A Linear-time Simulation of Deterministic d-Limited Automata

Alexander A. Rubtsov $^{[0000-0001-8850-9749]}$ *

National Research University Higher School of Economics rubtsov99@gmail.com

A d-limited automaton is a Turing machine that uses only the cells with the input word (and end-markers) and rewrites symbols only in the first d visits. This model was introduced by T. Hibbard in 1967 and he showed that d-limited automata recognize context-free languages for each $d \ge 2$. He also proved that languages recognizable by deterministic d-limited automata form a hierarchy and it was shown later by Pighizzini and Pisoni that it begins with deterministic context-free languages (DCFLs) (for d=2).

As well-known, DCFLs are widely used in practice, especially in compilers since they are linear-time recognizable and have the corresponding CF-grammars subclass (LR(1)-grammars). In this paper we present a linear time recognition algorithm for deterministic d-limited automata (in the RAM model) which opens an opportunity for their possible practical applications. We also generalize this algorithm to deterministic d-limited automata; the extension of deterministic d-limited automata, where d is not a constant, but a function depending on the input length n.

1 Introduction

Context-free languages (CFLs) play an important role in computer science. The most well-known practical application of CFLs is the application of their deterministic subclass (DCFL) to parsing algorithms in compilers, and the core of this application is connection between LR(1)-grammars that describe syntaxes of programming languages and deterministic pushdown automata (DPDA) that implement linear-time parsing of LR(1)-grammars. In 1965 D. Knuth showed [13] that LR(1)-grammars generate exactly DCFL, the class that is recognizable by DPDA. So, DCFL is a practically important subclass of CFL that is linear-time recognizable and LR(1)-grammars are linear-time parsable, i.e. there is a linear time algorithm that constructs a derivation tree of an input word. Note that these linear time results are for deterministic Turing machines while the complexity of algorithms from the results described below and the algorithm from this paper are measured in the RAM model.

The best known upper bound for CFL parsing is n^{ω} where $\omega \leq 2.373$ is the exponent of fast-matrix multiplication, and n is the length of the input word,

^{*} Supported by Russian Science Foundation grant 20–11–20203

was obtained by L. Valiant in 1975 [22]. It was shown in [14] and [1] why this bound is hard to improve. Some recent CFL studies were focused on subclasses that are hard to parse or at least to recognize [12], and on the subclasses that are linear-time recognizable [2]. In this paper, we show that a well-known subclass of CFLs is linear time recognizable. We move to the description of the subclass.

1.1 d-Limited Automata and d-DCFLs

Namely, we focus on d-DCFLs, as they were called by T. Hibbard who introduced this subclass of CFLs. To define this subclass, we define an auxiliary computational model. We provide here an informal definition, a formal definition could be found in the next section.

A d-limited automaton (d-LA) is a Turing machine (TM) that visits only the cells with the input word (and end-markers) and it rewrites a symbol in the cell (except end-markers) only in the first d visits. T. Hibbard showed [11] that for each $d \ge 2$ d-LA recognize (exactly) the class of CFLs, 1-LA recognize the class of regular languages [23] (Thm 12.1). Note that for $d = \infty$, d-LA turns to linear-bounded automata that recognizes context-sensitive languages, so it is quite a natural computational model for the Chomskian hierarchy.

Pighizzini and Pisoni showed in [18] that deterministic 2-LA recognize DCFL. T. Hibbard calls a subclass of CFL, recognizable by deterministic d-LA, a d-deterministic language (d-DCFL). He showed in [11] that for a fixed d, (d+1)-DCFLs strictly contain d-DCFLs (there exists a (d+1)-DCFL that is not a d-DCFL), so all d-DCFLs form a hierarchy.

We also mention that while DCFL is a widely-used subclass of CFL, it has the following practical flaws.

Consider DCFLs $L_d = \{da^nb^nc^m \mid n,m \geqslant 0\}$ and $L_e = \{ea^mb^nc^n \mid n,m \geqslant 0\}$. The language $L_d \cup L_e$ is also a DCFL, while the language $(L_d \cup L_e)^R$, consisting of the reversed words from $L_d \cup L_e$, is not a DCFL. If we allow deterministic pushdown automata (DPDA) to process the input either from left to right or from right to left, the language $(L_d \cup L_e)^R$ could be recognized by the DPDA M_R that acts as a DPDA M_L recognizing $(L_d \cup L_e)$ but M_R processing of the input from right to left. A deterministic 3-LA can shift the head to the rightmost cell and use the same approach of deterministic 2-LA recognizing $(L_d \cup L_e)$ to recognize the language $(L_d \cup L_e)^R$.

Consider now the language $L_{d,e} = \{a^nb^nc^m \mid n,m \geqslant 0\} \cup \{a^mb^nc^n \mid n,m \geqslant 0\}$ which is a union of two DCFLs. It is a well-known fact (see [19]) that $L_{d,e}$ is an inherently ambiguous language. As is also well known, each DCFL is generated by an unambiguous (particularly LR(1)) CF-grammar. Hibbard has also proved in [11] that each d-DCFL is generated by an unambiguous CF-grammar, so the union $\cup_{d=1}^{\infty} d$ -DCFL does not contain all CFLs. This fact implies that d-DCFLs share with DCFL another practical flaw: they are not closed under the union operation, so one needs to apply parallel computation to parse such languages as $L_{d,e}$ in linear time via d-LA.

1.2 d(n)-Limited Automata

We also consider the following natural extension of d-LAs. Assume now that d is not a constant, but a function d(n) depending on the input length n. So, the automaton can change the cell's content until the number of its visits becomes greater or equal d(n). It is the first time, when such a generalization is considered, for the best of our knowledge. In the classical restriction of TMs computation, the time limits are set to the total number of visits of all cells, while we set a time limit per each cell (and after exceeding this limit, it is still allowed to access the content of the cell). Currently, the interesting examples of our technique's application for the general case of d(n) is an open question, while the application for d(n) = O(1) for d-LAs is straight-forward and we describe it in the following subsection.

1.3 Our Contribution

The membership problem takes on the input the computational model M (the deterministic d(n)-LA in our case) and the input word w. The question of the problem, whether M accepts w. Denote by m the length of M's description and by n the length of w. We provide O(mnd(n)) algorithm in RAM for the membership problem that leads to a linear-time simulation algorithm if M is fixed and d(n) = O(1), that is the case of d-LAs.

Hennie proved in [10] that each language recognizable in linear time by a (deterministic) TM is regular (a more general result holds for nondeterministic TMs [21],[16]). So there is no linear-time TM that simulates a deterministic d-LA for $d \ge 2$. B. Guillon and L. Prigioniero proved that each deterministic 1-LA can be transformed into a linear-time TM [9] (and a similar result for the nondeterministic 1-LA). Their construction relies on the classical Shepherdson construction of simulating of two-way deterministic finite automata (2-DFA) by one-way DFA [20]. We also rely on this construction, but it cannot be applied directly due to the Hennie's result (in this case one could simulate a non-regular language by a linear-time TM).

So we transform a classical TM to a TM that operates on a doubly-linked list instead of a tape and transfer Shepherdson's construction to this model. This transformation allows us to obtain a linear-time simulation algorithm (for the d-LAs), but not an O(mn) algorithm for the membership problem. To achieve this algorithm we transform Birget's algebraic constructions [3] into the language of graph theory, and construct a linear-time algorithm which computes a variation of the mapping composition. The generalization to deterministic d(n)-LA simulation algorithm is straight-forward.

Linear-time recognizable languages are used on practice for parsing, especially in compilers. We have already mentioned DCFLs, below we describe PEGs which are very popular now, while their predecessors, top-down parsing languages, were abandoned due to limitations of computers in 1960s. Maybe there will be no direct practical applications of our constructions, but the fact that a language recognizable by a deterministic d-LA is linear-time recognizable can be used to

prove linear-time recognizability of some specific languages and transform the construction to other models, especially PEG.

We also prove the upper-bound $O(d(n)n^2)$ on on the running time of deterministic d(n)-LA's in the case of their simulation by the definition in the cases when the computation does not enter an infinite loop. This result implies the upper-bound $O(n^2)$ for the deterministic d-LA's which was the open question for the best of our knowledge. This bound is tight: it is achieved by a classical example of deterministic d-LA recognizing the language $\{a^nb^n \mid n \geq 0\}$.

From a theoretical point of view our results allow us to prove that some CFLs are easy (linear-time recognizable), while in general a CFL is recognizable in $O(n^{\omega})$ and due to the conditional results hard languages (recognizable at least in superlinear-time) "should" exist. We discuss the details in the following subsection.

1.4 Related Results

We begin with a description of linear-time recognizable subclasses of context-sensitive languages (CSLs) and CFLs. We start with a wide subclass of CSLs recognizable in linear time: the class of languages recognizable by two-way deterministic pushdown automata (2DPDA). A linear time simulation algorithm of 2DPDA was obtained by S. Cook in [5] and then simplified by R. Glück [8]. This class obviously contains DCFLs as a subclass, and it also contains the language of palindromes (over at least two-letters alphabet) that is a well-known example of CF-language that is not a DCFL. It is still an open question whether 2DPDA recognize all CF-languages and the works of L. Lee [14] and Abboud et al. [1] proves that it is very unlikely due to theoretical-complexity assumptions: any CFG parser with time complexity $O(gn^{3-\varepsilon})$, where g is the size of the grammar and n is the length of the input word, can be efficiently converted into an algorithm to multiply $m \times m$ Boolean matrices in time $O(m^{3-\varepsilon/3})$. From our results naturally follows the question: can 2PDPA simulate deterministic d-LA for $d \ge 3$?

Another result is a linear-time parsing algorithm for a non-trivial subclass of CFL: the regular closure of DCFL, obtained by E. Bertsch and M.-J. Nederhof [2]. Each language from this class can be described as follows. Let us take a regular expression and replace in it each letter by a DCFL. This class evidently contains the aforementioned language $L_{d,e}$ (as a union of DCFLs), so it is a strict extension of DCFL. Note that the language $L_{d,e}$ is also recognizable by 2-DPDA.

Another interesting linear-time recognizable subclass of CSLs is the one generated by parsing expression grammars (PEGs). Roughly speaking, PEGs are a modification of CF-grammars that allow recursive calls (and returns from the calls), we do not provide the formal definition here. PEGs are an upgraded version of top-down parsing languages [4] developed by A. Birman and J. D. Ullman. They have been developed by B. Ford who has constructed a practical linear time parser for this model [7]. The class of languages generated by PEGs contains DCFL, such CSLs as $\{a^nb^nc^n \mid n \geq 0\}$, and, as shown in [15] by B. Loff

et al., the language of palindromes with the length of a power of 2. It is an open question, whether PEGs recognize palindromes. The situation with CFLs for PEGs is the same as for 2DPDA: there is no example of a CFL that is not PEG-recognizable, while according to conditional results such a language "should" exist. Another natural question that arises from our results: can PEGs generate all d-DCFLs?

d-LAs were abandoned for decades but then the formal language's community returned to their study. G. Pighizzini, who actively worked on this topic, made a survey of results on d-LA automata and related models [17], focusing in part on state complexity.

We agree with the remark from [17] that T. Hibbard claimed in [11] that deterministic d-limited automata do not recognize palindromes for any d, while the formal proof is missing and would be interesting.

2 Definitions

We define the Hibbard's model, introduced in [11], as it is defined today [17]. An equivalent definition in a more formal style could be found in [18]. Since the difference between d(n)-LAs and d-LAs is not significant, we define firstly d-LAs since the definition is simpler and then generalize it to the case of d(n)-LAs.

2.1 d-Limited Automaton

Fix an integer $d \geqslant 0$. A deterministic d-limited automaton (d-LA) is a deterministic Turing machine with a single tape, which initially contains the input word bordered by the left end-marker \triangleright and the right end-marker \triangleleft with the following property. Each letter of the alphabet has the corresponding number from 0 to d called the rank, initially all the letters of the input word have the rank 0 and the end-markers have the rank d; when the head visits a cell with a letter with rank r < d it rewrites it with a letter of the rank r' such that $r < r' \leqslant d$, and if the letter has the rank d the head does not change the letter.

A d-LA \mathcal{A} is defined by a tuple

$$\mathcal{A} = (Q, \Sigma, \Gamma, \delta, q_0, F),$$

where Q is the finite set of states, Σ is the input alphabet (each letter has the rank 0), Γ is the work-tape alphabet $\Sigma \cup \{\triangleright, \lhd\} \subseteq \Gamma$, we denote by $\Gamma_r \subseteq \Gamma$ the letters of the rank r, q_0 is the initial state, F is the set of accepting states and δ is the transition function

$$\delta: Q \times \varGamma \to Q \times \varGamma \times \{\leftarrow, \to\}$$

so that

- for $a_r \in \Gamma_r$, r < d any transition has the form $\delta(q, a_r) = (q', a_{r'}, m)$, where $a_{r'} \in \Gamma_{r'} \setminus \{ \triangleright, \triangleleft \}, r < r' \leqslant d$.

- for $a_d \in \Gamma_d$ any transition has the form $\delta(q, a_d) = (q', a_d, m)$
- for \lhd any transition has the form $\delta(q, \lhd) = (q', \lhd, \leftarrow)$
- for \triangleright any transition has the form $\delta(q, \triangleright) = (q', \triangleright, \rightarrow)$

A d-LA starts processing of the input word in the state q_0 with the head on the first input symbol. It consequently applies the transition function: being in a state q with a letter a under the head it computes $\delta(q,a)=(q',a',m)$, replaces the letter a by a', changes the state q to q' and moves the head to the right if $m=\to$ or to the left if $m=\leftarrow$. The automaton accepts the input if it reaches an accepting state $q_f\in F$ when the head arrives on the right end-marker \lhd . At this point it stops the computation.

In fact we have modified the original definition a little, since we require the transition function to be the totally defined function. This modification does not change the class of recognizable languages and costs of adding only one extra-state.

2.2 Deleting LA

We also construct an auxiliar modified model as follows. In this model there is no constrain on d-visits, the tape is replaced by a doubly linked list, so an automaton can delete an arbitrary cell between the end-markers (but not the end-markers). Formally we modify only the transition function as follows

$$\delta: Q \times \Gamma \to (Q \times (\Gamma \cup \{\bot\}) \times \{\leftarrow, \to\} \cup \{\uparrow\}),$$

where the symbol \bot means that the cell would be deleted right after the head leaves the cell. After the deletion of the cell, the head moves from its left neighbour to its right neighbour when it moves to the right from the left neighbour and vice versa. If the transition function returns \uparrow the computation is over and the input is rejected. We have no extra-requirements on the transition function.

We call the modified model the deleting LA.

Deleting LAs obviously recognize (exactly) deterministic CSLs. They can simulate linear-bounded automata by the construction and the latter can simulate deleting LAs via marking deleted cell by a special symbol. We use doubly linked lists in this model to achieve the claimed upper bound for deterministic d(n)-LAs.

2.3 d(n)-Limited Automaton

In the case of d-LAs it is convenient to associate with a letter a_r its rank r (the number of the last head's visit). In the case of d(n)-LAs it is impossible, since the alphabet and the TM's description is fixed and cannot grow with an input length. So, in the case of d(n)-LAs, each cell has the corresponding counter of visits and the rank is associated with the cell, but not with a letter. Moreover, each cell (except the borders with $\triangleright, \triangleleft$) contains a pair (a, e), where $a \in \Gamma$ and e is the bit, which is set to 0 until the rank of the cell is less than d(n) and after

it becomes greater or equal to d(n) the bit e is set to 1. After e=1, the d(n)-LA is unable to change the content of the cell during the visits. The described modification of d-LAs does not affect the simulation algorithm, since the value of d(n) is the same during the whole simulation for the input of length n and the algorithm never uses the fact that d is a constant, but not a precomputed value. So we describe the simulation algorithm for d-LAs only.

3 Linear-Time Simulation Algorithm

In this section we provide a linear-time simulation algorithm for d-LAs. The main idea is as follows. If for all inputs each cell of a deleting LA is visited at most C times (for some constant C) then it works in linear time. It also works in linear time if the average number of visits per cell is at most C. So we construct a deleting LA which satisfies the latter property from a d-LA. When at some point the d-LA has on its tape a maximal subword with only letters of rank d, the deleting LA has only one cell with auxiliary information for this subword which it uses to simulate the behavior of the d-LA on processing this subword.

Our simulation idea is similar to the Shepherdson's well-known simulation algorithm of a two-way DFA by a one-way DFA [20]. Note that in the case d=0, d-LA is a two-way DFA. The deleting LA writes in the cells that should contain letters of rank d the corresponding mappings of possible moves and if two mappings are written in adjacent cells it deletes one of the cells and replaces the mapping in the other cell by the composition of the mappings. When the head arrives at the cell with a mapping

$$f: (Q \times \{\leftarrow, \rightarrow\}) \rightarrow (Q \times \{\leftarrow, \rightarrow\} \cup \{\uparrow\})$$

in the state q and the last move's direction was $m \in \{\leftarrow, \rightarrow\}$, the deleting LA computes f(q,m) = (q',m') and moves the head in the state q' to the direction m'. If f returns \uparrow it means that the d-LA entered an infinite loop, so the deleting LA rejects the input in this case.

To simplify the notation we use the following shortcuts:

$$\overleftarrow{Q} = Q \times \{\leftarrow\}, \ \overrightarrow{Q} = Q \times \{\rightarrow\}, \ \overleftarrow{Q} = \overleftarrow{Q} \cup \overrightarrow{Q}, \ A_{\uparrow} = A \cup \{\uparrow\} \text{ for } A \in \{\overleftarrow{Q}, \overrightarrow{Q}, \overleftarrow{Q}\}.$$

We also use arrows to indicate the elements of these sets, i. e. $\overleftarrow{q} \in \overleftarrow{Q}$, and call elements of \overrightarrow{Q} directed states. If $\delta(q,X) = (p,Y,m)$, where δ is the transition function of a d-LA or a deleting LA, we denote $\delta(q,X) = (\overleftarrow{p},Y)$, where the direction of p corresponds to the value of m.

We start with the technical details. We refer to d-LA as \mathcal{A} and to deleting LA (that simulates \mathcal{A}) as M. We enumerate all cells (of each automata) from 0 to n+1, where n is the length of the input word w and denote the content of the i-th cell on the current M's step as $W_M[i]$; we refer to the content of the

¹ that cannot be continued neither to the right nor to the left satisfying the following property

i-th cell right after the *t*-th step as W_M^t . So $W_M^0[0] = \triangleright$ and $W_M^0[n+1] = \triangleleft$. We refer to the content of \mathcal{A} 's tape as $W_{\mathcal{A}}$.

We fix this enumeration for the whole computation, and since a cell could be deleted we refer to the left (undeleted) neighbour of an i-th cell as i-prev and to the right neighbour as i-next.

Now we describe M. We use indices \mathcal{A} and M for the components of tuples that describe automata to make the notation clear. The working alphabet Γ_M is the union of the alphabet $\Gamma_{\mathcal{A}}$ and the set \mathcal{F} of all mappings

$$f: \overleftrightarrow{Q}_{\uparrow} \to \overleftrightarrow{Q}_{\uparrow}$$
, such that $f(\uparrow) = \uparrow$.

We continue the description of M after we state the auxiliary properties of the family \mathcal{F} .

Definition 1. Fix $f \in \mathcal{F}$, and a segment of the d-LA's tape

$$W_{\mathcal{A}}[l,r] = W_{\mathcal{A}}[l]W_{\mathcal{A}}[l+1]\cdots W_{\mathcal{A}}[r]$$

that contains only symbols of rank d. Define a directed state $\overleftrightarrow{p} \in \overleftrightarrow{Q}$ as follows. If \mathcal{A} arrives at the cell $W_{\mathcal{A}}[r]$ in a state q then $\overleftrightarrow{p} = f(\overleftarrow{q})$ unless $f(\overleftarrow{q}) = \uparrow$. Symmetrically, if \mathcal{A} arrives at the cell $W_{\mathcal{A}}[l]$ in a state q then $\overleftarrow{p} = f(\overrightarrow{q})$ unless $f(\overrightarrow{q}) = \uparrow$. We say that the mapping f describes the segment $W_{\mathcal{A}}[l, r]$ if for all q the automaton \mathcal{A}

- leaves the segment (firstly after arriving in the state q) in the state p and arrives at a cell r + 1 if $\overrightarrow{p} = \overrightarrow{p}$ (and the the directed state \overrightarrow{p} is defined)
- leaves the segment in the state p and arrives at a cell l-1 if $\overrightarrow{p} = \overleftarrow{p}$ (the same assumptions hold)
- never leaves the segment if f returned \uparrow .

It is clear that each segment $W_{\mathcal{A}}[l,r]$ with only the letters of rank d is described by some mapping $f \in \mathcal{F}$. A mapping $f \in \mathcal{F}$ may not describe any segment of tape of any run of \mathcal{A} , but we still consider such f as a possible description of a segment to define the directed composition of mappings formally.

Assume that a mapping f describes a segment $W_{\mathcal{A}}[L,r]$, and a mapping g describes a segment $W_{\mathcal{A}}[r+1,R]$. The directed composition or d-composition $f \diamond g$ is the mapping h that describes the segment $W_{\mathcal{A}}[L,R]$.

Proposition 2. The d-composition of mappings from \mathcal{F} is well-defined and is always a computable and an associative mapping. The d-composition is computable in $O(|Q_A|)$.

We provide proofs of Propositions 2 and 3 after the latter. As we mentioned in Subsection 1.3 our algorithm of directed composition computing is an effective variant of the Birget's construction, so we do not go deep into details here, since one can find them in [3].

Let f and g describe segments $W_{\mathcal{A}}[L, r]$ and $W_{\mathcal{A}}[r+1, R]$ and \overrightarrow{q} be a directed state such that \overrightarrow{q} means that the head arrives at $W_{\mathcal{A}}[r+1, R]$ from $W_{\mathcal{A}}[L, r]$ in

the state q and q means the arrival at $W_{\mathcal{A}}[L,r]$ from $W_{\mathcal{A}}[r+1,R]$. We define the mapping $D: \mathcal{F} \times \mathcal{F} \times \overrightarrow{Q} \to \overrightarrow{Q}_{\uparrow}$ that returns the directed state \overrightarrow{p} such that \mathcal{A} leaves the segment $W_{\mathcal{A}}[L,R]$ in the state p and the corresponding direction (after arriving in the directed state q) or \uparrow if the head never leaves $W_{\mathcal{A}}[L,R]$. We call D the departure function.

And finally we need the computability of function $CF : \Gamma_d \to \mathcal{F}$ that returns the mapping CF(X) that describes a cell with the letter X of rank d. We call CF the cell description function.

Proposition 3. The departure function D and the cell description function CF are computable in $O(|Q_A|)$.

Proof (of Propositions 2 and 3). The CF function is evidently $O(|Q_{\mathcal{A}}|)$ -computable. Denote $CF(a_d)$ by f. So $f({\stackrel{\hookleftarrow}{q'}}) = {\stackrel{\longleftarrow}{p'}}$ iff $\delta_{\mathcal{A}}(q,a_d) = ({\stackrel{\longleftarrow}{p'}},a_d)$.

We provide effective algorithms for computing D and \diamond mappings via graphs. A mapping $f \in \mathcal{F}$ that describes a segment L is represented via 4-parted graph with parts L in, L in, L out, and L out as follows. Each part is a copy of the set $Q_{\mathcal{A}}$. The graph has an edge $\overrightarrow{q} \to \overrightarrow{p}$, $\overrightarrow{q} \in L$ in, $\overrightarrow{p} \in L$ out iff $f(\overrightarrow{q}) = \overrightarrow{p}$. So, $f(\overrightarrow{q}) = \uparrow$ iff the vertex $\overrightarrow{q} \in L$ in has degree 0.

Assume that g describes the segment R adjacent to L (to the right). In terms of graphs, the directed composition $f \diamond g$ is computable as follows. We glue the graphs for f and g so that $L_{\mathsf{out}} = R_{\mathsf{in}}$ and $L_{\mathsf{in}} = R_{\mathsf{out}}$ (Fig. 1) and obtain the intermediate graph. So $f \diamond g({}^{\backprime}q) = p$ iff there is a path from q to p in the intermediate graph (by the graph's construction).

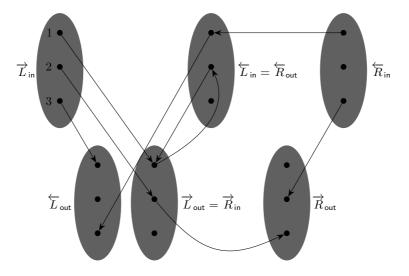


Fig. 1. Graph for computing the composition

To compute $h = f \diamond g$ in $O(|Q_A|)$ we use the algorithm on Fig. 2. Denote by V the set of all vertices. Note that each vertex $u \in V$ has out degree at most 1, so we denote by u.next either the end of the edge $u \to u$.next or u.next $= \uparrow$ if u has out degree 0. We store h in an array (enumerate all vertices in the set \overrightarrow{Q}) and also use the array of marks m.

```
1 \ m[u] := \downarrow \text{ for each } u \in V;
                                                                                              /* Mark vertices by the empty label */
  \mathbf{2} \ h[\overrightarrow{q}] := \downarrow \text{ for each } \overrightarrow{q} \in \overrightarrow{L}_{\text{in}} \cup \overleftarrow{R}_{\text{in}};
 3 for \overleftrightarrow{q} \in \overrightarrow{L}_{\mathsf{in}} \cup \overleftarrow{R}_{\mathsf{in}} do
                  u := \stackrel{\longleftarrow}{q} .\mathsf{next};
                   while u \notin \overleftarrow{L}_{\text{out}} \cup \overrightarrow{R}_{\text{out}} \cup \{\uparrow\} \text{ and } h[\overleftarrow{q}] = \downarrow \mathbf{do}
                           if m[u] = \downarrow then
m[u] := \stackrel{\longleftarrow}{q};
u := u.\mathsf{next};
else if m[u] = \stackrel{\longleftarrow}{q} then
   6
   7
   8
  9
                               h[\overrightarrow{q}] := \uparrow;
10
11
                              12
13
14
                  if h[\overleftrightarrow{q}] = \downarrow \mathbf{then} \ h[\overleftrightarrow{q}] := u;
15
```

Fig. 2. Algorithm of d-composition computing

The algorithm travels from the directed state \overrightarrow{q} on the input of h until it either reaches an output state, or it reaches a marked vertex. If the vertex of the intermediate graph is marked by \overrightarrow{s} it means that it was already used during traveling from the directed state \overrightarrow{s} . If $\overrightarrow{s} = \overrightarrow{q}$ it means that we met a loop, so $h(\overrightarrow{s}) = \uparrow$, otherwise $h(\overrightarrow{q}) = h(\overrightarrow{s})$ since the ends of paths for states \overrightarrow{q} and \overrightarrow{s} are the same. The marking guaranties that we never visit any edge twice. Since out degree of each vertex is at most 1 the number of edges |E| in the graph is O(|V|). So the algorithm runs in the time $O(|V| + |E|) = O(|V|) = O(|Q_A|)$.

To compute the departure function D we firstly compute the intermediate graph and run the algorithm (Fig. 2). The result of $D(f,g, \stackrel{\cdot}{s})$ is computed as follows. $D(f,g,\stackrel{\cdot}{s})=h[m[\stackrel{\cdot}{s}]]$ if $m[\stackrel{\cdot}{s}]\ne\downarrow$, since in this case $m[\stackrel{\cdot}{s}]=\stackrel{\cdot}{q}\in \overrightarrow{L}_{\rm in}\cup \overrightarrow{R}_{\rm in}$ and the result $h(\stackrel{\cdot}{q})$ has bin already computed by the algorithm (Fig. 2). If $m[\stackrel{\cdot}{s}]=\downarrow$, it means that the vertex $\stackrel{\cdot}{s}$ has not bin visited by the algorithm, so we travel in the intermediate graph from $\stackrel{\cdot}{s}$ (with labeling the vertices) until we reach either a labeled vertex u or a vertex v of out-degree 0. So $D(f,g,\stackrel{\cdot}{s})=\uparrow$ if u was reached and $m[u]=\stackrel{\cdot}{s}$ or v was reached and $v\not\in \overrightarrow{L}_{\rm out}\cup \overrightarrow{R}_{\rm out}$, or $D(f,g,\stackrel{\cdot}{s})=h[m[u]]$ if u was reached and $m[u]\not=\stackrel{\cdot}{s}$, or $D(f,g,\stackrel{\cdot}{s})=v$ if $v\in \overrightarrow{L}_{\rm out}\cup \overrightarrow{R}_{\rm out}$. The algorithm of computing of $D(f,g,\stackrel{\cdot}{s})$

is correct due to the construction of the intermediate graph and the definition of D.

3.1 Simulation algorithm

```
\mathbf{1} \stackrel{\longleftrightarrow}{q} := \overleftarrow{q_0}; \quad i := 1;
     while no result returned do
           case W_M[i] \in \Gamma_r \cup \{ \triangleright, \triangleleft \}, r < d-1 do
                                                                                                               /* A-move */
             (\stackrel{\longleftarrow}{p}, a_{r'}) := \delta_{\mathcal{A}}(q, W_M[i]); \ W_M[i] := a_{r'};
            case W_M[i] \in \Gamma_{d-1} do
                                                                                                  /* Deletion scan */
  5
                  (\stackrel{\longleftrightarrow}{p}, X) := \delta_{\mathcal{A}}(q, W_M[i]);
  6
                  g := CF(X);
                                                                            /* O(|Q|). See Proposition 3 */
  7
                  if i.\mathsf{prev} > 0 and W_M[i.\mathsf{prev}] \in \mathcal{F} then
  8
                        f := W_M[i.prev];
  9
                        if \overleftarrow{p} = \overleftarrow{p} then \overleftarrow{p} := D(f, g, \overleftarrow{p});
10
                        if \overrightarrow{p} = \uparrow then return Reject;
11
                        g:=f\diamond g; \;\; i.\mathsf{prev}:=(i.\mathsf{prev}).\mathsf{prev};
                                                                                                               /* O(|Q|). */
12
13
                  end
                  if i.next < n+1 and W_M[i.next] \in \mathcal{F} then
14
                        h := W_M[i.\mathsf{next}];
15
                        if \overrightarrow{p} = \overrightarrow{p} then \overrightarrow{p} = D(g, h, \overrightarrow{r});
                                                                                                              /* O(|Q|). */
16
                        if \overrightarrow{p} = \uparrow then return Reject;
17
                                                                                                               /* O(|Q|). */
                        g := g \diamond h; i.\mathsf{next} := (i.\mathsf{next}).\mathsf{next};
18
                  end
19
                  W_M[i] := g;
20
            case W_M[i] \in \mathcal{F} do
21
                  f := W_M[i];
22
                 if f(\overrightarrow{q}) = \uparrow then return Reject else \overrightarrow{p} = f(\overrightarrow{q});
23
24
            if \overrightarrow{p} = \overleftarrow{p} then i := i.prev else i := i.next;
25
           if q \in F_A and W[i] = \triangleleft then return Accept;
28 end
```

Fig. 3. Simulation Algorithm

We provide the pseudocode of the simulation algorithm in Figure 3 and now we also describe the algorithm. It provides a high-level description of M's transition function δ_M . When we describe M's behavior and say " \mathcal{A} moves", " \mathcal{A} acts", etc., we refer to the result of the move $\delta_{\mathcal{A}}(q,a)$ where $a \in \Gamma$ is either the mentioned symbol or the symbol under M's head and q is the mentioned state or \mathcal{A} 's state corresponding to M's state. \mathcal{A} moves described in lines 3...4 of the pseudocode. M has two kinds of moves: \mathcal{A} -moves and technical moves. The \mathcal{A} -moves correspond to moves of \mathcal{A} , so M always has a state $q \in Q_{\mathcal{A}}$ when it arrives at a cell after an \mathcal{A} -move.

Since our algorithm depends on the first head's arrival to the cell of rank d-1 (after which it becomes d), we have formal problems with the case d=0. To avoid them, assume that in the beginning all cells have rank -1 (only for d=0) and the d-LA does not change the symbols of rank -1, but changes their rank.

The automaton M acts as \mathcal{A} if M processes a letter of rank less than d-1 (\mathcal{A} -moves). When M visits a cell on the d-th time and \mathcal{A} should have written a letter X of rank d in that cell (and X is not an end-marker), M writes to the corresponding cell the mapping g = CF(X), recall that $g(\overrightarrow{q}) = \overrightarrow{p}$ iff $\delta_{\mathcal{A}}(q,X) = (\overrightarrow{p},X)$ for all $q \in Q_{\mathcal{A}}, X \in \Gamma_d$. When M writes a mapping g in the cell i for the first time, it scans the cells i-prev and i-next and performs the following procedure that we call a d-eletion s-can.

If only one of the cells i.prev and i.next contains a mapping (f or h respectively), then M writes to the cell i the mapping $f \diamond g$ for i.prev and the mapping $g \diamond h$ for i.next and deletes the neighbouring cell. If both neighbours have the rank d, M writes to the cell i the mapping $f \diamond g \diamond h$ and deletes both neighbouring cells. After the deletion scan the cell i contains the result mapping g while the neighbours of i contain letters, so g describes the segment $W_A[i.\text{prev}+1,i.\text{next}-1]$ and M moves the head to the same cell (i.prev or i.next) as A after A leaves the segment $W_A[i.\text{prev}+1,i.\text{next}-1]$. This cell is computed via the departure function D during the deletion scan.

We described the cases when M arrives at a cell of rank r < d from a cell of any rank and to a cell of rank d from a cell of rank d. So, it is left to describe M's action in the case when M arrives at a cell containing a mapping f in a directed state \overrightarrow{q} from a cell with a letter of rank r < d (lines 21...24). M computes $f(\overrightarrow{q}) = \overrightarrow{p}$ and moves the head to the left neighbour or to the right neighbour depending on the direction of \overrightarrow{p} and arrives at the neighbour in the state p.

Lemma 4. For each d-LA \mathcal{A} the described deleting LA M simulates \mathcal{A} , i.e. if on the t-th step \mathcal{A} visits the cell i with the letter $W_{\mathcal{A}}^{t}[i]$ of rank less than d-1 or an end-marker then $W_{\mathcal{A}}^{t}[i] = W_{M}^{t'}[i]$, where t' is the corresponding step² of M, and \mathcal{A} and M have the same states³; M accepts the input iff \mathcal{A} does.

Proof. Assume that the automaton \mathcal{A} has performed N moves

$$\delta_1, \dots, \delta_N, \quad \delta_i \in Q \times \Gamma \times \{\leftarrow, \rightarrow\}, \ \delta_1 = \delta(q_0, W_A^0[1])$$
 (1)

on a fixed input and either accepted the input or has come to a loop. We call a move $\delta_i=(q,a,m)$ a d-move if a is a symbol of rank d but not an end-marker, otherwise we call δ_i a regular move. Note that regular moves were described in the statement of the theorem. We call the sequence (1) a run. A run begins with a regular move and is partitioned into alternating segments of regular moves and d-moves.

² We demand that each such step t has the corresponding step t' and if $t_1 < t_2$ then $t'_1 < t'_2$.

³ When M arrives at a cell (except during a deletion scan) it always has a state $q \in Q_A$.

The definition of a run for the automaton M is the same, so the definition of regular moves is also the same. A d-move for M is either a move to a cell that contains a mapping or a move during a deletion scan that begins with arriving at a cell with a letter of rank d-1. We show that if we delete from runs of \mathcal{A} and M all d-moves then we obtain identical sequences of regular moves; we also show that after a series of d-moves both automata come to the same cell in the same state. These conditions imply that M accepts a word iff \mathcal{A} does, since each accepting move of \mathcal{A} is a regular move. Note that the correspondence between the indices of δ_i s in the sequences of regular moves is the bijection $t \mapsto t'$.

In fact, it is enough to prove only the second condition, because it implies that $W_M^{t'}[i] = W_A^t[i]$ if $W_A^t[i]$ is a letter of rank less than d, because, if for both automata, a series of regular moves starts in the same cells with the same states then M acts as A on this series due to the construction of M.

So, we prove that if the last moves of \mathcal{A} and M before the series of d-moves were the same, then after the series of d-moves both automata arrive at the same cell in the same state.

Since the series of regular moves of \mathcal{A} and M are the same if they began in the same cell in the same state, the last moves of the series are the same, so the first moves of both series of d-moves (after the series of regular moves) start in the same state.

There are two cases for the first move of the series of d-moves for \mathcal{A} . In the first case the head visits a cell i with a symbol a of rank d-1. In the end of this move, \mathcal{A} changes the symbol to a symbol X of rank d. On the corresponding move the automaton M writes to the cell i the mapping f_X that describes the segment of the only cell $W_{\mathcal{A}}[i]$ by f_X 's construction. If the d-series of \mathcal{A} consists of this move only or on the next move \mathcal{A} visits the cell with a letter of rank d-1, both automata arrive at the same cell in the same state due to the definition of the segment's description by the mapping. Otherwise, \mathcal{A} arrives at a neighbour of i with the letter of rank d and at some point leaves the segment of tape consisting of cells with letters of rank d. In this case M arrives at the same cell (at the end of the series) in the same state as \mathcal{A} , due to the construction of M and Propositions 2 and 3. If both automata arrive at a cell with a symbol of rank less than d-1 then they begin the next series of regular moves in the same cells with the same states. Otherwise both automata arrive at a cell with a symbol of rank d-1 and we follow the process from the begin of the first case.

In the second case the series of \mathcal{A} 's d-moves starts with a cell with a letter of rank d, so \mathcal{A} arrives at the segment of tape with letters of rank d (maybe consisting of the only cell) and M arrives at the cell that contains a mapping f that describes this segment (this invariant holds due to the deletion scan procedure that is correct due to Propositions 2 and 3). Since f describes the corresponding \mathcal{A} 's segment, after M moves according to the result of f, it arrives at the same cell as \mathcal{A} does and in the same state. Then either the d-series is finished and the required property holds or after \mathcal{A} leaves the segment it comes to a cell with a letter of rank d-1 and then we come to the first case.

If, during any of the cases, \mathcal{A} has come to a loop then M rejects the input according to the construction of the mapping that describes \mathcal{A} 's segments (the mapping returns \uparrow in the case of a loop).

Now we prove that the simulation algorithm for deterministic d-LAs works in linear time. We provide the prove for the general case of d(n)-LA. Note that the simulation algorithm for d(n)-LA is the same as for the ordinary d-LA since the algorithm's depends only on the fact whether the cell's last visit number equals to d(n) - 1 or it is less. The counters of cell's visits needed for d(n)-LAs can be easily implemented in the RAM model without affecting the asymptotic.

Lemma 5. The automaton M performs $O(d(n) \cdot |Q_A| \cdot n)$ steps on processing the input of length n.

Proof. We use amortized analysis [6], namely the accounting method. We describe the budget strategy. Each cell $i \in \{1, ..., n\}$ (on M's tape) has it's own budget B[i] (credit, in terms of [6]) and we denote its value after the t-th step as $B^t[i]$.

We account budgets according to the following rules. The budget $B^{t}[k]$ is defined for the cell k (either k = i or k = j) at the step t if it satisfies the corresponding rule:

- $-B^0[i] = 2d(n)$ for all i
- $-B^{t}[i] = B^{t-1}[i] 1$ if the cell i is visited at the step t and contains a letter (i.e. has been visited less than d(n) times)
- $-B^{t}[j] = B^{t-1}[j] 1$ if at the step t the head arrives at the cell i from the cell j and the i-th cell contains a mapping and the j-th cell contains a letter $-B^{t}[i] = B^{t-1}[i]$ if the previous rules are not applicable

We do not change the budget during deletion scans.

Fix a step t. Assume that the i-th cell contains a mapping $f = W_M^t[i]$ that describes a segment $W_A[l,r]$. Note that the cells $W_M[l-1] = W_A[l-1]$ and $W_M[r+1] = W_A[r+1]$, the neighbours of the segment, have a rank less than d(n), so the head has visited each neighbour of the segment fewer than d(n) times. So when the head moves from a neighbour to the segment the neighbour pays for this visit. Until the neighbour joins the segment it would not pay more than d(n)\$ for the segment's visits, and after it joins the segment (if it happens) the following visits from its side would be paid for a new neighbour. So, each cell pays 1\$ for each visit of itself and at most 1\$ for each visit of a mapping in a neighbouring cell until it becomes a mapping itself. Since the latter happens after at most d(n) visits of the cell, each cell pays at most 2d(n)\$. These explanations are consistent with the described budget strategy, so we have proved that for all t and i the assertions $B^t[i] \geqslant 0$ hold.

We did not count above visits of deletion scan. It is clear that there are at most O(n) deletion scans since each cell can invoke at most one deletion scan. During a deletion scan the algorithm computes at most two d-compositions, each of which runs in $O(|Q_A|)$. So, all deletion scans are done in $O(n|Q_A|)$ time.

All in all, there are the following cases for M operations:

- 1. A move that ends on a letter, but not an end-marker
- 2. A move that ends on a mapping
- 3. Deletion scan
- 4. A move that ends on an end-marker

Via amortized analysis we have shown that Cases 1 and 2 both takes $O(d(n) \cdot |Q_{\mathcal{A}}| \cdot n)$: the number of all such moves is bounded by $O(d(n) \cdot n)$ (since $\sum_{i=1}^{n} B^{0}[i] = 2n \cdot d(n)$) and each move is performed in at most $O(|Q_{\mathcal{A}}|)$ (Propositions 2 and 3). As we discussed, Case 3 takes $O(n|Q_{\mathcal{A}}|)$ time, so it is left to prove that Case 4 takes $O(d(n) \cdot |Q_{\mathcal{A}}| \cdot n)$ time as well.

Since after the head leaves the left end-marker \triangleright during the first move, each of the end-markers is visited only after arriving from the inner cell, the number of end-markers visits does not exceed the number of visits of all the rest cells which is $O(n \cdot d(n))$. Since each step of the simulation algorithm (Fig. 3) takes at most $O(|Q_A|)$ M's steps, M performs $O(d(n) \cdot |Q_A| \cdot n)$ steps during the visits as end-markers, and so as all the cells as well during processing of the input of length n.

Now we summarize the simulation results into the following theorem.

Theorem 6 (Main result). For each d(n)-LA \mathcal{A} the membership problem is solvable in time $O(d(n) \cdot m \cdot n)$, where n is the length of the input word w and m is the length of \mathcal{A} 's description, provided that d(n) is computable in time O(n). Algorithm 3 provides the simulation of \mathcal{A} on w and runs in time $O(d(n) \cdot m \cdot n)$ provided that subroutines are performed via Algorithm 2.

4 Upper bound on deterministic d(n)-LA runtime

In this section we provide the upper-bound on the runtime of deterministic d(n)-LA of the classical simulation (according to the definition).

Theorem 7. A deterministic d(n)-LA with m states runs in $O(d(n) \cdot n^2 \cdot m)$ steps on the input of length n in the case when it does not enter an infinite loop.

Proof. Note that, if the head travels through a segment of the tape with symbols of rank d(n) in more than mn steps, then there is a cell that has been visited at least twice in the same state, because the average number of states per cell is greater than m. So the computation has come to an infinite loop.

As described in the proof of Lemma 4, deterministic d(n)-LA's have two kinds of moves: regular ones, when the head arrives at a cell of rank less than d(n), and d(n)-moves, when the head arrives at the segment with the cells of rank d(n). A series of d(n)-moves cannot take more than mn steps if the computation does not enter an infinite loop. Note that each series of d(n)-moves shall precede a regular move and the number of regular moves is bounded by $O(n \cdot d(n))$ since after each regular move the rank of the arrived cell increases and it increases only up to d(n). Since a regular move takes O(1) steps and a series of d(n)-moves occurs

only after a regular move and takes O(mn) steps, the total number of steps is bounded by $O(d(n) \cdot n^2 \cdot m)$.

Theorem 7 implies that a deterministic d-LA runs in $O(dmn^2)$. This bound is obviously tight: a classical 2-LA recognizing the language $\{a^nb^n\mid n\geqslant 0\}$ runs in quadratic time. It works as follows: the head moves until meets the first b, then it goes left until finds the first a of rank 1 (which changes to 2 after the visit). After the corresponding a found, the automaton goes right to check, whether there is a b of rank 1 and if it is, then it looks for the corresponding a and so on. When the automaton meets the right end-marker \triangleleft , it checks that there are no a's of rank 1 left and accepts the input in this case and rejects otherwise.

Acknowledgments

The author is thankful to Dmitry Chistikov for the feedback and a discussion of the text's results, and for suggestions for improvements.

References

- 1. Abboud, A., Backurs, A., Williams, V.V.: If the current clique algorithms are optimal, so is valiant's parser. p. 98–117. FOCS '15, IEEE Computer Society, USA (2015)
- Bertsch, E., Nederhof, M.J.: Regular closure of deterministic languages. SIAM Journal on Computing 29(1), 81–102 (1999)
- Birget, J.C.: Concatenation of inputs in a two-way automaton. Theoretical Computer Science 63(2), 141–156 (1989)
- 4. Birman, A., Ullman, J.D.: Parsing algorithms with backtrack. In: 11th Annual Symposium on Switching and Automata Theory (swat 1970). pp. 153–174 (1970)
- 5. Cook, S.A.: Linear time simulation of deterministic two-way pushdown automata. Department of Computer Science, University of Toronto (1970)
- 6. Cormen, T., Leiserson, C., Rivest, R., Stein, C.: Introduction to Algorithms, fourth edition. MIT Press (2022)
- Ford, B.: Parsing expression grammars: A recognition-based syntactic foundation. SIGPLAN Not. 39(1), 111–122 (Jan 2004)
- 8. Glück, R.: Simulation of two-way pushdown automata revisited. In: Electronic Proceedings in Theoretical Computer Science. vol. 129, p. 250–258. Open Publishing Association (Sep 2013)
- 9. Guillon, B., Prigioniero, L.: Linear-time limited automata. Theor. Comput. Sci. **798**, 95–108 (2019)
- 10. Hennie, F.: One-tape, off-line turing machine computations. Information and Control 8(6), 553–578 (1965)
- Hibbard, T.N.: A generalization of context-free determinism. Information and Control 11(1/2), 196–238 (1967)
- 12. Jayaram, R., Saha, B.: Approximating language edit distance beyond fast matrix multiplication: Ultralinear grammars are where parsing becomes hard! In: ICALP 2017. pp. 19:1–19:15 (2017)
- 13. Knuth, D.: On the translation of languages from left to right. Information and Control 8, 607–639 (1965)

- Lee, L.: Fast context-free grammar parsing requires fast boolean matrix multiplication. J. ACM 49(1), 1–15 (2002)
- 15. Loff, B., Moreira, N., Reis, R.: The computational power of parsing expression grammars. In: DLT 2018. pp. 491–502. Springer, Cham (2018)
- 16. Pighizzini, G.: Nondeterministic one-tape off-line turing machines and their time complexity. J. Autom. Lang. Comb. 14(1), 107–124 (2009)
- 17. Pighizzini, G.: Limited automata: Properties, complexity and variants. In: DCFS 2019. pp. 57–73. Springer, Cham (2019)
- 18. Pighizzini, G., Pisoni, A.: Limited automata and context-free languages. In: Fundamenta Informaticae. vol. 136, pp. 157–176. IOS Press (2015)
- Shallit, J.O.: A Second Course in Formal Languages and Automata Theory. Cambridge University Press (2008)
- 20. Shepherdson, J.C.: The reduction of two-way automata to one-way automata. IBM Journal of Research and Development **3**(2), 198–200 (1959)
- Tadaki, K., Yamakami, T., Lin, J.C.: Theory of one-tape linear-time turing machines. Theoretical Computer Science 411(1), 22–43 (2010)
- 22. Valiant, L.G.: General context-free recognition in less than cubic time. J. Comput. Syst. Sci. 10(2), 308–315 (1975)
- 23. Wagner, K., Wechsung, G.: Computational complexity. Springer Netherlands (1986)