COMBINATORIAL OPTIMIZATION
APPROACHES TO DISCRETE PROBLEMS

COMBINATORIAL OPTIMIZATION APPROACHES TO DISCRETE PROBLEMS

By

MIN JING LIU, M.A.Sc, B.ENG.

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

Doctor of Philosophy (2013)   McMaster University
(Computing and Software)   Hamilton, Ontario, Canada

TITLE:	Combinatorial Optimization Approaches to Discrete Problems
AUTHOR:	MIN JING LIU M.A.Sc, (Computational Engineering) McMaster University, Hamilton, Canada B.ENG., (Electrical Engineering) McMaster University, Hamilton, Canada
SUPERVISOR:	Dr. Antoine Deza, Dr. Frantisek Franek
NUMBER OF PAGES:	xiii, 86

To my parents

Abstract

As stressed by the Society for Industrial and Applied Mathematics (SIAM): Applied mathematics, in partnership with computational science, is essential in solving many real-world problems. Combinatorial optimization focuses on problems arising from discrete structures such as graphs and polyhedra. This thesis deals with extremal graphs and strings and focuses on two problems: the Erdős' problem on multiplicities of complete subgraphs and the maximum number of distinct squares in a string.

The first part of the thesis deals with strengthening the bounds for the minimum proportion of monochromatic t cliques and t cocliques for all 2-colourings of the edges of the complete graph on n vertices. Denote by $k_t(G)$ the number of cliques of order t in a graph G. Let $k_t(n) = \min\{k_t(G) + k_t(\overline{G})\}$ where $\overline{G}$ denotes the complement of G of order n. Let $c_t(n) = k_t(n)/\binom{n}{t}$ and c_t be the limit of $c_t(n)$ for n going to infinity. A 1962 conjecture of Erdős stating that $c_t = 2^{1-\binom{t}{2}}$ was disproved by Thomason in 1989 for all $t \geq 4$. Tighter counterexamples have been constructed by Jagger, Šťovíček and Thomason in 1996, by Thomason for $t \leq 6$ in 1997, and by Franek for $t = 6$ in 2002. We present a computational framework to investigate tighter upper bounds for small t yielding the following improved upper bounds for $t = 6, 7$ and 8: $c_6 \leq 0.7445 \times 2^{1-\binom{6}{2}}$, $c_7 \leq 0.6869 \times 2^{1-\binom{7}{2}}$, and $c_8 \leq 0.7002 \times 2^{1-\binom{8}{2}}$. The constructions are based on a large but highly regular variant of Cayley graphs for which the number of cliques and cocliques can be expressed in closed form. Considering the quantity $e_t = 2^{\binom{t}{2}-1}c_t$, the

new upper bound of 0.687 for e_7 is the first bound for any e_t smaller than the lower bound of 0.695 for e_4 due to Giraud in 1979.

The second part of the thesis deals with extremal periodicities in strings: we consider the problem of the maximum number of distinct squares in a string. The importance of considering as key variables both the length n and the size d of the alphabet is stressed. Let (d,n)-string denote a string of length n with exactly d distinct symbols. We investigate the function $\sigma_d(n) = \max\ \{s(x)|x \text{ is a } (d,n)\text{-string}\}$ where $s(x)$ denotes the number of distinct primitively rooted squares in a (d,n)-string x. We discuss a computational framework for computing $\sigma_d(n)$ based on the notion of density and exploiting the tightness of the available lower bound. The obtained computational results substantiate the hypothesized upper bound of $n-d$ for $\sigma_d(n)$. The structural similarities with the approach used for investigating the Hirsch bound for the diameter of a polytope of dimension d having n facets is underlined. For example, the role played by $(d,2d)$-polytope was presented in 1967 by Klee and Walkup who showed the equivalency between the Hirsch conjecture and the d-step conjecture.

Acknowledgements

I would like to thank my supervisors, Antoine Deza and Frantisek Franek, who provided invaluable support and encouragement during my PhD studies. My special thanks go to the members of the examination committee :Dr. A. Rosa, Dr. R. Janicki, Dr. F. Hoppe and Dr. D. Froncek.

Thanks also to my colleagues who assisted me in my work and were excellent moral support.

Finally, I would like to thank my parents for their support and encouragement.

Contents

List of Tables

List of Figures

List of Abbreviations and Symbols

- $x_1 \triangle x_2$: symmetric difference between x_1 and x_2.
- $x_1 \cup x_2$: union of the sets x_1 and x_2, in case x_1 and x_2 are overlapping strings, this represents the join of the strings (see page 40 for the full definition).
- $x_1 \cap x_2$: intersection of the set x_1 and x_2.
- $x \subseteq y$: the set x is a subset of the set y, in case x and y are strings, this means that x is a substring of y.
- $x \subset y$: the set x is a proper subset of the set y.
- $|x|$: the size (cardinality) of the set x, or, if x is a string, the length of the string.
- $A \setminus B$: relative complement of B in A; that is, the set of all elements of A that are not elements of B.
- $S_d(n)$: set of all strings of length n with exactly d distinct symbols.
- $s(x)$: number of distinct primitively rooted squares in a string x.
- $\sigma_d(n)$: maximum number of distinct primitively rooted squares over all strings of length n with exactly d distinct symbols; that is, $\sigma_d(n) = max\{s(x) \mid x \in S_d(n)\}$.
- $\mathcal{A}(x)$: the alphabet of the string x, i.e. the set of all symbols occurring in x.

- a *singleton*, respectively *pair*, *triple*, or k-*tuple* in a string x refers to a symbol occurring exactly once, respectively twice, three times, or k times, in x.
- $x[i]$ for a string x referrers to the i-th symbol of the string x, in this work we index strings starting from 0.
- .. represents the range operator, thus $i..j$ represents all values for i inclusively to j inclusively.
- xy for strings x and y denotes the *concatenation* of the two strings.

Chapter 1

Preliminaries

1.1 Graph

A *directed* graph is denoted $D = (V, A)$ with V the set of its vertices and A the set of its ordered pair of vertices. The pairs are called *arcs*, *directed edges* or *arrows*. For example, the arc $a = (x, y) \in A$, a is directed from x to y. We also can say that y is adjacent to x. A graph is *undirected* if none of its edges have an orientation, and is denoted $G = (V, E)$ with V the set of vertices of G and E its edges, i.e, if there is an edge between vertex x and y, then $(x, y) \in E$. See Figure 1.1 for an illustration of a directed (left) and an undirected (right) graph.

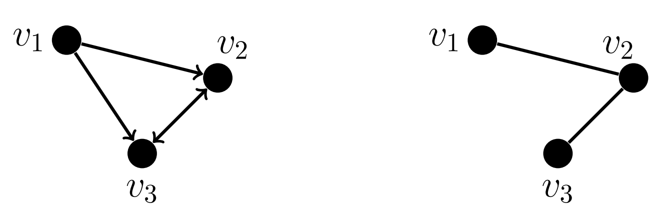

Figure 1.1: A directed graph and an undirected one

A *loop* is an edge (directed or undirected) which starts and ends on the same vertex.

A multiple-edge occurs if there exists more than one edge between two vertices. If a graph contains loops or multiple-edges it is called a *multi-graph*. An undirected graph without loop or multiple-edge is called *simple graph*. See Figure 1.2 for an illustration of a simple graph (left) and multi-graph (right).

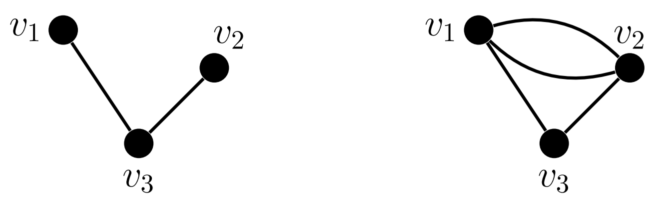

Figure 1.2: A simple graph and a multi-graph

A *complete* graph is a simple undirected graph such that any pair of distinct vertices is connected by an edge. The complete graph on n vertices is denoted as K_n. Thus K_1 is just a single vertex and K_2 is an edge.

Figure 1.3 shows the graph for K_3 and K_4.

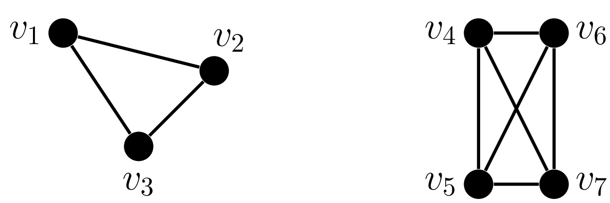

Figure 1.3: An illustration of the complete graphs K_3 and K_4

Given a graph $G = (V, E)$, its complement $\bar{G}$ is the graph with the same vertices as G but its edges correspond exactly to pairs of vertices non-adjacent in G. See Figure 1.4 for an illustration of a simple graph G and its complement $\bar{G}$.

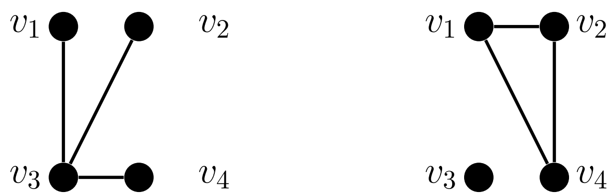

Figure 1.4: A graph G and its complement $\bar{G}$

A *clique* of order t of an undirected graph $G = (V, E)$ is a subgraph of G with t vertices forming a complete graph in G. A co-clique of order t of a graph G is a clique of order t of the complement of G. For example, there are five cliques of order 3 and two co-cliques of order 3 in the graph G shown in Figure 1.5: $\{v_2, v_3, v_5\}$, $\{v_2, v_3, v_4\}$,$\{v_2, v_4, v_5\}$,$\{v_3, v_4, v_5\}$ and $\{v_3, v_4, v_6\}$; and $\{v_1, v_2, v_6\}$ and $\{v_1, v_5, v_6\}$.

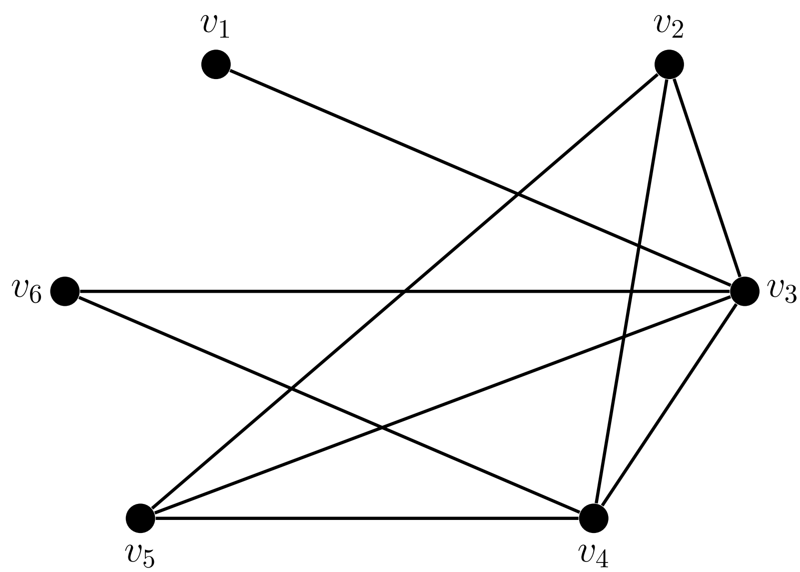

Figure 1.5: A graph G with five cliques of order 3 and two co-cliques of order 3

A graph G is *bipartite* if its vertices can be split into two parts V_1 and V_2 such that any edge in G connects a vertex of V_1 and a vertex of V_2. For example, see Figure 1.6 for an illustration of a bipartite graph G whose vertices can be split into two parts

$V_1 = (v_1, v_5, v_6)$ and $V_2 = (v_2, v_3, v_4)$ such that any edge of G crosses the dashed line separating V_1 and V_2.

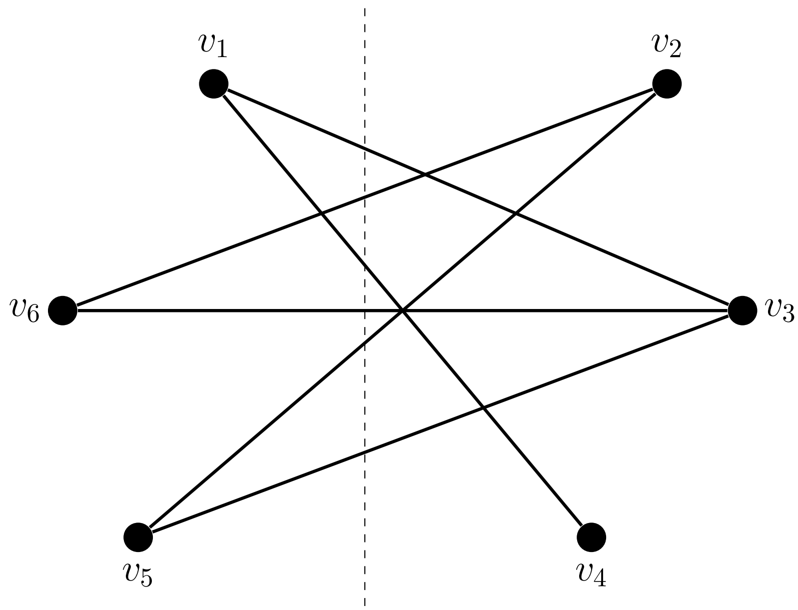

Figure 1.6: A bipartite graph G

1.2 Strings

A *string* x over an alphabet $\mathcal{A}$ is a contiguous sequence of symbols drawn from the non-empty finite set $\mathcal{A}$. $A(x)$ represents the sets of symbols occurring in x, and so $\mathcal{A}(x) \subseteq \mathcal{A}$. Often, strings are referred to as words, in particular in the discipline of *Combinatorics on words*. Both terms can be used interchangeably. We use indexing to refer to the symbols of a string and use the array notation for that, i.e. for a string x of length n, that is a string having n symbols, $x[0]$ refers to the very first symbol of x, $x[1]$ to the second symbol of x, ..., $x[n-1]$ to the very last symbol of x. We could easily index the symbols of a string starting with 1, but since the programs we used in our research are written in `C++` where strings are represented as character arrays and their indexing starts from 0, we use the same convention throughout the thesis. Thus, $x[0], x[1], ..., x[n-1]$ are the symbols of the string x of length n. If need be, we can use the range symbol .. as in $x[0..n-1]$ indicating that the index ranges from 0 to $n-1$. The notation $x = x[0..n-1]$ is used to indicate that x is of length n. A *substring* (or a *subword*) is a contiguous subsequence of a string. For example, *aabda* is a substring of the string *bbaabda*. In the range notation, $x[i..j]$ is a substring of $x = x[0..n-1]$ if $0 \leq i \leq j < n$.

The basic operations with strings is *concatenation*, i.e. joining of the two sequences. We denote the concatenation by simply listing the strings in the order to be concatenated. For instance, xy refers to a sequence $x[0], x[1], ..., x[n-1], y[0], ...,$ $y[m-1]$ where $x = x[0..n-1]$ and $y = y[0..m-1]$. A string y is said to be a *prefix* of a string x if there exist a string k such that $x = yk$. If k is non-empty, we speak of a *proper prefix*. If x is non-empty, we may call the prefix *non-trivial*. Similarly, a string y is said to be a *suffix* of a string x if there exist a string k such that $x = ky$. If k is non-empty, we speak of a *proper* suffix, and a non-empty suffix may be referred to as a *non-trivial* suffix.

A concatenation of the same string, say uu, is often abbreviated as u^2, uuu as u^3 etc. A string that is not of a form u^p for any string u and any integer $p \geq 2$ is called *primitive* and has a similar role among strings as the prime numbers have among numbers. A primitive string is a string that is not a self-concatenation of some other string. For example, $aaaa = a^4$ is not primitive while ab is primitive ($a \neq b$).

As indicated by the simplicity of the definition of a string, strings are very simple mathematical objects, we can say basic objects with no structure. Therefore, the periodicity of strings is important for investigation of properties of strings, and strings with high periodicities are of an interest to both mathematicians and computer scientists. In its generality, periodicity refers to all kinds of repeats and repetitions in strings. The most fundamental repetition is a *square*, or a string of the form uu. For example, $aabbaabb$ is a square where $aabb$ repeat twice. u of a square uu is referred to as the *generator* of the square, and the length of the generator, i.e. $|u|$ is referred to as the *period* of the square. A square uu is *primitively rooted* if its generator u is primitive. For example, the square $aabaab$ is a primitively rooted while the square $abababab$ is not.

There are very natural questions to ask: how many squares in a string can occur and how many different types of squares a string can have. The second problem is referred to in the literature as the *problem of the maximum number of distinct squares*. Since the number of distinct non-primitively rooted squares is bounded by $\lfloor \frac{n}{2} \rfloor - 1$, Kubica et al, [21], it is worthwhile to investigate the number of distinct primitively rooted squares in a string. To attack the problem of distinct squares computationally is not an easy task. For instance, to find $\sigma(n)$, the maximum number of distinct squares over all strings of length n, one would have to generate n^{n-1} strings by using the brute force search, for each compute the number of distinct squares, and find the maximum of these values. For binary strings, it is just not feasible to use the brute force approach beyond the length of approximately 32; the exact cutoff point depends on the hardware platform and the operating system used.

In the second part of the thesis, we describe a computational framework we developed to compute the maximum number of distinct primitively rooted squares for previously infeasible sizes, more than doubling the length that can be handled.

Part I

Erdős' conjecture

Chapter 2

Introduction

2.1 Erdős' Conjecture and earlier results

Denote by $k_t(G)$ the number of cliques of order t in a graph G having n vertices. Let $k_t(n) = \min\{k_t(G) + k_t(\overline{G})\}$ where $\overline{G}$ denotes the complement of G. The cliques in $\overline{G}$ are referred to as co-cliques. We use t-cliques and t-co-cliques when we want to be specific about their sizes. Let $c_t(n) = k_t(n)/\binom{n}{t}$ and $c_t = \lim_{n\to\infty} c_t(n)$. Viewing G and $\overline{G}$ as a 2-colouring of the edges of the complete graph K_n, $c_t(n)$ can be interpreted as the minimum proportion of monochromatic t-cliques over all 2-colourings of the edges of K_n [7].

A conjecture of Erdős related to Ramsey's Theorem [7], states that $c_t = 2^{1-\binom{t}{2}}$. The conjecture is true for $t = 2$. In 1959, Goodman [16] shows this conjecture holds for $t = 3$. Franek and Rödl [13] showed that the original conjecture for $t = 4$ is true for nearly quasirandom, and hence quasirandom graphs in 1992. Erdős and Moon [8] showed in 1964 that a modified conjecture for complete bipartite subgraphs of bipartite graphs is true. Sidorenko [22] showed that a modified conjecture for cycles is true. In 1989, Thomason [23] disproved the conjecture for $t \geq 4$ using an infinite sequences of graphs based on a single underlying seed graph. Namely, Thomason obtained the following results:

(a) $c_4 \leq 0.976 \times 2^{1-\binom{4}{2}} = 0.976 \times 2^{-5}$,

(b) $c_5 \leq 0.906 \times 2^{1-\binom{5}{2}} = 0.906 \times 2^{-9}$,

(c) $c_t \leq 0.936 \times 2^{1-\binom{t}{2}}$ for $t \geq 6$.

The underlying seed graphs used by Thomason are rather abstract and complicated and one could try to use simpler seed graphs to produce potential counterexamples to the Erdős' Conjecture. Thomason [24] further improved in 1997 the upper bounds for $t = 4$ and 5 to $c_4 \leq 0.9693 \times 2^{1-\binom{4}{2}}$ and $c_5 \leq 0.8801 \times 2^{-9}$. Franek and Rödl [14] presented computer generated counterexamples obtaining the same upper bounds for small t. The bounds were further improved to $c_6 \leq 0.7446 \times 2^{1-\binom{6}{2}}$ by Franek [10], and $c_t \leq 0.835 \times 2^{1-\binom{t}{2}}$ for $t \geq 7$ by Jagger, Šťovíček, and Thomason [19].

Concerning the lower bound, see Conlon [2] for an improvement over Erdős' original application of Ramsey's Theorem, and Giraud [15] who showed that $c_4 \geq 0.695 \times 2^{1-\binom{4}{2}}$.

2.2 New results

The construction used in [10] for $t = 6$ is based on the approach used by Franek and Rödl [14] who tied the best upper bound for c_4. We investigated a computational framework to search for tighter upper bounds for small t. In particular, we verified that the construction used in [10] for $t = 6$ also improves the previously known best upper bound for $t = 7$ to $c_7 \leq 0.7156 \times 2^{1-\binom{7}{2}}$. Further investigations combined with exhaustive search yields new bounds for bound for $t = 6, 7$ and 8 which are published in [6]. Note that the best upper bound for c_8 was obtained without a neighbouring search for potentially tighter constructions. The computational framework includes parallelization and use of heuristic searches. The following table indicates the results achieved by our underlying seed graphs. In particular, new bounds are in bold font.

i	Previous best bounds	Our bounds
4	$0.9693 \times 2^{1-\binom{4}{2}}$	$0.97650 \times 2^{1-\binom{4}{2}}$
5	$0.8801 \times 2^{1-\binom{5}{2}}$	$0.88584 \times 2^{1-\binom{5}{2}}$
6	$0.7446 \times 2^{1-\binom{6}{2}}$	$\mathbf{0.74444} \times 2^{1-\binom{6}{2}}$
7	$0.835 \times 2^{1-\binom{7}{2}}$	$\mathbf{0.68690} \times 2^{1-\binom{7}{2}}$
8	$0.835 \times 2^{1-\binom{8}{2}}$	$\mathbf{0.70014} \times 2^{1-\binom{8}{2}}$

Table 2.1: Results for the new graphs introduced

Chapter 3

Constructing Counterexamples

We pursue the approach used in [10, 14] to improve the upper bound for c_t for small t. In particular, we consider graphs for which the number of cliques and co-cliques can be expressed in a closed form as it allows a search for the ones exhibiting the lowest numbers of cliques and co-cliques.

3.1 Seed graphs

We consider the following family of seed graphs where $\triangle$ denotes the symmetric difference.

Definition 3.1 *For a set X and $F \subseteq \{1, 2, \ldots, |X|\}$, consider the graph $G_{X,F}$ whose vertices correspond to all $2^{|X|}$ subsets of $\{0, 1, \ldots, |X|-1\}$ and two distinct subsets x_i and x_j of $\{0, 1, \ldots, |X|-1\}$ are connected by an edge in $G_{X,F}$ if and only if $|x_i \triangle x_j| \in F$.*

See Figure 3.1 for an illustration of the graph $G_{X,F}$ with $|X| = 3$ and $F = \{2\}$. Note that $\overline{G}_{X,F} = G_{X,\overline{F}}$ where $\overline{F} = \{1, 2, \ldots, |X|\} \setminus F$.

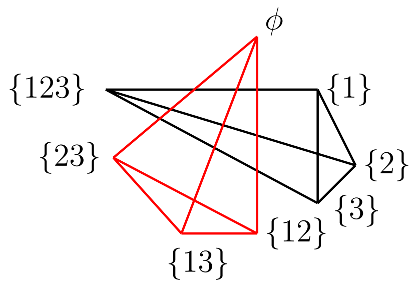

Figure 3.1: The graph $G_{X,F}$ with $|X| = 3$ and $F = \{2\}$

As Thomason [23] and Franek and Rödl [10, 14], we use the seed graph $G_{X,F}$ to produce an infinite sequence of graphs.

Definition 3.2 *For a positive integer d and a graph G of order n, the graph G^d is obtained by replacing each vertex of G by a d-clique; therefore G^d has dn vertices. Besides the edges within the created d-cliques, there is an edge between two vertices v_i and v_j of G^d if and only if an edge existed in G between the two vertices corresponding to the d-cliques containing v_i and v_j for $i \neq j$.*

Note that $G^1 = G$. See Figure 3.2 for an illustration with $d = 3$ and G having 3 vertices and 2 edges. The new graph G^3 has 9 vertices and 27 edges.

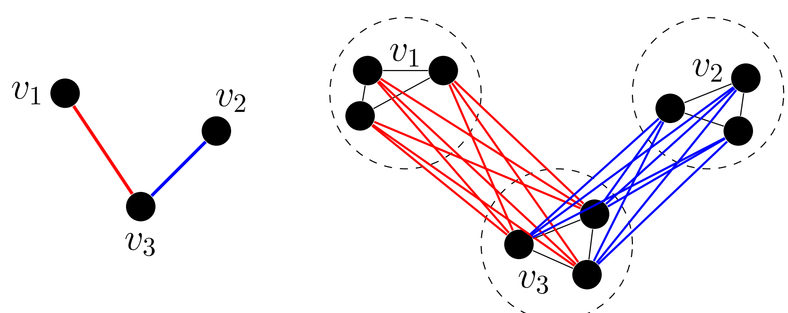

Figure 3.2: The graphs G and G^3

3.2 Determining $k_t(G^d_{X,F})$

As it is complicated to count cliques in $G^d_{X,F}$ directly, we introduce the notion of (X,F)-tuples.

Definition 3.3 *For $m \geq 1$, An ordered m-tuple, $\langle x_0, x_1, \cdots, x_{m-1}\rangle$ is an (X,F)-m-*tuple *if $x_i \subseteq X$ and $|x_i| \in F$ for $i < m$, and $|x_i \triangle x_j| \in F$ for all $i \neq j < m$.*

Lemma 3.4 *For a given X and F, let $S_m(X,F)$ denote the number of (X,F)-m-tuples, and $k_{m+1}(G_{X,F})$ denote the number of cliques of size $m+1$ in the graph $G_{X,F}$. We have:*

$$k_{m+1}(G_{X,F}) = \frac{2^n}{(m+1)!} S_m(X,F).$$

Proof

Case $m = 2$; that is, we wish to show that $k_3(G_{X,F}) = \frac{2^n}{(3)!} S_2(X,F)$. Let $\{a,b,c\}$ be a 3-clique in $G_{X,F}$. One can check that $\langle a\triangle b, a\triangle c\rangle$ is an (X,F)-2-tuple and that all the elements in the 2-tuple are mutually distinct. Since any permutation of two elements in the 2-tuple forms an (X,F)-2-tuple, there are 2 distinct (X,F)-2-tuples. While we choose a, we could have considered any of the vertices in the 3-clique to determine those 2 (X,F)-2-tuples. Therefore the clique $\{a,b,c\}$ determines $3 \times 2 = 6$ distinct (X,F)-2-tuples. On the other hand, one can easily show that if $\langle x_0, x_1\rangle$ is an (X,F)-2-tuple and if $a \subseteq X$, then $\{a, a\triangle x_0, a\triangle x_1\}$ is a 3-clique in $G_{X,F}$. Thus, there are exactly $\frac{2^n}{(3)!} S_2(X,F)$ 3-cliques in $G_{X,F}$.

Case $m = 3$; that is, we wish to show that $k_4(G_{X,F}) = \frac{2^n}{(4)!} S_3(X,F)$. Let $\{a,b,c,d\}$ be a 4-clique in $G_{X,F}$. One can check that $\langle a\triangle b, a\triangle c, a\triangle d\rangle$ is an (X,F)-3-tuple and that all the elements in the triple are mutually distinct. Since any permutation of three elements in the 3-tuple forms an (X,F)-3-tuple, there are 6 distinct (X,F)-3-tuples.

While we choose a, we could have considered any of the vertices of the cliques to determine those 6 (X,F)-3-tuples,. Therefore the clique $\{a,b,c,d\}$ determines $4\times 6=24$ distinct (X,F)-3-tuples. On the other hand, one can easily show that if $\langle x_0,x_1,x_2\rangle$ is an (X,F)-3-tuple and if $a\subseteq X$, then $\{a,a\triangle x_0,a\triangle x_1,a\triangle x_2\}$ is a 4-clique in $G_{X,F}$. Thus, there are exactly $\frac{2^n}{(4)!}S_3(X,F)$ 4-cliques in $G_{X,F}$.

Case $m=k$; that is, we wish to show that $k_{m+1}(G_{X,F})=\frac{2^n}{(m+1)!}S_m(X,F)$. Let $\{x_0,x_1,...x_m\}$ be a $(m+1)$-clique in $G_{X,F}$. One can check that $\langle x_0\triangle x_1,x_0\triangle x_2,...,x_0\triangle x_m\rangle$ is an (X,F)-m-tuple and all the elements in this m-tuple are mutually distinct. Since any permutation of those m elements in the m-tuple forms an (X,F)-m-tuple, there are $m!$ distinct (X,F)-m-tuples. While we choose x_0, we could have considered any of the vertices of the cliques to determine the $m!$ (X,F)-m-tuples. Therefore the clique $\{x_0,x_1,...,x_m\}$ determines $(m+1)\times m!=(m+1)!$ distinct (X,F)-m-tuples. On the other hand, one can easily show that if $\langle x_0\triangle x_1,x_0\triangle x_2,...,x_0\triangle x_m\rangle$ is an (X,F)-m-tuples and if $a\subseteq X$, then $\{a,a\triangle x_0,a\triangle x_1,...,a\triangle x_m\}$ is a m-clique in $G_{X,F}$. Thus, there are exactly $\frac{2^n}{(m+1)!}S_m(X,F)$ $(m+1)$-cliques in $G_{X,F}$. $\square$

Once we showed the relationship between the number of m-tuples and the number of $(m+1)$-cliques, we present the closed formula for $\lim_{d\to\infty}\frac{k_4(G^d)+k_4(\overline{G^d})}{\binom{nd}{4}}$ in Lemma 3.5. While this proof can be found in [12], we recall it as same ideas are used for larger t. Note that $c_4\leq\lim_{d\to\infty}\frac{k_4(G^d)+k_4(\overline{G^d})}{\binom{nd}{4}}$.

Lemma 3.5 *Consider the infinite sequence of graphs $\{G^d\}$ for $d\geq 1$ obtained from a seed graph G of size n, we have*

$$\lim_{d\to\infty}\frac{k_4(G^d)+k_4(\overline{G^d})}{\binom{nd}{4}}=\frac{24(k_4(G)+k_4(\bar{G}))+36k_3(G)+14k_2(G)+k_1(G)}{n^4}.$$

Proof Given n and d, we wish to determine the number of 4-cliques in G^d. We consider the 5 possible positions of the 4 vertices forming a 4-clique in G^d:

- Assume that the 4 vertices belong to d-cliques arising from 4 distinct vertices in G. Thus, there are $d^4 k_4(G)$ such 4-cliques in G^d. To simplify the presentation, we use $Q_1(d)$ to denote this number.

- Assume that 3 vertices belong to d-cliques arising from 3 distinct vertices in G and that the remaining vertex belong to one of these d-cliques. There are 3 ways to choose the d-clique with 2 vertices, $\binom{d}{2}$ ways to choose the 2 vertices within the clique, and the remaining 2 vertices can be chosen independently. Thus, there are $3\binom{d}{2}d^2 k_3(G)$ such 4-cliques in G^d. To simplify the presentation, we use $Q_2(d)$ to denote this number.

- Assume that 2 vertices belong to d-cliques arising from 2 distinct vertices in G and that the remaining 2 vertices belong to one of these d-cliques. There are 2 ways to choose the d-clique with 3 vertices, $\binom{d}{3}$ ways to choose the 3 vertices within the clique, and the remaining vertex can be chosen independently. Thus, there are $2\binom{d}{3}dk_2(G)$ such 4-cliques in G^d. To simplify the presentation, we use $Q_3(d)$ to denote this number.

- Assume that 2 vertices belong to a d-clique arising from a vertex in G and that the remaining 2 vertices belong to a d-clique arising from another vertex in G. There are $\binom{d}{2}$ ways to choose the two vertices in each of the d-cliques. Thus, there are $\binom{d}{2}^2 k_2(G)$ such 4-cliques in G^d. To simplify the presentation, we use $Q_4(d)$ to denote this number.

- Assume that all the 4 vertices belong to a d-clique arising from a vertex in G. There are $\binom{d}{4}$ ways to choose the 4 vertices in the d-clique. Thus, there are $\binom{d}{4}t$

such 4-cliques in G^d. To simplify the presentation, we use $Q_5(d)$ to denote this number.

Thus,

$$\begin{aligned}
k_4(G^d) &= Q_1(d) + Q_2(d) + Q_3(d) + Q_4(d) + Q_5(d) \\
&= \binom{d}{1}^4 k_4(G) + 3\binom{d}{2}\binom{d}{1}^2 k_3(G) + [2\binom{d}{3}\binom{d}{1} + \binom{d}{2}^2]k_2(G) + \binom{d}{4}k_1(G) \\
&= d^4k_4(G) + 3\binom{d}{2}d^2k_3(G) + [2\binom{d}{3}d + \binom{d}{2}^2]k_2(G) + \binom{d}{4}k_1(G) \\
&= d^4k_4(G) + \frac{3}{2}d^4k_3(G)O_1(d) + [\frac{2}{3!}d^4O_2(d) + \frac{1}{4}d^4O_3(d)]k_2(G) + \frac{1}{4!}d^4k_1(G)O_4(d)
\end{aligned}$$

where $O_1(d) = \frac{d-1}{d}$, $O_2(d) = \frac{(d-1)(d-2)}{d^2}$, $O_3(d) = \frac{(d-1)^2}{d^2}$, and $O_4(d) = \frac{(d-1)(d-2)(d-3)}{d^3}$. Since $\lim_{d\to\infty} O_i(d) = 1$, for $i = 1, 2, 3, 4$, we have:

$$\begin{aligned}
\lim_{d\to\infty}\frac{k_4(G^d)}{\binom{nd}{4}} &= \lim_{d\to\infty}\frac{\sum Q_i(d)}{\binom{nd}{4}} \\
&= \lim_{d\to\infty}\frac{d^4k_4(G)+\frac{3}{2}d^4k_3(G)O_1(d)+[\frac{2}{3!}d^4O_2(d)+\frac{1}{4}d^4O_3(d)]k_2(G)+\frac{1}{4!}d^4k_1(G)O_4(d)}{\binom{nd}{4}} \\
&= \lim_{d\to\infty}\frac{d^4[k_4(G)+\frac{3}{2}k_3(G)O_1(d)+[\frac{2}{3!}O_2(d)+\frac{1}{4}O_3(d)]k_2(G)+\frac{1}{4!}k_1(G)O_4(d)]}{\frac{(nd)^4}{4!}\frac{(nd-1)(nd-2)(nd-3)}{(nd)^3}} \\
&= \frac{\lim_{d\to\infty} d^4[k_4(G)+\frac{3}{2}k_3(G)O_1(d)+[\frac{2}{3!}O_2(d)+\frac{1}{4}O_3(d)]k_2(G)+\frac{1}{4!}k_1(G)O_4(d)]}{\lim_{d\to\infty}\frac{(nd)^4}{4!}\frac{(nd-1)(nd-2)(nd-3)}{(nd)^3}} \\
&= \frac{\lim_{d\to\infty} k_4(G) + \frac{3}{2}k_3(G) + [\frac{2}{3!} + \frac{1}{4}]k_2(G) + \frac{1}{4!}k_1(G)}{\lim_{d\to\infty}\frac{n^4}{4!}} \\
&= \lim_{d\to\infty}\frac{4! * [k_4(G) + \frac{3}{2}k_3(G) + [\frac{2}{3!} + \frac{1}{4}]k_2(G) + \frac{1}{4!}k_1(G)]}{4! * \frac{n^4}{4!}} \\
&= \lim_{d\to\infty}\frac{24k_4(G) + 36k_3(G) + 14k_2(G) + k_1(G)}{n^4}
\end{aligned}$$

which completes the proof. □

A similar method can be used for $t \geq 5$. Tables 3.1, 3.2 and 3.3 show the possible positions of the t vertices forming a t-clique for $t = 5, 6$, and 7. We use $\{a_1, a_2, a_3, ...\}$ to denote the possible positions with a_i denoting the the number of vertices of the t-cliques in the same d-clique arising from a vertex in G. For example, in Table 3.1, the case: $\{1, 1, 1, 2\}$ means that 4 vertices are in d-cliques arising from 4 distinct vertices of G and the remaining vertex is in one of these d-cliques.

Cases	$Q(d)$
{1,1,1,1,1}	$\binom{d}{1}^5 k_5(G)$
{1,1,1,2}	$4\binom{d}{2}\binom{d}{1}^3 k_4(G)$
{1,1,3} or {1,2,2}	$[3\binom{d}{3}\binom{d}{1}^2 + 3\binom{d}{2}^2\binom{d}{1}]k_3(G)$
{1,4} or {2,3}	$[2\binom{d}{1}\binom{d}{4} + 2\binom{d}{3}\binom{d}{2}]k_2(G)$
{5}	$\binom{d}{5}k_1(G)$

Table 3.1: Possible positions for $t = 5$ and associated number of 5-cliques

Cases	$Q(d)$
{1,1,1,1,1,1}	$\binom{d}{1}^6 k_6(G)$
{1,1,1,1,2}	$5\binom{d}{2}\binom{d}{1}^4 k_5(G)$
{1,1,1,3} or {1,1,2,2}	$[4\binom{d}{3}\binom{d}{1}^3 + \binom{4}{2}\binom{d}{2}^2\binom{d}{1}^2]k_4(G)$
{1,1,4} or {1,2,3} or {2,2,2}	$[3\binom{d}{4}\binom{d}{1}^2 + 3 * 2\binom{d}{3}\binom{d}{2}\binom{d}{1} + \binom{d}{2}^3]k_3(G)$
{1,5} or {2,4} or {3,3}	$[2\binom{d}{1}\binom{d}{5} + 2\binom{d}{2}\binom{d}{4} + \binom{d}{3}\binom{d}{3}]k_2(G)$
{6}	$\binom{d}{6}k_1(G)$

Table 3.2: Possible positions for $t = 6$ and associated number of 6-cliques

Cases	$Q(d)$
{1,1,1,1,1,1,1}	$\binom{d}{1}^7 k_7(G)$
{1,1,1,1,1,2}	$6\binom{d}{2}\binom{d}{1}^5 k_6(G)$
{1,1,1,1,3} or {1,1,1,2,2}	$[5\binom{d}{3}\binom{d}{1}^4 + \binom{5}{2}\binom{d}{2}^2\binom{d}{1}^3]k_5(G)$
{1,1,1,4} or {1,1,2,3} or {1,2,2,2}	$[4\binom{d}{4}\binom{d}{1}^3 + 4 * 3\binom{d}{3}\binom{d}{2}\binom{d}{1}^2 + 4\binom{d}{2}^3\binom{d}{1}]k_4(G)$
{1,1,5} or {1,2,4} or {1,3,3} or {2,2,3}	$[3\binom{d}{1}^2\binom{d}{5} + 3 * 2\binom{d}{1}\binom{d}{2}\binom{d}{4} + 3\binom{d}{1}\binom{d}{3}\binom{d}{3} + 3\binom{d}{2}^2\binom{d}{3}]k_3(G)$
{1,6} or {2,5} or {3,4}	$[2\binom{d}{1}\binom{d}{6} + 2\binom{d}{2}\binom{d}{5} + 2\binom{d}{3}\binom{d}{4}]k_2(G)$
{7}	$\binom{d}{7}k_1(G)$

Table 3.3: Possible positions for $t = 7$ and associated number of 7-cliques

Tables 3.1, 3.2 and 3.3 yield the following lemma.

Lemma 3.6

$$\lim_{d\to\infty} \frac{k_5(G^d) + k_5(\overline{G^d})}{\binom{nd}{5}} = \frac{120(k_5(G)+k_5(\bar{G}))+240k_4(G)+150k_3(G)+30k_2(G)+k_1(G)}{n^5}.$$

$$\lim_{d\to\infty} \frac{k_6(G^d) + k_6(\overline{G^d})}{\binom{nd}{6}} = \frac{720(k_6(G)+k_6(\bar{G}))+1800k_5(G)+1560k_4(G)+540k_3(G)+62k_2(G)+k_1(G)}{n^6}.$$

$$\lim_{d\to\infty} \frac{k_7(G^d) + k_7(\overline{G^d})}{\binom{nd}{7}} = \frac{5040(k_7(G)+k_7(\bar{G}))+15120k_6(G)+16800k_5(G)+8400k_4(G)+1806k_3(G)+126k_2(G)+k_1(G)}{n^7}.$$

Setting $G = G_{X,F}$ in lemmas 3.5 and 3.6 , and substituting $k_m(G_{X,F})$ by $S_{m-1}(X, F)$ using lemma 3.4 yield the following lemma.

Lemma 3.7 *Given a pair* $(|X|, F)$,

$$\lim_{d\to\infty}\frac{k_4(G^d_{X,F})+k_4(\overline{G^d_{X,F}})}{\binom{d2^n}{4}}=\frac{S_3(X,F)+S_3(X,\bar{F})+6S_2(X,F)+7S_1(X,F)+1}{2^{3n}},$$

$$\lim_{d\to\infty}\frac{k_5(G^d_{X,F})+k_5(\overline{G^d_{X,F}})}{\binom{d2^n}{5}}=\frac{S_4(X,F)+S_4(X,\bar{F})+10S_3(X,F)+25S_2(X,F)+15S_1(X,F)+1}{2^{4n}},$$

$$\lim_{d\to\infty}\frac{k_6(G^d_{X,F})+k_6(\overline{G^d_{X,F}})}{\binom{d2^n}{6}}$$
$$=\frac{S_5(X,F)+S_5(X,\bar{F})+15S_4(X,F)+65S_3(X,F)+90S_2(X,F)+31S_1(X,F)+1}{2^{5n}},$$

$$\lim_{d\to\infty}\frac{k_7(G^d_{X,F})+k_7(\overline{G^d_{X,F}})}{\binom{d2^n}{7}}$$
$$=\frac{S_6(X,F)+S_6(X,\bar{F})+21S_5(X,F)+140S_4(X,F)+350S_3(X,F)+301S_2(X,F)+63S_1(X,F)+1}{2^{6n}}.$$

Tables 3.4 and 3.5 give the coefficients of k_i and S_i with different values of t. These entries can and were computed by hand.

t	k_1	k_2	k_3	k_4	k_5	k_6	k_7	k_8
4	1	14	36	24				
5	1	30	150	240	120			
6	1	62	540	1560	1800	720		
7	1	126	1806	8400	16800	15120	5040	
8	1	254	5796	40824	126000	191520	141120	40320

Table 3.4: The coefficients of $k_i(X, F)$

t	S_1	S_2	S_3	S_4	S_5	S_6	S_7
4	7	6	1				
5	15	25	10	1			
6	31	90	65	15	1		
7	63	301	350	140	21	1	
8	127	966	1701	1050	266	28	1

Table 3.5: The coefficients of $S_i(X, F)$.

We notice the following relations between the computed coefficients of k_i and S_i given in Tables 3.4 and 3.5. Given t, let $\alpha_{i,t}$, respectively $\beta_{i,t}$, denote the coefficient of $k_i(X, F)$, respectively $S_i(X, F)$, we have

$$\alpha_{i,t} = (\alpha_{i,t-1} + \alpha_{i-1,t-1}) \times i.$$

$$\beta_{i,t} = \beta_{i,t-1} \times (i + 1) + \beta_{i-1,t-1},$$

Such relation between the coefficients of k_i and S_i could be used to determine a closed formula for c_i with $i \geq 8$. We calculated the coefficient of k_i and S_i by hand for $t = 8$ and the obtained values follow the same relation, see Tables 3.6 and 3.7 for the coefficients $\alpha_{i,8}$ of k_i and $\beta_{i,8}$ of S_i.

t	k_1	k_2	k_3	k_4	k_5	k_6	k_7	k_8
8	1	254	5796	40824	126000	191520	141120	40320

Table 3.6: The coefficients of $k_i(X, F)$ for $t = 8$.

t	S_1	S_2	S_3	S_4	S_5	S_6	S_7
8	127	966	1701	1050	266	28	1

Table 3.7: The coefficients of $S_i(X, F)$ for $t = 8$.

3.3 Selecting $S_i(X, F)$

We should now select a pair X and F potentially yielding a new upper bound for c_i. The method used to compute S_i is essentially the one used in [12].

3.3.1 Computing S_i

Computing $S_1(X, F)$

Given X and F, we need to determine the number of 1-tuples in $G_{X,F}$. Consider $\langle x_0 \rangle$ with $x_0 \subseteq X$. We can generate all possible $|x_0| \in F$, and then $S_1(X, F)$ is determined via $S_1(X, F) = \sum_{|x_0| \in F} \binom{|X|}{|x_0|}$. For example, for $|X| = 10$ and $F = \{1, 3, 4, 6\}$, $S_1(X, F) = \binom{10}{1} + \binom{10}{3} + \binom{10}{4} + \binom{10}{6}$.

Computing $S_2(X, F)$

Consider $\langle x_0, x_1 \rangle$ with x_0 and x_1 distinct subsets of X and let $m_0 = |x_0 \setminus x_1|$, $m_1 = |x_1 \setminus x_0|$ and $m_{01} = |x_0 \cap x_1|$. We have $m_0 + m_{01} = |x_0|$, $m_1 + m_{01} = |x_1|$, and $m_0 + m_1 = |x_0 \triangle x_1|$. See Figure 3.3 for an illustration of the relationship of those m_i's via a set diagram.

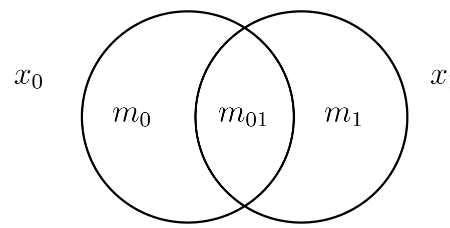

Figure 3.3: m_i's for S_2

If $\langle x_0, x_1\rangle$ is a (X, F)-2-tuple, see Definition 3.3, the following conditions must be satisfied: $|x_0|$, $|x_1|$ and $|x_0 \triangle x_1| \in F$. Thus, we determine

$$S_2(X,F) = \sum_{proper\ m_i's} \binom{|X|}{m_0} \cdot \binom{|X|-m_0}{m_1} \cdot \binom{|X|-m_0-m_1}{m_{01}}$$

for all possible $\langle m_0, m_1, m_{01}\rangle$ by generating all the possible combinations of $\{m_0, m_1, m_{01}\}$ satisfying the conditions on $|x_0|$, $|x_1|$ and $|x_0 \triangle x_1|$.

Computing $S_3(X, F)$

Consider $\langle x_0, x_1, x_2\rangle$ with x_0, x_1 and x_2 distinct subsets of X and let $m_{012} = |x_0 \cap x_1 \cap x_2|$, $m_{01} = |x_0 \cap x_1| \setminus m_{012}$, $m_{02} = |x_0 \cap x_2| \setminus m_{012}$, $m_{12} = |x_1 \cap x_2| \setminus m_{012}$, $m_0 = |x_0 \setminus (x_1 \cup x_2)|$, $m_1 = |x_1 \setminus (x_0 \cup x_2)|$, and $m_2 = |x_2 \setminus (x_0 \cup x_1)|$. See Figure 3.4 for an illustration of the relationship of those m_i's via a set diagram.

By Definition 3.3, the following conditions must be satisfied: $|x_i| \in F$ for $i = 0, 1, 2$, $|x_0 \triangle x_1| \in F$, $|x_0 \triangle x_2| \in F$, $|x_1 \triangle x_2| \in F$, and $|x_0 \cup x_1 \cup x_2| \leq |X|$. Restating these conditions in term of m_i's give: $m_0+m_{01}+m_{02}+m_{012} \in F$, $m_1+m_{01}+m_{12}+m_{012} \in F$, $m_2 + m_{12} + m_{02} + m_{012} \in F$, $m_0 + m_{02} + m_1 + m_{12} \in F$, $m_0 + m_{01} + m_2 + m_{12} \in F$, $m_1 + m_{01} + m_2 + m_{02} \in F$, and $m_0 + m_1 + m_2 + m_{01} + m_{02} + m_{12} + m_{012} \leq |X|$. Thus,

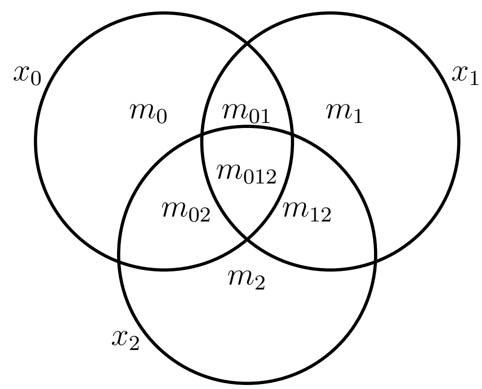

Figure 3.4: m's for S_3

we determine

$$S_3(X, F) = \sum_{proper\ m_i's} \binom{|X|}{m_0} \cdot \binom{|X|-m_0}{m_1} \cdot \binom{|X|-m_0-m_1}{m_{01}} \cdots$$

by generating all the possible combinations of m_i's satisfying these conditions.

Computing $S_i(X, F)$ for $i \geq 4$

Similarly to the determination of $S_2(X, F)$ and $S_3(X, F)$, we consider all ordered i-tuples $\langle x_0, x_1, x_2, ..., x_{i-1} \rangle$ of distinct subsets of X satisfying the associated conditions of the m_i's to compute

$$S_i(X, F) = \sum_{proper\ m_i's} \binom{|X|}{m_0} \cdot \binom{|X|-m_0}{m_1} \cdot \binom{|X|-m_0-m_1}{m_{01}} \cdots$$

Note that the number of potential proper m_i's increases quickly as to compute $S_i(X, F)$, we need to consider $(2^i - 1)$ possible m_i's. The intermediate computation of binomial coefficients is performed using dynamic Pascal triangle structure.

3.3.2 Computational speed-up

As outlined in the previous section, the main step of the computation is the determination of $S_j(X,F)$ and $S_j(X,\bar{F})$ for $j = 1, 2, \ldots, i-1$ which is achieved by finding all the m's satisfying some given conditions and computing the sum of the corresponding binomial coefficients. This process has an $O(2^{i|X|})$ worst-case complexity and therefore additional techniques are needed to make the computations tractable.

The main idea is an incremental approach illustrated by the following example underlying how the computations performed for S_j till $j = i-1$ can be exploited to obtain S_i. The approach is essentially based on the simple remark that Figure 3.4 can be obtained from Figure 3.3 by adding one more circle as illustrated in Figure 3.5.

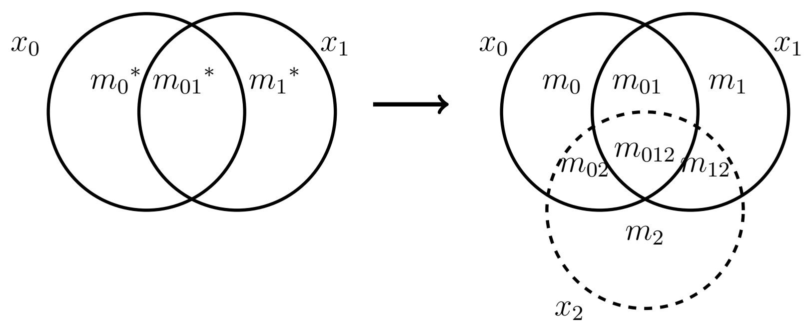

Figure 3.5: Obtaining S_3 using S_2

Considering a proper $m^* = \langle m_0^*, m_1^*, m_{01}^* \rangle$ for S_2, we can generate a proper $m = \langle m_0, m_1, m_2, m_{01}, m_{02}, m_{12}, m_{012} \rangle$ for S_3 via the equalities $m_0 + m_{02} = {m_0}^*$, $m_1 + m_{12} = {m_1}^*$ and $m_{01} + m_{012} = {m_{01}}^*$ combined with following constraints: $0 \le m_0 \le {m_0}^*$, $0 \le m_1 \le {m_1}^*$, and $0 \le m_{01} \le {m_{01}}^*$. Since $|x_2| \in F$, m_2 can be determined through $m_2 = z - m_{12} - m_{02} - m_{012}$ for $z \in F$. Finally, to check the symmetric difference constraints among the x_i's, it is enough to check $|x_0 \triangle x_2| \in F$ and $|x_1 \triangle x_2| \in F$.

In general, the determination of proper m's can be performed incrementally. Given a proper m^* for S_i and the associated product of binomial coefficients

$$Y^* = \binom{|X|}{m_0{}^*}\binom{|X|-m_0{}^*}{m_1{}^*}\binom{|X|-m_0{}^*-m_1{}^*}{m_2^*}\binom{|X|-m_0{}^*-m_1{}^*-m_2{}^*}{m_3{}^*}\cdots$$

and the corresponding proper m for S_{i+1} and product of binomial coefficients

$$Y = \binom{|X|}{m_0}\binom{|X|-m_0}{m_1}\binom{|X|-m_0-m_1}{m_2}\binom{|X|-m_0-m_1-m_2}{m_3}\cdots$$

we have:

$$Y = Y^* \cdot \binom{m_0{}^*}{m_0}\binom{m_1{}^*}{m_1}\cdots\binom{m_{01\cdots i}{}^*}{m_{01\cdots i}}\binom{|X|-m_0{}^*-m_1{}^*-m_{01}{}^*-\cdots}{m_i}.$$

While the worst-case complexity remains exponential due to the usual combinatorial explosion, the computational speed-up is significant for sizes we are considering.

3.3.3 Exploiting symmetry

An additional computational speed-up is obtained by exploiting the inherent symmetries of the m_i's. For example, consider $|X| = 10$ and $F = \{3, 4, 6, 7\}$ and the determination of S_2. As illustrated in Figure 3.6, proper $\langle m_0, m_1, m_{01}\rangle$ for S_2 with $m_0 \neq m_1$ yields another proper $\langle m_1, m_0, m_{01}\rangle$. Since the associated product of binomial coefficients for $\langle m_0, m_1, m_{01}\rangle$ and $\langle m_1, m_0, m_{01}\rangle$ are identical up to permutations, it is enough to compute one and count it twice.

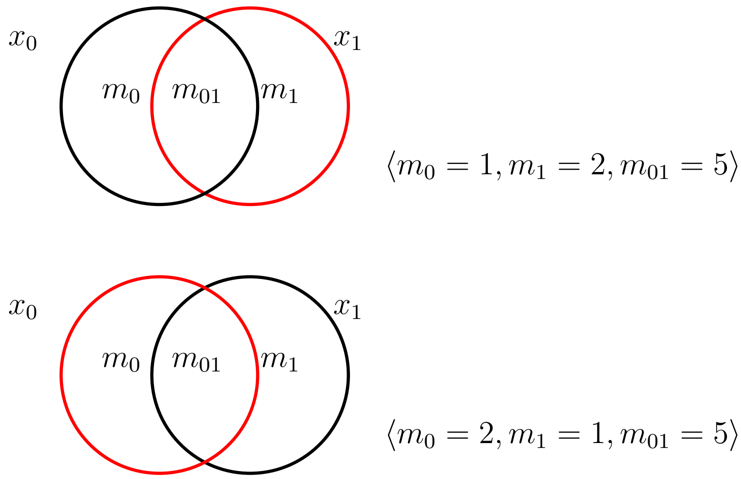

Figure 3.6: Symmetry with $|X| = 10$ and $F = \{3, 4, 6, 7\}$

In general, one can fix the order of the x_i's and take into account multiplicities by multiplying by the corresponding coefficients. For example, for S_4, see Table 3.8 for the coefficients corresponding to the different orderings of the x_i's.

Ordering	Coefficient
$\lvert x_0\rvert > \lvert x_1\rvert > \lvert x_2\rvert > \lvert x_3\rvert$	$4!$
$\lvert x_0\rvert > \lvert x_1\rvert > \lvert x_2\rvert = \lvert x_3\rvert$	$2\binom{4}{2}$
$\lvert x_0\rvert > \lvert x_1\rvert = \lvert x_2\rvert > \lvert x_3\rvert$	$2\binom{4}{2}$
$\lvert x_0\rvert = \lvert x_1\rvert > \lvert x_2\rvert > \lvert x_3\rvert$	$2\binom{4}{2}$
$\lvert x_0\rvert > \lvert x_1\rvert = \lvert x_2\rvert = \lvert x_3\rvert$	$\binom{4}{3}$
$\lvert x_0\rvert = \lvert x_1\rvert > \lvert x_2\rvert = \lvert x_3\rvert$	$\binom{4}{2}$
$\lvert x_0\rvert = \lvert x_1\rvert = \lvert x_2\rvert > \lvert x_3\rvert$	$\binom{4}{3}$
$\lvert x_0\rvert = \lvert x_1\rvert = \lvert x_2\rvert = \lvert x_3\rvert$	1

Table 3.8: Ordering of the x_i's and corresponding coefficients for S_4

As the size of the symmetry group increases with i, the computation gains increase accordingly as illustrated in Table 3.9 giving the number of proper instances before/after exploiting the symmetries to compute S_4, S_5 and S_6 with $(|X|, F) = (11, \{3, 4, 7, 8, 10, 11\})$. For S_7, the average ratio over all computations is about 1 %.

i	# instances (initial)	# instances (exploiting symmetry)	ratio
4	15,668	1,813	3.0%
5	377,196	17,625	0.5%
6	9,104,496	160,626	0.08%

Table 3.9: Exploiting symmetry for $(|X|, F) = (11, \{3, 4, 7, 8, 10, 11\})$

Chapter 4

Computation results

We developed a code written in `C++` to determine the S_i's given X and F and performed an exhaustive search over all $(|X|, F)$ for $|X| = 9$, 10, 11 and 12 for $t = 6$ and 7. The computation was run using Intel Quad core Q9550. We first run our code to re-compute previously known values given in [12, 14, 10] as testing and verification and to estimate the efficiency of the code. The computation of $S_1, \dots, S_6$ for all pairs $(|X|, F)$ considered in [12, 14, 10] yields the same values while requiring only a tiny fraction of the computation time previously required. As further testing and verification, we computed $S_1, \dots, S_7$ for *full families* because for such trivial family $\{1, 2, \dots, |X|\}$ the number of *i-tuples* can be expressed using Lemma 3.4 with a closed formula $S_i = \frac{(2^{|X|}-1)!}{(2^{|X|}-i-1)!}$. The computed values coincide with the ones given by the closed formula..

4.1 New upper bounds for c_6, c_7 and c_8

4.1.1 New upper bounds for c_6

The best results were achieved with $t = 6$ by $(|X|, F) = (10, \{1, 3, 4, 7, 8\})$ yielding $c_6 \leq 0.74444 \times 2^{1-\binom{6}{2}}$, see Table 4.1. Note that the same upper bound is achieved with $(|X|, F) = (10, \{3, 4, 7, 8, 9\})$.

i	$S_i(X,F)$	$S_i(X,\bar{F})$
1	505	518
2	125,010	135,726
3	14,562,090	17,463,606
4	726,780,600	1,028,265,840
5	13,191,935,400	26,106,252,480

Table 4.1: $S_i(X,F)$ and $S_i(X,\overline{F})$ for $(|X|, F) = (10, \{1, 3, 4, 7, 8\})$

4.1.2 New upper bounds for c_7

The best results were achieved with $t = 7$ by $(|X|, F) = (11, \{3, 4, 7, 8, 10, 11\})$ yielding $c_7 \leq 0.6869 \times 2^{1-\binom{7}{2}}$, see Table 4.2. Note that the same upper bound is achieved with $(|X|, F) = (11, \{2, 5, 6, 9, 10\})$.

i	$S_i(X,F)$	$S_i(X,\bar{F})$
1	1,002	1,045
2	490,050	556,842
3	113,148,090	146,860,362
4	11,590,147,800	17,896,958,640
5	506,500,533,000	950,437,303,200
6	14,677,396,549,200	21,359,851,904,160

Table 4.2: $S_i(X,F)$ and $S_i(X,\overline{F})$ for $(|X|, F) = (11, \{3, 4, 7, 8, 10, 11\})$

4.1.3 New upper bounds for c_8

Starting from $t = 8$, exhaustive search and computation become intractable even with the introduced computational speed-up. Therefore, we tried a guided local search using heuristics.

Search algorithm 1

We first noticed the following two patterns, shown in Figure 4.1, when plotting the upper bound c_t^+ as a function of t for a given F and $|X| = 10, 11, 12$. The left curve appears to be more common for larger c_4^+ while the right one appears to be more common for smaller c_4^+. Consequently, we simulate via partial computation the value for c_4 and reject $(|X|, F)$ returning values larger than a predetermined threshold. This approach was used for $t = 8$ and relatively small $|X|$ as it became intractable for large $|X|$.

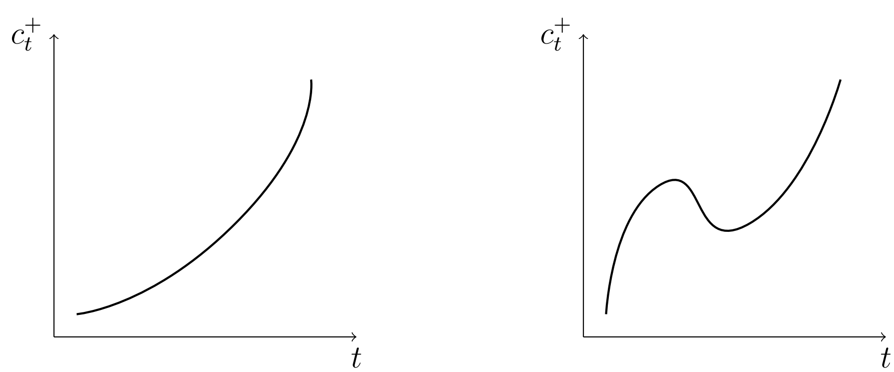

Figure 4.1: c_t^+ vs t for given $(|X|, F)$

Search algorithm 2

Represent the pair $(|X|, F)$ as the characteristic vector of F as a subset of $\{1, 2, \ldots, |X|\}$, one can notice some patterns among the $(|X|, F)$ achieving the best results for $t = 6$ and 7; that is, the best $(|X|, F)$ for $t = 6$, respectively $t = 7$, are obtained with $(|X|, F) = [1011001100]$, respectively $(|X|, F) = [00110011011]$. Extrapolating that such patterns remain valid for at least the few values of t, we restrict our search to $(|X|, F) = [\ldots 11001100 \ldots]$ when searching for a new upper bound for c_8. This search algorithm, combined with the previous one, yielded an improved upper bound $c_8 \leq 0.7002 \times 2^{1-\binom{8}{2}}$ for $(|X|, F) = (12, \{1, 3, 4, 7, 8, 11, 12\})$, see Table 4.3.

i	$S_i(X,F)$	$S_i(X,\overline{F})$
1	2,027	2,068
2	2,030,562	2,158,860
3	986,934,042	1,120,464,444
4	223,874,343,000	279,763,013,640
5	21,997,023,741,000	32,608,321,954,560
6	868,195,804,568,400	1,762,344,151,444,800
7	23,207,044,770,478,800	47,296,455,155,389,440

Table 4.3: $S_i(X, F)$ and $S_i(X, \overline{F})$ for $(|X|, F) = (12, \{1, 3, 4, 7, 8, 11, 12\})$

See Table 1 in the Appendix for results obtained via a few other seeds. Note that we tried the $(|X|, F)$ yielding the best bounds for c_5, c_6, c_7 and c_8 as seeds for $t = 9$ but were unable to improve the upper bound for c_9.

4.2 Conclusion and future work

We introduced a computational framework to search graphs potentially yielding improved upper bounds tightening the known counterexamples to the 1960 Erdős' Conjecture on multiplicities of complete subgraphs. We described significant computational speed-up allowing the determination of new upper bounds for $t = 6$, 7 and 8. We believe further investigation of the best, or near-best, graphs could help to refine the heuristics in order to tackle higher instances of t.

Part II

On square-maximal strings

Chapter 5

Introduction

5.1 Problem definition

In Chapter 1, we introduced the notion of strings, concatenation, string indexing, squares, primitive strings, and primitively rooted squares, and briefly introduced the problem. In this chapter we describe the problem in more details and give its background and history.

We start with notation: for integers d and n so that $2 \leq d \leq n$, the strings of length n with exactly d distinct symbols are referred to as (d, n)-*strings*. For instance, *aabbcdd* is a (4,7)-string. An integer function $\sigma_d(n)$ signifying the maximum number of distinct primitively rooted squares over all (d, n)-strings, is thus defined as $\sigma_d(n) = max\{s(x)|x \text{ is a } (d, n)\text{-string}\}$, where $s(x)$ denotes the number of distinct primitively rooted squares in a string x. An integer function $\sigma(n) = \max\{\ s_1(x)\ :\ |x| = n\}$ where $s_1(x)$ is the number of distinct squares in a string x is thus the maximum number of distinct squares over all strings of length n and so the problem of the maximum number of distinct squares is thus a determination of the value $\sigma(n)$ for any n. This is not an easy combinatorial problem and the chances of ever solving it by providing a closed formula are very slim. That is why most researchers really aim for reasonable lower and upper bounds of the function $\sigma(n)$.

5.2 Earlier results and conjectures

Fraenkel and Simpson in 1998 showed that the number of distinct squares in a string of length n is bounded from above by $2n$ and gave a lower bound of $n - o(n)$ asymptomatically approaching n from below for primitively rooted squares for infinitely many values of n, [9]. Their proof relied on a theorem by [3], describing the mutual configurations of three squares.

After a few years, in 2005, Ilie provided a simpler proof [17] avoiding the theorem of Crochemore and Rytter. In 2007, he presented an asymptotic upper bound $2n - \Theta(log\ n)$, [18]. In 2011, Deza, Franek and Jiang proposed a d-step approach to this problem for primitively rooted squares, [4]. They introduced the size of the alphabet, d, as a parameter in addition to the usual length of the string and instead of the function $\sigma(n)$ investigated the function $\sigma_d(n)$. They conjectured that $\sigma_d(n) \leq n - d$ and provided a strong supporting evidence for the bound. They investigated the fundamental properties of the function $\sigma_d(n)$, introduced the $(d, n-d)$ table where the value $\sigma_d(n)$ is the entry at the d-th row and the $(n-d)$-th column. They showed the critical role played by the main diagonal of the $(d, n-d)$ table and hence $(d, 2d)$-strings.

In [5], Deza, Franek and Jiang introduced a computational framework for determining $\sigma_d(n)$ values based on their investigation of the properties of $\sigma_d(n)$ in [4]. As mentioned in Chapter 1 in the brief introduction of the problem, a computational approach relying on brute force is not applicable for strings of length beyond approximately 32. They introduced the notion of s-cover, the basic tool for reduction of the search space. They were able to determine all the values of $\sigma_2(n)$ for $n \leq 53$ and $\sigma_3(n)$ for $n \leq 41$. It seems intuitively clear that that $\sigma_{d_1}(n) \geq \sigma_{d_2}(n)$ for $d_1 < d_2$, as for the given length a smaller alphabet gives a bigger freedom to create more squares; that is why most of the researchers in the field consider the case of the binary alphabet the hardest and most important. However, Deza, Franek, and Jiang discovered a counter-intuitive fact that $\sigma_2(33) < \sigma_3(33)$. Their effort lead to a slight improvement of the universal upper bound of Fraenkel and Simpson to $\sigma_2(n) \leq 2n - 66$ for $n \leq 53$.

Our contribution is an improvement of the sophistication of the computational framework introduced in [5], providing a significantly better efficiency and faster execution, allowing to carry out the computations for much higher values of n and d, doubling the reach of the method.

5.3 Previous computational framework

Our aim is to calculate the value of $\sigma_d(n)$ for given d and n. Without any reduction of the problem, d^n strings would have to be generated and for each of them, the number of distinct primitively rooted squares computed. Thus, the search space increases exponentially. There are some obvious simplifications - for instance, a string and a string that is created by permuting its alphabet have the same number of distinct primitively rooted squares, so in fact we can "only" generate d^{n-1} strings by fixing the very first character. Moreover, we could only generate strings whose first occurrences of symbols are in lexicographic order. For instance, a string *cbbacbba* has the same number of distinct primitively rooted squares as the string *abbcabbc*. In the former, the first occurrence of c precedes the first of occurrence of b, which precedes the first occurrence of a, so such string could be safely ignored. Despite all such improvements, and we will use all of them, the essential exponential nature of the problem cannot be avoided. Therefore, we want to avoid generating full strings that cannot be square-maximal, i.e. we must develop methods and techniques for determining from a partially generated string if it has any chance to be completed to a square-maximal one, and if not, abort its completion.

The computational framework used in [5] relies on the properties of s-cover to reduce the search space. In the following subsection, we describe how the s-cover is used to simplify the generation of the pool of possible square-maximal strings.

5.3.1 Structural properties of (d, n)-strings

We present two notions associated with (d, n)-strings in this section. One is the so-called *core vector*, and the other is the *s-cover*. Both of them figure in necessary conditions guaranteeing that the generated string x satisfies $s(x) > \sigma_d^-(n)$ for a given lower bound denoted as $\sigma_d^-(n)$. The basic setup is as follows: use some heuristics to obtain quickly and cheaply a lower bound $\sigma_d^-(n)$ for $\sigma_d(n)$. Then utilizing the s-cover and the core vector, generate only the strings that are not guaranteed to give a lower value than $\sigma_d^-(n)$, thus the closer the lower bound $\sigma_d^-(n)$ is to the real value of $\sigma_d(n)$, the better. Note that this approach can still generate strings with fewer than $\sigma_d^-(n)$ distinct primitively rooted squares, but the ones who are in an early stage guaranteed to have fewer than $\sigma_d^-(n)$ distinct primitively rooted squares are not completed. We have to start with definitions of the notions needed. Since we are only computing the number of distinct primitively rooted squares, for the sake of simplicity, by square we really mean a primitively rooted square.

Definition 5.1 *[5]*

1. *For a square S occurring in a string x, its* core *is the set of indices formed by the intersection of the indices of all its occurrences in x.*

2. *For a string x of length n, $k_i(x)$ is the number of non-empty cores of squares of x containing i for $i = 0, .., n-1$. The vector $k(x) = [k_0(x), k_2(x), .., k_{n-1}(x)]$ is referred as the* core vector *of x.*

The motivation behind the definition is simple. One of the biggest problems with estimating $s(x)$ is the fact that there is no obvious way to apply induction. One can see that there is no obvious relationship between $s(x) + s(y)$ and $s(xy)$ as the concatenation xy of x and y can both reduce the number of distinct squares (we cannot count the same type of square in x and in y twice), and increase the number of distinct squares by creating new squares not existing in either x or y. So, there is no apparent way to reduce the length of the string for the induction step. One way to apply reduction to a (d, n)-string is to "remove" one symbol from x: either remove it and concatenate the leftovers which results in a $(d, n-1)$-string (if the original string did not have any singletons), or replace it by a wholly new symbol which results in a $(d+1, n)$-string. If the number of distinct squares destroyed by this process is known to be less or equal to k, then $s(x) \le \sigma_d(n-1) - k$ in the former and $s(x) \le \sigma_{d+1}(n) - k$ in the latter case. Both values $\sigma_d(n-1)$ and $\sigma_{d+1}(n)$ are in the $(d, n-d)$ table to the left of the d, $n-d$ entry, and so this approach is conducive to the computation of the entries in the $(d, n-d)$ table in the fashion of dynamic programming. The co-ordinate $k_i(x)$ of the core vector for x tells us how many distinct squares would be destroyed if we "removed" the i-th symbol of x.

Definition 5.2 defines a notion of density for a string. Note that it depends on the availability of the lower bound $\sigma_d^-(n)$. A more proper way would be to define t-density and then talk of $\sigma_d^-(n)$-density. But since it is not used in any other context, for the sake of simplicity we use it as "density" knowing that it depends on whatever kind of $\sigma_d^-(n)$ we have available.

Definition 5.2 *A singleton-free (d,n)-string is* dense*, if its core vector $k(x)$ satisfies $k_i(x) > \sigma_d^-(n) - s(x[1 \,..\, i-1]) - m_i$ for $i = 1, ..., n$, where $m_i = max\{\sigma_{d'}(n-i) : d - |\mathcal{A}(x[1..i-1])| \leq d' \leq min(n-i, d)\}$.*

Again, the motivation behind this definition is straightforward. If a string is dense, any "removal" of any symbol of x will destroy too many distinct squares. Thus, for a string that is not dense, a "removal" of a symbol will result in a loss of too few distinct squares and so the maximum number of distinct squares for such a string will not exceed $\sigma_d^-(n)$. Also note that the definition of density uses two quantities, the exact number of distinct squares for $x[0..i-1]$ and just an estimate of the maximum number of distinct squares for $x[i+1..n-1]$. This is no coincidence. When a string will be partially generated, we will be able to compute the density of the part that is generated so far and reject the string if it is not dense, thus eliminating the generation of its completion, that would be hopeless anyway. This all is formalized in the following lemma whose proof is given in [5].

Lemma 5.3 *[5] If a (d,n)-string x is not dense, then $s(x) \leq \sigma_d^-(n)$.*

An s-cover is a generalization of a cover of a string and hence of the structure of the string. Therefore, the elements of the s-cover are substrings of x and their union gives the whole x. Deza, Franek and Jiang in [5] used the s-cover to represent the structure of a dense string and instead of generating strings, they generated the required s-covers whose unions are the strings.

We can encode a square as a triple (s, e, p) where s is the starting position of the square, e is the ending position of the square, and p is its period. E.g. *abab* in a string *ababaa* can be encoded as $(0, 3, 2)$. In fact, we could only use a pair (s, p) or (s, e) to uniquely encode a square as $e = s + 2p - 1$. We use the triple (s, p, e) for convenience in discussions of the framework, however in the computer programs it is really coded as (s, p).

For the upcoming definition of the s-cover, we must first properly define what we mean by the union of substrings. First, it is not referred to as union but rather as a *join* of substrings. Second, the usual symbol for set union $\cup$ is used, as it is clear from the context if a set union is meant or a join of substrings.

Let $x = x[0..n-1]$ and let $0 \leq i \leq j < n$, $0 \leq k \leq m < n]$. Then the *join* of $x[i..j]$ and $x[k..m]$ denoted as $x[i..j] \cup x[k..m]$ is defined only when $j \geq k$ and equals $x[min\{i,k\}..max\{j,m\}]$.

Definition 5.4 ([5]) *An s-cover of a string $x = x[0..n-1]$ is a sequence of primitively rooted squares $\{S_i = (s_i, e_i, p_i) | 1 \leq i \leq m\}$ if*

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 $1 \leq i \leq m$ so that S is a substring of S_i, denoted by $S \subseteq S_i$.*

The table 5.1 illustrates an s-cover $\{(0,3,5),(1,3,6),(2,3,7),(5,2,9)\}$ of a string $x = abbabbaba$.

string x	a	b	b	a	b	b	a	b	a
s_1	a	b	b	a	b	b			
s_2		b	b	a	b	b	a		
s_3			b	a	b	b	a	b	
s_4						b	a	b	a

Table 5.1: An s-cover of a string *abbabbaba*

A string that has an s-cover is referred to as s-covered. It is clear that from the condition (2) in the definition, an s-covered string must be singleton free.The following lemma shows that an s-cover of a string is unique and that we hence can speak of the s-cover of a string.

Lemma 5.5 ([5]) *If a string admits an s-cover, then the s-cover is unique.*

Lemma 5.6 ([5]) *If a string admits an s-cover, then the string must be singleton free.*

The two lemmas that follow help us narrow down the search space.

Lemma 5.7 ([5]) *If a singleton-free square-maximal (d, n)-string x does not have an s-cover, then $\sigma_d(n) = \sigma_d(n-1)$.*

Lemma 5.8 ([5]) *If a square-maximal (d, n)-string has a singleton, then $\sigma_d(n) = \sigma_{d-1}(n-1)$.*

Thus, we can divide the set of all the square-maximal (d, n)-strings into three parts:

- Part 1: All the singleton-free not s-covered square-maximal (d, n)-strings.
- Part 2: All the singleton-free square-maximal s-covered (d, n)-strings.
- Part 3: All the square-maximal (d, n)-strings that have singletons.

From lemma 5.7 and 5.8, it is easy to see that the upper bound of $\sigma_d(n)$ is already known for Part 1 and Part 3. Since we always make sure that the value of an available $\sigma_d^-(n)$ is at least equal to max $\{\ \sigma_d(n-1), \sigma_{d-1}(n-1)\ \}$, we only need to generate the strings from Part 2. This is how the search space is significantly reduced. To gage the significance, we generated a few sets of (d, n)-strings and compared the numbers of s-covered and not s-covered, see Figure 5.1.

```
d = 2, n = 10 Covered:  154        not Covered:  357
d = 2, n = 15 Covered:  4074       not Covered:  12,309
d = 2, n = 20 Covered:  109,437    not Covered:  414,850
d = 3, n = 10 Covered:  183        not Covered:  9,147
d = 3, n = 15 Covered:  21,681     not Covered:  2,353,420
d = 3, n = 20 Covered:  1,908,923  not Covered:  578,697,523
```

Figure 5.1: Comparing the numbers of s-covered and general strings

Thus, we only need to generate the s-covers of the strings to be generated during the computations. We also know that $s(x) \leq \sigma_d^-(n)$ if the string x is not dense from lemma 5.3, so we really only need to generate the s-covers whose union will be dense.

5.3.2 Generating the required (d, n)-strings

Previously, we discussed why it is sufficient to only generate s-covered strings that are dense as a pool of possible square-maximal strings whose number of distinct primitively rooted squares can exceed the value of $\sigma_d^-(n)$. This is utilized in the computational framework of [5] where all such s-covers are generated one square at a time. In fact, the s-covers must satisfy even more stringent conditions. Firstly, they cannot have consecutive adjacent squares. Secondly, we can check after each square is generated if the density condition is satisfied, and if not, there is no reason to continue with the generation of the complete s-cover. For a given d and n, we generate the first square, then we extend the partial s-cover by generating the next square and checking the density. All the generated strings are generated using the so-called *restricted growth* algorithm making sure that the introduction of previously unused symbol follows the lexicographic order as we discussed previously. This is achieved by checking the frequency of occurrences of all symbols and their first occurrences. Once the length n is reached, we compute the number of distinct primitively rooted squares in the string using the Crochemore partitioning algorithm-based program of Franek, Jiang and Weng [11]. The same program is also used to compute the core vector for checking the density. A simple pseudo-code of the computational framework is shown below in Figure 5.2.

- Step 1: Set the first square is $S_1 = (s_1, e_1, p_1)$, with $s_1 = 1$ and $p_1 = 1$. Let $j = 2$.
- Step 2: Fill the pattern of S_1, and count the frequency occurrence of each symbol.
- Step 3: Generate the next square $S_j = (s_j, e_j, p_j)$ by $s_j = s_{j-1} + 1$.
- Step 4: Fill the pattern of S_j.
- Step 5: Check the validity of S_j. If it is not valid, clear the pattern in S_j and go to Step 4. If the maximum length has been reached, go to Step 6, else go to Step 3 and increment $j = j + 1$.
- Step 6: Check the value of $s(x)$.
- Step 7: If all the strings have been generated, then go to Step 8, else update the value of $\sigma_d^-(n)$ and go to Step 2 with $p_1 = p_1 + 1$.
- Step 8: Output the value of $\sigma_d^-(n)$ as the value for $\sigma_d(n)$.

Figure 5.2: The computational framework in pseudo-code.

In Step 5 we need to check the validity of the partially generated s-cover, i.e. whether we can add S_j to it. What we must check, whether the string which will be the join of all squares in the s-cover will be dense. In a sense, we must predict and that is not an easy task: in general it is not true that if a prefix of a string is not dense than the whole string would not be dense. Why? Because when generating the remaining part of the string we might add distinct squares that start in the prefix (though they end in the remaining part) and this will make the prefix dense. However, it is true for the partially generated string up to the beginning of S_j, as we are guaranteed by the property of the s-cover that no square will be created that starts before S_j and ends past the e_{j-1}. Thus, the program of Franek, Jiang and Weng [11] is used to compute the core vector of $y = \bigcup_{1 \leq i \leq j} [0..e_j]$ and that is used for checking the density of y.

Chapter 6

Improving the original computational framework

In this section we describe efficient heuristics for the computation of the lower bound $\sigma_d^-(n)$. Let us explain and justify the term "efficient". By $efficient$ we really mean a heuristics that in a majority of cases gives the actual value of $\sigma_d(n)$. The heuristics we actually used for this project were derived from the properties of $\sigma_d(n)$ function as revealed by the $(d, n-d)$ table and they were really efficient in our sense as they provided the right values except in two or three cases.

6.1 The $(d, n-d)$ table

We use $(d, n-d)$ table to record the values of $\sigma_d(n)$, [4]. The rows of the table are indexed by d, and the columns are indexed by $n-d$. Each cell contains the corresponding value of $\sigma_d(n)$. We can observe how some of the properties of the function $\sigma_d(n)$ propagate through the table. Table 6.1 only shows the upper-left corner of the potentially infinite $(d, n-d)$ table for $d < 11$ and $n-d < 11$. The values in bold represent the main diagonal, i.e. the values of the type $\sigma_d(2d)$. The up-to-date values can be found on the website of Mei Jiang [20]

	$n-d$										
d	1	2	3	4	5	6	7	8	9	10	11
1	**1**	1	1	1	1	1	1	1	1	1	.
2	1	**2**	2	3	3	4	5	6	7	7	.
3	1	2	**3**	3	4	4	5	6	7	8	.
4	1	2	3	**4**	4	5	5	6	7	8	.
5	1	2	3	4	**5**	5	6	6	7	8	.
6	1	2	3	4	5	**6**	6	7	7	8	.
7	1	2	3	4	5	6	**7**	7	8	8	.
8	1	2	3	4	5	6	7	**8**	8	9	.
9	1	2	3	4	5	6	7	8	**9**	9	.
10	1	2	3	4	5	6	7	8	9	**10**	.
11	.	.	.	.	.	.	.	.	.	.	.

Table 6.1: $(d, n-d)$ table

6.2 Efficient heuristics for lower bound when $d > 2$

In the previous chapter, we introduced the computational framework for computations of the values of $\sigma_d(n)$. We also discussed the role of the available lower bound $\sigma_d^-(n)$ and the advantages of having a lower bound as close to the actual value as possible as we only need to generate the strings that have a chance of having more distinct primitively rooted squares than $\sigma_d^-(n)$. In this chapter, we discuss some efficient heuristics for finding a better $\sigma_d^-(n)$.

From table 6.2, it is clear that $\sigma_d(n) \geq \sigma_{d-1}(n-2)+1$, $\sigma_d(n) \geq \sigma_{d-1}(n-1)$ and $\sigma_d(n) \geq \sigma_d(n-1)$. For the proof, refer to [4]. This is depicted in Table 6.2.

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

Table 6.2: a piece of $(d, n-d)$ table

We can simply set $\sigma_d^-(n) = \max\{(\sigma_{d-1}(n-1), \sigma_{d-1}(n-2)+1, \sigma_d(n-1)\}$, provided that we have obtained the three values involved first. This motivates our method of filling in the values of the table in the fashion of dynamic programming, i.e. left-to-right and top-to-down. Though one can experiment, and we did, with various heuristics, we found this to be the most efficient and satisfactory one. In particular, we tried the same heuristics as for $d = 2$ (described below), and several of its variations, but it is much more computationally extensive and did not provide any better estimates.

6.3 Efficient heuristics for $d = 2$

When $d = 2$, we have $\sigma_d(n-1) \leq \sigma_d(n)$. Thus, we could simply set $\sigma_d^-(n) = \sigma_d(n-1)$. But this really does not help too much. As a consequence of the Fraenkel-Simpson result, we know, see [5], that $0 \leq |\sigma_d(n) - \sigma_d(n-1)| \leq 2$ and $1 \leq |\sigma_{d+1}(2d+2) - \sigma_d(2d)| \leq 2$, and so we would have a big gap between the lower bound and the actual value, since we believe that for the vast majority of entries, $\sigma_d(n) = \sigma_d(n-1) + 1$ and it costs significant computation to reject the value $\sigma_d(n-1) + 2$ in comparison to rejecting the value $\sigma_d(n-1) + 1$. There are some other heuristics that can help get a tighter $\sigma_2^-(n)$.

6.3.1 A better bound using a smaller search space

We performed many computational tests for small n's and produced the corresponding sets of square-maximal strings and investigated them. From these results we derived several heuristic rules that help get a better lower bound. The rules are listed below as Rule 1, Rule 2, and Rule 3.

- Rule 1: The string must be balanced over every prefix.

  By "balanced" we mean the following: denote the string as x. Let x_i denote the prefix of x of length i. The frequencies of a's and b's in x_i are denoted by $f_i(a)$ and $f_i(b)$, respectively. For a predefined constant c, determined empirically, it is required that $|f_i(a) - f_i(b)| \leq c$ for $1 \leq i \leq n$.

- Rule 2: The s-cover of the string must contain squares with periods bounded by a predefined constant, also determined empirically.

  Note that this significantly reduces the computational costs of generating the s-cover.

- Rule 3: The string must not contain triples of consecutively occurring symbols (i.e. aaa or bbb).

When we checked the square-maximal strings of small length, the differences of the frequencies of two symbols were quite small. This lead us to formulate Rule 1. Similarly, when investigating the s-covers of the strings, we noticed that they consisted mostly of shorter squares. This lead us to formulate Rule 2. Though we found some square-maximal strings containing aaa's or bbb's, the investigation of the strings revealed that for every $\sigma_2(n)$ there was a square-maximal string exhibiting no aaa's or bbb's. This lead to Rule 3.

It is quite obvious how these rules simplify the computations. If Rule 1 is used, a lot of strings are rejected during the generation early on when the balance is violated. If Rule 2 is used, many s-covers are also rejected during the generation early on and the program's looping is highly reduced. If Rule 3 is used, again, many strings will be rejected early on.

Denote by $L_2(n)$ the set of all s-covered $(2, n)$-strings satisfying those three rules, then we can set $\sigma_2^-(n) = \max\{\sigma_2(n-1), \max_{x \in L_2(n)} s(x)\}$.

During the computation, we first computed $\max_{x \in L_2(n)} s(x)$ separately. Then run the generation program setting the initial value of $\sigma_2^-(n)$ to $\max\{\sigma_2(n-1), max_{x \in L_2(n)} s(x)\}$. Although the program has been run twice, overall total running times have improved.

We also noticed following fact presented here as remark with a simple proof.

Remark 6.1 *Let the predefined constant in Rule 1 be c. If a string x satisfies Rule 1 and Rule 3, then $c \leq \lceil |x|/3 \rceil + r$ where r is the reminder of $|x|/3$.*

Proof Group any three continuous symbols in the string x, the difference of the frequencies of a's and b's will be equal to 1. There are $\lfloor |x|/3 \rfloor$ groups of substrings with length 3 if we counted the groups from the beginning of the string, and left 0, 1 or 2 symbols at the end (i.e. the remainder r). Thus the maximum value of c will be equal to $\lceil |x|/3 \rceil + r$. $\square$

In real practice, we used to set the value of c by $c = \min\{\lceil n/3 \rceil + r, c^*\}$ during the string generation, where n is the length of the string to be generate, and c^* is a predefined empirically determined constant. By decreasing c^* we were able to narrow down the set $L_2(n)$.

6.3.2 Find a better bound by using prefix and suffix construction

We used the previous heuristics to find the values of $\sigma_2(n)$ for $n-2 \leq 53$. The running times for $n-2 > 53$ were simply too long and the program runs were not terminating. Thus we begun to investigate the runs for smaller lengths trying finding some useful information among those square-maximal strings.

The tables 6.3, 6.4 and 6.5 show some of the square-maximal strings for $n-2 = 41$ to 46, $n-2 = 47$ to 51, $n-2 = 52$ to 53,

$n-2$	Square-maximal String
41	**aabaababaababaabaabaababaababaabaabaababaabaababb**
42	**aabaababaababaabaabaababaababaabaabaababaabaababb**b
43	**aabaababaababaabaabaababaababaabaabaababaabaababb**ab
44	**aabaababaababaabaabaababaababaabaabaababaabaababb**abb
45	**aabaababaababaabaabaababaababaabaabaababaabaababb**abba
46	**aabaababaababaabaabaababaababaabaabaababaabaababb**abbab **aabaababaababaabaabaababaababaabaabaababaabaababb**abbaa

Table 6.3: Some square-maximal strings for $n-2 = 41$ to 46

$n-d$	Square-maximal String
47	**ababbabbbabbbbabbbabbbbabbbbbabbbbabbbbbabbbbbabaa**
48	**ababbabbbabbbbabbbabbbbabbbbbabbbbabbbbbabbbbbabaa**a
49	**ababbabbbabbbbabbbabbbbabbbbbabbbbabbbbbabbbbbabaa**ba
50	**ababbabbbabbbbabbbabbbbabbbbbabbbbabbbbbabbbbbabaa**baa
51	**ababbabbbabbbbabbbabbbbabbbbbabbbbabbbbbabbbbbabaa**baab

Table 6.4: Some square-maximal strings for $n-2 = 47$ to 51

$n-d$	Square-maximal String
52	**aababbabbbabbabbbabbbbabbbabbbbabbbbbabbbbabbbbbabbbba**
53	**aababbabbbabbabbbabbbbabbbabbbbabbbbbabbbbabbbbbabbbba**b

Table 6.5: Some square-maximal strings for $n-2=52$ to 53

From those three tables we can see that all the square-maximal strings displayed contain the same prefix. The common prefix is marked by the bold font. For example, all the strings for $n-2=41$ to 46 contain the same prefix $aabaababaababaabaababaababaabaababaabaababb$.

The square-maximal string for given d and n may not unique. Table 6.6 also shows other square-maximal strings for $n-2=48,49$.

$n-d$	Square-maximal String
48	**ababbabbbbabbbbbabbbbabbbbbabbbbabbbabbbbabbbabbbb**
49	a**ababbabbbbabbbbbabbbbabbbbbabbbbabbbabbbbabbbabbbb**

Table 6.6: Some square-maximal strings for $n-2=48$ and 49

Again, we can see from this table that all the displayed square-maximal strings contain the same suffix, also denoted in bold. In particular, all the strings for $n-2=48$ and 49 contain the same prefix $ababbabbbbabbbbbabbbbabbbbbabbbbabbbabbbbabbbabbbb$.

Thus, it is a reasonable guess that if x is a square-maximal $(2,n)$-string, then xa, xb, ax or bx could also be square-maximal $(2,n+1)$-strings. We can ignore the case bx, since all the strings that we generate must start with a. Although such strings may not be really square-maximal strings for a given n, they might still help find a better lower bound $\sigma_2^-(n)$.

If we want to use the prefix and suffix construction to generate the next candidate string for the computation of L_2, there are several cases that need to be discussed.

- if the $(2,n)$-string x starts with aa, we could use xa and xb.

- if the $(2, n)$-string x starts with ab, we could use ax, xa and xb.

Denote by $P_2(n-1)$ the set of $(2, n-1)$-strings obtained by the above two rules, then we can set $\sigma_2^-(n) = \max\ \{\sigma_2(n-1),\ \underset{x \in L_2(n)}{max}\, s(x),\ \underset{x \in P_2(n)}{max}\, s(x)\}$.

Consider we already have a list of square-maximal $(2, n-1)$-strings. Every time we read one string from this list and generate the candidate $(2, n)$-string by using the above two rules. Then we could check the number of the squares in the candidate string, and update $\sigma_2^-(n)$. Note that the value for $\sigma_2(n)$ only can be $\sigma_2(n-1)$, $\sigma_2(n-1)+1$, or $\sigma_2(n-1)+2$. If $\sigma_2^-(n) = \sigma_2(n-1)+2$, we can terminate the program immediately. We could say $\sigma_2(n) = \sigma_2(n-1)+2$ and the candidate string is one of the square-maximal $(2, n)$-string. If we get $\sigma_2^-(n) = \sigma_2(n-1)+1$ after check all the possible candidate strings, then we need to check the string x in $L_2(n)$. If for some string $x \in L_2(n)$, $s(x) = \sigma_2(n-1)+2$, we could terminate the program and output the result. If we still get $\sigma_2^-(n) = \sigma_2(n-1)+1$, the brute force search need to be used. In real practice, most of time we could have obtained $\sigma_2^-(n) = \sigma_2(n-1)+1$ or $\sigma_2^-(n) = \sigma_2(n-1)+2$ when we build $P_2(n)$. So to compute $P_2(n)$ first will be a better choice. A pseudo-code of combined heuristics is also shown below:

6.4 Double Squares and their role

We have discussed several heuristics in order to find a better value $\sigma_d^-(n)$ to speed up the computations. As discussed previously, all the generated s-covers should yield dense strings. A double square may help us reduce the search space even further.

Definition 6.2 *A pair of primitively rooted squares (s, e, p) and (s, e', p') form a* double square *if $s' = s$, $p < 2p' < 2p$.*

- Step 1: Read a string from the list of square-maximal $(2, n-1)$-strings.
- Step 2: Using prefix or suffix construction, generate $(2, n)$-strings.
- Step 3: Count the number of squares in the new strings, and update $\sigma_2^-(n)$.
- Step 4: If $\sigma_2^-(n) = \sigma_2(n-1) + 2$, go to Step 8; if $\sigma_2^-(n) < \sigma_2(n-1) + 2$, go to Step 5.
- Step 5: Read the next string from the list of square-maximal $(2, n-1)$-strings. if no string can be read, go to Step 2; else go to Step 6.
- Step 6: Generate x which satisfied the rule of $L_2(n)$ with known $\sigma_2^-(n)$. If all the strings in $L_2(n)$ has been checked, go to Step 9.
- Step 7: Check the value of $s(x)$. if $s(x) = \sigma_2(n-1) + 2$, go to Step 8; if $\sigma_2^-(n) < \sigma_2(n-1) + 2$, go to Step 6.
- Step 8: $\sigma_2(n) = \sigma_2(n-1) + 2$, and output the result.
- Step 9: Update $\sigma_2^-(n)$ and run a brute force search, and output the result.

Figure 6.1: The improved computational framework in pseudo-code for $\sigma_2^-(n)$

In simple terms, a double square consists of two primitively rooted squares uu and UU starting at the same position with the smaller square being bigger than the generator of the larger square, i.e. $|U| < |uu| < |UU|$.

For example: *abaaba* is a prefix of a string $x =$ *abaababaabbabba*, and the squares *abaaba* and *abaababaab* both start at the same position 0. Thus we call the pair (*abaaba*, *abaababaab*) a double square.

Definition 6.3 *A* double square s-cover *is an s-cover whose first square is the larger square of a double square.*

Lemma 6.4 *Let a string x start with a double square (uu, UU). Then there are a primitive string t, a non-empty proper prefix c of t, and integers a and b so that* $uu = (t^a c)^2$ *and* $UU = (t^a c t^b)^2$.

Proof Since $|U| < |uu| < |UU|$, the first U overalaps with the second u. Let v denote this overlap. Then $u = v\bar{v}$ for some $\bar{v}$ and $U = uv$.

$U = uv = v\bar{v}v$ and $vU = vv\bar{v}v$. Since u is a prefix of $vU = vv\bar{v}v$, then u must be a prefix of $vv\bar{v}$, and so $u = v^m c'$ for some $m \geq 1$ and some prefix c' of v. c' must be a proper non-empty prefix, for otherwise u would not be primitive. Let t be the primitive root of v, i.e. t is primitive and $v = t^b$ for some $b \geq 1$ ($b = 1$ if v is primitive and hence its own primitive root). Then $u = t^{bm}c'$. Either c' is a proper non-empty prefix of t, or $c' = t^p c''$ for some $p \geq 1$ and some proper non-empty prefix c'' of t. In the former case, let $a = bm$ and let $c = c'$, in the latter case, let $a = bmp$ and $c = c''$. Therefore, $u = t^a c$ and $U = t^c v = t^c t^b$. It is clear that $a \geq b$. □

Lemma 6.5 *If a string x admits an s-cover that is not a double square s-cover, then $s(x) \leq \sigma_d(n-1) + 1$.*

Proof Let $\{S_j \mid 1 \leq j \leq m\}$ be the s-cover of $x = x[0..n-1]$ and let $y = x[1..n-1]$. Since S_1 is not a double square, $s(y) \geq s(x) - 1$. We also have $s(y) \leq \sigma_d(n-1)$. Thus $s(x) \leq \sigma_d(n-1) + 1$. □

Denote by $L^*_d(n)$ the set of (d,n)-strings that admit a double square s-cover and satisfy all the conditions described in section 6.2. Recall that

$$\sigma_2^-(n) = \max\{\sigma_2(n-1), \max_{x \in L_2(n)} s(x), \max_{x \in P_2(n)} s(x)\}$$

and

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

.

From Lemma 13 in [5] we have $\sigma_d(n) \leq \sigma_d(n-1) + 2$. Thus the value for $\sigma_d(n)$ only can be $\sigma_d(n-1)$, $\sigma_d(n-1) + 1$ or $\sigma_d(n-1) + 2$. There are two cases that need to be considered during the generation of the s-cover:

- $\sigma_d^-(n) = \sigma_d(n-1)$: We want to find a string x so that $s(x) > \sigma_d^-(n)$. We know that $\underset{x \in L_d^*(n)}{max} s(x) \leq \sigma_d(n-1)+1$. So $\sigma_d(n) = \sigma_d^-(n-1)$ or $\sigma_d(n) = \sigma_d^-(n-1)+1$. We still need to generate the set of all special s-covered (d,n)-strings.

- $\sigma_d^-(n) > \sigma_d(n-1)$: We want to find a string x so that $s(x) > \sigma_d^-(n)+1$. If the string does not admit a double square s-cover, then $s(x) \leq \sigma_d^-(n)+1$. Thus we only need to generate $L_d^*(n)$. Thus, in this situation we can generate a double square s-cover and due to Lemma 6.4 this is computationally way cheaper as for S_1 we only need to generate t and vary a, b, and c, rather than generating all possible generators for S_1.

6.5 Some details of the computational framework

The modified generation proceeds by extending a partially built s-cover in all possible ways. In the following, we discuss all the steps:

1. When $d = 2$

We obtain $\sigma_2^-(n)$ as described in the previous section. There are three cases to consider:

- Case 1: $\sigma_2^-(n) = \sigma_2(n-1) + 2$.

  Since $\sigma_2(n) \leq \sigma_2(n-1) + 2$, we have the value of $\sigma_2(n)$ and there is no need to compute any further. In addition, every square-maximal string must be in $L_2(n)$ or $P_2^*(n)$.

- Case 2: $\sigma_2^-(n) = \sigma_2(n-1) + 1$.

In this case $\sigma_2^-(n) > \sigma_2(n-1)$. We already obtained a string x so that $\sigma_2(x) = \sigma_2(n-1)+1$ either in $L_2(n)$ or $P_2^*(n)$. and we want to check whether there exists a string x' so that $s(x') = \sigma_d^-(n)+2$. By Lemma 6.5 we could use the double square to speed up the program: if we can generate a string with $s(x) = \sigma_2(n-1)+2$ by generating only double square s-covers, then $\sigma_2(n) = \sigma_2(n-1)+2$, otherwise, $\sigma_2(n) = \sigma_2(n-1)+1$.

- Case 3: $\sigma_2^-(n) = \sigma_2(n-1)$.

  In this case, we cannot use Lemma 6.5 to speed up the program. We still need to generate the set of all special s-covered $(2, n)$ -strings.

2. When $d > 2$

We obtain $\sigma_d^-(n)$ as described in the previous section. Sometimes we do not know all the required values $\sigma_{d-1}(n-1)$, $\sigma_{d-1}(n-2)$ and $\sigma_d(n-1)$ during the real computation. But we may still use some properties of $(d, n-d)$ table to find a suitable value for $\sigma_d^-(n)$. Denote $L_d^*(n)$ is the set of (d, n)-strings that admit a double square s-cover and satisfy all the conditions from the previous section.

There are two cases need to be discussed during the string generation:

- $\sigma_d^-(n) = \sigma_d(n-1)$: We want to find a string x with $s(x) > \sigma_d^-(n)$. However, we cannot use Lemma 6.5 to reduce the size of the search space. We need to generate the set of all special s-covered (d, n)-strings.

- $\sigma_d^-(n) > \sigma_d(n-1)$: The value of $\sigma_d(n)$ can only be $\sigma_d(n-1)+1$ or $\sigma_d(n-1)+2$. If we could find a string x with $s(x) = \sigma_d(n-1)+2$ by generating all double square s-covers, then $s(x) = \sigma_d(n-1)+2$, otherwise, $s(x) = \sigma_d(n-1)+1$.

Another aspect we need to mention: if we know only the value of $\sigma_{d-1}(n-2)$ and $\sigma_d(n-1)$, and $\sigma_{d-1}(n-2) = \sigma_d(n-1)$, then $\sigma_d^-(n)$ is at least equal to $\sigma_d(n-1)+1$.

We still can generate the double square s-covers for this case.

A pseudo-code of the modified version of the computational framework for using double square s-covers is shown in Figure 6.2.

- Step 1: Set the first square is $S_1 = (s_1, e_1, p_1)$, with $s_1 = 1$ and $p_1 = 1$. Let $j = 2$.
- Step 2: Fill the pattern of S_1, **and make sure it is a double square**, then count the frequency occurrence of each symbol.
- Step 3: Generate the next square $S_j = (s_j, e_j, p_j)$ by $s_j = s_{j-1} + 1$,
- Step 4: Fill the pattern of S_j.
- Step 5: Check the validity of S_j. If it is not valid, clear the pattern in S_j and go to Step 4. If the maximum length has been reached, go to Step 6, else go to Step 3 and increment $j = j + 1$.
- Step 6: Check the value of $s(x)$.
- Step 7: If all the strings have been generated, then go to Step 8, else update the value of $\sigma_d^-(n)$ and go to Step 2 with $p_1 = p_1 + 1$.
- Step 8: Output the value of $\sigma_d^-(n)$ as the value for $\sigma_d(n)$.

Figure 6.2: The computational framework using double square s-covers

Chapter 7

Computational results and discussion

In previous work, Deza, Franek and Jiang already found the values of $\sigma_2(n)$ for $n \le 53$ and $\sigma_3(n)$ for $n \le 41$ [5]. We presented an improved version of the computational framework in chapter 6. The improved framework was also implemented in `C++` as the original one and also incorporates the current work of Mei Jiang [5]. This code has been run on two desktops; one with AMD Phenom II*6 1055T and the other with Intel Quad core Q9550. Note that this is quite an achievement of the streamlining of the framework and the whole code as similar computations for runs performed by Andrew Baker [1] had to be performed on the facilities of the Shared Hierarchical Academic Research Computing Network (SHARCNET:www.sharcnet.ca).

7.1 Case when $d = 2$

We computed the values of $\sigma_d(n)$ till $n - 2 = 68$. The table 7.1 shows the square-maximal strings and the value of $\sigma_2(n)$ for $n - 2 = 52$ to 54. The up-to-date values can be found on the web at [20].

$n-d$	$\sigma_d(n)$	Square-maximal String
52	41	aababbabbbabbabbbabbbbabbbabbbbabbbbbabbbbabbbbbabbbba
53	42	aababbabbbabbabbbabbbbabbbabbbbabbbbbabbbbabbbbbabbbbab
54	43	aababbabbbabbabbbabbbbabbbabbbbabbbbbabbbbabbbbbabbbbaba

Table 7.1: Square-maximal strings for $n-d=52$ to 54

We used the same prefix to find the square-maximal strings for $n-2=52$ to 54, $n-2=55$ to 56, $n-2=58$ to 61, $n-2=62$ to 64 and $n-2=65$ to 68. We determined that $\sigma_2(n)=n-15$ when $63 \leq n \leq 68$ and $\sigma_2(n)=n-14$ when $58 \leq n \leq 62$, all slight improvements of the previous universal bounds.

7.2 Case when $d>2$

By using the modified version of computational framework as described in this thesis, we were able to compute some additional values of $\sigma_d(n)$; the results are shown in the following tables:

	$n-d$										
d	32	33	34	35	36	37	38	39	40	41	42
3	25	26	26	27	28	29	30	31	32	33	34

Table 7.2: $(d, n-d)$ table for $d=3$

	$n-d$										
d	24	25	26	27	28	29	30	31	32	33	34
4	19	20	21	22	22	23	24	25	25	26	27
5	19	20	21	22	23	23	24	25	26		
6	19	20	21	22	23	24					
7	19	20	21	22	23	24	25				

Table 7.3: $(d, n-d)$ table for $d = 4, 5, 6$ and 7

	$n-d$				
d	20	21	22	23	24
8	17	18			
9	17	18	19		
10	17	18	19	20	
11	17	18	19	20	21

Table 7.4: $(d, n-d)$ table for $d = 8, 9, 10$ and 11

We were also able to determine the values of $\sigma_{14}(21) = 19$, $\sigma_{15}(21) = 19$, $\sigma_{16}(21) = 19$, and $\sigma_{15}(22) = 20$.

7.3 Some interesting observations of the $(d, n-d)$ table

All the known values as of writing this thesis of the $(d, n-d)$ table are shown in [20]. Here we briefly summarize some interesting observations:

- First, we coloured the cells which cannot use the double square heuristics in $(d, n-d)$ table. Figure A.6 in Appendixes shows the colouring pattern when $n-d < 36$. Those coloured cells form several oblique lines when $n-d < 30$. Note that the number on each oblique line forms an arithmetic sequence with the consecutive terms equal to 1. For example, $\sigma_d(2d+1)$ line and $\sigma_d(2d+3)$ line. We could guess $\sigma_{12}(32) = 17$, and $\sigma_8(30) = 18$, etc.

The coloured pattern shows a little difference for $\sigma_3(35)$ and $\sigma_3(36)$. In both cases, we cannot use the double square heuristics. However, those two values were increased by 2 not by 1 on the oblique lines. We still can apply the double square heuristics when we compute $\sigma_4(37)$ and $\sigma_4(38)$.

If we only consider the value of $\sigma_d(n)$ with $n-d < 30$, we could guess those six lines following the rule of arithmetic sequence. But it requires a long time to compute the rest of blank cells.

- When $n - d < 30$, the uncoloured cells also form several oblique lines, and the numbers on each oblique line also form an arithmetic sequence with the consecutive terms equal to 1. For example, $\sigma_d(2d+2)$ line, $\sigma_d(2d+4)$ line. We may guess $\sigma_{16}(38) = 20$ and $\sigma_{16}(39) = 21$ if they follow the same rule.

- If $\sigma_d(n-1) = \sigma_{d-1}(n-2) = \sigma_{d-1}(n-1)$, then $\sigma_d(n) = \sigma_d(n-1)+1$. Most of cases happen at the left side of the coloured oblique line. If the values in the cell of the coloured oblique line always increase by 1 is true, and $\sigma_d(n-1) = \sigma_{d-1}(n-2) = \sigma_{d-1}(n-1)$ and $\sigma_{d-1}(n-1)$ has been coloured, then $\sigma_d(n+1) = \sigma_{d-1}(n-1)+1 = \sigma_{d-1}(n-2)+1$. Since $\sigma_d(n+1) \geq \sigma_d(n) \geq \sigma_d^-(n) = \sigma_{d-1}(n-2)+1$. Thus we could get $\sigma_d(n) = \sigma_{d-1}(n-2)+1$.

7.4 Discussion of future work

We presented the values of $\sigma_d(n)$ in a $(d, n-d)$ table which could help us to illustrate properties of the $\sigma_d(n)$ function. Then we explored some structural properties of square-maximal strings and combined those with Antoine, Franek and Jiang's work

[5] to narrow down the search space. Currently, we are trying to compute more results with bigger d and n. The use of the heuristics introduced in the previous chapter help us a lot to speed up the program. However, there still exist some cases that cannot use the heuristics to narrow down the search space, and the computation time for those cases is getting extremely large. For example, the computation time for $\sigma_4(36)$ is nearly 126 hours, while the computation time for $\sigma_4(37)$ is 28 seconds. This is caused by the fact that we could not use the double square s-cover when computing $\sigma_4(36)$, while we could use it for $\sigma_4(37)$. Thus, it is imperative to find some other heuristics to deal with those cases when the double square s-cover cannot be applied.

Recall the use of prefix and suffix construction for $d = 2$; there may exist some common string structures that can be used to generate square-maximal strings or to obtain a higher $\sigma_d^-(n)$. On the other hand, we know that when d increases, the complexity of generating the required strings increases exponentially. Hence, it may not be possible for such structures to exist.

Appendix A

Testing result for C_i with $i = 4$, to 8

The following tables shown the result for different (X, F) family. And the odd row record the value of C_i, the even row record the value of C_i with family $(X, \overline{F})$.

	$\|X\|$					
Pattern	8	9	10	11	12	13
0011001100	1.13574219	1.01171875	0.985961914	0.988203477	0.992839565	1.00856694
	1.10429955	1.01244998	0.990583271	0.990583271	0.993627459	1.00895621
1011001100	1.28100586	1.02922606	0.976899028	0.981060773	0.989830071	1.0064073
	1.21611595	1.02034855	0.97881791	0.982690584	0.990410786	1.0067391
00110011100		1.01172042	0.976899028	0.990705401	1.10362923	1.35268079
		1.01244831	0.97881791	0.989232015	1.10067049	1.35000702
1011001110		1.02952385	0.980664253	1.00323346	1.12227589	1.37098021
		1.02056575	0.97979483	1.00095846	1.11908785	1.36824149
0011001101			0.984628886	0.981060773	0.991888113	1.0354308
			0.988975167	0.982690584	0.991536526	1.03459036
0011001111			0.976501197	1.00323346	1.19816476	
			0.978155732	1.00095846	1.19411079	
1011001101			0.976501197	0.977259677	0.993698146	1.03879866
			0.978155732	0.978155732	0.9931526587	1.03790855

Continued on next page

	$\lvert X \rvert$					
Pattern	8	9	10	11	12	13
1011001111			0.981169432	1.02038656	1.2239581	
			0.980024934	1.01729667	1.21966597	
00110011001				0.985961914	0.98900395	1.00085492
				0.988204461	0.989584666	1.00091371
00110011101				0.992266204	1.12041712	1.44370894
				0.990662947	1.11722907	1.44071369
00110011011				0.979586568	0.993384663	1.0668259
				0.981080607	0.992817046	1.06566577
00110011111				1.00553703	1.22249763	1.87204384
				1.00313364	1.21820551	1.86784465
10110011001				0.979586568	0.9872274	1.00077168
				0.981080607	0.987603351	1.00077514
10110011101				1.00553703	1.1402746	1.46339556
				1.00313364	1.13685966	1.46033748
10110011011				0.976501197	0.996404774	1.07215569
				0.977260005	0.995641245	1.07094442
10110011111				1.02345503	1.25003951	1.90278896
				1.02023527	1.24550721	1.89851915
001100110001				0.992485575	1.00648551	
					0.993256112	1.00682031
001100111001					1.10501292	
					1.10203314	
001100110101					0.992113329	1.03998978
					0.991745492	1.03909068
001100110011					0.988723755	1.00073242
					0.989287314	1.00073528
001100111101					1.20015347	
					1.19607956	

Continued on next page

	$\|X\|$					
Pattern	8	9	10	11	12	13
001100111011					1.12188936	
					1.11868047	
001100110111					0.993672609	1.07351042
					0.993088943	1.07229016
001100111111					1.22457121	
					1.22025934	
101100110001					0.989549875	1.00455682
					0.990113435	1.00483451
101100111001					1.12374813	1.3831028
					1.12053924	1.38031746
101100110101					0.993986093	1.04353345
					0.993424555	1.04258504
101100110011					0.987013824	1.00087142
					0.987372418	1.00081864
101100111101					1.22603167	
					1.22171981	
101100111011					1.14190197	
					1.13846598	
101100110111					0.996760078	1.07911159
					0.995980298	1.0778399
101100111111					1.25227632	
					1.24772409	
0011001100001						1.00855966
						1.00894874
0011001101001						1.03547636
						1.03463513
0011001100101						1.0008548
						1.00091357

Continued on next page

	$\|X\|$					
Pattern	8	9	10	11	12	13
0011001100011						1.00648009
						1.00681472
0011001101101						1.06688342
						1.06572266
0011001101111						1.07356805
						1.0723472
1011001100001						1.00640188
						1.00673352
1011001101001						1.03884488
						1.037954
1011001100101						1.00077168
						1.00077514
1011001100011						1.00455143
						1.00482893
1011001101101						1.07221332
						1.07100146
1011001101111						1.07917456
						1.07790224

Table A.1: Testing result for C_4 with selected patterns

	$\lvert X \rvert$					
Pattern	8	9	10	11	12	13
0011001100	1.64294434	1.05883789	0.905601501	0.920398147	0.95145267	1.02806488
	1.4422704	1.06854648	0.937057157	0.937057157	0.956756186	1.03065581
1011001100	2.07093906	1.11152285	0.886526036	0.901127503	0.943189267	1.02107755
	1.64743435	1.0585006	0.900162114	0.912778417	0.947099479	1.02327373
00110011100		1.058838	0.886526036	0.948289626	1.30324776	2.13100996
		1.06854637	0.900162114	0.938713676	1.28343068	2.11213623
1011001110		1.11160141	0.908805624	0.995021358	1.37006841	2.20408733
		1.05850049	0.903623561	0.979794163	1.3484831	2.18463861
0011001101			0.902649402	0.901127503	0.959170491	1.11068933
			0.932256107	0.912778417	0.956848968	1.10503498
0011001111			0.885833698	0.995021358	1.59081704	
			0.897767649	0.979794163	1.56332478	
1011001101			0.885833698	0.890881679	0.963287215	1.11843094
			0.897767649	0.897767649	0.959747036	1.11247834
1011001111			0.910506458	1.06029295	1.68782318	
			0.903482151	1.03915152	1.65835795	
00110011001				0.905601501	0.939063275	1.00401167
				0.920428218	0.942974064	1.0044235
00110011101				0.958177668	1.36152484	2.43824039
				0.946974524	1.33989113	2.41683573
00110011011				0.891475171	0.962201681	1.20281799
				0.901335349	0.958357023	1.19506784
00110011111				1.00946286	1.68211997	4.01820908
				0.992648106	1.65259078	3.98612699
10110011001				0.891475171	0.934923535	1.00385899
				0.901335349	0.937475391	1.00390511
10110011101				1.00946286	1.43264499	2.51614085
				0.992648106	1.40927483	2.4941754

Continued on next page

	$\|X\|$					
Pattern	8	9	10	11	12	13
10110011011				0.885833698	0.970354167	1.21673035
				0.890892446	0.965263061	1.20865642
10110011111				1.08004683	1.78854166	4.16391977
				1.05727751	1.75699066	4.1310669
001100110001					0.950323635	1.02163258
					0.955509748	1.0238671
001100111001					1.30831564	
					1.28831394	
001100110101					0.959930845	1.1247456
					0.957507646	1.11866836
001100110011					0.938138485	1.00366306
					0.941934755	1.0037012
001100111101					1.59826003	
					1.57059729	
001100111011					1.36695465	
					1.34513805	
001100110111					0.963104275	1.22431807
					0.959160395	1.21612023
001100111111					1.68992303	
					1.66022495	
101100110001					0.942264477	1.01526468
					0.946060171	1.01710896
101100111001					1.37549822	2.24072002
					1.35373003	2.22093049
101100110101					0.964189809	1.13284167
					0.960550408	1.12646961
101100110011					0.934155635	1.00406393
					0.936590374	1.00373203

Continued on next page

	$\|X\|$					
Pattern	8	9	10	11	12	13
101100111101					1.69562623	
					1.66599211	
101100111011					1.4389729	
					1.41541379	
101100110111					0.971420595	1.23908756
					0.96622764	1.23056065
101100111111					1.79738575	
					1.7656595	
0011001100001						1.02806348
						1.03065427
0011001101001						1.11070933
						1.10505386
0011001100101						1.00401167
						1.00442349
0011001100011						1.02163163
						1.02386605
0011001101101						1.20284439
						1.19509312
0011001101111						1.22434447
						1.21614552
1011001100001						1.02107661
						1.02327268
1011001101001						1.11845101
						1.11249731
1011001100101						1.00385899
						1.00390511
1011001100011						1.01526373
						1.01710791

Continued on next page

	$\vert X\vert$					
Pattern	8	9	10	11	12	13
1011001101101						1.21675675
						1.2086817
1011001101111						1.23911808
						1.23058994

Table A.2: Testing result for C_5 with selected patterns.

	$\|X\|$					
Pattern	8	9	10	11	12	13
0011001100	3.75463867	1.29962158	0.764137268	0.831030083	0.860959751	1.07542521
	2.64658214	1.37267149	0.91637198	0.91637198	0.887034996	1.08844422
1011001100	5.15943325	1.4615983	0.744442503	0.787430871	0.844477544	1.05813184
	2.76985814	1.22335798	0.813008828	0.84871549	0.863689423	1.06909643
00110011100		1.2996216	0.744442503	0.906574674	1.72834927	4.10813343
		1.37267147	0.813008828	0.862400354	1.62617257	3.99641879
1011001110		1.46162837	0.817719245	1.03008148	1.92310213	4.37128719
		1.22335797	0.795091169	0.955976433	1.80951447	4.25469267
0011001101			0.760373532	0.787430871	0.899600397	1.27947767
			0.903738192	0.84871549	0.888356921	1.25061629
0011001111			0.744513802	1.03008148	2.45692938	
			0.805299098	0.955976433	2.30900079	
1011001101			0.744513802	0.765934053	0.906984454	1.2943036
			0.805299098	0.805299098	0.890137923	1.26408043
1011001111			0.823244758	1.2182199	2.76629258	
			0.791751134	1.11094514	2.60393576	
00110011001				0.764137268	0.833050913	1.01891786
				0.831520402	0.852239287	1.02107901
00110011101				0.95254279	1.89752947	5.0812162
				0.893452133	1.78360609	4.94981885
00110011011				0.744962024	0.90609014	1.49887112
				0.788981328	0.886900737	1.45908592
00110011111				1.09588979	2.75079098	10.8668026
				1.00733241	2.5878517	10.6379605
10110011001				0.744962024	0.826433963	1.01891688
				0.788981328	0.839050281	1.01926292
10110011101				1.09588979	2.1058764	5.36855646
				1.00733241	1.98066514	5.23217978

Continued on next page

	$\|X\|$					
Pattern	8	9	10	11	12	13
10110011011				0.744513802	0.923175255	1.52896967
				0.7661201	0.898167406	1.48756696
10110011111				1.31219744	3.10553143	11.5561778
				1.18969804	2.92727695	11.3184268
001100110001					0.85841642	1.06025944
					0.883906373	1.07150171
001100111001					1.74410978	
					1.64063532	
001100110101					0.901406649	1.31524438
					0.889694744	1.28400616
001100110011					0.830960512	1.01833653
					0.849587675	1.01862259
001100111101					2.48065632	
					2.33148075	
001100111011					1.9147145	
					1.79948977	
001100110111					0.908146649	1.55563566
					0.888505612	1.51320447
001100111111					2.77617861	
					2.61198368	
101100110001					0.842387143	1.04437561
					0.861037811	1.05359986
101100111001					1.94031438	4.478168
					1.82542538	4.35915372
101100110101					0.909040963	1.33065745
					0.891742797	1.2980783
101100110011					0.824624555	1.01944749
					0.836661449	1.01788614

Continued on next page

	$\|X\|$					
Pattern	8	9	10	11	12	13
101100111101					2.79168702	
					2.62807455	
101100111011					2.12743852	
					2.00082107	
101100110111					0.925542579	1.58801017
					0.900064071	1.54391142
101100111111					3.13672817	
					2.95709646	
0011001100001						1.0754249
						1.08844382
0011001101001						1.27948541
						1.25062276
0011001100101						1.01891786
						1.02107901
0011001100011						1.06025925
						1.07150145
0011001101101						1.49888194
						1.45909523
0011001101111						1.55564648
						1.51321378
1011001100001						1.05813164
						1.06909618
1011001101001						1.29431135
						1.26408692
1011001100101						1.01891688
						1.01926292
1011001100011						1.04437542
						1.05359961

Continued on next page

	$\lvert X \rvert$					
Pattern	8	9	10	11	12	13
1011001101101						1.52898049
						1.48757627
1011001101111						1.58802377
						1.54392327

Table A.3: Testing result for C_6 with selected patterns.

	$\lvert X \rvert$					
Pattern	8	9	10	11	12	13
0011001100	15.6404266	2.37059021	0.696735144	0.920512664	0.752163892	1.18718318
	8.27349231	2.80058627	1.28837345	1.28837345	0.857900071	1.24297615
1011001100	22.1876428	2.98765358	0.711478173	0.822064501	0.725645876	1.15071341
	6.82405159	2.01978382	0.997448181	1.09710481	0.804185414	1.19729334
00110011100		2.37059021	0.711478173	1.03019869	2.6289092	9.39849212
		2.80058627	0.997448181	0.88178303	2.17219888	8.75251054
1011001110		2.98767323	0.90870003	1.31557684	3.16582034	10.3807005
		2.01978381	0.850303513	1.02968145	2.63725011	9.6907159
0011001101			0.695896624	0.822064501	0.859414859	1.65001429
			1.25432295	1.09710481	0.818095197	1.52341927
0011001111			0.715527013	1.31557684	4.37059648	
			0.973946433	1.02968145	3.6526678	
1011001101			0.715527013	0.776602063	0.870780334	1.67659519
			0.973946433	0.973946433	0.808762628	1.54466694
1011001111			0.925617297	1.81970393	5.33644353	
			0.833299377	1.364722	4.51184068	
00110011001				0.696735144	0.705137931	1.0771821
				0.926017727	0.783941904	1.08673365
00110011101				1.2070841	3.08917633	12.6414473
				0.950258662	2.5621775	11.8344726
00110011011				0.686897717	0.876547311	2.13275819
				0.832096345	0.802127276	1.95416083
00110011111				1.56594785	5.28246286	35.8400854
				1.17350418	4.45688941	34.0837426
10110011001				0.686897717	0.696791459	1.0787578
				0.832096345	0.750273183	1.08070269
10110011101				1.56594785	3.67405025	13.7679673
				1.17350418	3.07389143	12.9135422

Continued on next page

	$\|X\|$					
Pattern	8	9	10	11	12	13
10110011011				0.715527013	0.906241085	2.19347162
				0.778801067	0.809988765	2.00762511
10110011111				2.20441979	6.47025741	39.4810632
				1.63011402	5.52452958	37.6117935
001100110001					0.747840965	1.15669522
					0.85125974	1.20480902
001100111001					2.67418484	
					2.2093656	
001100110101					0.862977484	1.73408593
					0.819908447	1.59569
001100110011					0.701641407	1.07745014
					0.778280095	1.07905684
001100111101					4.44286089	
					3.71613225	
001100111011					3.13994121	
					2.60458748	
001100110111					0.880459358	2.27149127
					0.804374506	2.07873465
001100111111					5.3628926	
					4.52814559	
101100110001					0.722149352	1.12319192
					0.798523606	1.16232659
101100111001					3.2169761	10.7127856
					2.68004203	10.0048569
101100110101					0.874692381	1.76153105
					0.811009858	1.61789974
101100110011					0.693704183	1.08103307
					0.74493175	1.07485364

Continued on next page

	$\|X\|$					
Pattern	8	9	10	11	12	13
101100111101					5.41697415	
					4.58319551	
101100111011					3.74489034	
					3.13495951	
101100110111					0.910647958	2.33787749
					0.812632312	2.13750287
101100111111					6.58222951	
					5.6254442	
0011001100001						1.18718309
						1.242976
0011001101001						1.65001746
						1.5234212
0011001100101						1.0771821
						1.08673365
0011001100011						1.15669516
						1.20480893
0011001101101						2.13276269
						1.9541637
0011001101111						2.27149578
						2.07873752
1011001100001						1.15071335
						1.19729325
1011001101001						1.67659836
						1.54466888
1011001100101						1.0787578
						1.08070269
1011001100011						1.12319186
						1.1623265

Continued on next page

	$\lvert X \rvert$					
Pattern	8	9	10	11	12	13
1011001101101						2.19347613
						2.00762798
1011001101111						2.33788387
						2.13750718

Table A.4: Testing result for C_7 with selected patterns.

	$\lvert X \rvert$					
Pattern	8	9	10	11	12	13
0011001100	128.253372	8.74427414	0.984892964	1.46973936	0.76409625	1.48263866
	47.0188292	9.89135676	2.9070464	2.9070464	1.1366751	1.69877705
1011001100	177.349806	11.698255	1.14482783	1.24099043	0.727891007	1.41279771
	30.5008442	6.02887988	2.06435626	2.32864316	1.00833879	1.59024969
00110011100		8.74427414	1.14482783	1.69418359	4.85224618	25.4785317
		9.89135676	2.06435626	1.23185136	2.94332529	21.5456009
1011001110		11.6982784	1.7355304	2.45188644	6.41986737	28.532693
		6.02887988	1.50976054	1.38407573	4.05119094	25.1784271
0011001101			0.998159357	1.24099043	1.01838178	2.56338501
			2.80795618	2.32864316	0.89485069	2.0498735
0011001111			1.16379499	2.45188644	9.2217146	
			1.9966276	1.38407573	5.84790891	
1011001101			1.16379499	1.15413189	1.03765705	2.6128971
			1.9966276	1.9966276	0.849579972	2.07523724
1011001111			1.7935309	4.04439457	12.4660823	
			1.44865945	2.08648501	8.28184404	
00110011001				0.984892964	0.711222611	1.32008891
				1.51783247	0.994210804	1.35141245
00110011101				2.46024615	6.17635356	37.4849793
				1.32820181	3.84976994	32.1670893
00110011011				1.02899109	1.07870786	3.66951922
				1.30599217	0.838105238	2.9061203
00110011111				3.49581873	12.2367316	142.863565
				1.73634595	8.07911251	128.610796
10110011001				1.02899109	0.703380536	1.32936735
				1.30599217	0.904478732	1.33195894
10110011101				3.49581873	7.94287674	41.4514504
				1.73634595	5.12729344	36.6509938

Continued on next page

	$\|X\|$					
Pattern	8	9	10	11	12	13
10110011011				1.16379499	1.12818721	3.79592895
				1.17518474	0.81883598	3.00117096
10110011111				5.76536453	16.6459974	165.224865
				2.93388737	11.4951773	149.405456
001100110001					0.759071059	1.42919586
					1.12380178	1.61362908
001100111001					4.98410401	
					3.02728062	
001100110101					1.0261544	2.7666588
					0.896880874	2.19228448
001100110011					0.707434748	1.33286925
					0.983357571	1.33314024
001100111101					9.45380211	
					6.01753665	
001100111011					6.32971522	
					3.95157077	
001100110111					1.08686359	4.02085095
					0.840857349	3.18334815
001100111111					12.5111951	
					8.28483629	
101100110001					0.724103144	1.36548379
					0.997485558	1.51257451
101100111001					6.57632804	29.6979321
					4.15575306	26.2014815
101100110101					1.045811278	2.81735814
					0.852332083	2.22411896
101100110011					0.700140914	1.34578013
					0.89385866	1.31648419

Continued on next page

	$\lvert X \rvert$					
Pattern	8	9	10	11	12	13
101100111101					12.7411016	
					8.48803641	
101100111011					8.19025662	
					5.30622818	
101100110111					1.13708601	4.16182279
					0.821945843	3.29056775
101100111111					17.0994919	
					11.8532381	
0011001100001						1.48263862
						1.69877695
0011001101001						2.56338671
						2.0449879
0011001100101						1.32008891
						1.3514245
0011001100011						1.42919584
						1.61362902
0011001101101						3.66952161
						2.90612112
0011001101111						4.02005334
						3.18334897
1011001100001						1.41279769
						1.59024963
1011001101001						2.6128988
						2.07523779
1011001100101						1.32936735
						1.33195894
1011001100011						1.36548377
						1.51257445

Continued on next page

	$\|X\|$					
Pattern	8	9	10	11	12	13
1011001101101						3.79593134
						3.00117178
1011001101111						4.16182661
						3.29056929

Table A.5: Testing result for C_8 with selected patterns.

	$n-d$																																
d	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	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
4	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	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	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	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	3	4	5	6	7	8	8	9	9	10	11	12	13	13	14	15	16	17	18														
9	3	4	5	6	7	8	9	9	10	10	11	12	13	14	14	15	16	17	18	19													
10	3	4	5	6	7	8	9	10	10	11	11	12	13	14	15	15	16	17	18	19	20												
11	3	4	5	6	7	8	9	10	11	11	12	12	13	14	15	16	16	17	18	19	20	21											
12	3	4	5	6	7	8	9	10	11	12	12	13	13	14	15	16	17																
13	3	4	5	6	7	8	9	10	11	12	13	13	14	14	15	16	17	18															
14	3	4	5	6	7	8	9	10	11	12	13	14	14	15	15	16	17	18	19														
15	3	4	5	6	7	8	9	10	11	12	13	14	15	15	16	16	17	18	19	20													
16	3	4	5	6	7	8	9	10	11	12	13	14	15	16	16	17	17	18	19														
17	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	17	18	18															
18	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	18	19															
19	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	19	20														
20	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	20														
21	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21														
22	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22													
23	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23												

Table A.6: the coloured $(d, n-d)$ table

Bibliography

[1] A. Baker. *Computational and Structural Approaches to Periodicities in Strings.* PhD thesis, McMaster University, 2013.

[2] D. Conlon. On the Ramsey multiplicity of complete graphs. *Combinatorica*, 32(2):171–186, 2012.

[3] M. Crochemore and W. Rytter. Squares, cubes, and time-space efficient string searching. *Algorithmica*, 13:405–425, 1995.

[4] A. Deza, F. Franek, and M. Jiang. A d-step approach for distinct squares in strings. In *Proceedings of the 22nd annual conference on Combinatorial pattern matching*, CPM'11, pages 77–89, Berlin, Heidelberg, 2011.

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

[6] A. Deza, F. Franek, and M. J. Liu. On a conjecture of Erdős for multiplicities of cliques. *Journal of Discrete Algorithms*, 17:9–14, 2012.

[7] P. Erdős. On the number of complete subgraphs contained in certain graphs. *Publications of the Mathematical Institute of the Hungarian Academy of Sciences*, 7:459–464, 1962.

[8] P. Erdős and J. W. Moon. On subgraphs on the complete bipartite graph. *Canadian Mathematical Bulletin*, 7:35–39, 1964.

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

[10] F. Franek. A note on Erdős' conjecture on multiplicities of complete subgraphs – lower upper bound for cliques of size 6. *Combinatorica*, 22(3):451–454, 2002.

[11] F. Franek, M. Jiang, and C.-C. Weng. An improved version of the runs algorithm based on crochemore's partitioning algorithm. In *Proceedings of Prague Stringology Conference 2011*, PSC'11, pages 98–105, 2011.

[12] F. Franek and V. Rödl. Disproving Erdős's conjecture on multiplicities of complete subgraphs using computer. Technical report, McMaster University, 1988.

[13] F. Franek and V. Rödl. Ramsey problem on multiplicities of complete subgraphs in nearly quasirandom graphs. *Graphs and Combinatorics*, 8:299–308, 1992.

[14] F. Franek and V. Rödl. 2-colorings of complete graphs with a small number of monochromatic k_4 subgraphs. *Discrete Mathematics*, 114:199–203, 1993.

[15] G. Giraud. Sur le problème de goodman pour les quadrangles etla majoration des nombres de ramsey. *Journal of Combinatorial Theory, Series B*, 27(3):237–253, 1979.

[16] A. W. Goodman. On sets of acquaintances and strangers at any party. *American Math Monthly*, 66:778–783, 1959.

[17] 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.

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

[19] C. Jagger, P. Šťovíček, and A. Thomason. Multiplicities of subgraphs. *Combinatorica*, 16:123–141, 1996.

[20] M. Jiang. Table of $\sigma_d(n)$ values: http://optlab.mcmaster.ca/jiangm5/ research/square.html/, 2012.

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

[22] A. F. Sidorenko. Cycles in graphs and functional inequalities. *Mathematical Notes*, 46:877–882, 1989.

[23] A. Thomason. A disproof of a conjecture of Erdős's in ramsey theory. *Journal of the London Mathematical Society*, 39(2):246–255, 1989.

[24] A. Thomason. Graph products and monochromatic multiplicities. *Combinatorica*, 17(1):125–134, 1997.