Streaming Property Testing of Visibly Pushdown Languages

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

1Univ Paris Diderot, Sorbonne Paris-Cité, LIAFA, CNRS, France
2CNRS, LIAFA, Univ Paris Diderot, Sorbonne Paris-Cité, France
3University 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 poly((log n)/ϵ).

This constructions relies on a (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.
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 NC1-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 [29, 28], 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 $\varepsilon$-tester for a language $L$ distinguishes with high probability the words in $L$ from those $\varepsilon$-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 $\varepsilon$-tester for well-parenthesized expressions with two types of parentheses is between $\Omega(n^{1/11})$ and $O(n^{2/3})$ [26], 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, 25].

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 $\varepsilon$-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 \textsc{VPl}, the \textit{balanced-edit distance}, which refines the edit distance on balanced words. It can also be interpreted as the edit distance on trees when trees are encoded as balanced words. Neutral symbol can be deleted/inserted, but any push symbol can only be deleted/inserted together with its matching pop symbol. Since our distance is larger than the standard edit distance, our testers are also valid for that 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 sequences. 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 (\textbf{Theorem 3.10}). Such a property tester has already been constructed in \cite{25} extending previous constructions for unweighted regular languages \cite{1, 24}. Our construction is slightly simpler and could be of independent interest. As a consequence we get a streaming property tester with polylogarithmic memory for recognizing peak instances of any given \textsc{VPl} (\textbf{Theorem 3.8}), a task already hard for streaming algorithms and property testers (\textbf{Fact 3.1}).

In Section 4, we construct our main tester for a \textsc{VPl} $L$ given by some \textsc{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 (\textbf{Proposition 4.4}) is to rewrite the input word as a peak formed by potentially a linear number of intermediate peaks, but with only a logarithmic number 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 (\textbf{Definition 4.6}). 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, 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 (\textbf{Lemma 4.12}). Then we prove the robustness of our algorithm, that is words that are $\varepsilon$-far from $L$ are rejected with high probability (\textbf{Lemma 4.15}). As a consequence, we get a one-pass streaming $\varepsilon$-tester for $L$ with one-sided error $\eta$ and memory space $O(m^5n^{3m^2}(\log n)^3(\log 1/\eta)/\varepsilon^4)$, where $m$ is the number of states of a \textsc{VPA} recognizing $L$ (\textbf{Theorem 4.8}).
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 $\Sigma$ we denote the set of finite words over $\Sigma$ by $\Sigma^*$. For a word $u = u(1)u(2) \cdots u(n)$, we call $n$ the length of $u$, and $u(i)$ the $i$th 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 $v \leq v'$, 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 $i \leq i' \leq j \leq j'$ 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 $S$ and $S'$, we say that $S \leq S'$ if for each factor $v \in S$ there is a corresponding factor $v' \in S'$ such that $v \leq v'$.

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 $\Sigma$ is a pair $(u, \lambda)$ where $u \in \Sigma^*$ and $\lambda$ is 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 $(u, \lambda)$ is the length of $u$. For simplicity, we will denote by $u$ the weighted word $(u, \lambda)$. 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 $l \geq 0$ is the smallest integer such that $|u[i, i+l]| \geq k$ 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 $u$. We implicitly also mention that both of them can be implemented using a standard reservoir sampling (see Algorithm 4 in Appendix A for letter sampling).

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 $\Pr(\mathcal{W}_2 \text{ samples } S_2) \geq \Pr(\mathcal{W}_1 \text{ samples } S_1)$, 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 $A = (Q, \Sigma, Q_{in}, Q_f, \Delta)$ where $Q$ is a finite set of control states, $\Sigma$ 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 $A$ from $p$ to $q$ while processing $u$, and we call $(p, q)$ a $u$-transitions. For $\Sigma' \subseteq \Sigma$, the $\Sigma'$-diameter (or simply diameter when $\Sigma' = \Sigma$) of $A$ 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 $A$ is $\Sigma'$-closed, when $p \xrightarrow{u} q$ for some $u \in \Sigma'$ if and only if $p \xrightarrow{u'} q$ for some $u' \in \Sigma'^*$.

A pushdown alphabet is a triple $\langle \Sigma_+, \Sigma_-, \Sigma_v \rangle$ that comprises three disjoint finite alphabets: $\Sigma_+$ is a finite set of push symbols, $\Sigma_-$ is a finite set of pop symbols, and $\Sigma_v$ is a finite set of neutral symbols. For any such triple, let $\Sigma = \Sigma_+ \cup \Sigma_- \cup \Sigma_v$. Intuitively, a visibly pushdown automaton [27] over $\langle \Sigma_+, \Sigma_-, \Sigma_v \rangle$ 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 [27]). A visibly pushdown automaton (VPA) over $\langle \Sigma_+, \Sigma_-, \Sigma_v \rangle$ is a tuple $A = (Q, \Sigma, \Gamma, Q_{in}, Q_f, \Delta)$ 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, $\Gamma$ is a finite stack alphabet, and $\Delta \subseteq (Q \times \Sigma_+ \times Q \times \Gamma) \cup (Q \times \Sigma_- \times \Gamma \times Q) \cup (Q \times \Sigma_v \times Q)$ is the transition relation.

To represent stacks we use a special bottom-of-stack symbol $\bot$ that is not in $\Gamma$. A configuration of a VPA
$\mathcal{A}$ is a pair $(\sigma, q)$, where $q \in Q$ and $\sigma \in \Sigma^*$. For $a \in \Sigma$, there is an $a$-transition from a configuration $(\sigma, q)$ to $(\sigma', q')$, 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 $q \xrightarrow{a}(q', \text{push} \gamma))$.
- If $a$ is a pop symbol, then $\sigma = \sigma' \gamma$ for some $(q, a, \gamma, q') \in \Delta$, and we write $(q, \text{pop} \gamma) \xrightarrow{a} q'$.
- 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 $(p, q) \in Q_m \times Q_f$ such that $(\bot, p) \xrightarrow{u} (\bot, q)$.

The language $L(A)$ 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 of $u$ 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 height($u$) = 0 and 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 height($u[i,j]$) = 0 and height($u[i,k]$) $> 0$ for all $i < k < j$. By extension, the height of $u(i)$ is height($u[1,i-1]$) when $u(i)$ is a push symbol, and height($u[1,i]$) otherwise.

For all balanced words $u$, the property $(\sigma, p) \xrightarrow{u \gamma} (\sigma, q)$ does not depend on $\sigma$, 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 $\Sigma'$-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 $\lambda$, for the new $u(i)$. Then the (standard) edit distance $\text{dist}(u, v)$ 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 $\Sigma'$, we also define $\text{dist}_{\Sigma'}(u, v)$ where the insertions are restricted to letters in $\Sigma'$.

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 a 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. We define the balanced-edit distance $\text{bdist}(u, v)$ between two balanced words as the total cost of a sequence of balanced-edit operations changing $u$ to $v$. Similarly to $\text{dist}_{\Sigma'}(u, v)$ we define $\text{bdist}_{\Sigma'}(u, v)$.

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 $(\varepsilon, \Sigma')$-far from $v$ if $\text{dist}_{\Sigma'}(u, v) > \varepsilon|u|$, or $\text{bdist}_{\Sigma'}(u, v) > \varepsilon|u|$, depending on the context. We omit $\Sigma'$ when $\Sigma' = \Sigma$.

**Streaming Property Testers.** An $\varepsilon$-tester for a language $L$ accepts all inputs which belong to $L$ with probability 1 and rejects with high probability all inputs which are $\varepsilon$-far from $L$, i.e. that are $\varepsilon$-far from any element of $L$. In particular, a tester for some given distance is also a tester for any other smaller distance. 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 $\varepsilon$-tester for $L$ with one-sided error $\eta$ and memory $s(n)$ is a randomized algorithm $A$ such that, for any input $u$ of length $n$
given as a data stream:
- If \( u \in L \), then \( A \) accepts with probability 1;
- If \( u \) is \( \varepsilon \)-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, i.e. elements of the language

\[
\Lambda = \bigcup_{j \geq 0}((\Sigma_+)^* \cdot \Sigma_+)^j \cdot ((\Sigma_-)^* \cdot (\Sigma_-)^*)^j.
\]

Those instances are already hard for both streaming algorithms and property testing algorithms. Indeed, consider the language \( \text{Disj} \subseteq \Lambda \) over alphabet \( \Sigma = \{0, 1, \overline{0}, \overline{1}, a\} \) and defined by the union of all languages

\[
a^* \cdot x(1) \cdot a^* \cdot x(2) \cdots x(j) \cdot a^* \cdot y(j) \cdot a^* \cdots y(1) \cdot a^*,
\]

where \( j \geq 1 \), \( x, y \in \{0, 1\}^j \), and \( x(i)y(i) \neq 1 \) for all \( i \).

Then \( \text{Disj} \) can be recognized by a VPA with 3 states, \( \Sigma_+ = \{0, 1\}, \Sigma_- = \{\overline{0}, \overline{1}\} \) and \( \Sigma_0 = \{a\} \). 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 [26] 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 \( \text{Disj} \) requires memory space \( \Omega(n/p) \), where \( n \) is the input length. Moreover, any (non-streaming) \( (2^{-6}) \)-tester for \( \text{Disj} \) 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 \( \varepsilon \)-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 \( u = \left( \prod_{i=1}^{j} v_i \cdot a_i \right) \cdot v_{j+1} \cdot \left( \prod_{i=j+1}^{l} \overline{b_i} \cdot w_i \right) \), where \( a_i \in \Sigma_+ \), \( b_i \in \Sigma_- \), and \( v_i, w_i \in (\Sigma_+)^* \). Define the slicing of \( u \) (see Figure 1) as the word \( \widehat{u} \) over the alphabet \( \widehat{\Sigma} = (\Sigma_+ \times \Sigma_+) \cup (\Sigma_- \times \{I\}) \cup (\{I\} \times \Sigma_-) \) defined by \( \widehat{u} = \left( \prod_{i=1}^{j} (v_i(1), I) \cdots (v_i(|v_i|), I) \cdot (I, w_i(1)) \cdots (I, w_i(|w_i|)) \cdot (a_i, \overline{b_i}) \right) \cdot (v_{j+1}(1), I) \cdots (v_{j+1}(|v_{j+1}|), I) \).

**Definition 3.2.** Let \( A = (Q, \Sigma, \Gamma, Q_m, Q_f, \Delta) \) be a VPA. The slicing of \( A \) is the finite automaton \( \widehat{A} = (\widehat{Q}, \widehat{\Sigma}, \widehat{Q_m}, \widehat{Q_f}, \widehat{\Delta}) \) where \( \widehat{Q} = Q \times Q \), \( \widehat{Q_m} = Q_m \times Q_f \), \( \widehat{Q_f} = \{(p, p) : p \in Q\} \), and the transitions \( \widehat{\Delta} \) are:
1. \( (p, q) \xrightarrow{(a,b)} (p', q') \) when \( p \xrightarrow{a}(p', \text{push}(\gamma)) \) and \( (q', \text{pop}(\gamma)) \xrightarrow{b} q \) are both transitions of \( \Delta \).
2. \( (p, q) \xrightarrow{(c)1} (p', q) \), resp. \( (p, q) \xrightarrow{(c)1} (p, q') \), when \( p \xrightarrow{c} p' \), resp. \( q \xrightarrow{c} q' \), is a transition of \( \Delta \).
Lemma 3.3. If $A$ is a VPA accepting $L$, then $\hat{A}$ is a finite automaton accepting $\hat{L} = \{ \hat{u} : u \in L \cap \Lambda \}$. 

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

Regular languages are known to be $\varepsilon$-testable for the Hamming distance with $O((\log 1/\varepsilon)/\varepsilon)$ non-adaptive queries on the input word $[\Pi]$, 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 $\text{bdist}(u, v) \leq 2\text{dist}(\hat{u}, \hat{v})$. The 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 $\Sigma_\varepsilon = \emptyset$.

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

Proof. The tester of $[\Pi]$ 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 $\hat{u}$ 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 $O(\log n)$ 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 $\Sigma_\varepsilon \neq \emptyset$ 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 $v \in (\Sigma_\varepsilon)^*$ (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 $A$ recognizing $L$. Then, we compress $v$ by its corresponding relation $R_v = \{(p, q) : p \xrightarrow{v} q\}$, and we see the subset $\hat{R}_v \subseteq Q \times Q$ as a new letter, call it $\hat{R}$, and the possible weights for $\hat{R}$ correspond to the weights of words $v$ such that $R = R_v$. We augment $\Sigma_\varepsilon$ by those new letters, and call this new (finite) alphabet $\Sigma_{\hat{\varepsilon}}$. 

![Figure 1: Slicing of a word $u \in \Lambda$ and evolution of the stack height for $u$.](image)
We also extend the automaton $A$ and the language $L$ with $\Sigma_0$. Doing so, we have compressed $u \in \Lambda$ to a weighted word of $\Lambda_1 = \bigcup_{j \geq 1} (\Sigma_0 \cdot \Sigma_\ast)^j \cdot \Sigma_0 \cdot (\Sigma_\ast \cdot \Sigma_0)^j$. Since there is a correspondence between letters $R \in \Sigma_0$ and words $v \in (\Sigma_\ast)^\ast$ with $|v| = |R|$ and $R = R_v$, we can arbitrarily reason on either the old or the new alphabet. Moreover, the corresponding slicing automaton $\hat{A}$ still has diameter at most $2m^2$.

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

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 $\hat{u}$ against $\hat{L}$, which is now a language of weighted words. More precisely, the weight of a letter in $\hat{u}$ is defined by $|(a_+, a_-)| = 1$ and $|(I, R)| = |(R, I)| = |R|$. In Section 3.2 we construct such a tester. The remaining difficulty is to provide to this tester an appropriate sampling on $\hat{u}$ while processing $u$.

Our tester for weighted regular languages is based on $k$-factor sampling on $\hat{u}$ 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 (see proof of Lemma 3.7 in Appendix B).

**Definition 3.6.** For a weighted word $u \in \Lambda$, denote by $W_k(u)$ the sampling over factors of $\hat{u}$ 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, extend 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 $W_k(u)$ over samples the $k$-factor sampling on $\hat{u}$.

**Theorem 3.8.** Let $A$ be a VPA for $L$ with $m \geq 2$ states, and let $\varepsilon, \eta > 0$. Then there is a streaming $\varepsilon$-tester for $L \cap \Lambda$ with one-sided error $\eta$ and memory space $O((m^3 \log(1/\eta)/\varepsilon^2)(m^3/\varepsilon + \log n))$, where $n$ is the input length.

**Proof.** The proof uses Theorem 3.10 for weighted regular languages. Observe that $\text{bdist}(u, v) \leq 2\text{dist}(\hat{u}, \hat{v})$, and moreover the slicing automaton $\hat{A}$ 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 $\Lambda_1$. We use Lemma 3.7 to obtain an over-sampling of $t$ independent $k$-factor samplings on $\hat{u}_1$ with a total of $4tk$ instances of $W_k(u)$, each using $k$ letters and $\log n$ bits to encode the height. From Theorem 3.10 $t = 4[4dm^3(\log 1/\eta)/\varepsilon]$ and $k = \lfloor 4dm/\varepsilon \rfloor$ suffice.

**3.2 Tested Weighting Regular Languages**

We first design a non-adaptive 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 [11] for the Hamming distance and we adapt this tester to weighted words for the simple case of edit distance. Such a property tester has been already constructed first for edit distance in [24], and later on for weighted words in [25], with an approach based on [11].

In this work, we take an alternative approach that we believe simpler, but slightly less efficient than the tester of [25]. We consider the graph of components of the automaton and focus on paths in this graph; we introduce however a new criterion, $\kappa$-saturation (for some parameter $0 < \kappa \leq 1$), that permits to significantly simplify the correctness proof of the tester compared to the one in [11] and in [25]. However, we prefer to differ this proof to Appendix C. In particular Lemma C.3 (in the appendix) permits to design a non-adaptive tester for $L$ and also to approximate the action of $u$ on $A$ as follows.

**Definition 3.9.** Let $\Sigma' \subseteq \Sigma$ and $R \subseteq Q \times Q$. Then $R (\varepsilon, \Sigma')$-approximates a word $u$ on $A$ (or simply $\varepsilon$-approximates when $\Sigma' = \Sigma$), if for all $p, q \in Q$: (1) $(p, q) \in R$ when $p \xrightarrow{u} q$; (2) $u$ is $(\varepsilon, \Sigma')$-close to some word $v$ satisfying $p \xrightarrow{v} q$ when $(p, q) \in R$. 

8
Theorem 3.10. Let $A$ 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 $t \geq 2d m^3(\log 1/\eta)/\varepsilon$;
2. Outputs a set $R \subseteq Q \times Q$ that $\varepsilon$-approximates $u$ on $A$ with one-sided error $\eta$, 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 $A$ is $\Sigma'$-closed, and $d$ is the $\Sigma'$-diameter of $A$, then $R$ also $(\varepsilon, \Sigma')$-approximates $u$ on $A$.

4 General case

4.1 Exact Algorithm

Fix a $VPA$ $A$ recognizing some $VPL$ $L$. A general balanced input instance $u$ will have more than one peak $v \in \Lambda$ and therefore 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 $\Sigma_0$ encoding all possible relations $R_v$ for $v \in \Sigma_+$. Then $\Lambda_{h+1}$ is simply $\Lambda$ over an alphabet $\Sigma_+ = \Sigma_h$, and $\Sigma_h$ encodes all possible relations $R_v$ for words $v \in (\Lambda_h)^*$. As before, we naturally augment the automaton $A$ and the language $L$ with these new sets. However we keep the notation $\Sigma$ as $\Sigma_+ \cup \Sigma_\infty \cup \Sigma_\infty$.

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

Fact 4.1. Let $A$ be a $VPA$ with $m$ states. Then the $\Sigma$-diameter of $A$ is at most $2^m$.

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 sketches. While having processed the prefix $u[1, i]$ of the data stream $u$, Algorithm 1 maintains a suffix $u_0 \in \text{Prefix}(\Lambda_\infty)$ of $u[1, i]$, that is an unfinished peak, with some simplifications of factors $v \in \Lambda_\infty$ 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 $R_{\text{temp}} \subseteq Q \times Q$ that is the set of transitions for the maximal prefix of $u[1, i]$ in $\Lambda_\infty$. When the stream is over, the set $R_{\text{temp}}$ is used to decide whether $u \in L$ or not.

We now need to define the $\bullet$ operation used by the algorithm, to concatenate while merging adjacent neutral symbols, and the depth of a factor for the analysis.

**Definition 4.2.** Let $u$ be a weighted word, and let $a, b$ be weighted letters. Then $(ua) \bullet b$ is defined as $uab$ when either $a$ or $b$ is not neutral, and otherwise as $u \cdot R_{ab}$, where $R_{ab}$ denotes the set of $ab$-transitions.

**Definition 4.3.** For each factor constructed in Algorithm 1, Depth is defined dynamically by $\text{Depth}(a) = 0$ when $a \in \Sigma$, $\text{Depth}(v) = \max_i \text{Depth}(v(i))$ and $\text{Depth}(R_v) = \text{Depth}(v) + 1$.

When a push symbol $a$ comes after the pop sequence, $u_0 \cdot a$ is no longer in $\text{Prefix}(\Lambda_\infty)$, and Algorithm 1 puts $u_0$ on the stack of unfinished peaks (see lines 10 to 11 and Figure 3a) and $u_0$ is reset to $a$. In other situations, one adds $a$ to $u_0$. In case $u_0$ becomes a word in $\Lambda_\infty$ (see lines 13 to 16 and Figure 3b), Algorithm 1 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 $R_{\text{temp}}$ by taking its composition with $R_{u_0}$.
Algorithm 1: Exact Tester for a VPL

Input: Well-balanced data stream $u$

Data structure:

1. $Stack \leftarrow \emptyset$ // Stack of items $v$ with $v \in \text{Prefix}(\Lambda_{\infty})$
2. $u_0 \leftarrow \emptyset$ // $u_0 \in \text{Prefix}(\Lambda_{\infty})$ is a suffix of the processed part $u[1,i]$ of $u$
3. $R_{\text{temp}} \leftarrow \{(p,p)\}_{p \in Q}$ // Set of transitions for the maximal prefix of $u[1,i]$ in $\Lambda_{\infty}$

Code:

1. While $u$ not finished
2. 
3. $a \leftarrow \text{Next}(u)$ // Read and process a new symbol $a$
4. If $a \in \Sigma$, and $u_0$ has a letter in $\Sigma$. // $u_0 \cdot a \notin \text{Prefix}(\Lambda_k)$
5. Push $u_0$ on $Stack$, $u_0 \leftarrow a$
6. Else $u_0 \leftarrow u_0 \cdot a$
7. If $u_0$ is well-balanced // $u_0 \in \Lambda_{\infty}$: compression
8. Compute $R_{u_0}$ the set of $u_0$-transitions
9. If $Stack = \emptyset$, then $R_{\text{temp}} \leftarrow R_{u_0} \cdot R_{u_0}$, $u_0 \leftarrow \emptyset$
10. Else Pop $v$ from $Stack$, $u_0 \leftarrow v \cdot R_{u_0}$
11. Let $(v_1, v_2) \leftarrow \text{top}(Stack)$ s.t. $v_2$ is maximal and well-balanced // $v_2 \in \Lambda_{\infty}$
12. If $|u_0| \geq |v_2|/2$ // $u_0$ is big enough and $v_2$ can be replaced by $R_{v_2}$
13. Compute $R_{v_2}$ the set of $v_2$-transitions, Pop $v$ from $Stack$, $u_0 \leftarrow (v_1 \cdot R_{v_2}) \cdot u_0$
14. If $(Q_{in} \times Q_f) \cap R_{\text{temp}} \neq \emptyset$, Accept; Else Reject // $u = u_0$ and $R_{\text{temp}} = R_u$

In order to bound the size of the stack, Algorithm 1 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 17 to 19 and Figure 3c). 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 1 is logarithmic in the length of the input stream.

The following proposition comes from a direct inspection of Algorithm 1.

**Proposition 4.4.** Algorithm 1 accepts exactly words $u \in L$, while maintaining a stack of at most $\log n$ items of types $v$ with $v \in \text{Prefix}(\Lambda_{\text{Depth}(v)})$, and a variable $u_0$ with $u_0 \in \text{Prefix}(\Lambda_{\text{Depth}(u_0)})$.

We state that Algorithm 1 considers at most $O(\log n)$ nested picks, that is, $\text{Depth}(u) = O(\log n)$, where Depth is dynamically defined in each letter and factor inside Algorithm 1.

**Lemma 4.5.** Let $v$ be the factor used to compute $R_{u_i}$ at line either 14 or 19 of Algorithm 1. Then $|v(i)| \leq 2|v|/3$, for all $i$. In particular, it holds that $\text{Depth}(u) = O(\log n)$.

**Proof.** One only has to consider letters in $\Sigma_{\infty}$. Hence, let $R_{w}$ belongs to $v$ for some $w$: either $w$ was simplified into $R_w$ at line 14 or at line 19 of Algorithm 1.

Let us first assume that it was done at line 19. Therefore, there is some $v' \in \text{Prefix}(\Lambda_{\infty})$ 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 14, then $w = u_0$ and this $u_0$ is well-balanced and compressed. We claim that at the previous round the test in line 18 failed, that is $|u_0| - 1 \leq |v_2|/2$ where $v_2$ is the maximal well-balanced suffix of top$(Stack)$. Indeed, when performing the sequence of actions following a positive test in line 18 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, $|R_w| \leq 2|v|/3$. 

10
Finally, the fact that Depth\((u) = O(\log n)\) 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 4.6.** Let \(u\) be a weighted word and let \(\alpha > 1\). An \(\alpha\)-suffix decomposition of \(u\) of size \(s\) is a sequence of suffixes \((u^l)^{1\leq l < s}\) 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 \((\alpha, t)\)-suffix sampling on \(u\) of size \(s\) is an \(\alpha\)-suffix decomposition of \(u\) of size \(s\) with \(t\) letter samplings on each suffix of the decomposition.

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

**Proposition 4.7.** Given as input an \((\alpha, t)\)-suffix sampling \(D_u\) on \(u\) of size \(s_u\) and another one \(D_v\) on \(v\) of size \(s_v\), there is an algorithm **Concatenate**\((D_u, D_v)\) computing an \((\alpha, t)\)-suffix sampling on the concatenated word \(u \cdot v\) of size at most \(s_u + s_v\) in time \(O(s_u)\).

Moreover, given as input an \((\alpha, t)\)-suffix sampling \(D_u\) on \(u\) of size \(s_u\), there is also an algorithm **Simplify**\((D_u)\) computing an \((\alpha, t)\)-suffix sampling on \(u\) of size at most \(2\log |u|/\log \alpha\) in time \(O(s_u)\).

**Proof.** We sketch those procedures. They are fully described in Algorithm 5 (Appendix A). For **Concatenate**, it suffices to do the following. For each suffix \(u^l\) of \(D_u\): (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 \(|v|/(|u| + |v|)\), for \(i = 1, \ldots, t\).

For **Simplify**, do the following. For each suffix \(u^l\) of \(D_u\), from \(l = s_u\) (the smallest one) to \(l = 1\) (the largest one): (1) replace all suffixes \(u^{l-1}, u^{l-2}, \ldots, u^m\) by 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 constructing online a suffix decomposition of small size. Starting with an empty suffix-sampling \(S\), simply concatenate \(S\) with the next processed letter \(a\) of the stream, and then simplify it. We formalize this together with functions **Concatenate** and **Simplify** in Algorithm 5 (Appendix A).

### 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 1 can be performed using our data structure. As a result our final algorithm is simply Algorithm 1 with the new data structure described in Algorithm 2 and the adapted operations defined in Algorithm 3. We will refer to the whole algorithm as Algorithm 1-3.

**Algorithm 2: Sketch for an unfinished peak**

<table>
<thead>
<tr>
<th>Parameters:</th>
<th>real (\varepsilon &gt; 0), integer (T \geq 1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data structure</td>
<td>for a weighted word (v \in \text{Prefix}(\Lambda_{\infty}^\varepsilon))</td>
</tr>
<tr>
<td></td>
<td>Weights of (v) and of its first letter (v(1))</td>
</tr>
<tr>
<td></td>
<td>Height of (v(1))</td>
</tr>
<tr>
<td></td>
<td>Boolean indicating whether (v) contains a pop symbol</td>
</tr>
<tr>
<td></td>
<td>((1 + \varepsilon))-suffix decomposition (v^1, \ldots, v^s) of (v) encoded by</td>
</tr>
<tr>
<td></td>
<td>Estimates (</td>
</tr>
<tr>
<td></td>
<td>(T) independent samplings (S_i) on (v^i) // see details below</td>
</tr>
<tr>
<td></td>
<td>with corresponding weights and heights</td>
</tr>
</tbody>
</table>


We now detail the methods, where we implicitly assume that each letter processed by the algorithm comes with its respective height and (exact or approximate) weight. They use functions \textbf{Concatenate} and \textbf{Simplify} described in Proposition 4.7 (and in details in Algorithm 5 of Appendix A), while adapting them.

Algorithm 3: Adaptation of Algorithm 1 using sketches

\begin{verbatim}
Adaptation of functions from Proposition 4.7

\textbf{Concatenate}(D_u, D_v)\ with\ an\ exact\ estimate\ of\ |v|\ is\ modified\ s.t.
the\ replacement\ probability\ is\ now\ \(|v|/(|u|_{\text{high}} + |v|)\)
and\ \(|u'|_{z_{\text{low}}} ← |u'|_{z} + |v|\)\ for\ \(z = \text{low}, \text{high}\)

\textbf{Simplify}(D_u)\ with\ \(\alpha = 1 + \epsilon'\)\ has\ now\ the\ relaxed\ condition\ \(|u'|_{\text{high}} \leq (1 + \epsilon')|u'|_{\text{low}}\)

Adaptation of operations on factors used in Algorithm 1

\textbf{Bullet-concatenation with a neutral letter:} \(v ← u \bullet a\)
\(b ←\) \text{last letter of} \(u\)
\(D_u ← \textbf{Concatenate}(D_u, a)\)
If \(b\) is neutral
\(R_{ba} ←\) set of \(ba\)-transitions
Delete suffix \(b\) from \(D_u\)
Replace every samples consisting of either \(a\) or \(b\) by \(R_{ab}\)
\(D_v ← \textbf{Simplify}(D_v)\)

\textbf{Compute relation:} \(R_v\)

Run the algorithm of Corollary 4.14 using samples in \(D_v\)

\textbf{Decomposition:} \(v_1 \cdot v_2 ← v\)
Find largest suffix \(v_i\) in \(D_v\) s.t. \(v_i \in \text{Prefix}(\Lambda_\infty)\) \(//\) i.e. \(v_i\) is \(v_2\)
\(D_{\forall v_1} ←\) suffixes \((v_i^j)_{i < i}\) with their samples
\(D_{v_2} ←\) suffix \(v_i\) with its samples and weight estimates: \(//\) for computing \(R_{v_2}\)
\(- (|v_i'|_{\text{high}}, |v_i'|_{\text{low}})\) when \(v_i' - 1\) and \(v_i\) differ by exactly one letter (then \(v_i' = v_2\))
\(- (|v_i' - 1|_{\text{high}}, |v_i'|_{\text{low}})\) otherwise

\textbf{Test:} \(|u_0| ≥ |v_2|/2\) using \(|v_2|_{\text{low}}\) instead of \(|v_2|\)

\textbf{Concatenation:} \(u_0 ← (v_1 \cdot R_{v_2}) \cdot u_0\)
\(D_{v'} ← (D_{\forall v_1}, R_{v_2})\) replacing each samples of \(D_{\forall v_1}\) in \(v_2\) by \(R_{v_2}\)
\(//\) The height of a sample determines whether it is in \(v_2\)
\(D_{u_0} ← \textbf{Simplify}(\textbf{Concatenate}(D_{v'}, D_{u_0}))\)
\end{verbatim}

In the next section, we show that the samplings \(S_{v'}\) are close enough to an \((1 + \epsilon')\)-suffix sampling on \(v'\). This lets us build an over sampling of an \((1 + \epsilon')\)-suffix sampling. We also show that it only requires a polylogarithmic number of samples. Then, we explain how to recursively apply an adaptation of Theorem 3.10 (with \(\epsilon'\)) in order to obtain the compressions at line 14 and 19 while keeping a cumulative error below \(\epsilon\). We now state our main result whose proof uses results from the following section.

\textbf{Theorem 4.8.} \textit{Let \(A\) be a VPA for \(L\) with \(m ≥ 2\) states, and let \(\epsilon, \eta > 0\). Then there is an \(\epsilon\)-streaming algorithm for \(L\) with one-sided error \(\eta\) and memory space \(O(m^{5}2^{5m^2}(\log^6 n)(\log 1/\eta)/\epsilon')\), where \(n\) is the input length.}

\textbf{Proof:} We use Algorithm 1 + 3 which the tester from Corollary 4.14 for the compressions at lines 14 and 19 of Algorithm 1. We know from Lemma 4.15 and Lemma 3.7 that it is enough to choose \(\epsilon' = \epsilon/(6 \log n)\), \(\eta' = \eta/n\), and Fact 4.1 gives us \(d = 2^m\). Therefore we need \(T = 2304m^42^{-2m^2}(\log^2 n)(\log 1/\eta)/\epsilon'\) independent \(k\)-factor samplings of \(n\) augmented by one, with \(k = 24m^2(\log n)/\epsilon\). Lemma 4.12 tells us that using twice more samples from our algorithm, that is for each \(S_{v'}\), is enough in order to over-sample them.

Because of the sampling variant we use, the size of each decomposition is at most \(96(\log^2 n)/\epsilon + O(\log n)\) by Lemma 4.12. 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 1 + 3. \(\square\)
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 $p\cdot u\overset{\epsilon}{\rightarrow} q$. We therefore consider the alphabet extension by any such relations $R$ with any weight. We define $\Sigma_Q$ to be the alphabet $\Sigma_-$ augmented by all such relations $R$, and we again extend the automaton and the language. Then, $\Lambda_Q$ is simply $\Lambda_1$ with $\Sigma_\ast = \Sigma_Q$.

**Proposition 4.9.** Each relation $R$ that Algorithm 1-3 produces is in $\Sigma_Q$.

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

**Proposition 4.10.** The slicing automaton $\hat{A}$ that we define over $(\Sigma_+ \cup \Sigma_- \cup \Sigma_Q)$ is $\hat{\Sigma}'$-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 4.11.** Let $v$ be an unfinished peak, and let $v^1, \ldots, v^s$ be the suffix decomposition maintained by the algorithm. The following is true:

1. $v^1, \ldots, v^s$ is a valid $(1 + \varepsilon')$-suffix decomposition of $v$.
2. For each letter $a$ of every $v^i$, and for every sample $s$, $\Pr[S_{i,j} = a] \geq |a|/|v^i|_{\text{high}}$.
3. Each $v^i$ satisfies $|v^i|_{\text{high}} - |v^i|_{\text{low}} \leq 2\varepsilon'|v^i|_{\text{low}}/3$.

**Proof.** Property (1) is guaranteed by the (modified) Simplify function used in Algorithm 3, which preserves even more suffixes than the original algorithm.

Properties (2) and (3) are proven by induction on the last letter read by Algorithm 1-3. Both are true when no symbol has been read yet.

We start with property (2). Let us first consider the case where we use bullet-concatenation after the last letter was read. Then for all $v^i$, the (modified) Concatenate function ensures $S_{i,j}$ becomes $a$ with probability $1/|v^i|_{\text{high}}$. Otherwise, $S_{i,j}$ remains unchanged and by induction $S_{i,j} = b$ with probability at least $(1 - 1/|v^i|_{\text{high}})|b|/(|v^i|_{\text{high}} - 1) = |b|/|v^i|_{\text{high}}$, for each other letter $b$ of $v^i$. If $a$ is a neutral symbol and $u_0$ ends with some $R \in \Sigma_Q$, any sample that would be either $R$ or $a$ is replaced by $R \cdot a$.

The other case is that some $R_{v_2}$ is computed at line 19 of Algorithm 1. In this case, $v^i$ is equal to some $(v_1 \cdot R_{v_2}) \cdot u_0$ concatenation. For each suffix $(v_1 \cdot v_2)^i$ in $D(v_1 \cdot v_2)$ containing $R_{v_2}$, we proceed in the same way with the Concatenate function, replacing any sample in $v_2$ with $R_{v_2}$. Now consider $v_2$ the largest suffix of $D(v_1 \cdot v_2)$ contained in $v_2$, and $v^i = R_{v_2} \cdot u_0$. We use the fact that Concatenate looks at $|v^i|_{\text{high}} \geq |u_0| + |R_{v_2}|$ for replacing samples. This means that we choose $R_{v_2}$ as a sample for $v^i$ with probability $(|v^i|_{\text{high}} - |u_0|)/|v^i|_{\text{high}} \geq |R_{v_2}|/|v^i|_{\text{high}}$, and therefore the property is verified.

We now prove property (3). If $v^i$ has just been created, it contains only one letter of weight 1, and obviously $|v^i|_{\text{low}} = |v^i|_{\text{high}} = |v^i|$. In addition, unless some $R_{v_2}$ has been computed at line 19 of Algorithm 1 when the last letter was read, then $|v^i|$ is only augmented by some exactly known $|a|$ or $|u_0|$ compared to the previous step. Therefore the difference $|v^i|_{\text{high}} - |v^i|_{\text{low}}$ does not change, and by induction it remains smaller than $2\varepsilon'|v^i|_{\text{low}}/3$ which can only increase. Now consider $R_{v_2}$ computed at line 19 and $v^i = R_{v_2} \cdot u_0$. We again consider $v_2$ for the largest suffix in the decomposition of $v_1 \cdot v_2$ that is contained within $v_2$, as used in Algorithm 3 and $v_2^{-1}$ is the suffix immediately preceding $v_2$ in that decomposition.

If $|v_2^{-1}|_{\text{high}} > (1 + \varepsilon')|v_2|_{\text{low}}$, then from the Simplify function, the difference between those two suffixes cannot be more than one letter, and then $v_2 = v_2$. Therefore, we have $|R_{v_2} \cdot u_0|_{\text{high}} = |v_2|_{\text{high}} + |u_0|$ and $|R_{v_2} \cdot u_0|_{\text{low}} = |v_2|_{\text{low}} + |u_0|$. We conclude by induction on $|v_2|$. 


We end with the case $|v_{1}^{t-1}|_{\text{high}} \leq (1 + \varepsilon')|v_{2}^{t}|_{\text{low}}$. By definition, $|R_{v_{2}} \cdot u_{0}|_{\text{high}} = |v_{2}^{t-1}|_{\text{high}} + |u_{0}|$ and $|R_{v_{2}} \cdot u_{0}|_{\text{low}} = |v_{2}^{t-1}|_{\text{low}} + |u_{0}|$. Therefore the difference $|v_{1}^{t}|_{\text{high}} - |v_{1}^{t-1}|_{\text{low}}$ is at most $\varepsilon'|v_{2}^{t}|_{\text{low}}$. Since the test at line 15 of Algorithm 1 (modified by Algorithm 2) was satisfied, we know that $|v_{2}^{t}|_{\text{low}} \leq 2|u_{0}|$, and finally $\varepsilon'|v_{2}^{t}|_{\text{low}} \leq 2\varepsilon(|v_{2}^{t}|_{\text{low}} + |u_{0}|)/3 \leq 2\varepsilon'|v_{1}^{t}|_{\text{low}}/3$, which concludes the proof.}

From this we prove that the $S_{v_{1}}$ 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 4.12.** Let $v, W$ be an unfinished peak with a sampling maintained by the algorithm. Then $W^\otimes 2$ over-samples an $(1 + \varepsilon')$-suffix sampling on $v$, and $W$ has size at most $144(\log |v|)(\log n)/\varepsilon + O(\log n)$. 

**Proof.** The first property is a direct consequence of property (1) and (2) in Proposition 4.11 as in the proof of Lemma 3.7.

The second is a consequence of the (modified) **Simplify** used in Algorithm 1: $D_{\text{temp}}$ is defined as the set of suffixes below with $m < l$ such that $|v_{m}^{|m|}|_{\text{high}} \leq (1 + \varepsilon')|v_{l}^{t}|_{\text{low}}$. Because **Simplify** deletes all but one elements from $D_{\text{temp}}$, it follows that $|v_{l}^{|l-2|}|_{\text{high}} > (1 + \varepsilon')|v_{l}^{t}|_{\text{low}}$. Now, from property (3) of Proposition 4.11 we have that $|v_{l}^{t}|_{\text{low}} \geq |v_{l}^{t-3}|_{\text{high}} - 2\varepsilon|v_{l}^{t}|_{\text{low}}/3 \geq (1 - 2\varepsilon/3)|v_{l}^{t}|_{\text{high}}$. Therefore we have that $|v_{l}^{t-2}|_{\text{high}} > (1 + \varepsilon')(1 - 2\varepsilon/3)|v_{l}^{t}|_{\text{high}}$

By successive applications, we obtain $|v_{l}^{t-6}|_{\text{high}} > (1 + \varepsilon')^{3}(1 - 2\varepsilon/3)^{3}|v_{l}^{t}|_{\text{high}}$. Now, as $|v_{l}^{t}|_{\text{high}} > |v_{l}^{t}|$ and $|v_{l}^{t}||v_{l}^{t}|_{\text{low}} \geq (1 - 2\varepsilon/3)|v_{l}^{t}|_{\text{high}}$ we have: $|v_{l}^{t-6}|/(1 - 2\varepsilon/3) > (1 + \varepsilon')^{3}(1 - 2\varepsilon/3)^{3}|v_{l}^{t}|$. Equivalently, $|v_{l}^{t-6}| > (1 + \varepsilon')^{3}(1 - 2\varepsilon/3)^{3}|v_{l}^{t}|$

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

**Robustness.** We first extend the notion of $\varepsilon$-approximation of words for a finite automaton (Definition 3.9) to any VPA when words are in $\Lambda Q$.

**Definition 4.13.** Let $R \subseteq Q^{2}$. Then $R (\varepsilon, \Sigma)$-approximates a balanced word $u \in (\Sigma_{+} \cup \Sigma_{-} \cup \Sigma Q)^{\ast}$ on $A$, if for all $p, q \in Q$: (1) $(p, q) \in R$ when $p \xrightarrow{u} q$; (2) $u$ is $(\varepsilon, \Sigma)$-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 $\Lambda Q$ instead of $\Lambda_{1}$. We present the result as an algorithm with an output $R$ as in Theorem 3.10. We also need to adapt to the sampling we have, where the suffixes do not exactly match the peaks we want to compress.

**Corollary 4.14.** Let $A$ be a VPA with $m \geq 2$ states and $\Sigma$-diameter $d \geq 2$. There is an algorithm that:

1. Take as input $\varepsilon', \eta > 0$ and $T$ $k$-factors of $z_{1}, \ldots, z_{T}$ of some weighted word $v \in \Lambda Q$, such that $T = 4kt, t = 2[4dm^{3}(\log 1/\eta)/\varepsilon']$ and $k = [4dm/\varepsilon']$;
2. Output a set $R \subseteq Q \times Q$ that $(\varepsilon', \Sigma)$-approximates $v$ on $A$ with bounded error $\eta$, when each factor $z_{i}$ come from an independent $k$-factor sampling on $\tilde{v}$.

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 $\tilde{v}'$ for each $z_{i}$ 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 3.10 for $\tilde{A}$ with restricted alphabet $\tilde{\Sigma}'$, where $\tilde{\Sigma}' = (\Sigma_{+} \cup \Sigma_{-} \cup \Sigma_{\infty})$. Remind that $A$ is $\Sigma'$-closed and its $\Sigma'$-diameter is the $\Sigma$-diameter of $A$. Lemma 3.7 gives us the sampling we need for Theorem 3.10 from our input, where we use that here $m/\varepsilon' > \log n$.

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 $R$. 

14
Algorithm 1 + 3. If \( u - 1 \) word \( \Sigma \) the chain from \( u \) are present in \( w \) of states for construction. still a (3 the example in Figure 2 this is the case for \( R \) to all the approximations eventually performed by the algorithm that did not involve a symbol already in \( \Sigma \), \( \text{bdist}_\Sigma (u, L) \leq \epsilon n \) with probability at least 1 – \( \eta \). From now on we assume that we are in this situation.

Let \( h = \text{Depth}(R_{\text{final}}) \). We will inductively construct sequences \( u_0 = u, \ldots, u_h = R_{\text{final}} \) and \( v_h = R_{\text{final}}, \ldots, v_0 \) such that for every \( 0 \leq l \leq h, u_l, v_l \in (\Sigma_+ \cup \Sigma_- \cup \Sigma_Q)^* \), \( \text{bdist}_\Sigma (u_l, v_l) \leq 3(h-l)\epsilon |u_l| \) and \( v_l \in L \). Furthermore, each word \( u_l \) will be the word \( w \) with some substitutions of factors by relations \( R \) computed by the tester. Therefore, \( \text{Depth}(u_l) \) is well defined and will satisfy \( \text{Depth}(u_l) = l \). This will conclude the proof using that \( \text{Depth}(u) \leq \log_3/2 n \) from Lemma 4.5. Indeed, since \( h \leq \text{Depth}(u) \), it will give us \( \text{bdist}_\Sigma (u, v_0) \leq 6\epsilon n \log n \leq \epsilon n \).

We first define the sequence \((u_l)_l\) (see Figure 2 for an illustration). Starting from \( u_0 = u \), let \( u_{l+1} \) be the word \( u_l \) where some factors in \( \Lambda_Q \) have been replaced by a \((3\epsilon', \Sigma)\)-approximation in \( \Sigma_Q \). These correspond to all the approximations eventually performed by the algorithm that did not involve a symbol already in \( \Sigma_Q \). Some approximations are eventually collapsed together into a single symbol by the \( \bullet \) operation (in the example in Figure 2 this is the case for \( R' \) for instance). Observe that after this collapse, the symbol is still a \((3\epsilon', \Sigma)\)-approximation. In particular, \( u_h = R_{\text{final}}, u_l \in (\Sigma_+ \cup \Sigma_- \cup \Sigma_Q)^* \) and \( \text{Depth}(u_l) = l \) 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 \( A \). Set \( v_h = R_{\text{final}} \) with an accepting run from \( p_m \) to \( q_f \) for some \((p_m, q_f) \in R_{\text{final}}(Q_m \times Q_f) \). Consider now some level \( l < h \). Then \( v_l \) is simply \( u_{l+1} \) 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 \( u_l \), or deleted by edit operations moving up from \( v_h \) to \( u_{l+1} \)).

Let \( w \in (\Lambda_Q)^* \) be one of those factors and \( R \in \Sigma_Q \) its respective \((3\epsilon', \Sigma)\)-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 \( p \rightarrow w'q \) and \( \text{bdist}_\Sigma (w, w') \leq 3\epsilon' |w| \), and annotate \( w' \) accordingly. By construction, the resulting word \( v_l \) satisfies \( v_l \in L \) and \( \text{bdist}_\Sigma (u_l, v_l) \leq 3(h-l)\epsilon' |u_l| \).

\[ \square \]
References


A Reservoir and suffix samplings

Algorithm 4: 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: |
| $i \leftarrow 1, a \leftarrow \text{Next}(u), \sigma \leftarrow |a|$ |
| $S \leftarrow t$ copies of $a$ |
| While $u$ not finished |
| $i \leftarrow i + 1, a \leftarrow \text{Next}(u), \sigma \leftarrow \sigma + |a|$ |
| For each $b \in S$ |
| Replace $b$ by $a$ with probability $|a|/\sigma$ |
| Output $S$ |

Algorithm 5: $\alpha$-Suffix Sampling

| Data structure: |
| // $D_u, D_v, D_{\text{temp}}$ stacks of items $(\sigma, b)$, one for each suffix |
| // of the decomposition where $\sigma$ encodes the weight and $b$ the $t$ samples |
| Code: |
| Concatenate$(D_u, D_v)$ |
| $(c_1, \ldots, c_t) \leftarrow$ all $t$ samples on $v$ (the largest suffix in $D_v$) |
| For each $(\sigma, b) \in S$ where $b = (b_1, \ldots, b_t)$ |
| Replace each $b_i$ by $c_i$ with probability $|v|/(|v| + \sigma)$ |
| Replace $(\sigma, b)$ by $(\sigma + |v|, b)$ |
| Append $D_v$ to the top of $D$ |
| Return $D$ |
| Simplify$(D_u)$ |
| $D \leftarrow D_u$ |
| For each $(\sigma, b) \in D$ from top to bottom |
| $D_{\text{temp}} \leftarrow$ elements $(\tau, c) \in D$ below $(\sigma, b)$ with $\tau \leq \alpha \sigma$ |
| Replace $D_{\text{temp}}$ in $D$ by the bottom most element of $D_{\text{temp}}$ |
| Return $D$ |
| Online-Suffix-Sampling |
| $D \leftarrow \emptyset$ |
| While $u$ not finished |
| $a \leftarrow \text{Next}(u)$ |
| Concatenate$(D, a)$ where $a$ encodes the suffix sampling $(|a|, (a, \ldots, a))$ |
| Simplify$(D)$ |
| Return $D$ |

Lemma A.1. Given a weighted word $u$ as a data stream and a parameter $\alpha > 1$, Online-Suffix-Sampling in Algorithm 5 constructs an $\alpha$-suffix sampling on $u$ of size at most $1 + 2 \lceil \log |u|/\log \alpha \rceil$.

B Missing proofs

Proof of Lemma 3.7. Denote by $\hat{W}$ the $k$-factor sampling on $\hat{u}$, and by $W$ some $4k$ independent copies of $W_k(u)$. For any $k$-factor $v$ of $\hat{u}$, we will show that the probability that $v$ is sampled by $\hat{W}$ is at most the probability that $v$ is a factor of an element sampled by $W$. 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 $\hat{W}$ equals the probability that $W_k(u)$ 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 \(\hat{\mathcal{W}}\) 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 \(\hat{\mathcal{W}}\) equals at most twice the probability that \(\mathcal{W}_k(u)\) samples the factor \(v\) augmented by one letter, as a \((\text{push, pop})\) pair in \(\hat{u}\) has weight 2 when a push has weight 1 in \(u\).

Hence, the probability that \(v\) is sampled by \(\hat{\mathcal{W}}\) 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 \(|\hat{u}| \geq |u|/2\), we get

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

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 \(\hat{\mathcal{W}}\) is at most the probability that \(v\) is a factor of an element sampled by \(\mathcal{W}\).

\[\square\]

**Proof of Fact 4.7**. 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 \(u \xrightarrow{p} q\), 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:

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

\[\square\]

**C A Tester for Weighted Regular Languages**

For the rest of this section, fix a regular language \(L\) recognized by some finite state automaton \(\mathcal{A}\) on \(\Sigma\) 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 $G_A$ is a maximal subset (w.r.t. inclusion) of vertices of $G_A$ such that for every $p_1, p_2$ in $C$ one has a path in $G_A$ from $p_1$ to $p_2$. The graph of components $G_A$ of $G_A$ describes the transition relation of $A$ on components of $G_A$: its vertices are the components and there is a directed edge $(C_1, C_2)$ if there is an edge of $G_A$ from a vertex in $C_1$ toward a vertex in $C_2$.

**Definition C.1.** Let $C$ be a component of $G_A$, let $\Pi = (C_1, \ldots, C_l)$ be a path in $G_A$.

- A word $u$ is $C$-compatible if there are states $p, q \in C$ such that $p \rightarrow u \rightarrow q$.
- A word $u$ is $\Pi$-compatible if it can be partitioned into $u = v_1a_1v_2 \ldots a_{l-1}v_l$ such that $p_i \rightarrow v_i \rightarrow q_i$, where $v_i$ is a factor, $a_i$ a letter, and $p_i, q_i \in C_i$.

- A sequence of factors $(v_1, \ldots, v_l)$ of a word $u$ is $\Pi$-compatible if they are factors of another $\Pi$-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 $\varepsilon$-far to every $\Pi$-compatible word. Note that it will not be used in the tester that we design in Theorem 3.10 for weighted regular languages, but only in Lemma C.3 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 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 $\Pi = (C_1, \ldots, C_l)$ in $G_A$, a parameter $0 < \kappa \leq 1$, and consider a weighted word $u$. We define a natural partition of $u$ according to $\Pi$, that we call the $\Pi$-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 $\Pi$, where most often $j = i + 1$, until the full word $u$ is processed. If we reach $j = l + 1$, we say that $u \kappa$-saturates $\Pi$ and the process stops. We now explain how we move on in $\Pi$. 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 $q \rightarrow q'$ with $q \in C_{j-1} \cup C_j$ and $q' \in C_j$, and $j = l + 1$ otherwise.

**Proposition C.2.** Let $0 < \kappa \leq \varepsilon/(2d^l)$. If $u$ is $\varepsilon$-far to every $\Pi$-compatible word, then $u \kappa$-saturates $\Pi$.

**Proof.** The proof is by contraposition. For this we assume that $u$ does not $\kappa$-saturate $\Pi$ and we correct $u$ to a $\Pi$-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 $\Pi$-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 $\Pi$. However, the word is not yet compatible with $\Pi$ 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 $A$.

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

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 C.3. Let \( u \) be a weighted word, let \( \Pi = C_1 \ldots C_l \) be a path in \( \mathcal{G}_A \). Let \( 0 < \kappa \leq \varepsilon / (2dl) \) and let \( \mathcal{W} \) denote the \( [2/\kappa] \)-factor sampling on \( u \). Then for every \( 0 < \eta < 1 \) and \( t \geq 2l(\log 1/\eta)/\kappa \), the probability \( P(u, \Pi) = \Pr_{(v_1, \ldots, v_t) \sim \mathcal{W}^\otimes t}[(v_1, \ldots, v_t) \text{ is } \Pi\text{-compatible}] \) satisfies \( P(u, \Pi) = 1 \) when \( u \) is \( \Pi \)-compatible, and \( P(u, \Pi) \leq \eta \) when \( u \) is \( \varepsilon \)-far for from being \( \Pi \)-compatible.

**Proof.** The first part of the theorem is immediate. For the second part, assume that \( u \) is \( \varepsilon \)-far from any \( \Pi \)-compatible word. For simplicity we assume that \( 2/\kappa \) and \( |u|/2 \) are integers. We first partition \( u \) according to \( \Pi \) and \( \kappa \). Then, Proposition C.2 tells us that \( u \) \( \kappa \)-saturates \( \Pi \). 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 \( 1 -(1 - \kappa/2)^l \).

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 \( 1 - (1 - \kappa)^l \).

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 \( 1 - (1 - \kappa)^l \).

By union bound, the probability that one of the above mentioned samples fails to occurs is at most \( l(1-\kappa)^l \leq \eta \). We assume now that they all occur, and we show that they form a \( \Pi \)-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 \( \Pi \)-compatible, and as a consequence \((v_1, \ldots, v_t)\) neither, so the result.

We can now conclude with the of Theorem 3.10

**Proof of Theorem 3.10** 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 \( \mathcal{G}_A \) of connected components of \( A \)

   (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 \( \eta \), the output set \( R \) only contains pairs \((p, q)\) such that there exists a path \( \Pi = C_1, \ldots, C_l \) on \( \mathcal{G}_A \) such that \( p \in C_1 \), \( q \in C_l \), and \( u \) is \( \Pi \)-compatible. In that case, there is an \( \varepsilon \)-close word \( v \) satisfying \( p \xrightarrow{v} q \).

Indeed, using \( l \leq m \) and Lemma C.3 with \( t, \kappa = \varepsilon/(2dm) \) and \( \eta' = \eta/2^m \), the samples satisfy \( P(u, \Pi) \leq \eta/2^m \), when \( u \) is not \( \Pi \)-compatible. Therefore, we can conclude using a union bound argument on all possible paths on \( \mathcal{G}_A \), which have cardinality at most \( 2^{m} \), that, with probability at least \( 1 - \eta \), there is no \( \Pi \) such that the samples are \( \Pi \)-compatible but \( u \) is not \( \Pi \)-compatible.

The structure of the tester is such that it has only more chances to reject a word that is not \( \Pi \)-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, \( A \) being \( \Sigma' \)-closed ensures that the notions of compatibility and saturation remain unchanged. Using the \( \Sigma' \)-diameter in Lemma C.3 (and therefore in Proposition C.2) let us use bridges in \( \Sigma'^r \) instead of \( \Sigma^r \) with weight at most \( d \).
D Figures for Algorithm 1
(a) Illustration of lines 10 to 11 from Algorithm 1

(b) Illustration of lines 13 to 16 from Algorithm 1

(c) Illustration of lines 17 to 19 from Algorithm 1

Figure 3: Illustration of Algorithm 1