Electronic Colloquium on Computational Complexity, Report No. 94 (2010)

Streaming algorithms for some problems in log-space

Ajesh Babu, Nutan Limaye, Girish Varma

Tata Institute of Fundamental Research, Mumbai, India
ajesh, nutan, girish@tcs.tifr.res.in

Abstract. In this paper, we give streaming algorithms for some problems which are known to be in deterministic log-space, when the number of passes made on the input is unbounded. If the input data is massive, the conventional deterministic log-space algorithms may not run efficiently. We study the complexity of the problems when the number of passes is bounded.

The first problem we consider is the membership testing problem for deterministic linear languages, DLIN. Extending the recent work of Magniez et al.[12](to appear in STOC 2010), we study the use of fingerprinting technique for this problem. We give the following streaming algorithms for the membership testing of DLINs: a randomized one pass algorithm that uses $O(\log n)$ space (one-sided error, inverse polynomial error probability), and also a p-pass $O(n/p)$-space deterministic algorithm. We also prove that there exists a language in DLIN, for which any p-pass deterministic algorithm for membership testing, requires $\Omega(n/p)$ space. We also study the application of fingerprinting technique to visibly pushdown languages, VPLs.

The other problem we consider is, given a degree sequence and a graph, checking whether the graph has the given degree sequence, Deg-Seq. We prove that, any p-pass deterministic algorithm that takes as its input a degree sequence, followed by an adjacency list of a graph, requires $\Omega(n/p)$ space to decide Deg-Seq. However, using randomness, for a more general input format: degree sequence, followed by a list of edges in any arbitrary order, Deg-Seq can be decided in $O(\log n)$ space. We also give a p-pass, $O((n \log n)/p)$-space deterministic algorithm for Deg-Seq.

1 Introduction

Conventional computational models such as Turing machines do not restrict the number of passes on the input. We wish to understand the conventional space bounded models of computation when the number of passes made on the input is restricted. The model of computation where the number of passes is bounded has been studied extensively [1, 15]. In this paper we study problems that are already known to be in deterministic log-space with no restrictions on the number of passes and re-analyze their complexity for a bounded number of passes.

ISSN 1433-8092

The paper is divided into two parts. In the first part, we consider the membership problem for subclasses of context-free languages, CFLs. The membership problem, for a fixed language L is: given a string w, checking whether $w \in L$. We consider some subclasses of CFLs for which log-space algorithms are already known. We study the number of passes versus space and randomness trade-offs for these problems.

Recently, Magniez et al. [12] studied the membership problem for Dyck_2 (the set of balanced strings over two types of parentheses). They proved that there is a $O(\sqrt{n} \log n)$ space, one pass randomized algorithm for Dyck_2. They also proved that any randomized algorithm that makes one pass on the input must use $\tilde{\Omega}(\sqrt{n})$[1] space.

Using their ideas of finger-printing the stack, we study the problem of membership testing of deterministic linear languages, DLIN, the class of languages generated by deterministic linear grammars for which the right hand side of every rule has at most one non-terminal. The language Dyck_2 does not belong to DLIN. The membership testing for languages in DLIN is in deterministic log-space when there is no restriction on the number of passes [8]. The most obvious log-space algorithm makes multiple passes on the input. We ask whether adding randomness leads to an algorithm with fewer passes. We prove the following theorem:

Theorem 1. *For any $L \in \mathsf{DLIN}$, there exists a constant c and a randomized one pass algorithm A_L that uses $O(\log n)$ space such that $\forall w \in \Sigma^*$ if $w \in L$, $\Pr[A_L(w) = 1] = 1$ and if $w \notin L$, $\Pr[A_L(w) = 1] \leq \frac{1}{n^c}$*

We also analyze the deterministic streaming complexity of the membership problem of DLIN. We prove the following two theorems:

Theorem 2. *There is a deterministic p-passes, $O(n/p)$-space algorithm for membership testing for languages in DLIN.*

Theorem 3. *Any deterministic algorithm that makes p-passes over the input, will require $\Omega(n/p)$ space for membership testing for languages in DLIN.*

We analyze the algorithm used to prove Theorem 1, and apply it to a subclass of visibly pushdown languages, VPLs. VPLs were defined by Mehlhorn [13] and Alur et al. [2]. Their membership testing was studied in [13, 5, 6]. Brahnmuhl et al. gave the first log-space algorithm for membership testing of VPLs and later Dymond settled its complexity to

[1] A function $f(n)$ is said to be $\tilde{\Omega}(\sqrt{n})$ if there exists a constant $\alpha > 0$ such that $f(n) = \Omega\left(\sqrt{n}(\log n)^{\alpha}\right)$

NC^1. VPLs are defined over tri-partitioned alphabet $\Sigma = \Sigma_c \cup \Sigma_r \cup \Sigma_l$. A grammar form for VPLs was given in [3]. A sub-class of VPLs, well-matched VPLs was defined in [2]. It is believed that an efficient algorithm for membership testing of wVPLs can help in XML type checking. For large XML documents, it may be inefficient to store the whole document to perform the type checking. It is therefore interesting to consider the model where the membership testing for languages in wVPL can be done using fewer passes on the input.

We answer the question in a more restricted setting. We consider a class of languages generated by grammars more restrictive than the grammar form for wVPLs. We denote such grammars by rest-VPG.

Let w be an input string over the input alphabet of L. And let $n = |w|$. An index $i \in [n]$ is said to be a *reversal* if $w[i-1] \in \Sigma_r$ and $w[i] \in \Sigma_c \cup \Sigma_l$.

Let $rev(L, n)$ be defined as maximum value of $rev(w)$ over all strings $w \in \Sigma^n$ such that $w \in L$. We denote this by $rev(n)$ if L is clear from the context.

Theorem 4. *For any L generated by rest-*VPG*, there exists a constant c and a randomized one pass algorithm A_L that uses $O(rev(L, n) \log n)$ space such that $\forall w \in \Sigma^*$ if $w \in L$, $\Pr[A_L(w) = 1] = 1$ and if $w \notin L$, $\Pr[A_L(w) = 1] \leq \frac{1}{n^c}$*

In the second part of the paper we consider the following graph problem:

Degree-Sequence, Deg-Seq:
Given: A degree sequence and a directed graph
Check: Do vertices $v_1, v_2, \ldots, v_n$ have out-degrees $d_1, d_2, \ldots, d_n$, respectively?

This problem is known to be in log-space (in fact in TC^0(see for example [16])). The most obvious log-space algorithm for this problem makes multiple passes on the input, as in the case of membership testing of DLIN.

It has been observed [4, 7] that the complexity of graph problems changes drastically depending on the order in which the input is presented to the streaming algorithm. If the input to Deg-Seq is such that the degree of a vertex along with all the edges out of that vertex are listed one after the other, then checking whether the graph has the given degree sequence is trivial. If the degrees sequence is listed first, followed by the adjacency list of the graph then we observe that a one pass deterministic algorithm needs $\Omega(n)$ space to compute Deg-Seq. For a more general ordering of the input where the degree sequence is followed by a list of edges in an arbitrary order, we prove the following theorem:

Theorem 5. *If the input is a degree sequence followed by a list of edges in an arbitrary order, then* Deg-Seq *can be solved*

- *by a one pass, $O(\log n)$ space randomized streaming algorithm such that if vertices $v_1, v_2, \ldots, v_n$ have out-degrees $d_1, d_2, \ldots, d_n$, respectively, then the algorithm accepts with probability* 1 *and rejects with probability* $1 - n^{-c}$*, otherwise.*
- *by a p-passes, $O((n \log n)/p)$-space deterministic streaming algorithm.*

The rest of the paper is organized as follows: In Section 2 we prove Theorem 1 and Theorem 2 . Here we also give a randomness efficient version of the algorithm used to prove Theorem 1. Theorem 3 is proved in Section 2.5. In Section 3, we modify the algorithm from Section 2.2 and prove Theorem 4. In Section 4, we analyze the complexity of Deg-Seq and prove Theorem 5.

2 Streaming algorithms for membership testing of DLINs

In this section we give streaming algorithms for the membership testing of languages in DLIN (Theorem 1). (See [9], for the basic definitions regarding context-free grammars, sentential forms and derivations.) We start with some definitions.

Definition 1 ([10, 8]). *A* ***deterministic linear CFG****,* DL-CFG*, is a CFG (Σ, N, P, S) for which, every production is of the form $A \longrightarrow a\omega$, where $a \in \Sigma$ and $\omega \in (N \cup \epsilon)\Sigma^*$ and for any two productions, $A \longrightarrow a\omega$ and $B \longrightarrow b\omega'$, if $A = B$ then $a \neq b$, where $a, b \in \Sigma$ and $\omega, \omega' \in N\Sigma^*$*

Definition 2. ***Deterministic linear CFLs****,* DLIN*, is the class of languages for which there exists a* DL-CFG *generating it.*

The algorithm for membership testing of DLINs is a modification of the algorithm of [12]. The working of our algorithm is easier to follow for a specific language in DLIN, generated by the following grammar: $S \longrightarrow (S) \mid [S] \mid (\ ) \mid [\ ]$, denoted as 1-turn-$\mathsf{Dyck}_2$.

2.1 Streaming algorithm for membership testing of 1-turn-Dyck$_2$

The strategy to check membership in 1-turn-Dyck_2 is to come up with a polynomial from the given string, such that this polynomial is the zero polynomial if and only if the string is in 1-turn-Dyck_2. The algorithm uses this underlying polynomial, and maintains its evaluation at randomly chosen elements from a large enough field. If the string is in 1-turn-Dyck_2 then the evaluation is zero. Otherwise the evaluation is non-zero with high probability.

Let us define a function, $type : \Sigma \rightarrow \{x_0, x_1\}$, where x_0, x_1 are formal variables. Let $type(\ '(' \ ) = type(\ ')' \ ) = x_0$ and $type(\ '[' \ ) = type(\ ']' \ ) = x_1$. Given a string $w \in \Sigma^*$, let $n = |w|$. Assuming n is even [2] we construct a bi-variate polynomial $q(x_0, x_1)$ as:

$$q(x_0, x_1) = \sum_{i=1}^{n/2} type(w[i])^i - \sum_{i=n/2+1}^{n} type(w[i])^{n-i-1}$$

It is easy to observe that this polynomial is the zero polynomial if and only if the string is in 1-turn-Dyck_2.

The algorithm maintains an evaluation of the polynomial $q(x_0, x_1)$ using the following $Hash$ function. Given a string over the alphabet $\Sigma = \{(, [,),]\}$ and n, the length of the string, $Hash_{\alpha,\beta} : \Sigma \times [n] \rightarrow \mathbb{F}_p$ is defined as follows [3], where p is a prime, $\mathbb{F}_p$ is a prime field and $\alpha, \beta \in \mathbb{F}_p$:

$$Hash_{\alpha,\beta}(w[i], i) = \begin{cases} \alpha^i \ (mod\ p) & \text{if } w[i] = \ '(' \\ \beta^i \ (mod\ p) & \text{if } w[i] = \ '[' \\ -\alpha^{n-i+1} \ (mod\ p) & \text{if } w[i] = \ ')' \\ -\beta^{n-i+1} \ (mod\ p) & \text{if } w[i] = \ ']' \end{cases}$$

The algorithm can now be described as:

Algorithm 1 Randomized one pass algorithm for 1-turn-Dyck_2

```
Choose $\alpha, \beta$ uniformly at random from $\mathbb{F}_p$. Let $Sum$ be initialized to 0.
for $i = 1$ to $n$ do
    $Sum \leftarrow Sum + Hash_{\alpha,\beta}(w[i], i)\ (mod\ p)$
end for
Output 'yes' if $Sum$ is zero and 'no' otherwise.
```

If the input w is in 1-turn-Dyck_2 then the value of Sum becomes zero for any choice of α, β, since the polynomial itself is zero. The multi-variate version of the Schwartz-Zippel Lemma (see for example [14], page 165) tells that if $w \notin$ 1-turn-Dyck_2 then Sum is non-zero with high probability (for appropriately chosen p).

[2] If n is not even, we reject the string.

[3] In [12], only one type of parenthesis is hashed. Our algorithm here is randomness inefficient. But this type of finger-printing helps for the general algorithm for DLIN

Since the algorithm only stores Sum and the current index i, the space requirement is $O(\log n)$ and the algorithm makes one pass on the input string.

2.2 A randomized streaming algorithm for DLIN

We observe a property of 1-turn-Dyck_2 which can be generalized for DLIN to obtain an efficient algorithm.

Observation 6 *For any string w in 1-turn-Dyck_2 and $i \in [\frac{n}{2}]$, the letter at location $w[i]$ completely determines the letter at location $w[n-i+1]$, where $n = |w|$.*

For a language in the class DLIN the observation may not be immediately applicable. For example, $S \to aBc$; $B \to aSb$ is a valid DL-CFG but on seeing the letter a at location $i < \frac{n}{2}$, we do not know whether to expect b or c at $w[n-i+1]$. However, something very similar to Observation 6 applies to DL-CFG.

In this section, for the sake of simplicity, we restrict ourselves to DL-CFG grammars that have rules of the form $A \to uBv$, where $|u| = |v| = 1$. It is easy to see that this can be generalized to all of DL-CFG. Let L be a DLIN. Any string in L is produced by repeated application of rules corresponding to the DL-CFG of L, say G_L. The sentential forms arising in the derivation of any $w \in L$ have at most one non-terminal in them. Let the current sentential form be uAu', where $u, u' \in \Sigma^*$, u and u' are prefix and suffix of w respectively, and $A \in N$. Let the next terminal symbol after u in w be a. Suppose there is a rule $A \to aBc$ then determinism forces that there is no other rule $A \to a'B'c'$ such that $a' = a$. Therefore, if the rule for A is to be applied and the letter to be generated is a, then the next sentential form is $uaBcu'$, i.e. A and a uniquely determine c.

Observation 7 *Let w be a string generated by a DL-CFG G that have rules only of the form $A \to uBv$, where $|u| = |v| = 1$. Let $i \in [\frac{n}{2}]$. The letter at location $w[i]$ and the rule that needs to be applied to produce it, completely determine the letter at location $w[n-i+1]$, where $n = |w|$.*

While processing $w[i]$, we add a monomial to the sum which is *expected* to be subtracted on reading $w[n-i+1]$.

In the case of 1-turn-Dyck_2 we could write down an explicit polynomial to be computed for a given input string. Here, the polynomial computation is more involved.

Let L be a DLIN, generated by a DL-CFG, $G_L = (N, \Sigma, P, S)$. We describe the multi-variate polynomial that we come up with for the given input string such that this polynomial is the zero polynomial if and only if the given string is in L.

Let $\Sigma = \{a_1, a_2, \ldots, a_k\}$. Let $\{x_1, x_2, \ldots, x_k\}$ be formal variables. Let *type* and *next* be two functions such that $type : \Sigma \times N \rightarrow \{x_1, \ldots, x_k\}$ and $next : \Sigma \times N \rightarrow N$.

If $A \rightarrow a_i B a_j$ is a rule in the grammar G_L, then $type(a_i, A)$ and $next(a_i, A)$ are defined to be x_j and B respectively. They are undefined otherwise. The determinism of the grammar ensures that for given A and a_i, x_j and B are unique.

We define the polynomial inductively using an extra variable *var* also maintained inductively.

Let $q_0(x_1, \ldots, x_k) = 0$ and $var_0 = S$. For $i \leq \frac{n}{2}$, we define:

$$\begin{aligned} q_i(x_1, \ldots, x_k) = & \quad q_{i-1}(x_1, \ldots, x_k) + (type(w[i], var_{i-1}))^i \\ var_i \quad = & \quad next(w[i], var_{i-1}). \end{aligned}$$

For $i > \frac{n}{2}$, define $q_i(x_1, \ldots, x_k)$ as $q_{i-1}(x_1, \ldots, x_k) - (map(w[i]))^{n-i+1}$, where $map(a_i) = x_i$.

It is easy to see that $q_n(x_1, \ldots, x_k)$ is the zero polynomial if and only if the given string is in L.

As in the case of 1-turn-Dyck$_2$, we will implicitly compute this polynomial. The idea is to maintain an evaluation of this polynomial at randomly points $\alpha_1 \cdots \alpha_k$ chosen from a enough field, $\mathbb{F}_p$.

We are now ready to describe our algorithm for membership test of DLIN.

Algorithm 2 Randomized one pass algorithm

```
Pick $\alpha_1, \alpha_2, \ldots, \alpha_k$ uniformly at random from $\mathbb{F}_p$, where $k = |\Sigma|$.
$Sum \leftarrow 0$
$var \leftarrow S$
for $i = 1$ to $n/2$ do
    Let $index$ be $j$ if $type(w[i], var) = x_j$
    $Sum \leftarrow (Sum + (\alpha_{index})^i)(mod\ p)$
    $var \leftarrow next(w[i], var)$
end for
for $i = n/2 + 1$ to $n$ do
    Let $index$ be $j$ if $map(w[i]) = x_j$
    $Sum \leftarrow (Sum - (\alpha_{index})^{n-i+1})(mod\ p)$
end for
Output 'yes' if $Sum$ is zero and 'no' otherwise.
```

It is easy to see that the above algorithm can be generalized when the DL-CFG grammar has rules of the form $A \rightarrow aBv$, where $|v| = 0$ or $|v| > 1$. We need to maintain an extra variable to keep track of the power to which α_js will be raised. We denote this variable by h. Suppose the rule $A \rightarrow aBv$ such that $|v| > 1$ is being applied at step $i \leq n/2$, then obtain the type of each letter inside v. Say the types are $x_{t_1}, x_{t_2}, \ldots x_{t_l}$ where $l = |v|$. Now add $\sum_{j=1}^{l} \alpha_{t_j}^{h+(l-j)}$ to the sum and set h to $h + l$. For $|v| = 0$, h and Sum remain unchanged.

The algorithm makes one pass on the input. We know that $q_n(x_1, x_2, \ldots, x_k)$ is non-zero when the given string is not in L. However, the evaluation may still be zero. Note that degree of $q_n(x_1, x_2, \ldots, x_k)$ is less than or equal to n. If the field size is chosen to be $n^{1+c} \leq p \leq n^{2+c}$, then due to Schwartz-Zippel lemma [14] the probability that Sum is zero but $q_n(x_1, x_2, \ldots, x_k)$ is non-zero is at most n/p which is at most n^{-c}. The amount of randomness used will be $(c|\Sigma| \log n)$. The algorithm needs to keep track of $\alpha_1, \ldots, \alpha_k$ and the value of Sum. Therefore, the amount of space used is also $(c|\Sigma| \log n)$.

2.3 Randomness Efficient version

In this section we will give a randomness efficient version of Algorithm 2 by reducing the membership testing problem for DLINs to membership testing in 1-turn-Dyck$_2$in a streaming way.

Definition 3 (Streaming Reduction). *Fix two alphabets Σ_1 and Σ_2. A problem P_1 is $f(n)$-streaming reducible to a problem P_2 with space $s(n)$ if for every input $x \in \Sigma_1^n$, there exists $y_1 y_2 \ldots y_n$ with*

$$y_i \in \cup_{i=1}^{f(n)} \Sigma_2^i \cup \{\epsilon\}$$

such that [4]*:*

- *y_i can be computed from x_i using space $s(n)$.*
- *From a solution of P_2 on input y, a solution on P_1 on input x can be computed in space $s(n)$.*

Note that our definition is a slight modification of the definition from [12].

In [12], it was observed that the membership testing of Dyck$_k$ $O(\log k)$-streaming reduces to membership testing of Dyck$_2$. We show that the membership testing for any language in DLIN $O(1)$-streaming reduces to

[4] In [12], y_i s are assumed to be of fixed length, i.e. from $\Sigma_2^{f(n)}$

membership testing in 1-turn-Dyck_k, where k is the alphabet size of the language.

As the membership testing for Dyck_2 requires only $(c \log n)$ random bits as opposed to $(ck \log n)$-bits used in Algorithm 2, this gives a randomness efficient algorithm. The main result in this section is stated below:

Theorem 8. *The membership testing for any language in* DLIN *$O(\log |\Sigma|)$-streaming reduces to membership testing in 1-turn-Dyck_2, where Σ is the alphabet of the language.*

Proof. Say L is a fixed DLIN. Given an input w, the streaming reduction outputs a string w' so that w' is in 1-turn-Dyck_k if and only if w belongs to L. In order to do this, we modify a step from Algorithm 2 where the *expected* sums are built. Let Σ_2 be $\{a_1, a_2, \ldots, a_k\}$. Let us define $\Sigma_2' = \{\overline{a_1}, \overline{a_2}, \ldots, \overline{a_k}\}$ such that $\overline{a_i}, a_i$ form a matching pair.

The streaming reduction can be described as follows: In the description of Algorithm 3, we have assumed that the grammar has rules of the form $A \rightarrow uBv$, where $|u| = |v| = 1$. It is easy to see that this can be generalized to all of DL-CFG.

Algorithm 3 Streaming reduction to 1-turn-Dyck_k

```
$var \leftarrow S$
for $i = 1$ to $n/2$ do
    Let $index$ be $j$ if $type(w[i], var) = x_j$
    Output $\overline{a_j}$
    $var \leftarrow next(w[i], var)$
end for
for $i = n/2 + 1$ to $n$ do
    Output $w[i]$
end for
```

2.4 A deterministic multi-pass algorithm

In this section we give a deterministic multi-pass algorithm for the membership testing of any language in DLIN (Theorem 2).

From Theorem 8, we know that any language in DLIN $O(\log |\Sigma|)$-streaming reduces to 1-turn-Dyck_2. Thus it suffices to give a p-passes, $O(n/p)$-space deterministic algorithm for membership testing of 1-turn-Dyck_2.

The algorithm divides the string into blocks of length $n/2p$. Let the blocks be called B_0, B_1, $\ldots$, B_{2p-1} from left to right. (i.e. $B_i =$

$w[i(n/2p)+1]\ w[i(n/2p)+2]\ \ldots w[(i+1)n/2p]$.) The algorithm considers a pair of blocks $(B_j,B_{2p-(j+1)})$ during the jth pass. Using the stack explicitly, the algorithm checks whether the string formed by the concatenation of B_j and $B_{2p-(j+1)}$ is balanced. If it is balanced, it proceeds to the next pair of blocks. The number of passes required is p. Each pass uses $O(n/p)$ space and the algorithm is deterministic.

2.5 Lower bound for the multi-pass streaming algorithm

Now we prove Theorem 3 which states that any deterministic algorithm that makes p passes over the input, will require $\Omega(n/p)$ space for membership testing of any language DLIN.

Proof. We reduce the two party communication problem of testing equality of strings to membership testing in 1-turn-Dyck_2. Given two string $x, y \in \{0,1\}^n$ we obtain a string $z \in \{(,[,],)\}^{2n}$ such that $\text{EQUALITY}(x,y) = 1$ if and only if $z \in$ 1-turn-Dyck_2. Take $z = x^{'}\overline{y}^{'}$ where $x^{'}$ is the string obtained by replacing $0,1$ of x by $[,($ respectively and $\overline{y}^{'}$ is the string obtained by first taking the reverse of y, and then replacing $0,1$ by $],)$ respectively. Clearly $\text{EQUALITY}(x,y) = 1$ if and only if $z \in$ 1-turn-Dyck_2. Since $\text{EQUALITY}(x,y)$ has a communication complexity of n bits even if multiple rounds of communication are allowed (see for example [11]) at least in one round, n/p bits are required to be communicated. Hence the theorem follows.

3 Streaming algorithm for a subclass of VPLs

In this section we prove Theorem 4. Visibly pushdown languages, VPLs, were defined by [13] and [2]. They are known to be a subclass of DCFLs. In [3], a grammar form for VPLs was defined. We denote the grammars generating VPLs by VPG. Here, we consider a restriction of VPG.

Consider a context-free grammar $G = (N, \Sigma, S, P)$ over a tri-partitioned alphabet $\Sigma = (\Sigma_c, \Sigma_r, \Sigma_l)$ having rules of the form $A \longrightarrow \epsilon$, $A \longrightarrow cB$, $A \longrightarrow aBb$ or $A \longrightarrow aBbD$, where $a \in \Sigma_c$, $b \in \Sigma_r$, $c \in \Sigma_l$, and $D \longrightarrow \epsilon \notin P$ (i.e. if $A \rightarrow \epsilon$, then there is no rule in P that has A as its second non-terminal.) If $A \longrightarrow a\omega$ and $A \longrightarrow a'\omega'$ are two rules such that $\omega, \omega' \in (N \cup \Sigma)^*$, then $a \neq a'$. We denote such grammars by rest-VPG and the languages generated by it as rest-VPLs.

The example language generated by such a grammar is $\{a^n b^n a^m b^m | n \geq 1, m \geq 1\}$. The rest-VPG, generating it is $S \longrightarrow aAbB,\ A \longrightarrow aAb \mid \epsilon, B \longrightarrow aAb$

We coin a notation to address these rules of rest-VPG. Let a rule be called *linear* if it has one non-terminal on the right hand side. Let a rule be called *quadratic* if there are two non-terminals on the right hand side.

We first make one crucial observation about the derivation of a string in a language generated by rest-VPG.

Observation 9 *Let L be generated by a rest-*VPG*, say G_L. Let $w \in \Sigma^n$. If $w \in L$, then for any derivation d of w in G_L, the number of times a quadratic rule is applied is at most $rev(L, n)$.*

Proof. Recall that an index $i \in [n]$ is called a reversal if $w[i-1] \in \Sigma_r$ and $w[i] \in \Sigma_c \cup \Sigma_l$.

As $w \in L$, and L is generated by G_L. Therefore, there is a derivation for w in G_L. Suppose there exists a derivation for w in G_L that needs more than $rev(L, n)$ applications of quadratic rules. Every time a quadratic rule is applied, it gives rise to one reversal. Therefore, the string w must have more than $rev(L, n)$ reversals, which is a contradiction.

Let L be a fixed rest-VPL and let L be generated by a rest-VPG G_L. Let w be the input string. For every application of a quadratic rule in the derivation of w, we perform a book keeping operation storing $O(\log n)$ bits in the memory. As the maximum number of times a quadratic rule is applied in the derivation of w is bounded by $rev(w)$ (due to Observation 9), we get the desired bound. (If the storage grows beyond $O(rev(n) \log n)$ the algorithm rejects and halts.)

We now describe the book keeping. Initially the expected sum is zero. As long as linear rules are applied, expected sums can be computed as in Algorithm 2. Consider the first time a quadratic rule is applied. The string w can be split into four parts at this stage. (See Figure 1 for example.) The first part consists of the string of length l that has been read so far, and the fourth part consists of the suffix of w of length l. The second (third) part consists of the substring of w generated by the first (second, respectively) non-terminal.

The expected sum accumulated during the first part needs to be used when the fourth part is being processed. The second non-terminal is needed when the third part is being processed. Thus, while starting to check the second part, the sum as well as the second non-terminal are stacked. Once, the second part is recursively checked, the stacked non-terminal is popped and it is recursively processed. After this, the algorithm needs to check the fourth part. During this stage, the sum that was stacked is popped and used as in Algorithm 2.

Algorithm 4 Streaming algorithm for membership testing of rest-VPLs

```
Pick $\alpha_1, \alpha_2, \ldots, \alpha_k$ uniformly at random from $\mathbb{F}_p$, where $k = |\Sigma|$.
$Sum \leftarrow 0$, $var \leftarrow S$, $h \leftarrow 0$, $Rev \leftarrow rev(n)$ (encoded along with the input)
for $i = 1$ to $n$ do
  if $w[i] \in (\Sigma_c \cup \Sigma_l)$ and $next_1(w[i], var)$ is undefined then
    reject and halt
  end if
  if $w[i] \in \Sigma_c$ then
    $v_2 \leftarrow next_2(w[i], var)$
    if $v_2 \neq \epsilon$ then
      if $Rev = 0$ then
        'reject' and halt
      else
        Stack.Push($v_2, Sum, h$)   (*Addressed as: StackTop.$Var$, StackTop.$Sum$,
        StackTop.$Height$*)
        $Sum = 0$
        $Rev \leftarrow Rev - 1$
      end if
    end if
    $h \leftarrow h + 1$
    Let $index$ be $j$ if $type(w[i], var) = x_j$
    $Sum \leftarrow Sum + \alpha_{index}^{h} (mod\ p)$ , $var \leftarrow next_1(w[i], var)$,
  else if $w[i] \in \Sigma_r$ then
    if $w[i-1] \in \Sigma_c$ then
      $var \leftarrow \epsilon$   (*This handles rules of the form $A \longrightarrow \epsilon$*)
    end if
    Let $index$ be $j$ if $w[i] = a_j$
    $Sum \leftarrow Sum - \alpha_{index}^{h} (mod\ p)$
    $h \leftarrow h - 1$
    if $h =$ StackTop.$Height$ then
      if $Sum \neq 0$ then
        reject and halt
      else if StackTop.$Var = \epsilon$ then
        $Sum \leftarrow$ StackTop.$Sum$
        Stack.Pop()
      else
        $var \leftarrow$ StackTop.$Var$
        StackTop.$Var \leftarrow \epsilon$
      end if
    end if
  else
    $var \leftarrow next_1(w[i], var)$ (*i.e. $w[i] \in \Sigma_l$*)
  end if
end for
```

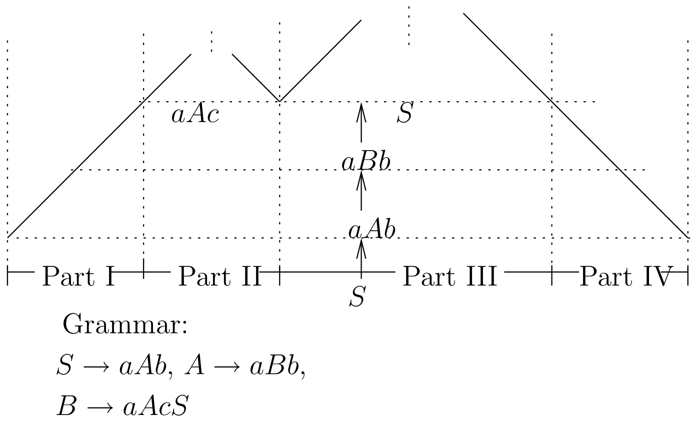

Fig. 1. The first time a quadratic rule is applied

We start with some notation. Let *type* and $next_1, next_2$ be functions such that $type : \Sigma \times N \rightarrow \{x_1, \ldots, x_k\}$ $next_1 : \Sigma \times N \rightarrow N$ and $next_2 : \Sigma \times N \rightarrow N$.

If $A \rightarrow a_i B a_j D$ is a rule in the grammar G_L, then the values of $type(a_i, A)$, $next_1(a_i, A)$, and $next_2(a_i, A)$ are x_j, B and D, respectively. (Here $a_i \in \Sigma_c$ and $a_j \in \Sigma_r$. Also, if $D = \epsilon$ then $next_2(a_i, A)$ is ϵ).

If $A \rightarrow a_i B$ is a rule in the grammar G_L, then the values of $next_1(a_i, A)$, $next_2(a_i, A)$, are B and ϵ, respectively. (Note that in this case, $a_i \in \Sigma_l$.)

3.1 Proof of correctness of algorithm 4

The algorithm maintains an explicit counter *Rev*, to keep track of the number of reversals seen so far. When the counter runs down to zero, the algorithm rejects and halts. *Rev* is decremented every time a quadratic rule is applied. This is upper bounded by $Rev(L, n)$ for length n inputs by Observation 9. Thus the space bound.

For any L in rest-VPL having a grammar $G = (N, \Sigma, P, S)$, we define a PDA $M_L = (\{q\}, \Sigma, N, \delta, q, S,)$, where the transition function δ is defined as follows:

- for each production of the form $A \rightarrow a\omega$, $\delta(q, a, A) = (q, \omega)$,
- for all $a \in \Sigma$, $\delta(q, a, a) = (q, \epsilon)$
- for each production of the form $A \rightarrow \epsilon$, $\delta(q, r, Ar) = (q, \epsilon)$ for all $r \in \Sigma_r$

M_L starts with only the start symbol S on the stack and it accepts by empty stack. It is easy to see that the language accepted by M_L is L.

Consider an algorithm similar to Algorithm 4, but maintains a polynomial, instead of its evaluation. Then it is easy to see that it simulates M_L on any input belonging to the language.

If $w \in L$, then M_L would accept with an empty stack. The algorithm will also accept because all the polynomials that it keeps track of during its execution will become the zero polynomial whenever the polynomial identity testing is performed. Each of these corresponds to the points at which the Algorithm 4 checks whether $Sum = 0$. As $w \in L$, the polynomial will be identically zero and hence the evaluation will also be zero.

If the input $w \notin L$ and M_L rejects, say on reading $w[i]$. Suppose $w[i] \in \Sigma_c \cup \Sigma_l$ then the algorithm will also reject on reading $w[i]$ because $next_1(w[i], var)$ will be undefined. Suppose $w[i] \in \Sigma_r$, then this implies that the stack top, say a_j of M_L did not match with the input $a_k = w[i]$. The polynomial maintained by the algorithm at this instant will have a term corresponding to x_j^h, where h is the height of the stack of M_L and the term subtracted is x_k^h. Note that var is ϵ at this point. From now on the height will keep decrementing and var will remain ϵ until the next polynomial identity testing is done. The polynomial identity test will fail. Hence, the algorithm will reject w. But the Algorithm 4 only keeps an evaluation of the polynomial at a random point chosen from a large enough field. The Schwartz-Zippel theorem gives us that with high probability the evaluation will be non-zero if the polynomial is non-zero. Hence Algorithm 4 rejects with high probability.

4 Streaming algorithms for checking degree sequence of graphs

The input to a graph streaming algorithm may be in the form of a sequence of edges. These may be provided in a specific order (eg: adjacency list), or in any arbitrary fashion. It has been observed [4, 7] that the complexity of graph problems changes drastically depending on the order in which the edges are presented to the streaming algorithm. The usual setting is: the edges are assumed to be presented in any arbitrary order. If one is able to provide an efficient algorithm in such a setting, then of course this gives the most general algorithm. However, more recently Das Sarma et al. [4] observed that it is also useful to consider other orderings of the edges. They observed that, the algorithm can be considered as a resource bounded verifier and that the input is presented to the verifier by a prover. In this setting, two models can be considered: the adversarial model and the best order model [4]. In the former the prover may provide the edges in the worst possible order. In contrast to this, in the latter model, the prover orders the edges in such a manner that the verifier can easily check the property.

We consider the problem Deg-Seq which we defined in Section 1. Under various assumptions on its input, we analyze its complexity. If the input to Deg-Seq is such that the degree of a vertex along with all the edges out of that vertex are listed one after the other, then checking whether the graph has the given degree sequence is trivial. If the degrees sequence is listed first, followed by the adjacency list of the graph then we observe the following:

Lemma 1. *Any p-pass deterministic algorithm for* Deg-Seq *needs $\Omega(n/p)$ space when the input is: a degrees sequence, followed by the adjacency list.*

Proof. Again, as in the proof of Theorem 3, we reduce the two party communication problem of testing equality to that of Deg-Seq. Given strings $x, y \in \{0,1\}^n$ we obtain a degree sequence $d = (d_1, d_2, \cdots, d_n)$ and a list of edges $e_1 e_2 \cdots e_m$. Take $d = x$ and for each i such that $y_i = 1$, add an edge (i, i). Clearly $\text{EQUALITY}(x, y) = 1$ if and only if d is the degree sequence of the graph with edges $e_1 e_2 \cdots e_m$. Since $\text{EQUALITY}(x, y)$ has a communication complexity of n bits(see for example [11]), in any p rounds protocol for EQUALITY with two players, atleast 1 message will be of size n/p. The reduction above translates a p pass algorithm for Deg-Seq to a p rounds protocol for $\text{EQUALITY}(x, y)$ and hence the lemma follows.

Now we give a p-pass, $O(n/p \log n)$-space determinsitic algorithm for Deg-Seqand hence prove part 2 of Theorem 5. The algorithm simply stores the degrees of n/p vertices during a pass and checks whether those vertices have exactly the degree sequence as stored. If the degree sequence is correct, then proceed to the next set of n/p vertices. The algorithm needs to store $O(n/p \log n)$ bits during any pass. The algorithm makes p-passes. Also Lemma 1 implies that this algorithm is optimal, ignoring a $\log n$ factor.

We present the proof of the first part of Theorem 5.

Proof (of part 1 of Theorem 5). We come up with a uni-variate polynomial from the given degree sequence and the set of edges such that the polynomial is identically zero if and only if the graph has the given degree sequence (assuming some predecided order on the vertices).

We do not store the polynomial explicitly. Instead, we evaluate this polynomial at a random point chosen from a large enough field and only maintain the evaluation of the polynomial. The Schwartz-Zippel lemma [14] gives us that with high probability the evaluation will be non-zero if the polynomial is non-zero. (If the polynomial is identically zero, its evaluation will also be zero.)

The uni-variate polynomial can be constructed as follows: Let the vertices be labelled from 1 to n (in the order in which the degrees appear in the degree sequence). Let $l : V(G) \to [n]$ be a function such that $l(v)$ gives the label for $v \in V(G)$.

The uni variate polynomial we construct is:

$$q(x) = \sum_i d_i x^i - \sum_{\exists v:(u,v) \in input} x^{l(u)}$$

The algorithm can be now described as:

Algorithm 5 Randomized streaming algorithm for Deg-Seq

```
Pick $\alpha \in_R \mathbb{F}_p$. ($p$ will be fixed later)
$Sum \leftarrow 0$.
for $i = 1$ to $n$ do
    $Sum \leftarrow Sum + d_i \alpha^i$
end for
for $i = 1$ to $m$ (where $m$ number of edges) do
    $Sum \leftarrow Sum - \alpha^{l(u)}$, where $e_i = (u, v)$
end for
if $Sum = 0$ then
    output "yes"
else
    output "no"
end if
```

It is easy to note that the algorithm requires only log-space as long as p is $O(poly(n))$. The input is being read only once from left to right. For the correctness, note that if the given degree sequence corresponds to that of the given graph, then $q(x)$ is identically zero and the value of Sum is also zero for any randomly picked α. We know that $q(x)$ is non-zero when the given degree sequence does not correspond to that of the given graph. However, the evaluation may still be zero. Note that degree of $q(x)$ is n. If the field size is chosen to be $n^{1+c} \leq p \leq n^{2+c}$ then due to

Schwartz-Zippel lemma [14] the probability that Sum is zero given that $q(x)$ is non-zero is at most n/p which is at most n^{-c}.

Acknowledgements: We thank Meena Mahajan and Jaikumar Radhakrishnan for useful discussions.

References

1. Noga Alon, Yossi Matias, and Mario Szegedy. The space complexity of approximating the frequency moments. In *STOC '96: Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, pages 20–29, 1996.
2. R. Alur and P. Madhusudan. Visibly pushdown languages. In *STOC*, pages 202–211, 2004.
3. Rajeev Alur and P. Madhusudan. Adding nesting structure to words. *J. ACM*, 56(3), 2009.
4. Danupon Nanongkai Atish Das Sarma, Richard J. Lipton. Best-order streaming model. In *The 6th Annual Conference on Theory and Applications of Models of Computation (TAMC)*, pages 178–191, 2009.
5. B. Von Braunmühl and R. Verbeek. Input-driven languages are recognized in $\log n$ space. In *Proc. FCT Conference, LNCS*, pages 40–51, 1983.
6. Patrick W. Dymond. Input-driven languages are in $\log n$ depth. *Information Processing Letters*, 26:247–250, 1988.
7. Joan Feigenbaum, Sampath Kannan, Martin Strauss, and Mahesh Viswanathan. Testing and spot-checking of data streams. *Algorithmica*, 34(1):67–80, 2002.
8. Markus Holzer and Klaus-Jörn Lange. On the complexities of linear LL(1) and LR(1) grammars. In *FCT '93: Proceedings of the 9th International Symposium on Fundamentals of Computation Theory*, pages 299–308, London, UK, 1993. Springer.
9. John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.
10. O.H. Ibarra, T. Jiang, and B. Ravikumar. Some subclasses of context-free languages in NC^1. *Information Processing Letters*, 29:111–117, 1988.
11. Eyal Kushilevitz and Noam Nisan. *Communication Complexity.* Cambridge University Press, New York, NY, USA, 2006.
12. Frédéric Magniez, Claire Mathieu, and Ashwin Nayak. Recognizing well-parenthesized expressions in the streaming model. In *STOC*, 2009.
13. K. Mehlhorn. Pebbling mountain ranges and its application to DCFL recognition. In *Proc. 7th ICALP*, pages 422–432, 1980.
14. Rajeev Motwani and Prabhakar Raghavan. Randomized algorithms. *ACM Comput. Surv.*, 28(1):33–37, 1996.
15. S. Muthukrishnan. Data streams: algorithms and applications. In *SODA '03: Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 413–413, 2003.
16. Heribert Vollmer. *Introduction to Circuit Complexity: A Uniform Approach.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.