

# Improved Streaming Algorithm for Dyck(s) Recognition

Anubhav Kumar Jain

A Thesis Submitted to  
Indian Institute of Technology Hyderabad  
In Partial Fulfillment of the Requirements for  
The Degree of Master of Technology



Department of Computer Science and Technology

June 2014

## **Declaration**

I declare that this written submission represents my ideas in my own words, and where ideas or words of others have been included, I have adequately cited and referenced the original sources. I also declare that I have adhered to all principles of academic honesty and integrity and have not misrepresented or fabricated or falsified any idea/data/fact/source in my submission. I understand that any violation of the above will be a cause for disciplinary action by the Institute and can also evoke penal action from the sources that have thus not been properly cited, or from whom proper permission has not been taken when needed.

---

(Signature)

---

(Anubhav Kumar Jain)

---

(Roll No.)

## Approval Sheet

This Thesis entitled Improved Streaming Algorithm for Dyck(s) Recognition by Anubhav Kumar Jain is approved for the degree of Master of Technology from IIT Hyderabad

---

(Dr. K. S. Rama Murty) Examiner  
Dept. of Electrical Engineering  
Indian Institute of Technology Hyderabad

---

(Dr. M. V. Panduranga Rao) Examiner  
Dept. of Computer Science and Engineering  
Indian Institute of Technology Hyderabad

---

(Dr. Subrahmanyam Kalyanasundaram) Adviser  
Dept. of Computer Science and Engineering  
Indian Institute of Technology Hyderabad

---

(Dr. C. Krishna Mohan) Chairman  
Dept. of Computer Science and Engineering  
Indian Institute of Technology Hyderabad

## **Acknowledgements**

I express my sincere gratitude towards my advisor Dr. Subrahmanyam Kalyanasundaram for his constant help, encouragement and inspiration throughout the thesis work. Without his invaluable guidance, this work would never have been a successful one. No amount of gratitude to my parents would be sufficient. I am very fortunate to be where I am today because of their constant support and encouragement.

## **Dedication**

This thesis is dedicated to my father, Sri Abhinandan Kumar Jain, and my mother, Smt. Shashi jain, who taught me the value of education and who left their native, "Chirimiri, Chattishgarh" a very small town, and shifted to a completely new city "Bhopal, Madhya Pradesh" for our education, their children, so that we could have the better opportunities of education that, they did not have.

## Abstract

Keeping in mind, that any context free language can be mapped to a subset of *Dyck* languages and by seeing various database applications of *Dyck*, mainly verifying the well-formedness of XML file, we study the randomized streaming algorithms for the recognition of *Dyck(s)* languages, with  $s$  different types of parenthesis. The main motivation of this work is well known space bound for any  $T$ -pass streaming algorithm is  $\Omega(\sqrt{n}/T)$ .

Let  $x$  be the input stream of length  $n$  with maximum height  $h_{max}$ . Here we present a single-pass randomized streaming algorithms to decide the membership of  $x$  in *Dyck(s)* using Counting Bloomfilter (CBF) with space  $O(h_{max})$  bits,  $\text{polylog}(n)$  time per letter with two-sided error probability. Two-sided error is because of the false negative and false positives of counting bloomfilter. This algorithms denies the necessity of streaming reduction of *Dyck(s)* into *Dyck(2)*, that reduces the space even further by the factor of  $O(\log s)$ , compared to those uses streaming reduction.

We also present an improved single-pass randomized streaming algorithm for recognizing *Dyck(2)* with space  $O(\sqrt{n})$  bits, which is the proven lower bound. Time bound is same  $\text{polylog}(n)$ , as other existing algorithms and error is one-sided. In this algorithm, we extended the existing approach of periodically compressing stack information. Existing approach uses two stacks and a linear hash function, instead of this we are using three stacks and same linear hash function to achieve space lower bound of  $O(\sqrt{n})$ .

We also present another single-pass streaming algorithm with  $O(h_{max})$  space that uses counting bloomfilter and directly acts on *Dyck(s)*.

# Contents

Declaration . . . . .	ii
Approval Sheet . . . . .	iii
Acknowledgements . . . . .	iv
Abstract . . . . .	vi
<b>Nomenclature</b>	<b>viii</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Overview of the Work . . . . .	2
1.2 Thesis Outline . . . . .	2
<b>2 Related work</b>	<b>3</b>
<b>3 Definition and Preliminaries</b>	<b>4</b>
3.1 Notations . . . . .	4
3.2 Definition . . . . .	4
3.3 Preliminaries . . . . .	5
3.4 Bloomfilter and Counting Bloomfilter . . . . .	6
<b>4 Existing Algorithms</b>	<b>8</b>
4.1 Single-pass Streaming Algorithm with $O(\sqrt{n \log n})$ space . . . . .	8
4.1.1 Hash Function . . . . .	9
4.1.2 Space Analysis . . . . .	10
4.2 Bidirectional Streaming Algorithm with $O((\log n)^2)$ space . . . . .	10
4.2.1 Time Analysis . . . . .	11
4.2.2 Space Analysis . . . . .	12
<b>5 Proposed Algorithms</b>	<b>13</b>
5.1 Single-pass Streaming Algorithm Using Counting Bloomfilter (CBF) . . . . .	13
5.1.1 Space Analysis . . . . .	14
5.1.2 Time Analysis . . . . .	15
5.1.3 Error . . . . .	15
5.2 Improved Single-pass Streaming Algorithm with $O(\sqrt{n})$ space . . . . .	16
5.2.1 Correctness . . . . .	18
5.2.2 Space Analysis . . . . .	19
5.2.3 Time Analysis . . . . .	19
5.2.4 Error . . . . .	19

<b>6 Results</b>	<b>21</b>
<b>7 Conclusion and Future Work</b>	<b>22</b>
<b>References</b>	<b>23</b>



# Chapter 1

## Introduction

The area of streaming algorithms has experienced huge growth in various applications in recent past. These algorithms sequentially scan entire input, letter by letter, in one pass or in very few number of passes (i.e. random access to the input is not possible) and uses limited space (less than linear or nearly polylogarithmic in input size). Streaming algorithms have various application like searching documents in web databases, analysis of Internet traffic[1], network monitoring for elephant flows, counting the number of distinct flows, estimating size of a join in SQL, XML validation[2] etc. which involves massive data that arrives rapidly and cannot be stored.

The work have been done to understand the applications of streaming algorithms in context of formal languages. We explore  $Dyck(s)$  problem, where  $s$  represents the number of types of parentheses. In this problem one has to check the well-formedness of the stream of parentheses. By the definition of regular language, they are decidable using deterministic finite state automaton(DFA) and streaming algorithm within constant space. The  $Dyck$  languages is one of the simplest context-free languages with significant importance in theory. Since every context-free language can be mapped to a subset of  $Dyck(s)$  [3], we study some of the recognition algorithms for  $Dyck$  languages. This study is similar to the problem of checking matching parentheses, which is very frequent in database applications like verifying the well-formedness of XML files.

The context-free languages are recognizable in  $O((\log n)^2)$  space, if access to the input stream is random[4]. With optimal tree representation[5] of  $Dyck(s)$ , logarithmic space is sufficient as it has to check parentheses of same level at each levels. Translating this approach into streaming algorithm is not easy, even with fewer number of passes over the stream.

Single-pass deterministic streaming algorithm with logarithmic space uses a height counter to decide membership in  $Dyck(1)$ . For  $Dyck(2)$ , deterministic Single-pass streaming algorithm requires linear space. The linear space can be proved using one-way communication complexity argument for EQUALITY. Magniez, Mathieu, and Nayak in [6] presented two randomized streaming algorithms for the membership problem of  $Dyck(2)$ , that can easily be extended to  $Dyck(s)$  using streaming reduction. First algorithms is single-pass algorithm with space of  $O(\sqrt{n \log n})$  and second is two-pass bidirectional algorithm with space  $O((\log n)^2)$  bits. Both of these algorithm uses stack compression technique to reduce the space requirement with  $polylog(n)$  time per letter and one-sided error.

We present an improved single-pass randomized streaming algorithm for  $Dyck(2)$  with space matching to the lower bound i.e.  $O(\sqrt{n}/T)$  where  $T \geq 1$  is the number of passes. Here the number of passes is only one and hence the space is  $O(\sqrt{n})$  with  $polylog(n)$  time per letter and one-sided error. This algorithms extends the approach of, Magniez et.al [6], from two stacks to three stacks and reduces the space to match the lower bound. We also present another single-pass streaming algorithm with  $O(h_{max})$  space that uses counting bloomfilter and directly acts on  $Dyck(s)$ .

## 1.1 Overview of the Work

As part of thesis work, we propose two single-pass streaming algorithms (a) single-pass streaming algorithm using Counting bloomfilter. The main benefit of this approach is, it avoids streaming reduction and can be applied directly to  $Dyck(s)$  (b) Improved single-pass streaming algorithm using three stacks, in this algorithm we matches the proven lower bound. We also provides the proof of correctness and space bound for both the algorithms.

## 1.2 Thesis Outline

The thesis is structured as follows. In chapter 2 we give a brief introduction of various similar problem on  $Dyck(s)$ . We discuss some standard definition, preliminaries and randomized data structure "Bloomfilter" in chapter 3. We explain the single-pass and bidirectional algorithms to decide the membership of  $Dyck(s)$  in chapter 4. In Chapter 5 we describes our two proposed algorithms, one of which matches the lower bound. We compare the space bound of our algorithms with existing algorithms in chapter 6. We discuss future work in chapter 7.

## Chapter 2

### Related work

The  $Dyck(s)$  membership problem has been studied earlier in the name of property testing algorithms for context free language. In [7] [8] [9] authors has defined the problem of property testing, in which algorithms accepts if any string  $x$  that satisfy the required property. A property tester for any language  $L$  accepts all strings  $x$ , if  $x \in L$  and rejects all those string which are at  $\varepsilon$  distance (normalized hamming distance) from the strings in  $L$ . In same context, [10] has proved that, any  $Dyck(1)$  is  $\varepsilon$ -testable in constant time, where  $\varepsilon > 0$  is fixed constant and in [11] Parnas et. al. has proved that any  $Dyck(s)$ , for  $s > 0$  is  $\varepsilon$ -testable in time  $O(n^{2/3})$ , with the lower bound of  $O(n^{1/11})$ .

[2] has presented an  $O(\log n)$  pass deterministic streaming algorithm with  $O((\log n)^2)$  space for XML validation problem (which is exactly same as  $Dyck(s)$  problem) with external memory. They have used FCNS encoding of the XML file as in intermediate step in the streaming manner.

In [12] Nathanael Francois and Frederic Magniez has addressed the problem of checking priority queues in the setting where sequence of insert and extract operation is given and one wants to decide a priory whether the sequence is valid. They used the similar approach as bidirectional algorithm ( see algorithm 2) of  $Dyck(2)$  and reduced the memory to  $O((\log n)^2)$ .

[13] discusses the problem of efficiently computing edit distance to  $Dyck$  languages and proposes a near-linear time algorithm. They proved that, if there exist an algorithm to calculate the edit distance of stream in  $\alpha(n)$  time with  $\beta(n)$ -approximation factor, then we can design an algorithm to calculate the edit distance of  $Dyck(s)$  in  $O(n^{1+\epsilon} + \alpha(n))$  with an approximation factor of  $O\left(\frac{1}{\epsilon}\beta(n) \log OPT\right)$ .

The two different works done by Chakrabarti, Cormode, Kondapally and McGregor [14] and Jain and Nayak [15] respectively has proved that multiple passes of the same stream in one direction will not help. Any T-pass, where  $T > 1$ , streaming algorithm to decide the membership of a stream in  $Dyck(2)$  with constant error probability takes  $\Omega(\sqrt{n}/T)$  space.

[16] also addresses the similar problem of  $Dyck(s)$  recognition when string contains few errors ( called 1-turn- $Dyck(2)$ ). They proved that, if stream  $x$  contains at most  $k$  errors, then there exist a randomized single-pass streaming algorithm to decide the membership of 1-turn- $Dyck(2)$  with space  $O(k \log n)$ ,  $O(k \log n)$  randomness, and  $\text{poly}(k \log n)$  time per letter with error at most  $1/n^{\Omega(1)}$ .

# Chapter 3

## Definition and Preliminaries

**Dyck language** is the language consisting of strings that have properly balanced parentheses. The Dyck language contains strings like  $a\bar{a}$  and  $a\bar{a}aa\bar{a}$  but not  $\bar{a}\bar{a}$ , where  $a$  represents upsteps and  $\bar{a}$  represents downsteps. The well-formedness/balancedness of the strings makes sense, if the types of parenthesis is more. If we see any programming language or any XML file, then we can see such cases. Let  $A = \{a_1, \dots, a_k\}$  be the set representing upsteps and  $\bar{A} = \{\bar{a}_1, \dots, \bar{a}_k\}$  be the disjoint set of corresponding downstep symbols. For example, the pair  $A = \{(\, [, do, if, < title \>, < html \>\}$  and  $\bar{A} = \{), ], od, fi, < \title \>, < \html \>\}$  represents the nested nature of programming languages and XML files. It is also important in the parsing of expressions that must have a correctly nested sequence of parentheses, such as arithmetic or algebraic expression. XML documents are the perfect examples of Dyck language.

### 3.1 Notations

In the following discussion,  $x$  represent the input stream  $x_1x_2 \dots x_n$  of length  $n$  from the alphabet  $\sum_2 = \{a, b, \bar{a}, \bar{b}\}$  or  $\sum_s = \{a_1, \bar{a}_1, a_2, \bar{a}_2, \dots, a_s, \bar{a}_s\}$ . An *upstep* is represented by small letters like  $a_i$  where as *downstep* by overlined letters  $\bar{a}_i$ . Then for integers  $i \leq j$ ,  $x[i, j]$  represents a subword  $x_i x_{i+1} \dots x_j$  and  $x[i]$  denotes the  $i^{th}$  letter in the stream  $x$ .  $|x|_p$  represents the number of times  $p$  has occurred in  $x$  and  $|x|$  without subscript represent the length of the word  $x$ . wherever we are using log, we mean log with base 2 through out this document.

### 3.2 Definition

**Definition 3.1** (Height). Let  $x \in \sum_2^n$  and  $|x|_a$  represents the number of upsteps of type  $a$  in  $x$  where as  $|x|_{\bar{a}}$  represents the number of downsteps of type  $\bar{a}$  in  $x$ . The height of  $x$  can be defined as

$$height(x) = |x|_a + |x|_b - |x|_{\bar{a}} - |x|_{\bar{b}}$$

**Definition 3.2** (Matching pair). A pair  $(i, j)$ , where  $1 \leq i < j \leq n$ , for  $x$  is a matching pair if,

$$height(x[1, i-1]) = height(x[1, j]) \text{ and}$$

$$height(x[1, k]) > height(x[1, i-1]) \text{ for all } k \in \{i, \dots, j-1\}$$

We can say the contribution of matching pair in the overall height of the stream is zero.

**Definition 3.3** (Well-formed). A matching pair  $(i, j)$  for  $x$  is Well-formed, if  $(x[i], x[j])$  equals  $(a, \bar{a})$  or  $(b, \bar{b})$ , ill-formed

otherwise. It means a letter  $x_i$  at index  $i$  can form a matching pair with at most one letter at other index  $j$ .

**Definition 3.4** (Partial order). *we say two words  $u$  and  $v$  are partially ordered i.e  $u \prec v$ , if and only if  $u$  can be obtained by removing zero or more matching pairs from  $v$ . This order is well defined, and in particular transitive, since matching pairs of  $u$  are still in  $v$ .*

### 3.3 Preliminaries

**Definition 3.5** ( $Dyck(s)$ ). *Let  $s \geq 1$  be a positive integer. Then  $Dyck(s)$  denotes the language over alphabets  $\Sigma_s = \{a_1, \bar{a}_1, a_2, \bar{a}_2, \dots, a_s, \bar{a}_s\}$  defined recursively by*

$$Dyck(s) = \epsilon + \sum_{i \leq s} a_i.Dyck(s).\bar{a}_i.Dyck(s)$$

where  $a_i$  represents upstep,  $\bar{a}_i$  represents downstep and  $\epsilon$  represents an empty string. The operator  $(+)$  represents the alteration operation, that allows to chose one from the possible option. Operands around  $+$  is the possible options. The operator  $(\cdot)$  is the concatenation operator. We can always reduce any  $Dyck(s)$  into  $Dyck(2)$  using streaming reduction see propostion 3.2 .

**Definition 3.6** (Streaming algorithm). *Fix an alphabet  $\Sigma$ . A  $k$ -pass streaming algorithm  $\mathbf{A}$  with space  $w(n)$  and time  $t(n)$  is an algorithm such that for every input stream  $x \in \Sigma^n$  :*

1.  $\mathbf{A}$  performs  $k$  sequential passes on  $x$ .
2.  $\mathbf{A}$  maintains a memory space of size  $w(n)$  bits while reading  $x$ .
3.  $\mathbf{A}$  has running time at most  $t(n)$  per letter  $x_i$ .
4.  $\mathbf{A}$  has pre-processing and post-processing time atmost  $t(n)$ .

$\mathbf{A}$  is bidirectional if it is allowed to access the input in forward as well as in reverse direction. The parameter  $k$  is the total number of passes in either direction.

**Definition 3.7** (Streaming reduction). *Fix two alphabet  $\Sigma_1$  and  $\Sigma_2$  for problem  $P_1$  and  $P_2$  respectively. Let  $f(n) : \Sigma_1 \rightarrow \Sigma_2^{f(n)}$  be a function used for reduction. A problem  $P_1$  is  $f(n)$ -streaming reducible to a problem  $P_2$  with space  $w(n)$  and time  $t(n)$ , if for every input  $x \in \Sigma_1^n$ , there exist  $y = y_1 y_2 \dots y_n$ , with  $y_i \in \Sigma_2^{f(n)}$ , such that*

1.  $y_i$  can be computed deterministically from  $x_i$  using space  $w(n)$  and time  $t(n)$ ;
2. From a solution of  $P_2$  with input  $y$ , a solution on  $P_1$  with input  $x$  can be computed with space  $w(n)$  and time  $t(n)$ .

**Proposition 3.1.** *Let  $P_1$  be  $f(n)$ -streaming reducible to a problem  $P_2$  with space  $w_0(n)$  and time  $t_0(n)$ . Let  $\mathbf{A}$  be a  $k$ -pass streaming algorithm for  $P_2$  with space  $w(n)$  and time  $t(n)$ . Then there is a  $k$ -pass streaming algorithm for  $P_1$  with space  $w(n \times f(n)) + w_0(n)$  and time  $t(n \times f(n)) + t_0(n)$  with the same properties as  $\mathbf{A}$  (deterministic/randomized, unidirectional/bidirectional ).*

**Proposition 3.2.**  *$Dyck(s)$  is  $\lceil \log s \rceil$ -streaming reducible to  $Dyck(2)$  with space and time  $O(\log s)$  .*

*Proof.* We encode a parenthesis  $a_i$  by a word of length  $l = \lceil \log s \rceil$  with only parenthesis of type  $b, c$ . We let  $f(a_i)$  be the binary expansion of  $i$  over  $l$  bits where 0 is replaced by  $b$  and 1 by  $c$ . Then  $f(\bar{a}_i)$  is defined similarly, except that we write the binary expansion of  $i$  in the opposite order and replace 0 by  $\bar{b}$  and 1 by  $\bar{c}$ . Then  $x_1 \dots x_n$  is in  $Dyck(s)$  if and only if  $f(x_1) \dots f(x_n)$  is in  $Dyck(2)$ .  $\square$

Since the parameter  $s$  is a constant, independent of the length of the input stream, the above reduction can be implemented with constant space and time.

For example, Let  $U = \{t_1, t_2, t_3, t_4\}$  be the set of  $s = 4$ , upsteps and  $\bar{U} = \{\bar{t}_1, \bar{t}_2, \bar{t}_3, \bar{t}_4\}$  be the disjoint set of corresponding downsteps. Let  $t = t_1 t_2 t_4 \bar{t}_3 \bar{t}_2 \bar{t}_1$ , be the input stream. Here  $\log s = \log 4 = 2$ , therefore  $t_1 = 00, t_2 = 01, t_3 = 10, t_4 = 11$ . Then we can write the encoding of  $t$  as  $0001111110011000 = bbbccc\bar{c}\bar{c}b\bar{b}\bar{c}\bar{c}b\bar{b}\bar{b}$ . We can generate this encoding of the stream while reading itself, *i.e.* character by character .

From Proposition 3.1, it is enough to design streaming algorithms for  $Dyck(2)$ , as we can convert any  $Dyck(s)$  into  $Dyck(2)$ . Which is the is discussed in the next section.

### 3.4 Bloomfilter and Counting Bloomfilter

A Bloomfilter[17],  $B$  is a randomized data structure which represents an un-ordered set  $S$  of  $n$  elements from a universe  $U$  using an array of  $m$  bits, represented by  $B[1], \dots, B[m]$ , all initialized to 0. The Bloomfilter uses a set  $H$  of  $k$  independent hash functions  $h_1, \dots, h_k$  with range  $\{1, \dots, m\}$ , that independently hash each element in the universe to a random number uniformly over the range. (This is a standard assumption and very convenient for the analysis of Bloomfilter). For each element  $x \in S$ , the bits  $B[h_i(x)]$  are set to 1 for  $1 \leq i \leq k$  (a bit can be set to 1 multiple times.). To answer a query of the form Does  $y \in S$ ?, we check whether all  $B[h_i(y)]$  are set to 1. If not,  $y$  is not a member of  $S$ , by the construction. If all  $B[h_i(y)]$  are set to 1, it is assumed that  $y$  is in  $S$ , and hence a Bloomfilter may yield a false positive. The probability of a false positive for an element not in the set can be derived easily. If  $p$  is the fraction of ones in the filter, it is simply  $p^k$ . A standard combinatorial argument gives that  $p$  is concentrated around its expectation

$$p = \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right) \approx \left(1 - e^{-kn/m}\right) \quad (3.1)$$

So false positive probability  $f$  which is equal to  $p^k$  will be:

$$f \approx \left(1 - e^{-kn/m}\right)^k \quad (3.2)$$

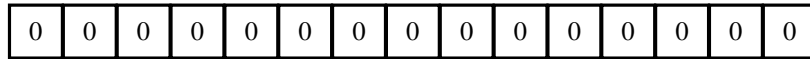


Figure 3.1:  $m$  bits Bloomfilter all filled with 0's

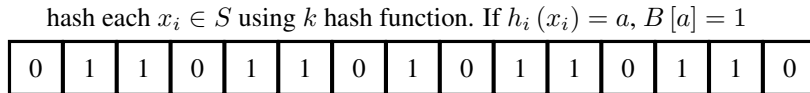


Figure 3.2: Bloomfilter after inserting each  $x_i \in S$

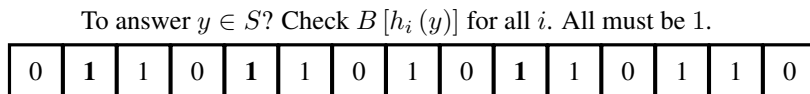


Figure 3.3: Bold 1's show the presence of  $y$  in  $S$

$B[h_i(z)]$  is 1 for all  $i$ . But  $z$  is not present.

0	1	1	0	1	1	0	1	0	1	1	0	1	1	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Figure 3.4: False positives in Bloomfilter

The expressions 3.1, 3.2 are minimized when  $k = \ln 2 \cdot (m/n)$ , giving a false positive probability  $f \approx (1/2)^k \approx (0.6185)^{m/n}$ . In practice,  $k$  must be an integer, and both  $n/m$  (the number of bits per set element) and  $k$  should be treated as constants. For example, when  $n/m = 10$  and  $k = 7$  the false positive probability is just over 0.008. Deleting elements from a Bloomfilter cannot be done simply by changing ones back to zeros, as a single bit may correspond to multiple elements. Standard Bloomfilter can be extended to Counting Bloomfilter(CBF) to support deletion at the cost of 4X space increment. A CBF uses an array of  $n$  counters instead of bits; the counters track the number of elements currently hashed to that location. Deletions can now be safely done by decrementing the relevant counters. A CBF can be converted to standard Bloomfilter by just setting all non-zero counters to 1. Counters must be chosen large enough to avoid overflow; for most applications, four bits suffice. We generally use the rule of four bits per counter when comparing results of our data structure with a standard CBF, although this can be reduced with some additional complexity.

Bloomfilter has been studied and modified in various ways to support deletion. Standard approach of using counter instead of a bit opens the door of improvement in space requirement, lookup performance, false positive and false negative rate. Some of the important variants are the compressed Bloomfilter[18], counting Bloomfilter (CBF)[19], distance-sensitive Bloomfilter [20], space-code Bloomfilter[21], spectral Bloomfilter[22], generalized Bloomfilter[23], Bloomier filter[24], Variable-increment counting Bloomfilter[25], d-left counting Bloomfilters (dl-CBFs). dl-CBFs, which adopt a hash table-based approach constructs a hash table for all known items by d-left hashing [6], but replaces each item with a short fingerprint (i.e., a bit string derived from the item using a hash function). The dl-CBFs can reduce the space cost of counting Bloomfilters, but still require twice the space of a space-optimized Bloomfilter.

There is one more data structure called "Cuckoo filter" recently proposed by Bin Fan et. al.[26] which gives the same functionality like CBF. It supports dynamic adding, removing of elements and gives higher lookup performance. Surprisingly it uses very less space compared to the standard non-deletion-supporting type Bloomfilter at the cost of false positive rate  $< 3\%$ .

# Chapter 4

## Existing Algorithms

### 4.1 Single-pass Streaming Algorithm with $O(\sqrt{n \log n})$ space

The following algorithm given by Magniez, Mathieu and Nayak[6] addresses the problem of deciding membership in  $Dyck(s)$ . They have shown that any  $Dyck(s)$  language is streaming reducible (Proposition 3.2) to  $Dyck(2)$  with just  $\log s$  factor of expansion in the length of input stream. The approach in this algorithm is easiest to understand if we consider input stream  $x = uv$ , where  $u$  is the continuous sequence of upsteps and  $v$  is the continuous sequence of only downsteps in equal numbers. This algorithm is motivated from the naive approach of stack, instead of using stack of size  $n/2$  it uses stack  $S_{temp}$  of size  $q = \sqrt{n \log n}$ . While reading letters from the first half of stream  $x$  it stores the upstep in  $S_{temp}$  (see figure 4.1). Once the stack is filled, it calculates the hash of all  $q$  elements of the stack and stores it in another stack  $S$  (assuming  $n$  is divisible by  $2q$ ). Stack  $S$  stores one entry for each block of  $q$  elements,  $hash([iq + 1, (i + 1)q])$  for each  $i \in \{0, 1, \dots, n/2q - 1\}$ . While reading the second half of the same stream it adds  $hash([jq + 1, (j + 1)q])$  to the  $hash([iq + 1, (i + 1)q])$  for  $j = n/q - i - 1$  and check for the sum to be zero. If the sum is 0, then the stream is well-formed and in  $Dyck(2)$  otherwise ill-formed.

**Algorithm 1** tries to collect sequence of  $l$  upsteps, while doing so, it checks for the matching pairs, by using standard stack-based algorithm. Upstep followed by downstep is checked for the well-formedness and then discarded. This check is performed using stack  $S_{temp}$ , it checks for every matching pair encountered in the stream till the limit of  $S_{temp}$  reaches. Once the stack  $S_{temp}$  is filled with  $l$  upsteps, then algorithm hashes  $v = v_1v_2v_3 \dots v_l$  (sequence of  $l$  upsteps) to  $hash(v)$

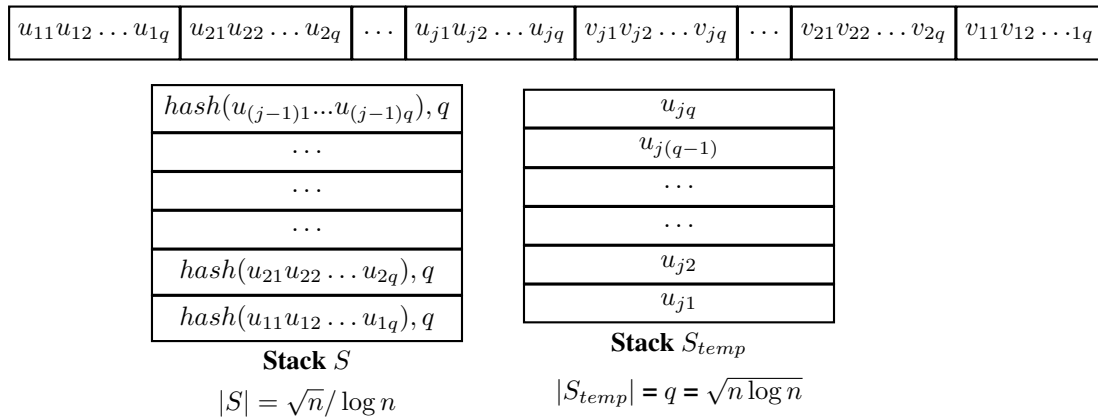


Figure 4.1: Shows the condition of stacks  $S_{temp}$  and  $S$  just before reading  $v$ .  $x = uv$ , is the input stream, where  $u$  is the continuous sequence of only upsteps and  $v$  is of only downsteps is same number.



and empties  $S_{temp}$ . The hash value  $h$  and along with the height  $l$  is pushed onto the stack  $S$ . Each entry of the stack  $S$  is of the form  $(h, l)$  and encodes the subword  $v$  of the stream  $x$  such that  $h = hash(v)$  and  $l = height(v)$ . The algorithm uses stack  $S$  to store the information about the blocks read so far.

When algorithm reads any downstep  $y$ , it either tries to match downstep with the top element of the  $S_{temp}$  or if  $S_{temp}$  is empty, adds the hash value of  $y$  to top element of stack  $S$ . More precisely, it updates given  $(h, l) = (hash(v), height(v))$  by  $hash(vy) = h + hash(y)$  and  $height(vy) = l - 1$ . In this way it incorporates the information of  $y$  in the stacks. The hash function which is used and its correctness is explained in the following section .

### 4.1.1 Hash Function

In the streaming model of  $Dyck(1)$  membership problem, one-pass of the stream using height counter is sufficient to decide membership. This is a deterministic algorithm takes only logarithmic space. But in case of  $Dyck(2)$ , it can only ensure that, at any instant, number of upsteps are always greater than or equal to the number of downsteps in the stream. After processing whole stream, if the height is zero, it ensures that number of upsteps and downsteps are equal but possibly ill-formed. This algorithm uses following hash function which keeps track of well-formedness while accepting any stream.

Let  $p$  be a prime number such that  $n^{1+\gamma} \leq p \leq 2n^{1+\gamma}$ , for some fixed  $\gamma > 0$ . Pick a uniformly random  $\alpha \in [0, p - 1]$ . Algorithms uses random hash function  $hash(\cdot)$  that maps subword  $v$  of  $x$  to integer in  $[0, p - 1]$ , as follows:

$$hash(x_{i_1}, x_{i_2}, \dots, x_{i_m}) = \sum_j hash(x_{i_j}), \text{ where} \quad (4.1)$$

$$hash(x_i) = \begin{cases} \alpha^{height(x[1, i-1]) \bmod p} & \text{if } x_i = a, \\ -\alpha^{height(x[1, i-1]) \bmod p} & \text{if } x_i = \bar{a}, \\ 0 & \text{otherwise} \end{cases} \quad (4.2)$$

Given  $x$  and  $v$ , the value of  $hash(v)$  is a polynomial in  $\alpha$  of degree bounded by the maximum height of a prefix of  $x$ , which is at most  $n$ . A polynomial of degree  $d$  over  $F_p$  has at most  $d$  roots. Therefore, if  $hash(v)$  is not identically zero, for a uniformly random  $\alpha$ , the probability that  $hash(v) = 0$  is at most  $n/p \leq n - \gamma$ .

**Proposition 4.1.** *Let  $x \in \sum^n$  be such that every prefix of  $x$  has non-negative height, and let  $v = x_{i_1}x_{i_2} \dots$  be a subword of  $x$ . If  $v \in Dyck(2)$ , then  $hash(v) = 0$  for all  $\alpha$ . Moreover, if there is a height  $d$  at which  $v$  has single ill-formed pair (and possible other ill-formed matching pairs at height  $\neq d$ ), then  $hash(v) \neq 0$  with probability at least  $1 - n^{-\gamma}$ , for a uniformly random integer  $\alpha \in [0, p - 1]$*

*Proof.* If  $v \in Dyck(2)$ , then  $v$  is well-formed and then each well-formed matching pair  $(i, j)$  at height  $d$  contributes

$$\begin{cases} \alpha^d - \alpha^d = 0, & \text{if } (x_i, x_j) = (a, \bar{a}); \\ 0 - 0 = 0, & \text{if } (x_i, x_j) = (b, \bar{b}). \end{cases}$$

Therefore, we get  $hash(v) = 0$ .

Now, assume there is a height  $d$  at which  $v$  has a single ill-formed pair. Since every prefix of  $x$  has non negative height, the value  $hash(v)$  is a polynomial  $q(z)$  evaluated at  $z = \alpha$ . Every well-formed pair at height  $d$  cancels, and so the coefficient of  $z^d$  in  $q$  is  $+1$  if  $(x_i, x_j) = (a, \bar{b})$ , and  $-1$  if  $(x_i, x_j) = (b, \bar{a})$ . Thus  $q$  is not identically zero. The claim follows from the uniformly random choice of  $\alpha$ .

For any letter  $x_i$  we may compute  $hash(x_i)$  in time  $polylog(n)$  and space  $O(\log n)$ . Moreover, for any word  $v$  the value of  $hash(v)$  can be maintained with  $O(\log n)$ .  $\square$

**Theorem 4.1.** *Algorithm 1 is a single-pass randomized streaming algorithm for Dyck(2) with space  $O(\sqrt{n \log n})$  and time  $\text{polylog}(n)$ . If the stream belongs to Dyck(2) then the algorithm accepts it with certainty; otherwise it rejects it with probability at least  $1 - n^{-c}$ , where  $c > 0$  is a constant.*

*Proof.* For proof of the theorem refer [6]. □

---

**Algorithm 1** Single-pass streaming algorithms using two stacks  $S_{temp}$  and  $S$

---

```

1:  $S_{temp} \leftarrow$  empty stack of upsteps;  $S \leftarrow$  empty stack of items  $(h, l)$ 
2:  $(h_{temp}, l_{temp}) \leftarrow (0, 0)$  {This pair encodes the subword contained in  $S_{temp}$  }
3: Compute a prime  $p$  such that  $n^{1+\gamma} \leq p \leq 2n^{1+\gamma}$ ; Pick a uniformly random  $\alpha \in [0, p - 1]$  {The pair  $(p, \alpha)$  are used in
   the function hash;  $\gamma > 0$  is a constant of our choice.}
4: while stream is not empty do
5:   read next letter  $y$  from stream
6:   if  $y$  is an upstep then
7:     push  $y$  on  $S_{temp}$ 
8:     update  $(h_{temp}, l_{temp})$  with  $y$ :  $h_{temp} \leftarrow (h_{temp} + \text{hash}(y) \bmod p)$ ;  $l_{temp} \leftarrow l_{temp} + 1$ 
9:     if  $S_{temp}$  has size  $\lceil \sqrt{n \log n} \rceil$  then
10:      push  $(h_{temp}, l_{temp})$  on to  $S$  and reset  $S_{temp}$  to empty;  $(h_{temp}, l_{temp}) \leftarrow (0, 0)$ 
11:    end if
12:   else { $y$  is a downstep}
13:     if  $S_{temp}$  is not empty then
14:       pop  $z$  from  $S_{temp}$ 
15:       check that  $zy$  is well-formed:  $zy \in \{a\bar{a}, b\bar{b}\}$  (if not, reject: "mismatch")
16:       update  $(h_{temp}, l_{temp})$  for removal of  $z$ :  $h_{temp} \leftarrow (h_{temp} - \text{hash}(z) \bmod p)$ ;  $l_{temp} \leftarrow l_{temp} - 1$ 
17:     else { $S_{temp}$  is empty}
18:       pop  $(h, l)$  from  $S$  (if empty, reject: "extra closing parenthesis")
19:       update  $(h, l)$  with  $y$ :  $h \leftarrow (h + \text{hash}(y) \bmod p)$ ;  $l \leftarrow l - 1$ 
20:       if  $l = 0$  then check that  $h = 0$  (if not, reject: "mismatch")
21:       else push  $(h, l)$  on  $S$ 
22:     end if
23:   end if
24: end while
25: if  $S$  and  $S_{temp}$  are not both empty then reject: "missing closing parenthesis"
26: end if
27: accept

```

---

### 4.1.2 Space Analysis

This algorithm uses two stacks namely  $S_{temp}$  and  $S$  to store the sequence of upsteps in compressed manner. Each entry in  $S_{temp}$  and  $S$  takes space  $O(1)$  and  $O(\log n)$  respectively. When  $S_{temp}$  gets filled, a new element is pushed into  $S$ . To get new element in  $S$ , algorithm has to process at least  $\lceil \sqrt{n \log n} \rceil$  letters for every new element in  $S$ , which bounds the size of  $S$  by  $\sqrt{n/\log n}$ . We know the size of each entry in  $S$  is  $O(\log n)$ . Hence  $S$  uses  $O(\sqrt{n \log n})$  space. So the total space requirement by the algorithm is  $O(\sqrt{n \log n}) + O(\sqrt{n \log n}) = O(\sqrt{n \log n})$ .

## 4.2 Bidirectional Streaming Algorithm with $O((\log n)^2)$ space

This algorithm, as name suggests, processes the stream from both the directions. It maintains the binary decomposition of stream seen so far. Binary decomposition involves breaking of the stream  $x$  where  $|x| = n$ , from left to right into  $m \leq \log n$  contiguous blocks of decreasing length, where length of each block is in power's of 2. For example, if stream

seen so far is  $x[1, j]$  then the lengths of blocks will be  $2^{i_m}, 2^{i_{m-1}} \dots 2^{i_1}$  such that  $j = \sum_{t=1}^m 2^{i_t}$ . It also assumes that the length of stream  $n = 2^k$ , for some  $k \geq 1$ , if not we can append the word  $(a\bar{a})$  to  $x$ .

Now the bidirectional algorithm (algorithm 2), applies algorithm 3 from both direction, first pass from left to right and second from right to left. During first pass it adds the necessary number of word  $a\bar{a}$  to make  $|x| = 2^k$ , while in second pass, letters  $\bar{a}, \bar{b}$  are treated as  $a, b$  and  $a, b$  as  $\bar{a}, \bar{b}$  respectively. The key idea of the algorithm is that, if we process the stream from both direction then, each matching pair will be checked at least once in either direction.

---

#### Algorithm 2 Bidirectional Algorithm

---

Compute a prime  $p$  such that  $n^{1+\gamma} \leq p \leq 2n^{1+\gamma}$ ; Pick a uniformly random  $\alpha \in [0, p-1]$   
 {The pair  $(p, \alpha)$  are used in the function hash;  $\gamma > 0$  is a constant of our choice.}  
 Run **Algorithm 3**, reading the stream from left to right  
 Run **Algorithm 3**, reading the stream from right to left  
 { while reading the stream right to left,  $\bar{a}, \bar{b}$  are interpreted as  $a, b$ , respectively (and vice-versa)}  
**accept**

---



---

#### Algorithm 3 One-pass of the bidirectional algorithm

---

- 1:  $S_{temp} \leftarrow$  empty stack of items  $(h, l, f)$
- 2:  $j \leftarrow 0$  { This records the length of the stream read so far }
- 3: **while** stream is not empty **do**
- 4:   read next letter  $y$ , and set  $j \leftarrow j + 1$
- 5:   **if**  $y$  is an upstep **then**
- 6:     push the item  $(hash(y), 1, j)$  on to  $S$  {This encodes the letter  $y$ }
- 7:   **else** { $y$  is a downstep}
- 8:     pop  $(h, l, f)$  from  $S$  (if empty, **reject**: "extra closing parenthesis")
- 9:     update  $(h, l, f)$  with  $h : h \leftarrow (h + hash(y) \bmod p)$ ;  $l \leftarrow l - 1$
- 10:    **if**  $l = 0$  **then** check that  $h = 0$  (if not, **reject**: "mismatch")
- 11:    **else** push  $(h, l, f)$  on  $S$
- 12:    **end if**
- 13: **end if**
- 14: **while** the top 2 elements of  $S$  both start in the last block of the binary partition of  $[1, j]$  **do**
- 15:    combine them into one element: pop  $(h_2, l_2, f_2)$ ;  $(h_1, l_1, f_1)$ ; push  $(h_1 + h_2, l_1 + l_2, f_1)$
- 16: **end while**
- 17: **end while**
- 18: **if**  $S$  is not empty **then reject**: "missing closing parenthesis"
- 19: **end if**

---

**Theorem 4.2.** *Algorithm 2 is a bidirectional two-pass randomized streaming algorithm for Dyck(2) with space  $O((\log n)^2)$  and time  $\text{polylog}(n)$ . If the stream belongs to Dyck(2) then the algorithm accepts it with certainty; otherwise it rejects it with probability at least  $1 - n^{-c}$ , where  $c > 0$  is a constant.*

*Proof.* For proof of the theorem refer [6]. □

### 4.2.1 Time Analysis

Processing time of this bidirectional algorithm for every element in the stream is dominated by the computation of the hash function as it involves modular exponentiation, which takes  $\text{polylog}(n)$  time[27]. So time per letter will be  $\text{polylog}(n)$ . The time to perform push and pop operation in the stacks is constant. So the overall processing time of the algorithm is  $O(\text{polylog}(n))$  per letter.

### 4.2.2 Space Analysis

This algorithm, unlike single-pass algorithm, uses only one stack. Each element of stack takes space  $O(\log n)$ . As this algorithm stores binary decomposition of stream in the stack, the stack size is bounded by  $2 \log n$ , hence the required space is  $O((\log n)^2)$ .

single-pass streaming algorithm takes  $O(\sqrt{n \log n})$  time and the bidirectional algorithm takes  $O((\log n)^2)$  time. It seems that if we can do multiple passes of the stream like bidirectional algorithm then we can reduce the space. But the two different works done by Chakrabarti, Cormode, Kondapally and McGregor [14] and Jain and Nayak [15] respectively has proved that multiple passes of the same stream in one direction will not help. Any T-pass, where  $T > 1$ , streaming algorithm to decide the membership of a stream in  $Dyck(2)$  with constant error probability takes  $\Omega(\sqrt{n}/T)$  space.

# Chapter 5

## Proposed Algorithms

### 5.1 Single-pass Streaming Algorithm Using Counting Bloomfilter (CBF)

Let  $x$  be a stream of alphabets from  $\sum_2$  then the naive approach to decide  $x \in Dyck(2)$  is to use a stack of size  $O(n)$  and whenever there is an upstep ( $a$  or  $b$ ), push it in the stack and whenever there is a downstep ( $\bar{a}$  or  $\bar{b}$ ), pop the top element from the stack and compare it with the current element. If they are of same type, continue the process else declare that  $x \notin Dyck(2)$ . In the end after processing complete stream, if the stack is empty then stream is in  $Dyck(2)$ .

**Proposition 5.1.** *Let  $s = |\sum_s|$ , where  $\sum_s$  is the alphabet set. In the stack based approach for deciding membership in  $Dyck(s)$ , the size of stack at any instant is always less than or equal to the maximum height attained by the stream.*

*Proof.* Let stream  $x = x_1x_2 \dots x_n$ . The proof is simple, whenever there is an upstep we push an element into the stack and pop when there is a downstep. So the number of element in the stack is  $max\{height(x[1, i]) = |x|_{a_i} - |x|_{\bar{a}_i}\}$  for all  $i \leq n$ .  $\square$

This naive approach works fine in the case of  $Dyck(2)$  where size of each upstep or downstep is fixed to one character. But to achieve this property we have to go through the process of streaming reduction which will increase the space and time by the factor of  $O(\log s)$ . Because of the streaming reduction of  $Dyck(s)$  to  $Dyck(2)$  each character in the reduced stream is of 1 bit and so the each entry of the stack. Thus the space required by the stack, having maximum of  $n$  elements of  $O(1)$  bit, is  $O(n)$ .

To reduce the space requirement, many works used various stack compression techniques, as discussed in the previous sections. Like in first algorithm, Magniez et. al. are using linear hash function to periodically (after every  $\sqrt{n/\log n}$  letters) compress the stack. In this approach they used two stacks  $S_{temp}$  and  $S$ . The size of  $S_{temp}$  is  $\sqrt{n \log n}$  and the size of each entry in  $S_{temp}$  is one bit where as size of  $S$  is  $\sqrt{n/\log n}$  and the size of each entry is  $O(\log n)$ . This streaming algorithm which takes single-pass of the stream uses  $O(\sqrt{n \log n})$  space and  $polylog(n)$  time per letter to decide the membership of stream in  $Dyck(2)$ . If the stream belongs to  $Dyck(2)$  then the algorithm accepts it with certainty; otherwise it rejects it with probability at least  $1 - n^{-c}$  where  $c > 0$  is a constant.

One of the most important application of this  $Dyck$  problem is XML verification. The XML verification problem is same as deciding the membership of  $Dyck(s)$  where, each opening xml-tag is considered as the opening parenthesis and the closing xml-tag as closing parenthesis in the stream of tags from the set of xml tags. But the problem is, size of each tag can be different. So the single-pass deterministic (randomized) streaming algorithms that takes linear *i.e*  $O(n)$  ( or  $O(\sqrt{n \log n})$  for the randomized algorithm) time cannot support the case of variable size tag without using streaming reduction. In other words, streaming reduction is a necessary step to decide the membership in  $Dyck(s)$  for variable size parenthesis.

In the proposed algorithm (*single-pass streaming algorithm using counting Bloomfilter*), we are not doing streaming reduction of the stream to *Dyck(2)* as our algorithm can handle any number of parenthesis of any size, Secondly, unlike previous approaches of using stack and then using compressing techniques to reduce the space, we are using Counting Bloomfilter (CBF). Our approach is simple, whenever there is an upstep ( $a_i$ ), hash it to the CBF *i.e.*, increment the counter at all  $k$  locations and whenever there is a downstep ( $\bar{a}_i$ ), decrement the counters at all  $k$  hash locations. If all the entries of the CBF is zero at the end after processing whole stream, it means stream is in *Dyck(S)*. At the same time during the computation of hash function, algorithm keep track of the height. It rejects if the height of any prefix is negative. The space, time and error of the algorithm is discussed in the following section.

---

**Algorithm 4** Single-pass algorithm using Counting Bloomfilter

---

```

1: Initialize counting bloomfilter  $B$  with all zeros.
2: Initialize  $k$  {It defines the number of hash function to be used}
3: Compute a prime  $p$  such that  $n^{1+\gamma} \leq p \leq 2n^{1+\gamma}$ ; {The pair  $(p, \alpha)$  are used in the function hash;  $\gamma > 0$  is a constant of our choice.}
4: Initialize hash function  $h_i$  for all  $i \in \{1, 2, \dots, k\}$  {Pick  $\alpha_i$  for all  $h_i$  uniformly at random in the range  $[0, p - 1]$  }
5: while stream is not empty do
6:   read next letter  $y$ 
7:   if  $y$  is an upstep then
8:     for all  $i \in \{1, 2, \dots, k\}$  do
9:       calculate hash location for  $y$  using  $h_i$ 
10:      increment the  $B[h_i(y)]$ 
11:    end for
12:   else { $y$  is a downstep}
13:     for all  $i \in \{1, 2, \dots, k\}$  do
14:       calculate hash location for  $y$  using  $h_i$  {Assuming  $y$  as upstep of the same type}
15:       if  $B[h_i(y)] > 0$  then
16:         decrement the  $B[h_i(y)]$ 
17:       else
18:         reject: "mismatch"
19:       end if
20:     end for
21:   end if
22: end while
23: if All entries in the  $B$  is not zero then reject: "missing closing parenthesis"
24: end if
25: accept

```

---

### 5.1.1 Space Analysis

Space requirement of our algorithm depends on the size of the Counting Bloomfilter (CBF). The size of CBF depends on the number of unique elements(keys) that are going to be hash on the CBF and the number of hash functions used.  $k$ , the number of hash function is a user defined parameter, generally  $k = 7$  suffices with very high probability. Specially, when the ratio between the number of elements in the stream and number of unique elements in the stream is 10. (as explained in section 3.4).

**Proposition 5.2.** *In algorithm 4, at any instance, no two hashed elements in the counting bloomfilter will have same height.*

*Proof.* We will prove it by contradiction. Let two elements  $x_i$  and  $x_j$ , where  $i < j$ , are hashed to CBF with the same height  $d$ . Then,

$$ht(x_i) = height(x[1, i]) = d \quad \text{and} \quad ht(x_j) = height(x[1, j]) = d$$

which means,

$$\text{height}(x[i + 1, j - 1]) = 0$$

Both  $x_i$  and  $x_j$  are upsteps, since both elements are still in CBF,

Then,

$$\text{ht}(x_j) = \text{height}(x[1, j]) = \text{ht}(x_i) + \text{height}(x[i + 1, j - 1]) + 1$$

$$\text{ht}(x_j) = \text{height}(x[1, j]) = d + 0 + 1 \neq \text{ht}(x_i)$$

Which contradicts our assumption. Hence two hashed elements in the CBF can not have same height. □

The hash functions that we are using uses type of parenthesis *i.e*  $a_i$  and the  $\text{height}(a_i)$  to calculate the hash location in CBF. In case of  $\text{Dyck}(s)$  language, number of unique elements is same as the number of different combination of  $a_i$  and  $\text{height}(a_i)$ . The important thing to notice is, *no two elements in the CBF can have same height*. So the number of elements in the CBF, at any instance is less than or equal to the maximum height attained. Hence the space required is in order of maximum height of the stream.

**Proposition 5.3.** *Let  $s = |\sum_s|$ , where  $\sum_s$  is the alphabet set. In the CBF based approach for deciding membership in  $\text{Dyck}(s)$ , the number of unique elements in CBF at any instant is always less than or equal to the maximum height attained by the stream.*

Proof is similar as Proposition 5.1.

In the worst case, when the stream  $x$  is of the form  $x = uv$ , where  $u$  has only upsteps and  $v$  has only downsteps, in equal numbers. Then the stream will have all possible height from 1 to  $n/2$  distributed (assume equally for calculation) among  $a_i$ 's where  $i \in \{1, 2, \dots, s\}$ . So the space requirement in the worst case will be  $(\frac{n}{2s} \cdot s \cdot k)$ , which is  $O(n)$ , as  $k$  is almost a constant. If we consider the case where upsteps and downsteps are randomly distributed (which is generally the case with practical application of  $\text{Dyck}$  like, XML recognition), then the maximum height of the stream will be very less, so is the number of combinations of  $a_i$  and  $\text{height}(a_i)$  and so the space required is  $O(h)$  where  $h$  is the maximum height of the stream  $x$ .

The algorithm described above works directly on  $\text{Dyck}(s)$  and does not require streaming reduction. It means, our algorithm saves  $O(\log s)$  times of space that other algorithms use to reduce  $\text{Dyck}(s)$  into  $\text{Dyck}(2)$ .

### 5.1.2 Time Analysis

Processing time of any element in the stream is dominated by the computation of the hash function. And we have seen in the previous algorithms that time to calculate hash function is  $\text{polylog}(n)$ , which takes  $\text{polylog}(n)$  time[27]. So in our case also time per letter will be  $\text{polylog}(n)$ , as we have seen in section 3.4 constant number of hash function is enough to give very less error probability, so we can consider  $k$  as a constant, so the time is  $O(\text{polylog}(n))$  per letter.

### 5.1.3 Error

Error involved in the algorithm is because of the following reasons:

- The probabilistic calculation of prime number in  $\text{polylog}(n)$  time[27]. The standard and most efficient procedure to find a prime number  $p$  such that  $n^{1+\gamma} \leq p \leq 2n^{1+\gamma}$ , outputs a prime number with probability  $1 - n^{-\gamma}$  where  $\gamma > 0$  is a constant.

- In our algorithm, whenever an upstep appears it increments all the  $k$  counters and when a downstep appears it tries to decrements the same counter using the same hash function. To decrements the same counter it raises a membership query to check the presence of corresponding upstep. While doing so because of the false positive error ( Section 3.4 ), counting Bloomfilter can answer wrong with error probability equal to the false positive rate of CBF.
- Latest research by Deke Gua et. al.[28] has proved that Bloomfilter returns false negatives because of incorrect deletion caused by the false positive rate. So the Bloomfilter can give wrong answer with probability equal to false negative rate of CBF.

## 5.2 Improved Single-pass Streaming Algorithm with $O(\sqrt{n})$ space

This algorithm is same as the existing single-pass streaming algorithm explained briefly in section 4.1. In that algorithm Magniez et. al. [6] are using two stacks  $S_{temp}$  and  $S$ . The size of  $S_{temp}$  is  $\sqrt{n \log n}$  and the size of each entry in  $S_{temp}$  is  $O(1)$  bit where as size of  $S$  is  $\sqrt{n/\log n}$  and the size of each entry is  $O(\log n)$ . This streaming algorithm which takes single-pass of the stream uses  $O(\sqrt{n \log n})$  space and  $polylog(n)$  time per letter to decide the membership of stream in  $Dyck(2)$ .

As we know the standard stack based approach uses only one stack and takes linear space to decide the membership where as algorithm explained earlier that two stack to do the same with reduced space,  $O(\sqrt{n \log n})$ . Two stacks, one for removing matching pairs and other for storing the fingerprints of the letters seen so far that remain to be checked. It gives an intuition that if we increase the number of stacks we can reduce the space further. Keeping in mind that the space lower bound[14][15] proved for the  $Dyck(2)$  membership problem is  $\Omega(\sqrt{n}/T)$ , where  $T$  is the number of passes that algorithm performs over the input stream in either direction. And the best existing algorithm till now in terms of space takes  $O(\sqrt{n \log n})$ . So we increase the number of stacks from two to three which results in the following algorithm that takes space matching to the proven lower bound *i.e.*  $O(\sqrt{n})$ .

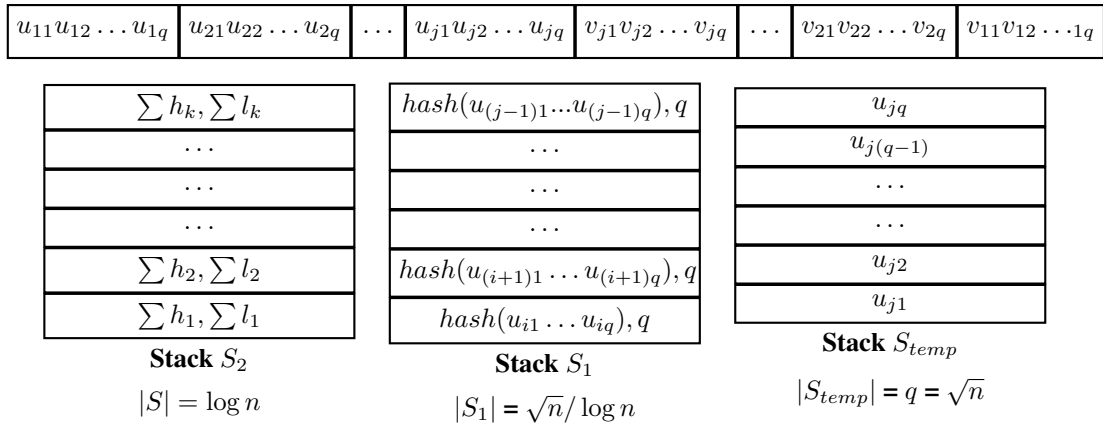


Figure 5.1: Shows the condition of stacks  $S_{temp}$   $S_1$  and  $S$  just before reading  $v$ .  $x = uv$ , is the input stream, where  $u$  is the continuous sequence of only upsteps and  $v$  is of only downsteps is same number.  $S_{temp}$  stores the upsteps,  $S_1$  stores the hash of  $q$  letter once  $S_{temp}$  is filled and  $S_2$  stores the sum of the items of  $S_1$ , once filled.

**Algorithm 5**(see figure 5.1) also tries to collect sequence of  $l$  upsteps, but this time  $l = \lceil \sqrt{n} \rceil$ , while doing so, it checks for the matching pairs, by using standard stack-based algorithm. Upstep followed by downstep is checked for the well-formedness and then discarded. This check is performed using stack  $S_{temp}$ , it checks for every matching pair encountered in the stream till the limit of  $S_{temp}$  reaches. Once the stack  $S_{temp}$  is filled with  $l$  upsteps, then algorithm hashes  $v = v_1v_2v_3 \dots v_l$  (sequence of  $l$  upsteps) to  $hash(v)$  and empties  $S_{temp}$ . The hash value  $h$  along with the height  $l$  is pushed into the stack  $S_1$ . If stack  $S_1$  reaches its limit, then comes the role of stack  $S_2$ . The purpose of  $S_2$  is same as



$S_1$ , only difference is, it takes values only when  $S_1$  gets filled.  $S_2$  stores the encoding ( say sum of all the hash values in stack  $S_1$  ) of stack  $S_1$ . To do so for every entry in  $S_1$ , update  $h_1$  as  $h_1 = h_1 + h_{temp}$  and  $l_1$  as  $l_1 = l_1 + l_{temp}$  and whenever there is an entry in stack  $S_2$  reset both  $h_1$  and  $l_1$ . Each entry of the stack  $S_1$  and  $S_2$  is of the form  $(h, l)$  encodes the subword  $v$  of the stream  $x$  such that  $h = hash(v)$  and  $l = height(v)$ . The algorithm uses stack  $S_1$  and  $S_2$  to store the information about the blocks read so far. The hash function used in this algorithm is also same as defined in section 4.1.1.

---

**Algorithm 5** Single-pass streaming algorithm with three stacks  $S_{temp}$ ,  $S_1$  and  $S_2$

---

```

1:  $S_{temp} \leftarrow$  empty stack of upsteps;  $S_1 \leftarrow$  empty first stack of items  $(h, l)$ ;  $S_2 \leftarrow$  empty second stack of items  $(h, l)$ 
2:  $(h_{temp}, l_{temp}) \leftarrow (0, 0)$  {This pair encodes the subword contained in  $S_{temp}$  }
3:  $(h_1, l_1) \leftarrow (0, 0)$  {This pair encodes the subword contained in stack  $S_1$  }
4: Compute a prime  $p$  such that  $n^{1+\gamma} \leq p \leq 2n^{1+\gamma}$ ; Pick a uniformly random  $\alpha \in [0, p-1]$  {The pair  $(p, \alpha)$  are used in the function hash;  $\gamma > 0$  is a constant of our choice.}
5: while stream is not empty do
6:   read next letter  $y$  from stream
7:   if  $y$  is an upstep then
8:     push  $y$  on  $S_{temp}$ 
9:     update  $(h_{temp}, l_{temp})$  with  $y$  :  $h_{temp} \leftarrow (h_{temp} + hash(y) \bmod p)$ ;  $l_{temp} \leftarrow l_{temp} + 1$ 
10:    if  $S_{temp}$  has size  $\lceil \sqrt{n} \rceil$  then
11:      update  $(h_1, l_1)$  such that  $h_1 \leftarrow (h_1 + h_{temp})$ ;  $l_1 \leftarrow l_1 + l_{temp}$ 
12:      push  $(h_{temp}, l_{temp})$  on to  $S_1$  and reset  $S_{temp}$  to empty;  $(h_{temp}, l_{temp}) \leftarrow (0, 0)$ 
13:      if  $S_1$  has size  $\lceil \sqrt{n} / \log n \rceil$  then
14:        push  $(h_1, l_1)$  on to  $S_2$  and reset  $S_1$  to empty;  $(h_1, l_1) \leftarrow (0, 0)$ 
15:      end if
16:    end if
17:  else { $y$  is a downstep}
18:    if  $S_{temp}$  is not empty then
19:      pop  $z$  from  $S_{temp}$ 
20:      check that  $zy$  is well-formed:  $zy \in \{a\bar{a}, b\bar{b}\}$  (if not, reject: "mismatch")
21:      update  $(h_{temp}, l_{temp})$  for removal of  $z$  :  $h_{temp} \leftarrow (h_{temp} - hash(z) \bmod p)$ ;  $l_{temp} \leftarrow l_{temp} - 1$ ;
22:    else if  $S_1$  is not empty then
23:      pop  $(h, l)$  from  $S_1$ 
24:      update  $(h, l)$  with  $y$  :  $h \leftarrow (h + hash(y) \bmod p)$ ;  $l \leftarrow l - 1$ ;
25:      if  $l = 0$  then check that  $h = 0$  (if not, reject: "mismatch")
26:      else push  $(h, l)$  on  $S_1$ 
27:      end if
28:    else if  $S_2$  is not empty then
29:      pop  $(h, l)$  from  $S_2$  (if empty, reject: "extra closing parenthesis")
30:      update  $(h, l)$  with  $y$  :  $h \leftarrow (h + hash(y) \bmod p)$ ;  $l \leftarrow l - 1$ ;
31:      if  $l = 0$  then check that  $h = 0$  (if not, reject: "mismatch")
32:      else push  $(h, l)$  on  $S_2$ 
33:      end if
34:    else reject: "extra closing parenthesis"
35:    end if
36:  end if
37: end while
38: if any of  $S_2$ ,  $S_1$  and  $S_{temp}$  is not empty then reject: "missing closing parenthesis"
39: end if
40: accept

```

---

For brevity, consider the same stream  $x = uv$  where  $u = u_1u_2 \dots u_{n/2}$  is the continuous sequence of only upsteps and  $v = v_1v_2 \dots v_{n/2}$  is the continuous sequence of same number of only downsteps. To decide membership of  $x$  in  $Dyck(2)$ , we read input in blocks of length  $q$ . For each block  $x[iq+1, (i+1)q]$  where  $i \in \{1, 2, \dots, n/2q-1\}$  store

item  $(h_i, l_i)$  where  $h_i = \text{hash}(v_{i_1}v_{i_2} \dots v_{i_q})$  and length  $l_i = q$  in the stack  $S_1$ . Similarly when  $S_1$  gets filled we store  $(\sum_i h_i, \sum_i l_i)$  in stack  $S_2$ .

Now in order to process  $v$ , sequence of downsteps, we either check for the match with top element of stack  $S_{temp}$  or add it to the top element of stack  $S_1$  and decrements  $l$ . If  $S_1$  is empty then add to the top element of stack  $S_2$  and decrement  $l$ . After decrementing, if  $l$  becomes zero and  $h$  is still non-zero then reject. If all three stacks are empty after processing the stream then algorithm outputs that,  $x \in \text{Dyck}(2)$  else outputs  $x \notin \text{Dyck}(2)$ .

### 5.2.1 Correctness

The correctness of this algorithm relies on the linearity of the hash function. Consider algorithm is correct upto first level stack *i.e.*  $S_1$  as all the steps are same as previous "Single-pass streaming algorithm with  $O(\sqrt{n \log n})$  space" refer section 4.1 till this point. Now it remains to verify if the use of second level stack *i.e.*  $S_2$  preserves the order of upsteps or not.

**Theorem 5.1.** *Algorithm 5 is a single-pass randomized steaming algorithm for Dyck(2) with space  $O(\sqrt{n})$  and time  $\text{polylog}(n)$ . If the stream belongs to Dyck(2) then the algorithm accepts it with certainty; otherwise it rejects it with probability at least  $1 - n^{-c}$ , where  $c > 0$  is a constant.*

*Proof.* Proof for both space and time complexity of the algorithm is explained in following section. □

**Lemma 5.1.** *Let  $v = v_{11}v_{12} \dots v_{1q} \dots v_{j1}v_{j2} \dots v_{jq}v_{temp}$  be the subword encoded by  $(S_{temp}, S_1, S_2)$ , where  $v_{i1} \dots v_{iq}$ , for all  $i \in \{1, 2, \dots, n/2q-1\}$  are the subwords encoded by  $S_1$  (in bottom-up order) as  $(h_i, l_i)$ .  $S_2$  stores  $(\sum_{1 \leq i \leq k} h_i, \sum_{1 \leq i \leq k} l_i)$  (in bottom-up order), where  $k = \sqrt{n}/\log n$ , is the size of stack  $S_1$ .  $v_{temp}$  is the sequence of upsteps in  $S_{temp}$  (in bottom-up order). Then the polynomial of degree  $\frac{q\sqrt{n}}{\log n}$  represented by an entry in  $S_2$  is same as the sum of  $k$  polynomials of degree  $q$ , represented by stack  $S_1$  just before pushing into  $S_2$ .*

*Proof.* For simplicity, assume  $v$  is the continuous sequence of upsteps of only one type *i.e.*  $a$ . So the height of  $v_{ij}$  represented as  $ht(v_{ij}) = \text{height}(x[1, iq + j - 1])$  (just to simplify the notation). Then  $ht(v_{ij}) = 1 + ht(v_{i(j+1)})$ , for  $1 \leq j \leq q$  and  $ht(v_{iq}) = 1 + ht(v_{(i+1)1})$ , for  $i \in \{1, 2, \dots, n/2q - 1\}$ .

Let  $v_{i1}v_{i2} \dots v_{iq}$ , for  $1 \leq i \leq k$ , be the subword encoded as  $(h_i, l_i)$  by the  $i_{th}$  entry of  $S_1$  then,

$$\begin{aligned} h_1 &= \alpha^{ht(v_{11})} \bmod p + \alpha^{ht(v_{12})} \bmod p + \dots + \alpha^{ht(v_{1q})} \bmod p \\ h_2 &= \alpha^{ht(v_{21})} \bmod p + \alpha^{ht(v_{22})} \bmod p + \dots + \alpha^{ht(v_{2q})} \bmod p \\ &\quad \vdots \quad \quad \quad \vdots \quad \quad \quad \vdots \quad \quad \quad \vdots \\ h_k &= \alpha^{ht(v_{k1})} \bmod p + \alpha^{ht(v_{k2})} \bmod p + \dots + \alpha^{ht(v_{kq})} \bmod p \end{aligned}$$

So the sum of  $k$  polynomials of degree  $k$  will be

$$\sum_{1 \leq i \leq k} h_i = \alpha^{ht(v_{11})} \bmod p + \dots + \alpha^{ht(v_{1q})} \bmod p \dots + \alpha^{ht(v_{k1})} \bmod p + \dots + \alpha^{ht(v_{kq})} \bmod p$$

Since the degree of each term is greater than it preceding term, therefore

$$\begin{aligned} \sum_{1 \leq i \leq k} h_i &= h_1 + h_2 + \dots + h_k \\ \sum_{1 \leq i \leq k} l_i &= l_1 + h_2 + \dots + l_k \end{aligned}$$

which is the polynomial in  $\alpha$  of degree  $\frac{q\sqrt{n}}{\log n}$ .

Now consider the case where both types of upsteps ( $a, b$ ) are present. According to the hash function used, hash value for the upsteps of type  $b$  will be zero, but it will still contribute to the height calculation. So these terms will be missing in the polynomial. But the degree of the polynomial depends on the highest degree term which will be from  $a$  type upsteps and will be same for both the polynomials represented by the entries of  $S_1$  and  $S_2$ . Hence the polynomial will be same.  $\square$

### 5.2.2 Space Analysis

This algorithm uses three stacks namely  $S_{temp}, S_1$  and  $S_2$  to store the sequence of upsteps in compressed manner. Each entry in  $S_{temp}, S_1$  and  $S_2$  takes space  $O(1)$ ,  $O(\log n)$  and  $O(\log n)$  respectively. Size of stack  $S_{temp}$  is bounded by  $\lceil \sqrt{n} \rceil$ , therefore it uses space  $O(\sqrt{n})$ . When  $S_{temp}$  (where size of  $S_{temp}$  is  $\sqrt{n}$ ) get filled, a new element is pushed into the  $S_1$  and  $S_{temp}$  is emptied. Size of stack  $S_1$  is bounded by  $\lceil \sqrt{n}/\log n \rceil$ , therefore it uses space  $O(\sqrt{n})$ .

**Lemma 5.2.** *Each entry in  $S_2$  is the sum of  $k$  entries of  $S_1$ . If size of each entry of  $S_1$  is  $O(\log n)$ , then size of entry in  $S_2$  is also  $O(\log n)$ .*

*Proof.* Proof is just a simple calculation of number of bits required to store sum of  $k$   $O(\log n)$  bits numbers. Each entry of  $S_1$  is of size  $O(\log n)$  and there are  $k = \sqrt{n}/\log n$  entries in  $S_1$ . Maximum value an entry can take is  $2^{O(\log n)} = O(n)$ . So the maximum value that sum ( $h^s$ ) of  $k$  entries can take is equal to  $k \cdot O(n) = \sqrt{n}/\log n \cdot O(n) = O(n^{3/2}/\log n) \leq O(n^{3/2})$ . Hence the number of bits required to store the sum  $h^s = O(\log n^{3/2}) = O(\log n)$ .  $\square$

Similarly, when  $S_1$  get filled, a new element is pushed into  $S_2$ . To get new element in  $S_2$ , algorithm has to process at least  $\lceil \frac{\sqrt{n}}{\log n} \times \sqrt{n} \rceil = \lceil \frac{n}{\log n} \rceil$  (size of  $S_1$  times size of  $S_{temp}$ ) letters for every new element in  $S_2$ , which bounds the size of  $S_2$  by  $\log n$ . We know (by lemma 5.2) the size of each entry in  $S_2$  is  $O(\log n)$ . Hence  $S_2$  uses  $O((\log n)^2)$  space. So the total space requirement by the algorithm is  $O(\sqrt{n}) + O(\sqrt{n}) + O((\log n)^2) = O(\sqrt{n})$ .

### 5.2.3 Time Analysis

Processing time of every element in the stream is dominated by the computation of the hash function as it involves modular exponentiation. We have seen in the previous algorithms that time to calculate hash function is  $polylog(n)$ . So in our case also time per letter will be  $polylog(n)$ . The time to perform push and pop operation in the stacks is constant. The time to calculate the sum of entries in  $S_1$  is also constant as we are summing them up in incremental fashion. So the overall processing time of the algorithm is  $O(polylog(n))$  per letter.

### 5.2.4 Error

Error involved in the algorithm is because of the following reasons:

- The probabilistic calculation of prime number in  $polylog(n)$  time[27]. The standard and most efficient procedure to find a prime number  $p$  such that  $n^{1+\gamma} \leq p \leq 2n^{1+\gamma}$ , outputs a prime number with probability  $1 - n^{-\gamma}$  where  $\gamma > 0$  is a constant.
- Hash function, while calculating hash value forms a polynomial of degree  $d$  in  $\alpha$ , where  $d$  is bounded by the maximum height of the stream, at most  $n$  and  $\alpha$  is a randomly selected number in the range  $[1, p - 1]$ . This polynomial of degree  $d$  can have at most  $d$  roots ( $n$  in worst case). So the probability that  $hash(v) = 0$  even when stream is not in  $Dyck(2)$  is at most  $n/p \leq n^{-\gamma}$ .

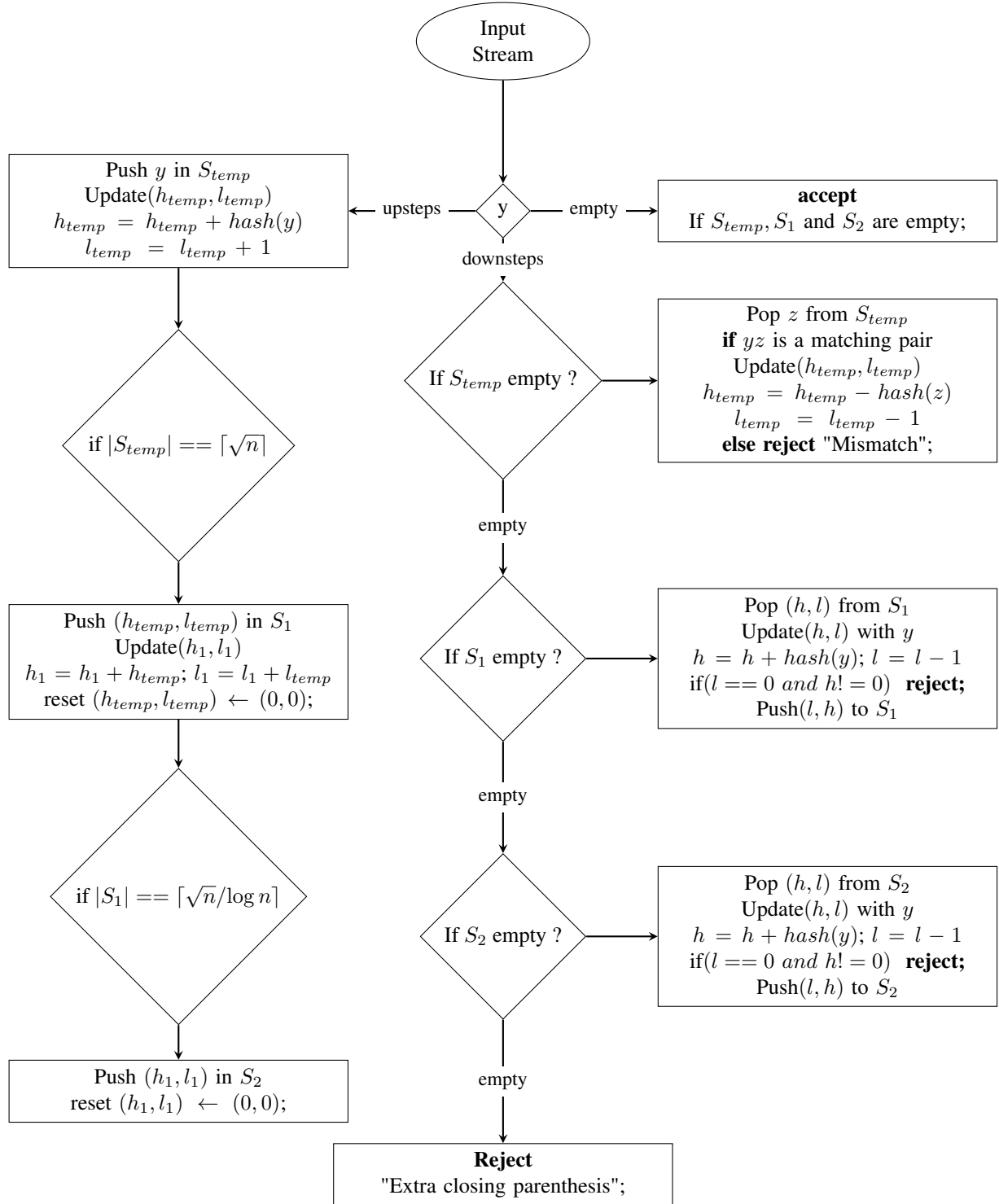


Figure 5.2: Flow chart explains the flow of algorithm, it accepts stream  $x$ , if all three stacks  $S_{temp}, S_1$  and  $S_2$  are empty in the end after processing whole stream. Then input stream  $x \in Dyck(2)$ .

# Chapter 6

## Results

In the table 6 we compare the space, time and error bound of our proposed algorithms with the algorithms presented by Frederic et. al. in [6]. The time bound for both single-pass streaming algorithms and for two-pass bidirectional algorithm is  $O(\text{polylog}(n))$ . All algorithms explained in this report, except the one uses counting bloomfilter, gives only one-sided error. All these algorithms uses the same linear hash function for stack compression and gives same one-sided error  $(1 - n^{-\gamma})$ . The two-sided error in one-pass streaming algorithm with counting bloomfilter is because of the false positives and false negatives of CBF. False positive in bloomfilter is well know problem where as false negative problem of CBF is observed recently by Deke et. al in [28].

The space requirement of existing single-pass streaming algorithm is  $O(\sqrt{n \log n})$ , which is not close enough to the lower bound ( which is  $\Omega(\sqrt{n}/T)$ , for  $T$  pass algorithm ). This algorithm uses two stacks  $S_{temp}$  and  $S$  of size  $\sqrt{n \log n}$  and  $\sqrt{n}/\log n$  respectively, to reduce the space requirement from linear time. In the proposed improved single-pass streaming algorithm, instead of two we are using three stacks  $S_{temp}, S_1$  and  $S_2$  of sizes  $\sqrt{n}, \sqrt{n}/\log n$  and  $\log n$  respectively. By doing so we got the significant improvement in the space requirement from  $O(\sqrt{n \log n})$  to  $O(\sqrt{n})$  which matches the lower bound.

Single-pass streaming algorithm that uses counting bloomfilter takes space  $O(h_{max})$ , where  $h_{max}$  is the maximum height of the input stream. This algorithm directly acts on the  $Dyck(s)$ , avoiding the necessity of streaming reduction. This algorithm saves the  $O(\log s)$  expansion in the length of input stream, and so the space. Even in the worst case when stream is the continuous sequence of upsteps followed by the same number of downsteps, the space requirement will we  $O(\log s)$  time less than the naive approach. If there is any bloomfilter compression technique like stack compression, then we can achieve the similar lower bound as for  $Dyck(2)$ , that too without streaming reduction.

Number of Passes	Time per letter	Space	Error	Remarks
1 (unidirectional)	$O(\text{polylog}(n))$	$O(\sqrt{n \log n})$	$1 - n^{-\gamma}$	uses two stacks $S$ and $S_{temp}$
2 (bidirectional)	$O(\text{polylog}(n))$	$O((\log n)^2)$	$1 - n^{-\gamma}$	uses one stack with binary decomposition
1 (unidirectional)	$O(\text{polylog}(n))$	$O(\sqrt{n})$	$1 - n^{-\gamma}$	uses three stacks and matches lower bound
1 (unidirectional)	$O(1)$	$O(h_{max})$	$fpp$ of CBF	uses CBF and acts on $Dyck(s)$

Table 6.1: Comparison of algorithm

## Chapter 7

# Conclusion and Future Work

In this work we presented two single-pass randomized streaming algorithm, (a) Single-pass Streaming Algorithm using Counting Bloomfilter, (b) Improved Single-pass Streaming Algorithm with  $(\sqrt{n})$  space. Algorithm (a) uses space  $O(h_{max})$  to decide membership in  $Dyck(s)$  and (b) is the extension of existing approach of two stacks to three stacks. In (b) approach we achieved the lower bound  $(\sqrt{n})$ , for the space. We have proved the correctness and space bound for both the algorithm. We proved that error in (a) is one-sided and in (b) error is two-sided because of false positives and false negatives in CBF.

We have seen that algorithm (a) takes space in the order of maximum height of the stream. In worst case height of the stream can be linear, so to reduce the space one can think for some bloomfilter compression techniques and can achieve better bound. The three stacks approach of  $Dyck(2)$  can be used to address the problem of checking priority queues as defined in [12].

# Bibliography

- [1] N. Alon, Y. Matias, and M. Szegedy. The space complexity of approximating the frequency moments. In Proceedings of the twenty-eighth annual ACM symposium on Theory of computing. ACM, 1996 20–29.
- [2] C. Konrad and F. Magniez. Validating XML documents in the streaming model with external memory. *ACM Transactions on Database Systems (TODS)* 38, (2013) 27.
- [3] N. Chomsky and M. Schützenberger. The Algebraic Theory of Context-free Languages. North-Holland Publishing Company, 1961.
- [4] J. E. Hopcroft and J. D. Ullman. Formal languages and their relation to automata .
- [5] R. F. Geary, N. Rahman, R. Raman, and V. Raman. A simple optimal representation for balanced parentheses. *Theoretical Computer Science* 368, (2006) 231–246.
- [6] F. Magniez, C. Mathieu, and A. Nayak. Recognizing well-parenthesized expressions in the streaming model. In Proceedings of the forty-second ACM symposium on Theory of computing. ACM, 2010 261–270.
- [7] M. Blum and S. Kannan. Designing programs that check their work. *Journal of the ACM (JACM)* 42, (1995) 269–291.
- [8] M. Blum, M. Luby, and R. Rubinfeld. Self-testing/correcting with applications to numerical problems. *Journal of computer and system sciences* 47, (1993) 549–595.
- [9] O. Goldreich, S. Goldwasser, and D. Ron. Property testing and its connection to learning and approximation. *Journal of the ACM (JACM)* 45, (1998) 653–750.
- [10] N. Alon, M. Krivelevich, I. Newman, and M. Szegedy. Regular languages are testable with a constant number of queries. *SIAM Journal on Computing* 30, (2001) 1842–1862.
- [11] M. Parnas, D. Ron, and R. Rubinfeld. Testing membership in parenthesis languages. *Random Structures & Algorithms* 22, (2003) 98–138.
- [12] N. François and F. Magniez. Streaming Complexity of Checking Priority Queues. *CoRR* abs/1209.4971.
- [13] B. Saha. Efficiently Computing Edit Distance to Dyck Language. *CoRR* abs/1311.2557.
- [14] A. Chakrabarti, G. Cormode, R. Kondapally, and A. McGregor. Information cost tradeoffs for augmented index and streaming language recognition. *SIAM Journal on Computing* 42, (2013) 61–83.
- [15] R. Jain and A. Nayak. The space complexity of recognizing well-parenthesized expressions. *CoRR* abs/1004.3165.
- [16] A. Krebs, N. Limaye, and S. Srinivasan. Streaming Algorithms for Recognizing Nearly Well-Parenthesized Expressions. In F. Murlak and P. Sankowski, eds., *Mathematical Foundations of Computer Science 2011*, volume 6907 of *Lecture Notes in Computer Science*, 412–423. Springer Berlin Heidelberg, 2011.

- [17] A. Kirsch and M. Mitzenmacher. Building a better bloom filter. In In Proceedings of the 14th Annual European Symposium on Algorithms (ESA. Citeseer, 2005 .
- [18] M. Mitzenmacher. Compressed bloom filters. *IEEE/ACM Transactions on Networking (TON)* 10, (2002) 604–612.
- [19] F. Bonomi, M. Mitzenmacher, R. Panigrahy, S. Singh, and G. Varghese. An improved construction for counting bloom filters. In Algorithms–ESA 2006, 684–695. Springer, 2006.
- [20] A. Kirsch and M. Mitzenmacher. Distance-Sensitive Bloom Filters. In ALENEX, volume 6. SIAM, 2006 41–50.
- [21] A. Kumar, J. Xu, and J. Wang. Space-code bloom filter for efficient per-flow traffic measurement. *Selected Areas in Communications, IEEE Journal on* 24, (2006) 2327–2339.
- [22] S. Cohen and Y. Matias. Spectral bloom filters. In Proceedings of the 2003 ACM SIGMOD international conference on Management of data. ACM, 2003 241–252.
- [23] R. P. Laufer, P. B. Velloso, and O. Duarte. Generalized bloom filters. *COPPE/UFRJ, Tech. Rep. GTA-05-43, Sept* .
- [24] B. Chazelle, J. Kilian, R. Rubinfeld, and A. Tal. The Bloomier filter: an efficient data structure for static support lookup tables. In Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms. Society for Industrial and Applied Mathematics, 2004 30–39.
- [25] O. Rottenstreich, Y. Kanizo, and I. Keslassy. The variable-increment counting Bloom filter. In INFOCOM, 2012 Proceedings IEEE. IEEE, 2012 1880–1888.
- [26] B. Fan, D. G. Andersen, and M. Kaminsky. The Cuckoo Filter: It’s Better Than Bloom .
- [27] E. Bach. Algorithmic Number Theory: Efficient Algorithms, volume 1. 1996.
- [28] D. Guo, Y. Liu, X. Li, and P. Yang. False Negative Problem of Counting Bloom Filter. *IEEE Transactions on Knowledge and Data Engineering* 22, (2010) 651–664.



## Approval Sheet

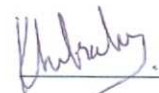
This Thesis entitled Improved Streaming Algorithm for Dyck(s) Recognition by Anubhav Kumar Jain is approved for the degree of Master of Technology from IIT Hyderabad



(Dr. K. S. Rama Murty) Examiner  
Dept. of Electrical Engineering  
Indian Institute of Technology Hyderabad



(Dr. M. V. Panduranga Rao) Examiner  
Dept. of Computer Science and Engineering  
Indian Institute of Technology Hyderabad



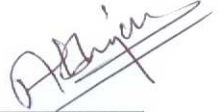
(Dr. Subrahmanyam Kalyanasundaram) Adviser  
Dept. of Computer Science and Engineering  
Indian Institute of Technology Hyderabad



(Dr. C. Krishna Mohan) Chairman  
Dept. of Computer Science and Engineering  
Indian Institute of Technology Hyderabad

## Declaration

I declare that this written submission represents my ideas in my own words, and where ideas or words of others have been included, I have adequately cited and referenced the original sources. I also declare that I have adhered to all principles of academic honesty and integrity and have not misrepresented or fabricated or falsified any idea/data/fact/source in my submission. I understand that any violation of the above will be a cause for disciplinary action by the Institute and can also evoke penal action from the sources that have thus not been properly cited, or from whom proper permission has not been taken when needed.



---

(Signature)

---

(Anubhav Kumar Jain)

CS11M04

---

(Roll No.)