ON THE NUMBER OF DISTINCT SQUARES IN STRINGS

ON THE NUMBER OF DISTINCT SQUARES IN STRINGS

By

MEI JIANG, B.C.S.

A Thesis

Submitted to the Department of Computing and Software

and the School of Graduate Studies

of McMaster University

in Partial Fulfilment of the Requirements

for the Degree of

Doctor of Philosophy

McMaster University © Copyright by Mei Jiang, January 2014

Doctor of Philosophy (2014) McMaster University
(Computing and Software) Hamilton, Ontario, Canada

TITLE: On the number of distinct squares in strings

AUTHOR: Mei Jiang
M.Sc. (Transferred to Ph.D. Program)
McMaster University, Hamilton, Canada
B.C.S.
University of New Brunswick, Fredericton, Canada

SUPERVISOR: Dr. Antoine Deza, Dr. Frantisek Franek

NUMBER OF PAGES: xiv, 118

To my family

Abstract

We investigate the problem of the maximum number of distinct primitively rooted squares in a string. In comparison to considering general strings, the number of distinct symbols in the string is introduced as an additional parameter of the problem. Let $\sigma_d(n) = \max\{s(x) \mid x \text{ is a } (d,n)\text{-string}\}$, where $s(x)$ denotes the number of distinct primitively rooted squares in a string x and a (d,n)-string denotes a string of length n with exactly d distinct symbols.

Inspired by the d-step approach which was instrumental in Santos' tackling of the Hirsch conjecture, we introduce a $(d, n-d)$-table with entries $\sigma_d(n)$ where d is the index for the rows and $n-d$ is the index for the columns. We examine the properties of the $\sigma_d(n)$ function in the context of $(d, n-d)$-table and conjecture that the value of $\sigma_d(n)$ is no more than $n-d$. We present several equivalent properties with the conjecture. We discuss the significance of the main diagonal of the $(d, n-d)$-table, i.e. the square-maximal $(d, 2d)$-strings for their relevance to the conjectured bound for all strings. We explore their structural properties under both assumptions, complying or not complying with the conjecture, with the intention to derive a contradiction. The result yields novel properties and statements equivalent with the conjecture with computational application to the determination of the values $\sigma_d(n)$.

To further populate the $(d, n-d)$-table, we design and implement an efficient computational framework for computing $\sigma_d(n)$. Instead of generating all possible

(d, n)-strings as the brute-force approach needs to do, the computational effort is significantly reduced by narrowing down the search space for square-maximal strings. With an easily accessible lower bound obtained either from the previously computed values inductively or by an effective heuristic search, only a relatively small set of candidate strings that might possibly exceed the lower bound is generated. To this end, the notions of s-cover and the density of a string are introduced and utilized. In special circumstances, the computational efficiency can be further improved by starting the s-cover with a double square structure. In addition, we present an auxiliary algorithm that returns the required information including the number of distinct squares for each generated candidate string. This algorithm is a modified version of FJW algorithm, an implementation based on Crochemore's partition algorithm, developed by Franek, Jiang and Weng. As of writing of this thesis, we have been able to obtain the maximum number of distinct squares in binary strings till the length of 70.

Acknowledgements

I would like to express my gratitude to my supervisors, Dr. Antoine Deza and Dr. Frantisek Franek, for their invaluable guidance, generous support and continuous encouragement to my research studies and my life.

My special thanks go to the members of the supervisory and defence committees: Dr. Antoine Deza, Dr. Frantisek Franek, Dr. Fred Hoppe, Dr. Shmuel Tomi Klein, Dr. Zdislav Kovarik, and Dr. Nedialko Nedialkov.

I appreciate the help and moral support from all my colleagues including the members of Advanced Optimization Laboratory.

Furthermore, I am grateful for the financial aid provided by the Ontario Graduate Scholarship program and the Queen Elizabeth II Graduate Scholarship in Science and Technology program.

Finally, I would like to thank my family and friends from the bottom of my heart for always believing in me, and for their constant love, support and encouragement.

Contents

Abstract **iii**

Acknowledgements **v**

List of Abbreviations and Symbols **xiii**

1 Introduction **1**

1.1 Preliminaries 1

1.1.1 String 1

1.1.2 Repeat 3

1.1.3 Repetition 3

1.1.4 Run 4

1.2 Background 6

1.3 Thesis Outline 7

1.4 Notations 8

2 A d-Step Approach **9**

2.1 Hirsch Conjecture 9

2.2 Auxiliary Lemmas 11

2.3 $(d, n-d)$-Table 14

2.4 Properties of $(d, n-d)$-Table 17
2.5 $\sigma_d(n)$ Conjecture 21

3 Structure of Square-Maximal Strings 24

3.1 Square-Maximal Strings with $\sigma_d(2d) = \sigma_d(2d+1)$ 25
3.2 Square-Maximal Strings with $\sigma_d(2d) > d$ 26
3.2.1 Auxiliary Lemma 26
3.2.2 Pair 27
3.2.3 Triple 28
3.2.4 Singletons Estimation 30
3.3 Additional Combinatorial Property Equivalent with the Conjectured Upper Bound 38

4 Computational Approach 39

4.1 Dense s-Covered Strings 40
4.1.1 The Notion of s-Cover 41
4.1.2 Core Vector 43
4.1.3 Dense Strings 45
4.1.4 s-Covered Strings 45
4.2 Generating s-Covers 49
4.2.1 Algorithm 50
4.2.2 Conflict Check 53
4.2.3 Primitiveness Check 54
4.2.4 No Intermediate Square Check 55
4.2.5 Density Check 57
4.2.6 Parity Condition 59
4.3 Lower Bound Determination 62

4.3.1 Lower Bound $\sigma_d^-(n)$ . . . 63
4.3.2 Lower Bound $\sigma_d^-(2d)$ . . . 64
4.3.3 Heuristic Search for $\sigma_2^-(n)$ . . . 64
4.4 Using Upper Bound to Simplify Computation . . . 68
4.4.1 Double Square . . . 68
4.4.2 Algorithm . . . 69

5 Computing Periodicity **72**
5.1 Crochemore's Repetition Algorithm . . . 73
5.2 FJW Algorithm . . . 76
5.2.1 Data Structures . . . 78
5.2.2 Gap Function . . . 83
5.3 Extend FJW to Produce k-Vector and p-Vector . . . 86
5.3.1 k-Vector . . . 87
5.3.2 p-Vector . . . 90
5.3.3 Data Structures . . . 92

6 Computational Results **93**
6.1 Values in $(d, n-d)$-Table . . . 93
6.1.1 Three Consecutive Equal Values . . . 94
6.1.2 Increasing on Descending Diagonals . . . 95
6.2 Current Bound for $\sigma_d(n)$ . . . 95

7 Conclusion **98**
7.1 Relation to $\rho_d(n)$ . . . 99
7.1.1 Similarities and Differences . . . 99
7.1.2 Differences on Values . . . 102

7.1.3 Strings Achieving Both Square- and Run-Maximality 103
7.2 Future Work 105

A Tables of $\sigma_d(n)$ **108**

A.1 $(d, n-d)$-Table 108
A.2 $(d, n-2d)$-Table 108

B Tables of $\rho_d(n) - \sigma_d(n)$ **111**

B.1 $(d, n-d)$-Table 111
B.2 $(d, n-2d)$-Table 111

List of Tables

2.1 $(d, n-d)$-Table of $\sigma_d(n)$ for $2 \leq d \leq 15$ and $2 \leq n-d \leq 15$. 15

4.1 Comparison on number of s-covered strings and non s-covered strings. 49

4.2 Determining a lower bound for $\sigma_d(n)$ in $(d, n-d)$-table. 63

4.3 Heuristic search parameters for $d = 2$, $34 \leq n \leq 38$. 66

4.4 Strings of length $n{+}1$ with $\sigma_2(n){+}1$ distinct squares for $45 \leq n \leq 46$. 67

4.5 Strings of length $n{+}1$ with $\sigma_2(n){+}1$ distinct squares for $52 \leq n \leq 55$. 67

4.6 Strings of length $n{+}1$ with $\sigma_2(n){+}1$ distinct squares for $n = 50$. . . . 68

7.1 Existence of strings achieving both square- and run-maximality. . . . 104

7.2 $(d, n-d)$-Table of $\rho_d(n) - \sigma_d(n)$ for $2 \leq d \leq 10$ and $2 \leq n-d \leq 10$. . 104

7.3 Dual square/run-maximal strings for $d = 2$, $n = 5, 6, 7, 9$. 105

A.1 $(d, n-d)$-Table with larger d and n. 109

A.2 $(d, n-2d)$-Table with larger d and n. 110

B.3 $(d, n-d)$-Table of $\rho_d(n) - \sigma_d(n)$. 112

B.4 $(d, n-2d)$-Table of $\rho_d(n) - \sigma_d(n)$. 113

List of Figures

4.1 s-Cover of string x. 42

4.2 Core of each square in string x. 43

4.3 Core vector of string x. 44

4.4 Candidate strings. 48

4.5 s-Cover generation. 51

5.1 Example of Crochemore's repetition algorithm. 74

5.2 Core computation: core and new occurrence are not intersected. . . . 89

5.3 Core computation: new occurrence is on the left. 89

5.4 Core computation: core is on the left. 89

5.5 Core computation: core is contained in new occurrence. 90

5.6 p-Vector of string x. 91

5.7 p-Vector computation: for every $s = x[i_1 .. i_2]$ is the leftmost occurrence, $p_i(x) = p_i(x) + 1$, for $i_2 \leq i \leq n$. 92

List of Algorithms

4.1 Conflict check. 54

4.2 Primitiveness check. 56

4.3 No intermediate square check. 57

4.4 Density check. 59

4.5 Double square s-cover generation. 71

List of Abbreviations and Symbols

- $x[1 \,..\, n]$: a string x of length n with indices from 1 to n.
- $x[i]$: the i-th symbol of string x.
- xy: when x and y are strings denotes the concatenation of the two strings.
- ε: empty string.
- $\emptyset$: empty set.
- $|x|$: the size (cardinality) of the set x; or, if x is a string, the length of the string.
- $x_1 \cup x_2$: union of the sets x_1 and x_2; or if x_1 and x_2 are adjacent or overlapping strings, this represents the union of the two strings, i.e. the concatenation of x_1 and x_2 with the overlapping portion removed.
- $\bigcup_{1 \le i \le m} S_i$: represents the union of all S_i where $1 \le i \le m$; that is, $S_1 \cup S_2 \cup \,..\, \cup S_m$.
- $x_1 \cap x_2$: intersection of the sets x_1 and x_2; or if x_1 and x_2 are overlapping strings, this represents the overlapping portion of the two strings.
- $x \subseteq y$: the set x is a subset of the set y; or if x and y are strings, this represents that x is a substring of y.
- (d, n)-string: a string of length n with exactly d distinct symbols.

- $s(x)$: the number of distinct primitively rooted squares in string x.
- $\sigma_d(n)$: the maximum number of distinct primitively rooted squares over all strings of length n with exactly d distinct symbols.
- $\rho_d(n)$: the maximum number of runs over all strings of length n with exactly d distinct symbols.
- $\mathcal{A}(x)$: the alphabet of string x, i.e. the set of all symbols occurring in x.
- *singleton*, *pair*, *triple*, and k-*tuple*: refers to a symbol occurring in a string exactly once, twice, three times, and k times, respectively.

Chapter 1

Introduction

Periodicity is the most studied and important topic in the discipline of *combinatorics on words* and *algorithms on strings*, going back to Thue [33]. It has been applied in many different fields such as data mining, pattern matching, data compression, and computational biology.

This thesis entails our investigation of the problem of the maximum number of distinct primitively rooted squares in strings. In this chapter we first introduce some basic terminology followed by some background information on the problem, and then we present the outline of the thesis and the notation used throughout the thesis.

1.1 Preliminaries

1.1.1 String

A ***string*** x is a finite contiguous sequence of characters referred as ***symbols*** drawn from a non-empty finite ***alphabet*** $\mathcal{A}$. Denote $\mathcal{A}(x)$ the set of distinct symbols that occur in x, then $\mathcal{A}(x) \subseteq \mathcal{A}$ and $|\mathcal{A}(x)|$ is the number of distinct symbols in x. We present a string x as an array $x[1\ ..\ n]$, $n \geq 1$, of symbols, where $n = |x|$ is the ***length***

of the string and $x[i]$ represents a symbol at position i. The empty string is denoted as ε. We use $x[i \mathinner{..} j]$ to denote a substring of x starting at position i of length $j-i+1$, where $1 \le i \le j \le n$. Using this notation we can write a ***prefix*** of x as $x[1 \mathinner{..} i]$ where $1 \le i \le n$. A prefix is said to be a ***proper prefix*** when $i < n$. Similarly, a ***suffix*** of x is denoted as $x[i \mathinner{..} n]$ where $1 \le i \le n$; a suffix is said to be a ***proper suffix*** if $i > 1$. Consider an example $x[1 \mathinner{..} 3] = aba$, substring $x[1 \mathinner{..} 2] = ab$ is a proper prefix of x, and $x[2 \mathinner{..} 3] = ba$ is a proper suffix of x.

The basic operation for strings is ***concatenation***; that is, joining two strings together into one. The concatenation of two strings is denoted in the order that they are concatenated. For instance, xy represents a string that is the concatenation of string x followed by string y; that is, $xy = x[1]x[2]\cdots x[i]y[1]y[2]\cdots y[j]$, where $x = x[1 \mathinner{..} i]$ and $y = y[1 \mathinner{..} j]$. Let us consider $x = ab$ and $y = b$, then $xy = abb$ and $yx = bab$.

The ***union*** $x[i_1 \mathinner{..} i_k] \ \cup \ x[j_1 \mathinner{..} j_m]$ of two substrings of a string $x = x[1 \mathinner{..} n]$ is defined if $i_1 \le j_1 \le i_k + 1$ and then $x[i_1 \mathinner{..} i_k] \cup x[j_1 \mathinner{..} j_m] = x[i_1 \mathinner{..} \max\{i_k, j_m\}]$; or if $j_1 \le i_1 \le j_m + 1$ and then $x[i_1 \mathinner{..} i_k] \cup x[j_1 \mathinner{..} j_m] = x[j_1 \mathinner{..} \max\{i_k, j_m\}]$. Simply put, the union is defined when the two substrings either are adjacent or overlapped. For instance, $x[1 \mathinner{..} 8] = abaababa$, substring $x_1 = x[2 \mathinner{..} 5] = baab$ and $x_2 = x[4 \mathinner{..} 6] = aba$, then $x_1 \cup x_2 = x[2 \mathinner{..} 6] = baaba$.

Similarly, the ***intersection*** $x[i_1 \mathinner{..} i_k] \ \cap \ x[j_1 \mathinner{..} j_m]$ of two substrings of a string $x = x[1 \mathinner{..} n]$ is defined only when the two substrings are overlapped; that is, if $i_1 \le j_1 \le i_k$ and then $x[i_1 \mathinner{..} i_k] \cap x[j_1 \mathinner{..} j_m] = x[j_1 \mathinner{..} \min\{i_k, j_m\}]$, or if $j_1 \le i_1 \le j_m$ and then $x[i_1 \mathinner{..} i_k] \cap x[j_1 \mathinner{..} j_m] = x[i_1 \mathinner{..} \min\{i_k, j_m\}]$. Consider the same example above, $x_1 \cap x_2 = x[4 \mathinner{..} 5] = ab$.

1.1.2 Repeat

A ***repeat*** is a collection of identical repeating substrings in $x[1 \,..\, n]$ described by $u = x[i_1 \,..\, i_1+p-1] = x[i_2 \,..\, i_2+p-1] = \cdots = x[i_q \,..\, i_q+p-1]$, where $1 \leq i_1 < i_2 < \cdots < i_q \leq n$, $q \geq 2$, and $p \geq 1$. The repeating substring u is the ***generator*** of the repeat, and $|u| = p$. For example, $x[1 \,..\, 5] = abbab$ contains a repeat with generator ab at position 1 and 4.

1.1.3 Repetition

A ***repetition*** or a tandem repeat is a repeat with adjacent repeating substrings; that is, for every two consecutive identical substrings in a repeat, the gap between their starting positions is equal to the length of the generator. A repetition with generator u repeating q times can be presented as u^q, where u is a non-empty string, and $q \geq 2$. We refer $p = |u|$ the ***period*** and q the ***exponent*** (or ***power***) of the repetition. If u is irreducible, i.e. itself is not a repetition, then we say u is ***primitive***, and u^q is a ***primitively rooted repetition***. A ***square*** is a repetition with power of 2. If u is primitive, then u^2 is a ***primitively rooted square***. Consider square $(ababab)^2$ is not a primitively rooted square as the generator $ababab$ is not primitive, i.e. $ababab$ is a repetition and can be written in the form of $(ab)^3$.

If a repetition $u^q = x[s \,..\, s+qp-1]$ where $p = |u|$, can be extended by another copy of u to the left of x, that is, $x[s-p \,..\, s-1] = u$, then we say the repetition can be extended to the left. A repetition is called a ***left-maximal repetition*** if it cannot be extended to the left. Similarly, if we cannot extend another copy of u to the right, then the repetition is a ***right-maximal repetition***. A ***maximal repetition*** refers to a repetition that cannot be extended to either left or right. In the context of this thesis, when we use the term repetition, we mean maximal repetition. Consider

$x[1 \mathinner{..} 7] = ababaa$, repetition $x[3 \mathinner{..} 6]$ is right-maximal, but it's not left-maximal since there is another copy of ab at $x[1 \mathinner{..} 2]$, similarly, repetition $x[1 \mathinner{..} 4]$ is left-maximal but not right-maximal; and repetition $x[1 \mathinner{..} 6]$ is a maximal repetition since it's not extendible to either left or right.

A repetition $u^q = x[s \mathinner{..} s + qp - 1]$ where $p = |u|$, is ***left shiftable*** if $x[s - 1]$ is defined (i.e. $s > 1$) and $x[s - 1] = x[s + p - 1]$; that is, when we move one position to the left, $x[s - 1 \mathinner{..} s + qp - 2]$ is also a repetition with the same period of p. Similarly, ***right shiftable*** can be defined as $x[s + qp]$ is defined (i.e. $s + qp \leq |x|$) and $x[s + (q - 1)p] = x[s + qp]$. Consider the same example used above $x[1 \mathinner{..} 7] = abababa$, maximal repetition $x[1 \mathinner{..} 6] = ababab$ is right shiftable because $x[5] = x[7]$, i.e. $x[2 \mathinner{..} 7] = bababa$ is also a repetition with the same period when moving one position to the right.

A repetition can be encoded in a triple of a form $(\boldsymbol{s}, \boldsymbol{e}, \boldsymbol{p})$ where s specifies the starting position, e specifies the ending position, and p is the period of the repetition. The exponent of the repetition can be calculated by $(e - s + 1)/p$. For instance, $x[1 \mathinner{..} 7] = abababb$ has three repetitions: $(1, 6, 2)$ represents $(ab)^3$, $(2, 5, 2)$ represents $(ba)^2$, and $(6, 7, 1)$ represents $(b)^2$; where the last two are squares.

1.1.4 Run

The notion of ***run*** that captures maximal repetitions was first introduced by Main [30], where it was called ***maximal periodicity***. The term *maximal repetition* short for ***maximal fractional repetition*** is also used [27]. To avoid the ambiguity, we refer to maximal repetitions as repetitions, not runs. A run is formed by a maximal repetition followed by a proper prefix (possibly empty) of the generator we call the ***tail*** of the run. Thus, a repetition is a run with an empty tail. If the tail of a run is

non-empty, we say it is a ***non-trivial run***. For example, $x[1 \mathinner{..} 10] = abaabaabaa = (aba)^3a$ (formed by repetition $(aba)^3$ followed by a prefix of the generator a) is a run with period $p = |aba| = 3$. A run can be also seen as a fractional repetition as x contains 3 full generators and $\frac{1}{3}$ partial generator, i.e. the tail. Formally, we define a run as follows:

Definition 1.1. *A string $u^qu' = x[s \mathinner{..} s+qp+t-1]$ where u' is a proper prefix (possibly empty) of u, $|u| = p$, and $|u'| = t$, is a run if:*

- *$x[s+i \mathinner{..} s+i+qp-1]$ is a maximal repetition with period of p, for every $0 \le i \le t$;*
- *$x[s-1]$ is defined and $x[s-1] \ne x[s+p-1]$, i.e. repetition $x[s \mathinner{..} s+qp-1]$ is not left shiftable;*
- *$x[s+t+qp]$ is defined and $x[s+t+(q-1)p] \ne x[s+t+qp]$, i.e. repetition $x[s+t \mathinner{..} s+t+qp-1]$ is not right shiftable;*
- *u is primitive.*

A run thus is a compressed form of one or more repetitions. Consider the example mentioned above, $x[1 \mathinner{..} 10] = abaabaabaa = (aba)^3a$ encodes three maximal repetitions: (1,9,3) represents $(aba)^3$, (2,10,3) represents $(baa)^3$, and (3, 8, 3) represents $(aab)^2$. Note that the exponent of the last repetition $(aab)^2$ is 2 while the exponent of the first two repetitions is 3. We refer the first repetition in a run as the *leading repetition*. The rotation of a string is a string resulting from moving the first symbol to the end of the original string, i.e. *baba* is a rotation of *abab*. In general, a run with period of p, exponent of q, and tail length of t, contains $t+1$ repetitions (leading repetition and its t rotations) with exponent of q, and $p-t-1$ repetitions with exponent of $q-1$. For $q = 2$, a run contains $t+1$ squares.

A run can also be encoded as a triple (s, e, p) where s, e, and p specifies the starting position, the ending position, and the period of the run, respectively. The exponent and the tail size of the run can be easily computed through equations $q = (e - s + 1)/p$ (integer division) and $t = (e - s + 1)\%p$ (modulus), respectively. For example, $x[1 \mathinner{..} 10] = abaabaabaa$ is encoded as $(1, 10, 3)$ with exponent $q = (10 - 1 + 1)/3 = 3$, and tail length $t = (10 - 1 + 1)\%3 = 1$.

1.2 Background

While all forms of periodicities are important for investigation of properties of strings, the problem we are interested in is the maximum number of distinct primitively rooted squares when the types of squares rather than their occurrences are counted. The problem of counting the occurrences of squares is essentially the problem of counting runs; Baker, Deza, and Franek investigated the problem of the maximum number of runs using a similar approach to ours and entailed the results in [1]. We elaborate on the interrelations between the investigations of the two problems in Chapter 7. Kubica et al. [28] showed the number of distinct non-primitively rooted squares is bounded by $\lfloor \frac{n}{2} \rfloor - 1$ where n is the length of the string, therefore it is worthwhile to focus on the problem of distinct primitively rooted squares.

In 1998 Fraenkel and Simpson [14] showed that the maximum number of distinct squares (not necessarily primitively rooted) in a string of length n is bounded by $2n$. Based on the empirical evidence, they conjectured that the upper bound should be n. Ilie [21] first gave a simpler proof of the result and later provided an asymptotic upper bound of $2n - \Theta(\log n)$ in 2007 [22]. The most recent results on the bound of distinct primitively rooted squares in strings was given by Deza, Franek, and Thierry [13]. They proved that there are at most $\frac{11n}{6}$ distinct squares in a string of length n,

by showing there are at most $\frac{5n}{6}$ double squares in the string.

1.3 Thesis Outline

As opposed to considering it for general length, our approach to the problem of distinct primitively rooted squares is that we introduce the size of the alphabet, i.e. the number of distinct symbols in the string, as a parameter into the problem. We explore the problem on both theoretical and computational levels. For the theoretical aspects, we investigate how the function of the maximum number of distinct squares behaves with respect to the length and the number of distinct symbols of the input string in the context of a so-called $(d, n-d)$-table in Chapter 2; and a conjectured upper bound is introduced and a series of reformulation to the conjecture are discussed in the chapter. In Chapter 3, we examine the combinatorial properties of the square-maximal strings that are under both assumptions of complying and not complying with the conjectured upper bound; these structural insights not only point to a direction of possibly proving the conjecture, but can also be applied in the computational aspects of the problem. Chapter 4 describes a computational framework to efficiently compute the maximum number of distinct squares in strings exploiting the combinatorial properties of the square-maximal strings. In addition, the underlying algorithm that returns the required information for the computational framework is discussed in Chapter 5. Some of the main computational results are shown in Chapter 6. Finally, we briefly discuss the interrelation to the problem of runs investigated using a similar approach, and some opportunities for future work of the problem in Chapter 7.

1.4 Notations

We use the following notations throughout this thesis. A ***(d, n)-string*** denotes a string of length n with exactly d distinct symbols. ***$s(x)$*** denotes the number of distinct primitively rooted squares in a string x. As a convention, when we use the term *distinct squares*, we always refer to the distinct primitively rooted squares. ***$\sigma_d(n)$*** denotes the maximum number of distinct primitively rooted squares in strings with length n and number of distinct symbols d; that is, $\sigma_d(n) = \max\{s(x) \mid x \text{ is a } (d, n)\text{-string}\}$. A ***square-maximal*** string refers to a string that contains the maximum number of distinct primitively rooted squares; that is, a (d, n)-string x is square-maximal when $s(x) = \sigma_d(n)$. ***$\mathcal{A}(x)$*** denotes the alphabet of a string x and thus $|\mathcal{A}(x)| = d$ is the number of distinct symbols in x. A ***singleton*** of x refers to a symbol in a string x that occurs exactly once. Similarly, a ***pair***, a ***triple*** and in general a ***k-tuple*** of x refers to a symbol that occurs in x exactly twice, three times and k times, respectively.

Chapter 2

A d-Step Approach

In this chapter we discuss a theoretical approach to the number of distinct squares problem. We have adopted a so-called d-step approach which was used originally in the investigation of the Hirsch conjecture problem. We briefly introduce the Hirsch conjecture and the d-step technique, followed by several auxiliary lemmas which will be used later in the thesis. Then we discuss how we apply the d-step approach to the problem of the maximum number of distinct squares to generate the so-called $(d, n-d)$-table and the basic properties of $\sigma_d(n)$ are exhibited. In Section 2.4, we show and prove several more complex properties of $\sigma_d(n)$. In the last section of this Chapter, we conjecture an upper bound for the number of distinct primitively rooted squares, and present several propositions equivalent with the conjectured upper bound. These properties may indicate possible ways to tackle the problem. Most of the contents in this chapter were reported in [11] and some in [12].

2.1 Hirsch Conjecture

Before we look into the Hirsch conjecture, let us define some basic terminologies.

Definition 2.1. *Let $a \in \mathbb{R}^d$, $a \neq 0$, and $c \in \mathbb{R}$, the set of $x \in \mathbb{R}^d$ satisfying $a^T x = c$ is called a hyperplane.*

If we replace the above equality with inequality, we get a ***half space***. When the inequality includes the equality, then it is a ***closed half space***, otherwise it is an ***open half space***. A ***polyhedron*** is an intersection of finite number of closed half spaces. A bounded polyhedron is called a ***polytope***. A ***face*** of a polytope P is the intersection of P with a supporting hyperplane, a hyperplane that intersects the boundary but not the interior of P. A face of dimension k is called a ***k*-face**. 0-faces are called ***vertices***, 1-faces are called ***edges***, and $(d-1)$-faces are called ***facets***. The facets of a d-dimensional polytope are $(d-1)$-faces with one dimension less than the polytope itself. For instances, the facets of a line are its 0-faces or vertices. The ***diameter*** $\boldsymbol{d}(P)$ of a polytope P is the smallest integer such that any pair of vertices of P can be connected by an edge-path of length $\boldsymbol{d}(P)$ or less. A ***(d,n)-polytope*** is a polytope of dimension d having n facets. Let $\boldsymbol{\Delta(d,n)}$ denote the maximum possible diameter over all (d,n)-polytopes. A ***$(d,n-d)$-table*** is a table with entries of $\Delta(d,n)$, rows are indexed by d and columns are indexed by $n-d$. Note that the entries on the ***main diagonal*** of the table are $\Delta(d,2d)$, which are the maximum diameters for $(d,2d)$-polytopes.

The Hirsch conjecture, first posed in 1957 by Hirsch [8] and stating that $\Delta(d,n) \leq n-d$, was disproved by Santos [32] in 2012 by exhibiting a violation on the main diagonal of the $(d,n-d)$-table; specifically, a $(43,86)$-polytope with a diameter more than 43. The associated $(d,n-d)$-table exhibits similar regularities as the $(d,n-d)$-table for $\sigma_d(n)$ we will present in Section 2.3. Namely, corresponding to Proposition 2.6, it is known that $\Delta(d,n) \leq \Delta(d,n+1)$, $\Delta(d,n) \leq \Delta(d+1,n+1)$, and $\Delta(d,n) < \Delta(d+1,n+2)$ for $n \geq d \geq 2$; and that $\Delta(d,n) = \Delta(d+1,n+1)$ for $2d \geq n \geq d \geq 2$.

In other words, the maximum of $\Delta(d, n)$ within a column in the $(d, n-d)$-table is achieved on the main diagonal and all values below a value on the main diagonal are equal to that value; which is also true for the $(d, n-d)$-table of $\sigma_d(n)$. The role played by the main diagonal of the $(d, n-d)$-table was underlined in 1967 by Klee and Walkup [24] who showed the equivalency between the Hirsch conjecture and the d-step conjecture stating that $\Delta(d, 2d) \leq d$ for all $d \geq 2$ (corresponding to Theorem 2.14). Note that the d-cube is a $(d, 2d)$-polytope having diameter d and therefore $\Delta(d, 2d) \geq d$ for any d (corresponding to Lemma 2.7). In other words, the $(d, 2d)$-string $aabbcc\cdots$ consisting of d pairs can be viewed as an analogue of the d-cube. We examine the structure of square-maximal $(d, 2d)$-strings in Chapter 3.

2.2 Auxiliary Lemmas

Here we present auxiliary lemmas that are frequently used in later sections. Lemma 2.2 is quite intuitive and we provide a formal proof that a removal of a singleton or appending one does not reduce the number of distinct squares in a string. Lemma 2.4 shows that, if the frequency of every symbol in a square-maximal string does not exceed 3, then every pair can be placed adjacent to each other in the end of the string while keeping its maximality. Lemma 2.5 is a stronger version of Lemma 2.4 and states that if the frequency of every symbol is less or equal than 2, then every pair must be adjacent. Lemma 2.3 is used by Lemma 2.4 and other proofs in the thesis, it shows that if a pair occurs in more than one square, then in fact it occurs in a non-trivial run formed by all these squares.

Lemma 2.2. *The singletons of a string can be removed or appended to end of the string without reducing the number of distinct squares in the string.*

Proof. Let x be a string containing a singleton C at a position i, that is $x[i] = C$. We

can write the string as $x = x[1 .. i-1]Cx[i+1 .. n]$.

If we remove C or move it to the end, the string becomes either $x' = x[1 .. i-1]x[i+1 .. n]$ or $x'' = x[1 .. i-1]x[i+1 .. n]C$. Since $x[i]$ is the only occurrence of C in x, C is not in any squares as a square would require at least 2 occurrences of every symbol involved in the square. Therefore, we do not destroy any existing squares. However, the concatenation of $x[1 .. i-1]$ and $x[i+1 .. n]$ may create new squares, but they might be of a type already existing. Thus, $s(x) \leq s(x') = s(x'')$. □

Lemma 2.3. *If a pair of C's occurs in two or more squares in a string x, then all these squares form a non-trivial run.*

Proof. Without loss of generality, let us assume C's occur in two distinct squares $uCvuCv$ and $u'Cv'u'Cv'$, where $v \neq v'$ and $u \neq u'$. Since $vu = v'u'$ as the symbols occur between the C's are the same for both squares, the periods of the two squares are the same as $|uCv| = |Cvu|$, $|u'Cv'| = |Cv'u'|$ and $|Cvu| = |Cv'u'|$. The length of the overlapping portion of the two squares is at least $|CvuC| = |Cv'u'C|$ which is more than the period. Thus, $uCvuCv$ and $u'Cv'u'Cv'$ are rotations of each other and therefore form a non-trivial run. □

Lemma 2.4. *Let x be a square-maximal (d, n)-string, and every symbol of x occurs at most 3 times. Then every pair in x can be moved together so that they are adjacent in the end of the string, without destroying the maximality of x.*

Proof. Let us suppose that there is a non-adjacent pair of C's in x.

(i) If the C's do not occur in any squares, then we could move both C's to the end of the string. This would not destroy any squares of x, but we gain a new square CC which contradicts the maximality of x.

(ii) If the C's occur in exactly one square $uCvuCv$ (where u and v are some strings and at least one of them is non-empty), we can move both C's to the end of x to form a new string y. The squares created by this move are $uvuv$ and CC while the old square $uCvuCv$ is destroyed ($uCvuCv$ does not exist in any other part of x since C has only 2 occurrences and it would require C to appear at least 3 times in order to have another occurrence of $uCvuCv$). If $uvuv$ is a new square type and does not exist in any other part of x, then $s(y) = s(x) + 2 - 1 > s(x)$ which contradicts the maximality of x; if $uvuv$ already existed in some other part of x, we should not consider $uvuv$ a new square, then we only gained CC and lost $uCvuCv$, therefore $s(y) = s(x) + 1 - 1 = s(x)$.

(iii) If the C's occur in more than one square, these squares must form a non-trivial run by Lemma 2.3. Let the form of such non-trivial run be $tuCvtuCvt$, where t is a non-empty string with each symbol occurs 1 time (total of 3 times). Without loss of generality, let us assume t has only one symbol. If $u = v = \varepsilon$, then the run is $tCtCt$ containing two distinct squares $tCtC$ and $CtCt$. We can move the C's to the end forming $tttCC$, destroying the two squares, but gaining two new squares tt and CC. Thus, the number of distinct squares in x is unchanged. If either $u \neq \varepsilon$ or $v \neq \varepsilon$, then by moving both C's to the end of x, we destroyed the two distinct squares $tuCvtuCv$ and $uCvtuCvt$, but we gained three new squares types $tuvtuv$, $uvtuvt$, and CC. Note that neither $tuvtuv$ nor $uvtuvt$ can exist anywhere else in x because that would require each symbol in t occurs at least 4 times which it is not possible. Thus, we have more distinct squares than x, which contradicts the maximality of x.

Therefore, we have shown for a square-maximal string with each symbol occurring no more than 3 times, either non-adjacent pairs do not exist or, if they do, we can

safely move the pairs together to the end without reducing the number of distinct squares in the string. □

Lemma 2.5. *Let x be a square-maximal (d, n)-string, and every symbol of x occurs at most 2 times. Then every pair in x must be adjacent.*

Proof. The proof is similar as Lemma 2.4. Let us suppose that there is a non-adjacent pair of C's in x.

(i) If the C's do not occur in any squares, move both C's to the end of string. By this we gain a new square CC which contradicts the maximality of x.

(ii) If the C's occur in at least one square, let us move the two C's to the end of the string. $uCvuCv$ (where u and v are strings and at least one of them is non-empty) is destroyed by the removal of the C's, and $uvuv$ is created. The square $uvuv$ is a new square type because if $uvuv$ already existed in some other part of x, every symbol of uv would have to occur in x at least 3 times, which is not possible since every symbol of x occurs at most twice. Thus, the destroyed square $uCvuCv$ is replaced by a new square $uvuv$, in addition we gain a new square CC. This contradicts the maximality of x.

Therefore, every pair in x must be adjacent. □

2.3 $(d, n-d)$-Table

Inspired by the $(d, n-d)$-table used for investigating the Hirsch bound for the diameter of polytopes as discussed in Section 2.1, we adopt the similar approach to the problem of distinct squares in strings. A $(d, n-d)$-table with entries of $\sigma_d(n)$, whose rows are indexed by the number of distinct symbols d and columns are indexed by the difference

of the string length and number of distinct symbols $n-d$, is constructed. A fragment of the table for $d \leq 15$ and $n-d \leq 15$ is shown in Table 2.1. A larger portion of the table is shown in Table A.1, and the completed table with up to date values is maintained in [10]. Note that we only consider the maximum number of distinct primitively rooted squares for $d \geq 2$ and $n-d \geq 2$ since $\sigma_d(n) = 1$ for $d = 1$ or $n-d = 1$ are the trivial cases.

d \ $n-d$	2	3	4	5	6	7	8	9	10	11	12	13	14	15
2	**2**	2	3	3	4	5	6	7	7	8	9	10	11	12
3	2	**3**	3	4	4	5	6	7	8	8	9	10	11	12
4	2	3	**4**	4	5	5	6	7	8	9	9	10	11	12
5	2	3	4	**5**	5	6	6	7	8	9	10	10	11	12
6	2	3	4	5	**6**	6	7	7	8	9	10	11	11	12
7	2	3	4	5	6	**7**	7	8	8	9	10	11	12	12
8	2	3	4	5	6	7	**8**	8	9	9	10	11	12	13
9	2	3	4	5	6	7	8	**9**	9	10	10	11	12	13
10	2	3	4	5	6	7	8	9	**10**	10	11	11	12	13
11	2	3	4	5	6	7	8	9	10	**11**	11	12	12	13
12	2	3	4	5	6	7	8	9	10	11	**12**	12	13	13
13	2	3	4	5	6	7	8	9	10	11	12	**13**	13	14
14	2	3	4	5	6	7	8	9	10	11	12	13	**14**	14
15	2	3	4	5	6	7	8	9	10	11	12	13	14	**15**

Table 2.1: $(d, n-d)$-Table of $\sigma_d(n)$ for $2 \leq d \leq 15$ and $2 \leq n-d \leq 15$.

Viewing the table, one can make several observations, such as: the entries are non-decreasing from left to right per row, and the entries are non-decreasing from top to down per column. Proposition 2.6 summarizes the elementary properties of the $(d, n-d)$-table.

Proposition 2.6. *For any* $2 \leq d \leq n-2$*:*

(a) $\sigma_d(n) \leq \sigma_d(n+1)$*, i.e. the values are non-decreasing when moving left-to-right along a row.*

(b) $\sigma_d(n) \leq \sigma_{d+1}(n+1)$*, i.e. the values are non-decreasing when moving top-to-bottom along a column.*

(c) $\sigma_d(n) < \sigma_{d+1}(n+2)$*, i.e. the values are strictly increasing when moving top-left to bottom-right along a descending diagonal.*

(d) $\sigma_d(n) = \sigma_{d+1}(n+1)$ *for* $n \leq 2d$*, i.e. the values under and on the main diagonal along a column are constant.*

Proof. (a) Let x be a square-maximal (d, n)-string. Let y be x appended with a symbol $a \in \mathcal{A}(x)$, then y is a $(d, n+1)$-string. By doing this, we will not destroy any squares in x, and we may have created a new square; for example, if the last symbol in x is a, and x does not contain a square aa, therefore we have created a new square aa by adding a symbol a in the end of x. Thus, $\sigma_d(n) = s(x) \leq s(y) \leq \sigma_d(n+1)$.

(b) Let x be a square-maximal (d, n)-string. Let y be x appended with a symbol $a \notin \mathcal{A}(x)$, then y is a $(d+1, n+1)$-string. Since a is a new symbol and does not occur in x, it will not destroy or create squares for x. Therefore, by adding a in the end of x will not change the number of distinct squares in x. Thus, $\sigma_d(n) = s(x) = s(y) \leq \sigma_{d+1}(n+1)$.

(c) Let x be a square-maximal (d, n)-string. Let y be x concatenated with aa in the end, where $a \notin \mathcal{A}(x)$, then y is a $(d+1, n+2)$-string. With the similar argument in the proof of (b), this will not destroy or create squares for x; however, aa is a square itself and does not occur in x, therefore y is guaranteed to have one more square than x. Thus, $\sigma_d(n) = s(x) < s(x) + 1 = s(y) \leq \sigma_{d+1}(n+2)$.

(d) Let $n \leq 2d$ and x be a square-maximal $(d+1, n+1)$-string. Suppose every symbol in x occur at least twice, then we would need $2(d+1)$ symbols, since $2d \geq n$,

$2(d+1) \geq n+2$, but we only have $n+1$ symbols in x, therefore x contains at least one singleton. Let y be x with the singleton removed, then y is a (d,n)-string. By auxiliary Lemma 2.2, this will not destroy any squares while some squares may be created when left portion and right portion of the string are concatenated. Thus, $\sigma_d(n) \geq s(y) \geq s(x) = \sigma_{d+1}(n+1)$. By (b), $\sigma_d(n) \leq \sigma_{d+1}(n+1)$. Therefore, $\sigma_d(n) = \sigma_{d+1}(n+1)$ for $n \leq 2d$. When $n = 2d$, $\sigma_d(n)$ is a value on the main diagonal of the $(d, n-d)$-table.

□

When we combine the fact that entries are non-decreasing in every column by Proposition 2.6 (b), and entries on and under the main diagonal entry are constant in every column by Proposition 2.6 (d), we can conclude that the maximum entry in every column is the entry on the main diagonal of the $(d, n-d)$-table; namely, $\sigma_d(2d)$.

2.4 Properties of $(d, n-d)$-Table

In this section, we present a set of more complex properties of the $(d, n-d)$-table. Lemma 2.7 shows the fact that the entries on and under the main diagonal are at least as big as its column index (i.e. the conjectured value we will show in Section 2.5). Lemma 2.8 provides the lower bound for the two entries immediately on the right neighbourhood of the main diagonal entry.

Lemma 2.7. *For any* $2 \leq d$, $d+2 \leq n \leq 2d$, $\sigma_d(n) \geq n-d$.

Proof. Let $n \leq 2d$ and consider string $x = aabbcc\ldots$ consisting of $n-d$ adjacent pairs. Then x is a $(n-d, 2n-2d)$-string and $s(x) = n-d$. By Proposition 2.6(d), any value under the main diagonal equals to the value on the main diagonal along a column. Thus, $\sigma_d(n) = \sigma_{n-d}(2n-2d) \geq s(x) = n-d$. □

Lemma 2.8. *For any* $2 \le d$, $\sigma_d(2d+1) \ge d$ *and* $\sigma_d(2d+2) \ge d+1$.

Proof. Consider string $x = aaabbcc\ldots$ consisting of $d-1$ adjacent pairs except the first three symbols being aaa. Then x is a $(d, 2d+1)$-string and $s(x) = d$. Thus, $\sigma_d(2d+1) \ge s(x) = d$. Similarly, consider string $y = aababaccdd\ldots$ consisting of $d-2$ adjacent pairs except the first six symbols being $aababa$ containing three distinct squares aa, $abab$ and $baba$. Then y is a $(d, 2d+2)$-string and $s(y) = d+1$. Thus, $\sigma_d(2d+2) \ge s(y) = d+1$. □

Lemma 2.9 shows the difference between the value on the main diagonal and the value immediately above it is no more than 1.

Lemma 2.9. *For any* $3 \le d$, $\sigma_d(2d) - \sigma_{d-1}(2d-1) \le 1$.

Proof. Let x be a square-maximal $(d, 2d)$-string.

(i) If x has a singleton, let y be x with the singleton removed, then y is a $(d-1, 2d-1)$-string and $s(y) \ge s(x)$. It follows that $\sigma_d(2d) = s(x) \le s(y) \le \sigma_{d-1}(2d-1)$, and since $\sigma_d(2d) \ge \sigma_{d-1}(2d-1)$ by Proposition 2.6(b), we get $\sigma_d(2d) = \sigma_{d-1}(2d-1)$.

(ii) If x does not have a singleton, then x only consists of pairs. By Lemma 2.5, all pairs in x must be adjacent. Thus, $\sigma_d(2d) = s(x) = d$. By Lemma 2.8, $\sigma_{d-1}(2d-1) \ge d-1$. Therefore, $\sigma_{d-1}(2d-1) \ge \sigma_d(2d) - 1$, i.e., $\sigma_d(2d) - \sigma_{d-1}(2d-1) \le 1$.

□

Based on Fraenkel and Simpson's lemma [14], Lemma 2.10 and Lemma 2.11 show that in the $(d, n-d)$-table the difference between any two consecutive entries along a row, or along the main diagonal, is bounded by 2. They essentially provide an upper bound for the next unknown entry when computing the value of $\sigma_d(n)$ inductively

in the $(d, n-d)$-table, as we discuss the details of the computational framework for distinct squares in Chapter 4.

Lemma 2.10. *For any* $2 \leq d \leq n-2$, $\sigma_d(n+1) - \sigma_d(n) \leq 2$.

Proof. Let x be a square-maximal $(d, n+1)$-string. Without loss of generality we can assume that the first symbol in x is not a singleton because otherwise we can move all singletons from the beginning of x to the end of x without destroying any squares by Lemma 2.2. Remove the first symbol of x to form string y. Then y is a (d, n)-string as the number of symbols d is unchanged because the first symbol is not a singleton. By Fraenkel and Simpson [14], there are at most two rightmost occurrences of square types starting at the same position in a string. In other words, the removal of the first symbol destroyed at most two distinct squares. That is, $s(x) - 2 \leq s(y)$. Therefore, $\sigma_d(n+1) - 2 \leq s(y) \leq \sigma_d(n)$, implying $\sigma_d(n+1) - \sigma_d(n) \leq 2$. □

Lemma 2.11. *For any* $2 \leq d$, $\sigma_{d+1}(2d+2) - \sigma_d(2d) \leq 2$.

Proof. By Lemma 2.10, $\sigma_{d+1}(2d+2) - \sigma_{d+1}(2d+1) \leq 2$. By Proposition 2.6 (d), the entries under and on the main diagonal along a column are constant; that is, $\sigma_{d+1}(2d+1) = \sigma_d(2d)$. Therefore, $\sigma_{d+1}(2d+2) - \sigma_d(2d) \leq 2$. □

Lemma 2.12 shows that the two entries immediately above the main diagonal entry along a column are identical.

Lemma 2.12. *For any* $3 \leq d$, $\sigma_d(2d+1) = \sigma_{d-1}(2d)$.

Proof. We prove it by induction. Let H_d be the statement that $\sigma_d(2d+1) = \sigma_{d-1}(2d)$. H_d for $3 \leq d \leq 14$ is true from Table 2.1. This takes care of the base case of the induction. Assume that H_{d-1} is true, and let us prove that H_d is true. Let x be a square-maximal $(d, 2d+1)$-string.

(*i*) If x contains a singleton, remove it to form a new string y. Then y is a $(d-1, 2d)$-string, and $\sigma_d(2d+1) = s(x) \leq s(y) \leq \sigma_{d-1}(2d)$. Since $\sigma_d(2d+1) \geq \sigma_{d-1}(2d)$ by Proposition 2.6 (b), $\sigma_d(2d+1) = \sigma_{d-1}(2d)$.

(*ii*) If x contains no singletons, then x contains exactly $d-1$ pairs and 1 triple. By Lemma 2.4, we can move all the pairs to be adjacent without destroying the maximality of x. Then we can assume x has the form of $aaabbccdd\cdots$. Remove one pair to form string z. Then z is a $(d-1, 2d-1)$-string and $\sigma_d(2d+1)-1 = s(x)-1 = s(z) \leq \sigma_{d-1}(2d-1)$. Since $\sigma_d(2d+1)-1 \geq \sigma_{d-1}(2d-1)$ by Proposition 2.6 (c), $\sigma_d(2d+1) = \sigma_{d-1}(2d-1)+1$. Since $H_{d-1} : \sigma_{d-1}(2d-1) = \sigma_{d-2}(2d-2)$ holds, $\sigma_d(2d+1) = \sigma_{d-2}(2d-2)+1$. By Proposition 2.6 (c), $\sigma_{d-1}(2d) \geq \sigma_{d-2}(2d-2)+1$. Thus, $\sigma_{d-1}(2d) \geq \sigma_d(2d+1)$. Since $\sigma_{d-1}(2d) \leq \sigma_d(2d+1)$ according to Proposition 2.6 (b), hence $\sigma_d(2d+1) = \sigma_{d-1}(2d)$.

□

Let ***(d, 2d)-diagonal*** denote the main diagonal of the $(d, n-d)$-table as the entries on the main diagonal are $\sigma_d(2d)$. Similarly, ***(d, 2d + i)-diagonal*** where $i \geq 1$ denotes a diagonal above the main diagonal containing the entries of $\sigma_d(2d+i)$. For example, $(d, 2d+1)$-diagonal refers to the diagonal immediately above the main diagonal. Corollary 2.13 demonstrates the fact that the difference between any two consecutive entries on $(d, 2d+1)$-diagonal and $(d, 2d+2)$-diagonal of the $(d, n-d)$-table is also bounded by 2.

Corollary 2.13. *For any* $3 \leq d$, $\sigma_d(2d+1) - \sigma_{d-1}(2d-1) \leq 2$ *and* $\sigma_d(2d+2) - \sigma_{d-1}(2d) \leq 2$.

Proof. By Lemma 2.10, $\sigma_{d-1}(2d) - \sigma_{d-1}(2d-1) \leq 2$; and by Lemma 2.12, $\sigma_{d-1}(2d) = \sigma_d(2d+1)$. Therefore, $\sigma_d(2d+1) - \sigma_{d-1}(2d-1) \leq 2$. Similarly, $\sigma_d(2d+2) - \sigma_d(2d+$

1) ≤ 2 by Lemma 2.10, and $\sigma_d(2d+1) = \sigma_{d-1}(2d)$ by Lemma 2.12. Therefore, $\sigma_d(2d+2) - \sigma_{d-1}(2d) \leq 2$. □

2.5 $\sigma_d(n)$ Conjecture

It is widely believed that the number of distinct squares in a string is bounded by its length n. Inspired by the d-step approach, we conjecture that the upper bound for the number of distinct primitively rooted squares in a string is $n-d$, that is, $\sigma_d(n) \leq n-d$, for $2 \leq d \leq n-2$. It is observed that this conjectured upper bound holds for all known entries in the $(d, n-d)$-table [10], i.e. every entry in the table is smaller or equal to its column index. This section contains the reformulations of the conjectured upper bound for $\sigma_d(n)$ in Theorems 2.14 and 2.15. We also present conditions that lead to a slightly stronger upper bound in Theorems 2.16 and 2.17.

It can be observed in Table 2.1 that the known entries on the main diagonal satisfy $\sigma_d(2d) = d$. Theorem 2.14 shows that, indeed, this observation is equivalent with the conjectured bound. In essence, Theorem 2.14 shows that if the conjectured upper bound is violated somewhere in the $(d, n-d)$-table, then there must be a violation on the main diagonal of the same column as well. In fact, the violation propagates through the descending diagonals because of the strictly increase on the entries by Proposition 2.6 (c); that is, if d column is the first column that contains a counter-example to the conjectured upper bound, then so do the subsequent columns bigger than d. On the other hand, if $\sigma_d(2d) = d$ is true for some column d in the $(d, n-d)$-table, then all the entries on the same column are less than d thus complying with the conjectured upper bound, since $\sigma_d(2d)$ is the maximal value in the column. This fact can be used to confirm that $\sigma_d(n) \leq n-d$ holds up to certain column without even computing the actual values of the entries. Note that this theorem corresponds

to the equivalency between the d-step conjecture and the Hirsch conjecture showed by Klee and Walkup [24].

Theorem 2.14. *The conjectured upper bound $\sigma_d(n) \le n - d$ holding true for all $2 \le d \le n-2$, is equivalent with the statement: $\sigma_d(2d) \le d$ for every $d \ge 2$.*

Proof. $\sigma_d(n) \le n - d$ clearly implies that $\sigma_d(2d) \le d$ when $n = 2d$; that is, by Lemma 2.7, $\sigma_d(2d) = d$. To prove the other direction, we consider (i) $n \le 2d$: by Proposition 2.6 (d) we have $\sigma_d(n) = \sigma_{n-d}(2n - 2d) \le n - d$; (ii) $n > 2d$: by Proposition 2.6 (b) we have $\sigma_d(n) \le \sigma_{n-d}(2n - 2d) \le n - d$. □

Another observation in Table 2.1 is that the value on the main diagonal and the value of its right neighbour are identical. Theorem 2.15 shows that the inequality of the difference is bounded by 1, is equivalent with the conjectured upper bound; while the equality of the two gives rise to a slightly stronger upper bound given in Theorem 2.17.

Theorem 2.15. *The conjectured upper bound $\sigma_d(n) \le n - d$ holding true for all $2 \le d \le n-2$, is equivalent with the statement: $\sigma_d(2d+1) - \sigma_d(2d) \le 1$ for every $d \ge 2$.*

Proof. That the statement follows from the conjectured upper bound is clear. Let us thus prove the opposite direction. We shall prove by contradiction that $\sigma_d(2d) \le d$ for $d \ge 2$, as it is equivalent to the conjectured upper bound by Theorem 2.14. Let $d \ge 2$ be the smallest such that $\sigma_d(2d) > d$. Let x be a square-maximal $(d, 2d)$-string. If x does not have a singleton, then x consists of only pairs. By Lemma 2.5, all the pairs in x must be adjacent, and thus $\sigma_d(2d) = d$, a contradiction. Thus, x must have a singleton. Let y be x with the a singleton removed. Then y is a $(d-1, 2d-1)$-string and $s(y) \ge s(x)$ by Lemma 2.2. Thus, $\sigma_{d-1}(2d-1) \ge s(y) \ge s(x) = \sigma_d(2d)$. By

assumption, $\sigma_{d-1}(2d-1) \leq \sigma_{d-1}(2d-2)+1 \leq d-1+1 = d$. Thus, $d \geq \sigma_{d-1}(2d-1) = \sigma_d(2d) > d$, a contradiction. Therefore, $\sigma_d(2d) \leq d$ for every $d \geq 2$ holds and the conjectured upper bound follows. □

The portion of the $(d, n-d)$-table given in Table 2.1 shows that not only $\sigma_d(2d)$ is bounded by d, but also it is true for $\sigma_d(2d+1)$. Theorem 2.16 shows that this property implies a slightly stronger upper bound.

Theorem 2.16. *If $\sigma_d(2d+1) \leq d$ for every $d \geq 2$, then $\sigma_d(n) \leq n-d-1$ for $n > 2d$ and $\sigma_d(n) = n-d$ for $n \leq 2d$.*

Proof. By Lemma 2.7 and Proposition 2.6 (a), we have $d \leq \sigma_d(2d)$, and $\sigma_d(2d) \leq \sigma_d(2d+1)$, respectively. Since $\sigma_d(2d+1) \leq d$, then $\sigma_d(2d) = \sigma_d(2d+1) = d$. It implies that $\sigma_d(n) = n-d$ for $n \leq 2d$ by Proposition 2.6 (d). For $n > 2d$ we have, by Proposition 2.6(b), $\sigma_d(n) \leq \sigma_{n-d-1}(2n-2d-1) = n-d-1$. □

Theorem 2.17. *If $\sigma_d(2d) = \sigma_d(2d+1)$ for every $d \geq 2$, then $\sigma_d(n) \leq n-d-1$ for $n > 2d$ and $\sigma_d(n) = n-d$ for $n \leq 2d$.*

Proof. The results follow from Theorem 2.16 and the fact that $\sigma_d(2d) = \sigma_d(2d+1) = d$ for every $d \geq 2$. To show $\sigma_d(2d) = \sigma_d(2d+1) = d$ for every $d \geq 2$, let us argue by contradiction. Let d be the smallest such that $\sigma_d(2d) = \sigma_d(2d+1) > d$. Thus, $d-1 = \sigma_{d-1}(2d-2) = \sigma_{d-1}(2d-1)$. By Lemma 2.9, $\sigma_d(2d) - \sigma_{d-1}(2d-1) \leq 1$. It follows that $\sigma_d(2d) - (d-1) \leq 1$. i.e. $\sigma_d(2d) \leq d$, a contradiction. □

Chapter 3

Structure of Square-Maximal Strings

In Chapter 2 we conjectured that $n - d$ is an upper bound for the number of distinct primitively rooted squares in a string, where n is the length and d is the number of symbols of the string. We also formulated several properties equivalent with the conjecture. In this chapter, we explore in more depth the structure of square-maximal strings, in particular the strings that are on the main diagonal of the $(d, n-d)$-table, that is, the strings with $n = 2d$. The reason is that not only their compliance with the conjectured upper bound was proven to be equivalent to the conjecture for all $n - 2 \geq d \geq 2$ in Theorem 2.14; but more importantly, they are more structured and combinatorially less complex to analyze in comparison with other square-maximal strings.

Section 3.1 gives the unique structure of square-maximal strings on the main diagonal if $\sigma_d(2d)$ values are identical to their right neighbours in the $(d, n-d)$-table. Section 3.2 shows several necessary conditions for the structure of strings on the main diagonal if they do not comply with the conjectured upper bound. The purpose of this

exercise is to show that the strings either comply with the conjecture, or otherwise they have a very special structure and thus are rather less possible to achieve. Of course, if we could prove that such structure is not possible, then we would have proved our conjecture on the maximum number of distinct squares. At this time, we hope that this points to a possible direction on how to tackle the problem. By analyzing the structure of such strings, we can estimate the number of singletons in them; from this estimation, another property equivalent with the conjecture is given in Section 3.3. The results of this chapter were reported in [11].

3.1 Square-Maximal Strings with $\sigma_d(2d) = \sigma_d(2d+1)$

In Theorem 2.17 we showed a slightly stronger upper bound with the condition that all the values on the main diagonal of the $(d, n-d)$-table are identical to the values of their right neighbour. Lemma 3.1 shows the unique structure of such square-maximal strings on the main diagonal.

Lemma 3.1. *If $\sigma_d(2d) = \sigma_d(2d+1)$ for every $d \geq 2$, then for any $d \geq 2$ and any square-maximal $(d, 2d)$-string x, x consists of d adjacent pairs, i.e. equals to $aabbcc\ldots$ up to relabeling of the alphabet.*

Proof. If x contains only pairs, by Lemma 2.5 all these pairs have to be adjacent. If x does not consist only of pairs, then it would have to have a singleton. Let y be a string obtained from x by removing a singleton, then y is a $(d-1, 2d-1)$-string and $s(y) \geq s(x)$ by Lemma 2.2. Since $\sigma_{d-1}(2d-1) = \sigma_{d-1}(2d-2)$ and $\sigma_{d-1}(2d-2) = d-1$ by Theorem 2.17, $d-1 = \sigma_{d-1}(2d-1) \geq s(y) \geq s(x) = \sigma_d(2d) = d$ which is contradiction. Therefore, x contains only d adjacent pairs and is up to relabeling, unique and equals to $x = aabbcc\ldots$. □

3.2 Square-Maximal Strings with $\sigma_d(2d) > d$

In this section we investigate the square-maximal strings that are not in compliance with the conjectured upper bound, if there are any at all. The main goal of this investigation is to either find a counterexample on the main diagonal, if there is one; or to show that there are no counterexamples on the main diagonal, and thus prove the conjectured upper bound for all strings. We show that a square-maximal string from the main diagonal either complies with the conjectured upper bound or has to have many singletons based on the facts that such string (a) cannot contain pairs, see Lemma 3.3, and (b) if it contains a triple, it is must be a very special triple, implying the existence of a symbol occurring at least 6 times, see Lemma 3.6. We hope that it might be possible to show that counterexamples on the main diagonal do not exist by showing that their structure would be impossible.

3.2.1 Auxiliary Lemma

Auxiliary Lemma 3.2 is used to estimate the number of squares that span from one part of a string to the other part, and relies on the result of Fraenkel and Simpson [14].

Lemma 3.2. *Consider non-empty strings w, u, and v. The number of distinct primitively rooted squares of the string wuv that start in w and end in v is at most $|w|+|v|$ where $|w|$ and $|v|$ denotes the length of w and the length of v, respectively.*

Proof. We discuss two cases:

(i) If $|w| \leq |v|$, we count the rightmost occurrences of squares. By Fraenkel and Simpson [14], there are at most two such squares starting at the same position. Thus, there are at most $2|w|$ squares that start in w, and $2|w| \leq |w| + |v|$.

(ii) If $|w| > |v|$, let $\overline{x}$ denote the reversal of the string x, and $\overline{x} = \overline{wuv} = \overline{v}\ \overline{u}\ \overline{w}$. By the previous argument, there are at most $2|\overline{v}|$ rightmost squares of the string starting in $\overline{v}$. It follows that there are at most $2|v|$ squares of wuv that end in v and $2|v| < |w| + |v|$.

□

3.2.2 Pair

Our conjectured upper bound holds for all the values in the $(d, n-d)$-table we have computed so far. Lemma 3.3 shows that the square-maximal strings in first unknown position on the main diagonal either continue complying with the conjectured upper bound or cannot contain a pair. Let us remark that since the main diagonal entry is the largest entry along a column in the $(d, n-d)$-table by Proposition 2.6 (b) and (d), for any column with $\sigma_{d'}(2d') \leq d'$, the entries in the same column are also bounded by the column index d'. This fact is used in the proofs of the lemmas in this chapter when deducing contradictions to the assumption.

Lemma 3.3. *Let $\sigma_{d'}(2d') \leq d'$ for every $d' < d$. Let x be a square-maximal $(d, 2d)$-string. Then either $s(x) = \sigma_d(2d) = d$ or x does not contain a pair.*

Proof. We shall prove it by contradiction. Let us assume that $s(x) = \sigma_d(2d) > d$ and x contains a pair of C's at positions i_0 and i_1, so $x[i_0] = x[i_1] = C$. If the pair occurs in at most one square, then we can replace the first C with a new symbol $\hat{C} \notin \mathcal{A}(x)$. Let y be x with $x[i_0]$ replaced by $\hat{C}$. Then y is a $(d+1, 2d)$-string. Since $2d - (d+1) = d - 1 < d$ and $\sigma_{d-1}(2d-2) \leq d-1$, $\sigma_{d+1}(2d) \leq d-1$. Thus, $d - 1 \geq \sigma_{d+1}(2d) \geq s(y) \geq s(x) - 1 = \sigma_d(2d) - 1$, i.e. $d \geq \sigma_d(2d)$, a contradiction.

Therefore, the pair must occur in at least two squares, in fact in a non-trivial run $uvCwuvCwu$ by Lemma 2.3, where $|u| \geq 1$. Let us form a new string z by

removing all the symbols between the C's so that the run becomes $uvCCwu$. By doing this, we may have destroyed $|u| + 1$ squares, i.e. square $uvCwuvCw$ and its $|u|$ rotations. For the removal of wuv, the type of every square in w is preserved, as z has w as a substring. The same is true for u, v, wu, and uv. Thus, the only squares of wuv we may have destroyed are the squares that start in w and end in v. By Lemma 3.2, there are at most $|w| + |v|$ such squares. So, altogether, we may have destroyed at most $|w| + |u| + |v| + 1$ squares, but we created a new one CC. Thus, $s(z) \geq s(x) - (|w| + |u| + |v|)$. Clearly, $\mathcal{A}(z) = \mathcal{A}(x)$ as all the symbols occurring in wuv are preserved, so z is a $(d, 2d-k)$-string where $k = |w| + |u| + |v|$. By the assumption of this lemma as $2d - k - d = d - k < d$, we have $d - k \geq \sigma_d(2d-k) \geq s(z) \geq s(x) - k = \sigma_d(2d) - k$. Thus, $d \geq \sigma_d(2d)$, a contradiction. □

3.2.3 Triple

As discussed in Section 3.2.2, Lemmas 3.4 and 3.5 use the same scenario investigating the square-maximal strings in the first unknown position on the main diagonal and showing that they either comply with the conjectured upper bound or may contain only very specific triples.

Lemma 3.4. *Let $\sigma_{d'}(2d') \leq d'$ for every $d' < d$. Let x be a square-maximal $(d, 2d)$-string. Then either $s(x) = \sigma_d(2d) = d$, or if x contains a triple, then the triple has to occur in two distinct runs.*

Proof. Let us assume that $s(x) = \sigma_d(2d) > d$. Let $x[i_0] = x[i_1] = x[i_2] = C$ be a triple in x. We first show that all three symbols occur in some run. Assume that $x[i_0]$ does not occur in any run. If $x[i_0]$ does not occur in any run, then $x[i_0]$ does not occur in any square. Let y be x with $x[i_0]$ replaced by $\hat{C}$, and $\hat{C} \notin \mathcal{A}(x)$. Then y is a $(d+1, 2d)$-string and $\sigma_{d+1}(2d) \geq s(y) = s(x) = \sigma_d(2d)$. Since $2d - (d+1) < d$, we get

$2d-(d+1) \geq \sigma_{d+1}(2d) \geq \sigma_d(2d)$, i.e. $d-1 \geq \sigma_d(2d)$, a contradiction. Similarly, $x[i_2]$ must occur in some run. If $x[i_1]$ does not occur in any run, then none of the elements of the triple occur in any run. As we proved that $x[i_0]$ and $x[i_2]$ must occur in some runs, this would lead to a contradiction.

We have to show that three symbols cannot occur in the same run. Assume they do occur in the same run $uvCwuvCwuvCwu$. We can remove wuv between the first and second C and proceed with the same proof as in Lemma 3.3 to derive a contradiction. □

Lemma 3.5. *Let $\sigma_{d'}(2d') \leq d'$ for every $d' < d$. Let x be a square-maximal $(d, 2d)$-string. Then either $s(x) = \sigma_d(2d) = d$, or if x has a triple $x[i_0] = x[i_1] = x[i_2] = C$ occurring in two distinct runs $u_1v_1x[i_0]w_1u_1v_1x[i_1]w_1u_1 = u_1v_1Cw_1u_1v_1Cw_1u_1$ and $u_2v_2x[i_1]w_2u_2v_2x[i_2]w_2u_2 = u_2v_2Cw_2u_2v_2Cw_2u_2$, then $|u_1| \geq 1$ and $|u_2| \geq 1$, and either u_2v_2 is not a suffix of u_1v_1 or w_1u_1 is not a prefix of w_2u_2.*

Proof. Let us assume that $s(x) = \sigma_d(2d) > d$. If $|u_1| = 0$, then $x[i_0]$ occurs in a trivial run, i.e. a single square $v_1Cw_1v_1Cw_1$. Let y be x with $x[i_0]$ replaced by $\hat{C}$, and $\hat{C} \notin \mathcal{A}(x)$. By this replacement, we have destroyed one square. Then y is a $(d+1, 2d)$-string and $\sigma_{d+1}(2d) \geq s(y) = s(x)-1 = \sigma_d(2d)-1$. Since $2d-(d+1) < d$, we get $2d-(d+1) \geq \sigma_{d+1}(2d) \geq \sigma_d(2d)-1$, i.e. $d \geq \sigma_d(2d)$, a contradiction. It follows that $|u_1| \geq 1$. To show $|u_2| \geq 1$, the proof is the same. Thus, $|u_1| \geq 1$ and $|u_2| \geq 1$.

Let us assume that both u_2v_2 is a suffix of u_1v_1 and w_1u_1 a prefix of w_2u_2. Let us form a new string from x by removing $w_1u_1v_1$ between $x[i_0]$ and $x[i_1]$ and removing $w_2u_2v_2$ between $x[i_1]$ and $x[i_2]$ so that the form of the two runs becomes $u_1v_1CCCw_2u_2$. How many squares have we destroyed? We might have destroyed $|u_1|+1$ squares of $u_1v_1Cw_1u_1v_1Cw_1u_1$ ($u_1v_1Cw_1u_1v_1Cw_1$ and its $|u_1|$ rotations) and $|u_2|+1$ squares of

$u_2v_2Cw_2u_2v_2Cw_2u_2$ ($u_2v_2Cw_2u_2v_2Cw_2$ and its $|u_2|$ rotations). For $w_1u_1v_1$, u_1v_1 has been preserved, w_1u_1 is also preserved since it is a prefix of w_2u_2, so the only squares we might have destroyed are the ones that start in w_1 and end in v_1. By Lemma 3.2 there are at most $|w_1| + |v_1|$ of them. Similarly for $w_2u_2v_2$, w_2u_2 is preserved and u_2v_2 is also preserved as it is a suffix of u_1v_1, the number of squares that span from w_2 to v_2 is at most $|w_2| + |v_2|$. Thus, in total we might have destroyed at most $|w_1| + |u_1| + |v_1| + |w_2| + |u_2| + |v_2| + 2 = k + 2$ squares, where $k = |w_1| + |u_1| + |v_1| + |w_2| + |u_2| + |v_2|$, and we gained one square from CCC. Replace the first C by a new symbol $\hat{C} \notin \mathcal{A}(x)$ to form a string z. This will not destroy any square as we still have square CC, but the number of distinct symbols is increased by one. Then z is a $(d+1, 2d-k)$-string, and $\sigma_{d+1}(2d-k) \geq s(z) \geq s(x) - (k+2) + 1 = \sigma_d(2d) - k - 1$. Since $2d-k-(d+1) = d-k-1 < d$, we have $d-k-1 \geq \sigma_{d+1}(2d-k) \geq \sigma_d(2d)-k-1$, i.e. $d \geq \sigma_d(2d)$, a contradiction. It follows that either u_2v_2 is not a suffix of u_1v_1, in which case u_1v_1 is a suffix of u_2v_2; or w_1u_1 is not a prefix of w_2u_2, in which case w_2u_2 is a prefix of w_1u_1. □

3.2.4 Singletons Estimation

Lemma 3.6 utilizes the previous lemmas and shows that any square-maximal string in the first unknown position on the main diagonal either complies with the conjectured upper bound, or if it contains a triple, it must be a very specific one giving rise to a symbol that must occur at least 6 times. Thus, each triple occurring in the string must be balanced by an existence of a unique set of 5 occurrences of a certain symbol. Though the symbol may not be unique to a particular triple, the set of occurrences are mutually disjoint. Thus, every triple with its assigned set of 5 occurrences is balanced by an existence of at least 4 singletons unique to the triple and its assigned set.

Lemma 3.6. *Let $\sigma_{d'}(2d') \le d'$ for every $d' < d$. Let x be a square-maximal $(d, 2d)$-string. Then either $s(x) = \sigma_d(2d) = d$ or x has at least $\lceil \frac{2d}{3} \rceil$ singletons.*

Proof. Let us assume that $s(x) = \sigma_d(2d) > d$. From Lemma 3.3 it follows that x does not have any pair. From Lemmas 3.4 and 3.5, any triple $x[i_0] = x[i_1] = x[i_2] = C$ of x must satisfy

1. $x[i_0]$ and $x[i_1]$ occur in a run $r_1 = u_1v_1Cw_1u_1v_1Cw_1u_1$, where $|u_1| \ge 1$,
2. $x[i_1]$ and $x[i_2]$ occur in a run $r_2 = u_2v_2Cw_2u_2v_2Cw_2u_2$, where $|u_2| \ge 1$, and where $i_1 - i_0 \ne i_2 - i_1$ as otherwise the two runs would merge into a single one,
3. either u_1v_1 is a proper suffix of u_2v_2, or w_2u_2 is a proper prefix of w_1u_1.

Let us discuss the case when u_1v_1 is a proper suffix of u_2v_2; the case of w_2u_2 being a proper prefix of w_1u_1 is the same just argued from the opposite direction. Let the run $r_1 = u_1v_1Cw_1u_1v_1Cw_1u_1$ start at position t of x. Suppose $x[t] = a$. If there is no other occurrence of a in u_1v_1 except the first position, then we can replace all the occurrences of a in $x[1..i_0 - 1]$ with a new symbol, forming a string y, while destroying a single square $u_1v_1Cw_1u_1v_1Cw_1$ of x. Thus, y is a $(d + 1, 2d)$-string, $2d - d - 1 \ge \sigma_{d+1}(2d) \ge s(y) = s(x) - 1 = \sigma_d(2d) - 1$, so $d \ge \sigma_d(2d)$, a contradiction. Thus, a occurs at least twice in u_1v_1. Since u_1v_1 is a proper suffix of u_2v_2, a occurs at least 4 more times – twice in each occurrence of u_2v_2. Thus, $x[t]$ occurs in x at least 6 times, the last occurrence before the last C. We assign to the triple the sequence of positions of the first five occurrences of a after the position t and denote it by $As(C) = \langle j_0, j_1, j_2, j_3, j_4 \rangle$, where $t < j_0 < j_1 < j_2 < j_3 < j_4 < i_2$ and $j_0 < i_0$ and t is the start of the run r_1 and $x[t] = x[j_0] = x[j_1] = x[j_2] = x[j_3] = x[j_4]$. For the case that w_2u_2 is a proper prefix of w_1u_1, the assignment is from the opposite side of r_2. Then $As(C) = \langle j_0, j_1, j_2, j_3, j_4 \rangle$, where $i_0 < j_4 < j_3 < j_2 < j_1 < j_0 < t$ and $i_2 < j_0$, t

is the end of the run r_2 and $x[j_4] = x[j_3] = x[j_2] = x[j_1] = x[j_0] = x[t]$. Lemma 3.9 shows that such assignments are mutually disjoint, i.e. if C's and D's are different triples, then $As(C) \cap As(D) = \emptyset$.

Now we can estimate the number of singletons in x. Let m_0 be the number of triples in x. Let m_1 be the number of multiply occurring symbols that are not assigned to triples – since there are no pairs, it follows that such symbols occur at least 4 times. Let m_2 be the number of singletons in x. The following two inequalities must hold:

$$2d \geq (3+5)m_0 + 4m_1 + m_2 \tag{3.1}$$

$$d \leq 2m_0 + m_1 + m_2 \tag{3.2}$$

In 3.1 we underestimated the length of x as there may be symbols that occur more than 4 times, and in 3.2 we overestimated the number of distinct symbols d because while the set of five occurrences of a symbol assigned to the triples are disjoint, the symbol may not be unique to a triple, i.e. two triples are assigned the same symbol. Solving the inequalities, we get $2d \geq 8m_0 + 4m_1 + m_2 = 4(2m_0 + m_1) + m_2 \geq 4(d - m_2) + m_2$, which gives $3m_2 \geq 2d$. Thus, $m_2 \geq \lceil \frac{2d}{3} \rceil$. □

In Lemma 3.5 it is shown that a triple of C's can exist in x only if it occurs in two distinct non-trivial runs $u_1v_1Cw_1u_1v_1Cw_1u_1$ and $u_2v_2Cw_2u_2v_2Cw_2u_2$. We refer to u_1v_1 and w_2u_2 as the ***appendices***, and we say that u_1v_1 is a ***short appendix*** if u_1v_1 is a proper suffix of u_2v_2, similarly we say that w_2u_2 is a short appendix if it is a proper prefix of w_1u_1. As we defined in 3.6, short appendix determines the assignment to a triple $As(C)$; that is, the five occurrences of the assignment counts from the short appendix side of the runs. Lemma 3.5 also stipulates that at least one of the appendices must be short.

Before we prove that the two assignments of the triples are mutually disjoint, let us first show that the square-maximal strings cannot contain parallel k-tuples.

Definition 3.7. *A k-tuple of C's occurring at positions $\{i_1, \cdots, i_k\}$ and a k-tuple of D's occurring at positions $\{j_1, \cdots, j_k\}$ are **parallel** if $i_1 < j_1 < i_2 < j_2 < \cdots < i_k < j_k$, and $j_b - i_a > 1$ for any $1 \leq a, b \leq k$ and $j_b > i_a$.*

Definition 3.7 ensures that C's and D's are interleaved one by one, and the strings in between any C and D are non-empty.

Lemma 3.8. *Let x be a square-maximal (d, n)-string. Then x cannot contain two parallel k-tuples for any $k \geq 2$.*

Proof. Let us assume that x contains two parallel k-tuples of C's and D's. Let us move all the D's to the end of x forming a new string y. Similarly to the proof in Lemma 2.4, the squares that are destroyed by the replacement of D's will be replaced by the same number of new squares. For illustration: $uCvDuCvD$ becomes $uCvuCv$, where u and v are non-empty strings. Note that $uCvuCv$ is a new square type because it did not have other occurrences in x since any square that contained C's would have contained D's as they were interleaved. In addition, moving the D's to the end creates a new square DD. Thus, $s(y) > s(x)$, a contradiction with the maximality of x. □

Lemma 3.9. *Let $\sigma_{d'}(2d') \leq d'$ for every $d' < d$. Let x be a square-maximal $(d, 2d)$-string. Then either $s(x) = \sigma_d(2d) = d$ or if x contains triples C's and D's, then $As(C) \cap As(D) = \emptyset$.*

Proof. Let $As(C) = \langle j_0, j_1, j_2, j_3, j_4 \rangle$ and $As(D) = \langle k_0, k_1, k_2, k_3, k_4 \rangle$. If $x[j_0] \neq x[k_0]$, then $As(C) \cap As(D) = \emptyset$. Below, we discuss the case when $x[j_0] = x[k_0] = a$.

Without loss of generality, let us assume that the first C precedes the first D. We must discuss all the possible configurations of the two triples. For better readability, we will denote the first occurrence of C by C_1, and the second occurrence

of C by C_2, etc. Similarly for D's. Then C's occur in two non-trivial runs $r_1 = u_1v_1C_1w_1u_1v_1C_2w_1u_1$ and $r_2 = u_2v_2C_2w_2u_2v_2C_3w_2u_2$, while the D's occur in two non-trivial runs $r_3 = u_3v_3D_1w_3u_3v_3D_2w_3u_3$ and $r_4 = u_4v_4D_2w_4u_4v_4D_3w_4u_4$.

1. The two triples do not interleave (schematically $C_1\ C_2\ C_3\ D_1\ D_2\ D_3$).

   (a) First we consider the case when the appendix determining $As(C)$ and the appendix determining $As(D)$ are on the opposite sides. Thus, the short appendix determining $As(C)$ is on the left and the short appendix determining $As(D)$ is on the right. Then we are guaranteed the following pattern of occurrences of a in x (for C's, a's are shown in bold; for D's, a's are shown underscored): $\boldsymbol{a}\ \boldsymbol{a}\ C_1\ \boldsymbol{a}\ \boldsymbol{a}\ C_2\ \boldsymbol{a}\ \boldsymbol{a}\ C_3\ D_1\ \underline{a}\ \underline{a}\ D_2\ \underline{a}\ \underline{a}\ D_3\ \underline{a}\ \underline{a}$, so $x[j_4]$ occurs before C_3, while the $x[k_4]$ occurs after D_1. Therefore, $j_4 < k_4$ and $As(C) \cap As(D) = \emptyset$.

   (b) Next we consider the case when the appendix determining $As(C)$ and the appendix determining $As(D)$ are facing each other. Thus, for the C's we are using the right appendix, and for the D's we are using the left appendix. Then the pattern of occurrences of a in x should be: $C_1\ \boldsymbol{a}\ \boldsymbol{a}\ C_2\ \boldsymbol{a}\ \boldsymbol{a}\ C_3\ \underline{\boldsymbol{a}}\ \underline{\boldsymbol{a}}\ D_1\ \underline{a}\ \underline{a}\ D_2\ \underline{a}\ \underline{a}\ D_3$. Note that it is possible that the two a's between C_3 and D_1 are the same. However, since we do not take the first occurrence of a for the assignments, $As(C) \cap As(D) = \emptyset$.

   (c) Here we consider the case when the appendix determining $As(C)$ and the appendix determining $As(D)$ are on the same side. Without loss of generality, we can assume that both appendices used are on the left. Then the pattern of occurrences of a in x is: $\boldsymbol{a}\ \boldsymbol{a}\ C_1\ \boldsymbol{a}\ \boldsymbol{a}\ C_2\ \boldsymbol{a}\ \boldsymbol{a}\ C_3\ \underline{a}\ \underline{a}\ D_1\ \underline{a}\ \underline{a}\ D_2\ \underline{a}\ \underline{a}\ D_3$. Why cannot the first two $\underline{a}$'s be the same as the last two $\boldsymbol{a}$'s? If it were the case, then C would be in the appendix for the D's, i.e. a part of the

run r_3 and hence repeat later. So, again $As(C) \cap As(D) = \emptyset$.

2. Case $C_1\ D_1\ D_2\ C_2\ C_3\ D_3$, $C_1\ D_1\ D_2\ C_2\ D_3\ C_3$, and $C_1\ D_1\ D_2\ D_3\ C_2\ C_3$ are not possible.

   For the first two cases, if either D_1 or D_2 occurred in u_1v_1, then there would be a D preceding C_1. Thus, both D_1 and D_2 occur in w_1, but then D occurs at least 4 times since there is another w_1 after C_2, a contradiction. For the third case, the proof is the same except all the D's must occur in w_1 but there is no more D after C_2, a contradiction.

3. Case $C_1\ D_1\ C_2\ D_2\ D_3\ C_3$ is not possible.

   As in the previous cases, but arguing from the opposite side, both D_2 and D_3 must occur in v_2 since if either of them occurs in w_2u_2 then there would be a D after C_3. Hence D must occur at least 4 times, a contradiction.

4. Case $C_1\ D_1\ C_2\ D_2\ C_3\ D_3$ is not possible.

   By Lemma 3.8 C's and D's cannot be parallel.

5. Case $C_1\ D_1\ C_2\ C_3\ D_2\ D_3$ is not possible.

   This case is clearly impossible because D_1 occurs in w_1 since there is no D preceding C_1. But there is no D occurring between C_2 and C_3, a contradiction.

6. Case $C_1\ C_2\ D_1\ D_2\ C_3\ D_3$ and $C_1\ C_2\ D_1\ D_2\ D_3\ C_3$ are not possible.

   For $C_1\ C_2\ D_1\ D_2\ C_3\ D_3$, both D_1, D_2 must occur in w_2 as there is no D preceding C_2. But then D occurs at least 4 times since there is another w_2 follows C_3, a contradiction. Proof for $C_1\ C_2\ D_1\ D_2\ D_3\ C_3$ is the same except we would need D occurring at least 6 times which is impossible.

7. Case $C_1\ C_2\ D_1\ C_3\ D_2\ D_3$.

   We denote by $w_2^{(1)}$ the first occurrence of w_2 in x, by $w_2^{(2)}$ the second occurrence

of w_2 in x, etc.

If D_1 occurred in $(u_2v_2)^{(2)}$, there would be a D preceding C_2. Hence D_1 must occur in $w_2^{(1)}$ and hence D_2 occurs in $w_2^{(2)}$. Since the distance between C_2 and C_3 is the period of r_2, and the distance between D_1 and D_2 is the period of r_3, and the distances are equal, it follows that $r_2 = r_3 = u_2v_2C_2w_2'D_1w_2''u_2v_2C_3w_2'D_2w_2''u_2$. Note that $u_3 = u_2$ and $v_3 = v_2Cw_2'$ and $w_3 = w_2''$.

Schematically:

$$\begin{array}{ll} r_1: & u_1v_1C_1w_1u_1v_1C_2w_1u_1 \\ r_2 = r_3: & \qquad u_2v_2C_2w_2'D_1w_2''u_2v_2C_3w_2'D_2w_2''u_2 \\ r_4: & \qquad\qquad\qquad u_4v_4D_2w_4u_4v_4D_3w_4u_4 \end{array}$$

Consider the two runs r_1 and r_2. Since D_1 cannot occur in $(w_1u_1)^{(2)}$, it follows that the w_1u_1 is a prefix of w_2' and hence of $w_2'D_1w_2''u_2$, and so the appendix $w_2'D_2w_2''u_2$ is not short and by Lemma 3.5, u_1v_1 must be a short appendix and is used to determine $As(C)$.

Consider the two runs r_3 and r_4. Since C_3 cannot occur in $(u_4v_4)^{(1)}$, u_4v_4 is a suffix of w_2' and hence of $u_2v_2C_3w_2'$, and so the appendix $u_2v_2C_2w_2'$ is not short. By Lemma 3.5, w_4u_4 must be a short appendix and is used to determine $As(D)$.

(a) Let a occur twice in u_1 and twice in u_4 (*the dots indicate the occurrences*).

$$\begin{array}{ll} r_1: & \ddot{u}_1v_1C_1w_1\ddot{u}_1v_1C_2w_1\ddot{u}_1 \\ r_2 = r_3: & \qquad u_2v_2C_2w_2'D_1w_2''u_2v_2C_3w_2'D_2w_2''u_2 \\ r_4: & \qquad\qquad\qquad \underset{\cdot\cdot}{u_4}v_4D_2w_4\underset{\cdot\cdot}{u_4}v_4D_3w_4\underset{\cdot\cdot}{u_4} \end{array}$$

Then a occurs twice in each occurrence of u_1 and hence $x[j_4]$ occurs before D_1 since w_1u_1 is a prefix of w_2'. Similarly, a occurs twice in each occurrence of u_4 and hence $x[k_4]$ occurs after D_1 since u_4v_4 is a suffix of w_2'. Thus, $As(C) \cap As(D) = \emptyset$.

(b) Let a occur only once in u_1 and twice in u_4.

$$
\begin{array}{ll}
r_1: & \dot{u}_1\dot{v}_1C_1w_1\dot{u}_1\dot{v}_1C_2w_1\dot{u}_1 \\
r_2 = r_3: & \quad u_2v_2C_2w_2{}'D_1w_2{}''\dot{u}_2\dot{v}_2C_3w_2{}'D_2w_2{}''u_2 \\
r_4: & \qquad\qquad u_4v_4D_2w_4u_4v_4D_3w_4u_4
\end{array}
$$

Then a must occur in v_1. Since u_1v_1 is a suffix of u_2v_2 and since w_1u_1 is a prefix of w_2', we have 7 occurrences of a from the left and 6 occurrences of a from the right, so again $As(C) \cap As(D) = \emptyset$.

(c) Let a occur twice in u_1 and only once in u_4.

This is symmetric to the previous case, we have 6 occurrences of a from the left, and 7 occurrences of a from the right. Thus, $As(C) \cap As(D) = \emptyset$.

(d) Let a occur in u_1 only once and in u_4 also only once.

$$
\begin{array}{ll}
r_1: & \dot{u}_1\dot{v}_1C_1w_1\dot{u}_1\dot{v}_1C_2w_1\dot{u}_1 \\
r_2 = r_3: & \quad u_2v_2C_2w_2{}'D_1w_2{}''\dot{u}_2\dot{v}_2C_3\dot{w}_2{}'D_2w_2{}''u_2 \\
r_4: & \qquad\qquad u_4v_4D_2w_4u_4v_4D_3w_4u_4
\end{array}
$$

From the left there are 8 occurrences of a: a must occur in v_1 and since u_1v_1 is a suffix of u_2v_2, it must occur twice in $(u_2v_2)^{(2)}$, and since u_1 is a substring of w_2', a must occur in all the occurrences of w_2'. Similarly, there are 8 occurrences of a from the right. Even though it is possible for some of the occurrences from the left and from the right are the same, the first 5 occurrences from the left and 5 occurrences from the right are disjoint, and so $As(C) \cap As(D) = \emptyset$.

□

3.3 Additional Combinatorial Property Equivalent with the Conjectured Upper Bound

We gave a lower bound for the number of singletons in the first square-maximal string on the main diagonal that violates the conjectured upper bound. Theorem 3.10 stresses the fact that such violation implies the existence of a counterexample higher up in the $(d, n-d)$-table. Similarly as Theorems 2.14 and 2.15, this is yet another combinatorial property equivalent with the conjectured upper bound.

Theorem 3.10. *The conjectured upper bound $\sigma_d(n) \leq n - d$ holding true for all $2 \leq d \leq n-2$ is equivalent with the statement: $\sigma_d(4d) \leq 3d$ for every $d \geq 2$.*

Proof. Setting $n = 4d$ in $\sigma_d(n) \leq n - d$ directly yields $\sigma_d(4d) \leq 3d$. Then, assume that there is a counterexample to the inequality $\sigma_d(n) \leq n - d$. By Theorem 2.14, it follows that there is a counterexample x on the main diagonal, i.e. a square-maximal $(d, 2d)$-string x with $s(x) = \sigma_d(2d) > d$. Let us consider the smallest d in the table where the counterexample occurs. By Lemma 3.6, x has at least $\lceil \frac{2d}{3} \rceil$ singletons. If we remove these $\lceil \frac{2d}{3} \rceil$ singletons from x, we get a (d', n')-string y where $d' = d - \lceil \frac{2d}{3} \rceil$ and $n' = 2d - \lceil \frac{2d}{3} \rceil$. Then $\sigma_{d'}(n') \geq s(y) \geq s(x) = \sigma_d(2d) > d$. Moreover, $4d' = 4(d - \lceil \frac{2d}{3} \rceil) = 4d - 4 \cdot \lceil \frac{2d}{3} \rceil = 4d - 2d - \lceil \frac{2d}{3} \rceil = 2d - \lceil \frac{2d}{3} \rceil = n'$. So we have $\sigma_{d'}(4d') > d$. Since $3d' = 4d' - d' = n' - d' = (2d - \lceil \frac{2d}{3} \rceil) - (d - \lceil \frac{2d}{3} \rceil) = d$, it follows that $\sigma_{d'}(4d') > 3d'$. Therefore, we have a counterexample that is a $(d', 4d')$-string. □

Chapter 4

Computational Approach

To compute $\sigma_d(n)$ by brute force, we would have to generate all possible (d, n)-strings, compute the number of distinct primitively rooted squares for each of them, and find out the maximal value among them. This was how we populated the fragment of the $(d, n-d)$-table shown in Table 2.1. One could employ some computationally efficient techniques like generating the strings only in an increasing lexicographic order utilizing the fact that a relabelling of the alphabet does not change the number of distinct primitively rooted squares, nevertheless the number of strings to be generated is $O(d^n)$. Clearly, this approach becomes intractable for larger d and n for two reasons – one is the strings and the numbers to handle becoming too big, and the other is the computation time becoming too long; with the latter being the most difficult one to deal with.

To overcome this limitation, and to expand the $(d, n-d)$-table as much as we can in order to explore the behaviour of $\sigma_d(n)$ in further depth, we designed a computational framework to compute the maximum number of distinct primitively rooted squares in strings. The main strategy of this framework is based on reducing the set of strings needed to be computer generated as much as possible. In other words, instead

of generating all possible (d, n)-strings as in brute force approach, we would like to generate only those strings that are likely to produce $\sigma_d(n)$. Thus, we can limit our search to strings that are guaranteed not to give a worse number than a lower bound established by some other means: we call them candidate strings. We will show that this reduction is in fact very significant and it allows us to virtually double the range of computationally tractable instances.

This chapter outlines the computational framework in details, and specifies conditions that the candidate strings must satisfy. Partial contents of this chapter were reported in [12], and some work was a collaboration with Liu who reported the results in her Ph.D. thesis [29].

4.1 Dense s-Covered Strings

In this section, we mainly focus on the theoretical aspects of the computational framework. Suppose that a lower bound $\sigma_d^-(n)$ for the maximum number of distinct primitively rooted squares for (d, n)-strings is known. The value of $\sigma_d(n)$ must either be larger or equal to $\sigma_d^-(n)$. If there is no (d, n)-string x with $s(x) > \sigma_d^-(n)$, then $\sigma_d(n) = \sigma_d^-(n)$. Therefore, we have to focus on strings x such that $s(x) > \sigma_d^-(n)$. If there is no such string, we established that $\sigma_d(n) = \sigma_d^-(n)$; if there is one, we can set $\sigma_d^-(n)$ to this new improved value and repeat the whole process. If we can show, and we can, that $|\sigma_d^-(n) - \sigma_d(n)| \leq 2$, in the worst case after two iterations we have the real value of $\sigma_d(n)$.

We show that in order to obtain a string x with $s(x) > \sigma_d^-(n)$, x has to satisfy certain combinatorial properties. These properties are based on the notions of *s-cover*, *core vector*, and *density* defined in the section. Specifically, the square-maximal strings with the number of distinct primitively rooted squares potentially exceeding $\sigma_d^-(n)$

must be *dense s-covered* strings.

4.1.1 The Notion of s-Cover

Definition 4.1. *An* ***s-cover*** *of a string* $x = x[1 \,..\, n]$ *is a sequence of primitively rooted squares* $\{S_i = (s_i, e_i, p_i) \mid 1 \leq i \leq m\}$ *so that:*

(1) *for any* $1 \leq i < m$, $s_i < s_{i+1} \leq e_i + 1$ *and* $e_i < e_{i+1}$*;*

(2) $\bigcup_{1 \leq i \leq m} S_i = x$*;*

(3) *for any occurrence of square* S *in* x*, there is an* i*, where* $1 \leq i \leq m$ *such that* S *is a substring of* S_i*, denoted by* $S \subseteq S_i$*.*

Let us analyze each property of the s-cover. We say a square is an *s-cover square* if it is in the s-cover of the string. Definition 4.1 (1) means the two consecutive s-cover squares are either adjacent or overlapping. Definition 4.1 (2) enforces that the union of all s-cover squares forms the entire string x, that is, every position in x is covered. Definition 4.1 (3) constraints that every occurrence of a square in x is completely contained within one of the s-cover squares; in other words, there is no intermediate square between the two s-cover squares; that is, there is no square starting in the non-overlapping portion, i.e. the portion that does not overlap with other s-cover squares, of one s-cover square and ending in the non-overlapping portion of another s-cover square.

Consider the example given in Figure 4.1 where an s-cover $\{S_1 = (1, 2, 1),\ S_2 = (2, 7, 3),\ S_3 = (3, 8, 3),\ S_4 = (4, 9, 3)\}$ of the string x is shown. It satisfies all properties defined in Definition 4.1: any two consecutive squares in the s-cover overlap; every position of x is contained in a square of the s-cover; and there are no intermediate squares.

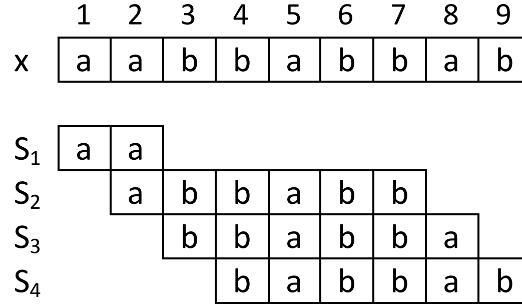

Figure 4.1: s-Cover of string x.

Lemma 4.2 shows that if a string has an s-cover, then the s-cover must be unique. Therefore, a string cannot have more than one s-cover, and thus the relationship between a string and its s-cover is one to one. This ensures that by generating all possible s-covers for given d and n, we are actually generating all possible (d, n)-strings admitting s-covers. We call a string with an s-cover, an *s-covered string*.

Lemma 4.2. *An s-cover of a string is unique.*

Proof. Let us assume that we have two different s-covers of x, $\{S_i \mid 1 \leq i \leq m\}$ and $\{S'_i \mid 1 \leq i \leq k\}$. We shall prove by induction that they are identical. By Definition 4.1 (3), $S_1 \subseteq S'_1$. By the same argument, $S'_1 \subseteq S_1$. Thus, $S_1 = S'_1$, which is the base case. Let the induction hypothesis be $S_i = S'_i$ for $1 \leq i \leq t$. If $t = m = k$, we have $\bigcup_{1 \leq i \leq t} S_i = \bigcup_{1 \leq i \leq t} S'_i = x$ and we are done. Otherwise consider S_{t+1}. By Definition 4.1 (3), there is an S'_v so that $S_{t+1} \subseteq S'_v$ and $v \geq t+1$. We need to show that $v = t+1$. Suppose $v > t+1$, then there exists a square S'_{t+1} in x such that the starting position of S'_{t+1} is in the non-overlapping portion of S_t and ending position is in the non-overlapping portion of S_{t+1}, which forms an intermediate square, contradicting with Definition 4.1 (3). Thus, $v = t+1$, and $S_{t+1} \subseteq S'_{t+1}$. By the same argument, $S'_{t+1} \subseteq S_{t+1}$, therefore $S_{t+1} = S'_{t+1}$. □

Lemma 4.3 shows that any s-covered string is singleton-free.

Lemma 4.3. *An s-covered string is singleton-free.*

Proof. Let $\{S_j \mid 1 \le j \le m\}$ be the s-cover of string $x[1 \mathinner{..} n]$. For any $1 \le i \le n$, $x[i] \in S_t$ for some $1 \le t \le m$ by Definition 4.1 (2). Since S_t is a square, the symbol $x[i]$ occurs in x at least twice. □

4.1.2 Core Vector

Before we can define what a *dense string* is, we must first define the notion of the *core of square*, followed by the definition of the *core vector*. The intuition behind the notion of the core of a square is similar to the core of a run [15], i.e. the set of indices i of x such that if $x[i]$ is replaced by a symbol not occurring in x, all occurrences of the square are destroyed.

Definition 4.4. *The* **core** *of a square S is the set of indices formed by the intersection of the sets of indices of all occurrences of S in the string.*

	1	2	3	4	5	6	7	8	9	10
x	a	a	b	a	b	a	b	b	a	a

Square	Occurrences	Core
aa	(1, 2, 1), (9, 10, 1)	∅
abab	(2, 5, 2), (4, 7, 2)	{4, 5}
baba	(3, 6, 2)	{3, 4, 5, 6}
bb	(7, 8, 1)	{7, 8}

Figure 4.2: Core of each square in string x.

The example shown in Figure 4.2 demonstrates three different scenarios when determining the core of a square: square *abab* has two occurrences in x, and the intersection of the indices of these two occurrences is the core for *abab*; square *aa* also has two occurrences, however they do not intersect, thus the intersection of the

indices is an empty set and therefore so is the core; and the core for single occurrence squares like *baba* and *bb* is the indices of itself.

Definition 4.5. *For a string x with length of n, the* ***core vector*** $k(x) = [k_1(x), k_2(x), \cdots, k_n(x)]$ *of x is defined by $k_i(x) =$ the number of cores of squares containing i in x for $1 \leq i \leq n$.*

To put it simply, the core vector $k(x)$ denotes the number of distinct squares that a position contains in x. For instance, if $k_i(x) = p$ for position i in x, it means there are p number of cores of squares that contain position i. Therefore, if we were to replace the symbol $x[i]$ with a new symbol that is not in $\mathcal{A}(x)$, the number of distinct primitively rooted squares in x would be decreased by p. Note that we also refer to the core vector as *k-vector* in Section 5.3.

	1	2	3	4	5	6	7	8	9	10
x	a	a	b	a	b	a	b	b	a	a
k	0	0	1	2	2	1	1	1	0	0

Figure 4.3: Core vector of string x.

As illustration, Figure 4.3 shows the core vector $k(x)$ of the same string x which was used in the previous example. The value of each position in $k(x)$ is determined by the number of cores of squares this position contains, i.e. the number of times this position occurs in the “Core” column of the table shown in Figure 4.2. For example, position 4 and 5 occurs twice, that means there are two distinct squares whose cores contain position 4 and 5, thus $k_4(x) = k_5(x) = 2$; similarly, position 3, 6, 7, and 8 occur once, then $k_3(x) = k_6(x) = k_7(x) = k_8(x) = 1$, and position 1, 2, 9, and 10 does not occur, then $k_1(x) = k_2(x) = k_9(x) = k_{10}(x) = 0$.

4.1.3 Dense Strings

Definition 4.6. *For a given lower bound $\sigma_d^-(n)$, a (d,n)-string x is **dense**, if its core vector $k(x)$ satisfies $k_i(x) > \sigma_d^-(n) - s(x[1 \,..\, i-1]) - m_i$, for any $1 \le i \le n$, where $m_i = \max\{\sigma_{d'}(n-i) \;:\; d - |\mathcal{A}(x[1 \,..\, i-1])| \le d' \le \min(n-i, d)\}$.*

In Definition 4.6, m_i denotes the maximum number of distinct primitively rooted squares that a string of length $n-i$ and proper number of distinct symbols d' could contain. Let y be such a string, then $s(y) = \sigma_{d'}(n-i) = m_i$, and $|\mathcal{A}(y) \cup \mathcal{A}(x[1 \,..\, i-1])| = d$.

When we generate s-covered candidate strings, the density condition enforces the minimum number of cores each position must contain in order for the number of distinct primitively rooted squares of the candidate string exceeding $\sigma_d^-(n)$. Since $s(x[1 \,..\, i-1])$ is the number of distinct squares for the portion of candidate string we have generated so far, m_i is an estimate for the portion of string we have not generated yet. If any position of the core vector does not meet the density condition, further generation of the candidate string aborts, otherwise it continues till the entire string is built. We will discuss the details in Section 4.2.

4.1.4 s-Covered Strings

In this section we make use of the definitions from the previous sections and present a number of lemmas showing that the set of square-maximal strings with number of distinct primitively rooted squares exceeding $\sigma_d^-(n)$ is a subset of the set of dense s-covered strings.

Lemma 4.7. *If a (d,n)-string x is not dense, then $s(x) \le \sigma_d^-(n)$.*

Proof. If x is not dense, by Definition 4.6, for some i_0, $k_{i_0}(x) \le \sigma_d^-(n) - s(x[1 \,..\, i_0 - 1]) - m_{i_0}$. The proof follows from the basic observation that for any string x, $s(x) \le$

$s(x[1 .. i-1])+k_i(x)+s(x[i+1 .. n])$, where $k(x)$ is the core vector of x, and $1 \leq i \leq n$. Note that the inequality occurs when there are the same types of squares occurring in both $x[1 .. i-1]$ and $x[i+1 .. n]$. Thus, $s(x) \leq s(x[1 .. i_0-1])+k_{i_0}(x)+s(x[i_0+1 .. n])$. Since $s(x[i_0+1 .. n]) \leq m_{i_0}$ as m_{i_0} is the maximum number of distinct squares with length $n-i_0$ and appropriate d'. Therefore, $s(x) \leq s(x[1 .. i_0-1]) + k_{i_0}(x) + m_{i_0} \leq s(x[1 .. i_0-1]) + (\sigma_d^-(n) - s(x[1 .. i_0-1]) - m_{i_0}) + m_{i_0} = \sigma_d^-(n)$. □

Lemma 4.8. *If the core vector $k(x)$ of a (d,n)-string x satisfies $k_i(x) > 0$ for every $1 \leq i \leq n$, then x has an s-cover.*

Proof. We build an s-cover by induction. Since $k_1(x) \geq 1$, position 1 is in at least one core, hence there must be at least one square starting at position 1. Among all squares starting at position 1, set the one with the largest period to be S_1. Suppose that we have built the s-cover $\{S_i = (s_i, e_i, p_i) \mid 1 \leq i \leq t\}$. If $\bigcup_{1 \leq i \leq t} S_i = x$, we are done. Otherwise $\bigcup_{1 \leq i \leq t} S_i = x[1 .. e_t]$ where $e_t < n$. Since $k_{e_t+1}(x) \geq 1$, there is at least one square (s, e, p) in x so that $s \leq e_t + 1 \leq e$. From all such squares choose the leftmost ones, and among them choose the one with the largest period and set it to S_{t+1}. We continue this process until every position of x is covered. It is straightforward to verify that all the conditions of Definition 4.1 are satisfied and that we have built the s-cover of x. □

Note that for a (d,n)-string having an s-cover implies the string being singleton-free by Lemma 4.3. However it does not imply that $k_i(x) \geq 1$ for every $1 \leq i \leq n$ where $k(x)$ is the core vector of x, even though it is quite close to it. Consider the s-cover $\{S_j = (s_j, e_j, p_j) \mid 1 \leq j \leq m\}$ of x. If S_1 has another occurrence in x and there is no other square in x starting at position 1, then position 1 is not in any core and $k_1(x) = 0$. Similarly, if the s-cover has two consecutive adjacent squares S_j and S_{j+1}, S_{j+1} is the only square that starts at position s_{j+1} and S_{j+1} has some

other occurrence in x, then $k_{s_{j+1}}(x) = 0$. In a sense, the s-cover is a computationally efficient structural generalization of the property that every $k_i(x) \geq 1$.

Lemma 4.9. *If a singleton-free square-maximal (d, n)-string x does not have an s-cover, then $\sigma_d(n) = \sigma_d(n-1)$.*

Proof. Since x does not have an s-cover, there exist some i_0 such that $k_{i_0} = 0$ by Lemma 4.8. Remove $x[i_0]$ to form a $(d, n-1)$-string y. This will not decrease the number of distinct squares in x since there is no core of any square containing i_0. Then $\sigma_d(n) = s(x) \leq s(y) \leq \sigma_d(n-1)$. Since $\sigma_d(n) \geq \sigma_d(n-1)$ by Proposition 2.6 (a), $\sigma_d(n) = \sigma_d(n-1)$. □

Lemma 4.10. *If a square-maximal (d, n)-string has a singleton, then $\sigma_d(n) = \sigma_{d-1}(n-1)$.*

Proof. Remove the singleton to form a $(d-1, n-1)$-string y. Then $\sigma_d(n) = s(x) \leq s(y) \leq \sigma_{d-1}(n-1)$. Since $\sigma_d(n) \geq \sigma_{d-1}(n-1)$ by Proposition 2.6 (b), $\sigma_d(n) = \sigma_{d-1}(n-1)$. □

Lemma 4.11. *If a square-maximal (d, n)-string x has an s-cover with two consecutive adjacent squares, then $\sigma_d(n) \leq \sigma_{d_1}(n_1) + \sigma_{d_2}(n_2)$ for some $2 \leq d_1, d_2 \leq d \leq d_1 + d_2$ and some n_1, n_2, possibly equal to zero, such that $n_1 + n_2 = n$.*

Proof. Let $\{S_i = (s_i, e_i, p_i) \mid 1 \leq i \leq m\}$ be the s-cover of x and $S_j \cap S_{j+1} = \varepsilon$, i.e. S_j and S_{j+1} are adjacent. Then $s(x) \leq s(x_1) + s(x_2)$, where $x_1 = \bigcup_{1 \leq i \leq j} S_i$ and $x_2 = \bigcup_{j+1 \leq i \leq m} S_i$. The inequality occurs when there are the same types of squares appearing in both x_1 and x_2. Therefore, $\sigma_d(n) = s(x) \leq s(x_1) + s(x_2) \leq \sigma_{d_1}(n_1) + \sigma_{d_2}(n_2)$ where x_1 and x_2 is a (d_1, n_1)-string and a (d_2, n_2)-string, respectively. □

Figure 4.4 summarizes the situation: for any square-maximal (d, n)-string, if it has singletons, then $\sigma_d(n) = \sigma_{d-1}(n-1)$ by Lemma 4.10. If it is singleton free and

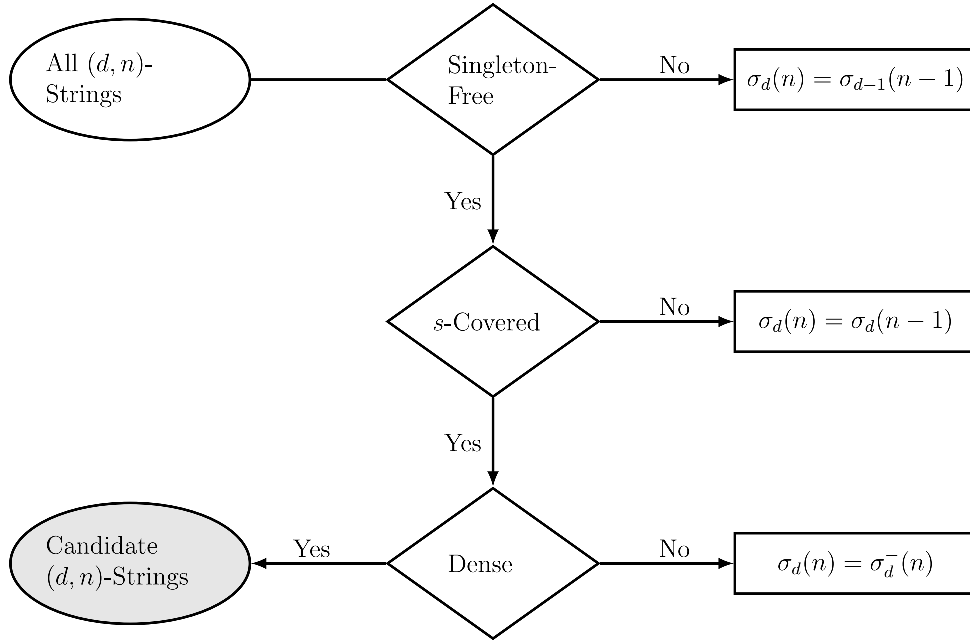

Figure 4.4: Candidate strings.

not s-covered, then $\sigma_d(n) = \sigma_d(n-1)$ by Lemma 4.9. If it is singleton free, s-covered, and not dense, then $\sigma_d(n) = \sigma_d^-(n)$ by Lemma 4.7. Thus, the only unknown is represented by singleton free, s-covered, and dense strings which we call candidate strings. Thus, computing the maximum m over all candidate strings, and selecting $\max\{\sigma_{d-1}(n-1),\ \sigma_d(n-1),\ \sigma_d^-(n),\ m\}$ gives us the true value of $\sigma_d(n)$. Note that as we will discuss in Section 4.3.1, $\sigma_d^-(n)$ was obtained from the maximum of the previously known values including $\sigma_{d-1}(n-1)$ and $\sigma_d(n-1)$, therefore the true value of $\sigma_d(n)$ is essentially $\max\{\sigma_d^-(n),\ m\}$.

Furthermore, Lemma 4.11 shows that if an s-covered square-maximal string has two consecutive adjacent squares, then $\sigma_d(n)$ is no bigger than the sum of $\sigma_{d_1}(n_1)$ and $\sigma_{d_2}(n_2)$ with appropriate d_1, d_2, and $n_1 + n_2 = n$; in other words, the value of $\sigma_d(n)$ can be determined by the previously known values as we incrementally compute the maximum number of distinct squares to populate the $(d, n-d)$-table from top to

down and left to right directions.

Therefore, in order to compute $\sigma_d(n)$, we are only interested in those dense s-covered strings with no consecutive adjacent s-cover squares because they are the candidate strings with the number of distinct squares that could possibly exceed the lower bound. To illustrate how significant is the discussed reduction of the pool of candidate strings, we show in Table 4.1 the comparison of the number of s-covered strings and non s-covered strings for selected small values of d and n. As we can see, the number of s-covered strings is significantly smaller than the non s-covered string for every row. Thus, for our computational interests, the number of s-covered strings that are also dense and with no consecutive adjacent squares is even a more significant reduction.

d	n	# s-Covered Strings	# Non s-Covered Strings
2	10	154	357
2	15	4074	12,309
2	20	109,437	414,850
3	10	183	9,147
3	15	21,681	2,353,420
3	20	1,908,923	578,697,523

Table 4.1: Comparison on number of s-covered strings and non s-covered strings.

4.2 Generating s-Covers

In Section 4.1 we gave the definitions of s-cover and dense strings, we also proved that in order to compute the maximum number of distinct primitively rooted squares for (d, n)-strings, it is sufficient to consider a set of dense s-covered strings with no consecutive adjacent squares in the s-cover. In this section, we focus on the programming perspectives on how to recursively generate these strings.

4.2.1 Algorithm

Recall Lemma 4.2 that every s-covered string has a unique s-cover. Therefore, we can speak of the s-cover of a string and thus building a string is the same thing as building the s-cover. When building an s-covered string, the program starts with the first square of the s-cover. Let the period of the first square be denoted as p_1, then p_1 ranges from 1 to $\lfloor \frac{n}{2} \rfloor$. For each value of p_1, **Main()** function of the program calls **ExtendSquare()** to build all possible primitive generators for the first square. Once a generator is properly built, it then calls **TidySqaure()**, which conducts a series of validations of the generator. Once all the validations are passed, and the length of the built string reaches n and number of distinct symbols reaches d, the entire string is built, its number of distinct squares is calculated and the result is outputted. However, if the length is not full, the program calls **NextSqaure()** to build the next square of the s-cover, all possible generators with different periods are built. Depending on the starting position and the period length of the next square, the generator of the next square may be partially or completely determined from the previous square, i.e. whether the overlapped portion of the two squares is large enough to exceed the period length. If the generator of next square is completely determined from the previous square, **TidySqaure()** is called; otherwise, **ExtendSquare()** is called to finish building the generator. It recursively does this process until the string of the full length is built. When all s-covered strings with first square period of current p_1 are built, the program continues with next possible value for p_1 for the first square. Figure 4.5 shows the flow chart of the program, and the dash boxed portion presents the recursive generation of s-covered strings.

ExtendSquare(): This function is used to build a full generator (when it is called by **Main()** function to build the first square), or a partial generator (when it is

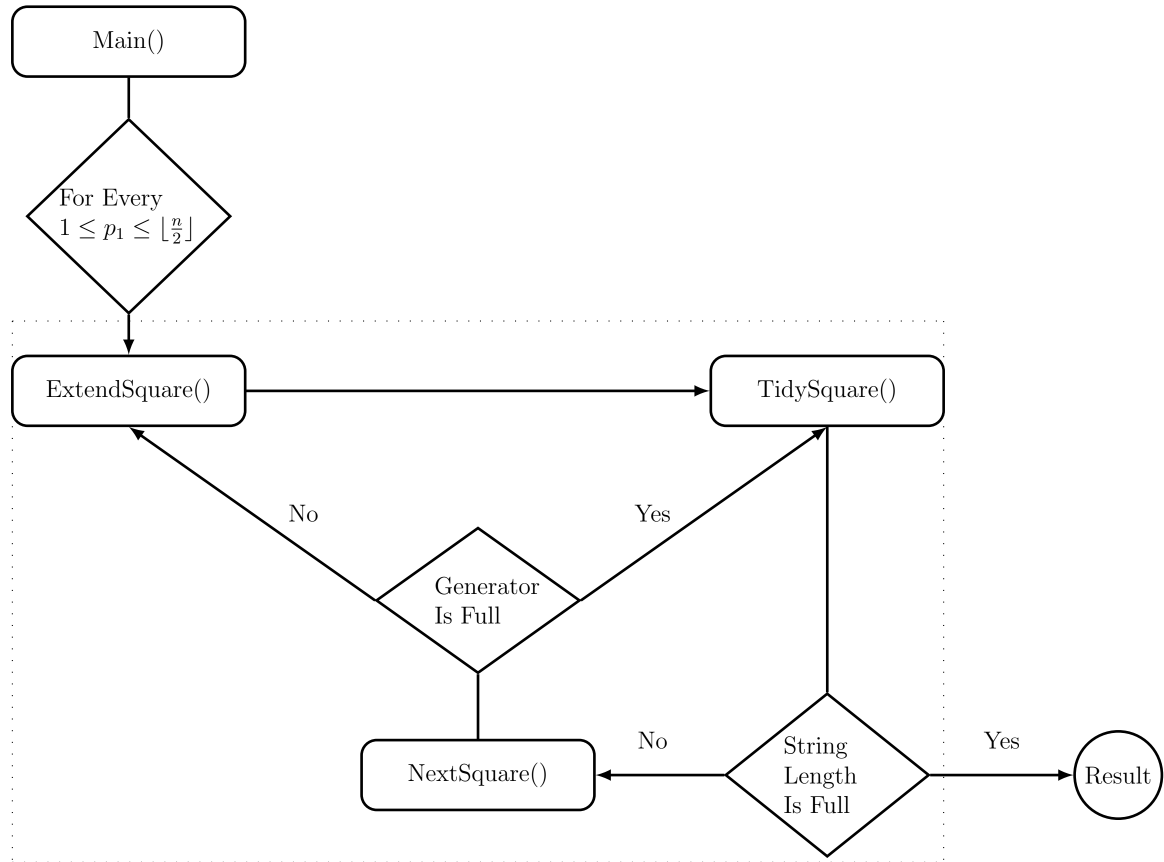

Figure 4.5: s-Cover generation.

called by **NextSquare()** to build a subsequent square) of the s-cover square. It iteratively sets each position with all possible symbols that are either previously used or unused, until the required period length is reached. Since we are working with ordered alphabet, the function ensures that the symbol for each position does not exceed the maximum symbol, that is, the largest symbol possible in corresponding to the number of distinct symbols d in the string. It also ensures that the symbol in the current position is no bigger than 1 compared with the symbol in the previous position so that the resulting string is in its smallest lexicographic form. For example, ac is not allowed since it is not in its smallest lexicographic form as we can replace c by b.

TidySquare(): When a generator is fully built, this function is called for a series of validation checks to ensure that the square we have built satisfies all the requirements so that the resulting string can be a candidate for computing $\sigma_d(n)$. These validation checks include conflict check, primitiveness check, no intermediate square check and density check, which will be discussed individually in the next few sections. Once all the validations have been passed, the function checks if the string generated so far has the required length n and number of distinct symbols d; if so, the entire string has been built and it then calculates the number of distinct squares of the string and output the result if a better value is produced compared with the known lower bound; if not, function **NextSquare()** is called to continue the process of generating the subsequent squares of the string.

NextSquare(): This function is called by **TidySquare()** if the length of the string built so far is not of a full length and more s-cover squares are needed. By definition of s-cover, two consecutive s-cover squares are either overlapping or adjacent. By Lemma 4.11, we do not need to consider the case of adjacent s-cover squares. Thus, the starting position of the current square is ranged from the first position after the starting position of the previous square, to the ending position of the previous square. Note that when applying the density check, the range of the starting position of the current square may be reduced. We will discuss the details in Section 4.2.5. Depending on the size of the overlapping portion and the period of the current square, the generator of the current square is either fully or partially determined from the previous square. If it is fully determined, then the program calls **TidySquare()** to perform the validation checks to either accept or reject the square; if it is only partially determined,

ExtendSquare() is called to build all possible completions of the generator. For example, the first s-cover square S_1 is built and it is *aabaab*. When building $S_2 = (s_2, e_2, p_2)$, then s_2 ranges from position 2 to position 6. If $s_2 = 2$ and $p_2 = 4$ then the generator of S_2 is fully determined; if $s_2 = 4$ and $p_2 = 4$, then the generator of S_2 is partially determined.

4.2.2 Conflict Check

As one of the validations in **TidySquare()**, conflict check is required when the current square's generator is fully contained within the previous square. It ensures that the remaining overlapping portion of the two squares is identical to the starting portion of the current square, i.e. the remaining overlapping portion is a prefix of the current square's generator, because otherwise it would be impossible to form a square. For the same example used previously, S_1 is built and it is *aabaab*. If $s_2 = 2$ and $p_2 = 4$, the generator of S_2 is fully determined. However, s_2 fails the conflict check as the remaining portion b is not a prefix of the generator *abaa*. Thus, S_2 cannot form a square.

The function iterates through every remaining overlapping position, and compares it with the corresponding starting position of the current square; if any of them are not identical, then this generator is impossible to form a square and thus it is rejected. Algorithm 4.1 shows the pseudo-code for the conflict check. $period[]$ is an array containing the period for each s-cover square, i.e. $period[currentSQ]$ denotes the period of the current square. $start[]$ contains the starting position for each s-cover square and $str[]$ contains the s-covered string built so far, i.e. $str[i]$ denotes the symbol at position i in the string.

Algorithm 4.1 Conflict check.

```
p ← period[currentSQ]
i ← start[currentSQ] + p
while str[i]! = NULL do
    if str[i]! = str[i − p] then
        return False
    end if
    i ← i + 1
end while
return True
```

4.2.3 Primitiveness Check

By Definition 4.1, s-cover is a series of primitively rooted squares. Therefore, the s-cover squares we build have to be primitive. Primitiveness check ensures the generator of the s-cover square is primitive; that is, itself is not a repetition. Recall a non-empty string v is not primitive if it can be written in the form of $v = (u)^e$, where u is a non-empty string and e is an integer such that $e \geq 2$. Logically, we would have to check every possible $e \geq 2$ to see if v can be equally divided into e identical parts. However, a simple observation can save us some computational effort. Consider a non-primitive string v can be written in the form of $v = (u)^4$, note that it can be also written in the form of $v = (u')^2$ where $u' = (u)^2$. Therefore, if a string can be equally divided into 4 identical parts, then it has to be able to be divided into 2 identical parts as well since 4 is divisible by 2. Similarly for other multiples of 2 such as 6, 8, etc. if v can be divided into 6, 8, etc. identical parts, then v must be able to divided into 2 identical parts. Same argument applies to the multiples of 3, 5, 7, etc. This observation shows that we do not have to check every number of $e \geq 2$ since any multiples of the previously checked numbers would be redundant, i.e. if v can be equally divided for $e = 4$, then we would have been found out it when we checked $e = 2$. In other words, to check a string if it is primitive, it is sufficient

to check only the prime numbers that are greater or equal than 2. The pseudo-code of the primitiveness check is listed in Algorithm 4.2. *len* denotes the length of the generator, and *start* denotes the starting position of the generator.

Let us remark that there exist linear algorithms for checking primitiveness. A string v is primitive if in v^2 there are only two occurrences of v. Thus, we can form v^2 and use any linear pattern matching algorithm such as KMP (Knuth, Morris, and Pratt's algorithm) [25] to determine whether v^2 contains 2 or more occurrences of v. If the result is 2, v is primitive, otherwise it is not. However, the implementations of efficient linear pattern matchers are not trivial and would thus significantly complicate and extend our code, and since we are not dealing with very large numbers of n and d, our approach using the prime numbers that are not computed on the fly but pre-computed and stored is simple and efficient.

4.2.4 No Intermediate Square Check

By the definition of s-cover in Definition 4.1 (3), there cannot be any intermediate squares occurring across two s-cover squares; that is, there cannot exist a square that starts in one s-cover square, and ends in another s-cover square. Therefore, when we recursively build s-cover squares, we need to explicitly check if there is an intermediate square created between the previously built s-cover squares and the current s-cover square; if there is, then we must reject the current square for the s-cover being built.

Algorithm 4.3 is the pseudo-code for this function. Basically, the program iterates all possible intervals of the string, where the starting position of the interval is ranged from position 1 to the position before the starting position of the current s-cover square, and ending position of the interval is ranged from the starting position of the current s-cover square to the ending position of the current s-cover square. For

Algorithm 4.2 Primitiveness check.

```
for each prime e ≤ len do
  if (len%e)! = 0 then
    CONTINUE
  end if
  isPower ←True
  u ← len/e
  for i = start → start + u − 1 do
    for j = 1 → e − 1 do
      if str[i]! = str[i + j × u]) then
        isPower ←False
        BREAK
      end if
    end for
    if !isPower then
      BREAK
    end if
  end for
  if isPower then
    return False
  end if
end for
return True
```

each such interval, it checks whether that interval forms a square; if it does, then the program checks if the square is one of the s-cover squares; if not, that means an intermediate square is found that does not belong to the s-cover and occurs across from previously built s-cover squares to the current s-cover square. $intStart$ respective $intEnd$ denote the starting respective ending position of the interval. $intPer$ is the period of the interval. $curStart$ is the starting position of the current s-cover square, and $curEnd$ is the ending position. $SQ[intStart, intEnd]$ denotes the square that is formed by the interval, and $SCoverList$ contains the list of the s-cover squares we built so far.

Algorithm 4.3 No intermediate square check.

```
intStart ← 1
while intStart < curStart do
  intEnd ← curStart
  while intEnd ≤ curEnd do
    if (intEnd − intStart)%2) == 1 then
      intPer ← (intEnd − intStart + 1)/2
      i ← 0
      while i < intPer and str[intStart + i] == str[intStart + i + intPer] do
        i ← i + 1
      end while
      if i == intPer and SQ[intStart, intEnd] ∉ SCoverList then
        return True
      end if
    end if
    intEnd ← intEnd + 1
  end while
  intStart ← intStart + 1
end while
return False
```

4.2.5 Density Check

As discussed previously, computationally we are interested in dense s-covered strings as they are the candidates to have possibly more distinct primitively rooted squares than the lower bound. Density check is to ensure the s-covered string being generated satisfies the conditions of density as defined in Definition 4.6.

If we were to apply density check to the entire s-covered string after it has been built, we would not gain much computational efficiency as the whole string would have been built just to be discarded. In addition, once having the whole string, one could directly compute its number of distinct squares and output the result. If a density check could be applied to a partially built string as each s-cover square is added, we could eliminate the non-dense strings as early as possible. However, in general, the density of a prefix is not an indicator of the density of the same prefix after the

whole string is generated, as many additional squares starting in the prefix might be generated. The only situation when the density of a prefix cannot be increased is when the prefix is the portion of the string from the beginning to the last position before an s-cover square starts. To be able to apply density check to a partial s-covered string, we need to show the strings we reject at early stage as non-dense, will not become dense when they are completed. Lemma 4.12 shows that for any position, the value in the core vector of any prefix that ends in the last position before an s-cover square starts is greater or equal to the value in the core vector of the entire string.

Lemma 4.12. *Let* $\{S_i = (s_i, e_i, p_i) \mid 1 \le i \le m\}$ *be an s-cover of x. Let $k(x)$ be the core vector of x. Then for any* $1 \le i < m$ *and the core vector* $k'(x[1\ ..\ e_i])$, $(\forall\ 1 \le j < s_{i+1})(k'_j(x[1\ ..\ e_i]) \ge k_j(x))$.

Proof. Let us assume for some $1 \le i < m$ there exists a j such that $1 \le j < s_{i+1}$, and $k_j(x) > k'_j(x[1\ ..\ e_i])$. Then there must exists a core of a square containing j in x, but not in $x[1\ ..\ e_i]$. In other words, there exists a square (s, e, p) in x that is not a square of $x[1\ ..\ e_i]$, i.e. $s < j < s_{i+1}$ and $e > e_i$. Thus, this square is an intermediate square which violates the definition of s-cover in Definition 4.1 (3), a contradiction. □

Lemma 4.12 ensures that as we recursively build the s-covered string, the values in its core vector will not increase as we add more s-cover squares to the string. In other words, if the density condition is not met for the partially built s-covered string, it will never be met for the entire string no matter what subsequent squares we append to it. Therefore, it is possible to perform density check on partially built s-covered strings.

Algorithm 4.4 lists the pseudo-code for the density check function. The program iterates each position of the current square, the first position that does not satisfy the density condition, is set to the maximal starting position of the next square. That

is because if the starting position of the next square were after this position, the resulting s-covered string would contain this non-dense position, and thus the string would not be dense and should be eliminated. If every position of the current square satisfies the density condition, then the maximal starting position of the next square is set to the ending position of the current square as discussed in the **NextSquare()** function of Section 4.2.1. $cut[nextSQ]$ denotes the maximal starting position of the next square. $start[currentSQ]$ and $end[currentSQ]$ denotes the starting and ending position of the current square, respectively. $prefixSQ[i-1]$ contains the number of distinct squares in the prefix string $x[1..i-1]$, $k[i]$ is the number of cores containing position i, and $maxSQ[n-i]$ contains the maximal number of distinct squares that a string with length $n-i$ could contain, namely, m_i in Definition 4.6.

Algorithm 4.4 Density check.

```
cut[nextSQ] ← 0
i ← start[currentSQ]
while cut[nextSQ] == 0 do
  if prefixSQ[i − 1] + k[i] + maxSQ[n − i] ≤ σ_d^−(n) then
    cut[nextSQ] ← i
  end if
  i ← i + 1
  if i > end[currentSQ] and cut[nextSQ] == 0 then
    cut[nextSQ] ← end[currentSQ]
  end if
end while
```

4.2.6 Parity Condition

When computing the values on the main diagonal of the $(d, n-d)$-table, that is, $\sigma_d(2d)$, we can enforce an additional necessary condition, called *parity condition*, for the s-covered strings we build. Definition 4.13 formally defines the parity condition.

Definition 4.13. *The s-cover* $\{S_i = (s_i, e_i, p_i) : 1 \le i \le m\}$ *of* $x[1 \mathinner{..} n]$ *satisfies the* ***parity condition*** *if for any* $1 \le i < m$, $\mathcal{A}(x[1 \mathinner{..} e_i]) \cap \mathcal{A}(x[s_{i+1} \mathinner{..} n]) \subseteq \mathcal{A}(x[s_{i+1} \mathinner{..} e_i])$.

In short, an s-covered string satisfies the parity condition if for any overlapping s-cover squares, the symbols occurring in both the left non-overlapping portion and the right non-overlapping portion, have to also occur in their overlapping portion. For instance,if the overlapping portion of two s-cover squares in a string $x[1 \mathinner{..} n]$ is $x[i_1 \mathinner{..} i_2]$, then any symbol occurring both in $x[1 \mathinner{..} i_1 - 1]$ and $x[i_2 + 1 \mathinner{..} n]$ must also occur in $x[i_1 \mathinner{..} i_2]$

Consider a square-maximal $(d, 2d)$-string. Either it is singleton-free, which means every symbol occurs exactly twice and then $\sigma_d(2d) = d$ by Lemma 2.5, or it contains singletons. Let us investigate such a string. First, all the singletons can be safely moved to the end of the string without reducing the number of distinct primitively rooted squares, see Lemma 2.2. Lemma 4.14 shows that the singleton-free portion of the string has to have a s-cover satisfying the parity condition. Therefore, for computing $\sigma_d(2d)$, it is sufficient to build s-covers that satisfy the parity condition.

Lemma 4.14. *The singleton-free part of a square-maximal* $(d, 2d)$*-string* x *with all its singletons at the end has an s-cover satisfying the parity condition.*

Proof. We can assume that x has v singletons all at the end, and $0 \le v \le d - 2$ since there are at least 2 symbols that are not singletons. Let $k(x)$ be the core vector of x. Suppose the singleton-free part $x[1 \mathinner{..} 2d - v]$ does not have an s-cover, then there exist some $1 \le i_0 \le 2d - v$ such that $k_{i_0}(x) = 0$, by Lemma 4.8. Remove $x[i_0]$ to form a $(d, 2d - 1)$-string y. Since $k_{i_0}(x) = 0$, there is no core of square containing i_0, then removal of $x[i_0]$ will not decrease the number of squares in x. Thus, $\sigma_d(2d) = s(x) \le s(y) \le \sigma_d(2d - 1)$. By Proposition 2.6 (d), $\sigma_d(2d - 1) = \sigma_{d-1}(2d - 2)$, hence

$\sigma_d(2d) \leq \sigma_{d-1}(2d-2)$, a contradiction since $\sigma_d(2d) > \sigma_{d-1}(2d-2)$ by Proposition 2.6 (c). Therefore, $x[1 .. 2d-v]$ has an s-cover $\{S_i : 1 \leq i \leq m\}$. Let us assume that the s-cover does not satisfy the parity condition.

(i) $\bigcup_{1\leq i\leq t} S_i$ and $\bigcup_{t<i\leq m} S_i$ for some $1 \leq t \leq m$ are adjacent and their respective alphabets have at least one symbol in common, say C. If we replace C in $\bigcup_{1\leq i\leq t} S_i$ by a new symbol $\hat{C} \notin \mathcal{A}(x)$, we get a new $(d+1, 2d)$-string y. $s(y) \geq s(x)$ because the squares with C in $\bigcup_{1\leq i\leq t} S_i$ are replaced by the same number of squares with $\hat{C}$, and we may increase the number of squares if there existed same square types containing C occurring in both $\bigcup_{1\leq i\leq t} S_i$ and $\bigcup_{t<i\leq m} S_i$ in x. Thus, $\sigma_d(2d) = s(x) \leq s(y) \leq \sigma_{d+1}(2d) = \sigma_{d-1}(2d-2)$ by Proposition 2.6 (d), a contradiction since $\sigma_d(2d) > \sigma_{d-1}(2d-2)$ by Proposition 2.6 (c).

(ii) $\bigcup_{1\leq i\leq t} S_i$ and $\bigcup_{t<i\leq m} S_i$ for some $1 \leq t \leq m$ are overlapping, and there is a symbol C occurring in $\bigcup_{1\leq i\leq t} S_i$ and in $\bigcup_{t<i\leq m} S_i$, but not in the overlapping portion $S_t \cap S_{t+1}$. If we replace C in $\bigcup_{1\leq i\leq t} S_i$ by a new symbol $\hat{C} \notin \mathcal{A}(x)$, we get a new $(d+1, 2d)$-string y. By a similar argument as above, $s(y) \geq s(x)$. Thus, $\sigma_d(2d) = s(x) \leq s(y) \leq \sigma_{d+1}(2d) = \sigma_{d-1}(2d-2)$, a contradiction.

□

With additional assumptions, Lemma 4.14 can be strengthened to exclude the consecutive adjacent squares from the s-cover of a square-maximal $(d, 2d)$-string.

Lemma 4.15. *Let $\sigma_{d'}(2d') = d'$ for every $d' < d$. Either $\sigma_d(2d) = d$ or for every square-maximal $(d, 2d)$-string x with v singletons all at the end, $0 \leq v \leq d-2$, and its singleton-free part $x[1 .. 2d-v]$ has an s-cover that has no consecutive adjacent squares and that satisfies the parity condition.*

Proof. The existence of the s-cover $\{S_i \mid 1 \leq i \leq m\}$ of $x[1 \,..\, 2d - v]$ satisfying the parity condition follows from Lemma 4.14. We need to prove that either $\sigma_d(2d) = d$ or there are no adjacent squares in the s-cover. Since $\sigma_{d'}(2d') = d'$ for any $d' < d$, $\sigma_{d'}(n') \leq n' - d'$ for any $n' - d' < d$. Let us assume that the s-cover of x has two adjacent squares S_t and S_{t+1}. Let $x_1 = \bigcup_{1 \leq i \leq t} S_i$ and let $x_2 = \bigcup_{t < i \leq m} S_i$. Then $s(x) \leq s(x_1) + s(x_2)$ where x_1 and x_2 is a (d_1, n_1)-string and a (d_2, n_2)-string, respectively, with $n_1 + n_2 = 2d - v$ and $d_1 + d_2 \geq d - v$. Note that the inequality occurs when there are the same types of squares in both x_1 and x_2. Since the s-cover satisfies the parity condition, $\mathcal{A}(x_1)$ and $\mathcal{A}(x_2)$ are disjoint and hence $d_1 + d_2 = d - v$. Therefore, $(n_1 - d_1) + (n_2 - d_2) = (n_1 + n_2) - (d_1 + d_2) = (2d - v) - (d - v) = d$. Since $n_1 - d_1 < d$ and $n_2 - d_2 < d$, $\sigma_d(2d) = s(x) \leq s(x_1) + s(x_2) \leq \sigma_{d_1}(n_1) + \sigma_{d_2}(n_2) \leq (n_1 - d_1) + (n_2 - d_2) = d$, a contradiction. □

When computing $\sigma_d(2d)$ on the main diagonal of the $(d, n-d)$-table, it is clear that d is a lower bound $\sigma_d^-(2d) = d$. To exceed this lower bound, that is, $\sigma_d(2d) > d$, the square-maximal $(d, 2d)$-strings must contain at least $\lceil \frac{2d}{3} \rceil$ singletons, by Lemma 3.6. Therefore, with $\lceil \frac{2d}{3} \rceil$ singletons, the non-singleton part of $(d, 2d)$-strings are essentially $(d - \lceil \frac{2d}{3} \rceil, 2d - \lceil \frac{2d}{3} \rceil)$-strings. By Lemmas 4.14 and 4.15, to determine $\sigma_d(2d)$, we only need to consider dense s-covered $(d - \lceil \frac{2d}{3} \rceil, 2d - \lceil \frac{2d}{3} \rceil)$-strings (with lower bound of d) which contain no consecutive adjacent squares and satisfy the parity condition.

4.3 Lower Bound Determination

In the previous sections we discussed our framework for computing the maximum number of distinct primitively rooted squares for given d and n. Our discussion was based on an assumption that a lower bound $\sigma_d^-(n)$ for $\sigma_d(n)$ is available. Thus, in our framework we generate a set of s-covered strings that are dense enough to have at

least $\sigma_d^-(n)+1$ distinct primitively rooted squares. Therefore, to save computational efforts, it is essential to determine a lower bound as close to $\sigma_d(n)$ as possible since, the tighter the lower bound is, the higher the threshold for the density check is; that is, the smaller the pool of candidates is. In this section, we discuss how to obtain an efficient lower bound $\sigma_d^-(n)$ for $d \geq 2$ and $n > 2d$, $\sigma_d^-(2d)$ for the values on the main diagonal, and $\sigma_2^-(n)$ for $d = 2$.

4.3.1 Lower Bound $\sigma_d^-(n)$

As we populate the $(d, n-d)$-table incrementally from left to right and top to down, that is, computing the first (leftmost along the row and topmost along the column) unknown entry $\sigma_d(n)$ in the table based on the previously known values; namely, the value immediately to its left $\sigma_d(n-1)$, the value immediately above it $\sigma_{d-1}(n-1)$, and the value to its top-left along the descending diagonal $\sigma_{d-1}(n-2)$. To visualize these entries in the $(d, n-d)$-table, the positions of $\sigma_d(n)$ and the surrounding entries that determine its lower bound are shown in Table 4.2. Note that the three known entries are displayed in gray. In Section 2.3, a number of basic properties of the $(d, n-d)$-

	$\cdots$	$n-d-1$	$n-d$	$\cdots$
$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$
$d-1$	$\cdots$	$\sigma_{d-1}(n-2)$	$\sigma_{d-1}(n-1)$	$\cdots$
d	$\cdots$	$\sigma_d(n-1)$	$\sigma_d(n)$	
$\cdots$	$\cdots$	$\cdots$		

Table 4.2: Determining a lower bound for $\sigma_d(n)$ in $(d, n-d)$-table.

table was discussed and proved. These properties can be used as a foundation for determining the lower bound. Proposition 2.6 (a) and (b) proves that the entries are non-decreasing in both left-to-right and top-to-down directions in the table; that is, $\sigma_d(d)$ is at least as big as $\sigma_d(n-1)$ and $\sigma_{d-1}(n-1)$. In addition, Proposition 2.6(c)

shows that the values are strictly increasing when moving from top-left to bottom-right along any descending diagonal in the table; that is, $\sigma_d(d)$ is at least as big as $\sigma_{d-1}(n-2)+1$. Therefore, the lower bound $\sigma_d^-(d)$ is the maximal value among the three. Therefore, we define $\sigma_d^-(n)$ for general d and n, i.e. when $d \geq 2$ and $n > 2d$.

$$\sigma_d^-(n) = \max\,\{\sigma_d(n-1),\ \sigma_{d-1}(n-1),\ \sigma_{d-1}(n-2)+1\}$$

4.3.2 Lower Bound $\sigma_d^-(2d)$

For entries on the main diagonal of the $(d, n-d)$-table, the determination of a suitable lower bound could be the same as for the general case discussed in Section 4.3.1. That is, we could set

$\sigma_d^-(2d) = \max\,\{\sigma_d(2d-1),\ \sigma_{d-1}(2d-1),\ \sigma_{d-1}(2d-2)+1\}.$

But we can do slightly better. By Proposition 2.6(d), the values under and on the main diagonal along a column are constant. In other words, for the first unknown $\sigma_d(2d)$, the value on its left and the value on its top-left are identical, that is, $\sigma_d(2d-1) = \sigma_{d-1}(2d-2)$, hence $\sigma_d(2d-1) < \sigma_{d-1}(2d-2)+1$. Also as mentioned in Section 4.2.6, we could directly give a lower bound of d for $\sigma_d(2d)$ by constructing a $(d, 2d)$-strings with only pairs, i.e. singleton-free. Therefore, the lower bound for $\sigma_d^-(2d)$ is defined as followed.

$$\sigma_d^-(2d) = \max\,\{d,\ \sigma_{d-1}(2d-1),\ \sigma_{d-1}(2d-2)+1\}$$

4.3.3 Heuristic Search for $\sigma_2^-(n)$

When considering $\sigma_2^-(n)$, the situation is slightly different from the previously discussed method of obtaining a suitable lower bound. The reason is that $d = 2$ row in

the $(d, n-d)$-table is the top-most row and hence there are no entries above it or on its top-left along the descending diagonal. That is, $\sigma_1(n-1) = \sigma_1(n-2) = 1 < \sigma_2(n-1)$, for $n \geq 5$. Therefore, $\sigma_2^-(n) = \sigma_2(n-1)$. In other words, the best lower bound we could get from the existing computed values is the same value as the entry to its left. As we will discuss in Section 4.4, $\sigma_d(n)$ is at most $\sigma_d(n-1) + 2$. So far, from all the computed values, no n satisfy $\sigma_2(n) = \sigma_2(n-1) + 2$, and only relatively few n satisfy $\sigma_2(n) = \sigma_2(n-1)$. Thus, to obtain a computationally efficient lower bound for $\sigma_2(n)$, we fist try to generate a $(2, n)$-string with $\sigma_2(n-1) + 1$ distinct primitively rooted squares which, if it exists, would provide a lower bound $\sigma_2^-(n)$.

Therefore, when computing $\sigma_2(n)$, before recursively generating all the s-covered strings, we generate only a small set of s-covered strings with certain properties based on a heuristic and search among them for a string giving $\sigma_2(n-1) + 1$ distinct primitively rooted squares. This heuristic was determined by analyzing the previously computed square-maximal strings of shorter lengths with a hope that s-covered strings with these properties would likely produce $\sigma_2(n-1) + 1$ distinct primitively rooted squares. Thus, three parameters were introduced into the heuristic search for $\sigma_2^-(n)$: the heuristic can be simply described as (*i*) generate only s-covered strings with a "balanced" number of a's and b's – defined by the maximum difference parameter, (*ii*) the s-cover squares have limited periods – defined by the maximum period parameter, and (*iii*) the generated strings are forbidden to contain segments of consecutive identical symbols – defined by the maximum block parameter. They are described below.

Maximum Difference

In any prefix of the generated string, the difference of the number of a's and the number of b's is no more than a predefined constant we call *maximum difference.*

Maximum Period

The *maximum period* parameter specifies the maximum period that is allowed for each s-cover square. That is, for any square in the s-cover of the string, its period does not exceed this predefined parameter.

Maximum block

The *maximum block* parameter specifies the maximum number of consecutive occurrences of a symbol that is allowed in a string. For example, if *maximum block* is set to 2 in the heuristic search, then it means aaa or bbb is not allowed when building the s-covered string.

	Max.Diff.	Max.Per.	Max.Bl.	String Found with $\sigma_2(n-1)+1$ Squares
$\sigma_2^-(34)$	3	8	2	$abbabbababbababaababaabaababaabaab$
$\sigma_2^-(35)$	3	8	2	$abbabbababbababaababaabaababaabaaba$
$\sigma_2^-(36)$	3	8	2	$abbabbababbababaababaabaababaabaabab$
$\sigma_2^-(37)$	4	12	2	$aabababbabababbababbabababbababbababa$
$\sigma_2^-(38)$	4	12	2	$aabababbababbabababbababbabababbababab$

Table 4.3: Heuristic search parameters for $d = 2$, $34 \leq n \leq 38$.

Table 4.3 illustrates the parameters we used for the heuristic search for $\sigma_2^-(34)$ to $\sigma_2^-(38)$. The first string that contains $\sigma_2(n-1)+1$ distinct primitively rooted squares for each entry is listed in the table and all of these strings were found within a few seconds. As one would expect, the values of the three parameters gradually increase as the length of strings increases.

Now we can formulate how the lower bound for $\sigma_2(n)$ is obtained by the heuristic search. Let $\mathcal{T}_2(n)$ denote the set of all dense s-covered strings complying with the three heuristic conditions discussed above. Then,

$$\sigma_2^-(n) = \max \{ \ \sigma_2(n-1), \ \max_{x \in \mathcal{T}_2(n)} s(x) \}.$$

When we performed the heuristic searches to generate the pool of strings achieving $\sigma_2(n-1)+1$ distinct primitively rooted squares for $\sigma_2^-(n)$ for various n's, we observed an interesting property, namely, a string that achieves $\sigma_2(n)+1$ when computing $\sigma_2^-(n+1)$, may have exactly the same structure as a string that achieves $\sigma_2(n-1)+1$ when computing $\sigma_2^-(n)$, except that there is an a or b appended to the end. In other words, once we have a string achieving $\sigma_2(n-1)+1$ for $\sigma_2^-(n)$, when we add one more character a or b to the end of the string, the total number of distinct squares is increased by 1, and thus this string admits $\sigma_2(n)+1$ distinct squares and can serve as a lower bound for $\sigma_2(n+1)$. Tables 4.4 and 4.5 show the strings that hold this property for $45 \le n \le 46$ and $52 \le n \le 55$, respectively.

	String x		$\lvert x \rvert$	$s(x)$
$\sigma_2(44)$=33	*aabaababaababaabaababaababaabaababaabaababbab*		45	34
$\sigma_2(45)$=34	*aabaababaababaabaababaababaabaababaabaababbab*	*b*	46	35
$\sigma_2(46)$=35	*aabaababaababaabaababaababaabaababaabaababbab*	*ba*	47	36

Table 4.4: Strings of length n+1 with $\sigma_2(n)$+1 distinct squares for $45 \le n \le 46$.

	String x		$\lvert x \rvert$	$s(x)$
$\sigma_2(51)$=38	*aababbabbbabbabbbabbbbabbbabbbbabbbbbabbbbabbbbbabbb*		52	39
$\sigma_2(52)$=39	*aababbabbbabbabbbabbbbabbbabbbbabbbbbabbbbabbbbbabbb*	*b*	53	40
$\sigma_2(53)$=40	*aababbabbbabbabbbabbbbabbbabbbbabbbbbabbbbabbbbbabbb*	*ba*	54	41
$\sigma_2(54)$=41	*aababbabbbabbabbbabbbbabbbabbbbabbbbbabbbbabbbbbabbb*	*bab*	55	42
$\sigma_2(55)$=42	*aababbabbbabbabbbabbbbabbbabbbbabbbbbabbbbabbbbbabbb*	*baba*	56	43

Table 4.5: Strings of length n+1 with $\sigma_2(n)$+1 distinct squares for $52 \le n \le 55$.

Similarly, we also found an instance when prepending a makes the string to admit $\sigma_2(n)+1$ distinct squares. This instance is listed in Table 4.6.

Therefore, when searching for a suitable lower bound for the next n, we can take the previously found maximal string for $\sigma_2(n-1)$ and then manipulate it by either prepending or appending a or b to see whether it increases the number of distinct

	String x	$\|x\|$	$s(x)$
$\sigma_2(49)$=37	*ababbabbbbabbbbbabbbbabbbbbabbbbabbbabbbbabbbabbbb*	50	37
$\sigma_2(50)$=37	[*a*]*ababbabbbbabbbbbabbbbabbbbbabbbbabbbabbbbabbbabbbb*	51	38

Table 4.6: Strings of length n+1 with $\sigma_2(n)$+1 distinct squares for $n = 50$.

squares. This can be used as an efficient alternative method to quickly find a $(2, n)$-string with $\sigma_2(n-1)+1$ distinct primitively rooted squares, and thus serves a lower bound for $\sigma_2(n)$, in addition to the heuristic search.

4.4 Using Upper Bound to Simplify Computation

In terms of the upper bound, by Lemma 2.10 which was deduced from Fraenkel and Simpson [14], we get $\sigma_d(n) \leq \sigma_d(n-1)+2$. In this section, we show how this constraint on the upper bound helps with the computation of $\sigma_d(n)$.

4.4.1 Double Square

Consider the first unknown value $\sigma_d(n)$ in the $(d, n-d)$-table. If the lower bound we have determined by the methods discussed in Section 4.3 is large enough, i.e. if $\sigma_d^-(n) = \sigma_d(n-1)+1$, then $\sigma_d^-(n) \leq \sigma_d(n) \leq \sigma_d(n-1)+2$. Thus, to determine the true value of $\sigma_d(n)$, instead of generating the set of all s-covered strings that are dense enough to exceed $\sigma_d^-(n)$, we shall generate a much smaller set of s-covered strings dense enough to produce $\sigma_d^-(n)+1$. If we succeed, then $\sigma_d(n) = \sigma_d^-(n)+1$, and if we do not succeed, then $\sigma_d(n) = \sigma_d^-(n)$.

As we will discuss below, for a string to be able to produce $\sigma_d(n-1)+2$ distinct squares, it has to have a very specific structure and therefore the size of the set of s-cover strings we have to generate is dramatically reduced, and thus the performance of the computation is significantly improved.

Lemma 4.16. *Let $k(x)$ be the core vector of square-maximal (d, n)-string $x[1 .. n]$, if $k_1(x) \leq 1$, then $\sigma_d(n) \leq \sigma_d(n-1) + 1$.*

Proof. Since by Lemma 2.2 we can move all possible singletons of x to the end without reducing the number of distinct squares in x, without loss of generality we can assume that all singletons of x, if any, are at the end of x. Remove the first symbol in x, thereby forming a new string y. Because the first symbol is not a singleton, y is a $(d, n-1)$-string. Since $k_1(x) \leq 1$, there is at most one core of square starting at position 1, hence we destroyed at most 1 distinct square in x. Thus, $\sigma_d(n) = s(x) \leq s(y) + 1 \leq \sigma_d(n-1) + 1$. □

By Lemma 4.16, in order for $\sigma_d(n) = \sigma_d(n-1) + 2$, the square-maximal string x has to satisfy the density condition $k_1(x) > 1$ where $k(x)$ is the core vector of x. According to Fraenkel and Simpson's lemma [14], there are at most two rightmost squares starting at the same position; in other words, $k_1(x) \leq 2$, that is, position 1 can contain the core of at most two squares; therefore, $k_1(x) = 2$. When two rightmost occurring squares start at the same position, we call it a *double square*. Therefore, for the case of $\sigma_d^-(n) = \sigma_d(n-1) + 1$, we generate a set of s-covered strings where the first square is the longer square of a double square.

4.4.2 Algorithm

The combinatorial structure of a double square is given by Deza, Franek, and Thierry [13]: $u^p u_1 u^{p+q} u_1 u^q$, where u is a primitive non-empty string, u_1 is a proper prefix of u, and $1 \leq q \leq p$. The two squares of the double square $u^p u_1 u^{p+q} u_1 u^q$ are $(u^p u_1)(u^p u_1)$ – the shorter one, and $(u^p u_1 u^q)(u^p u_1 u^q)$ – the longer one. Note that $|(u^p u_1)(u^p u_1)| > |(u^p u_1 u^q)|$ since $p \geq q$. String $(ab)^2 a(ab)^3 a(ab)$ is an example of a double square, where $u = ab$, $u_1 = a$, $p = 2$ and $q = 1$; the two squares are $(ababa)(ababa)$ and

$(ababaab)(ababaab)$. For computational simplicity, we shall consider the form of a double square is $(u^p u_1 u^q)^2$.

The uniqueness of the structure of a double square is proved [13]. Therefore, to generate s-covered (d, n)-strings with the first square is a double square, the key is to generate the first s-cover square in the form of $(u^p u_1 u^q)^2$. The program generates all possible primitive strings u, all suitable p's and q's, and all possible lengths of u_1 such that $1 \leq |u_1| < |u|$, under a condition that the total length does not exceeds n. Algorithm 4.5 shows the pseudo-code. $uLen$ and u_1Len denotes the length of u and u_1, respectively. $GENERATE(u)$ represents the process of generating a primitive string u with length of $uLen$. $SET(u^p u_1 u^q)$ represents the operations of setting the generator of the first square in the form of $u^p u_1 u^q$. And $tidySquare()$ is the function we introduced in Section 4.2.1, it verifies the generator of the first square and then continue to build the next square if all the validations are passed.

Algorithm 4.5 Double square s-cover generation.

```
uLen ← 2
p, q, u_1Len ← 1
while (uLen × (p + q) + u_1Len) × 2 ≤ n do
  GENERATE (u) {generate a primitive string u of length uLen}
  while (uLen × (p + q) + u_1Len) × 2 ≤ n do
    for u_1Len = 1 → uLen − 1 do
      for q = 1 → p do
        if (uLen × (p + q) + u_1Len) × 2 ≤ n then
          SET (u^p u_1 u^q) {set first square generator}
          tidySquare() {verify first square generator and build next square}
        else
          BREAK
        end if
      end for
    end for
    p ← p + 1
  end while
  uLen ← uLen + 1
  p, q, u_1Len ← 1
end while
```

Chapter 5

Computing Periodicity

In Chapter 4 we discussed in details how to generate the set of candidate strings, i.e. singleton free, dense s-covered strings. In this chapter we focus on the underlying algorithm to compute the periodicity of a given string; that is, in our case, the number of distinct primitively rooted squares including the core vectors.

We first introduce Crochemore's partitioning repetition algorithm which is the foundation for the underlying algorithm of the computational framework for maximum number of distinct squares problem. Then, we present an algorithm, referred to as FJW, developed by Franek, Jiang, and Weng [19]. It is a space-efficient modification and implementation of Crochemore's algorithm to compute three different periodicity measurements of a string: maximal repetitions, runs, and distinct primitively rooted squares. In the end of this chapter, we will discuss how we extend FJW not only to compute the periodicity of a string, but also to return its core vector, and the vector contains the number of distinct squares of its prefixes with the length up to the indices of the vector; namely, k-vector and p-vector. These two parameters are required by the density check during the generation of the candidate strings as discussed in Section 4.2.5.

5.1 Crochemore's Repetition Algorithm

Recall from Section 1.1.3, a repetition is formed by tandem repeats. When the power of a repetition is 2, it forms a square. Crochemore's repetitions algorithm [6], often also referred to as Crochemore's partitioning algorithm, was introduced in 1981. It was the first $O(n \log n)$ algorithm to compute maximal repetitions in a string of length n and was shown to be optimal. The main idea of this approach is to successively refine the indices of the string into equivalent classes.

Definition 5.1. *For a string $x[1 \mathinner{..} n]$, we define an equivalence $\boldsymbol{\approx_p}$ on positions $1 \le i \le n$ and $1 \le j \le n$, by $i \approx_p j$ if $x[i \mathinner{..} i+p-1] = x[j \mathinner{..} j+p-1]$.*

For technical reasons, a sentinel symbol \$ is used to denote the end of the input string; it is considered to be the lexicographically smallest character. An example is shown in Figure 5.1, the indices of string x have been grouped into equivalent classes of $\approx_p$, where $p = 1, \cdots, 7$, and the classes are refined level by level from $l = 1$ to $l = 7$. Each class that has two or more elements represents a repeat of period p, where p equals to the current level of refinement l. The subscript of each class represents the repeating string of that class. For instance, class $\{3, 6\}_{aab}$ on level $l = 3$ represents a class of equivalence $\approx_3$, where the elements in the class are the starting positions of the same substring *aab*.

Starting from level 1, Crochemore's repetitions algorithm continuously refines the indices of the equivalent classes until all classes have been reduced to a singleton class (i.e. a class contains only one element). A naive approach to refine a level of classes into the next level would be taking every element from each class and compare the character on its next position of the string, and then group the elements based on the comparison. This approach would lead to an $O(n^2)$ complexity, as there are potentially $O(n)$ indices to be processed on each level and there can be potentially

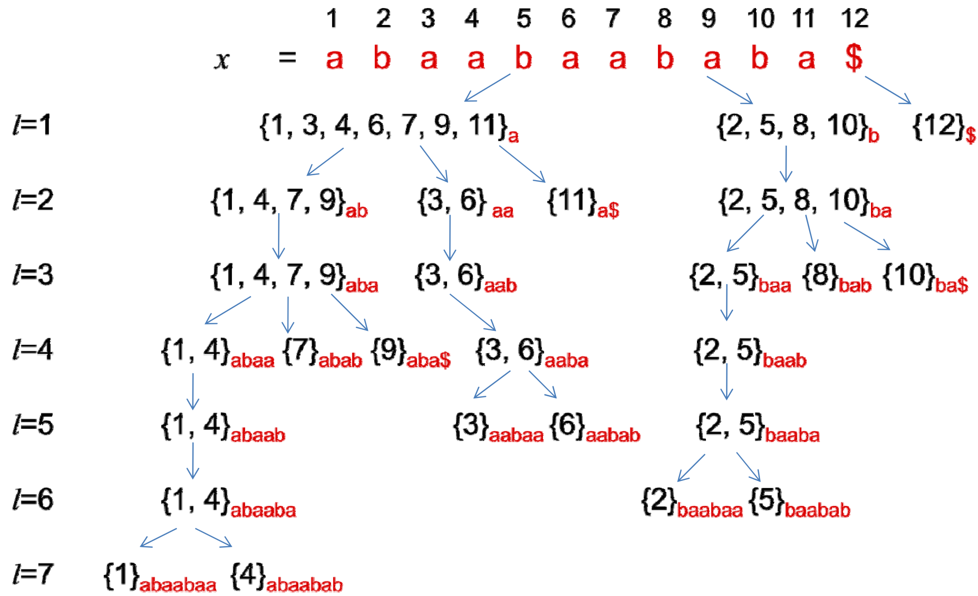

Figure 5.1: Example of Crochemore's repetition algorithm.

$O(n)$ levels. The fundamental idea of Crochemore's algorithm is that the refinement of each equivalent class at each level is not performed directly against the original string; rather, it is refined using the classes of the previous level. Note that not all the levels of classes are saved, all we need is the previous level to compute the next level. Consider a class $\mathcal{C}$ and a class $\mathcal{D}$ on level $\mathcal{L}$, in order to refine class $\mathcal{C}$ by $\mathcal{D}$ into the next level $\mathcal{L}+1$, we take $i, j \in \mathcal{C}$, if $i+1, j+1 \in \mathcal{D}$, then we move them into one class, otherwise we must separate them into different classes. For instance, let us refine class $\mathcal{C} = \{1,3,4,6,7,9,11\}$ by class $\mathcal{D} = \{2,5,8,10\}$ on level 1: 1 and 3 must be placed into different classes as 2 and 4 are not both in $\mathcal{D}$, 1 and 4 will be put in the same class, since 2 and 5 are both in $\mathcal{D}$. In fact $\mathcal{C}$ is refined into three classes, one consisting of the indices from $\mathcal{D}$ with one position shifted to the left ($\{1,4,7,9\}$), one containing the indices that were separated ($\{3,6\}$), and the last one ($\{11\}$) is actually refined using class $\{12\}$ on level 1. Computationally, the same results can be achieved by processing the elements in $\mathcal{D}$; that is, for every element $i, j \in \mathcal{D}$, if $i-1, j-1 \in \mathcal{C}$,

then $i-1, j-1$ are put into the same class. In this case, $\{2-1, 5-1, 8-1, 10-1\}$, i.e. $\{1, 4, 7, 9\}$, is refined into one class; then class $\{12\}$ is processed as $\{12-1\}$, i.e. $\{11\}$, is refined into another class; and $\{3, 6\}$ is a class with left over elements. Similarly $\mathcal{D} = \{2, 5, 8, 10\}$ can be refined using $\mathcal{C} = \{1, 3, 4, 6, 7, 9, 11\}$. If we use all the classes for refinements, we end up with the next level.

Despite using the classes of the previous level for refinement, the algorithm would still have $O(n^2)$ complexity. Thus, a major innovation of Crochemore's algorithm is to use only some of the classes of the previous level for refinements. The classes used for refinement are so-called small classes. In the next paragraph, we explain the notion of small classes.

We say classes form a "family" if they form a refinement of same class from the previous level. For instance, consider the example in Figure 5.1, classes $\{1, 4, 7, 9\}$, $\{3, 6\}$ and $\{11\}$ on level 2 form a family as they are the refinement of $\{1, 3, 4, 6, 7, 9, 11\}$ on level 1. In other words, $\{1, 3, 4, 6, 7, 9, 11\}$ is the parent class, and $\{1, 4, 7, 9\}$, $\{3, 6\}$ and $\{11\}$ are the children classes. Consider class $\{2, 5, 8, 10\}$ on level 2, in order to refine it, we would need to use all other classes on the same level, that is, $\{1, 4, 7, 9\}$, $\{3, 6\}$ and $\{11\}$. The resulting classes on level 3 are $\{2, 5\}$ which was obtained by refinement using $\{3, 6\}$, $\{8\}$ which was obtained by refinement using $\{1, 4, 7, 9\}$, and $\{10\}$ which was obtained by refinement using $\{11\}$. A simple observation can be made that $\{2, 5, 8, 10\}$ are refined into three classes of repeating strings with different suffixes that are the repeating strings of $\{1, 4, 7, 9\}$, $\{3, 6\}$ and $\{11\}$, which are from the same family. Thus, we do not have to use for refinement all three classes as we could just use two of them and the last one is automatically formed by the left-over elements. In this case, we could use $\{3, 6\}$ to get $\{2, 5\}$ and $\{11\}$ to get $\{10\}$, and $\{8\}$ is a left-over to form a class of its own. In other words, we do not have to use all the classes from of the family; in fact, we could use all the classes except one. For

computational efficiency, we should not use the class of the largest size, and use all the other classes. That is, for each family we identify the largest class which contains the most elements and call all the other classes small. By using only small classes for refinements, $O(n \log n)$ complexity can be achieved since every element in the string can occur in at most $O(\log n)$ small classes. Note that for the first level $l = 1$, every class is considered small.

When it comes to the reporting of the repetitions, every equivalent class that represents repeats of the same substring needs to be examined for the "gaps" between consecutive indices. According to the definition in Section 1.1.3, repetitions are tandem repeats. Thus, the algorithm needs to consider only those consecutive indices that have a gap of the same size as the current level l, as they then represent tandem repetitions of period l. If the gap is bigger than l, it indicates a ***split repeats***; if the gap is smaller than l, it indicates an ***overlapping repeats***. Consider the example shown in Figure 5.1, at level 3, class $\{1, 4, 7, 9\}$, the gap between index 1 and 4 is 3 which equals to the period (level) 3, and the gap between 4 and 7 is also 3; but the gap between 7 and 9 is 2 which is smaller than 3, that indicates overlapping repeats. Thus, when counting the exponent of this repetition, we should count index 1, 4, and 7, but not 9, which gives us the final output $(1, 9, 3)$ in the form of (s, e, p), represents the repetition $(aba)^3$ in the string. We will discuss the gap function in details in the following section, where the FJW algorithm, a space-efficient implementation of Crochemore's algorithm, is discussed.

5.2 FJW Algorithm

One advantage of Crochemore's repetition algorithm is its independence on the size of the alphabet of the string. Its disadvantage is in the implementation, because

the data structures required for keeping track of the refinements and the gaps demand a substantial storage and a complex machinery to update and maintain them. Originally, when Crochemore's algorithm was introduced, the estimated storage requirement for its implementation was about $20n$ of integers, where n is the length of the input string. In 2003, Franek, Smyth, and Xiao [20] implemented the algorithm using several memory saving techniques lowering the requirement to $14n$ integers. To extend its capability, Franek and Jiang [16] [18] developed a number of extensions to Franek, Smyth, and Xiao's implementation [20], to compute the runs in a string, and the performance of these extensions was benchmarked. The strategy was rather straightforward, it simply took the maximal repetitions as outputted by the origical algorithm and consolidated them into runs. Depending on the timing when and how the consolidation is performed, the extension algorithms have different storage requirements and complexity, but all needed an extra $O(n \log n)$ storage.

In 2011, Franek, Jiang, and Weng [19] re-designed Franek, Smyth, and Xiao's implementation [20], and we refer it to as FJW. This implementation not only reduces the storage requirement down to $13n$ of integers, but also it computes three periodicity measurements of the input string: namely the number of distinct primitively rooted squares, maximal repetitions, and runs. Unlike the previous implementation by Franek and Jiang [16] [18] where runs were computed through the consolidation of the maximal repetitions, the computation of the three periodicity measurements in FJW is directly carried out from the gap function, which we will discuss the details in Section 5.2.2.

5.2.1 Data Structures

We first describe a naive implementation and its data structures without any regard for the size of required memory. This leads to an implementation requiring $20n$ of integers. Then we use several techniques to reduce the memory to $13n$ integers. For practical reasons, we present the data structures as static; in fact they are all allocated on the heap, but only once at the outset of the program's processing. So, there is no memory allocation or deallocation performed during the processing of a string. Moreover, we do not want to recompile the program each time a different string is to be processed. The data structures are essentially arrays used to emulate doubly-linked lists, stacks, and queues to maintain the information about *classes*, *families*, and *gap lists*, as discussed in Section 5.1.

The first seven arrays deal with classes contained with elements:

1. An integer array ***CStart***[] stores the very first element of a class. $CStart[i] = j$ means that the first element of class i is j. *This emulates a pointer to the beginning of a class.*

2. An integer array ***CEnd***[] stores the very last element of a class. $CEnd[i] = j$ means that the last element of class i is j. *This emulates a pointer to the end of a class.*

3. An integer array ***CNext***[] stores the next element in the class or $NULL$. $CNext[i] = j$ indicates that i and j are in the same class and that j is the next element after i, while $CNext[i] = NULL$ indicates i is the last element in the class. *This emulates the forward links of a class list.*

4. An integer array ***CPrev***[] stores the previous element in the class or $NULL$. $CPrev[i] = j$ indicates that i and j are in the same class and that j is the

element just before i, while $CPrev[i] = NULL$ indicates i is the first element in the class. *This emulates the backward links of a class list.*

5. An integer array $\boldsymbol{CMember}$[] stores the class membership of each element. $CMember[i] = j$ means that element i belongs to class j.

6. An integer array $\boldsymbol{CSize}$[] stores the sizes of classes. $CSize[i] = j$ means that class i has j elements.

7. An integer array $\boldsymbol{CEmpty}$[] is used as a stack of empty classes that can be used.

The following five arrays deal with families of classes as described above:

8. An integer array $\boldsymbol{FStart}$[] is used as a stack. It stores the very first class that belongs to a family. $FStart[i] = j$ means that class j is the first class in family i. *This emulates a pointer to the beginning of a family.*

9. An integer array $\boldsymbol{FEnd}$[] stores the very last class that belongs to a family. $FEnd[i] = j$ means that class j is the last class in family i. *This emulates a pointer to the end of a family.*

10. An integer array $\boldsymbol{FNext}$[] stores the next class in the family or $NULL$. $FNext[i] = j$ indicates that i and j are in the same family and that j is the next class after i, while $FNext[i] = NULL$ indicates i is the last class in the family. *This emulates the forward links of a family list.*

11. An integer array $\boldsymbol{FPrev}$[] stores the previous class in the family or $NULL$. $FPrev[i] = j$ indicates that i and j are in the same family and that j is the class just before i, while $FPrev[i] = NULL$ indicates i is the first class in the family. *This emulates the backward links of a family list.*

12. An integer array $\boldsymbol{FMember}[]$ stores the family membership of each class. $FMember[i] = j$ means that class i belongs to family j.

The following four arrays implement the list of gaps between the consecutive elements:

13. An integer array $\boldsymbol{Gap}[]$ stores the first element of a gap list. $Gap[i] = j$ indicates that the first element in the list of gap of i is j, i.e. $j - i$ and j are consecutive elements in the class and $j - i$ is j's predecessor.

14. An integer array $\boldsymbol{GNext}[]$ stores the next element in the gap list or $NULL$. $GNext[i] = j$ indicates that i and j are in the same gap list and that j is the next element after i, while $GNext[i] = NULL$ indicates i is the last element in the gap list. *This emulates the forward links of a gap list.*

15. An integer array $\boldsymbol{GPrev}[]$ stores the previous element in the gap list or $NULL$. $GPrev[i] = j$ indicates that i and j are in the same gap list and that j is the element just before i, while $GPrev[i] = NULL$ indicates i is the first element in the gap list. *This emulates the backward links of a gap list.*

16. An integer array $\boldsymbol{GMember}[]$ stores the gap membership of each element. $GMember[i] = j$ means that element i belongs to the list of gap of j.

The last four arrays are used for the refinement process:

17. An integer array $\boldsymbol{Refine}[]$ is used to memorize the destination class of a class element after it is refined. $Refine[i] = j$ means that an element from class i should be moved to class j during the refinement.

18. An integer array $\boldsymbol{RStack}[]$ is used as a stack to memorize which positions in $Refine[]$ were occupied, so it can be cleared without any need to traverse the whole array $Refine[]$, as that would destroy the $O(n \log n)$ complexity.

19. An integer array ***Sel***[] is used as a queue containing elements from all the small classes.

20. An integer array ***Sc***[] is used as a queue to store the last element of each small class. Thus, the information in $Sel[]$ and $Sc[]$ implements a list of elements of small classes with indicators where one small class ends and the next small class starts.

The above adds up a total of 20 integer arrays. Our memory saving techniques are mainly based on a simple observation; that is, for the double-linked list structure, the previous element of the head (first element) of a list is always $NULL$; similarly, the next element of the tail (last element) of a list is also $NULL$. Therefore, we can make use of these unused spaces in the double-linked lists for the information which we would need a whole array to store. The following is the summary on how we save seven integer arrays in the FJW algorithm.

1. The last element of a class which was stored in ***CEnd***[] can be stored in the previous element of the first element of the class. In other words, function $CEnd()$ replaces integer array $CEnd[]$, and $CEnd(i) = CPrev[CStart[i]]$.

2. Similarly, array ***FEnd***[] is replaced by function $FEnd()$ and $FEnd(i) = FPrev[FStart[i]]$.

3. The size of a class which was stored in ***CSize***[] can be stored in the next element of the last element of the class. That is, function $CSize(i) = CNext[CPrev[CStart[i]]]$ as $CEnd(i) = CPrev[CStart[i]]$ in item 1.

4. ***GMember***[] can be eliminated as the gap membership of an element can be

directly calculated through function:

$$GMember(i) = \begin{cases} NULL & \text{if } i \text{ is not a member of any class} \\ NULL & \text{if } i \text{ is the first memeber of a class} \\ i - CPrev[i] & \text{otherwise} \end{cases}$$

5. $FStart[]$ is used as a stack, so the spaces after the stack pointer are unused. Similar to item 3, space for the next element of the tail of a family list is also unused. Therefore, array $\boldsymbol{FMember}[]$ is replaced it by a function $FMember()$ that utilizes these spaces:

$$FMember(i) = \begin{cases} FStart[i] & \text{if the stack pointer is } NULL \\ FStart[i] & \text{if } i > \text{the stack pointer} \\ FNext[FPrev[FStart[i]]] & \text{otherwise} \end{cases}$$

6. To replace $\boldsymbol{CMember}[]$, we neither can directly compute the class membership of an element like $GMember()$ does in item 4, nor we can utilize the unused spaces in other class arrays such as $CPrev[]$ and $CNext[]$, as they have been occupied to replace $CEnd[]$ and $CSize[]$ in item 1 and 3. Therefore, we will make use of the unused spaces in $Gap[]$ and $GNext[]$ in a way similar to function $FMember()$ in item 5. However, unlike $FStart[]$ is used as a stack and the stack pointer is an indicator such that the spaces beyond it are not used, the spaces that are occupied in array $Gap[]$ are completely arbitrary depending on the gaps between the elements in the classes. To distinguish whether the stored value represents a class membership of an element or the first element from a gap list, we store the class membership in its negative value. Therefore, we use `long`

instead of `unsigned long` as the data type for the arrays, thus allowing us to store negative values and this limits the maximal possible length of an input string to `LONG MAX`, which for a 32-bit machine equal to $2,147,483,647$, which is quite sufficient for all practical purposes.

$$CMember(i) = \begin{cases} NULL & \text{if } Gap[i] \text{ is } NULL \\ NULL & \text{if } GNext[GPrev[Gap[i]]] \text{ is } NULL \\ 0 - Gap[i] & \text{if } Gap[i] < 0 \\ 0 - GNext[GPrev[Gap[i]]] & \text{otherwise} \end{cases}$$

7. Array ***CEmpty***[] and ***Sc***[] share the same data structure. $Sc[]$ saves data from left to right of the array; and on the other hand, $CEmpty[]$ grows from right to left.

5.2.2 Gap Function

As mentioned in Section 5.1, to compute the periodicity of a string, the program has to go through the gap list and report the appropriate information as desired. In order to keep a running complexity of $O(n \log n)$, the gap lists are maintained throughout the process of refinements; that is, every time an element is removed from or added to a class, the gap list is updated according to the change. Recall $Gap[p]$ points to the first element whose immediate predecessor in its class is exactly at a distance of p, while $GNext[]$ and $GPrev[]$ allow us to traverse the whole gap list in either direction and to update the list in constant time. When computing the periodicity at level p after the refinement, we are dealing with the gap list with the first element stored in $Gap[p]$. If $Gap[p] = i$, then there is a primitively rooted square of period p starting at

position $i-p$. In the following, we will discuss how the three functions that compute the three periodicity measurements including the number of distinct squares, maximal repetitions and runs, by directly extracting information from the gap list.

Computing Distinct Squares Function ***traceSquares()*** computes the number of distinct primitively rooted squares in a string. It traverses the gap list for the current level, once it identifies the first square in each class, it ignores the identification of the rest from the same class as they represent the same type of squares. For computing the number of distinct squares, we are only interested in the types of the squares, not the occurrences. Consider the example shown in Figure 5.1, at level 3, element 4 and 7 of class $\{1,4,7,9\}_{aba}$ are in the gap list because the distance between them and their predecessors are both equal to the current level, i.e. the period. As the function goes through the gap list, square $(1,6,3)$ is identified while $(4,9,3)$ is ignored as they both represent the same type of square, namely $(aba)^2$. We use $Refine[]$ and $RStack[]$ that are only needed during the refinement process as auxiliary data structures here to keep track of which class, i.e. square type, we already have a representative from. That is, we use $Refine[i]$ to store the representative square from class i, and we use $RStack[]$ as a stack to memorize which positions in $Refine[]$ are occupied. In fact, for each type of square, we only store its right most occurrence for display purpose.

Computing Maximal Repetitions Function ***traceMaxReps()*** is used to compute the maximal repetitions of a string. Unlike computing the distinct squares, not only we need to compute all the occurrences of the repetitions, but also we need to extend the repetition to its maximum and count the exponent of the repetition. The algorithm traverses the gap list, and for each entry it checks how

far left and how far right it can extend the square. Thus, during the tracing at level p, all the individual squares identified are consolidated into maximal repetitions. A brief description of how the algorithm determines if the square can be extended to the left: the entry i from gap list $Gap[p]$ indicates that there is a primitively rooted square starting at position $i-p$ and element $i-p$ is in the same class as element i. In fact, $i-p$ is the previous element of i in the class list. Since the elements of each class are stored in the natural order, the algorithm iterates through the class list that entry i belongs to, and checks if the previous element of i is p distance away from it; if so, then the repetition is extendible to the left starting at position $i-2p$; we do this check until we fail to find such element or we reach the first element of the class. Note that if we were able to extend the repetition to the left, $i-p$ must be in the gap list as well. However, it is possible that the element $i-p$ is further away in the gap list since they are stored in arbitrary order. In order not to process element $i-p$ again as it would be a redundant work, we again use $Refine[]$ and $Rstack[]$ to indicate that this entry has already been processed. Similar check applies to the right extendibility of the repetition.

Computing Runs The computation of runs in a string is performed by function ***TraceRuns()***. The idea is very similar to the procedure of tracing maximal repetitions: the identified primitively rooted squares are consolidated into runs during the traversal of the gap list. We refer to the first square occurs in a run as the *leading square*. Consider the leading square of a run (s, e, p) must be primitively rooted by definition; and at every position $s+i$, where $0 \leq i \leq (q-2)p+t$ where $q = (e-s+1)/p$ (integer division) and $t = (e-s+1)\%p$ (modulus), there is a primitively rooted square. This fact is based on a simple

observation that a rotation of a primitive string is also primitive. The algorithm has to consolidate the runs from all of the primitively rooted squares encoded in the gap list. Thus, having identified a square, not only must we check if it can be extended to the left or right as a repetition, but also we have to check if the repetition can be shifted to the left or right, that is, for every position we move to its left or right, whether there exists a primitively rooted repetition with the same period. We perform this check until the repetition can not be shifted anymore. Again, we use $Refine[]$ and $RStack[]$ as auxiliary data structures to indicate which element of the gap list had been previously processed so that we do not process it again.

5.3 Extend FJW to Produce k-Vector and p-Vector

In Section 5.2 we discussed the FJW in details, a program we developed to compute three types of periodicity measurements in a given string. One of the measurements is the number of distinct primitively rooted squares, which forms the foundation of the underlying program for the computational framework for computing the maximum number of distinct primitively rooted squares in strings. Recall that in order to compute the maximum number of distinct squares in strings of a given length and a given number of distinct symbols, we would have to generate a set of singleton free, dense, s-covered strings. When we generate these candidate strings, not only we are interested in their respective number of distinct squares; but more importantly, we need to be able to identify as early as possible whether they are dense enough. By Lemma 4.12, the density check is done throughout the process of the generation of the s-cover squares; that is, for each square we build, we check if the partial string we have built so far is dense enough, if it is, we continue with the generation of the

next square, otherwise the current square is discarded. Therefore, the earlier we can eliminate the non-dense strings, the more computational efficiency we can gain.

By Definition 4.6, in order to determine if a string is dense or not, we would have to check if its core vector $k(x)$ satisfies $k_i(x) > \sigma_d^-(n) - s(x[1 \mathinner{..} i-1]) - m_i$ for every position $1 \leq i \leq n$. How a suitable lower bound $\sigma_d^-(n)$ is obtained is discussed in Section 4.3. Each m_i is the maximum number of distinct squares that a $(d', n-i)$-string y could contain, where $|\mathcal{A}(y)| = d'$ and $|\mathcal{A}(y) \cup \mathcal{A}(x[1 \mathinner{..} i-1])| = d$, which can be obtained from the previously computed values in the $(d, n-d)$-table. Thus, we need to compute the core vector $k(x)$, let's call it ***k-vector***; and a vector of $s(x[1 \mathinner{..} i-1])$, the number of distinct primitively rooted squares in the prefix of the string up to position $i-1$, for every $1 \leq i-1 \leq n$, let's call it ***p-vector***. Instead of developing a new algorithm to compute these two vectors, these information can be directly extracted from the gap list in the FJW algorithm as we compute the distinct squares. In this section we discuss the modified version of the FJW algorithm that not only computes the number of distinct squares, but also returns the k-vector and p-vector of the string.

5.3.1 k-Vector

By Definition 4.5, the k-vector, i.e. core vector, of a string records the number of cores of squares that each position contains. By Definition 4.4, the core of a square type is a set of indices that are the intersection of the indices of all its occurrences. Note that the non-empty core of a square must be either a single index, where the intersection is one position; or is a set of continuous indices, where the intersection are the indices of a portion of the string. Thus, we can use the starting index and ending index of the set of intersection indices to denote a core, as every index in between is

also in the core. Therefore, to populate the k-vector, we would have to find the core of each square type, and then within the range of its core, we increment the k-vector accordingly. This process can be done as we go through the gap list to identify the squares of the string.

When we traverse the gap list at each level, every entry in the gap list for that level identifies a square. As discussed before, when computing the distinct squares, we would only take one representative from each class, i.e. square type; however, to compute the core of a square, we would have to consider all the occurrences of the square. Therefore, the idea is for every square identified in the gap list, we check if it is a new square type, i.e. from a new class; if it is, then we store the starting index and ending index of its core to be the same as the square itself; if not, then it means we already have a core for this square type, we would have to update the core information for this square accordingly. Let us consider string $x[1 \mathinner{..} n]$, as we go through the gap list at level p, we find a square type s at $x[i_1 \mathinner{..} i_2]$. If s is a new type, then the core of s is $\{i_1, i_1+1, \cdots, i_2\}$. For demonstration purpose, let us use a substring of x to represent the position of core of s, denoted as $c_s = x[i_1 \mathinner{..} i_2]$, where every index from i_1 to i_2, inclusively, is in the core of s. If s is not new, then it is another occurrence for square type s which we have found previously in the gap list, this indicates the core for previously found occurrences of square type s must exist, let it be $c_s = x[i_3 \mathinner{..} i_4]$; depending on how the position of s to the position of c_s, we update the starting index and ending index of c_s accordingly. Once c_s is updated for every occurrence of square type s found in the gap list, c_s represents the core of square type s for all its occurrences in x. There are four different scenarios described as follows. We refer $s = x[i_1 \mathinner{..} i_2]$ as the new occurrence of square type s since it is already been found before.

1. **$s = x[i_1 .. i_2]$ and $c_s = x[i_3 .. i_4]$ are not overlapped, or c_s is empty.** In this case the intersection of the two is empty, therefore the core is a empty set, and thus there is no incrementation for the k-vector for this square type. Figure 5.2 shows this scenario.

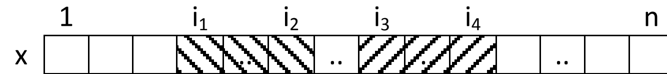

Figure 5.2: Core computation: core and new occurrence are not intersected.

2. **$s = x[i_1 .. i_2]$ and $c_s = x[i_3 .. i_4]$ are overlapped, and s is on the left of c_s in x.** In this case, the ending index of the core is updated to i_2 while the starting index is unchanged. Thus, $c_s = x[i_3 .. i_2]$. As shown in Figure 5.3, indices of the overlapping portion is the new core of square type s.

Figure 5.3: Core computation: new occurrence is on the left.

3. **$s = x[i_1 .. i_2]$ and $c_s = x[i_3 .. i_4]$ are overlapped, and c_s is on the left of s in x.** Similar to above case, the starting index of the core is updated to i_1 while the ending index is unchanged. Thus, $c_s = x[i_1 .. i_4]$, as shown in Figure 5.4.

Figure 5.4: Core computation: core is on the left.

4. **$c_s = x[i_3 .. i_4]$ is contained in $s = x[i_1 .. i_2]$.** In this case, both starting index and ending index of the core remain unchanged since the overlapping

portion is equivalent to the core as shown in Figure 5.5, that is $c_s = x[i_3 \mathinner{..} i_4]$.

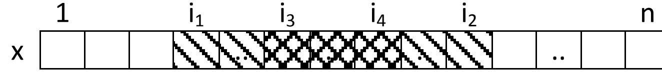

Figure 5.5: Core computation: core is contained in new occurrence.

Note that it is not possible for $s = x[i_1 \mathinner{..} i_2]$ to be contained in $c_s = x[i_3 \mathinner{..} i_4]$. This is because the core of a square is a set of indices that are the intersection indices of its all occurrences, and therefore the distance between its starting index and ending index, is always equal to or smaller than the length of the square.

Once we process the entire gap list for the level, we have obtained the core information for every type of squares. Then we shall increment the k-vector on the positions according to the range of each core. That is, for every core, from its starting index to its ending index, inclusively, the corresponding positions in k-vector are incremented by one.

5.3.2 p-Vector

Before we go into any computational details, let us formally define the p-vector.

Definition 5.2. *For a string x with length of n, the* ***p-vector*** $p(x) = [p_1(x), p_2(x), \cdots,$ $p_n(x)]$ *of x is defined by $p_i(x) =$ the number of distinct primitively rooted squares contained in $x[1 \mathinner{..} i]$ for $1 \le i \le n$.*

Figure 5.6 shows the p-vector of an example string x. Note that every position in the p-vector contains the number of distinct primitively rooted squares in the prefix of x up to that position. That is, for every $1 \le i \le n$, $p_i(x) = s(x[1 \mathinner{..} i])$.

Instead of computing the p-vector by brute force, that is, take every possible prefix of the string and compute the number of distinct squares of it; we could directly extract

	1	2	3	4	5	6	7	8	9	10
x	a	a	b	a	b	a	b	b	a	a
p	0	1	1	1	2	3	3	4	4	4

Figure 5.6: p-Vector of string x.

this information as we go through the gap list to identify different types of squares. The approach is based on a simple observation, consider any position j in $x[1 .. n]$, for any distinct square that appears on the left and up to j should be counted into the value of $p_i(x)$ where $j \leq i \leq n$. This is because any prefix of x with longer length contains the prefixes of x with shorter length, and thus contains all their distinct squares. Therefore, as we identify each square $s = x[i_1 .. i_2]$ in the gap list, every prefix of x that contains s should take it into account; that is, for every $i_2 \leq i \leq n$, $x[1 .. i]$ contains s, $p_i(x)$ should be incremented by one. If the square has more than one occurrences in the string, the prefixes that contain some or all occurrences of the square should count it only once since we are interested in the type of the squares, not the occurrences. Therefore, only the leftmost occurrence of the square should be considered when calculating the p-vector as any occurrences appear after the leftmost one will be ignored when counting the number of distinct squares in the prefixes of the string.

The algorithm traverses through the gap list, and every entry in the list indicates a square of the string. If the square is a new type, i.e. from a new class, then we set the ending position of its leftmost occurrence (let us refer it as leftmost ending position for short), to be the ending position of the square. Otherwise it is not new, thus we have found it in the gap list before, then there must exist a leftmost ending position for this square type; therefore, we shall compare the leftmost ending position with the ending position of the square we just found and update it accordingly, i.e.

set it to the ending position of the square if it is bigger. Once we finish processing the entire gap list for the level, we have obtained the ending position of the leftmost occurrence for every square type. For each square type, we increment the values in the p-vector from its leftmost ending position to the end of the vector. Figure 5.7 demonstrates this process for each square type, the positions in the p-vector that are incremented are indicated in gray.

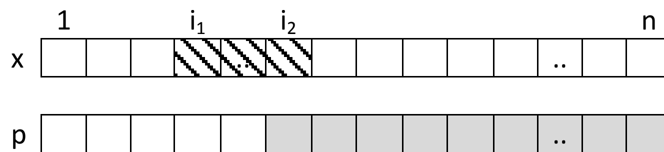

Figure 5.7: p-Vector computation: for every $s = x[i_1 .. i_2]$ is the leftmost occurrence, $p_i(x) = p_i(x) + 1$, for $i_2 \leq i \leq n$.

5.3.3 Data Structures

As discussed in Section 5.2, the required data storage for the FJW algorithm is $13n$ of integers where n is the length of the string. For the modified version of the FJW algorithm we described in this section, the required data storage is increased to $15n$ of integers, where the two extra integer arrays are used to store the starting indices and ending indices of the cores of squares when the algorithm computes the k-vector. Array $Refine[]$ is used to store the ending positions of the leftmost occurrences of squares to produce the p-vector, and it is also used as an indicator to memorize the square type that has been processed similar to the original FJW algorithm, and array $RStack[]$ is again used as a stack to keep track of which positions of the above three arrays are occupied so that they can be cleared out in constant time after a level of computation is done.

Chapter 6

Computational Results

In Chapters 4 and 5 we discussed the computational framework for distinct primitively rooted squares in strings in details; including the generation of singleton free, dense s-covered strings in Chapter 4, and the underlying modified FJW algorithm to compute the distinct squares for each generated string in Chapter 5. In this chapter we focus on the computational results obtained in our implementation of the framework. In Section 6.1 we present the computation environment for the framework and the $(d, n-d)$-table in a larger scale in comparison to the one presented in Table 2.1, where a number of interesting properties are observed and discussed. Section 6.2 shows the current known bounds for $\sigma_d(n)$ deduced from the values we have obtained so far.

6.1 Values in $(d, n-d)$-Table

The complete computational framework including both the s-covered string generation module and the modified FJW module are implemented in C++ programming language for its flexibility and efficiency. Most of the computations were performed on a workstation contains 8 *AMD Dual-Core Opteron*^{TM}*885* processors with frequency of

2.6 GHz, 64 GB of RAM and 500 GB disks. In Table 2.1 we showed a $(d, n-d)$-table with small values of d and n, which was computed by brute force. After implementing our enhanced computational framework, not only we were able to verify the values we computed by brute force in a timely manner, but also we were able to expand the table into a much larger scale. We were able to compute all $\sigma_2(n)$ values for $n \leq 70$. The 10 largest new values are: $\sigma_2(61) = 47$, $\sigma_2(62) = 48$, $\sigma_2(63) = 48$, $\sigma_2(64) = 49$, $\sigma_2(65) = 50$, $\sigma_2(66) = 51$, $\sigma_2(67) = 52$, $\sigma_2(68) = 53$, $\sigma_2(69) = 54$ and $\sigma_2(70) = 55$. Due to the limited space, only a portion of the $(d, n-d)$-table is shown in Table A.1 of the Appendix. The complete $(d, n-d)$-table with the currently known values is actively maintained and can be found in [10]. Some values in the $(d, n-d)$-table [10] are provided with a list of sample square-maximal strings for the corresponding d and n that were generated by the computation framework. From the values we currently obtained, the following two interesting properties can be observed from the $(d, n-d)$-table.

6.1.1 Three Consecutive Equal Values

Throughout the values we have computed so far, the number of consecutive equal values along a row is two in most cases except one instance of three consecutive equal values along a row; that is, $\sigma_2(31) = \sigma_2(32) = \sigma_2(33)$, which are highlighted in light gray in Table A.1. These three consecutive equal values lead to some previously unobserved behaviour of the values in the table. One is that the existence of an increase of 2 between two consecutive values on a descending diagonal; that is, $\sigma_{d+1}(n+2) - \sigma_d(n) > 1$ when $\sigma_d(n) = \sigma_2(33), \sigma_2(34)$, whereas in all other instances in the table the increase is 1. The second one is an instance of strings with more symbols of the same length producing more distinct squares. This is quite counter-intuitive,

because with the same length, the more distinct symbols we have, the less likely we can repeat the symbols to produce a square; e.g. in the extreme case when a string contains the same number of symbols as its length, no square can be formed. This unusual case is $\sigma_2(33) < \sigma_3(33)$; that is, among all strings of length 33, no binary string achieves the maximum number of distinct primitively rooted squares.

6.1.2 Increasing on Descending Diagonals

By Property 2.6 (c) we presented in Chapter 2, there is a strict increase between the two values along a descending diagonal from top-left to bottom-right direction. As mentioned in Section 6.1.1, almost every instance in our $(d, n-d)$-table shown in Table A.1, the increase of the two consecutive values along the descending diagonals is 1 except the two instances having increase of 2 which is caused by the three consecutive equal values. Table A.2 of the Appendix is a $(d, n-2d)$-table where the column index is $n-2d$ instead of $n-d$ as in the $(d, n-d)$-table. When our values are presented in the $(d, n-2d)$-table, the descending diagonals of the $(d, n-d)$-table become the columns of the $(d, n-2d)$-table. As we can clearly see from Table A.2, the entries on every column form an arithmetic sequence with the consecutive term equal to 1, i.e. every entry in a column is incremented by 1 at each time from top-to-bottom direction. The only two exceptions as we mentioned are the increment of 2; that is $\sigma_3(35) - \sigma_2(33) = 2$ and $\sigma_3(36) - \sigma_2(34) = 2$. These four values are highlighted in light gray. The complete $(d, n-2d)$-table can be also found in [10].

6.2 Current Bound for $\sigma_d(n)$

Fraenkel and Simpson [14] gave the upper bound of $2n-8$ for $n \geq 5$ and any d, and $\sigma_2(n) \leq 2n-29$ for $n \geq 22$. Ilie [22] provided an asymptomatic bound of $2n-\Theta(\log n)$.

In 2013, Deza, Franek and Thierry [13] showed that a string of length n contains at most $\frac{11n}{6}$ distinct squares.

Remark 6.1. *For any* $2 \leq d \leq n - d_0$, $\sigma_d(n) \leq 2n - 2d - d_0$, *where* d_0 *is the largest* d *such that* $\sigma_d(2d) = d$.

Proof. By Theorem 2.14 and Lemma 2.10, $\sigma_d(n) \leq d_0 + 2k$, where $n - d = d_0 + k$ and $k \geq 0$. Thus, $\sigma_d(n) = \sigma_d(d_0 + k + d) \leq d_0 + 2k = 2(d_0 + k + d) - 2d - d_0$. Therefore, $\sigma_d(n) \leq 2n - 2d - d_0$ for $2 \leq d \leq n - d_0$. □

Remark 6.1 can be intuitively observed from the $(d, n-d)$-table: for each step we move from left to right along a row, the maximum increase of values is 2 by Lemma 2.10, and there are total of $n - d$ steps; and we have shown the increase of each step in the first d_0 columns of the $(d, n-d)$-table is at most 1, thus yields a bound of $\sigma_d(n) \leq 2n - 2d - d_0$. As of writing this thesis, $d_0 = 24$.

In addition, from the currently computed values, the bound specifically for each $2 \leq d \leq 10$ are listed as follows:

$$\sigma_2(70) = 55, \text{thus } \sigma_2(n) \leq 2n - 85 \text{ for } n \geq 70;$$

$$\sigma_3(45) = 34, \text{thus } \sigma_3(n) \leq 2n - 56 \text{ for } n \geq 45;$$

$$\sigma_4(38) = 27, \text{thus } \sigma_4(n) \leq 2n - 49 \text{ for } n \geq 38;$$

$$\sigma_5(37) = 26, \text{thus } \sigma_5(n) \leq 2n - 48 \text{ for } n \geq 37;$$

$$\sigma_6(35) = 24, \text{thus } \sigma_6(n) \leq 2n - 46 \text{ for } n \geq 35;$$

$$\sigma_7(37) = 25, \text{thus } \sigma_7(n) \leq 2n - 49 \text{ for } n \geq 37;$$

$$\sigma_8(29) = 18, \text{thus } \sigma_8(n) \leq 2n - 40 \text{ for } n \geq 29;$$

$$\sigma_9(31) = 19, \text{thus } \sigma_9(n) \leq 2n - 43 \text{ for } n \geq 31;$$

$$\sigma_{10}(33) = 20, \text{thus } \sigma_{10}(n) \leq 2n - 46 \text{ for } n \geq 33.$$

Chapter 7

Conclusion

The previous chapters entail our investigation of the maximum number of distinct primitively rooted squares problem of both combinatorial and computational approach. Chapter 2 introduces the d-step approach to the problem, where $\sigma_d(n)$ is presented in a $(d, n-d)$-table; a number of properties of $\sigma_d(n)$ and then the $(d, n-d)$-table are discussed and possible directions to solving the conjecture on the size of the upper bound are presented. Chapter 3 focuses on the structural aspects of the square-maximal $(d, 2d)$-strings under both conditions, complying with the conjectured upper bound, and not complying. In the latter case, the first square-maximal $(d, 2d)$-strings with $\sigma_d(2d) > d$ if there are any, are discussed. Chapters 4 and 5 describe the computational framework for computing the values of $\sigma_d(n)$ efficiently by significantly reducing the search space based on the notion of singleton free, dense s-covered strings and utilizing a pre-determined lower bound. Finally, Chapter 6 presents the main results of the computational experiments and a few interesting observations of the values obtained are discussed.

In this chapter, we first briefly examine the relation of the maximum number of distinct primitively rooted squares problem to the problem of maximum number of

runs as it was investigated by Baker, Deza and Franek, see Baker's Ph.D. thesis [1], using a similar approach to the one presented in this thesis. The direction of possible future work to improve either the theoretical or the computational results are pointed and discussed in the end of this chapter.

7.1 Relation to $\rho_d(n)$

As we mentioned in Section 1.1.4, the notion of run was conceptuality introduced by Main [30], and the term itself was later coined by Iliopoulos, Moore, and Smyth [23]. Kolpakov and Kucherov [26] showed that the number of runs in a string is $O(n)$ and conjectured that maximum number of runs in a string should no more than its length. Let $\rho(n)$ denote the maximum number of runs over all strings of length n. Several authors have presented asymptotic bounds on $\rho(n)$; Crochemore, Ilie and Tinta presented the currently best asymptotic upper bound in [7], and Matsubara et al. presented the currently best asymptotic lower bound in [31].

Similar to our approach to the problem of maximum number of distinct squares, besides the length of the string, the number of distinct symbols in the string was considered as a parameter when the problem of runs was investigated by Baker, Deza, and Franek [1] [3] [4] [5] [9]. Let $\rho_d(n)$ be the maximum number of runs over all strings of length n with exactly d distinct symbols. A run-maximal string refers to a string containing the maximum number of runs. In the next few sections, we will discuss briefly on how their findings on $\rho_d(n)$ relate to our investigation on $\sigma_d(n)$.

7.1.1 Similarities and Differences

Similarly to our approach, the values of $\rho_d(n)$ were presented in a $(d, n-d)$-table with row index d and column index $n-d$, and $\rho_d(n)$ is conjectured to be no more than

$n-d$. The basic properties of the $(d, n-d)$-table for $\sigma_d(n)$ as discussed in Proposition 2.6 also apply to the $(d, n-d)$-table for $\rho_d(n)$, such as the values are non-decreasing from left to right along a row; the values are non-decreasing from top to down along a column; the values are strictly increasing along a descending diagonal; and the values on and under the main diagonal along a column are constant [9]. In addition, some other properties such as the lower bound for the entries on and under the main diagonal, see Lemma 2.7, the lower bound for the two immediate entries on the right of the main diagonal entries, see Lemma 2.8, and the upper bound for the difference of the main diagonal entry and the entry immediately above it, see Lemma 2.9, are also proved for $\rho_d(n)$ [4] [9]. A number of equivalent statements to the conjectured upper bound, which correspond to Theorems 2.14, 2.15, 2.16, and 2.17, are shown [9]. Let us remark that though the properties are similar, the proofs of the corresponding lemmas are rather different.

While the functions $\sigma_d(n)$ and $\rho_d(d)$ exhibit many similarities, there are still quite a few differences concerning both the methodologies and the properties due to the different nature of the problems. For example, the removal of singletons does not reduce the number of distinct squares in a string, see Lemma 2.2, but may cause a merge of two runs into one and thus a reduction of the number of runs. On the other hand, when concatenating two strings into one, the runs occurring in both strings are counted, while it is not true for distinct squares since only the types of squares are considered.

While Lemma 2.12 shows that the two consecutive entries immediately above the main diagonal entry along a column are identical, the three consecutive identical entries above the main diagonal entry are proved in the $(d, n-d)$-table for $\rho_d(n)$ [4].

Based on Fraenkel and Simpson's Lemma [14], we were able to deduce an upper bound of 2 for a difference between any two consecutive values along a row or along

the main diagonal, and along the two descending diagonals $(d, 2d+1)$-diagonal and $(d, 2d+2)$-diagonal (as we defined in Section 2.4) that are immediately above the main diagonal in the $(d, n-d)$-table for $\sigma_d(n)$, see Lemmas 2.10 and 2.11, and Corollary 2.13, which are not true for $\rho_d(n)$. On the other hand, the function of $\rho_d(n)$ has the property of $\rho_d(n+2) > \rho_d(n)$ [9]; that is, for any three consecutive entries along a row in the $(d, n-d)$-table of $\rho_d(n)$, the third entry is always strictly bigger than the first entry, which eliminates the possibility of three identical consecutive entries along a row as it occurs in the $(d, n-d)$-table for $\sigma_d(n)$, in particular $\sigma_2(31) = \sigma_2(32) = \sigma_2(33)$.

In terms of the structure of the run-maximal strings, there is an unique run-maximal string $aabbcc\cdots$ on the main diagonal if $\rho_d(2d) = \rho_d(2d+1)$ (correspond to Lemma 3.1); or the first counterexample on the main diagonal, if there is any, does not contain a symbol occurring exactly 2, 3, $\cdots$, 7, or 8 times, which yields the fact that such a counter-example string would contain at least $\lceil \frac{7d}{8} \rceil$ singletons [4]. In our investigation of the structure of a possible first counterexample on the main diagonal turned out to be more complicated, as shown in Lemmas 3.3, 3.4, 3.5, and 3.6, we were able to eliminate only the pairs and some forms of triples to prove the existence of at least $\lceil \frac{2d}{3} \rceil$ singletons, obtaining a weaker Theorem 3.10.

Baker, Deza, and Franek [5] described a computational framework for computing $\rho_d(n)$. Similar to the s-cover as discussed in Chapter 4, they adopted a r-cover structure and notion of dense strings in respect to a pre-determined lower bound $\rho_d^-(n)$ to reduce the search space for run-maximal strings. However, as we mentioned above, Lemma 2.10 specifies the property that the difference of any two consecutive values along a row is bound by 2 is not applicable for the $(d, n-d)$-table of $\rho_d(n)$, thus the structure of double square entailed in Section 4.4.1 that further improves the computational efficiency could not be used to compute $\rho_d(n)$. The most up to date values of $\rho_d(n)$ are listed in [2].

7.1.2 Differences on Values

Consider string $x = abbab$, there is 1 distinct square and 1 run bb in x; for string $y = ababa$, there are 2 distinct squares $abab$ and $baba$, and there is only 1 run $ababa$; for string $z = aabaa$, there is 1 distinct square aa, but there are 2 runs with aa occurring twice in z. Thus, we have exhibited three scenarios such that for strings with the same length and the number of symbols, some have the same number of distinct squares and runs, and some have more distinct squares than runs, some have more runs than distinct squares. Therefore, it is not easy to discern any relationship between the values of $\sigma_d(n)$ and $\rho_d(n)$.

Based on experimental results, Kolpakov and Kucherov [27] suggested the maximum number of distinct squares is smaller than or equal than the maximum number of runs for a given length. Our current computational results on $\sigma_d(n)$ and the $\rho_d(n)$ values obtained from [2] support this suggestion; that is, for every currently known value, $\sigma_d(n) \leq \rho_d(n)$. Table B.3 in the Appendix shows a fragment of the $(d, n-d)$-table with entries of $\rho_d(n) - \sigma_d(n)$, and the values on the main diagonal of the table are shown in gray. As we expected, the values on and under the main diagonal values are 0, this is because $\sigma_d(2d) = \rho_d(2d) = d$ for all currently known d's and the values under the main diagonal values are constant and equal to the main diagonal value on the same column for both $\sigma_d(n)$ and $\rho_d(n)$ tables. Consider every descending diagonal above the main diagonal, $(d, 2d+i)$-diagonal for $i \geq 1$; that is, the diagonal contains the entries of $\rho_d(2d+1) - \sigma_d(2d+1)$, the diagonal contains the entries of $\rho_d(2d+2) - \sigma_d(2d+2)$, etc., the values are either zero or greater than zero, and remain constant along each diagonal except few exceptions which are caused by the *singularity*. For the purpose of illustration, let us define singularity as a (d, n) pair such that $\sigma_{d+1}(n+2) - \sigma_d(n) \geq 2$, or $\rho_{d+1}(n+2) - \rho_d(n) \geq 2$, respectively. As discussed

in Section 6.1.2, there are two instances where there exist an increase of 2 on the descending diagonal; that is $\sigma_3(35) - \sigma_2(33) = 2$ and $\sigma_3(36) - \sigma_2(34) = 2$, therefore $(2, 33)$ and $(2, 34)$ are singularities. For $\rho_d(n)$, there are three singularities that have been found so far: $(2, 13)$, $(2, 41)$, and $(3, 42)$. Note that $(3, 42)$ is inferred by the fact that $\rho_4(44) \geq \rho_3(43) = 35$ and $\rho_3(42) = 33$, thus $\rho_4(44) - \rho_3(42) \geq 2$. The values that are not constant along the descending diagonals are highlighted in light gray in Table B.3. These values correspond to the singularities we listed above except $(2, 41)$ and $(3, 42)$ due to the limited space. Table B.4 lists the values of $\rho_d(n) - \sigma_d(n)$ in a $(d, n - 2d)$-table, where the descending diagonals of the $(d, n - d)$-table become the columns and each column are constant except the singularities shown in light gray.

The complete $(d, n - d)$-table and $(d, n - 2d)$-table for $\rho_d(n) - \sigma_d(n)$ with the up to date values can be found in [10].

7.1.3 Strings Achieving Both Square- and Run-Maximality

In Section 7.1.2 we examined the differences of $\rho_d(n)$ and $\sigma_d(n)$. We provided examples of strings with the same, more, or fewer distinct squares than runs. One may ask if there exists a square-maximal string that is also a run-maximal string; in other words, whether there exists a maximal string achieving both square- and run-maximality. It turned out that there are such strings. The results for $2 \leq d \leq 10$ and $2 \leq n - d \leq 10$ are listed in Table 7.1, where 1 represents the existence of such a string for the corresponding d and n, and 0 represents that no such string exists.

It is expected that the square-maximal strings and run-maximal strings for entries on and under the main diagonal have the duality, since both $\sigma_d(n)$ and $\rho_d(n)$ hold the property that the values on and under the main diagonal values are constant, and the maximal $(d, 2d)$-strings are unique in the form of $aabbcc\cdots$ for the currently known

d's. Let us consider the existence of such strings occurring on the descending diagonals which are highlighted in Table 7.1, precisely when $n = 2d+i$ where $i = 1, 2, 3, 5$. Table 7.2 is a fragment of the $(d, n-d)$-table for $\rho_d(n) - \sigma_d(n)$, which was highlighted on the same descending diagonals to shows that the dual maximal strings could either have the same number of distinct squares and runs (where the entries are 0), or have a different number of distinct squares and runs (where the entries are 1).

d \ $n-d$	2	3	4	5	6	7	8	9	10
2	**1**	1	1	1	0	1	0	0	0
3	1	**1**	1	1	1	0	1	0	0
4	1	1	**1**	1	1	1	0	1	0
5	1	1	1	**1**	1	1	1	0	1
6	1	1	1	1	**1**	1	1	1	0
7	1	1	1	1	1	**1**	1	1	1
8	1	1	1	1	1	1	**1**	1	1
9	1	1	1	1	1	1	1	**1**	1
10	1	1	1	1	1	1	1	1	**1**

Table 7.1: Existence of strings achieving both square- and run-maximality.

d \ $n-d$	2	3	4	5	6	7	8	9	10
2	**0**	0	0	1	1	0	0	0	1
3	0	**0**	0	0	1	1	0	0	0
4	0	0	**0**	0	0	1	1	0	0
5	0	0	0	**0**	0	0	1	1	0
6	0	0	0	0	**0**	0	0	1	1
7	0	0	0	0	0	**0**	0	0	1
8	0	0	0	0	0	0	**0**	0	0
9	0	0	0	0	0	0	0	**0**	0
10	0	0	0	0	0	0	0	0	**0**

Table 7.2: $(d, n-d)$-Table of $\rho_d(n) - \sigma_d(n)$ for $2 \le d \le 10$ and $2 \le n-d \le 10$.

Table 7.3 lists the strings that are both square-maximal and run-maximal for $d = 2$, and $n = 5, 6, 7, 9$. The distinct squares and runs are listed for each string. Most of them are equal for distinct squares and runs except a few instances where the strings contained runs with tails, such as run *aaa* for $(2, 5)$-string *aaabb* and *aabaaba* for $(2, 9)$-string *aabaababb*. In addition, the second occurrence of *aa* for $(2, 7)$-string was counted for runs, but not for distinct squares since each type of square is only considered once. The complete list of dual strings for square- and run-maximality for $2 \le d \le 10$ and $2 \le n-d \le 10$ can be found in [10].

d	n	Maximal String	$\sigma_d(n)$	$\rho_d(n)$	Distinct Squares/Runs
2	5	*aaabb*	2	2	$\{aa/aaa,\ bb\}$
2	5	*aabab*	2	2	$\{aa,\ abab\}$
2	6	*aababb*	3	3	$\{aa,\ abab,\ bb\}$
2	7	*aabaabb*	3	4	$\{aa,\ aabaab,\ bb,\ \varepsilon/aa\}$
2	9	*aabaababb*	5	5	$\{aa,\ aabaab/aabaaba,\ abaaba,\ abab,\ bb\}$
2	9	*aababbaba*	5	5	$\{aa,\ babbab,\ abab,\ baba,\ bb\}$

Table 7.3: Dual square/run-maximal strings for $d = 2$, $n = 5, 6, 7, 9$.

7.2 Future Work

Deza and Franek [9] hypothesized an upper bound for $\sigma_d(n)$ based on currently known values. To refine the hypothesized upper bound and to examine the properties of $\sigma_d(n)$ function, more computational results are needed, thus the computational efficiency is deemed to be improved.

Recall the underlying algorithm of the computational framework for computing the number of distinct squares is a modified version of FJW (Chapter 5), which is an implementation based on Crochemore's repetition algorithm. The advantage of Crochemore's algorithm is that it relies on successive refinements of equivalence classes, a process that can be naturally parallelized as the refinement of one class is independent from the refinements of other classes. Franek and Jiang [17] proposed a set of methodologies to parallelize the extended Crochemore's algorithm, which was developed by the same authors for computing runs [16] [18], under the shared memory model. In comparison with the extension algorithm, FJW is a much improved algorithm and with functionality extended to compute runs and distinct squares in addition to maximal repetitions, nevertheless they are both based on the Crochemore's partitioning algorithm, thus the core of them are essentially the same except a few implementation details. Therefore, for the future work it is important to design and implement a parallel version of FJW. This will not only help use the computational

framework of this thesis to extend the $(d, n-d)$-table to previously intractable values, but would also be valuable as a standalone program.

In addition, the s-covered strings generation module of the computational framework also can be parallelized. According to Baker [1], the generation of r-covers for large values were carried by seeding the serial program with different periods and prefixes of the first square, and then run the set of serial programs simultaneously. For our s-covered strings generation, instead of manually enumerate all possible periods and prefixes, we can automate this process in a parallel program so that the master processor automatically assign the computation load to each idle processors based on a pre-defined work load divider. Note that besides the shared memory model, the parallelization of both the s-cover generation and the underlying FJW can be implemented under a distributed memory parallel model. When designing algorithm under these two models, different issues should be considered. The shared memory model poses the problems of locking/unlocking the shared data structures to prevent corruption of the data, while requiring very little overhead for communication. The distributed memory model, on the other hand, does not have to worry about accidental corruption of the data, however the communication overhead may be significant.

When increasing the length of a string and number of distinct symbols, the number of possible strings increases exponentially. Thus, one effective approach to improve the efficiency of the computational framework is to further reduce the number of strings it has to generate; that is, to reduce the size of the search space for square-maximal strings. In Section 4.4.1 we introduced a special structure referred to as double square that can be applied for the computation of $\sigma_d(n)$, where $\sigma_d^-(n) = \sigma_d(n-1)+1$. This fixed-form structure dramatically reduces the search space for the square-maximal strings of $\sigma_d(n)$. On the other hand, there are no special properties that can be applied to $\sigma_d(n)$ in general when $\sigma_d^-(n) = \sigma_d(n-1)$, thus all the dense s-covered

strings have to be generated. To illustrate the difference, the computation time for $\sigma_2(58) = 44$ was over 58 hours since $\sigma_2^-(58) = \sigma_2(57) = 44$; and the computation time for $\sigma_2(59) = 45$ was under 7 minutes since the heuristic search found a lower bound of $\sigma_2^-(59) = \sigma_2(58) + 1 = 45$. In other words, whenever there is a potential tie along a row in the $(d, n-d)$-table, the computation time increases significantly. Therefore, we hope to find more structural insight into the case of ties and thus to conquer the bottleneck of the computation.

As discussed in Section 4.3.3, the square-maximal strings for some entries exhibit an interesting property of having common prefixes; that is, by appending one extra character to a square-maximal string, it forms a string for the next length with one more square and thus becoming a suitable lower bound for the next length (Table 4.4, 4.5). This observation indicates to us that certain amount of computation in generating the s-covered strings could be redundant as the length increases since their square-maximal strings have similar structure. Therefore, we hope to design and develop a mechanism to recursively store the previously generated s-covers including their density information, and when the length increases, only the necessary extension to the stored s-covers are generated and stored for the next length. This way, tremendous amount of work could be saved in the process of s-covered strings generation.

While we have to make efforts to improve the efficiency of the computational framewrok to obtain more values of $\sigma_d(n)$, we should also continue the investigation of the structure of the square-maximal strings as discussed in Chapter 3. To further examine the structure of the first counterexample, if there is one, we could either confirm that $\sigma_d(n) \leq n - d$ for larger values of d and n by estimating the singletons; or to prove the uniqueness of the maximal strings on the main diagonal.

Appendix A

Tables of $\sigma_d(n)$

A.1 $(d, n-d)$-Table

A.2 $(d, n-2d)$-Table

d \ $n-d$	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35
2	2	2	3	3	4	5	6	7	7	8	9	10	11	12	12	13	13	14	15	16	17	18	19	20	20	21	22	23	23	23	24	25	26	27
3	2	3	3	4	4	5	6	7	8	8	9	10	11	12	13	13	14	14	15	16	17	18	19	20	21	21	22	23	24	24	25	26	26	27
4	2	3	4	4	5	5	6	7	8	9	9	10	11	12	13	14	14	15	15	16	17	18	19	20	21	22	22	23	24	25	25	26	27	
5	2	3	4	5	5	6	6	7	8	9	10	10	11	12	13	14	15	15	16	16	17	18	19	20	21	22	23	23	24	25	26			
6	2	3	4	5	6	6	7	7	8	9	10	11	11	12	13	14	15	16	16	17	17	18	19	20	21	22	23	24						
7	2	3	4	5	6	7	7	8	8	9	10	11	12	12	13	14	15	16	17	17	18	18	19	20	21	22	23	24	25					
8	2	3	4	5	6	7	8	8	9	9	10	11	12	13	13	14	15	16	17	18														
9	2	3	4	5	6	7	8	9	9	10	10	11	12	13	14	14	15	16	17	18	19													
10	2	3	4	5	6	7	8	9	10	10	11	11	12	13	14	15	15	16	17	18	19	20												
11	2	3	4	5	6	7	8	9	10	11	11	12	12	13	14	15	16	16	17	18	19	20	21											
12	2	3	4	5	6	7	8	9	10	11	12	12	13	13	14	15	16	17																
13	2	3	4	5	6	7	8	9	10	11	12	13	13	14	14	15	16	17	18															
14	2	3	4	5	6	7	8	9	10	11	12	13	14	14	15	15	16	17	18	19														
15	2	3	4	5	6	7	8	9	10	11	12	13	14	15	15	16	16	17	18	19	20													
16	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	16	17	17	18	19														
17	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	17	18	18															
18	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	18	19															
19	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	19	20														
20	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	20														
21	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21														
22	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22													
23	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23												
24	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24											

Table A.1: $(d, n-d)$-Table with larger d and n.

	$n-2d$																																	
d	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33
2	2	2	3	3	4	5	6	7	7	8	9	10	11	12	12	13	13	14	15	16	17	18	19	20	20	21	22	23	23	23	24	25	26	27
3	3	3	4	4	5	6	7	8	8	9	10	11	12	13	13	14	14	15	16	17	18	19	20	21	21	22	23	24	24	25	26	26	27	28
4	4	4	5	5	6	7	8	9	9	10	11	12	13	14	14	15	15	16	17	18	19	20	21	22	22	23	24	25	25	26	27			
5	5	5	6	6	7	8	9	10	10	11	12	13	14	15	15	16	16	17	18	19	20	21	22	23	23	24	25	26						
6	6	6	7	7	8	9	10	11	11	12	13	14	15	16	16	17	17	18	19	20	21	22	23	24										
7	7	7	8	8	9	10	11	12	12	13	14	15	16	17	17	18	18	19	20	21	22	23	24	25										
8	8	8	9	9	10	11	12	13	13	14	15	16	17	18																				
9	9	9	10	10	11	12	13	14	14	15	16	17	18	19																				
10	10	10	11	11	12	13	14	15	15	16	17	18	19	20																				
11	11	11	12	12	13	14	15	16	16	17	18	19	20	21																				
12	12	12	13	13	14	15	16	17																										
13	13	13	14	14	15	16	17	18																										
14	14	14	15	15	16	17	18	19																										
15	15	15	16	16	17	18	19	20																										
16	16	16	17	17	18	19																												
17	17	17	18	18																														
18	18	18	19																															
19	19	19	20																															
20	20	20																																
21	21																																	
22	22																																	
23	23																																	
24	24																																	

Table A.2: $(d, n-2d)$-Table with larger d and n.

Appendix B

Tables of $\rho_d(n) - \sigma_d(n)$

B.1 $(d, n-d)$-Table

B.2 $(d, n-2d)$-Table

d \ $n-d$	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35
2	0	0	0	1	1	0	0	0	1	0	1	0	0	0	1	1	2	1	1	1	1	1	1	1	2	2	2	2	3	4	3	3	3	3
3	0	0	0	0	1	1	0	0	0	1	1	1	0	0	0	1	1	2	1	1	1	1	1	1	1	2	2	2	2	3	3	2	3	3
4	0	0	0	0	0	1	1	0	0	0	1	1	1	0	0	0	1	1	2	1	1	1	1	1	1	1	2	2	2	2	3	3	2	
5	0	0	0	0	0	0	1	1	0	0	0	1	1	1	0	0	0	1	1	2	1	1	1	1	1	1	1	2						
6	0	0	0	0	0	0	0	1	1	0	0	0	1	1	1	0	0	0	1	1	2	1	1	1										
7	0	0	0	0	0	0	0	0	1	1	0	0	0	1	1	1	0	0	0	1	1	2												
8	0	0	0	0	0	0	0	0	0	1	1	0	0	0	1	1	1	0	0	0														
9	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	1	1	1	0	0														
10	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	1	1																
11	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0																			
12	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1																			
13	0	0	0	0	0	0	0	0	0	0	0	0	0	0																				
14	0	0	0	0	0	0	0	0	0	0	0	0	0	0																				

Table B.3: $(d, n-d)$-Table of $\rho_d(n) - \sigma_d(n)$.

	$n-2d$																																	
d	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33
2	0	0	0	1	1	0	0	0	1	0	1	0	0	0	1	1	2	1	1	1	1	1	1	1	2	2	2	2	3	4	3	3	3	3
3	0	0	0	1	1	0	0	0	1	1	1	0	0	0	1	1	2	1	1	1	1	1	1	1	2	2	2	2	3	3	2	3	3	3
4	0	0	0	1	1	0	0	0	1	1	1	0	0	0	1	1	2	1	1	1	1	1	1	1	2	2	2	2	3	3	2			
5	0	0	0	1	1	0	0	0	1	1	1	0	0	0	1	1	2	1	1	1	1	1	1	1	2									
6	0	0	0	1	1	0	0	0	1	1	1	0	0	0	1	1	2	1	1	1														
7	0	0	0	1	1	0	0	0	1	1	1	0	0	0	1	1	2																	
8	0	0	0	1	1	0	0	0	1	1	1	0	0	0																				
9	0	0	0	1	1	0	0	0	1	1	1	0	0																					
10	0	0	0	1	1	0	0	0	1	1																								
11	0	0	0	1	1	0																												
12	0	0	0	1	1																													
13	0	0	0																															
14	0	0																																

Table B.4: $(d, n-2d)$-Table of $\rho_d(n) - \sigma_d(n)$.

Bibliography

[1] A. Baker. Computational and structural approaches to periodicities in strings. Ph.D. Thesis, Department of Computing and Software, McMaster University, Ontario, Canada, December 2012. `http://optlab.mcmaster.ca//component/option,com_docman/task,doc_details/gid,174/Itemid,92/`.

[2] A. Baker, A. Deza, and F. Franek. Run-maximal strings. `http://optlab.mcmaster.ca/~bakerar2/research/runmax/index.html`.

[3] A. Baker, A. Deza, and F. Franek. On the structure of run-maximal strings. *Journal of Discrete Algorithms*, 10:10–14, 2012.

[4] A. Baker, A. Deza, and F. Franek. A parameterized formulation for the maximum number of runs problem. In J. Holub, B. W. Watson, and J. Žd'árek, editors, *Festschrift for Bořivoj Melichar*, pages 102–117. Czech Technical University, Prague, Czech Republic, 2012.

[5] A. Baker, A. Deza, and F. Franek. A computational framework for determining run-maximal strings. *Journal of Discrete Algorithms*, 20:43–50, 2013.

[6] M. Crochemore. An optimal algorithm for computing the repetitions in a word. *Information Processing Letters*, 12(5):244–250, 1981.

[7] M. Crochemore, L. Ilie, and L. Tinta. The "runs" conjecture. http://www.csd.uwo.ca/faculty/ilie/runs.html.

[8] G. B. Dantzig. *Linear programming and extensions*. Princeton University Press, 1998.

[9] A. Deza and F. Franek. A *d*-step approach to the maximum number of distinct squares and runs in strings. *Discrete Applied Mathematics*, 163:268–274, 2014.

[10] A. Deza, F. Franek, and M. Jiang. Square-maximal strings. http://optlab.mcmaster.ca/~jiangm5/research/square.html.

[11] A. Deza, F. Franek, and M. Jiang. A *d*-step approach for distinct squares in strings. In R. Giancarlo and G. Manzini, editors, *Proceedings of the 22nd Annual Symposium on Combinatorial Pattern Matching - CPM 2011*, volume 6661 of *Lecture Notes in Computer Science*, pages 77–89. Springer, 2011.

[12] A. Deza, F. Franek, and M. Jiang. A computational framework for determining square-maximal strings. In J. Holub and J. Žd'árek, editors, *Proceedings of the Prague Stringology Conference 2012*, pages 111–119. Czech Technical University, Prague, Czech Republic, 2012.

[13] A. Deza, F. Franek, and A. Thierry. How many double squares can a string contain? AdvOL-Report 2013/1, Department of Computing and Software, McMaster University, Ontario, Canada, 2013.

[14] A. S. Fraenkel and J. Simpson. How many squares can a string contain? *Journal of Combinatorial Theory, Series A*, 82(1):112–120, 1998.

[15] F. Franek and J. Holub. A different proof of Crochemore-Ilie lemma concerning microruns. In J. Chan, J. W. Daykin, and M. S. Rahman, editors, *London*

Algorithmics 2008: Theory and Practice, pages 1–9. King's College, London, United Kingdom, 2009.

[16] F. Franek and M. Jiang. Crochemore's repetitions algorithm revisited - computing runs. In J. Holub and J. Žd'árek, editors, *Proceedings of the Prague Stringology Conference 2009*, pages 214–224. Czech Technical University, Prague, Czech Republic, 2009.

[17] F. Franek and M. Jiang. A parallel approach to computing runs in a string. AdvOL-Report 2010/5, Department of Computing and Software, McMaster University, Ontario, Canada, 2010.

[18] F. Franek and M. Jiang. Crochemore's repetitions algorithm revisited: Computing runs. *International Journal of Foundations of Computer Science*, 23(2):389–401, 2012.

[19] F. Franek, M. Jiang, and C.-C. Weng. An improved version of the runs algorithm based on Crochemore's partitioning algorithm. In J. Holub and J. Žd'árek, editors, *Stringology*, pages 98–105. Czech Technical University, Prague, Czech Republic, 2011.

[20] F. Franek, W. F. Smyth, and X. Xiao. A note on Crochemore's repetitions algorithm - a fast space-efficient approach. *Nordic Journal of Computing*, 10(1):21–28, 2003.

[21] L. Ilie. A simple proof that a word of length n has at most $2n$ distinct squares. *Journal of Combinatorial Theory, Series A*, 112(1):163–164, 2005.

[22] L. Ilie. A note on the number of squares in a word. *Theoretical Computer Science*, 380(3):373–376, 2007.

[23] C. S. Iliopoulos, D. Moore, and W. F. Smyth. A characterization of the squares in a fibonacci string. *Theoretical Computer Science*, 172(1-2):281–291, 1997.

[24] V. Klee and D. W. Walkup. The d-step conjecture for polyhedra of dimension $d < 6$. *Acta Mathematica*, 117(1):53–78, 1967.

[25] D. E. Knuth, J. H. Morris(Jr.), and V. R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323–350, 1977.

[26] R. M. Kolpakov and G. Kucherov. Finding maximal repetitions in a word in linear time. In *Proceedings of the 1999 Symposium on Foundations of Computer Science*, pages 596–604. IEEE Computer Society, 1999.

[27] R. M. Kolpakov and G. Kucherov. On maximal repetitions in words. In G. Ciobanu and G. Păun, editors, *Proceedings of the 12th International Symposium on Fundamentals of Computation Theory*, volume 1684 of *Lecture Notes in Computer Science*, pages 374–385. Springer, 1999.

[28] M. Kubica, J. Radoszewski, W. Rytter, and T. Waleń. On the maximum number of cubic subwords in a word. *European Journal of Combinatorics*, 34(1):27–37, 2013.

[29] M. J. J. Liu. Combinatorial optimization approaches to discrete problems. Ph.D. Thesis, Department of Computing and Software, McMaster University, Ontario, Canada, September 2013. `http://optlab.mcmaster.ca//component/option,com_docman/task,doc_details/gid,178/Itemid,92/`.

[30] M. G. Main. Detecting leftmost maximal periodicities. *Discrete Applied Mathematics*, 25(1-2):145–153, 1989.

[31] W. Matsubara, K. Kusano, A. Ishino, H. Bannai, and A. Shinohara. Lower bounds for the maximum number of runs in a string. http://www.shino.ecei.tohoku.ac.jp/runs/.

[32] F. Santos. A counterexample to the Hirsch conjecture. *Annals of Mathematics*, 176(1):383–412, 2012.

[33] A. Thue. Über unendliche Zeichenreihen. *Norske Vid. Selsk. Skr., I. Mat.-Nat. Kl., Christiania*, 7:1–22, 1906.