Edit Distance Neighbourhoods of Input-Driven Pushdown Automata

Alexander Okhotin[1(✉)] and Kai Salomaa[2]

[1] St. Petersburg State University, 14th Line V.O., 29B, Saint Petersburg 199178, Russia
alexander.okhotin@spbu.ru

[2] School of Computing, Queen's University, Kingston, ON K7L 2N8, Canada
ksalomaa@cs.queensu.ca

Abstract. Edit distance ℓ-neighbourhood of a formal language is the set of all strings that can be transformed to one of the strings in this language by at most ℓ insertions and deletions. Both the regular and the context-free languages are known to be closed under this operation, whereas the deterministic pushdown automata are not. This paper establishes the closure of the family of input-driven pushdown automata (IDPDA), also known as visibly pushdown automata, under the edit distance neighbourhood operation. A construction of automata representing the result of the operation is given, and close lower bounds on the size of any such automata are presented.

1 Introduction

Edit distance is the standard measure of similarity between two strings: this is the least number of elementary edit operations—such as inserting a symbol, removing a symbol or replacing a symbol with another symbol—necessary to transform one string into another. Algorithms and methods related to the edit distance are useful in numerous applications: whenever DNA sequences are checked for similarity, misspelled words are matched to their most probable spelling, etc.

Many problems involving edit distance are formulated in terms of formal languages. In particular, one can consider the edit distance between a string and a language, which is relevant to assessing the number of syntax errors in an input string, as well as to correcting those errors [13]. There is also a notion of a distance between a pair of languages, studied, in particular, by Chatterjee et al. [6]. The shortest distance between two languages is uncomputable if both languages are given by grammars [11], whereas the distance between a grammar and a regular language is computable [7].

In connection with the distance between a string and a language, there is a convenient notion of *edit distance ℓ-neighbourhood* of a given language: this is a set of all strings at edit distance at most ℓ from some element of that language. The edit distance ℓ-neighbourhood is then an operation on languages.

© Springer International Publishing AG 2017
P. Weil (Ed.): CSR 2017, LNCS 10304, pp. 260–272, 2017.
DOI: 10.1007/978-3-319-58747-9_23

It is known that the regular languages are closed under this operation. In particular, Povarov [20] determined an optimal construction for the 1-neighbourhood of a given automaton; this result was extended to the ℓ-neighbourhood in the papers by Salomaa and Schofield [21] and by Ng, Rappaport and Salomaa [12].

The edit distance operation is no less relevant in formal grammars. For context-free grammars, the work by Aho and Peterson [1] on error recovery in parsers contains a direct construction of a grammar, which is sufficient to prove the closure under edit distance neighbourhood. Also, ℓ-neighbourhood is computable by a nondeterministic finite transducer (NFT), and by the closure of grammars under all such transductions, the closure follows; the same argument applies to all families closed under NFT, such as the linear grammars. On the other hand, for deterministic pushdown automata (DPDA)—or, equivalently, for LR(k) grammars—there is a simple example witnessing their non-closure under the 1-neighbourhood operation: the language $L = \{ ca^n b^n \mid n \geqslant 0 \} \cup \{ da^n b^{2n} \mid n \geqslant 0 \}$ is recognized by a DPDA, whereas its 1-neighbourhood, under intersection with a^*b^*, is the language $\{ a^n b^n \mid n \geqslant 0 \} \cup \{ a^n b^{2n} \mid n \geqslant 0 \}$, which is a classical example of a language not recognized by any DPDA.

This paper investigates ℓ-neighbourhoods for an important subclass of DPDA: the *input-driven pushdown automata* (IDPDA), also known under the name of *visibly pushdown automata*. In these automata, the input symbol determines whether the automaton should push a stack symbol, pop a stack symbol or leave the stack untouched. These symbols are called *left brackets*, *right brackets* and *neutral symbols*, and the symbol pushed at each left bracket is always popped when reading the corresponding right bracket. Input-driven automata are important as a model of hierarchically structured data, such as XML documents or computation traces for recursive procedure calls. They are also notable for their appealing theoretical properties, resembling those of finite automata.

Input-driven automata were first studied by Mehlhorn [10] and by von Braunmühl and Verbeek [4], who determined that the languages they recognize lie in logarithmic space. Von Braunmühl and Verbeek [4] also proved that deterministic and nondeterministic variants of the model are equal in power. Later, Alur and Madhusudan [2,3] reintroduced the model under the names "visibly pushdown automata" and "nested word automata", and carried out its language-theoretic study, in particular, establishing the closure of the corresponding family under the basic operations on languages. Their contribution inspired further work on the closure properties of input-driven automata and on their descriptional complexity [8,16–18].

The main result of this paper, presented in Sect. 3, is that the family of languages recognized by input-driven automata is closed under the edit distance neighbourhood operation. The main difficulty in the construction is that when the symbol inserted or deleted is a bracket, then adding or removing that symbol changes the bracket structure of the string, so that other brackets may now be matched not to the same brackets as before. It is shown how, given an NIDPDA for the original language, to construct an NIDPDA with one edit operation applied.

The question of whether these constructions are optimal in terms of the number of states is addressed in Sect. 4, where some lower bounds on the worst-case size of an NIDPDA representing the edit distance neighbourhood of an n-state NIDPDA are established. These bounds confirm that the constructions presented in this paper are fairly close to optimal.

In Sect. 5, a similar construction is presented for deterministic input-driven automata. The construction uses exponentially many states, and is accompanied with a fairly close lower bound, showing that a DIDPDA for the edit distance neighbourhood requires $2^{\Omega(n^2)}$ states in the worst case.

2 Input-Driven Automata

An *input-driven pushdown automaton* (IDPDA) [2,3,10] is a special case of a deterministic pushdown automaton, in which the input alphabet Σ is split into three disjoint sets of *left brackets* Σ_{+1}, *right brackets* Σ_{-1} and *neutral symbols* Σ_0. If the input symbol is a left bracket from Σ_{+1}, then the automaton always pushes one symbol onto the stack. For a right bracket from Σ_{-1}, the automaton must pop one symbol. Finally, for a neutral symbol in Σ_0, the automaton may not use the stack. In this paper, symbols from Σ_{+1} and Σ_{-1} shall be denoted by left and right angled brackets, respectively ($<$, $>$), whereas lower-case Latin letters from the beginning of the alphabet $(a, b, c, \ldots)$ shall be used for symbols from Σ_0. Input-driven automata may be deterministic (DIDPDA) and nondeterministic (NIDPDA).

Under the simpler definition, input-driven automata operate on input strings, in which the brackets are *well-nested*. When an input-driven automaton reads a left bracket $< \in \Sigma_{+1}$, it pushes a symbol onto the stack. This symbol is popped at the exact moment when the automaton encounters the matching right bracket $> \in \Sigma_{-1}$. Thus, a computation of an input-driven automaton on any well-nested substring leaves the stack contents untouched, as illustrated in Fig. 1.

The more general definition of input-driven automata assumed in this paper also allows ill-nested input strings. For every unmatched left bracket, the symbol pushed to the stack when reading this bracket is never popped, and remains in the stack to the end of the computation. An unmatched right bracket is read with an empty stack: instead of popping a stack symbol, the automaton merely

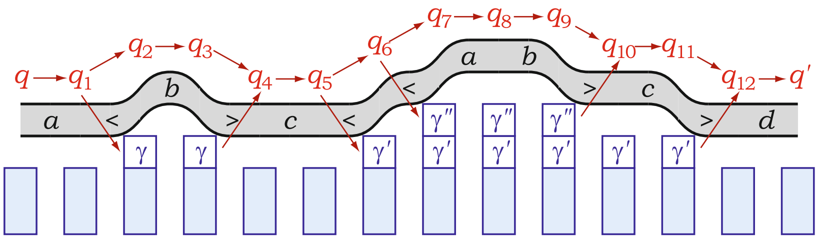

Fig. 1. The computation of an IDPDA on a well-nested string.

detects that the stack is empty and makes a special transition, which leaves the stack empty.

Definition 1 (von Braunmühl and Verbeek [4]; Alur and Madhusudan [2]). *A nondeterministic input-driven pushdown automaton (NIDPDA) over an alphabet* $\widetilde{\Sigma} = (\Sigma_{+1}, \Sigma_{-1}, \Sigma_0)$ *consists of*

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

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

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

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

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

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

Another basic construction for DIDPDA that will be used in this paper is computing the *behaviour function* of a given DIDPDA by another DIDPDA. When a DIDPDA with a set of states Q processes a well-nested string w and begins in a state q, it finishes reading that string in some state $f(q)$, where $f \colon Q \to Q$ is its *behaviour function on* w, and the stack is left untouched. Thus, f completely characterizes the behaviour of a DIDPDA on w. For any given

DIDPDA A, it is possible to construct an n^n-state DIDPDA, where $n = |Q|$, that reaches the end of an input w in a state representing the behaviour of A on the longest well-nested suffix of w. This construction is necessary for optimal constructions representing operations on DIDPDA [17].

For more details on input-driven automata and their complexity, the readers are directed to a recent survey [15].

3 Edit Distance for Input-Driven Automata

Let Σ be an alphabet, let $a \in \Sigma$ be a symbol. Then, for a string $w \in \Sigma^*$, the set of strings obtained by inserting a at any position is denoted by $\text{insert}_a(w) = \{\, uav \mid w = uv \,\}$. Similarly, the set of strings obtained by erasing a is $\text{delete}_a(w) = \{\, uv \mid w = uav \,\}$. These operations are extended to any language $L \subseteq \Sigma^*$ elementwise, with $\text{insert}_a(L) = \bigcup_{w \in L} \text{insert}_a(w)$ and $\text{delete}_a(L) = \bigcup_{w \in L} \text{delete}_a(w)$.

The set of strings at edit distance at most ℓ from a given string w is called its *ℓ-neighbourhood*, denoted by $E_\ell(w)$ and defined as follows.

$$\begin{aligned} E_0(w) &= \{w\} \\ E_{\ell+1}(w) &= E_\ell(w) \cup \bigcup_{w' \in E_\ell(w)} \bigcup_{a \in \Sigma} \big(\text{insert}_a(w') \cup \text{delete}_a(w')\big) \end{aligned}$$

The ℓ-neighbourhood of a language $L \subseteq \Sigma^*$ is the set of strings at edit distance at most ℓ from any string in L.

$$E_\ell(L) = \bigcup_{w \in L} E_\ell(w)$$

The definition of edit distance often includes the operation of replacing one symbol with another. According to the above definition, replacement can be implemented as a combination of one deletion and one insertion. This difference affects the resulting edit distance. In this paper, the simpler definition is assumed, because it makes the constructions easier; however, the constructions in this paper can be extended to implement replacement as well.

In this paper, the above definitions are applied to languages over an alphabet $\Sigma = \Sigma_{+1} \cup \Sigma_{-1} \cup \Sigma_0$ recognized by an IDPDA, with the intention of constructing another IDPDA that recognizes the edit distance neighbourhood of the given language. A construction shall be obtained by first implementing the elementary operations of inserting or deleting a single symbol. According to the definition of the neighbourhood, all three types of symbols in Σ can be either inserted or deleted. However, since IDPDA handle different types of symbols differently, these six cases generally require separate treatment.

Neutral symbols are the easiest to insert or delete, the construction is the same as for finite automata.

Lemma 1. *Let L be a language recognized by an NIDPDA, let Q be its set of states, and Γ its stack alphabet. Let $c \in \Sigma_0$ be a neutral symbol. Then, both* *languages* $\mathrm{insert}_c(L)$ *and* $\mathrm{delete}_c(L)$ *are recognized by NIDPDA with the set of states* $Q \cup \widetilde{Q}$*, where* $\widetilde{Q} = \{\, \widetilde{q} \mid q \in Q \,\}$*, and with the stack alphabet* Γ*.*

There is also an NIDPDA with the same set of states Q *and the same set of stack symbols* Γ *that recognizes the language* $\bigcup_{c \in \Sigma_0} \big(\mathrm{insert}_c(L) \cup \mathrm{delete}_c(L)\big)$*.*

The second case is that of inserting a left bracket. The main difficulty is, that once a new left bracket is inserted into a given string, it may match some existing right bracket, which was formerly matched to a different left bracket. This disrupts the operation of the simulated NIDPDA, and requires some efforts to re-create it.

Lemma 2. *Let* L *be a language recognized by an NIDPDA over an alphabet with the set of states* Q *and with the stack alphabet* Γ*. Let* $\ll \in \Sigma_{+1}$ *be a left bracket. Then, the language* $\mathrm{insert}_{\ll}(L)$ *is recognized by an NIDPDA with the set of states* $Q \cup \widetilde{Q} \cup (Q \times \Gamma)$*, where* $\widetilde{Q} = \{\, \widetilde{q} \mid q \in Q \,\}$*, and with the stack alphabet* $\Gamma \cup \{\Box\} \cup (\Gamma \times \Gamma)$*.*

There is also an NIDPDA with the same states Q *and the same stack symbols* Γ *that recognizes* $\bigcup_{\ll \in \Sigma_{+1}} \mathrm{insert}_{\ll}(L)$*.*

Proof. The first two types of states in the new automaton are the states q and $\widetilde{q}$, for any $q \in Q$. In either state, the new automaton simulates the original automaton being in the state q.

In the beginning, the new automaton uses the states from $\widetilde{Q}$ to simulate the operation of the original automaton before it encounters the new left bracket that has been inserted. At some point, the new automaton guesses that the currently observed left bracket is the new one, and executes a special transition: when passing the inserted left bracket ($\ll$) in a state $\widetilde{q}$, the new automaton pushes a special box symbol ($\Box$) into the stack and enters the state q: in these states, the new automaton knows that the inserted symbol has already been encountered, and simulates the original automaton as it is.

Later, when the automaton pops the box ($\Box$) upon reading some right bracket ($>$), it knows that the stack symbol in the original computation corresponding to this bracket lies in its stack one level deeper. Being an input-driven automaton, it cannot pop it yet, but it can guess what that symbol is going to be. If γ is the guessed stack symbol, then the automaton simulates the transition upon popping γ and enters a state of the form (q, γ), where q is the result of the transition, and γ is remembered in the state for later verification.

In states of the form (q, γ), neutral symbols are being read without modifying the remembered stack symbol. Whenever a left bracket ($<$) occurs, and the original automaton would enter a state r and push a stack symbol σ, the new automaton enters the state r and pushes a special stack symbol (σ, γ), which maintains the remembered stack symbol in the second component, and restores it upon reading the well-nested substring.

When, in a state of the form (q, γ), the new automaton reaches a right bracket ($>$), first, it verifies that the symbol being popped is indeed γ. The stack symbol needed to carry out the present transition is again located one level deeper in the stack, and therefore the automaton has to guess another stack symbol γ', and store it in the second component of the pair, etc. This completes the construction.

Since the automaton does not need to know the particular bracket symbol $\ll \in \Sigma_{+1}$ that has been inserted before and after encountering it, the same construction yields an NIDPDA for the language $\bigcup_{\ll \in \Sigma_{+1}} \text{insert}_{\ll}(L)$. □

The case of erasing a left bracket is carried out slightly differently.

Lemma 3. *Let L be a language recognized by an NIDPDA with the set of states Q and with the stack alphabet Γ. Let $\ll \in \Sigma_{+1}$ be a left bracket. Then, the language* $\text{delete}_{\ll}(L)$ *is recognized by an NIDPDA with the set of states $Q \cup \widetilde{Q} \cup (Q \times \Gamma)$ and with the stack alphabet $\Gamma \cup (\Gamma \times \Gamma)$.*

Also, the language $\bigcup_{\ll \in \Sigma_{+1}} \text{delete}_{\ll}(L)$*, is recognized by an NIDPDA with the same states and stack symbols.*

Proof. The plan is that the new automaton is in a state $\widetilde{q}$ before passing the place where a left bracket ($\ll$) was erased. State (q, γ) means the situation after passing the erased left bracket ($\ll$), while remembering the stack symbol that the original automaton would push when reading that erased bracket ($\ll$). This state means that γ is an extra stack symbol simulated on the top of the actual stack. In a state q, the new automaton operates normally, as the erased symbol is no longer expected.

Transitions in the state $\widetilde{q}$ are the same as those in q, except that, upon reading any symbol, the new automaton may decide that directly after that symbol there was a left bracket ($\ll$) that got erased. Then, the new automaton simulates a transition by these two symbols at once, and, assuming that the original automaton's transition upon the left bracket ($\ll$) is to a state r along with pushing a symbol γ, the new automaton enters the state (r, γ).

In a state of the form $(q, \widehat{\gamma})$, upon reading any left bracket ($<$), the automaton pushes a pair of stack symbols $(\gamma, \widehat{\gamma})$, where γ is the symbol that the original automaton would push, and enters the same state that the original automaton would enter. Later, upon reading the matching right bracket ($>$) and popping the pair $(\gamma, \widehat{\gamma})$, the automaton enters the state $(r, \widehat{\gamma})$, assuming that the original automaton would enter the state r.

When the new automaton encounters a right bracket ($>$) in a state of the form $(q, \widehat{\gamma})$, popping a stack symbol γ, it simulates the original automaton's transition in the state q upon popping the stack symbol $\widehat{\gamma}$, and enters the state (r, γ), assuming that r is the state that the original automaton would enter.

In a state of the form $(q, \widehat{\gamma})$, upon reaching the bottom of the stack, the automaton simulates the transition upon popping γ and enters a normal state r, the same that the original automaton would enter. □

The constructions for insertion and deletion of right brackets are symmetric, the number of states is the same.

Now, an NIDPDA for edit distance 1-neighbourhood can be obtained by using all the six constructions within a single automation.

Theorem 1. *Let L be a language recognized by an NIDPDA with n states and k stack symbols. Then there exists an NIDPDA recognizing the language $E_1(L)$ that has $10n + 4kn + 1$ states and $k^2 + k + 1$ stack symbols.*

The ℓ-neighbourhood can be obtained by applying this construction ℓ times.

4 Lower Bounds for the Nondeterministic Case

Several constructions of automata have been presented, and the question is, whether those constructions are optimal. This is proved by presenting *witness languages*, that is, families of languages L_n recognized by an NIDPDA of size n, such that every NIDPDA for the corresponding edit distance operation on L_n requires at least $f(n)$ states. The methods for establishing such results were originally developed for finite automata, and later were generalized for NIDPDA.

The *stack height* of a string w is the height of the stack of an NIDPDA after reading w. The height of the stack depends only on w.

Definition 2. *Let $\widetilde{\Sigma} = (\Sigma_{+1}, \Sigma_{-1}, \Sigma_0)$ be an alphabet and let $L \subseteq \Sigma^*$. A set of pairs $F = \{(x_1, y_1), \ldots, (x_m, y_m)\}$ is said to be a* fooling set of depth k *for L, if each string x_i has stack height k and*

(i) $x_i y_i \in L$ for all $i \in \{1, 2, \ldots, m\}$, and
(ii) for all i, j with $1 \leqslant i < j \leqslant m$, $x_i y_j \notin L$ or $x_j y_i \notin L$.

Lemma 4 ([8,18]). *Let A be a nondeterministic input-driven pushdown automaton with a set of states Q and a set of stack symbols Γ. If $L(A)$ has a fooling set F of depth k, then $|\Gamma|^k \cdot |Q| \geqslant |F|$.*

First consider the insertion or deletion of a single symbol.
Choose $\Sigma_{+1} = \{<\}$, $\Sigma_{-1} = \{>\}$ and $\Sigma_0 = \{a, b, c, \$\}$. For $n \geqslant 1$ define

$$L_n = \{c^i {<} c^k a^i b^j \$ b^j {>} a^i \mid 1 \leqslant i, j \leqslant n,\ k \geqslant 0\}.$$

Lemma 5. *(i) There exists a constant $C \geqslant 1$, such that, for each $n \geqslant 1$, the language L_n is recognized by a DIDPDA A with $C \cdot n$ states and n stack symbols.*
(ii) Any NIDPDA recognizing the language $\text{delete}_{<}(L_n)$ *needs at least n^2 states.*

Proof. (i) The following discussion assumes that the input string is in $c^+ {<} c^+ a^+ b^+ \$ b^+ {>} a^+$. It is easy to see that by increasing the number of states of A by a multiplicative constant, the computation can be made to reject all strings not of this form.

The computation counts the length i of the prefix in c^+ preceding the left bracket $<$ and pushes this value to the stack. If $i > n$, A rejects. Then A skips the following symbols c, checks that the maximal substring in a^+ has length i, and counts the number of b's preceding the marker \$. This number is compared with the number of b's after \$. At the right bracket $>$ the stack is popped and the computation verifies that the suffix of symbols a has length i.

(ii) Choose

$$S_n = \{(c^{n+1}a^i b^j,\ \$b^j{>}a^i) \mid 1 \leqslant i, j \leqslant n\}.$$

For all $i, j \in \{1, \ldots, n\}$, the string $c^{n+1}a^i b^j \cdot \$b^j{>}a^i$ is obtained from a string from L_n by deleting a left bracket. On the other hand, for $(i, j) \neq (i', j')$, with $i, j, i', j' \in \{1, \ldots, n\}$, the string $c^{n+1}a^i b^j \cdot \$b^{j'}{>}a^{i'}$ is not in $\text{delete}_{<}(L_n)$, because $i \neq i'$ or $j \neq j'$. This means that S_n is a fooling set of depth 0 for $\text{delete}_{<}(L_n)$ and, by Lemma 4, any NIDPDA for $\text{delete}_{<}(L_n)$ needs $|S_n| = n^2$ states. □

Lemma 6. *Any NIDPDA recognizing the language* $\text{insert}_{>}(L_n)$ *needs at least* n^2 *states.*

Proof. Define

$$S'_n = \{(c^i{<}{>}ca^i b^j,\ \$b^j{>}a^i) \mid 1 \leqslant i, j \leqslant n\}.$$

Again, for all $1 \leqslant i, j \leqslant n$, $c^i{<}{>}ca^i b^j \cdot \$b^j{>}a^i$ is obtained from a string of L_n by inserting a right bracket and, on the other hand, for $(i, j) \neq (i', j')$, $c^i{<}{>}ca^i b^j \cdot \$b^{j'}{>}a^{i'} \notin \text{insert}_{>}(L_n)$. This means that S'_n is a fooling set for $\text{insert}_{>}(L_n)$, and the claim follows from Lemma 4. □

The reversal L^R of the language L recognized by an NIDPDA A can be recognized by an NIDPDA with the same number of states and stack symbols as A, when the left brackets (respectively, right brackets) in the original string are interpreted as right brackets (respectively, left brackets) in the reversed string [3]. Since inserting a left bracket into a language L is the same as inserting a right bracket into L^R, and deleting a right bracket from L is the same as deleting a left bracket from L^R, Lemmas 5 and 6 imply a tight bound on the complexity of inserting left brackets and deleting right brackets in terms of the number of states in NIDPDA.

Corollary 1. *For each* $n \geqslant 1$ *there exists a language* L'_n *recognized by a NIDPDA with* $O(n)$ *states such that any NIDPDA for the neighbourhoods* $\text{delete}_{>}(L'_n)$ *and* $\text{insert}_{<}(L'_n)$ *needs* n^2 *states.*

It remains to consider the cases of inserting and deleting a neutral symbol. Povarov [20] has shown that the Hamming neighbourhood of radius r of an n state NFA language can be recognized by an NFA with $n \cdot (r + 1)$ states and this number of states is needed in the worst case. Since an input-driven computation on strings consisting of neutral symbols is just an NFA, the lower bound for the number of states applies also for NIDPDAs. Together with Lemma 1 this implies:

Proposition 1. *For an NIDPDA* A *with* n *states and* $\sigma \in \Sigma_0$, *the neighbourhoods* $\text{delete}_\sigma(L(A))$ *and* $\text{insert}_\sigma(L(A))$ *can be recognized by an NIDPDA with* $2 \cdot n$ *states and this number of states is needed in the worst case.*

The construction of Lemma 5 can be extended to yield a lower bound for the cost of deleting multiple symbols. The result is stated in terms of neighbourhoods of a given radius.

Choose $\Sigma_{+1} = \{<\}$, $\Sigma_{-1} = \{>\}$ and $\Sigma_0 = \{a, b, c, \$\}$. For $n \geqslant 1$ define

$$H_n = \{<a^{i_1}c<a^{i_2}c \cdots <a^{i_r}cb^j\$b^j>a^{i_r}>a^{i_{r-1}} \cdots >a^{i_1} \mid r \geqslant 1,\ i_1, \dots, i_r, j \in \{1, \dots, n\}\}.$$

Lemma 7. *(i) The language H_n can be recognized by an NIDPDA with $C \cdot n$ states and n stack symbols, for some constant C.*

(ii) For $r \geqslant 1$, any NIDPDA for the neighbourhood $E_r(H_n)$ needs at least n^{r+1} states.

5 The Deterministic Case

The construction for the edit distance neighbourhood given in the previous section produces an NIDPDA out of an NIDPDA. If the goal is to obtain a deterministic automaton, then the resulting NIDPDA can of course be determinized, at the cost of a $2^{\Theta(n^2)}$ blow-up in size. This section presents some preliminary results on a direct construction for this operation, which transforms a DIDPDA to a DIDPDA for the language with one left bracket erased.

Lemma 8. *Let L be a language recognized by a DIDPDA with the set of states Q and with the stack alphabet Γ. Let $\ll \in \Sigma_{+1}$ be a left bracket. Then, the language* $\text{delete}_{\ll}(L)$ *is recognized by a DIDPDA with the set of states $Q' = Q \times 2^{Q \times (\Gamma \cup \{\perp\})} \times Q^Q$ and with the stack alphabet $\Gamma' = \Sigma_{+1} \times \Gamma \times 2^{Q \times (\Gamma \cup \{\perp\})} \times Q^Q$, where Q^Q denotes the set of all functions from Q to Q.*

Proof (sketch). At each level of brackets, the new automaton simulates the normal operation of the first automaton (Q), as well as constructs two data structures. The first data structure ($2^{Q \times (\Gamma \cup \{\perp\})}$) is a set of pairs of a state q and a stack symbol γ, each representing a situation when the computation on this level, having processed some erased bracket ($\ll$) at some position, has pushed γ upon reading that bracket, and finished reading the substring on this level in the state q. The second data structure (Q^Q) is the behaviour function for the well-nested substring at the current level. □

There is a close lower bound for this construction. Let the alphabet be $\Sigma_{+1} = \{<\}$, $\Sigma_{-1} = \{>\}$ and $\Sigma_0 = \{a, b, c, d\}$. For each $n \geqslant 1$, the language K_n is defined as follows.

$$K_n = \left\{ uc<vd^{|u|_a + |v|_b \bmod n}>a^{|u|_a \bmod n} \;\middle|\; u, v \in \{a, b, c\}^* \right\}$$

Lemma 9. *The language K_n is recognized by a DIDPDA with $C \cdot n$ states.*

Proof. First, the DIDPDA counts the number of symbols a in u modulo n. Then, upon encountering the left bracket ($<$) and verifying that it is preceded by c, it pushes the count of symbols a modulo n to the stack, and continues the counting modulo n on the string v, this time counting the symbols b. After reading v, the automaton remembers the sum $|u|_a + |v|_b$ modulo n, and can then test that the number of symbols d is correct. Finally, upon reading the right bracket ($>$), the automaton pops the number $|u|_a$ modulo n from the stack and checks this number against the suffix $a^{|u|_a \bmod n}$. $\square$

Lemma 10. *Every DIDPDA recognizing* $\mathrm{delete}_<(K_n)$ *needs at least* 2^{n^2} *states.*

Proof (Sketch of proof). A DIDPDA is faced with recognizing the following language.

$$K'_n = \big\{ wd^{i+j}{>}a^i \;\big|\; w \in \{a,b,c\}^*, \text{ and there exists a partition } w = ucv, \\ \text{with } i = |u|_a \bmod n \text{ and } j = |v|_b \bmod n \big\}$$

In the absence of left brackets, the automaton is essentially a DFA. The idea of the lower bound argument is that a DFA should remember all pairs (i, j) corresponding to different partitions of w as $w = ucv$. $\square$

This was just one of the four interesting cases of edit operations on DIDPDA. The other three cases shall be dealt with in the full version of this paper. However, this single case alone already implies a lower bound on the complexity of edit distance 1-neighbourhood of DIDPDA: indeed, any DIDPDA recognizing $E_1(K_n)$ needs at least 2^{n^2} states.

It can be concluded that the edit distance neighbourhood can be efficiently expressed in nondeterministic IDPDA, and incurs a significant blow-up in the deterministic case.

6 Future Work

It would be interesting to consider the edit distance neighbourhood operation for other automaton models related to IDPDA that are relevant to processing hierarchical data. Among such models, there are, in particular, the *transducer-driven automata* (TDPDA), introduced independently by Caucal [5] (as synchronized pushdown automata) and by Kutrib et al. [9].

In addition to the input-driven automaton models, the same question of the expressibility of edit distance neighbourhood would be interesting to investigate for other families of formal grammars besides the ordinary "context-free" grammars. The families proposed for investigation are the *multi-component grammars* [22], which are an established model in computational linguistics and have good closure properties, and the *conjunctive grammars*, which extend the ordinary grammars with a conjunction operation. In particular, it would be interesting to investigate the edit distance for the *linear conjunctive grammars* [14], which are notable for their equivalence with one-way real-time cellular automata, as well as for their non-trivial expressive power [23].

References

1. Aho, A.V., Peterson, T.G.: A minimum distance error-correcting parser for context-free languages. SIAM J. Comput. **1**(4), 305–312 (1972). http://dx.doi.org/doi/10.1137/0201022
2. Alur, R., Madhusudan, P.: Visibly pushdown languages. In: ACM Symposium on Theory of Computing, STOC 2004, Chicago, USA 13–16 June 2004, pp. 202–211 (2004). http://dx.doi.org/10.1145/1007352.1007390
3. Alur, R., Madhusudan, P.: Adding nesting structure to words. J. ACM **56**(3) (2009). http://dx.doi.org/10.1145/1516512.1516518
4. von Braunmühl, B., Verbeek, R.: Input driven languages are recognized in log n space. Ann. Discrete Math. **24**, 1–20 (1985). http://dx.doi.org/10.1016/S0304-0208(08)73072-X
5. Caucal, D.: Synchronization of pushdown automata. In: Ibarra, O.H., Dang, Z. (eds.) DLT 2006. LNCS, vol. 4036, pp. 120–132. Springer, Heidelberg (2006). doi:10.1007/11779148_12
6. Chatterjee, K., Henzinger, T.A., Ibsen-Jensen, R., Otop, J.: Edit distance for pushdown automata. In: Halldórsson, M.M., Iwama, K., Kobayashi, N., Speckmann, B. (eds.) ICALP 2015. LNCS, vol. 9135, pp. 121–133. Springer, Heidelberg (2015). doi:10.1007/978-3-662-47666-6_10
7. Han, Y.-S., Ko, K., Salomaa, K.: Approximate matching between a context-free grammar and a finite-state automaton. Inf. Comput. **247**, 278–289 (2016). http://dx.doi.org/10.1016/j.ic.2016.02.001
8. Han, Y.-S., Salomaa, K.: Nondeterministic state complexity of nested word automata. Theoret. Comput. Sci. **410**, 2961–2971 (2009)
9. Kutrib, M., Malcher, A., Wendlandt, M.: Tinput-driven pushdown automata. In: Durand-Lose, J., Nagy, B. (eds.) MCU 2015. LNCS, vol. 9288, pp. 94–112. Springer, Cham (2015). doi:10.1007/978-3-319-23111-2_7
10. Mehlhorn, K.: Pebbling mountain ranges and its application to DCFL-recognition. In: Bakker, J., Leeuwen, J. (eds.) ICALP 1980. LNCS, vol. 85, pp. 422–435. Springer, Heidelberg (1980). doi:10.1007/3-540-10003-2_89
11. Mohri, M.: Edit-distance of weighted automata: general definitions and algorithms. Int. J. Found. Comput. Sci. **14**(6), 957–982 (2003)
12. Han, Y.-S., Ko, S.-K., Salomaa, K.: Generalizations of code languages with marginal errors. In: Potapov, I. (ed.) DLT 2015. LNCS, vol. 9168, pp. 264–275. Springer, Cham (2015). doi:10.1007/978-3-319-21500-6_21
13. Ng, T., Rappaport, D., Salomaa, K.: Descriptional complexity of error detection. In: Adamatzky, A. (ed.) Emergent Computation. ECC, vol. 24, pp. 101–119. Springer, Cham (2017). doi:10.1007/978-3-319-46376-6_6
14. Okhotin, A.: Input-driven languages are linear conjunctive. Theoret. Comput. Sci. **618**, 52–71 (2016). http://dx.doi.org/10.1016/j.tcs.2016.01.007
15. Okhotin, A., Salomaa, K.: Complexity of input-driven pushdown automata. SIGACT News **45**(2), 47–67 (2014). http://doi.acm.org/10.1145/2636805.2636821
16. Okhotin, A., Salomaa, K.: Descriptional complexity of unambiguous input-driven pushdown automata. Theoret. Comput. Sci. **566**, 1–11 (2015). http://dx.doi.org/10.1016/j.tcs.2014.11.015
17. Okhotin, A., Salomaa, K.: State complexity of operations on input-driven pushdown automata. J. Comput. Syst. Sci. **86**, 207–228 (2017). http://dx.doi.org/10.1016/j.jcss.2017.02.001

18. Piao, X., Salomaa, K.: Operational state complexity of nested word automata. Theoret. Comput. Sci. **410**, 3290–3302 (2009). http://dx.doi.org/10.1016/j.tcs.2009.05.002
19. Pighizzini, G.: How hard is computing the edit distance? Inf. Comput. **165**, 1–13 (2001)
20. Povarov, G.: Descriptive complexity of the Hamming neighborhood of a regular language. In: LATA 2007, pp. 509–520 (2007)
21. Salomaa, K., Schofield, P.N.: State complexity of additive weighted finite automata. Int. J. Found. Comput. Sci. **18**(6), 1407–1416 (2007)
22. Seki, H., Matsumura, T., Fujii, M., Kasami, T.: On multiple context-free grammars. Theoret. Comput. Sci. **88**(2), 191–229 (1991). http://dx.doi.org/10.1016/0304-3975(91)90374-B
23. Terrier, V.: Recognition of linear-slender context-free languages by real time one-way cellular automata. In: Kari, J. (ed.) AUTOMATA 2015. LNCS, vol. 9099, pp. 251–262. Springer, Heidelberg (2015). doi:10.1007/978-3-662-47221-7_19