Electronic Colloquium on Computational Complexity, Revision 1 of Report No. 104 (2015)

Streaming Property Testing of Visibly Pushdown Languages*

Nathanaël François†1, Frédéric Magniez‡2, Michel de Rougemont§3, and Olivier Serre¶2

[1]Univ Paris Diderot, Sorbonne Paris-Cité, LIAFA, CNRS, France
[2]CNRS, LIAFA, Univ Paris Diderot, Sorbonne Paris-Cité, France
[3]University of Paris II and LIAFA, CNRS, France

Abstract

In the context of language recognition, we demonstrate the superiority of streaming property testers against streaming algorithms and property testers, when they are not combined. Initiated by Feigenbaum *et al*, a streaming property tester is a streaming algorithm recognizing a language under the property testing approximation: it must distinguish inputs of the language from those that are ε-far from it, while using the smallest possible memory (rather than limiting its number of input queries).

Our main result is a streaming ε-property tester for visibly pushdown languages (VPL) with one-sided error using memory space $\mathrm{poly}((\log n)/\varepsilon)$.

This constructions relies on a new (non-streaming) property tester for weighted regular languages based on a previous tester by Alon *et al*. We provide a simple application of this tester for streaming testing special cases of instances of VPL that are already hard for both streaming algorithms and property testers.

Our main algorithm is a combination of an original simulation of visibly pushdown automata using a stack with small height but possible items of linear size. In a second step, those items are replaced by small sketches. Those sketches relies on a notion of suffix-sampling we introduce. This sampling is the key idea connecting our streaming tester algorithm to property testers.

*Partially supported by the French ANR projects ANR-12-BS02-005 (RDAM) and ANR-14-CE25-0017 (AGREG)
†nathanael.francois@liafa.univ-paris-diderot.fr
‡frederic.magniez@cnrs.fr
§mdr@liafa.univ-paris-diderot.fr
¶Olivier.Serre@cnrs.fr

ISSN 1433-8092

1 Introduction

Visibly pushdown languages (VPL) play an important role in formal languages with crucial applications for databases and program analysis. In the context of structured documents, they are closely related with regular languages of unranked trees as captured by hedge automata. A well-known result [3] states that, when the tree is given by its depth-first traversal, such automata correspond to visibly pushdown automata (VPA) (see *e.g.* [19] for an overview on automata and logic for unranked trees). In databases, this word encoding of trees is known as XML encoding, where DTD specifications are examples of often considered subclasses of VPL. In program analysis, VPA also permit to express natural properties of traces of executions of recursive finite-state programs, including non-regular ones such as those with pre and post conditions as expressed in the temporal logic of calls and returns (CaRet) [5, 4].

Historically VPL got several names such as input-driven languages or, more recently, languages of nested words. Intuitively, a VPA is a pushdown automaton whose actions on stack (push, pop or nothing) are solely decided by the currently read symbol. As a consequence, symbols can be partitioned in three parts: push, pop and neutral symbols. The complexity of VPL recognition has been addressed in various computational models. The first results go back to the design of logarithmic space algorithm [11] as well as NC^1-circuits [13]. Later on, other models motivated by the context of massive data were considered such as streaming algorithms and property testers (described below).

Streaming algorithms (see *e.g.* [23]) have only a sequential access to their input, on which they can perform a unique pass, or sometimes a small number of additional passes. The size of their internal (random access) memory is the crucial complexity parameter, which should be sublinear in the input size, and even polylogarithmic if possible. The area of streaming algorithms has experienced tremendous growth in many applications since the late 1990s. The analysis of Internet traffic [2], in which traffic logs are queried, was one of their first applications. Nowadays, they have found applications with big data, notably to test graphs properties, and more recently in language recognition on very large inputs. The streaming complexity of language recognition has been firstly considered for languages that arise in the context of memory checking [8, 12], of databases [28, 27], and later on for formal languages [21, 7]. However, even for simple VPL, any randomized streaming algorithm with p passes requires memory $\Omega(n/p)$, where n is the input size [18].

As opposed to streaming algorithms, (standard) property testers [9, 10, 16] have random access to their input but in the query model. They must query each piece of the input they need to access. They should sample only a sublinear fraction of their input, and ideally make a constant number of queries. In order to make the task of verification possible, decision problems need to be approximated as follows. Given a distance on words, an ε-tester for a language L distinguishes with high probability the words in L from those ε-far from L, using as few queries as possible. Property testing of regular languages was first considered for the Hamming distance [1]. When the distance allows sufficiently modifications of the input, such as moves of arbitrarily large factors, it has been shown that any context-free languages become testable with a constant number of queries [20, 15]. However, for more realistic distances, property testers for simple languages require a large number of queries, especially if they have one-sided error only. For example the complexity of an ε-tester for well-parenthesized expressions with two types of parentheses is between $\Omega(n^{1/11})$ and $O(n^{2/3})$ [25], and it becomes linear, even for one type of parentheses, if we require one-sided error [1]. The difficulty of testing regular tree languages was also addressed when the tester can directly query the tree structure [24].

Faced by the intrinsic hardness of VPL in both streaming and property testing, we initiate the complexity of *streaming property testers* of formal languages, a model of algorithms combining both approaches. Such testers were historically introduced for testing a specific notion (groupedness) [14] relevant for network data.

It was later on studied in the context of testing the insert/extract-sequence of a priority-queue structure [12]. A streaming property tester is a streaming algorithm recognizing a language under the property testing approximation: it must distinguish inputs of the language from those that are ε-far from it, while using the smallest possible memory (rather than limiting its number of input queries). Such an algorithm can simulate any standard non-adaptive property tester. Moreover, we will see that, using its full scan of the input, it can construct better sketches than in the query model.

In this paper, we consider streaming property testing for a natural notion of distance for VPL, the *balanced-edit distance*, which lies between the edit distance and the Hamming distance. It can be interpreted as the edit distance on trees: any neutral symbol can be deleted/inserted, but any push symbol can only be deleted/inserted together with its matching pop symbol (the first pop symbol acting at the same height of the stack than the push symbol). It is not clear for now if our results still apply for the Hamming distance.

In Section 3, we start by the simple case of languages consisting of non-alternating sequences, that is a sequence u_+ of push and neutral symbols followed by a sequence u_- of pop and neutral symbols, with the same number of push and pop symbols. We call *peaks* those well-balanced expressions. The simplicity of those instances will let us highlight our first idea. Moreover, they are already expressive enough in order to demonstrate the superiority of streaming testers against streaming algorithms and property testers, when they are not combined. We first reduce the problem of streaming testing such instances to the problem of testing regular languages in the standard model of property testing. Since our reduction induces weights on the letters of the new input word, we design a new tester for weighted regular languages (**Theorem 3.13**). This tester uses some ideas of the original tester of [1], while considering another distance and new notions in order to take into consideration weights. As a consequence we get a streaming property tester with polylogarithmic memory for recognizing peak instances of any given VPL (**Theorem 3.8**), a task already hard for streaming algorithms and property testers (**Fact 3.1**).

In Section 4, we construct our main tester for a VPL L given by some VPA. We first design an algorithm that maintains a small stack but whose items can be of linear size. Items are prefixes of some peaks, that we call unfinished peaks. They will be later on compressed using a notion of suffix sampling that we introduce for our purpose. Our algorithm is not the standard simulation of a pushdown automaton which usually has a stack of potentially linear size but of constant size items. Indeed, our algorithm compresses an unfinished peak $u = u_+v_-$ when it is followed by a long enough sequence. More precisely, the compression applies to the peak v_+v_- obtained by disregarding part of the prefix of push sequence u_+. Those peaks are then inductively replaced, and therefore compressed, by the state-transition relation they define on the given automaton. The relation is then considered as a single symbol whose weight is the size of the peak it represents. In addition to maintain a stack of logarithmic depth, one of the crucial properties of our algorithm (**Proposition 4.2**) is to rewrite the input word as a peak formed by potentially a linear number of intermediate peaks, but with only a logarithmic numbed of nested peaks.

Next, stack items are replaced by small sketches made of a polylogarithmic number of samples. They are based on a notion of suffix sampling we introduce (**Definition 4.4**). This sampling consists in a decomposition of the string in an increasing sequence of suffixes, whose weights increase geometrically. Such a decomposition can be computed online on a data stream (**Lemma 4.6**), and one can maintain samples in each suffix of the decomposition using a standard reservoir sampling. This suffix decomposition will allow us to simulate an appropriate sampling on the peaks we compress, even if we do not know yet where they start at first. Our sampling can be used to perform an approximate computation of the compressed relation by our new property tester of weighted regular languages that we also use for single peaks. We first establish a result of stability which basically states that we can assume that our algorithm knows in advance where the peak it will compress starts (**Lemma 4.11**). Then we prove the robustness of our algorithm, that is words that are ε-far from L are rejected with high probability (**Lemma 4.14**). As a consequence, we get a one-pass streaming ε-tester for L with one-sided error and memory space $\mathrm{O}(m^5 2^{3m^2}(\log n)^6/\varepsilon^4)$, where m is the number of states of a VPA recognizing L (**Theorem 4.7**).

2 Definitions and Preliminaries

Let $\mathbb{N}^*$ be the set of positive integers, and for any integer $n \in \mathbb{N}^*$, let $[n] = \{1, 2, \ldots, n\}$. A t-subset of a set S is any subset of S of size t. For a finite alphabet Σ we denote the set of finite words over Σ by Σ^*. For a word $u = u(1)u(2)\cdots u(n)$, we call n the *length* of u, and $u(i)$ the ith letter in u. We write $u[i, j]$ for the factor $u(i)u(i+1)\cdots u(j)$ of u. When we mention letters and factors of u we implicitly also mention their positions in u. We say that v is a *sub-factor* of v', denoted $v \le v'$, if $v = u[i, j]$ and $v' = u[i', j']$ with $[i, j] \subseteq [i', j']$. Similarly we say that $v = v'$ if $[i, j] = [i', j']$. If $i \le i' \le j \le j'$ we say that the *overlap* of v and v' is $u[i', j]$. If v is a sub-factor of v' then the overlap of v and v' is v. Given two multisets of factors S ans S', we say that $S \le S'$ if for each factor $v \in S$ there is a corresponding factor $v' \in S'$ such that $v \le v'$.

Weighted Words and Sampling. A *weight function* on a word u with n letters is a function $\lambda : [n] \to \mathbb{N}^*$ on the letters of u, whose value $\lambda(i)$ is called the *weight of* $u(i)$. A *weighted word* over Σ is a pair (u, λ) where $u \in \Sigma^*$ and λ a weight function on u. We define $|u(i)| = \lambda(i)$ and $|u[i, j]| = \lambda(i) + \lambda(i+1) + \ldots + \lambda(j)$. The length of (u, λ) is the length of u. For simplicity, we will denote by u the weighted word (u, λ). Weighted letters will be used to substitute factors of same weights. Therefore, restrictions may exist on available weights for a given letter.

Our algorithms will be based on a sampling of small factors according to their weights. We introduce a very specific notion adapted to our setting. For a weighted word u, we denote by *k-factor sampling on u* the sampling over factors $u[i, i+l]$ with probability $|u(i)|/|u|$, where $l \ge 0$ is the smallest integer such that $|u[i, i+l]| \ge k$ if it exists, otherwise l is such that $i + l$ is the last letter of u. More generally we call k-factor such a factor. For the special case of $k = 1$, we call this sampling a *letter sampling on u*. Observe that both of them can be implemented using a standard reservoir sampling (see Algorithm 1 for letter sampling).

Algorithm 1: Reservoir Sampling

Input: Data stream u, Integer parameter $t > 1$
Data structure:
  $\sigma \leftarrow 0$ // *Current weight of the processed stream*
  $S \leftarrow$ empty multiset // *Multiset of sampled letters*
Code:
$i \leftarrow 1$, $a \leftarrow \mathrm{Next}(u)$, $\sigma \leftarrow |a|$
$S \leftarrow$ t copies of a
While u not finished
  $i{+}{+}$, $a \leftarrow \mathrm{Next}(u)$, $\sigma \leftarrow \sigma + |a|$
  For each $b \in S$
    Replace b by a with probability $|a|/\sigma$
Output S

Even if our algorithm will require several samples from a k-factor sampling, we will often only be able to simulate this sampling by sampling either larger factors, more factors, or both. Let $\mathcal{W}_1$ be a sampler producing a random multiset S_1 of factors of some given weighted word u. Then $\mathcal{W}_2$ *over samples* $\mathcal{W}_1$ if it produces a random multiset S_2 of factors of u such that $\Pr(\mathcal{W}_2 \text{ samples } S_2) \ge \Pr(\mathcal{W}_1 \text{ samples } S_1)$, where each probability term refers to random choices of the corresponding sampler.

Finite State Automata and Visibly Pushdown Automata. A *finite state automaton* is a tuple of the form $\mathcal{A} = (Q, \Sigma, Q_{in}, Q_f, \Delta)$ where Q is a finite set of control states, Σ is a finite input alphabet, $Q_{in} \subseteq Q$ is a subset of initial states, $Q_f \subseteq Q$ is a subset of final states and $\Delta \subseteq Q \times \Sigma \times Q$ is a transition relation. We write $p \xrightarrow{u} q$, to mean that there is a sequence of transitions in $\mathcal{A}$ from p to q while processing u, and we call (p,q) a *u-transitions*. For $\Sigma' \subseteq \Sigma$, the *Σ'-diameter* (or simply *diameter* when $\Sigma' = \Sigma$) of $\mathcal{A}$ is the maximum over all possible pairs $(p,q) \in Q^2$ of $\min\{|u| : p \xrightarrow{u} q \text{ and } u \in \Sigma'^*\}$, whenever this minimum is not over an empty set. We say that $\mathcal{A}$ is *Σ'-closed*, when $p \xrightarrow{u} q$ for some $u \in \Sigma^*$ iff $p \xrightarrow{u'} q$ for some $u' \in \Sigma'^*$.

A *pushdown alphabet* is a triple $\langle \Sigma_+, \Sigma_-, \Sigma_= \rangle$ that comprises three disjoint finite alphabets: Σ_+ is a finite set of *push symbols*, Σ_- is a finite set of *pop symbols*, and $\Sigma_=$ is a finite set of *neutral symbols*. For any such triple, let $\Sigma = \Sigma_+ \cup \Sigma_- \cup \Sigma_=$. Intuitively, a *visibly pushdown automaton* [26] over $\langle \Sigma_+, \Sigma_-, \Sigma_= \rangle$ is a pushdown automaton restricted such that it pushes onto the stack only on reading a push, it pops the stack only on reading a pop, and it does not modify the stack on reading a neutral symbol. Up to coding, this notion is similar to the one of input driven pushdown automata [22] and of nested word automata [6].

Definition 2.1 (Visibly pushdown automaton [26])**.** *A* visibly pushdown automaton *(*VPA*) over $\langle \Sigma_+, \Sigma_-, \Sigma_= \rangle$ is a tuple $\mathcal{A} = (Q, \Sigma, \Gamma, Q_{in}, Q_f, \Delta)$ where Q is a finite set of states, $Q_{in} \subseteq Q$ is a set of initial states, $Q_f \subseteq Q$ is a set of final states, Γ is a finite stack alphabet, and $\Delta \subseteq (Q \times \Sigma_+ \times Q \times \Gamma) \cup (Q \times \Sigma_- \times \Gamma \times Q) \cup (Q \times \Sigma_= \times Q)$ is the transition relation.*

To represent stacks we use a special bottom-of-stack symbol $\perp$ that is not in Γ. A *configuration* of a VPA $\mathcal{A}$ is a pair (σ, q), where $q \in Q$ and $\sigma \in \perp \cdot \Gamma^*$. For $a \in \Sigma$, there is an *a-transition* from a configuration (σ, q) to (σ', q'), denoted $(\sigma, q) \xrightarrow{a} (\sigma', q')$, in the following cases:

- If a is a push symbol, then $\sigma' = \sigma\gamma$ for some $(q, a, q', \gamma) \in \Delta$, and we write $q \xrightarrow{a} (q', \mathsf{push}(\gamma))$.
- If a is a pop symbol, then $\sigma = \sigma'\gamma$ for some $(q, a, \gamma, q') \in \Delta$, and we write $(q, \mathsf{pop}(\gamma)) \xrightarrow{a} q'$.
- If a is a neutral symbol, then $\sigma = \sigma'$ and $(q, a, q') \in \Delta$, and we write $q \xrightarrow{a} q'$.

For a finite word $u = a_1 \cdots a_n \in \Sigma^*$, if $(\sigma_{i-1}, q_{i-1}) \xrightarrow{a_i} (\sigma_i, q_i)$ for every $1 \leq i \leq n$, we also write $(\sigma_0, q_0) \xrightarrow{u} (\sigma_n, q_n)$. The word u is *accepted* by a VPA if there is $(p,q) \in Q_{in} \times Q_f$ such that $(\perp, p) \xrightarrow{u} (\perp, q)$. The language $L(\mathcal{A})$ of $\mathcal{A}$ is the set of words accepted by $\mathcal{A}$, and we refer to such a language as a *visibly pushdown language* (VPL).

At each step, the height of the stack is pre-determined by the prefix of u read so far. The *height* $\mathrm{height}(u)$ of $u \in \Sigma^*$ is the difference between the number of its push symbols and of its pop symbols. A word u is *balanced* if $\mathrm{height}(u) = 0$ and $\mathrm{height}(u[1,i]) \geq 0$ for all i. We also say that a push symbol $u(i)$ *matches* a pop symbol $u(j)$ if $\mathrm{height}(u[i,j]) = 0$ and $\mathrm{height}(u[i,k]) > 0$ for all $i < k < j$.

For all balanced words u, the property $(\sigma, p) \xrightarrow{u} (\sigma, q)$ does not depend on σ, therefore we simply write $p \xrightarrow{u} q$, and say that (p,q) is a *u-transition*. We also define similarly to finite automata the *Σ'-diameter* of $\mathcal{A}$ (or simply diameter) on balanced words only.

Our model is inherently restricted to input words having no prefix of negative stack height, and moreover we have defined acceptance with empty stack. This implies that only balanced words can be accepted. From now on, we will always assume the input is balanced as verifying this in a streaming context is easy.

Balanced/Standard Edit Distance. The usual distance between words in property testing is the Hamming distance. In this work, we consider an easier distance to manipulate in property testing but still relevant for most applications, which is the edit distance, that we adapt for weighted words.

Given any word u, we define two possible *edit operations*: a *deletion* of a letter in position i with corresponding cost $|u(i)|$, and its converse operation the *insertion* where we also select a weight, compatible with the restrictions on λ, for the new $u(i)$. Then the *(standard) edit distance* $\mathsf{dist}(u,v)$ between two weighted words u and v is simply defined as the minimum total cost of a sequence of edit operations changing u to v. Note that all letters that have not been inserted or deleted must keep the same weight. For a restricted set of letters Σ', we also define $\mathsf{dist}_{\Sigma'}(u,v)$ where the insertions are restricted to letters in Σ'.

We will also consider a restricted version of this distance for balanced words, motivated by our study of VPL. Similarly, *balanced-edit operations* can be deletions or insertions of letters, but each deletion of a push symbol (resp. pop symbol) requires the deletion of the matching pop symbol (resp. push symbol). Similarly for insertions: if a push (resp. pop) symbol is inserted, then the matching pop (resp. push) symbol must also be inserted simultaneously. The cost of these operations is the weight of the affected letters, as with the edit operations. Again, only insertions of letters with weight 1 are allowed. We define the *balanced-edit distance* $\mathsf{bdist}(u, v)$ between two balanced words as the total cost of a sequence of balanced-edit operations changing u to v. Similarly to $\mathsf{dist}_{\Sigma'}(u, v)$ we define $\mathsf{bdist}_{\Sigma'}(u, v)$.

When dealing with a visibly pushdown language, we will always use the balanced-edit distance, whereas we will use the standard-edit distance for regular languages. We also say that u is (ε, Σ')*-far* from v if $\mathsf{dist}_{\Sigma'}(u, v) > \varepsilon|u|$, or $\mathsf{bdist}_{\Sigma'}(u, v) > \varepsilon|u|$, depending on the context. We omit Σ' when $\Sigma' = \Sigma$.

Streaming Property Testers. An ε-tester for a language L accepts all inputs which belong to L with probability 1 and rejects with high probability all inputs which are ε-far from L, *i.e.* that are ε-far from any element of L. Two-sided error testers have also been studied but in this paper we stay with the notion of one-sided testers, that we adapt in the context of streaming algorithm as in [14].

Definition 2.2 (Streaming property tester)**.** *Let $\varepsilon > 0$ and let L be a language. A* streaming ε-tester *for L with one-sided error η and memory $s(n)$ is a randomized algorithm A such that, for any input x of length n given as a data stream:*

- *If $u \in L$, then A accepts with probability 1;*
- *If u is ε-far from L, then A rejects with probability at least $1 - \eta$;*
- *A processes u within a single sequential pass while maintaining a memory space of $O(s(n))$ bits.*

3 Simple case

3.1 Non-Alternating Sequences

We first consider restricted instances consisting only of a *peak*, that is sequences of push symbols followed by a sequence of pop symbols, with possibly intermediate neutral symbols, that is elements of the language $\Lambda = \bigcup_{j \geq 0} ((\Sigma_=)^* \cdot \Sigma_+)^j \cdot (\Sigma_=)^* \cdot (\Sigma_- \cdot (\Sigma_=)^*)^j$.

Those instances are already hard for both streaming algorithms and property testing algorithms. Indeed, consider the language $\mathrm{Disj} \subseteq \Lambda$ over alphabet $\Sigma = \{0, 1, \overline{0}, \overline{1}, a\}$ and defined by the union of all languages $a^* \cdot x(1) \cdot a^* \cdot \ldots \cdot x(j) \cdot a^* \cdot \overline{y(j)} \cdot a^* \cdot \ldots \cdot \overline{y(1)} \cdot a^*$, where $j \geq 1$, $x, y \in \{0, 1\}^j$, and $x(i)y(i) \neq 1$ for all i.

Then Disj can be recognized by a VPA with 3 states, $\Sigma_+ = \{0, 1\}$, $\Sigma_- = \{\overline{0}, \overline{1}\}$ and $\Sigma_= = \{a\}$. However, the following fact states its hardness for both models. The hardness for streaming algorithms (without any notion of approximation) comes from a standard reduction to a communication complexity problem known as Set-Disjointness, and remains valid for *p-pass* streaming algorithms, that is streaming algorithms that are allowed to make up to p sequential passes (in any direction) on the input stream. The hardness for property testing algorithms (that have only access to the input via queries) comes from a similar result due to [25] for parenthesis languages with two types of parenthesis, and for the Hamming distance. The result remains valid for both our language and the balanced-edit distance.

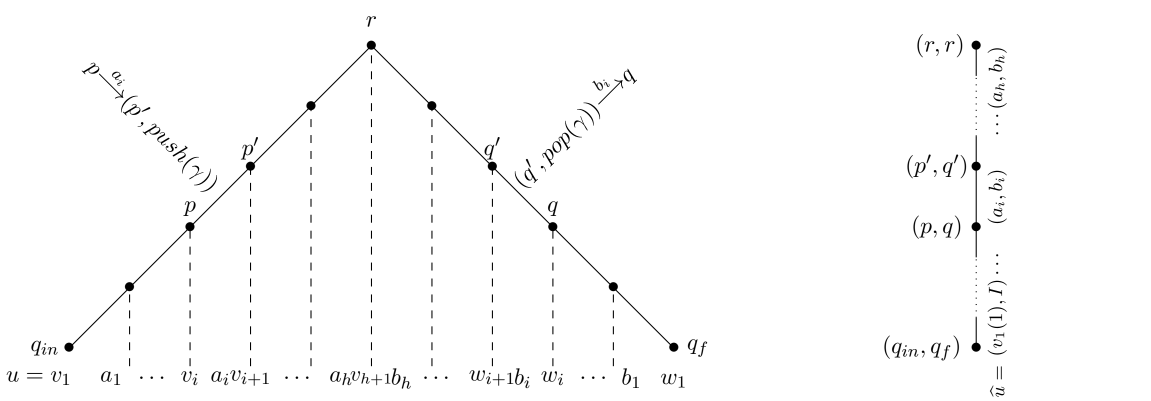
Fact 3.1. *Any randomized p-pass streaming algorithm for* Disj *requires memory space $\Omega(n/p)$, where n is the input length. Moreover, any (non-streaming) (2^{-6})-tester for* Disj *requires to query $\Omega(n^{1/11}/\log n)$ letters of the input word.*

Surprisingly, for every $\varepsilon > 0$, such languages (actually any language of the form $L \cap \Lambda$ where L is a VPL) become easy to ε-test by streaming algorithms. This is mainly because, given their full access to the input, streaming algorithms can perform an input sampling which makes the property testing task easy, using only a single pass and few memory.

We first show that, for every VPL L, one can construct a regular language $\widehat{L}$ such that testing whether $u \in L \cap \Lambda$ is equivalent to test whether some other word $\widehat{u}$ belongs to $\widehat{L}$. For this, let I be a special symbol not in $\Sigma_=$. Consider a word $u = \left(\prod_{i=1}^{j} v_i \cdot a_i\right) \cdot v_{j+1} \cdot \left(\prod_{i=j}^{1} \overline{b_i} \cdot w_i\right)$, where $a_i \in \Sigma_+$, $\overline{b_i} \in \Sigma_-$, and $v_i, w_i \in (\Sigma_=)^*$. Define the *slicing* of u (see Figure 1) as the word $\widehat{u}$ over the alphabet $\widehat{\Sigma} = (\Sigma_+ \times \Sigma_-) \cup (\Sigma_= \times \{\mathrm{I}\}) \cup (\{\mathrm{I}\} \times \Sigma_=)$ defined by $\widehat{u} = \left(\prod_{i=1}^{j} (v_i(1), \mathrm{I}) \cdots (v_i(|v_i|), \mathrm{I}) \cdot (\mathrm{I}, w_i(1)) \ldots (\mathrm{I}, w_i(|w_i|)) \cdot (a_i, \overline{b_i})\right) \cdot (v_{j+1}(1), \mathrm{I}) \cdots (v_{j+1}(|v_{j+1}|), \mathrm{I})$.

Definition 3.2. *Let $\mathcal{A} = (Q, \Sigma, \Gamma, Q_{in}, Q_f, \Delta)$ be a* VPA. *The* slicing *of $\mathcal{A}$ is the finite automaton $\widehat{\mathcal{A}} = (\widehat{Q}, \widehat{\Sigma}, \widehat{Q_{in}}, \widehat{Q_f}, \widehat{\Delta})$ where $\widehat{Q} = Q \times Q$, $\widehat{Q_{in}} = Q_{in} \times Q_f$, $\widehat{Q_f} = \{(p,p) : p \in Q\}$, and the transitions $\widehat{\Delta}$ are:*

1. $(p,q) \xrightarrow{(a,b)} (p',q')$ *when* $p \xrightarrow{a} (p', \mathsf{push}(\gamma))$ *and* $(q', \mathsf{pop}(\gamma)) \xrightarrow{b} q$ *are both transitions of* Δ.
2. $(p,q) \xrightarrow{(c,\mathrm{I})} (p',q)$, *resp.* $(p,q) \xrightarrow{(\mathrm{I},c)} (p,q')$, *when* $p \xrightarrow{c} p'$, *resp.* $q \xrightarrow{c} q'$, *is a transition of* Δ.



Run in the VPA $\mathcal{A}$ on u Run in the slicing automaton $\widehat{\mathcal{A}}$ on $\widehat{u}$

Figure 1: Slicing of a word $u \in \Lambda$ and evolution of the stack height for u.

Lemma 3.3. *If $\mathcal{A}$ is a* VPA *accepting L, then $\widehat{\mathcal{A}}$ is a finite automaton accepting $\widehat{L} = \{\widehat{u} : u \in L \cap \Lambda\}$.*

Proof. Because transitions on push symbols do not depend on the top of the stack, transitions in $\widehat{\Delta}$ correspond to slices that are valid for Δ (see Figure 1). Finally, $\widehat{Q_{in}}$ ensures that a run for L must start in Q_{in} and end in Q_f, and $\widehat{Q_f}$ that a state at the top of the peak is consistent from both sides. □

Regular languages are known to be ε-testable for the Hamming distance with $\mathrm{O}((\log 1/\varepsilon)/\varepsilon)$ non-adaptative queries on the input word [1], that is queries that can be all made simultaneously. Since Hamming distance is larger than the edit distance, those testers are also valid for the later distance. Observe also that, for $u, v \in \Lambda$, we have $\mathsf{bdist}(u,v) \leq 2\mathsf{dist}(\widehat{u}, \widehat{v})$. Those samples can be understood as a random sketch. To adapt this to a streaming algorithm for testing whether $u \in L \cap \Lambda$, we need to build an appropriate sampling procedure on u. We first do it for the simple case where $\Sigma_= = \emptyset$.

Corollary 3.4. *Let $\mathcal{A}$ be a* VPA *for L with $\Sigma_= = \emptyset$ and let $\varepsilon > 0$. There is a streaming ε-tester for $L \cap \Lambda$ with constant one-sided error with memory space $\mathrm{O}((c \log n)(\log 1/\varepsilon)/\varepsilon)$, where n is the input length and $c > 0$ depends only on $\mathcal{A}$.*

Proof. The tester of [1] samples uniformly at random several factors of the input word of several given lengths and it is still correct if it takes an over-sampling. Those samples on $\widehat{u}$ can be done in two steps. We describe it for a single factor of length k. Let u_+ be the prefix of u before its first pop symbol, and let u_- be the remaining suffix including the first pop symbol. First we sample uniformly a random position in u_+ and remember its position, which requires $\mathrm{O}(\log n)$ memory, and the following k letters in u_+. This sampling can be done without knowing the length of u_+ in advance, using standard reservoir sampling techniques. Second, we complete the factor while reading u_-. That way, we simply have more letters than needed in the sampled factor. □

We could directly generalize the previous algorithm when $\Sigma_= \neq \emptyset$ by slightly modifying our sampling procedure. However, we prefer to take a different approach enlightening the main idea of our general algorithm in Section 4. Given any maximal factor $v \in (\Sigma_=)^*$ (for the sub-factor relation $\leq$) of the input stream, we will consider it as a single letter of weight $|v|$. More precisely, fix a VPA $\mathcal{A}$ recognizing L. Then, we compress v by its corresponding relation $R_v = \{(p,q) : p \xrightarrow{v} q\}$, and we see the subset $R_v \subseteq Q \times Q$ as a new letter, call it R, and the possible weights for R correspond to the weights of words v such that $R = R_v$. We augment $\Sigma_=$ by those new letters, and call this new (finite) alphabet Σ_0.

We also extend the automaton $\mathcal{A}$ and the language L with Σ_0. Doing so, we have compressed $u \in \Lambda$ to a weighted word of $\Lambda_1 = \bigcup_{j \geq 1} (\Sigma_0 \cdot \Sigma_+)^j \cdot \Sigma_0 \cdot (\Sigma_- \cdot \Sigma_0)^j$. Since there is a correspondence between letters $R \in \Sigma_0$ and words $v \in (\Sigma_=)^*$ with $|v| = |R|$ and $R = R_v$, we can arbitrarily reason on either the old or the new alphabet. Moreover, the corresponding slicing automaton $\widehat{\mathcal{A}}$ still has diameter at most $2m^2$.

Proposition 3.5. *Let $v \in \Lambda_1$ be s.t. $(p,q) \xrightarrow{\widehat{v}} (p',q')$. There is $w \in \Lambda_1$ s.t. $|w| \leq 2m^2$ and $(p,q) \xrightarrow{\widehat{w}} (p',q')$.*

We are now ready to build a tester for $L \cap \Lambda$ using the same idea as in Corollary 3.4: to test a word u we use a tester for $\widehat{u}$ against $\widehat{L}$, which is now a language of weighted words. More precisely, the weight of a letter in $\widehat{u}$ is defined by $|(a_+, a_-)| = 1$ and $|(\mathrm{I}, R)| = |(R, \mathrm{I})| = |R|$. In Section 3.2 we construct such a tester. The remaining difficulty is to provide to this tester an appropriate sampling on $\widehat{u}$ while processing u.

Our tester for weighted regular languages is based on k-factor sampling on $\widehat{u}$ that we will simulate by an over-sampling built from a letter sampling on u, that is according to the weights of the letters of u only. This new sampling can be easily performed given a stream of u using a standard reservoir sampling.

Definition 3.6. *For a weighted word $u \in \Lambda$, denote by $\mathcal{W}_k(u)$ the sampling over factors of $\widehat{u}$ constructed as follows: (1) sample a letter $u(i)$ of u with probability $|u(i)|/|u|$; (2) if $u(i)$ is in a push sequence, extends it to the factor $u[i, i+l+1]$ where $u[i, i+l]$ is a k-factor, and complete it with its matching pop sequence.*

Lemma 3.7. *Let u be a weighted word, and let k be such that $4k \leq |u|$. Then $4k$ independent copies of $\mathcal{W}_k(u)$ over samples the k-factor sampling on $\widehat{u}$.*

Proof. Denote by $\widehat{\mathcal{W}}$ the k-factor sampling on $\widehat{u}$, and by $\mathcal{W}$ some $4k$ independent copies of $\mathcal{W}_k(u)$. For any k-factor v of $\widehat{u}$, we will show that the probability that v is sampled by $\widehat{\mathcal{W}}$ is at most the probability that v is a factor of an element sampled by $\mathcal{W}$. For that, we distinguish the following three cases:

- v is a single letter. Then, if $v = (R, I)$ the probability that it is sampled by $\widehat{\mathcal{W}}$ equals the probability that $\mathcal{W}_k(u)$ samples the factor v augmented by one letter; if $v = (I, R)$ the probability that it is sampled by $\mathcal{W}$ again equals the probability that $\mathcal{W}_k(u)$ samples it. Hence, the probability that v is sampled by $\widehat{\mathcal{W}}$ is at most the probability that v is a factor of an element sampled by $\mathcal{W}$.

- v is not a single letter and starts by a letter in $\Sigma_+ \times \Sigma_-$ or by a letter in $\Sigma_0 \times \{I\}$. Then the probability that it is sampled by $\widehat{\mathcal{W}}$ equals at most twice the probability that $\mathcal{W}_k(u)$ samples the factor v augmented by one letter, as a (push,pop) pair in $\widehat{u}$ has weight 2 when a push has weight 1 in u. Hence, the probability that v is sampled by $\widehat{\mathcal{W}}$ is at most the probability that v is a factor of an element sampled by $\mathcal{W}$.
- v is not a single letter and starts by a letter in $\Sigma_0 \times \{I\}$. Since $|\widehat{u}| \geq |u|/2$, we get

$$\Pr(\mathcal{W}_k(u) \text{ samples the factor } (a,b)\cdot v) = 1/|u| \quad \text{and} \quad \Pr(\widehat{\mathcal{W}} \text{ samples } v) \leq k/|\widehat{u}| \leq 2k/|u|.$$

  Thus the probability that one of the $4k$ samples of $\mathcal{W}$ has the factor $(a,b)\cdot v$ is $1-(1-1/|u|)^{4k}$. As $1-(1-1/|u|)^{4k} \geq 1 - \frac{1}{1+4k/|u|} = \frac{4k}{|u|+4k} \geq 2k/|u|$ when $|u| \geq 4k$, we conclude again that the probability that v is sampled by $\widehat{\mathcal{W}}$ is at most the probability that v is a factor of an element sampled by $\mathcal{W}$.

$\square$

Theorem 3.8. *Let $\mathcal{A}$ be a* VPA *for L with $m \geq 2$ states, and let $\varepsilon, \eta > 0$. Then there is a streaming ε-tester for $L \cap \Lambda$ with one-sided error η and memory space $\mathrm{O}(m^8(\log 1/\eta)/\varepsilon^2)$, where n is the input length.*

Proof. The proof uses Theorem 3.13 for weighted regular languages. Observe that $\mathsf{bdist}(u,v) \leq 2\mathsf{dist}(\widehat{u},\widehat{v})$, and moreover the slicing automaton has diameter d at most $2m^2$. Given a word u as a data stream, we simulate a data stream on its compression u_1, which is a weighted word in Λ_1, and then obtain with Lemma 3.7 an over-sampling of t k-factor samplings on $\widehat{u_1}$, with $t = 4\lceil 4dm^3(\log 1/\eta)/\varepsilon\rceil$ and $k = \lceil 4dm/\varepsilon\rceil$. $\square$

3.2 Testing Weighted Regular Languages

We first design a non-adaptative property tester for weighted regular languages that will serve as a basic routine of our more general algorithm. Property testing of regular languages was first considered in [1] for the Hamming distance and we adapt this tester to weighted words for the simple case of edit distance. In particular following [1] we consider the graph of components of the automaton and focus on paths in this graph; however note that we introduce a new criterion, κ-saturation (for some parameter $0 < \kappa \leq 1$), that permits to significantly simplify the correctness proof of the tester compared to the one in [1].

For the rest of this section, fix a regular language L recognized by some finite state automaton $\mathcal{A}$ on Σ with a set of states Q of size $m \geq 2$, and a diameter $d \geq 2$. Define the directed graph $G_\mathcal{A}$ on vertex set Q whose edges are pairs (p,q) when $p \xrightarrow{a} q$ for some $a \in \Sigma$.

A *component* C of $G_\mathcal{A}$ is a maximal subset (w.r.t. inclusion) of vertices of $G_\mathcal{A}$ such that for every p_1, p_2 in C one has a path in $G_\mathcal{A}$ from p_1 to p_2. The *graph of components* $\mathcal{G}_\mathcal{A}$ of $G_\mathcal{A}$ describes the transition relation of $\mathcal{A}$ on components of $G_\mathcal{A}$: its vertices are the components and there is a directed edge (C_1, C_2) if there is an edge of $G_\mathcal{A}$ from a vertex in C_1 toward a vertex in C_2.

Definition 3.9. *Let C be a component of $G_\mathcal{A}$, let $\Pi = (C_1, \ldots, C_l)$ be a path in $\mathcal{G}_\mathcal{A}$.*
- *A word u is C-*compatible *if there are states $p, q \in C$ such that $p \xrightarrow{u} q$.*
- *A word u is Π-*compatible *if u can be partitioned into $u = v_1 a_1 v_2 \ldots a_{l-1} v_l$ such that $p_i \xrightarrow{v_i} q_i$ and $q_i \xrightarrow{a_i} p_{i+1}$, where v_i is a factor, a_i a letter, and $p_i, q_i \in C_i$.*
- *A sequence of factors $(v_1, \ldots, v_t)$ of a word u is Π-*compatible *if they are factors of another Π-*compatible *word with the same relative order and same overlap.*

Note that the above properties are easy to check. Indeed, C-compatibility is a reachability property while the two others easily follow from C-compatibility checking.

We now give a criterion that characterizes those words u that are ε-far to every Π-compatible word. Note that it will not be used in the tester that we design in Theorem 3.13 for weighted regular languages, but only in Lemma 3.11 which is the key tool to prove its correctness.

For a component C and a C-incompatible word v, let $v_1 \cdot a$ be the shortest C-incompatible prefix of v. We define and denote the *C-cut* of v as $v = v_1 \cdot a \cdot v_2$. When v_1 is not the empty word, we say that v_1 is a *C-factor* and a is a *C-separator for* v_1, otherwise we say that a is a *strong C-separator*.

Fix a path $\Pi = (C_1, \dots, C_l)$ in $\mathcal{G}_\mathcal{A}$, a parameter $0 < \kappa \leq 1$, and consider a weighted word u. We define a natural partition of u according to Π, that we call the *Π-partition* of u. For this, start with the first component $C = C_1$, and consider the C_1-cut $u_1 \cdot a \cdot u_2$ of u. Next, we inductively continue this process with either the suffix $a \cdot u_2$ if a is a C_1-separator, or the suffix u_2 if a is a strong C_1-separator. Based on some criterion defined below we will move from the current component C_i to a next component C_j of Π, where most often $j = i + 1$, until the full word u is processed. If we reach $j = l + 1$, we say that u *κ-saturates* Π and the process stops. We now explain how we move on in Π. We stay within C_i as long as both the number of C_i-factors and the total weight of strong C_i-separators are at most $\kappa|u|$ each. Then, we continue the decomposition with some fresh counting and using a new component C_j selected as follows. One sets $j = i + 1$ except when the transition is the consequence of a strong C_i-separator a of weight greater than $\kappa|u|$, that we call a *heavy strong separator*. In that case only, one lets $j \geq i + 1$, if exists, to be the minimal integer such that $q \xrightarrow{a} q'$ with $q \in C_{j-1} \cup C_j$ and $q' \in C_j$, and $j = l + 1$ otherwise.

Proposition 3.10. *Let $0 < \kappa \leq \varepsilon/(2dl)$. If u is ε-far to every Π-compatible word, then u κ-saturates Π.*

Proof. The proof is by contraposition. For this we assume that u does not κ-saturate Π and we correct u to a Π-compatible word as follows.

First, we delete each strong separator of weight less that $\kappa|u|$. Their total weight is at most $2l\kappa|u|$. Because u does not saturate, each strong separator of weight larger than $\kappa|u|$ fits in the Π-partition, and does not need to be deleted.

We now have a sequence of consecutive C_i-factors and of heavy strong C_i-separators, for some $1 \leq i \leq l$, in an order compatible with Π. However, the word is not yet compatible with Π since each factor may end with a state different than the first state of the next factor. However, for each such pair there is a path connecting them. We can therefore bridge all factors by inserting a factor of weight at most d, the diameter of $\mathcal{A}$.

The resulting word is then Π-compatible by construction, and the total cost of the edit operations is at most $(2l + dl)\kappa|u| \leq \varepsilon|u|$, since $d \geq 2$. $\square$

For a weighted word u, we remind that the k-factor sampling on u is defined in Section 2. The following lemma is the key lemma for the tester for weighted regular languages.

Lemma 3.11. *Let u be a weighted word, let $\Pi = C_1 \dots C_l$ be a path in $\mathcal{G}_\mathcal{A}$. Let $0 < \kappa \leq \varepsilon/(2dl)$ and let $\mathcal{W}$ denote the $\lceil 2/\kappa \rceil$-factor sampling on u. Then for every $0 < \eta < 1$ and $t \geq 2l(\log 1/\eta)/\kappa$, the probability $P(u, \Pi) = \Pr_{(v_1,\dots,v_t)\sim\mathcal{W}^{\otimes t}}[(v_1, \dots, v_t)$ is Π-compatible$]$ satisfies $P(u, \Pi) = 1$ when u is Π-compatible, and $P(u, \Pi) \leq \eta$ when u is ε-far for from being Π-compatible.*

Proof. The first part of the theorem is immediate. For the second part, assume that u is ε-far from any Π-compatible word. For simplicity we assume that $2/\kappa$ and $\kappa|u|/2$ are integers. We first partition u according to Π and κ. Then, Proposition 3.10 tells us that u κ-saturates Π. For each C_i, we have three possible cases.

1. There are $\kappa|u|$ disjoint C_i-factors in u. Since they have total weight at most $|u|$, there are at least $\kappa|u|/2$ of them whose weight is at most $2/\kappa$ each. Since each letter has weight at least 1, the total weight of the first letters of each of those factors is at least $\kappa|u|/2$. Therefore one of them together with its C_i-separator is a sub-factor of some sampled factor v_j with probability at least $1-(1-\kappa/2)^t$.
2. The total weight of strong C_i-separators of u is at least $\kappa|u|$. Therefore one of them is the first letter of some sampled factor v_j with probability at least $1-(1-\kappa)^t$.
3. There is not any C_i-factor and any C_i-separator of u, because of a strong $C_{i'}$-separator of weight greater than $\kappa|u|$, for some $i' < i$. This separator is the first letter of some sampled factor v_j with probability at least $1-(1-\kappa)^t$.

By union bound, the probability that one of the above mentioned samples fails to occurs is at most $l(1-\kappa)^t \leq \eta$. We assume now that they all occur, and we show that they form a Π-incompatible sequence. For each i, let w_i be the above described sub-factors of those samples. Each w_i appears in u after w_{i-1} or, in the case of a strong separator of heavy weight, $w_i = w_{i-1}$. Moreover each factor w_i which is distinct from w_{i-1} forces next factors to start from some component $C_{i'}$ with $i' > i$. As a result $(w_1, \ldots, w_l)$ is not Π-compatible, and as a consequence $(v_1, \ldots, v_t)$ neither, so the result. $\square$

Lemma 3.11 permits to design a non-adaptative tester for L and also to approximate the action of u on $\mathcal{A}$ as follows.

Definition 3.12. *Let $\Sigma' \subseteq \Sigma$ and $R \subseteq Q \times Q$. Then R (ε, Σ')*-approximates a word u on $\mathcal{A}$ (or simply ε-approximates when $\Sigma' = \Sigma$)*, if for all $p, q \in Q$: (1) $(p,q) \in R$ when $p \xrightarrow{u} q$; (2) u is (ε, Σ')-close to some word v satisfying $p \xrightarrow{v} q$ when $(p,q) \in R$.*

Theorem 3.13. *Let $\mathcal{A}$ be an automaton with $m \geq 2$ states and diameter $d \geq 2$. There is an algorithm that:*
1. *Takes as input $\varepsilon > 0$, $\eta > 0$ and t factors of $v_1, \ldots, v_t$ of some weighted word u, such that $t \geq 2\lceil 2dm^3(\log 1/\eta)/\varepsilon \rceil$;*
2. *Outputs a set $R \subseteq Q \times Q$ that ε-approximates u on $\mathcal{A}$ with one-sided error η, when each factor v_i comes from an independent k-factor sampling on u with $k \geq \lceil 2dm/\varepsilon \rceil$.*

This is still true with any combination of the following generalization:
- *The algorithm is given an over-sampling of each of factors v_i instead.*
- *When $\mathcal{A}$ is Σ'-closed, and d is the Σ'-diameter of $\mathcal{A}$, then R also (ε, Σ')-approximates u on $\mathcal{A}$.*

Proof. The algorithm is very simple:

1. Set $R = \emptyset$
2. For all states $p, q \in Q$
   (a) Check if factors $v_1, \ldots, v_t$ could come from a word v such that $p \xrightarrow{v} q$
   // Step (a) is done using the graph $\mathcal{G}_\mathcal{A}$ of connected components of $\mathcal{A}$
   (b) If yes, then add (p,q) to R
3. Return R

It is clear that this R contains every (p,q) such that $p \xrightarrow{u} q$. Now for the converse, we will show that, with bounded error η, the output set R only contains pairs (p,q) such that there exists a path $\Pi = C_1, \ldots, C_l$ on $\mathcal{G}_\mathcal{A}$ such that $p \in C_1$, $q \in C_l$, and u is Π-compatible. In that case, there is an ε-close word v satisfying $p \xrightarrow{v} q$.

Indeed, using $l \leq m$ and Lemma 3.11 with t, $\kappa = \varepsilon/(2dm)$ and $\eta' = \eta/2^m$, the samples satisfy $P(u, \Pi) \leq \eta/2^m$, when u is not Π-compatible. Therefore, we can conclude using a union bound argument on all possible paths on $\mathcal{G}_{\mathcal{A}}$, which have cardinality at most 2^m, that, with probability at least $1-\eta$, there is no Π such that the samples are Π-compatible but u is not Π-compatible.

The structure of the tester is such that it has only more chances to reject a word that is not Π-compatible given an over-sampling as input instead. Words u such that $p \xrightarrow{u} q$ will always be accepted no matter the amount and length of samples. Therefore the theorem still holds with an over sampling.

Last, $\mathcal{A}$ being Σ'-closed ensures that the notions of compatibility and saturations remain unchanged. Using the Σ'-diameter in Lemma 3.11 (and therefore in Proposition 3.10) let us use bridges in Σ'^* instead of Σ^* with weight at most d. □

4 General case

4.1 Exact Algorithm

Fix a VPA $\mathcal{A}$ recognizing some VPL L. A general balanced input instance u will have more than one peak $v \in \Lambda$ and we cannot easily interpret u as an element of a regular language. However, we will recursively replace each factor $v \in \Lambda$ by $R_v = \{(p,q) : p \xrightarrow{v} q\}$ with weight $|v|$. The alphabet $\Sigma_=$ of neutral symbols will increase as follows. We start with Σ_0 encoding all possible relations R_v for $v \in \Sigma_=^*$. Then Λ_{h+1} is simply Λ over an alphabet $\Sigma_= = \Sigma_h$, and Σ_h encodes all possible relations R_v for words $v \in (\Lambda_h)^*$. As before, we naturally augment the automaton $\mathcal{A}$ and the language L with these new sets. However we keep the notation Σ as $\Sigma_+ \cup \Sigma_- \cup \Sigma_=$.

Since there is a finite set of possible relations, this construction has smallest fixed points Σ_∞ and Λ_∞. Denote by $\mathrm{Prefix}(\Lambda_\infty)$ the language of prefixes of words in Λ_∞. For $\Sigma' = (\Sigma_+ \cup \Sigma_- \cup \Sigma_\infty)$, the $\widehat{\Sigma'}$-diameter of the slicing automaton $\widehat{A}$ is simply the Σ-diameter of $\mathcal{A}$, that we bound as follows. For simpler languages, as those coming from DTD, this bound can be lowered to m.

Fact 4.1. *Let $\mathcal{A}$ be a* VPA *with m states. Then the Σ-diameter of $\mathcal{A}$ is at most 2^{m^2}.*

Proof. A similar statement is well known for any context-free grammar given in Chomsky normal form. Let N be the number of non-terminal symbols used in the grammar. If the grammar produces one balanced word from some non-terminal symbol, then it can also produce one whose length is at most 2^N from the same non-terminal symbol. This is proved using a pumping argument on the derivation tree. We refer the reader to the textbook [17].

Now, in the setting of visibly pushdown languages one needs to transform $\mathcal{A}$ into a context-free grammar in Chomsky normal form. For that, consider first an intermediate grammar whose non-terminal symbols are all the X_{pq} where p and q are states from $\mathcal{A}$: such a non-terminal symbol will produce exactly those words u such that $p \xrightarrow{u} q$, hence our initial symbol will be those of the form $X_{q_0 q_f}$ where q_0 is an initial state and q_f is a final state. The rewriting rules are the following ones:

- $X_{pp} \to \varepsilon$
- $X_{pq} \to X_{pr} X_{rq}$ for any state r
- $X_{pq} \to a X_{p'q'} b$ whenever one has in the automaton $p \xrightarrow{a} (p', \mathsf{push}(\gamma))$ and $(q', \mathsf{pop}(\gamma)) \xrightarrow{a} q$ for some push symbol a, pop symbol b and stack letter γ.
- $X_{pq} \to a X_{p'q}$ whenever one has in the automaton $p \xrightarrow{a} p'$ for some neutral symbol a.
- $X_{pq} \to X_{pq'} a$ whenever one has in the automaton $q' \xrightarrow{a} q$ for some neutral symbol a.

Obviously, this grammar generates language $L(\mathcal{A})$.

As we are here interested only in the length of the balanced words produced by the grammar, we can replace any terminal symbol by a dummy symbol $\sharp$. Now, once this is done we can put the grammar into

Algorithm 2: Exact Tester for a VPL

```
1  Data structure:
2     Stack ← empty stack // Stack of items $v$ with $v \in \mathrm{Prefix}(\Lambda_\infty)$
3     $u_0 \leftarrow \emptyset$ // $u_0 \in \mathrm{Prefix}(\Lambda_\infty)$ is a suffix of the processed part $u[1,i]$ of $u$
4           // with possibly some factors $v \in \Lambda_\infty$ replaced by $R_v$
5     $R_{\mathtt{temp}} \leftarrow \{(p,p)\}_{p\in Q}$ // Set of transitions for the maximal prefix of $u[1,i]$ in $\Lambda_\infty$
6     Define $u \bullet v$ as the usual concatenation $u \cdot v$ where the last letter $u_l$ of $u$
7       and the first letter $v_f$ of $v$ are replaced by $R_{u_l v_f}$ when $u_l, v_f \in \Sigma_\infty$
8     // For the analysis only, define dynamically Depth on each letter and factor
9     // by $\mathrm{Depth}(a) = 0$ when $a \in \Sigma$, $\mathrm{Depth}(R_v) = \mathrm{Depth}(v) + 1$ where $\mathrm{Depth}(v) = \max_i \mathrm{Depth}(v(i))$
10 Code:
11 While $u$ not finished
12     $a \leftarrow \mathrm{Next}(u)$ //Read and process a new symbol $a$
13     If $a \in \Sigma_+$ and $u_0$ has a letter in $\Sigma_-$ // $u_0 \cdot a \notin \mathrm{Prefix}(\Lambda_k)$
14        Push $u_0$ on Stack, $u_0 \leftarrow a$
15     Else $u_0 \leftarrow u_0 \bullet a$
16     If $u_0$ is well-balanced // $u_0 \in \Lambda_\infty$: compression
17        Compute $R_{u_0}$ the sent of $u_0$-transitions
18        If $Stack = \emptyset$, then $R_{\mathtt{temp}} \leftarrow R_{\mathtt{temp}} \bullet R_{u_0}$, $u_0 \leftarrow \emptyset$
19        Else Pop $v$ from Stack, $u_0 \leftarrow v \bullet R_{u_0}$
20     Let $(v_1 \cdot v_2) = \mathrm{top}(Stack)$ s.t. $v_2$ is maximal and well-balanced // $v_2 \in \Lambda_\infty$
21     If $|u_0| \geq |v_2|/2$  // $u_0$ is big enough and $v_2$ can be replaced by $R_{v_2}$
22        Compute $R_{v_2}$ the set of $v_2$-transitions, Pop $v$ from Stack, $u_0 \leftarrow v_1 \cdot (R_{v_2} \cdot u_0)$
23 If $(Q_{in} \times Q_f) \cap R_{\mathtt{temp}} \neq \emptyset$, Accept; Else Reject // $u = u_0$ and $R_{\mathtt{temp}} = R_u$
```

Chomsky normal form by using an extra non-terminal symbol (call it $X_\sharp$ as it is used to produce the $\sharp$ terminal). As we have m^2+1 non-terminal in the resulting grammar we are almost done. To get to the tight bound announced in the statement, one simply removes the extra non-terminal symbol $X_\sharp$ and reasons on the length of the derivation directly. $\square$

We start by a simple algorithm maintaining a stack of small height, but whose elements can be of linear size. We will later explain how to replace the stack elements by appropriated small sketchs. While having processed the prefix $u[1,i]$ of the data stream u, Algorithm 2 maintains a suffix $u_0 \in \mathrm{Prefix}(\Lambda_\infty)$ of $u[1,i]$, that is an unfinished peak, with some simplifications of factors v in Λ_∞ by their corresponding relation R_v. Therefore u_0 consists of a sequence of push symbols and neutral symbols possibly followed by a sequence of pop symbols and neutral symbols. The algorithm also maintains a subset $R_{\mathtt{temp}} \subseteq Q \times Q$ that is the set of transitions for the maximal prefix of $u[1,i]$ in Λ_∞. When the stream is over, the set $R_{\mathtt{temp}}$ is used to decide whether $u \in L$ or not.

When a push symbol a comes after the pop sequence, u_0 is no longer in $\mathrm{Prefix}(\Lambda_\infty)$, and Algorithm 2 puts it on a stack of unfinished peaks (see lines 13 to 14 and Figure 2a) and u_0 is reset to a. In other situations, one adds a to u_0. In case u_0 becomes a word of Λ_∞ (see lines 16 to 19 and Figure 2b), Algorithm 2 computes the set of u_0-transitions $R_{u_0} \in \Sigma_\infty$, and adds R_{u_0} to the previous unfinished peak, which is found on top of the stack and now becomes the current unfinished peak; in the special case where the stack is empty one simply updates the set $R_{\mathtt{temp}}$ by taking its composition with R_{u_0}.

In order to bound the size of the stack, Algorithm 2 considers the maximal well-balanced suffix v_2 of the topmost element $v_1 \cdot v_2$ of the stack and, when $|u_0| \geq |v_2|/2$, it computes the relation R_{v_2} and continues with a bigger current peak starting with v_1 (see lines 20 to 22 and Figure 2c). A consequence of this compression is that the elements in the stack have geometrically decreasing weight and therefore the height of the stack used by Algorithm 2 is logarithmic in the length of the input stream.

The following proposition comes from a direct inspection of Algorithm 2.

Proposition 4.2. *Algorithm 2 accepts exactly words $u \in L$, while maintaining a stack of at most $\log n$ items of types v with $v \in \mathrm{Prefix}(\Lambda_{\mathrm{Depth}(v)})$, and a variable u_0 with $u_0 \in \mathrm{Prefix}(\Lambda_{\mathrm{Depth}(u_0)})$.*

We state that Algorithm 2 considers at most $\mathrm{O}(\log n)$ nested picks, that is $\mathrm{Depth}(u) = \mathrm{O}(\log n)$, where Depth is dynamically defined in each letter and factor inside Algorithm 2.

Lemma 4.3. *Let v be the factor used to compute R_v at line either 17 or 22 of Algorithm 2. Then $|v(i)| \leq 2|v|/3$, for all i. In particular, it holds that $\mathrm{Depth}(u) = \mathrm{O}(\log n)$.*

Proof. One only has to consider letters in Σ_∞. Hence, let R_w belongs to v for some w: either w was simplified into R_w at line 17 or at line 22 of Algorithm 2.

Let us first assume that it was done at line 22. Therefore, there is some $v' \in \mathrm{Prefix}(\Lambda_\infty)$ to the right of w with total weight greater than $|w|/2 = |R_w|/2$. This factor v' is entirely contained within v: indeed, when R_w is computed v includes v'. Therefore $|R_w| \leq 2|v|/3$.

If R_w comes from line 17, then $w = u_0$ and this u_0 is well-balanced and compressed. We claim that at the previous round the test in line 21 failed, that is $|u_0| - 1 \leq |v_2|/2$ where v_2 is the maximal well-balanced suffix of $\mathrm{top}(Stack)$. Indeed, when performing the sequence of actions following a positive test in line 21, the number of unmatched push symbols in the new u_0 is augmented at least by 1 from the previous u_0: hence, it cannot be equal to 1 as the elements in the stack have pending call symbols and therefore in the next round u_0 cannot be well-balanced. Therefore one has $|u_0| - 1 \leq |v_2|/2$. Now when $R_w = R_{u_0}$ is created, it is contains in a factor that also contains v_2 and at least one pending call before v_2. Hence, $|R_w| \leq 2|v|/3$.

Finally, the fact that $\mathrm{Depth}(u) = \mathrm{O}(\log n)$ is a direct consequence of the definition of Depth and of the fact that the weight decreases at least geometrically with nesting. □

4.2 Sketching using Suffix Sampling

We now describe the sketches our algorithm uses. They are based on a notion of suffix samplings, which ensures a good letter sampling on each suffix of some data stream. Recall that the letter sampling on a weighted word u samples a random letter $u(i)$ (with its position) with probability $|u(i)|/|u|$.

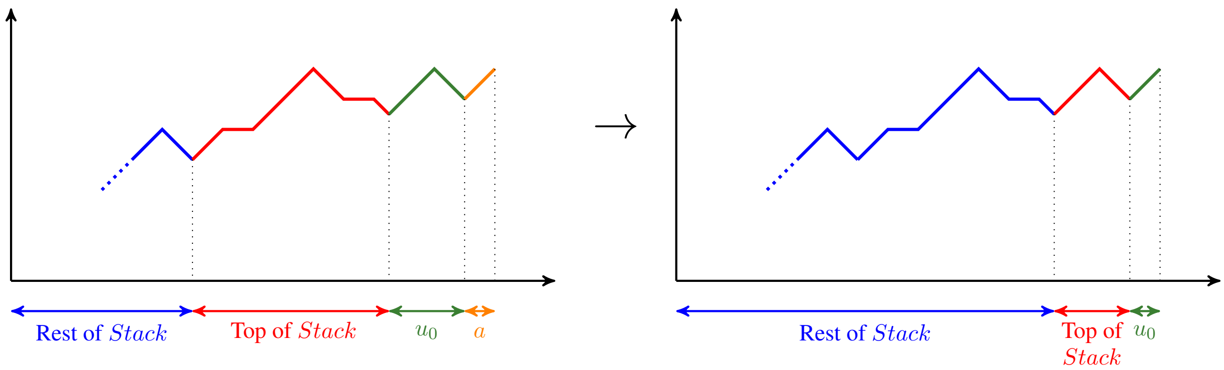
Definition 4.4. *Let u be a weighted word and let $\alpha > 1$. An α-*suffix decomposition *of u* of size *s is a sequence of suffixes $(u_l)_{1 \leq l \leq s}$ of u such that: $u_1 = u$, u_s is the last letter of u, and for all l, u_{l+1} is a strict suffix of u_l and if $|u_l| > \alpha|u_{l+1}|$ then $u_l = a \cdot u_{l+1}$ where a is a single letter.*

*An (α, t)-*suffix sampling *on u* of size *s is an α-suffix decomposition of u of size s with t letter samplings on each suffix of the decomposition.*

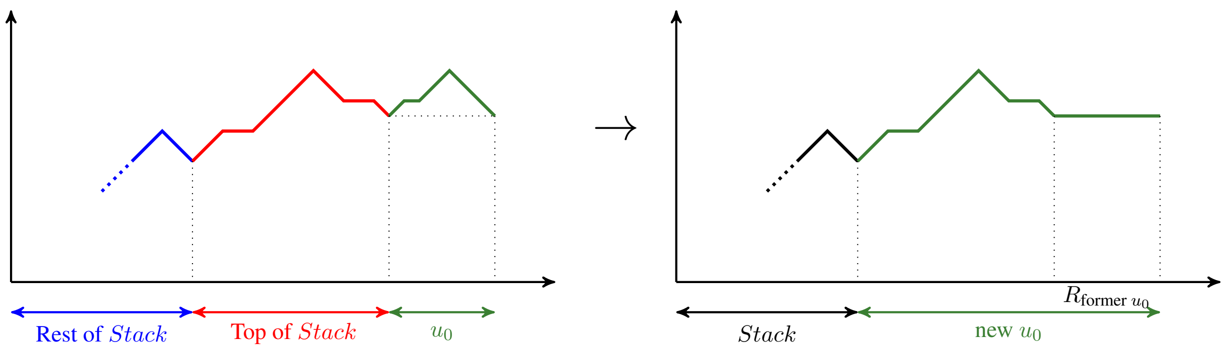
An (α, t)-suffix sampling can be either concatenated to another one, or compressed as stated below.

Proposition 4.5. *Given as input an (α, t)-suffix sampling S_u on u of size s_u and another one S_v on v of size s_v, there is an algorithm computing an (α, t)-suffix sampling on the concatenated word $u \cdot v$ of size at most $s_u + s_v$ in time $\mathrm{O}(s_u)$. There is also an algorithm computing an (α, t)-suffix sampling on u of size at most $2\lceil \log|u| / \log \alpha \rceil$ in time $\mathrm{O}(s_u)$.*

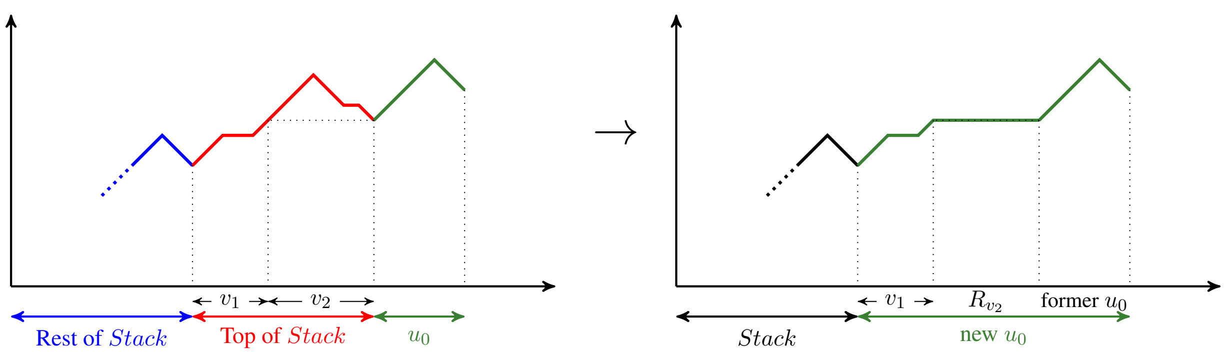
Proof. For the first algorithm, it suffices to do the following. For each suffix u_l of S_u: (1) replace u_l by $u_l \cdot v$; and (2) replace the i-th sampling of u_l by the i-th sampling of v with probability $|v|/|u|$, for $i = 1, \ldots, t$.



(a) Illustration of lines 13 to 14 from Algorithm 2



(b) Illustration of lines 16 to 19 from Algorithm 2



(c) Illustration of lines 20 to 22 from Algorithm 2

Figure 2: Illustration of Algorithm 2

For the second algorithm, do the following. For each suffix u_l of S_u (from $l = s_u$, the smallest one, to $l = 1$, the largest one): (1) replace all suffixes $u_{l-1}, u_{l-2}, \dots, u_m$ by the u_m the largest suffix u_m such that $|u_m| \leq \alpha |u_l|$; and (2) suppress all samples from deleted suffixes. □

Using this proposition, one can easily design a streaming algorithm (see Algorithm 3) constructing online a suffix decomposition of small size. Then one can slightly modify Algorithm 3 so that within each suffix of the decomposition it simulates t letter samplings in order to construct an (α, t)-suffix sampling. Letter sampling can be implemented using a standard reservoir sampling as in Algorithm 1.

Algorithm 3: α-Suffix Decomposition

```
Input: Data stream u, Real parameter α > 1
Data structure:
  i ← 0 // Current letter position in the stream
  S ← empty stack // Stack of current suffixes of the decomposition
    // Each suffix is encoded by its current weight and its starting position
Code:
While u not finished
   i++, a ← Next(u)
   For each (σ, j) ∈ S
      Replace (σ, j) by (σ + |a|, j)
   Push (|a|, i) at the top of S
   For each (σ, j) ∈ S from top to bottom
       T ← set of elements (τ, j') ∈ S below (σ, j) with τ ≤ ασ
       Replace T in S by the bottom most element of T
Output S
```

Lemma 4.6. *Given a weighted word u as a data stream and a parameter $\alpha > 1$, Algorithm 3 constructs an α-suffix decomposition on u of size at most* $1 + 2\lceil \log |u| / \log \alpha \rceil$.

4.3 Algorithm with sketches

We first describe a data structure that can be used to encode each unfinished peak v of the stack and u_0. Then, we explain how the operations of Algorithm 2 can be performed using our data structure. As a result our final algorithm is simply Algorithm 2 with the new data structure.

For each such unfinished peak v, our algorithm maintains the following sketch:

1. Its weight $|v|$, first letter $v(1)$ (with its height and weight), and whether v contains or not a pop symbol.
2. A $(1 + \varepsilon')$-suffix decomposition $w_1, \dots, w_{i_v}$ of v with $\varepsilon' = \varepsilon/(6 \log n)$ and, for each of the w_i:
   (a) A low (resp. high) estimate of its weight $|w_i|_{\texttt{low}}$ (resp. $|w_i|_{\texttt{high}}$).
   (b) T independent copies of letter sampling S_{w_i} on w_i (with their height and weight).
   (c) If a letter S_{w_i} is in a push sequence, we append to it the k-factor starting at S_{w_i} augmented by one letter (with their height and weight), as well as the corresponding matching pop sequence.

When the algorithm computes some R_{v_2} at line 22 (resp. R_{u_0} at line 17), it uses as a subroutine the algorithm of Corollary 4.13 in Section 4.4 (a variant of Theorem 3.8) on S_{w_i} with w_i the largest suffix of the decomposition inside v_2 (resp. on $S_{w_1} = S_{u_0}$). The test at line 21 is performed using the exact value of $|u_0|$ and the value of $|w_i|_{\texttt{low}}$, where w_i is similarly defined. We now explain how the sketch is updated when running Algorithm 2 (note that 2.(c) is immediate).

1. Easy to maintain. Observe that $v(1)$ is never affected by a compression unless the whole unfinished peak is being compressed, in which case we still know its exact weight.

2. The suffix decomposition is maintained using Algorithm 3 with parameter $\alpha = 1+\varepsilon'$ and the following modifications: (i) low and high estimates $(\sigma_{\texttt{low}}, \sigma_{\texttt{high}})$ replace respective suffix weights σ, and (ii) line 13 is modified such that T is now the set of $((\tau_{\texttt{low}}, \tau_{\texttt{high}}), j') \in S$ below $((\sigma_{\texttt{low}}, \sigma_{\texttt{high}}), j)$ with $\tau_{\texttt{high}} \leq (1+\varepsilon')\sigma_{\texttt{low}}$.

2.(a) Estimates of the weight of the decomposition are maintained exactly (i.e. $|w_i|_{\texttt{high}} = |w_i|_{\texttt{low}}$) whenever possible. When a new suffix has to be created because some R_{v_2} is computed at line 22, consider the largest suffix w_i in the decomposition of v that is contained within v_2 (such a w_i always exists because v_2 has at least one letter, and can easily be identified by the height of its starting element). By definition w_{i-1} is the smallest suffix in the decomposition of v strictly containing v_2. If w_{i-1} and w_i differ by exactly one letter, then $w_i = v_2$. In this case, we have $|R_{v_2} \cdot u_0|_{\texttt{high}} = |v_2|_{\texttt{high}} + |u_0|$ and $|R_{v_2} \cdot u_0|_{\texttt{low}} = |v_2|_{\texttt{low}} + |u_0|$. Otherwise, we define $|R_{v_2} \cdot u_0|_{\texttt{high}} = |w_{i-1}|_{\texttt{high}} + |u_0|$ and $|R_{v_2} \cdot u_0|_{\texttt{low}} = |w_i|_{\texttt{low}} + |u_0|$. In all other instances, the estimates $|w|_{\texttt{high}}$ and $|w|_{\texttt{low}}$ will increase by the weight of the letter that is added.

2.(b) The sampling are obtained by a reservoir sampling on each of the S_{w_i} using probability $|a|/|w_i|_{\texttt{high}}$ instead of $|a|/|w_i|$ of replacing the current S_{w_i} with a. New suffixes need to be created when two unfinished peaks (u_0 and v from the top of the stack) are merged. However, in this case, suffixes from v can be naturally extended to suffixes of $v \cdot u_0$ that still satisfy the condition of the suffix decomposition.

2.(a)&2.(b) When two unfinished peaks merge after some R_{v_2} (resp. R_{u_0}) has been computed at line 22 (resp. line 19), we replace each existing sample S_{w_i} by one sample S_{u_0} (resp. by R_{u_0}, since u_0 is compressed) with probability $|u_0|/|w_i \cdot u_0|_{\texttt{high}}$. This will ensure that the updated samples are roughly distributed according to a letter sampling on w_i augmented by u_0 (resp. R_{u_0}). The algorithm also easily deals with the case of merging letters with the $\bullet$ operation, as it knows the first and last letter of each unfinished peak.

In the next section, we show that the samplings S_{w_i} are close enough to an $(1+\varepsilon')$-suffix sampling on the w_i. This lets us build an over sampling of an $(1+\varepsilon')$-suffix sampling. We also show that it only require a polylogarithmic number of samples. Then, we explain how to recursively apply an adaptation of Theorem 3.13 (with ε') in order to obtain the compressions at line 17 and 22 while keeping a cumulative error below ε. We now state our main result whose proof uses results from the following section.

Theorem 4.7. *Let $\mathcal{A}$ be a* VPA *for L with $m \geq 2$ states, and let $\varepsilon, \eta > 0$. Then there is an ε-streaming algorithm for L with one-sided error η and memory space $\mathrm{O}(m^5 2^{3m^2}(\log^6 n)(\log 1/\eta)/\varepsilon^4)$, where n is the input length.*

Proof. We use Algorithm 2 with the tester from Corollary 4.13 for the compressions at lines 17 and 22. We know from Lemma 4.14 and Lemma 3.7 that it is enough to choose $\varepsilon' = \varepsilon/(6\log n)$, $\eta' = \eta/n$, and Fact 4.1 gives us $d = 2^{m^2}$. Therefore we need $T = 2304m^4 2^{2m^2}(\log^2 n)(\log 1/\eta)/\varepsilon^2$ independent k-factor samplings of u augmented by one, with $k = 24m2^{m^2}(\log n)/\varepsilon$. Lemma 4.11 tells us that using twice more samples from our algorithm, that is for each S_{w_i}, is enough in order to over-sample them.

Because of the sampling variant we use, the size of each decomposition is at most $96(\log^2 n)/\varepsilon + \mathrm{O}(\log n)$ by Lemma 4.11. The samplings in each element of the decomposition use memory space k, and there are $2T$ of them. Furthermore, each element of the stack has its own sketch, and the stack is of height at most $\log n$. Multiplying all those together gives us the upper bound on the memory space used by Algorithm 2. □

4.4 Final analysis

As our final algorithm may fail at various steps, the relations it considers may not correspond to any word. But still, it will produces relations R such that for any $(p, q) \in R$, there is a balanced word $u \in \Sigma^*$, such that

$p \xrightarrow{u} q$. We therefore consider the alphabet extension by any such relations R with any weight. We define Σ_Q to be the alphabet $\Sigma_=$ augmented by all such relations R, and we again extend the automaton and the language. Then, Λ_Q is simply Λ_1 with $\Sigma_= = \Sigma_Q$.

Proposition 4.8. *Each relation R that Algorithm 2 with sketches produces is in Σ_Q.*

Still the resulting automaton is $\widehat{\Sigma'}$-closed with $\Sigma' = (\Sigma_+ \cup \Sigma_- \cup \Sigma_\infty)$, and we remind that Fact 4.1 bounds the $\widehat{\Sigma'}$-diameter of $\widehat{\mathcal{A}}$ by 2^{m^2}.

Proposition 4.9. *The slicing automaton $\widehat{\mathcal{A}}$ (defined over $(\Sigma_+ \cup \widehat{\Sigma_- \cup} \Sigma_Q)$) is $\widehat{\Sigma'}$-closed with $\Sigma' = (\Sigma_+ \cup \Sigma_- \cup \Sigma_\infty)$.*

Stability. We want to show that the decomposition, weights and sampling we maintain are close enough to an $(1+\varepsilon')$-suffix sampling with correct weights. Recall that $\varepsilon' = \varepsilon/(6\log n)$.

Proposition 4.10. *Let v be an unfinished peak, and let $w_1, \ldots, w_{i_v}$ be the suffix decomposition maintained by the algorithm. The following is true:*
(1) $w_1, \ldots, w_{i_v}$ is a valid $(1+\varepsilon')$-suffix decomposition of v.
(2) For each letter a of every w_i, and for every sample s, $\Pr[S_{w_i} = a] \geq |a|/|w_i|_{\mathtt{high}}$.
(3) Each w_i, satisfies $|w_i|_{\mathtt{high}} - |w_i|_{\mathtt{low}} \leq 2\varepsilon'|w_i|_{\mathtt{low}}/3$.

Proof. Property (1) is guaranteed by the modification of Algorithm 3, which preserves even more suffixes than the original algorithm.

Properties (2) and (3) are proven by induction on the current symbol being read. Both are true when no symbol has been read yet.

We start with property (2). Let us first consider the case where no relation is computed when the current symbol a (of weight 1) is processed. Assume also that $u_0 \bullet a = u_0 \cdot a$. Then for all w_i, S_{w_i} becomes a with probability $1/|w_i|_{\mathtt{high}}$. Otherwise, S_{w_i} remains unchanged and by induction $S_{w_i} = b$ with probability at least $(1 - 1/|w_i|_{\mathtt{high}})|b|/(|w_i|_{\mathtt{high}} - 1) = |b|/|w_i|_{\mathtt{high}}$, for each other letter b of w_i. If $a \in \Sigma_=$ and u_0 ends with some $R \in \Sigma_Q$, any sample that would be either R or a is replaced by $R \bullet a$.

If instead, reading the current symbol causes R_{u_0} to be computed at line 17, the argument is as above by replacing a with R_{u_0}.

Finally, if some R_{v_2} is computed at line 22, then for each suffix w_i from v containing v_2 we proceed in the same way, replacing S_{w_i} with S_{u_0} with probability $|u_0|/|w_i|_{\mathtt{high}}$, and changing it to R_{v_2} only if it would otherwise be in v_2. For the new $w_i = R_{v_2} \cdot u_0$, we choose R_{v_2} as a sample for w_i with probability $(|w_i|_{\mathtt{high}} - |u_0|)/|w_i|_{\mathtt{high}} \geq |R|/|w_i|_{\mathtt{high}}$ and with the remaining probability $|u_0|/|w_i|_{\mathtt{high}}$ we pick the sample S_{u_0}. If a has probability at least $|a|/|u_0| = |a|/|u_0|_{\mathtt{high}}$ to be selected in u_0, it has probability at least $|a|/|w_i|_{\mathtt{high}}$ to be selected now.

We now prove property (3). If w_i has just been created, it contains only one letter of weight 1, and obviously $|w_i|_{\mathtt{low}} = |w_i|_{\mathtt{high}} = |w_i|$. In addition, if R is not some R_{v_2} computed at line 22 when the last letter was read, then $|w_i|$ is only augmented by 1 compared to the previous step. Therefore the difference $|w_i|_{\mathtt{high}} - |w_i|_{\mathtt{low}}$ does not change, and by induction it remains smaller than $2\varepsilon'|w_i|_{\mathtt{low}}/3$ which can only increase. Now consider R_{v_2} computed at line 22 and $w_i = R_{v_2} \cdot u_0$. Let w'_j be the largest suffix in the decomposition of $v_1 \cdot v_2$ that is contained within v_2. Then w'_{j-1} is the suffix immediately preceding w'_j in that decomposition.

If $|w'_{j-1}|_{\mathtt{high}} > (1+\varepsilon')|w'_j|_{\mathtt{low}}$, then from the modified Algorithm 3, the difference between those two suffixes cannot be more than one letter, and then $w_i = v_2$. Therefore, we have $|R_{v_2} \cdot u_0|_{\mathtt{high}} = |v_2|_{\mathtt{high}} + |u_0|$ and $|R_{v_2} \cdot u_0|_{\mathtt{low}} = |v_2|_{\mathtt{low}} + |u_0|$. We conclude by induction on $|v_2|$.

We end with the case $|w'_{j-1}|_{\mathtt{high}} \leq (1+\varepsilon')|w'_j|_{\mathtt{low}}$. By definition, $|R_{v_2} \cdot u_0|_{\mathtt{high}} = |w'_{j-1}|_{\mathtt{high}} + |u_0|$ and $|R_{v_2} \cdot u_0|_{\mathtt{low}} = |w'_j|_{\mathtt{low}} + |u_0|$. Therefore the difference $|w_i|_{\mathtt{high}} - |w_i|_{\mathtt{low}}$ is at most $\varepsilon'|w'_j|_{\mathtt{low}}$. Since the test at line 21 was satisfied, we know that $|w'_j|_{\mathtt{low}} \leq 2|u_0|$, and finally $\varepsilon'|w'_j|_{\mathtt{low}} \leq 2\varepsilon'(|w'_j|_{\mathtt{low}} + |u_0|)/3 \leq 2\varepsilon'|w_i|_{\mathtt{low}}/3$, as requested. □

From this we prove that the S_{w_i} can actually generate a $(1+\varepsilon')$-suffix sampling on the suffix decomposition, and that this decomposition is not too large so it will fit in our polylogarithmic memory.

Lemma 4.11. *Let $v, \mathcal{W}$ be an unfinished peak with a suffix sampling maintained by the algorithm. Then $\mathcal{W}^{\otimes 2}$ over-samples an $(1+\varepsilon')$-suffix sampling on v, and $\mathcal{W}$ has size at most $144(\log|v|)(\log n)/\varepsilon + \mathrm{O}(\log n)$.*

Proof. The first property is a direct consequence of property (1) and (2) in Proposition 4.10 as in the proof of Lemma 3.7.

The second is a consequence of the modified Algorithm 3: We remove all but one sample in the set of elements $((\tau_{\mathtt{low}}, \tau_{\mathtt{high}}), j') \in S$ below $((\sigma_{\mathtt{low}}, \sigma_{\mathtt{high}}), j)$ such that $\tau_{\mathtt{high}} \leq (1+\varepsilon')\sigma_{\mathtt{low}}$. Hence, it follows that $|w_{i-2}|_{\mathtt{high}} > (1+\varepsilon')|w_i|_{\mathtt{low}}$. Now, from property (3) of Proposition 4.10 we have that $|w_i|_{\mathtt{low}} \geq |w_i|_{\mathtt{high}} - 2\varepsilon'|w_i|_{\mathtt{low}}/3 \geq (1-2\varepsilon'/3)|w_i|_{\mathtt{high}}$. Therefore we have that $|w_{i-2}|_{\mathtt{high}} > (1+\varepsilon')(1-2\varepsilon'/3)|w_i|_{\mathtt{high}}$

By successive applications, we obtain that $|w_{i-6}|_{\mathtt{high}} > (1+\varepsilon')^3(1-2\varepsilon'/3)^3|w_i|_{\mathtt{high}}$. Now, as $|w_i|_{\mathtt{high}} > |w_i|$ and as $|w_i| \geq |w_i|_{\mathtt{low}} \geq (1-2\varepsilon'/3)|w_i|_{\mathtt{high}}$ we have: $|w_{i-6}|/(1-2\varepsilon'/3) > (1+\varepsilon')^3(1-2\varepsilon'/3)^3|w_i|$, equivalently $|w_{i-6}| > (1+\varepsilon')^3(1-2\varepsilon'/3)^4|w_i|$

Thus, the size of the suffix decomposition is at most $6\log_{(1+\varepsilon')^3(1-2\varepsilon'/3)^4}|v| \leq 6\log|v|/\log(1+\varepsilon'/3 + \mathrm{O}(\varepsilon'^2)) \leq 144(\log|v|)(\log n)/\varepsilon + \mathrm{O}(\log(n))$. □

Robustness. We first extend the notion of ε-approximation of words for a finite automaton (Definition 3.12) to any VPA when words are in Λ_Q.

Definition 4.12. *Let $R \subseteq Q^2$. Then R (ε, Σ)*-approximates *a balanced word $u \in (\Sigma_+ \cup \Sigma_- \cup \Sigma_Q)^*$ on $\mathcal{A}$, if for all $p, q \in Q$: (1) $(p,q) \in R$ when $p \xrightarrow{u} q$; (2) u is (ε, Σ)-close to some word v satisfying $p \xrightarrow{v} q$ when $(p,q) \in R$.*

Then, we state an analogue of Theorem 3.8 for words in Λ_Q instead of Λ_1. We present the result as an algorithm with an output R as in Theorem 3.13. We also need to adapt it to the sampling we have, where the suffixes do not exactly match the peaks we want to compress.

Corollary 4.13. *Let $\mathcal{A}$ be a VPA with $m \geq 2$ states and Σ-diameter $d \geq 2$. There is an algorithm that:*

1. *Take as input $\varepsilon', \eta > 0$ and T k-factors of $w_1, \ldots, w_T$ of some weighted word $v \in \Lambda_Q$, such that $T = 4kt$, $t = 2\lceil 4dm^3(\log 1/\eta)/\varepsilon' \rceil$ and $k = \lceil 4dm/\varepsilon' \rceil$;*
2. *Output a set $R \subseteq Q \times Q$ that (ε', Σ)-approximates v on $\mathcal{A}$ with bounded error η, when each factor w_i come from an independent k-factor sampling on $\widehat{v}$.*

Let v' be obtained from v by at most $\varepsilon'|v|$ balanced deletions. Then, the conclusion is still true if the algorithm is given an independent k-factor sampling on $\widehat{v'}$ for each w_i instead, except that R now provides a $(3\varepsilon', \Sigma)$-approximation. Last, each sampling can be replaced also by an over-sampling.

Proof. The argument is similar to the one of Theorem 3.8, and we use again as a subroutine the algorithm of Theorem 3.13 for $\widehat{\mathcal{A}}$ with restricted alphabet $\widehat{\Sigma'}$, where $\Sigma' = (\Sigma_+ \cup \Sigma_- \cup \Sigma_\infty)$. Remind that $\mathcal{A}$ is Σ'-closed and its Σ'-diameter is the Σ-diameter of $\mathcal{A}$.

For the case when we do not have exact k-factor sampling on v however, we need to compensate for the prefix of v of size $\varepsilon'|v|$ that may not be included in the sampling. This introduces potentially an additional error of weight $2\varepsilon'|v|$ on the approximation R. □

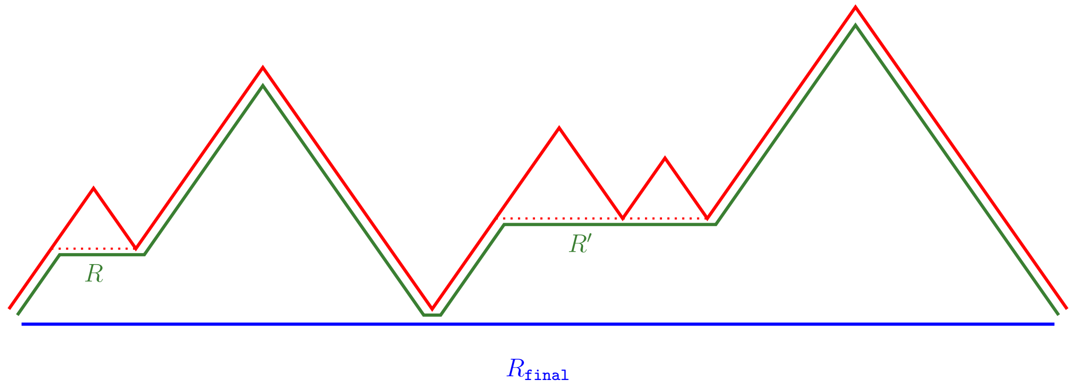


Figure 3: Constructing the words u_0, u_1 and u_2 as in Lemma 4.14 where $\mathrm{Depth}(R_{\texttt{final}}) = 2$

We are now ready to state the robustness of our algorithm. For $u \in \Sigma^n$, we apply all compressions from lines 17 and 22 of Algorithm 2 using the tester from Corollary 4.13 with $\varepsilon' = \varepsilon/(6 \log n)$ and $\eta' = \eta/n$. This leads to a final $R_{\texttt{temp}} \in \Sigma_Q$.

Lemma 4.14. *Let $\mathcal{A}$ a* VPA *recognizing L and let $u \in \Sigma^n$. Let $R_{\texttt{final}}$ the final value of $R_{\texttt{temp}}$ in the Algorithm 2 with sketches. If $u \in L$, then $R_{\texttt{final}} \in L$; and if $R_{\texttt{final}} \in L$, then $\mathsf{bdist}_\Sigma(u, L) \leq \varepsilon n$ with probability at least $1 - \eta$.*

Proof. One way is easy. A direct inspection reveals that each substitution of a factor w by a relation R enlarges the set of possible w-transitions.Therefore $R_{\texttt{final}} \in L$ when $u \in L$.

For the other way, consider some word u such that $R_{\texttt{final}} \in L$. Since the tester of Corollary 4.13 has bounded error $\eta' = \eta/n$ and was called at most than n times, none of the calls fails with probability at least $1 - \eta$. From now on we assume that we are in this situation.

Let $h = \mathrm{Depth}(R_{\texttt{final}})$. We will inductively construct sequences $u_0 = u, \ldots, u_h = R_{\texttt{final}}$ and $v_h = R_{\texttt{final}}, \ldots, v_0$ such that for every $0 \leq l \leq h$, $u_l, v_l \in (\Sigma_+ \cup \Sigma_- \cup \Sigma_Q)^*$, $\mathsf{bdist}_\Sigma(u_l, v_l) \leq 3(h-l)\varepsilon'|u_l|$ and $v_l \in L$. Furthermore, each word u_l will be the word u with some substitutions of factors by relations R computed by the tester. Therefore, $\mathrm{Depth}(u_l)$ is well defined and will satisfy $\mathrm{Depth}(u_l) = l$. This will conclude the proof using that $\mathrm{Depth}(u) \leq \log_{3/2} n$ from Lemma 4.3. Indeed, since $h \leq \mathrm{Depth}(u)$, it will give us $\mathsf{bdist}_\Sigma(u, v_0) \leq 6\varepsilon' n \log n \leq \varepsilon n$.

We first define the sequence $(u_l)_l$ (see Figure 3 for an illustration). Starting from $u_0 = u$, let u_{l+1} be the word u_l where some factors in Λ_Q have been replaced by a $(3\varepsilon', \Sigma)$-approximation in Σ_Q. These correspond to all the approximations eventually performed by the algorithm that did not involve a symbol already in Σ_Q. Some approximations are eventually collapsed together into a single symbol by the $\bullet$ operation (in the example in Figure 3 this is the case for R' for instance). Observe that after this collapse, the symbol is still a $(3\varepsilon', \Sigma)$-approximation. In particular, $u_h = R_{\texttt{final}}$, $u_l \in (\Sigma_+ \cup \Sigma_- \cup \Sigma_Q)^*$ and $\mathrm{Depth}(u_l) = l$ by construction.

We now define the sequence $(v_l)_l$ such that $v_l \in L$. Each letter of v_l will be annotated by an accepting run of states for $\mathcal{A}$. Set $v_h = R_{\texttt{final}}$ with an accepting run from p_{in} to q_f for some $(p_{in}, q_f) \in R_{\texttt{final}} \cap (Q_{in} \times Q_f)$. Consider now some level $l < h$. Then v_l is simply v_{l+1} where some letters $R \in \Sigma_Q$ in common with u_{l+1} are replaced by some factors in $w \in (\Lambda_Q)^*$ as explained in the next paragraph. Those letters are the ones that are present in u_l but not u_{l+1}, and are still present in v_{l+1} (i.e. they have not been further approximated down the chain from u_{l+1} to u_h, or deleted by edit operations moving up from v_h to v_{l+1}).

Let $w \in (\Lambda_Q)^*$ be one of those factors and $R \in \Sigma_Q$ its respective $(3\varepsilon', \Sigma)$-approximation. By hypothesis R is still in v_{l+1} and corresponds to a transition (p, q) of the accepting run of v_{l+1}. We replace R by a factor w' such that $p \xrightarrow{w'} q$ and $\mathsf{bdist}_\Sigma(w, w') \leq 3\varepsilon'|w|$, and annotate w' accordingly. By construction, the resulting word v_l satisfies $v_l \in L$ and $\mathsf{bdist}_\Sigma(u_l, v_l) \leq 3(h-l)\varepsilon'|u_l|$. □

References

[1] N. Alon, M. Krivelevich, I. Newman, and M. Szegedy. Regular languages are testable with a constant number of queries. *SIAM Journal on Computing*, 30(6), 2000.

[2] N. Alon, Y. Matias, and M. Szegedy. The space complexity of approximating the frequency moments. *Journal of Computer and System Sciences*, 58(1):137–147, 1999.

[3] R. Alur. Marrying words and trees. In *Proc. of 26th ACM Symposium on Principles of Database Systems*, pages 233–242, 2007.

[4] R. Alur, M. Arenas, P. Barceló, K. Etessami, N. Immerman, and L. Libkin. First-order and temporal logics for nested words. In *Proc. of 22nd IEEE Symposium on Logic in Computer Science*, pages 151–160, 2007.

[5] R. Alur, K. Etessami, and P. Madhusudan. A temporal logic of nested calls and returns. In *Proc. of 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 467–481, 2004.

[6] R. Alur and P. Madhusudan. Adding nesting structure to words. *Journal of the ACM*, 56(3), 2009.

[7] A. Babu, N. Limaye, and G. Varma. Streaming algorithms for some problems in log-space. In *Proc. of 7th Conference on Theory and Applications of Models of Computation*, pages 94–104, 2010.

[8] M. Blum, W. Evans, P. Gemmell, S. Kannan, and M. Naor. Checking the correctness of memories. *Algorithmica*, pages 90–99, 1995.

[9] M. Blum and S. Kannan. Designing programs that check their work. *Journal of the ACM*, 42(1):269–291, 1995.

[10] M. Blum, M. Luby, and R. Rubinfeld. Self-testing/correcting with applications to numerical problems. *Journal of Computer and System Sciences*, 47(3):549–595, 1993.

[11] B. von Braunmühl and R. Verbeek. Input-driven languages are recognized in log n space. In *Proc. of 4th Conference on Fundamentals of Computation Theory*, volume 158, pages 40–51, 1983.

[12] M. Chu, S. Kannan, and A. McGregor. Checking and spot-checking the correctness of priority queues. In *Proc. of 34th International Colloquium on Automata, Languages and Programming*, pages 728–739, 2007.

[13] P. Dymond. Input-driven languages are in $\log n$ depth. *Information Processing Letters*, 26(5):247–250, 1988.

[14] J. Feigenbaum, S. Kannan, M. Strauss, and M. Viswanathan. Testing and spot-checking of data streams. *Algorithmica*, 34(1):67–80, 2002.

[15] E. Fischer, F. Magniez, and M. de Rougemont. Approximate satisfiability and equivalence. *SIAM Journal on Computing*, 39(6):2251–2281, 2010.

[16] O. Goldreich, S. Goldwasser, and D. Ron. Property testing and its connection to learning and approximation. In *Proc. of 37th IEEE Symposium on Foundations of Computer Science*, pages 339–348, 1996.

[17] J. Hopcroft, R. Motwani, and J. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley, 2006.

[18] C. Konrad and F. Magniez. Validating XML documents in the streaming model with external memory. *ACM Transactions on Database Systems*, 38(4):27, 2013. Special issue of ICDT'12.

[19] L. Libkin. Logics for unranked trees: An overview. *Logical Methods in Computer Science*, 2(3), 2006.

[20] F. Magniez and M. de Rougemont. Property testing of regular tree languages. *Algorithmica*, 49(2):127–146, 2007.

[21] F. Magniez, C. Mathieu, and A. Nayak. Recognizing well-parenthesized expressions in the streaming model. *SIAM Journal on Computing*, 43(6):1880–1905, 2014.

[22] K. Mehlorn. Pebbling mountain ranges and its application to dcfl-recognition. In *Proc. of 7th International Colloquium on Automata, Languages, and Programming*, pages 422–435, 1980.

[23] S. Muthukrishnan. Data streams: Algorithms and applications. *Foundations and Trends in Theoretical Computer Science*, 1(2):117–236, 2005.

[24] A. Ndione, A. Lemay, and J. Niehren. Approximate membership for regular languages modulo the edit distance. *Theoretical Computer Science*, 487:37–49, 2013.

[25] M. Parnas, D. Ron, and R. Rubinfeld. Testing membership in parenthesis languages. *Random Structures & Algorithms*, 22(1):98–138, 2003.

[26] A. Rajeev and P. Madhusudan. Visibly pushdown languages. In *Proc. of 36th ACM Symposium on Theory of Computing*, pages 202–211, 2004.

[27] L. Segoufin and C. Sirangelo. Constant-memory validation of streaming XML documents against DTDs. In *Proc. of 11th International Conference on Database Theory*, pages 299–313, 2007.

[28] L. Segoufin and V. Vianu. Validating streaming XML documents. In *Proc. of 11th ACM Symposium on Principles of Database Systems,*, pages 53–64, 2002.

ECCC ISSN 1433-8092

http://eccc.hpi-web.de