Streaming Property Testing of Visibly Pushdown Languages *

Nathanaël François, Frédéric Magniez, Michel De Rougemont, Olivier Serre

Streaming Property Testing of Visibly Pushdown Languages*

Nathanaël François[1], Frédéric Magniez[2], Michel de Rougemont[3], and Olivier Serre[4]

1 **Fakultät für Informatik, TU Dortmund, Germany**
2 **CNRS, IRIF, Univ Paris Diderot, Sorbonne Paris-Cité, France**
3 **University of Paris II and IRIF, CNRS, France**
4 **CNRS, IRIF, Univ Paris Diderot, Sorbonne Paris-Cité, France**

Abstract

In the context of formal 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 memory space $\mathrm{poly}((\log n)/\varepsilon)$.

Our construction is done in three steps. First, we simulate a visibly pushdown automaton in one pass using a stack of small height but whose items can be of linear size. In a second step, those items are replaced by small sketches. Those sketches rely on a notion of suffix-sampling we introduce. This sampling is the key idea for taking benefit of both streaming algorithms and property testers in the third step. Indeed, the last step relies on a (non-streaming) property tester for weighted regular languages based on a previous tester by Alon *et al.* This tester can directly be used for streaming testing special cases of instances of Vpl that are already hard for both streaming algorithms and property testers. We then use it to decide the correctness of completed items, given their sketches, before removing them from the stack.

1998 ACM Subject Classification F.1.2 Modes of Computation, F.4.3 Formal Languages

Keywords and phrases Streaming Algorithm, Property Testing, Visibly Pushdown Languages

Digital Object Identifier 10.4230/LIPIcs.ESA.2016.43

1 Introduction

We focus on streams representing data with both a linear ordering and a hierarchically nested matching of items. Data with such dual linear-hierarchical structure arise in various context, *e.g.* in semi-structured data management when handling HTML/XML documents or in program analysis when considering executions of recursive programs. Regular languages, as recognized by finite state automata, revealed a natural and successful tool to express properties of streams but lack the ability to handle the hierarchical structure. Context-free languages easily capture the latter but turn out to be too expressive hence, quickly lead to intractable complexity. In contrast, visibly pushdown languages (Vpl) [6] while encompassing regular languages, enjoy most of its good properties and permit to handle

* Partially supported by the French ANR projects ANR-12-BS02-005 (RDAM) and ANR-14-CE25-0017 (AGREG). A full version of this paper is available at `http://arxiv.org/abs/1505.03334`.

data with both a linear and a hierarchical structure. In the context of semi-structured documents, they are closely related with regular languages of unranked trees as captured by hedge automata: indeed, 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.* [18] for an overview on automata and logic for unranked trees). In databases, this word encoding of XML document is known as SAX representation: the document is a linear sequence of text characters, along with a hierarchically nested matching of open-tags with closing tags. Numerous popular subclasses of XML documents (*e.g.* those satisfying a given DTD specifications) are subclasses of VPL. In program analysis, VPA permit to capture natural properties of execution traces of recursive finite-state programs. For such programs, desirable specifications are expressed on the call-stack (*e.g.* "a module A should be invoked only if the module B belongs to the call-stack"): such properties can be expressed in the temporal logic of calls and returns (CaRet) [5, 4] that itself is captured by VPA. Hence, the analysis of execution traces boils down to check membership in a VPL.

Therefore, the study of VPL is central to understand how massive semi-structured data (*e.g.* large semi-structured documents or execution traces) can be analyzed by sublinear algorithms, such as streaming algorithms and property testers.

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 into three groups: 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 algorithms [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.* [22]) have only a sequential access to their input, on which they can perform a single 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 [20, 7]. However, even for simple VPL, any randomized streaming algorithm with p passes requires memory $\Omega(n/p)$, where n is the input size [17].

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 sufficient modifications of the input, such as moves of arbitrarily large factors, it has been shown that any context-free language becomes testable with a constant number of queries [19, 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 [23, 24].

Faced by the intrinsic hardness of VPL in both streaming and property testing, we study the complexity of *streaming property testers* of formal languages, a model of algorithms combining both approaches. Such testers were historically introduced for testing specific problems (groupedness) [14] relevant for network data. They were later studied in the context of testing the insert/extract-sequence of a priority-queue structure [12]. We extend these studies to classes of problems. 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 a natural notion of distance for VPL, the *balanced-edit distance*, which refines the edit distance on *balanced words* (where for each push symbol there is a matching pop symbol at the same height of the stack, and conversely). It can be interpreted as the edit distance on trees when trees are encoded as balanced words. Neutral symbols 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 the edit distance.

In Section 3, we first design an exact algorithm that maintains a small stack but whose items can be of linear size as opposed to the standard simulation of a pushdown automaton which usually has a stack of possible linear size but with constant size items. In our algorithm, stack items are prefixes of some peaks (which we call unfinished peaks), where a *peak* is a balanced factor whose push symbols appear all before the first pop symbol. 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 6**) is rewriting the input word as a peak formed by potentially a linear number of intermediate peaks, but with only a logarithmic number of nested peaks.

In Section 4, for the case of a single peak, we show how to sketch the current unfinished peak of our algorithm. 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 (**Theorem 16**). Since our reduction induces weights on the letters of the new input word, we need a tester for weighted regular languages. Such a property tester has previously been devised in [24] extending constructions for unweighted regular languages [1, 23]. However, we consider a slightly simpler construction that could be of independent interest. As a consequence we get a streaming property tester with polylogarithmic memory for recognizing peak instances of any given VPL (**Theorem 17**), a task already hard for streaming algorithms and property testers (**Fact 8**).

In Section 5, we construct our main tester for a VPL L given by some VPA. For this we introduce a more involved notion of sketches made of a polylogarithmic number of samples. They are based on a new notion of suffix sampling (**Definition 18**). This sampling consists in a decomposition of the string into 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 yet know where they start. Our sampling can be used to perform an approximate computation of the compressed relation by our new property tester of weighted regular languages which we also used 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 22**). Then we prove the robustness of our algorithm: words that are ε-far from L are rejected with high probability (**Lemma 23**). As a consequence, we get a one-pass streaming ε-tester for L with one-sided error η and memory space $\mathrm{O}(m^5 2^{3m^2}(\log n)^6(\log 1/\eta)/\varepsilon^4)$, where m is the number of states of a VPA recognizing L (**Theorem 20**).

2 Definitions and Preliminaries

Let $\mathbb{N}^*$ be the set of positive integers, and for any $n \in \mathbb{N}^*$, 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^*. We denote by $u \cdot v$ (or simply uv) the word obtained by concatenating u and v. 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. A *factor* of u is a word $u[i,j] = u(i)u(i+1)\cdots u(j)$ with $1 \leq i \leq j \leq n$. 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 there is an injection $f : S \mapsto S'$ such that for each factor $v \in S$, $v \leq f(v)$.

2.1 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 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, 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.

Our algorithms will be based on 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*. In fact the general case $k > 1$ simply reduces to $k = 1$. Indeed, simply observe that k-factor sampling can be obtained from letter sampling by sampling on the first letters of the factors and online completing any sampled letter to produce its associated k-factor. Therefore, from now on, we only focus on how to perform letter samplings, that we implicitly extend to samplings on k-factors when

Algorithm 1 Reservoir Sampling

```
Input: Data stream u, Integer t > 1 standing for the number of samples
Data structure:
  σ ← 0 // Current weight of the processed stream
  S ← empty multiset // Multiset of sampled letters
Code:
a ← Next(u), σ ← |a|
S ← t copies of a
While u not finished
    a ← Next(u), σ ← σ + |a|
    For each b ∈ S
        Replace b by a with probability |a|/σ
Output S
```

required. In particular, without further constraints, letter sampling can be implemented using a standard reservoir sampling (see Algorithm 1).

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. We introduce the notion of *over-sampling* to formalize this:

▶ **Definition 1.** 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 for each factor v of u, we have $\Pr(\exists v' \in S_2$ such that v is a factor of $v') \geq \Pr(\exists v' \in S_1$ such that v is a factor of $v')$.

2.2 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-transition*. A word u is accepted if $q_{in} \xrightarrow{u} q_f$ for some $q_{in} \in Q_{in}$ and $q_f \in Q_f$. 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 *regular language*. 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 $\min\{|u| : p \xrightarrow{u} q$ 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 $u \in \Sigma^*$ if and only if $p \xrightarrow{u'} q$ 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 so 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 [21] and of nested word automata [6].

▶ **Definition 2.** 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 $i < k < j$. By extension, the height of $u(i)$ is $\mathrm{height}(u[1, i-1])$ when $u(i)$ is a push symbol, and $\mathrm{height}(u[1, i])$ otherwise.

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 the notions for finite automata above the Σ'*-diameter* of $\mathcal{A}$ (or simply diameter) and the notion of $\mathcal{A}$ being Σ'*-closed*. These definitions only consider balanced words.

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

2.3 Streaming Property Testers

Assume we have, for any $\varepsilon > 0$, a criterion to declare that an input u is ε-far from a language L. An ε-tester for L accepts all inputs in L with probability 1 and rejects with high probability all inputs ε-far from 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 3.** 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 u 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

Even if we only focus on the space complexity of streaming testers, all our streaming testers have polylogarithmic (in n/ε) time per processing letter.

For a distance d between words, we say that a word u is ε-far from a language L if $d(u, v) > \varepsilon|u|$ for every $v \in L$, *i.e.* the ε-neighborhood of u does not intersect L. Hence, any distance on words leads to a notion of streaming property tester. Remark that any ε-tester for some distance d_1 turns out to be also a $(c\varepsilon)$-tester for any other distance d_2 such that $d_2 \leq cd_1$, where $c > 0$ is some constant.

2.4 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 to weighted words.

Given a word u, we define two possible *edit operations*: the *deletion* of a letter in position i with corresponding cost $|u(i)|$, and its converse operation, the *insertion* where we also select a weight for the new $u(i)$. Note that, for simplicity, we drop the usual substitution operation, leading to a possible multiplicative factor of 2 in the resulting distance. This is not an issue when designing streaming property testers as observed above. The *(standard) edit distance* $\mathsf{dist}(u, v)$ between two weighted words u and v is defined as the minimum total cost of a sequence of edit operations changing u to v. All letters that have not been inserted nor deleted must keep the same weight. For a restricted set of letters Σ', define $\mathsf{dist}_{\Sigma'}(u, v)$ when insertions (but not deletions) are restricted to letters in Σ' (this makes $\mathsf{dist}_{\Sigma'}$ not symmetric).

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* $\mathsf{bdist}(u, v)$ 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 $\mathsf{bdist}_{\Sigma'}(u, v)$. We omit Σ' when $\Sigma' = \Sigma$.

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. Note that since balanced-edit distance is larger than the standard edit distance, our testers will also be valid for that distance.

3 Exact Algorithm

Fix a Vpa $\mathcal{A}$ recognizing some Vpl L on $\Sigma = \Sigma_+ \cup \Sigma_- \cup \Sigma_=$. In this section, we design an exact streaming algorithm that decides whether an input belongs to L. Algorithm 2 maintains a stack of small height but whose items can be of linear size. In Section 5, we replace stack items by appropriate small sketches.

3.1 Notations and Algorithm Description

Call a *peak* a sequence of push symbols followed by an equal number of pop symbols, with possibly intermediate neutral symbols, *i.e.* an element of the language $\Lambda = \bigcup_{j\geq 0}((\Sigma_=)^* \cdot \Sigma_+)^j \cdot (\Sigma_=)^* \cdot (\Sigma_- \cdot (\Sigma_=)^* \cdot \Sigma_-)^j$. One can compress any peak $v \in \Lambda$ by the set $R_v = \{(p, q) : p \xrightarrow{v} q\}$ of the v-transitions, and consider R_v as a new neutral symbol with weight $|v|$. In fact, for the purpose of the analysis of our algorithm, we augment neutral symbols by many more relations for which $\mathcal{A}$ remains Σ-closed. Indeed, we allow any relation R of any weight such that, when $(p, q) \in R$, there is a $v \in \Lambda$ such that $p \xrightarrow{v} q$, but that v could be different for every $(p, q) \in R$. For the rest of the paper, they will be the only symbols with weight potentially larger than 1.

▶ **Definition 4.** Let Σ_Q be $\Sigma_=$ augmented by all letters 'R' encoding a relation $R \subseteq Q \times Q$ such that for every $(p, q) \in R$ there is a balanced word $u \in \Sigma^*$ with $p \xrightarrow{u} q$. In addition we allow any weight $|R| \geq 1$ for those letters. Let Λ_Q be Λ where $\Sigma_=$ is replaced by Σ_Q.

Algorithm 2 Exact Tester for a VPL

```
1  Input: Balanced data stream u
2  Data structure:
3  Stack ← empty stack // Stack of items v with v ∈ Prefix(Λ_Q)
4  u_0 ← ∅ // u_0 ∈ Prefix(Λ_Q) is a suffix of the processed part u[1,i] of u
5        // with possibly some factors v ∈ Λ_Q replaced by R_v
6  R_temp ← {(p,p)}_{p∈Q} //Set of transitions for the max. prefix of u[1,i] in Λ_Q
7  Code:
8  While u not finished
9    a ← Next(u) //Read and process a new symbol a
10   If a ∈ Σ_+ and u_0 has a letter in Σ_- // u_0 · a ∉ Prefix(Λ_Q)
11     Push u_0 on Stack, u_0 ← a
12   Else u_0 ← u_0 · a
13   If u_0 is balanced // u_0 ∈ Λ_Q: compression
14     Compute R_{u_0} the set of u_0-transitions
15     If Stack = ∅, then R_temp ← R_temp ∘ R_{u_0}, u_0 ← ∅
16        // where ∘ denotes the composition of relations
17     Else Pop v from Stack, u_0 ← v · R_{u_0}
18   Let (v_1 · v_2) ← top(Stack) s.t. v_2 is maximal and balanced // v_2 ∈ Λ_Q
19   If |u_0| ≥ |v_2|/2  // u_0 is big enough and v_2 can be replaced by R_{v_2}
20     Compute R_{v_2} the v_2-transitions, Pop v from Stack, u_0 ← (v_1 · R_{v_2}) · u_0
21 If (Q_in × Q_f) ∩ R_temp ≠ ∅, Accept; Else Reject // R_temp = R_u
```

We then write $p \xrightarrow{R} q$ whenever $(p, q) \in R$, and extend $\mathcal{A}$ and L accordingly. Of course, our notion of distance will be solely based on the initial alphabet Σ. If $R_1, R_2 \subseteq Q \times Q$ are two relations on Q we define their composition $R_1 \circ R_2$ to be $\{(x, z) \mid \exists y \text{ s.t. } (x, y) \in R_1 \text{ and } (y, z) \in R_2\}$.

A general balanced input instance u will consist of many nested peaks. However, we will recursively replace each factor $v \in \Lambda_Q$ by R_v with weight $|v|$.

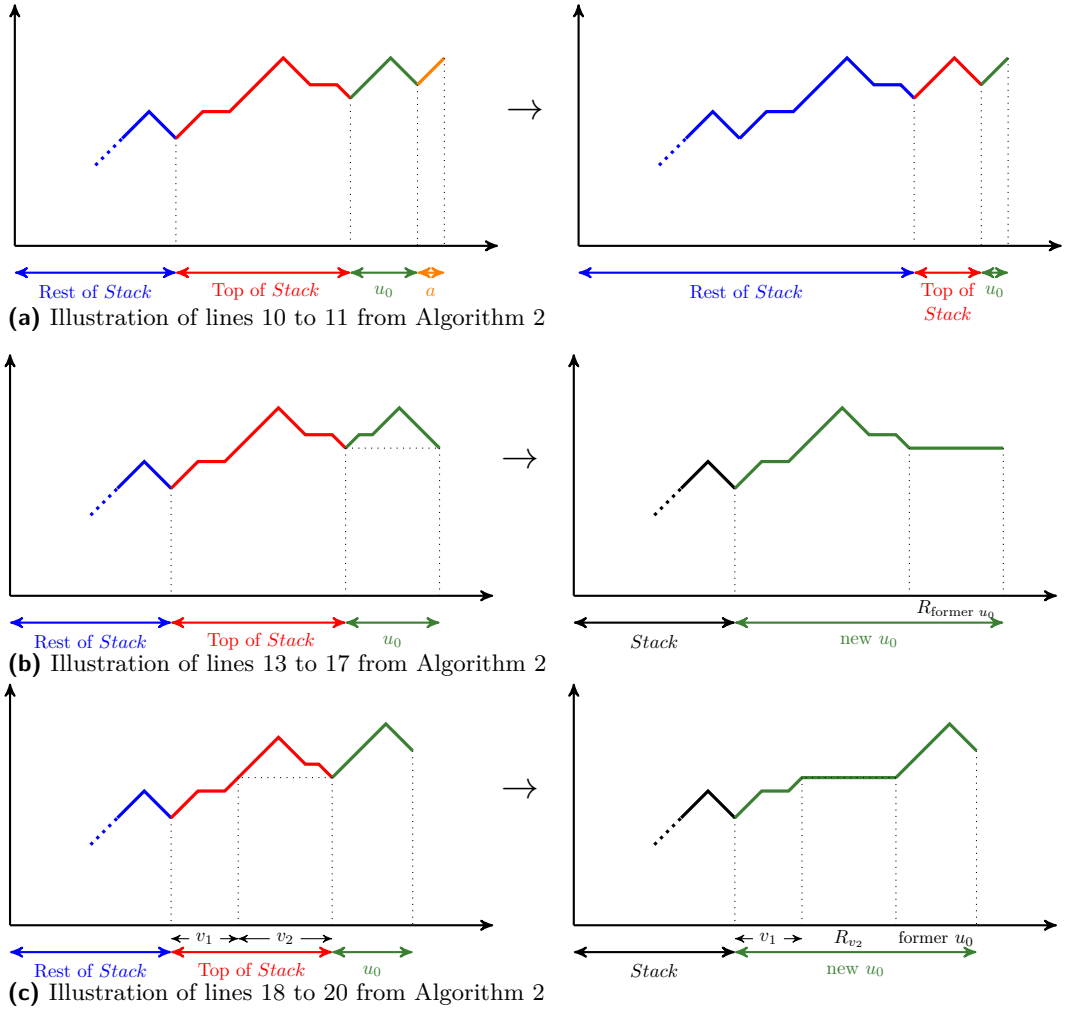
Denote by $\text{Prefix}(\Lambda_Q)$ the language of prefixes of words in Λ_Q. While processing the prefix $u[1, i]$ of the data stream u, Algorithm 2 maintains a suffix $u_0 \in \text{Prefix}(\Lambda_Q)$ of $u[1, i]$, that is an unfinished peak, with some simplifications of factors v in Λ_Q by their 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 $R_{\texttt{temp}} \subseteq Q \times Q$ that is the set of transitions for the maximal prefix of $u[1, i]$ in Λ_Q. When the stream is over, the set $R_{\texttt{temp}}$ is used to decide whether $u \in L$ or not.

When a push symbol a comes after a pop sequence, $u_0 \cdot a$ is no longer in $\text{Prefix}(\Lambda_Q)$ hence, Algorithm 2 puts u_0 on the stack of unfinished peaks (see lines 10 to 11 and Figure 1a) and u_0 is reset to a. In other situations, it adds a to u_0. In case u_0 becomes a word in Λ_Q (see lines 13 to 17 and Figure 1b), Algorithm 2 computes the set of u_0-transitions $R_{u_0} \in \Sigma_Q$, and adds R_{u_0} to the previous unfinished peak that is retrieved on top of the stack and becomes the current unfinished peak; in the special case where the stack is empty it simply updates $R_{\texttt{temp}}$ by taking its composition with R_{u_0}.

3.2 Algorithm Analysis

For each factor v constructed in Algorithm 2, we define $\text{Depth}(v)$ as the number of processed nested peaks in v. This is formalized as follows.

▶ **Definition 5.** For each factor constructed in Algorithm 2, Depth is defined dynamically by $\text{Depth}(a) = 0$ when $a \in \Sigma$, $\text{Depth}(v) = \max_i \text{Depth}(v(i))$ and $\text{Depth}(R_v) = \text{Depth}(v) + 1$.



(a) Illustration of lines 10 to 11 from Algorithm 2

(b) Illustration of lines 13 to 17 from Algorithm 2

(c) Illustration of lines 18 to 20 from Algorithm 2

Figure 1 Illustration of Algorithm 2.

In order to bound the size of the stack, Algorithm 2 considers the maximal balanced suffix v_2 of the topmost element $v_1 \cdot v_2$ of the stack and, whenever $|u_0| \geq |v_2|/2$, it computes the relation R_{v_2} and continues with a bigger current peak starting with v_1 (see lines 18 to 20 and Figure 1c). 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. This can be proved by a direct inspection of Algorithm 2.

▶ **Proposition 6.** *Algorithm 2 accepts exactly when $u \in L$, while maintaining a stack of at most $\log |u|$ items.*

We state that Algorithm 2, when processing an input u of length n, considers at most $O(\log n)$ nested peaks, that is $\text{Depth}(v) = O(\log n)$ for all factors constructed in Algorithm 2.

▶ **Lemma 7.** *Let v be the factor used to compute R_v at line either 14 or 20 of Algorithm 2. Then $|v(i)| \leq 2|v|/3$, for all i. Moreover, for any factor w constructed by Algorithm 2 it holds that $\text{Depth}(w) = O(\log |w|)$.*

4 The Special Case Of Peaks

We now consider restricted instances consisting of a *single peak*. For these instances, Algorithm 2 never uses its stack but u_0 can be of linear size. We show how to replace u_0 by a small random sketch in order to get a streaming property tester using polylogarithmic memory. In Section 5, this notion of sketch will be later extended to obtain our final streaming property tester for general instances.

4.1 Hard Peak Instances

Peaks are already hard for both streaming algorithms and property testers. Indeed, consider the language Disj $\subseteq \Lambda$ over alphabet $\Sigma = \{0, 1, \overline{0}, \overline{1}, a\}$ and defined as 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 non-approximation streaming algorithms comes for a standard reduction to Set-Disjointness. The hardness for property testing algorithms is a corollary of a similar result due to [25] for parenthesis languages with two types of parentheses.

▶ **Fact 8.** *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 $\varepsilon > 0$, we will show that languages 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 little memory.

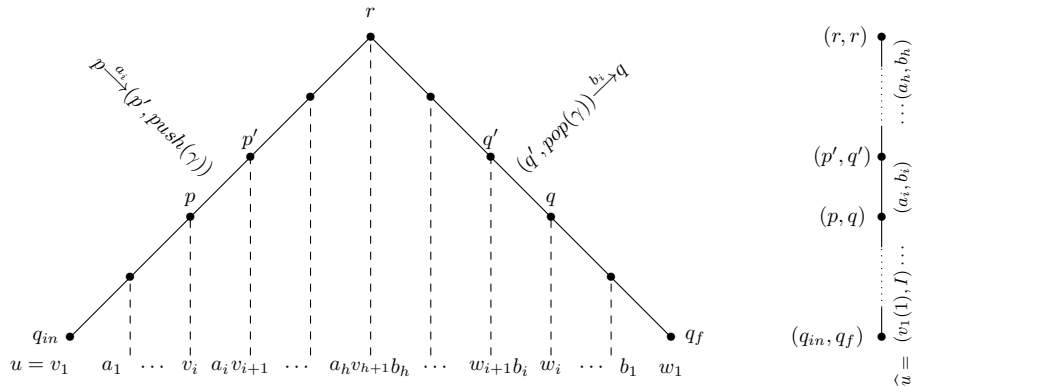
4.2 Slicing Automaton

Observe that Algorithm 2 will never use the stack in the case of a single peak. After Algorithm 2 has processed the i-th letter of the data stream, u_0 contains $u[1,i]$ where the eventual initial sequence of neutral symbols has been removed. We will show how to compute R_{u_0} at line 14 using a standard finite state automaton without any stack.

Indeed, 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 $\Sigma_=$ encoding the relation set $\{(p,p) : p \in Q\}$. For a word $v \in \Sigma_=^l$, write $[v, \mathrm{I}]$ for the word $(v(1), \mathrm{I}) \cdot (v(2), \mathrm{I}) \cdots (v(l), \mathrm{I})$, and similarly $[\mathrm{I}, v]$. Consider a weighted word of the form $u = \left(\prod_{i=1}^{j} v_i \cdot a_i\right) \cdot v_{j+1} \cdot \left(\prod_{i=j}^{1} b_i \cdot w_i\right)$, where $a_i \in \Sigma_+$, $b_i \in \Sigma_-$, and $v_i, w_i \in \Sigma_=^*$. Then the *slicing* of u (see Figure 2) is the word $\widehat{u}$ over the alphabet $\widehat{\Sigma} = (\Sigma_+ \times \Sigma_-) \cup (\Sigma_= \times \{\mathrm{I}\}) \cup (\{\mathrm{I}\} \times \Sigma_=)$ defined by $\widehat{u} = \left(\prod_{i=1}^{j} [v_i, \mathrm{I}] \cdot [\mathrm{I}, w_i] \cdot (a_i, b_i)\right) \cdot [v_{j+1}, \mathrm{I}]$, and which has weight $\left(\sum_{i=1}^{j} \lambda(v_i) + \lambda(w_i) + 2\right) + \lambda(v_{j+1})$.

▶ **Definition 9.** Let $\mathcal{A} = (Q, \Sigma, \Gamma, Q_{in}, Q_f, \Delta)$ be a VPA. Define $\widehat{Q} = Q \times Q$, $\widehat{Q_{in}} = Q_{in} \times Q_f$, $\widehat{Q_f} = \{(p,p) : p \in Q\}$. 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 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 2 Slicing of a word $u \in \Lambda$.

This construction will be later used in Section 5 for weighted languages. In that case, we define the weight of a letter in $\widehat{u}$ by $|(a,b)| = |a| + |b|$, with the convention that $|I| = 0$. Moreover, we write $\widehat{\Sigma_Q}$ for the alphabet obtained similarly to $\widehat{\Sigma}$ using Σ_Q instead of $\Sigma_=$. Note that the slicing automaton $\widehat{\mathcal{A}}$ defined on $\widehat{\Sigma_Q}$ is $\widehat{\Sigma}$-closed and has $\widehat{\Sigma}$-diameter at most $2m^2$ where $m = |Q|$. Indeed, the slicing automaton has m^2 states and every letter in $\widehat{\Sigma}$ has weight at most 2, hence the shortest path from two states (when exists) has weight at most $2m^2$. In particular, it directly implies the following.

▶ **Proposition 10.** *Let $v \in \Lambda$ be such that $(p,q) \xrightarrow{\widehat{v}} (p',q')$. There is $w \in \Lambda$ such that $|w| \leq 2m^2$ and $(p,q) \xrightarrow{\widehat{w}} (p',q')$.*

▶ **Lemma 11.** *If $\mathcal{A}$ is a* VPA *accepting L, then $\widehat{\mathcal{A}}$ is accepts $\widehat{L} = \{\widehat{u} : u \in L \cap \Lambda\}$.*

4.3 Random Sketches

We are now ready to build a tester for $L \cap \Lambda$. To test a word u we use a property tester for the regular language $\widehat{L}$. Regular languages are known to be ε-testable for the Hamming distance with $O((\log 1/\varepsilon)/\varepsilon)$ non-adaptive queries on the input word [1], that is queries that can all be made simultaneously. Those queries define a small random sketch of u that can be sent to the tester for approximating R_u. Since the Hamming distance is larger than the edit distance, those testers are also valid for the latter distance. Observe also that, for $v_1, v_2 \in \Lambda_Q$, we have $\mathsf{bdist}(v_1, v_2) \leq 2\mathsf{dist}(\widehat{v_1}, \widehat{v_2})$. The only remaining difficulty is to provide to the tester an appropriate sampling on $\widehat{u}$ while processing u.

We will proceed similarly for the general case in Section 5, but then we will have to consider weighted words. Therefore we show how to sketch u in that general case already. Indeed, the tester of [1] was simplified for the edit distance in [23], and later on adapted for weighted words in [24]. We consider here an alternative approach that we believe to be simpler, but slightly less efficient than the tester of [24].

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.

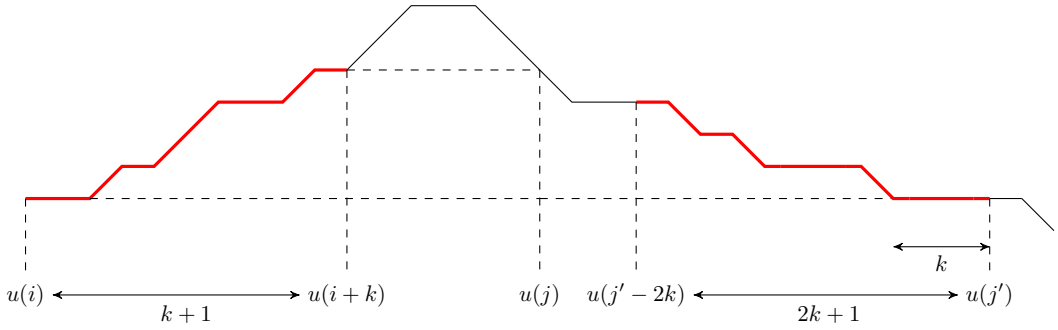


Figure 3 The sampling $\mathcal{W}_k(u)$ from Definition 12: sample is in red.

Let $u \in \Lambda$ and let $u[i, i+k]$ be a factor that contains at least one push symbol. Call i_1 (resp. i_2) the smallest (resp. largest) integer such that $i_1 \geq i$ (resp. $i_2 \leq i+k$) and $u(i_1)$ (resp. $u(i_2)$) is a push symbol. Then the *matching pop sequence* of $u[i, i+k]$ is defined as $u[j_1, j_2]$ where $u(j_1)$ (resp. $u(j_2)$) is the matching pop symbol of $u(i_1)$ (resp. $u(i_2)$).

▶ **Definition 12.** For a weighted word $u \in \Lambda_Q$, denote by $\mathcal{W}_k(u)$ the sampling over subwords of u constructed as follows (see Figure 3):

(1) Sample a factor $u[i, i+k]$ of u with probability $|u(i)|/|u|$.

(2) If $u[i, i+k]$ contains at least one push symbol, let $u[j, j']$ be the matching pop sequence of $u[i, i+k]$, extended by the first k neutral symbols after the last pop symbol, if any. Add $u[\max(j, j'-2k), j']$ to the sample (hence, some matching pops of $u[i, i+k]$ may not belong to $u[\max(j, j'-2k), j']$).

Let us stress that in the above definition the weight of letters only matter in (1), and not in (2) which cares about matching push and pop symbols, which are of weight 1. One consequence is that one can design a randomized streaming algorithm performing this sampling.

▶ **Fact 13.** *There is a randomized streaming algorithm with memory* $\mathrm{O}(k + \log n)$ *which, given* k *and* u *as input, samples* $\mathcal{W}_k(u)$.

▶ **Lemma 14.** *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-sample the* k*-factor sampling on* $\widehat{u}$.

We can now give an analogue of the property tester for weighted regular languages in $L \cap \Lambda_Q$. For that, we use the following notion of approximation.

▶ **Definition 15.** Let $R \subseteq Q^2$ and $\varepsilon \geq 0$. 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) If $p \xrightarrow{u} q$, then $(p, q) \in R$;

(2) If $(p, q) \in R$, there is a word v such that $\mathsf{dist}_\Sigma(u, v) \leq \varepsilon|u|$ and $p \xrightarrow{v} q$.

Our tester is going to be robust enough in order to consider samples that do not exactly match the peaks we want to compress.

▶ **Theorem 16.** *Let* $\mathcal{A}$ *be a* VPA *with* $m \geq 2$ *states and* Σ*-diameter* $d \geq 2$. *Let* $\varepsilon > 0$, $\eta > 0$, $t = 2\lceil 4dm^3(\log 1/\eta)/\varepsilon \rceil$, $k = \lceil 4dm/\varepsilon \rceil$ *and* $T = 4kt$. *There is an algorithm that, given* T *random subwords* $z_1, \ldots, z_T$ *of some weighted word* $v \in \Lambda_Q$, *such that each* z_i *comes from an independent sampling* $\mathcal{W}_k(v)$, *outputs a set* $R \subseteq Q \times Q$ *that* (ε, Σ)*-approximates* v *on* $\mathcal{A}$ *with bounded error* η.

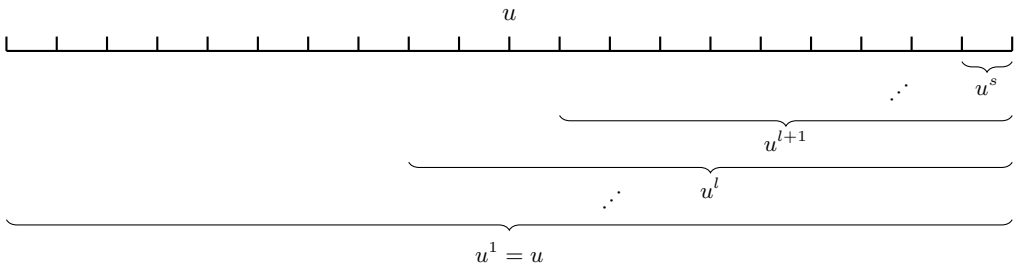


Figure 4 An α suffix decomposition of u of size s. For every l, either $|u^l| \leq \alpha|u^{l+1}|$, or $u^l = a \cdot u^{l+1}$ where a is a letter.

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 $\mathcal{W}_k(v')$ for each z_i instead, except that R now provides a $(3\varepsilon, \Sigma)$-approximation. Last, each sampling can be replaced by an over-sampling.

As a consequence we get our first streaming tester for $L \cap \Lambda$.

▶ **Theorem 17.** *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 $\mathrm{O}((m^8 \log(1/\eta)/\varepsilon^2)(m^3/\varepsilon + \log n))$, where n is the input length.*

Proof. We use Algorithm 2 where we replace the current factor u_0 by $T = 4kt$ independent samplings $\mathcal{W}_k(u_0)$. We know that such samplings can be computed using memory space $\mathrm{O}(k + \log n)$ by Fact 13. By Proposition 10, the slicing automaton has $\widehat{\Sigma}$-diameter d at most $2m^2$. Therefore, from Theorem 16, taking $t = 4\lceil 4dm^3(\log 1/\eta)/\varepsilon \rceil$ and $k = \lceil 4dm/\varepsilon \rceil$ leads to the desired conclusion. ◀

5 Algorithm With Sketching

5.1 Sketching Using Suffix Samplings

We now describe the sketches used by our main algorithm. They are based on the generalization of the random sketches described in Section 4.3. Moreover, they rely on a notion of suffix sampling, that ensures a good letter sampling on each suffix of a data stream. Recall (see Section 2.1) that a letter sampling on a weighted word u samples a random letter $u(i)$ (with its position) with probability $|u(i)|/|u|$, and that a sampling on k-factors can be derived from a letter sampling. Therefore we will sample k-factors using an (α, t)-suffix sampling.

▶ **Definition 18.** Let u be a weighted word and let $\alpha > 1$. An *α-suffix decomposition of u of size s* (see Figure 4) 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.

We observe that (α, t)-suffix samplings can be either concatenated or compressed as stated below.

▶ **Proposition 19.** *Given an (α, 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 (α, t)-suffix sampling on the concatenated word $u \cdot v$ of size at most $s_u + s_v$ in time $\mathrm{O}(s_u)$.*

Data Structure 3 Sketch for an unfinished peak

```
1 Parameters: real ε′ > 0, integers T ≥ 1 and k ≥ 1.
2 Data structure for a weighted word v ∈ Prefix(Λ_Q)
3   Weights of v and of its first letter v(1)
4   Height of v(1)
5   Boolean indicating whether v contains a pop symbol
6   (1 + ε′)-suffix decomposition v^1, ..., v^s of v encoded for l = 1, ..., s by
7      Estimates |v^l|_low and |v^l|_high of |v^l|
8      T independent samplings S_{v^l} on k-factors of v^l //See details
9         below with corresponding weights and heights
```

Moreover, given an (α, t)-suffix sampling D_u on u of size s_u, there is an algorithm ***Simplify***(D_u) *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 **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 polylogarithmic size. Starting with an empty suffix-sampling S, simply concatenate S with the next processed letter a of the stream, and then simplify it.

5.2 Final Algorithm

Our final algorithm is a modification of Algorithm 2: in particular it approximates relations R_v (in the spirit of Definition 15) by elements in Σ_Q, instead of exactly computing them. Let us stress that even if some R_v is approximated by an R that does not correspond to any R_u, one has $R \in \Sigma_Q$, which means that for any $(p, q) \in R$, there is a balanced word $u \in \Sigma^*$ depending on (p, q) with $p \xrightarrow{u} q$.

To mimic Algorithm 2 we need to encode (compactly) each unfinished peak v of the stack and u_0: for that we use the data structure described in Data Structure 3. Our final algorithm, Algorithm 4, is simply Algorithm 2 with this new data structure and corresponding adapted operations, where $\varepsilon' = \varepsilon/(6 \log n)$, $T = 4608 m^4 2^{2m^2} (\log^2 n)(\log 1/\eta)/\varepsilon^2$ and $k = 24 m 2^{m^2} (\log n)/\varepsilon$.

The methods are described in Algorithm 4, where we implicitly assume that each letter processed by the algorithm comes with its respective height and (exact or approximate) weight. They use functions **Concatenate** and **Simplify** described in Proposition 19, while adapting them.

In the next section, we show that the samplings S_{v^l} are close enough to an $(1 + \varepsilon')$-suffix sampling on v^l. This lets us build an over-sampling of an $(1 + \varepsilon')$-suffix sampling. We also show that it only requires a polylogarithmic number of samples. Then, we explain how to recursively apply the tester from Theorem 16 (with ε') in order to obtain the compressions at line 14 and 20 while keeping a cumulative error below ε. We now state our main result whose proof relies on Lemmas 22 and 23.

Algorithm 4 Adaptation of Algorithm 2 using sketches

Run Algorithm 2 using Data structure 3 with the following adaptations:
Adaption of functions from Proposition 19
`Concatenate`(D_u, D_v) `with an exact estimate of` $|v|$ `is modified s.t.`
`the replacement probability is now` $|v|/(|u|_{\texttt{high}} + |v|)$
`and` $|u^l \cdot v|_z \leftarrow |u^l|_z + |v|$`, for` $z = \texttt{low}, \texttt{high}$
`Simplify`(D_u) `with` $\alpha = 1 + \varepsilon'$ `has now relaxed condition` $|u^m|_{\texttt{high}} \leq (1+\varepsilon')|u^l|_{\texttt{low}}$
`Online-Suffix-Sampling is unchanged except for doing` k`-factor sampling.`
Adaption of operations on factors used in Algorithm 2
Compute relation: R_v
`Run the algorithm of Theorem 16 using samples in` D_v
Decomposition: $v_1 \cdot v_2 \leftarrow v$
`Find largest suffix` v^i `in` D_v `s.t.` $v^i \in \mathrm{Prefix}(\Lambda_Q)$ *//i.e.* v^i *is in* v_2
$D_{v|v_1} \leftarrow$`suffixes` $(v^l)_{l<i}$ `with their samples`
$D_{v_2} \leftarrow$`suffix` v^i `with its samples & weight estimates` *//to compute* R_{v_2}
-$(|v^i|_{\texttt{high}}, |v^i|_{\texttt{low}})$ `when` v^{i-1} `and` v^i `differ by only one letter (then` $v^i = v_2$`)`
-$(|v^{i-1}|_{\texttt{high}}, |v^i|_{\texttt{low}})$ `otherwise`
Test: $|u_0| \geq |v_2|/2$ `using` $|v_2|_{\texttt{low}}$ `instead of` $|v_2|$
Concatenation: $u_0 \leftarrow (v_1 \cdot R_{v_2}) \cdot u_0$
$D_{v'} \leftarrow (D_{v|v_1}, R_{v_2})$ `replacing each sample of` $D_{v|v_1}$ `in` v_2 `by` R_{v_2}
// The height of a sample determines whether it is in v_2
$D_{u_0} \leftarrow$ `Simplify(Concatenate`$(D_{v'}, D_{u_0})$`)`

▶ **Theorem 20.** *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.*

5.3 Final Analysis

As Algorithm 4 may fail at various steps, the relations it considers may not correspond to any word. However, each relation R that it produces is still in Σ_Q. Furthermore, the slicing automaton $\widehat{\mathcal{A}}$ over $\widehat{\Sigma_Q}$ is $\widehat{\Sigma}$-closed. Fact 21 below bounds the $\widehat{\Sigma}$-diameter of $\widehat{\mathcal{A}}$ (which is equal to the Σ-diameter of $\mathcal{A}$) by 2^{m^2}. For simpler languages, as those coming from a DTD, this bound can be lowered to m.

▶ **Fact 21.** *Let $\mathcal{A}$ be a* VPA *with m states. Then the Σ-diameter of $\mathcal{A}$ is at most 2^{m^2}.*

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

▶ **Lemma 22** (Stability lemma). *Let v be an unfinished peak with $\mathcal{W}_1, \mathcal{W}_2$ two of the T samplers maintained by Algorithm 4. Then the joint process $(\mathcal{W}_1, \mathcal{W}_2)$ over-samples an $(1+\varepsilon')$-suffix sampling on v, and the decomposition has size at most $144(\log |v|)(\log n)/\varepsilon + \mathrm{O}(\log n)$.*

Using the tester from Theorem 16 for computing each R, we get our robustness lemma.

▶ **Lemma 23** (Robustness lemma). *Let $\mathcal{A}$ be a* VPA *recognizing L and let $u \in \Sigma^n$. Let $R_{\texttt{final}}$ be the final value of $R_{\texttt{temp}}$ in Algorithm 4.*

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 $1 - \eta$.

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. doi:10.1145/200836.200880.

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. doi:10.1016/0022-0000(93)90044-W.

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. doi:10.1007/s00453-002-0959-4.

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 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.

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

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

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

21 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.

22 S. Muthukrishnan. Data streams: Algorithms and applications. *Foundations and Trends in Theoretical Computer Science*, 1(2):117–236, 2005. doi:10.1561/0400000002.

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

24 A. Ndione, A. Lemay, and J. Niehren. Sublinear DTD validity. In *Proc. of 19th International Conference on Language and Automata Theory and Applications*, pages 739–751, 2015.

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.