion problems for DIDPDA are P-complete.*

Proof. It is sufficient to present an algorithm for the inclusion problem. Given two DIDPDAs A and B over a common alphabet, construct a DIDPDA C that recognizes the language $L(A) \setminus L(B)$. This can be done in polynomial time using the direct product construction, as in Lemma 3. Since $L(A) \subseteq L(B)$ if and only if $L(C) = \emptyset$, it is left to test C for emptiness.

The P-hardness follows from Theorem 10, by fixing the second automaton to recognize the empty language. □

For nondeterministic automata, these problems become harder. Already the problem of testing whether a given NIDPDA accepts all strings (the *universality problem*) is complete for exponential time.

Theorem 12 (Alur and Madhusudan [2])**.** *The universality, equivalence and inclusion problems for NIDPDA are EXPTIME-complete.*

Sketch of a proof. Exponential-time algorithms deciding these properties proceed by determinizing a given NIDPDA and then applying the algorithms from Theorem 11.

The EXPTIME-hardness of the universality problem is proved by directly simulating a polynomial-space alternating Turing machine. For every such machine M and for every input string u, the goal is to construct an NIDPDA that accepts all its inputs if and only if $u \notin L(M)$. First, define an alphabet $\Sigma = \Sigma_{+1} \cup \Sigma_{-1} \cup \Sigma_0$, where Σ_{+1} contains sufficiently many brackets to represent every configuration of M as a string (that is, there are different left brackets representing

	Membership		Properties of a language		
	fixed	uniform	emptiness	equality	inclusion
DFA	regular	L	NL	NL	NL
NFA	regular	NL	NL	PSPACE	PSPACE
DIDPDA	in NC^1 [12]	in SC^2	P [19]	P	P
NIDPDA	in NC^1 [12]	in P	P [19]	EXPTIME [2]	EXPTIME [2]
DPDA	in $NC^2 \cap SC^2$ [5]	P	P	decidable	co-r.e.
CF	in NC^2	P	P	co-r.e.	co-r.e.

Table 2: Complexity of languages and decision problems for finite automata (DFA, NFA), input-driven automata (DIDPDA, NIDPDA), deterministic pushdown automata (DPDA) and context-free grammars (CF). For each class with complete problems (L, NL, P, PSPACE, EXPTIME), the given problem is complete for that class.

tape symbols and states of M, etc.), and Σ_{-1} consists of equally many corresponding right brackets. Neutral symbols are not needed, that is, $\Sigma_0 = \emptyset$.

Assume that M allows at most binary universal branching. Then, each computation of M on u is a finite binary tree, and every node in this tree is identified by a binary string $x \in \{0,1\}^*$, with $x = \epsilon$ identifying the root. Every such tree shall be represented as a string over Σ as follows. Let $\alpha_x \in \Sigma_{+1}^*$ denote the tape contents of M at the node identified by x. Denote the reversal of α_x, in which left brackets are renamed to right brackets, by $\alpha_x^R \in \Sigma_{-1}^*$. For an accepting configuration α_x, the corresponding string is $w_x = \alpha_x \alpha_x^R$. If α_x is a universal configuration, then it has two sons, α_{x0} and α_{x1}. Let w_{x0} and w_{x1} be the string representations of the subtrees corresponding to these configurations. Then the subtree of α_x is defined as $w_x = \alpha_x w_{x0} \alpha_x^R w_{x1}$. The string $w_\epsilon \in \Sigma^*$ represents this entire computation tree.

The desired NIDPDA A is defined to accept all strings over Σ *except* valid string representations of accepting computation trees of M on u. This is done by scanning the input string for any deviations from the above definition. The possible deviations are as follows: (i) a configuration α_x does not match the next configuration w_{x0}; (ii) a configuration α_x does not match its reversal α_x^R; (iii) a configuration α_x^R (written in reverse) does not match the next configuration w_{x1}. The NIDPDA uses nondeterminism to guess the position of the error, and then verifies that the error indeed occurs. In the cases (i),(iii), it is sufficient to remember the position of the error on the tape, and then use polynomially many states to check it. In the case (ii), the automaton pushes each symbol of α_x onto the stack and then pops them when reading the matching symbols in α_x^R; any mismatches are then detected directly.

Now, if $u \notin L(M)$, then there are no accepting computation trees and no valid string representations thereof, and therefore $L(A) = \Sigma^*$. If $u \in L(M)$, then at least one accepting tree exists, and its string representation is rejected by A. □

The complexity of basic decision problems for IDPDAs is compared to the similar results for related models in Table 2. See a recent survey on formal grammars [29] for missing references and further results. The complexity of some properties of IDPDAs related to bisimulation was determined by Srba [39].

7 Extended models and further work

All basic questions on descriptional and computational complexity of input-driven pushdown automata have been answered in the literature. However, the ongoing study of input-driven automata has identified a number of interesting related models, which have unresolved complexity questions. For instance, one can study the complexity of the *height-deterministic pushdown automata* [21, 26], that is, NPDAs, where the stack height after processing an input string w is the same in all computations on w. One can consider descriptional complexity of input-driven automata operating on infinite strings, as defined by Alur and Madhusudan [2, 3]. Other possible extensions of the model include two-way IDPDA and alternating IDPDA. For input-driven grammars, one can consider their variants augmented with Boolean operations, following the study of conjunctive and Boolean grammars [29]; input-driven Boolean grammars should generate the same family of languages as ordinary input-driven grammars, but their complexity is most likely different.

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. Berstel, L. Boasson, "Balanced grammars and their languages", *Formal and Natural Computing 2002*, LNCS 2300, 3–25.

[5] B. von Braunmühl, S. Cook, K. Mehlhorn, R. Verbeek, "The recognition of deterministic CFLs in small time and space", *Information and Control*, 56:1–2 (1983), 34–51.

[6] B. von Braunmühl, R. Verbeek, "Input driven languages are recognized in $\log n$ space", *Annals of Discrete Mathematics*, 24 (1985), 1–20.

[7] S. R. Buss, "The Boolean formula value problem is in ALOGTIME", *Proceedings of the 19th Annual ACM Symposium on Theory of Computing* (STOC 1987, New York, USA), 123–131.

[8] 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.

[9] D. Chistikov, R. Majumdar, "A uniformization theorem for nested word to word transductions", *Implementation and Application of Automata* (CIAA 2013, Halifax, Canada, July 16–19, 2013), LNCS 7982, Springer, 97–108.

[10] 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.

[11] D. Debarbieux, O. Gauwin, J. Niehren, T. Sebastian, M. Zergaoui, "Early nested word automata for XPath query answering on XML streams", *Implementation and Application of Automata* (CIAA 2013, Halifax, Canada, July 16–19, 2013), LNCS 7982, Springer, 292–305.

[12] P. W. Dymond, "Input-driven languages are in $\log n$ depth", *Information Processing Letters*, 26 (1988), 247–250.

[13] Y. Gao, N. Moreira, R. Reis, S. Yu, "A review on state complexity of individual operations", Faculdade de Ciencias, Universidade do Porto, Technical Report DCC-2011-8 `www.dcc.fc.up.pt/dcc/Pubs/TReports/TR11/dcc-2011-08.pdf`

[14] O. Gauwin, J. Niehren, Y. Roos, "Streaming tree automata", *Information Processing Letters*, 109 (2008) 13–17.

[15] S. Ginsburg, M.A. Harrison, "Bracketed context-free languages" *Journal of Computer and System Sciences*, 1 (1967), 1–23.

[16] A. Gupta, *A Fast Parallel Algorithm for Recognition of Parenthesis Languages*, M.Sc. thesis, University of Toronto, 1985.

[17] Y.-S. Han, K. Salomaa, "Nondeterministic state complexity of nested word automata", *Theoretical Computer Science*, 410 (2009), 2961–2971.

[18] J. Hromkovič, S. Seibert, J. Karhumäki, H. Klauck, G. Schnitger, "Communication complexity method for measuring nondeterminism in finite automata", *Information and Computation,* 172 (2002) 202–217.

[19] M. Lange, "P-hardness of the emptiness problem for visibly pushdown languages", *Information Processing Letters*, 111:7 (2011), 338–341.

[20] H. Leung, "Descriptional complexity of NFA of different ambiguity", *International Journal of Foundations of Computer Science*, 16 (2005) 975–984.

[21] N. Limaye, M. Mahajan, A. Meyer, "On the complexity of membership and counting in height-deterministic pushdown automata", *3rd International Computer Science Symposium in Russia* (CSR 2008), LNCS 5010, 240–251.

[22] N. A. Lynch, "Log space recognition and translation of parenthesis languages", *Journal of the ACM*, 24:4 (1977), 583–590.

[23] R. McNaughton, "Parenthesis grammars", *Journal of the ACM*, 14:3 (1967), 490–500.

[24] K. Mehlhorn, "Bracket-languages are recognizable in logarithmic space", *Information Processing Letters* 5:6 (1976), 168–170.

[25] K. Mehlhorn, "Pebbling mountain ranges and its application to DCFL-recognition", *Automata, Languages and Programming* (ICALP 1980, Noordweijkerhout, The Netherlands, 14–18 July 1980), LNCS 85, 422–435.

[26] D. Nowotka, J. Srba, "Height-deterministic pushdown automata", *Mathematical Foundations of Computer Science* (MFCS 2007), LNCS 4708, 125–134.

[27] 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.

[28] A. Okhotin, "Unambiguous finite automata over a unary alphabet", *Information and Computation*, 212 (2012), 15–36.

[29] A. Okhotin, "Conjunctive and Boolean grammars: the true general case of the context-free grammars", *Computer Science Review*, 9 (2013), 27–59.

[30] A. Okhotin, X. Piao, K. Salomaa, "Descriptional complexity of input-driven pushdown automata", In: H. Bordihn, M. Kutrib, B. Truthe (Eds.), *Languages Alive: Essays Dedicated to Jürgen Dassow on the Occasion of His 65th Birthday*, LNCS 7300, 2012, 186–206.

[31] A. Okhotin, K. Salomaa, "Descriptional complexity of unambiguous nested word automata", Language and Automata Theory and Applications (LATA 2011, Tarragona, Spain, 26–31 May 2011), LNCS 6638, 414–426. (Full version "Descriptional complexity of input-driven pushdown automata", 15 pp., under review in Theoret. Comput. Sci.)

[32] A. Okhotin, K. Salomaa, "State complexity of operations on input-driven pushdown automata", *Mathematical Foundations of Computer Science* (MFCS 2011, Warsaw, Poland, 22–26 August 2011), LNCS 6907, 485–496. (Full version in preparation.)

[33] A. Okhotin, K. Salomaa, "Input-driven pushdown automata with limited nondeterminism", *Developments in Language Theory* (DLT 2014, Ekaterinburg, Russia, 26–29 August 2014), to appear.

[34] A. Palioudakis, K. Salomaa, S.G. Akl, "State complexity of finite tree width NFAs", *Journal of Automata, Languages, and Combinatorics*, 17 (2012) 245–264.

[35] X. Piao, K. Salomaa, "Operational state complexity of nested word automata", *Theoretical Computer Science*, 410 (2009), 3290–3302.

[36] W. Rytter, "An application of Mehlhorn's algorithm for bracket languages to $\log n$ space recognition of input-driven languages", *Information Processing Letters*, 23 (1986) 81–84.

[37] K. Salomaa, "Limitations of lower bound methods for deterministic nested word automata", *Information and Computation*, 209 (2011), 580–589.

[38] E.M. Schmidt, *Succinctness of Description of Context-Free, Regular and Unambiguous Languages*. Ph. D. thesis, Cornell University, 1978.

[39] J. Srba, "Beyond language equivalence on visibly pushdown automata", *Logical Methods in Computer Science*, 5 (2009) 1-22.