Semidefinite programming, binary codes and a graph coloring problem

by

Chao Li

A Thesis

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Master of Science

in

Computer Science

by

May 2015

APPROVED:

Professor William Martin, Major Thesis Advisor

Professor Craig Wills, Head of Department

Abstract

Experts in information theory have long been interested in the maximal size $A(n, d)$ of a binary error-correcting code of length n and distance d, The problem of determining $A(n, d)$ involves both the construction of good codes and the search for good upper bounds. It has been a long time that Delsarte's linear programming approach has been the dominant approach to obtain the strongest general purpose bound on the efficiency of error-correcting codes.

From 1973 to 2003, the linear programming bound found many applications, but there were few significant theoretical advances until Schrijver proposed a new code upper bound via semidefinite programming. Using the Terwilliger algebra, a recently introduced extension of the Bose-Mesner algebra, Schrijver formulated a new SDP strengthening of the LP approach.

In this project we look at the dual solutions of the semidefinite program bound, and explore the combinatorial meaning of these variables for small n and d, such as $n = 4$ and $d = 2$. To obtain information like this, we wrote a program with both MATLAB and CVX modules to get solution of our primal SDP formulation. Our program efficiently generates the primal solutions with corresponding constraints for any n and d. We also wrote a program in C++ to parse the output of the primal SDP problem, and another MATLAB script to generate the dual SDP problem, which could be used to examining combinatorial meaning of the difference between the dual solution of the dual SDP problem and the primal solution of the primal SDP problem. These values are very useful for later study of the combinatorial meaning of such solutions.

Acknowledgements

In this master's thesis project, I am very grateful to my thesis advisor, Prof. Martin. As a computer science student, I am very interested in graph theory and combinatorics, but I have no experience in doing research using algebra before. Since my lack of background of many knowledge used in this project, he had to teach me a lot from the beginning. He is very knowledgeable and always patient to me. I learned a lot which are not easily seen in the subjects of computer science which will direct my study for several years.

I am also great thankful to Prof. Hofri and Prof. Trapp. During my graduate study they advised me in different research projects respectively, which widen my knowledge in computer science and mathematics. They also provide important suggestions in my thesis presentation. Prof. Hofri is my acdemic advisor in computer science department, and he is my thesis reader as well. He gives my many suggestions from the beginning of my graduate study, and he gave great suggestions in both my thesis project and thesis report.

Contents

List of Figures

List of Tables

Chapter 1

Background and introduction

1.1 Problem introduction

Let $F = \{0, 1\}$ be the binary field. Then F^n is the set of all binary string with n bits. Define A *binary error-correcting code* C of length n and distance d is a collection of elements from F^n where $\partial(x, y) \geq d$, for all $x \neq y \in C$, in which $\partial(x, y)$ is the *Hamming distance* between codewords x and y. Two n-tuples x, y are at Hamming distance k if $x_i \neq y_i$ for exactly k values of i.

In real world, our communication systems are built based on error-correcting codes. To make the systems reliable and efficient, we always seek large codes with large minimum distance to fulfill this role. Hence our work is inspired by this requirement. We aim at finding the maximum size $A(n, d)$ of an error-correcting code C of length n and distance d. More specifically, in this project we study the upper bounds on $A(n, d)$.

Since Shannon, coding theorists have wondered what the optimal efficiency might be for codes of each given finite block length. The best known general purpose upper bound on the efficiency of a code is Delsarte's linear programming bound from 1973.

Delsarte used algebraic methods to study association schemes and applied these to coding theory. He formulated a linear programming model based on the Hamming scheme and the optimal solution gives the upper bound of $A(n, d)$.

Without solving the problem to optimality, but using some generating functions, McEliece, Rodemich, Rumsey and Welch at JPL found an asymptotic linear programming bound, which is a feasible solution that is valid for arbitrarily large n. This is called the "MRRW bound". As it is discussed in [6]:

Theorem 1.1. For any (n, M, d) code,

$$R \leq H_2\left(\frac{1}{2} - \sqrt{\frac{d}{n}\left(1 - \frac{d}{n}\right)}\right) \tag{1.1}$$

Gilbert and Varshamov gave an analytical function for the code size lower bound. As the following theorem mentioned in [6]:

Theorem 1.2. Suppose $0 \leq \delta < \frac{1}{2}$. Then there exists an infinite sequence of $[n, k, d]$ binary linear codes with $d/n \geq \delta$ and rate $R = k/n$ satisfying

$$R \geq 1 - H_2\left(\frac{d}{n}\right) \tag{1.2}$$

In the above theorems $H_2(x) = -x\log_2(x) - (1 - x)\log_2(1 - x)$ is the binary entropy function.

The two asymptotic bounds are shown in Figure 1.1. It is well-known that all codes lie on or below the McEliece-Rodemich-Rumsy-Welch upper bound, while the best codes lie on or above the Gilbert-Varshamov lower bound. For more information, we refer [6]. For the proof of this theorem we also refer to [6].

Around 2000, Alex Samorodnitsky proved in [7] that Delsarte's LP was not powerful enough to answer the question: there would always be a gap between the best random construction (asymptotically) and the best upper bound obtained via this method. Meanwhile, in the 1990s, two developments occurred in very different areas. Polynomial time algorithms for semidefinite programming were obtained and many applications of SDP emerged. Around the same time the (commutative) matrix algebra that Delsarte used to formulate his LP bound was extended by Terwilliger to a non-commutative semi-simple algebra for the n-cube which captures more detailed information about binary codes. These two ideas came togther in 2003, when Schrijver obtained a semidefinite programming bound for binary codes using the Terwilliger algebra [8]. Computationally, this was less than spectacular: computers can only handle SDPs for codes of length roughly 40 or less, and for these values, the improvements in the bounds were minor. Motivated by a problem in quantum information theory, de Klerk and Pasechnik applied the Schrijver technique to bound the size of a code in which every pair of codewords is at Hamming distance exactly $n/2$, encoded as the orthogonality graph. While they, too, could push the computer only to $n = 32$, their data suggests that the SDP bound might be exponentially better than the linear programming bound for this specific type of binary code. So the challenge is to find a "MRRW-style" bound for this problem using the Terwilliger algebra. Our work is mainly based on the work of Schrijver in [8] and of de Klerk and Pasechnik in [5].

Schrijver regards the Hamming scheme as another algebra's basis and uses another modeling technique to formulate the mathematical model of this problem and got a tighter upper bound than Delsarte's work. He applies the Terwilliger algebra of the Hamming scheme, and then uses the C^*-algebra structure to transform the problem size from exponential to polynomial. He formulated the semidefinite programming model for $A(n, d)$ and obtained a better upper bound than the linear programming bound.

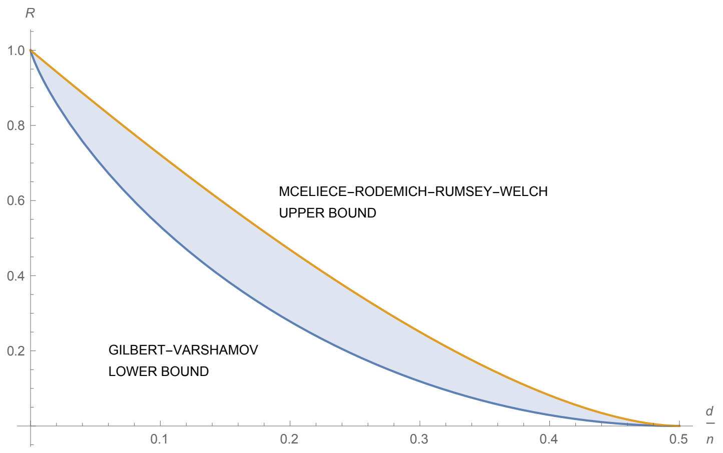

Figure 1.1: Asymptotic bounds on the best binary codes

1.2 Hamming graph and its adjacency matrix

Definition 1.1. A Hamming graph has the vertex set F^n and two vertices are adjacent if they differ in exactly one coordinate.

Our problem is restricted to binary codes so we will always look at binary codes in this paper. Recall that F^n is the set of all binary strings of length n where n is a positive integer. The Hamming graph $H(n, 2)$ has vertex set $X = F^n$. The d^{th} adjacency matrix of it is defined by

$$(A_d)_{xy} = \begin{cases} 1 & \text{if } \partial(x, y) = d \\ 0 & o.w. \end{cases}, \tag{1.3}$$

Figure 1.2 shows three Hamming graphs. Their corresponding adjacency matrices

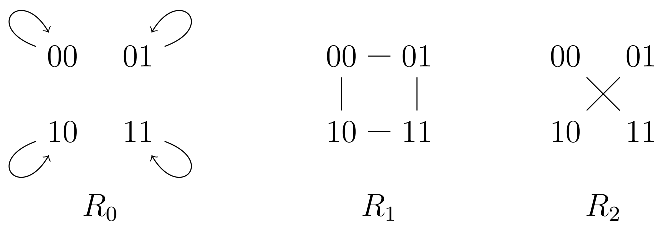

Figure 1.2: Hamming graph for $n = 2$ and $d = 0, 1, 2$

are shown in (1.4).

$$A_0 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad A_1 = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix} \quad A_2 = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \tag{1.4}$$

1.3 Association schemes

In this section, we introduce some basic ideas of association schemes. For more information, we recommend the elaboration in [6]. The definition of an association scheme is given as the following:

Definition 1.2. An *association scheme* with n classes (or relations) consists of a finite set X of together with $n+1$ relations $R_0, R_1, \ldots, R_n$ defined on X which satisfy:

1. Each R_i is symmetric: $(x, y) \in R_i \Rightarrow (y, x) \in R_i$.

2. For every $x, y \in X, (x, y) \in R_i$ for exactly one i.

3. $R_0 = \{(x, x) : x \in X\}$ is the identity relation.

4. If $(x, y) \in R_k$, the number of $z \in X$ such that $(x, z) \in R_i$ and $(y, z) \in R_j$ is a constant p_{ij}^k depending on i, j, k but not on the particular choice of x and y.

For simplicity, we often describe the relations by their adjacency matrices. Let A_i be the adjacency matrix of R_i (for $i = 0, \ldots, n$). Then it is a $v \times v$ matrix with rows and columns labeled by the points of X, defined by

$$(A_i)_{x,y} = \begin{cases} 1 & \text{if } (x,y) \in R_i \\ 0 & o.w. \end{cases} . \tag{1.5}$$

Then the 4 requirements of the definition of an association scheme can be rephrased as an A_i with $v \times v$ $(0,1)$-entries satisfying:

$$1) \quad A_i = A_i^{\intercal}, \text{ in which } A_i^{\intercal} \text{ denotes the transpose of } A_i. \tag{1.6}$$

$$2) \quad \sum_{i=0}^{n} A_i = J, \text{ in which } J \text{ denotes the all-ones matrix.} \tag{1.7}$$

$$3) \quad A_0 = I, \text{ in which } I \text{ denotes the identity matrix.} \tag{1.8}$$

$$4) \quad A_iA_j = \sum_{k=0}^{n} p_{ij}^k A_k = A_jA_i, \quad i,j = 0, \ldots, n \tag{1.9}$$

Now let's look at a simple example of an association scheme. Assume we have an association scheme with 3 classes, and our finite set X consists of 6 vertices. There relations are described as the "relation matrix" $\sum_{i=0}^{3} iA_i$ below:

$$\begin{array}{c|cccccc} & 1 & 2 & 3 & 4 & 5 & 6 \\ \hline 1 & 0 & 1 & 1 & 2 & 3 & 3 \\ 2 & 1 & 0 & 1 & 3 & 2 & 3 \\ 3 & 1 & 1 & 0 & 3 & 3 & 2 \\ 4 & 2 & 3 & 3 & 0 & 1 & 1 \\ 5 & 3 & 2 & 3 & 1 & 0 & 1 \\ 6 & 3 & 3 & 2 & 1 & 1 & 0 \end{array} \tag{1.10}$$

in which each color represents a relation, the 1 entries of D_i corresponding to relation R_i for $i = 0, \ldots, 3$, and the relations satisfy (1.6) - (1.9).

1.3.1 The Hamming schemes

After introducing the general association schemes, let us look at the Hamming schemes. Substituting the adjacency matrices A_i of relation R_i in the general definition of the association scheme with the adjacency matrices of the Hamming graphs, we will obtain the linear algebra representation of the Hamming schemes.
Let R_i be the set of ordered pairs $(a, b) \in X \times X$ with $\partial(a, b) = i$. Then $(X, \{R_d\}_{d=0}^{n})$ is a *symmetric association scheme* and its *Bose-Mesner algebra* is the vector space $\text{span}(A_0, A_1, \ldots, A_n)$ where A_i is the adjacency matrix of the graph (X, R_i).

1.4 The Bose-Mesner algebra

Once we have obtained the linear algebra representation of the Hamming schemes, we can examine the algebraic properties of it which will be introduced in the following sections. In this chapter we will introduce the algebraic properties we will use to get the upper bound on the code size, which is used to formulate the linear program by Delsarte. First, let us introduce some notation. For two nonempty finite set X_1 and X_2, we denote $\mathbb{C}(X_1, X_2)$ as the set of matrices M with dimension $|X_1| \times |X_2|$ over the complex field $\mathbb{C}$. We will denote by $M(x_1, x_2)$ the entry (x_1, x_2) of matrix M, in which $x_1 \in X_1$ and $x_2 \in X_2$.
Essentially due to Bose and Mesner, the following theorem is given to express an association scheme in an algebraic aspect.

Theorem 1.3. (Bose & Mesner [1])Let $R = \{R_0, R_1, \ldots, R_n\}$ be a set of $n + 1$ relations on a finite set X, satisfying (1.8). Define $\mathcal{A}$ to be a linear subspace of

$\mathbb{C}(X,X)$ generated by the adjacency matrix D_i of R_i, $i = 0, 1, \ldots, n$. Then (X, R) is an association scheme, with n classes, if and only if $\mathcal{A}$ is a commutative $(n+1)$-dimensional subalgebra of $\mathbb{C}(X,X)$, all of whose elements are normal matrices.

Formally the linear algebra

$$\mathcal{A} = \left\{ \sum_{t=0}^{n} \alpha_t D_t | \alpha_t \in \mathbb{C} \right\} \tag{1.11}$$

is called the *Bose-Mesner algebra* of the association scheme (X, R), where D_t represents the relation matrix of relation R_t. Figure 1.2 shows the usual basis of the Hamming scheme for $n = 2$ in graph form, which is easier for people to study and (1.4) is a representation in matrix form, which is easier to process by computers.

1.5 The graph coloring problem

1.5.1 Orthogonality graph

Observe that $(\pm)$1-vectors, u, v in $\mathbb{R}^n$ are orthogonal iff the corresponding binary vectors are at Hamming distance $n/2$. For example, for the following vectors $u = [1\ 1\ 1\ -1\ -1\ -1]$ and $v = [1\ 1\ -1\ -1\ 1\ 1]$, their corresponding binary vectors are $w = [1\ 1\ 1\ 1\ 1\ 1]$ and $\hat{u} = [0\ 0\ 0\ 1\ 1\ 1]$. The graph with all 01-tuples as vertices is called the *orthogonality graph* if $x \sim y$ iff $\partial(x, y) = \frac{n}{2}$. We denote this by $\Omega(n)$. We note that $\Omega(n)$ is k-regular for $k = \binom{n}{n/2}$. A k-regular graph is a graph with each vertex has the same number of neighbors, say k neighbors. In this definition we know that $k = \binom{n}{n/2}$. For example a 3-regular graph is shown as Figure 1.3:

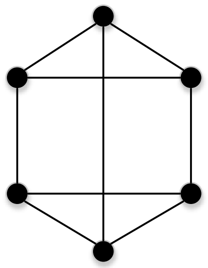

Figure 1.3: A 3-regular graph

1.5.2 A quantum information game

The orthogonality graph coloring problem is inspired from a quantum information game, expanding to classical bits.

Two players A and B are asked questions x_A and x_B, coded as n-bit rings satisfying $\partial(x_A, x_B) \in \{0, n/2\}$. A and B win the game if they give answers y_A and y_B, coded as binary string of length r such that $y_A = y_B \Leftrightarrow x_A = x_B$. Galliard et al. pointed out that whether or not the game can always be won is equivalent to the question

$$\chi(\Omega(n)) \leq r?$$

where $\chi(\Omega(n))$ means the number of colors needed to color $\Omega(n)$.

1.5.3 The graph coloring problem

We want to find the minimum number of colors required to color $\Omega(n)$ a priori, so that the two questions x_A and x_B are viewed as two vertices of $\Omega(n)$ and A and B answer their respective questions by giving the colors of the vertices x_A and x_B respectively, coded as binary string of length $log_2(n) = r$. If the two vertices have the same color, then they win.

1.6 Framework of this report

In this chapter we have introduced some basic background which will be used in the following chapters. In chapter 2, we will introduce the linear programming bound for code size. In chapter 3, we will introduce the basic idea of semidefinite programming and the following chapter gives the introduction of semidefinite programming bound by Schrijver and our implementation of the semidefinite programming formulation. In chapter 4, we also introduced the SDP bound applying to the graph coloring problem. Finally, chapter 5 will cover our studying of the dual SDP problem for the graph coloring problem.

Chapter 2

Linear programming bound for codes

In the previous chapter we have introduced the basic idea of the Bose-Mesner algebra of an association schemes. In this chapter, we will introduce the idea of using this algebra to obtain the linear programming bound for $A(n,d)$.

2.1 The characteristic vector

Recall that $X = F^n$ represents the collection of all binary strings with length n. If $C \subseteq X$ is a binary code, define x_C as the *characteristic vector* of C has one entry for each codeword $c \in F^n$, $x_c = 1$ if $c \in C$; $x_c = 0$ otherwise. For example, for $F^2 = \{00, 01, 10, 11\}$, a code $C = \{00, 01\}$ will have the characteristic vector $x_C = [1,1,0,0]^\intercal$. Given a Bose-Mesner algebra of a Hamming scheme

$$\mathcal{A} = \left\{ \sum_{i=0}^{n} \alpha_i A_i \quad | \quad \alpha_i \in \mathbb{C} \right\},$$

define

$$a_i := \frac{1}{|C|} x_C^\intercal A_i x_C$$

as the average number of codewords of distance i from $c \in C$. To see this, let's look at an example.

Let $n = 2$ and then we can obtain our basis of the Bose-Mesner algebra as

$$A_0 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad A_1 = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix} \quad A_2 = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}. \tag{2.1}$$

For $C = \{00, 01\}$, the characteristic vector is $x_C = [1, 1, 0, 0]^\intercal$. Let $B_i = x_C^\intercal A_i$. Then $B_0 = [1, 1, 0, 0], B_1 = [1, 1, 1, 1], B_2 = [0, 0, 1, 1]$. Let $d_i = x_C^\intercal A_i x_C$. Then $d_0 = 2, d_1 = 2$ and $d_2 = 0$, which gives the number of pairs of codewords at distance i. So $a_i = \frac{1}{|C|} x_C^\intercal A_i x_C$ gives the average number of codewords of distance i from $c \in C$, and the summation of a_i gives the code size $A(n, d)$. This holds because $\sum A_i = J$, the all ones matrix and $x_C^\intercal J x_C = |C|^2$.

2.2 Basis of orthogonal idempotents

An $n \times n$ complex matrix is *Hermitian* if $E^\dagger = E$ where $\dagger$ denotes conjugate transpose. A Hermitian matrix is positive semidefinite (PSD) if $x^\intercal E x \geq 0$ for all x. It is known that the Bose-Mesner algebra admits a basis of positive semidefinite matrices

$E_0, E_1, \ldots, E_d$ satisfying

$$E_j E_j = E_j$$
$$E_i E_j = 0 \tag{2.2}$$

which are known as the *Orthogonal Idempotents*. The following lemma and fact also holds for the orthogonal idempotents:

Lemma 2.1. The summation of all the primitive idempotents is identity matrix, say, $\sum E_i = I$

Fact 2.1. The rank of the j^{th} idempotent is $\text{rank}(E_j) = \binom{n}{j}$, and the eigenvalues are $0, 1$. Each E_j is positive semidefinite, $E_j \succeq 0$.

Geometrically, E_j represents orthogonal projection onto a maximal common eigenspace of $A_0, A_1, \ldots, A_n$. The orthogonal projection is unique. To see this, let's look at an example shown in Figure 2.1. $V = \{V_1, V_2, V_3\}$ is a vector space, and $U, P_U \in V$. $P \in \{V_1, V_2\}$ which is a subspace of V. The orthogonal projection from U onto space $\{V_1, V_2\}$ of U is P_U, which is unique. And then we have $E_j = U_j {U_j}^{\intercal}$ where columns of U_j form an orthonormal basis for the j^{th} eigenspace V_j. The change-of-basis matrix Q from the A_i to the E_i is unique and known as the second eigenmatrix given by $Q_{ij} = K_j(i)$ where K_k is the *Krawtchouk polynomial*

$$K_k(x) := \sum_{j=0}^{k} (-1)^j (q-1)^{k-j} \binom{x}{j} \binom{n-x}{k-j} \tag{2.3}$$

where $q = 2$ in our binary case.

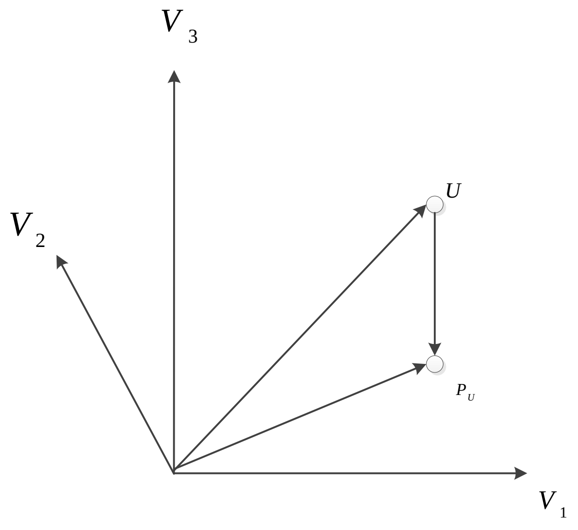

Figure 2.1: Orthogonal projection

Let's look at an example. For $n = 2$, define

$$\mathcal{A} = \left\{ \begin{bmatrix} a & b & b & c \\ b & a & c & b \\ b & c & a & b \\ c & b & b & a \end{bmatrix} : a, b, c \in \mathbb{C} \right\} \tag{2.4}$$

as the Bose-Mesner algebra of the Hamming scheme. $\mathcal{A}$ is a 3-dimensional linear space with the 01-basis indicated by a, b, c elements. Its corresponding basis of primitive idempotents is

$$E_0 = \frac{1}{4} \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix} \quad E_1 = \frac{1}{2} \begin{bmatrix} 1 & 0 & 0 & -1 \\ 0 & 1 & -1 & 0 \\ 0 & -1 & 1 & 0 \\ -1 & 0 & 0 & 1 \end{bmatrix} \quad E_2 = \frac{1}{4} \begin{bmatrix} 1 & -1 & -1 & 1 \\ -1 & 1 & 1 & -1 \\ -1 & 1 & 1 & -1 \\ 1 & -1 & -1 & 1 \end{bmatrix} \tag{2.5}$$

from Lemma 2.1 and Fact 2.1. Observe that E_0 represents orthogonal projection onto *span* $\{[1\ 1\ 1\ 1]\}$, E_1 represents orthogonal projection onto *span*$\{[1\ 1\ -1\ -1],$ $[1\ -1\ 1\ -1]\}$, and E_2 onto *span* $\{[1\ -1\ -1\ 1]\}$.

Lemma 2.2. For any $v \in \mathbb{R}^N, N = 2^n$, $v^\intercal E_j v \geq 0$

Proof.

$$\begin{aligned} v^\intercal E_j v &= v^\intercal (U_j {U_j}^\intercal) v \\ &= ({U_j}^\intercal v)^\intercal ({U_j}^\intercal) v \\ &= ||{U_j}^\intercal v||^2 \geq 0 \end{aligned}$$

□

A *linear character* of a finite abelian group G is a group homomorphism from G to the multiplicative group of nonzero complex numbers. In our case, consider $\chi : F^n \to \mathbb{C}$, as a character, satisfying $\chi(a+b) = \chi(a)\chi(b)$. Then the following fact holds:

Fact 2.2. There is one character for each binary n-tuple a defined as

$$\chi_a(b) = (-1)^{a \cdot b (\text{mod } 2)} \tag{2.6}$$

Fact 2.3. The characters χ of F^n give us an orthogonal basis of eigenvectors.

Define $wt(c)$ as the *Hamming weight* of codeword c which gives the Hamming distance between codeword c and the zero vector. Then the following lemma holds:

Lemma 2.3. $A_j \chi_c = Q_{ij} \chi_c$ for all j and c, where $i = wt(c)$ and $Q_{ij} = K_j(i)$.

Proof. From (2.6) we can obtain that entry a of $A_j\chi_c$ is

$$\sum_{\partial(a,b)=j} \chi_c(b) = \sum_{\partial(a,b)=j} (-1)^{b\cdot c}$$

let $b = a \bigoplus b'$, then

$$\begin{aligned}\sum_{\partial(a,b)=j} \chi_c(b) &= \sum_{wt(b')=j} (-1)^{(a+b')\cdot c} \\ &= (-1)^{a\cdot c}\left(\sum_{wt(b')=j} (-1)^{b'\cdot c}\right)\end{aligned}$$

$$\sum_{wt(b')=j} (-1)^{b'\cdot c} = \sum_{J\subseteq[n],|J|=j} \prod_{h\in J} (-1)^{c_h} \tag{2.7a}$$

where $c_h = 1$ iff $h \in J$, and J is the set of positions of bits if the bit is 1. By summing over h from 0 to i, we can change (2.7a) into

$$\sum_{wt(b')=j} (-1)^{b'\cdot c} = \sum_{h=0}^{i} (-1)^h \binom{i}{h}\binom{n-i}{j-h} \tag{2.8}$$

$\square$

We know that $A_iU_j = Q_{ij}U_j$, so $A_iE_j = (Q_{ij})E_j$, since

$$\begin{aligned}A_iI &= A_i(E_0 + E_1 + \cdots + E_n) \\ A_i &= Q_{i0}E_0 + Q_{i1}E_1 + \cdots + Q_{in}E_n\end{aligned}$$

As a fact $Q^2 = 2^nI$, $Q^{-1} = \frac{1}{2^n}Q$. Then we can get

$$E_j = \frac{1}{2^n}\sum_{i=0}^{n} Q_{ij}A_i \tag{2.9}$$

2.3 The linear programming formulation

Now let's derive the linear program formulation for finding the upper bound on the size of a binary code. Consider a binary code $C \subseteq F^n$ of length n. Recall that we have defined a_i as the average number of elements of C at distance i from b. More formally,

$$a_i = \frac{1}{|\mathcal{C}|} \cdot card\{(a,b) \in C \times C \mid \partial(a,b) = i\}$$

and

$$b_j = \frac{2^n}{|C|} x_C{}^{\intercal} E_j x_C \tag{2.10}$$

b is very similar with a, so we can check the properties of b_j by observing the above formulation, and get that

$$\begin{array}{llcl} \text{(I)} & b_0 & = & |C| \\ \text{(II)} & b_j & \geq & 0 \quad \text{for all } j \text{ since } E_j \succeq 0 \\ \text{(III)} & b_j & = & \sum_{i=0}^{n} Q_{ij} a_i (\geq 0) \end{array} \tag{2.11}$$

Now let us prove (III) of (2.11):

Proof. From the definition of b_j we can get the following derivation:

$$\begin{aligned} b_j &= \frac{2^n}{|C|} x_C{}^{\intercal} \left(\frac{1}{2^n} \sum Q_{ij} A_i \right) x_C \\ &= \frac{1}{|C|} \sum_{i=0}^{n} Q_{ij} \left(x_C{}^{\intercal} A_i x_C \right) \\ &= \sum_{i=0}^{n} Q_{ij} \left(\frac{1}{|C|} x_C{}^{\intercal} A_i x_C \right) \\ &= \sum_{i=0}^{n} a_i Q_{ij} \end{aligned}$$

□

So for any code C, the characteristic vector of C satisfies not only the condition on $x^\intercal A_i x$ imposed by its combinatorial properties but also $x^\intercal E_j x \geq 0$ for all $j = 0, \ldots, n$. The substitution $a_i = \frac{1}{|C|} x^\intercal A_i x$ gives rise to a linear programming formulation

$$\begin{aligned} \max \quad & \textstyle\sum_{i=0}^{n} a_i \\ \text{s.t.} \quad & aQ \geq 0 \\ & a \geq 0 \\ & a_0 = 1 \\ & a_1 = \cdots = a_{d-1} = 0 \end{aligned} \tag{2.12}$$

for the max size of a binary codes of minimum distance d. This was discovered by Delsarte in 1973 [2]. And the LP bound has been the strongest general purpose bound on the efficiency of error-correcting codes.

Let's look at a simple example for $n = 4$ and forbidding Hamming distance $d = 3$ using that formulation. For $n = 4$, we could obtain the second eigenmatrix Q using (2.3) as

$$Q = \begin{bmatrix} 1 & 4 & 6 & 4 & 1 \\ 1 & 2 & 0 & -1 & -1 \\ 1 & 0 & -2 & 0 & 1 \\ 1 & -2 & 0 & 2 & -1 \\ 1 & -4 & 6 & -4 & 1 \end{bmatrix} \tag{2.13}$$

from (2.12) we can obtain our linear programming formulation:

$$\begin{aligned}
\max \quad & a_0 + a_1 + a_2 + a_3 + a_4 \\
\text{s.t.} \quad & 4a_0 + 2a_1 - 2a_3 - 4a_4 \geq 0 \\
& 6a_0 - 2a_2 + 6a_4 \geq 0 \\
& 4a_0 - 2a_1 + 2a_3 - 4a_4 \geq 0 \\
& a_0 - a_1 + a_2 - a_3 + a_4 \geq 0 \\
& a_1, a_2, a_3, a_4 \geq 0 \\
& a_0 = 1
\end{aligned} \tag{2.14}$$

by substituting $a_0 = 1$ and $a_3 = 0$ we will get a simplified linear program:

$$\begin{aligned}
\max \quad & 1 + a_1 + a_2 + a_4 \\
\text{s.t.} \quad & 2a_1 - 4a_4 \geq -4 \\
& -2a_2 + 6a_4 \geq -6 \\
& -2a_1 - 4a_4 \geq -4 \\
& -a_1 + a_2 + a_4 \geq -1 \\
& a_1, a_2, a_3, a_4 \geq 0
\end{aligned} \tag{2.15}$$

Delsarte showed that the optimal objective value of this LP is 8 for $n = 4$ and $d = 3$ forbidden. And the code $C = \{0000, 0011, 0101, 1001, 0110, 1010, 1100, 1111\}$ achieves this bound.

2.4 Implementation of LP bound

To write a software suite capable of obtaining the LP bound for different values of codeword length n and minimum Hamming distance d, we used C++ to generate the eigenmatrix Q and then connect to IBM CPLEX to formulate the objective function and constraints iteratively. Our program is based on the following flowchart shown in Figure 2.2.

As the flowchart shows, the program starts by giving initial values of n and d.

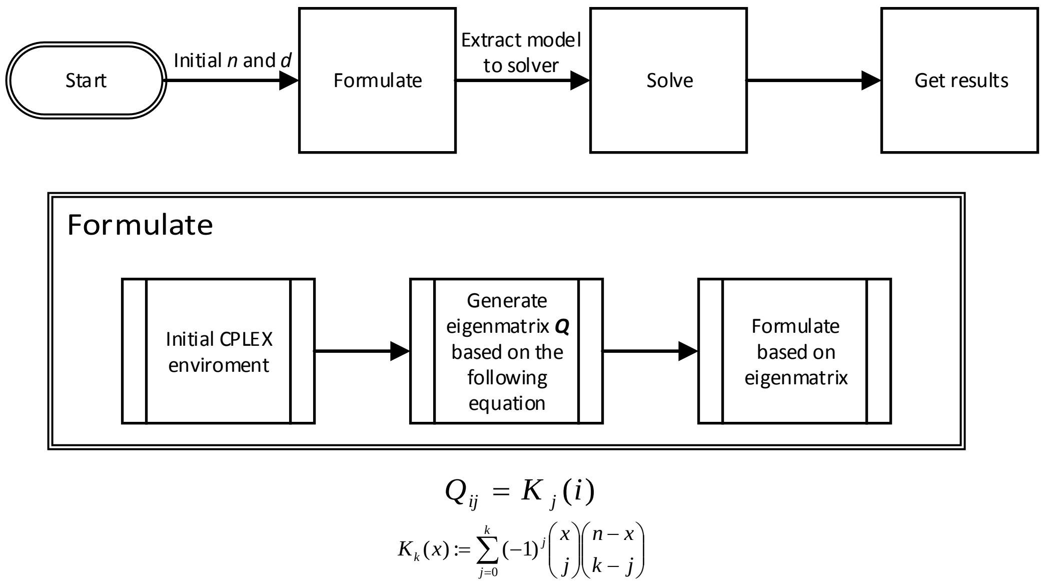
Start
Initial n and d
Formulate
Extract model to solver
Solve
Get results
Formulate
Initial CPLEX enviroment
Generate eigenmatrix Q based on the following equation
Formulate based on eigenmatrix
$Q_{ij} = K_j(i)$
$K_k(x) := \sum_{j=0}^{k} (-1)^j \binom{x}{j}\binom{n-x}{k-j}$

Figure 2.2: Flowchart of LP bound program

Based on the initial values, the formulate subprocess is invoked. Within the formulate subprocess, we first setup the CPLEX environment, which includes declaring the environment variables, initializing model variables, etc. Once the CPLEX environment is ready, we generate the eigenmatrix Q based on $Q_{ij} = K_j(i)$ where $K_k(x) := \sum_{j=0}^{k}(-1)^j \binom{x}{j}\binom{n-x}{k-j}$. Once the eigenmatrix is generated, we can build up the constraints with respect to the eigenmatrix column by column, since the fact that the entries of the first column of the eigenmatrix are all 1s and our problem is basically

$$\begin{aligned} \max \quad & aQ_0 \succeq 0 \\ \text{s.t.} \quad & aQ_j \qquad \forall j \neq 0 \\ & a_0 = 1 \\ & a_j \geq 0 \qquad \forall j \neq 0 \end{aligned} \tag{2.16}$$

where Q_j means the jth column of the eigenmatrix Q. After we finish building up our formulation, we export the model to the CPLEX solver and get the results of our model. Due to the limitation of the computability of today's computers and algorithmic packages, we only can get the accurate solution for relatively small values of n and d. It will overflow starting from $n = 32$ and $d = 2$. Some results are shown in the following table:

n	d	$A(n,d)$
19	2	262,144
19	3	26,214
19	4	13,107
19	13	3
20	2	524,287
20	3	47,662
20	4	26,214
20	13	3

Table 2.1: Some results for LP bound

As the results shown, $A(n,d) = 2^{n-1}$ when $d = 2$, $A(n,d) \approx \frac{2^{n-1}}{n+1}$ when $d = 3$ and $A(n,d) \leq 3$ when $d > 2n/3$. This can all be verified without the use of a computer by applying Delsarte's method.

Chapter 3

Semidefinite programming

Semidefinite programming is a new branch of conic programming, and our project is mainly based on a formulation using semidefinite programming. It searches for solutions on a section of a positive semidefinite cone. Because the semidefinite cone is convex, this is a convex optimization problem. A cone is a set C that for every $x \in C$ and $\theta \geq 0$ we have $\theta x \in C$ and for every $x, y \in C$, $x + y \in C$ also. A set C is a convex if for every $x_1, x_2 \in C$ and $\theta_1, \theta_2 \geq 0$ with $\theta_1 + \theta_2 = 1$, we have $\theta_1 x_1 + \theta_2 x_2 \in C$. Conic programming works in a Euclidean space, which is any vector space E, over $\mathbb{R}$ with positive definite inner product. An inner product $\langle \cdot, \cdot \rangle$ is positive definite if for all $x \in E$, $\langle x, x \rangle \geq 0$ and only equals 0 when x is a zero vector.

First let us look at two special cases of conic programming. A polyhedral cone is one with finitely many facets: $C = \{x : Bx \geq 0\}$ for some matrix B. The first one is linear programming where the vector space E is $\mathbb{R}^n$. Let C be a polyhedral cone and a general linear programming (LP) is:

$$\begin{aligned} \min \quad & C^\mathsf{T} x \\ \text{s.t.} \quad & Ax = b \quad x \in C, \end{aligned} \tag{3.1}$$

and the dual problem of it is

$$\begin{aligned} \max \quad & b^\intercal y \\ \text{s.t.} \quad & A^\intercal y + s = c \quad s \in C^*, \end{aligned} \tag{3.2}$$

where C^* denotes the *dual cone* of C.

The second one refers to our main tool used in this project, semidefinite programming. The basic form of a semidefinite programming problem is as the following:

$$\begin{aligned} \max \quad & \langle C, \chi \rangle \\ \text{s.t.} \quad & \langle A_i, \chi \rangle = b_i (1 \le i \le m) \\ & \langle B_j, \chi \rangle \le d_j (1 \le j \le k) \\ & \chi \succeq 0 \quad , \end{aligned} \tag{3.3}$$

where χ represents the semidefinite variable and $\chi \succeq 0$ represents the semidefinite cone constraint. $\langle C, \chi \rangle$ means the inner product of matrix C and matrix χ, which equals to the trace of the product of $C^\intercal$ and χ, say, $tr(C^\intercal \chi)$. In semidefinite programming, the cone we are using requires matrices to be positive semidefinite.

3.1 Background for semidefinite programming

Definition 3.1. A real symmetric matrix A is *positive semidefinite* if for all $v \in \mathbb{R}^n$, $v^\intercal A v \ge 0$, and it is *positive definite* if $\forall v \in \mathbb{R}^n$, when $v \ne 0$ then $v^\intercal B v > 0$.

To see this, let's look at some examples. Given three matrices A_1, A_2 and A_3 for

$$A_1 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 3 \end{bmatrix} \qquad A_2 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 0 \end{bmatrix} \qquad A_3 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & -3 \end{bmatrix}$$

Let's look at the positive definite properties of them. It is obvious that A_i are all real symmetric matrices. Define vector $v \in \mathbb{R}^n$. From definition 3.1 we need to check when $v \neq 0$, $v^\intercal A_i v$. Let $v = [v1\ v2\ v3]^\intercal$, and then

$$r_1 = v^\intercal A_1 v = v_1^2 + 2v_2^2 + 3v_3^2$$
$$r_2 = v^\intercal A_2 v = v_1^2 + 2v_2^2 + 0v_3^2$$
$$r_3 = v^\intercal A_3 v = v_1^2 + 2v_2^2 - 3v_3^2$$

It is obvious that for any $v \neq 0$, $r_1 > 0$, and for any $v \neq 0$, $r_2 \geq 0$. But r_3 could be any value for $v \neq 0$. Hence A_1 is positive definite. A_2 is positive semidefinite and A_3 is non-positive semidefinite. To introduce semidefinite programming, we need some background which could lead us to defining semidefinite matrices and understanding their properties.

Let A be a n by n symmetric matrix, if for some non-zero real number λ, and non-zero vector v, $Av = \lambda v$ holds, then λ is known as the *eigenvalue* associated with the *eigenvector* v.

If M is a n by n matrix over complex field, then define $M^\dagger$ as the conjugate transpose of matrix M. For example:

$$\begin{aligned} &\text{if } M = \begin{bmatrix} 1-i & i \\ 2 & 3 \end{bmatrix} \\ &\text{then } M^{\dagger} = \begin{bmatrix} 1+i & 2 \\ -i & 3 \end{bmatrix} \end{aligned} \tag{3.4}$$

M is called *Hermitean* if and only if $M = M^{\dagger}$. So intuitively we can imagine that when M is over real field, it is Hermitean if and only if M is symmetric.

Lemma 3.1. Every eigenvalue of any Hermitean matrix is real.

Proof. For Hermitean matrix M, assume there exists an eigenvalue λ associated with some non-zero vector v, then we have $Mv = \lambda v$.

Then we will have

$$\lambda\langle v, v\rangle = \lambda(v^{\intercal}v)$$

since λ is a scalar, we have

$$\begin{aligned} \lambda\langle v, v\rangle &= v^{\intercal}(\lambda v) \\ \lambda\langle v, v\rangle &= v^{\intercal}(Mv) \end{aligned}$$

since M is Hermitean, we have

$$\lambda\langle v, v\rangle = (v^{\intercal} M^{\dagger})v$$
$$\lambda\langle v, v\rangle = (vM)^{\intercal} v$$
$$\lambda\langle v, v\rangle = \langle Mv, v\rangle$$
$$\lambda\langle v, v\rangle = \langle \lambda v, v\rangle$$
$$\lambda\langle v, v\rangle = \bar{\lambda}\langle v, v\rangle$$

where $\bar{\lambda}$ means the complex conjugate of λ. So $\lambda = \bar{\lambda}$, which means every λ is real. □

Lemma 3.2. Assume we have $A \succeq 0$ and $B \succeq 0$, then $A + B \succeq 0$

Proof. Let $C = A + B$ and let $v \in \mathbb{C}^n$ then we have

$$C = A + B$$
$$v^{\dagger}Cv = v^{\dagger}Av + v^{\dagger}Bv \geq 0$$

□

3.2 Comparison between LP and SDP

Let's take a quick look at the relation between LP and SDP. A LP can be always transformed into a SDP by making the semidefinite variable a diagonal matrix. If we don't add constraints to force any entry of the semidefinite variable equals to 0, the SDP will be a relaxation to the LP problem. The general form of the two problems are like the following:

$$\begin{array}{llllll} \text{min/max} & C^{\intercal}x & \text{min/max} & \langle C,\chi\rangle & & \\ \text{s.t.} & Ax=b & \text{s.t.} & \langle A_i,\chi\rangle = b_i & (1\le i\le m) & \\ & Bx\ge d & & \langle B_j,\chi\rangle \le d_j & (1\le j\le k) & \\ & x\ge 0 & & \chi\succeq 0 & & \end{array} \tag{3.5}$$

Let's look at a very simple LP problem and its corresponding SDP problem. e.g.

$$\begin{array}{llll} \max & 3x_1-x_2+2x_3 & \max & \langle C,\chi\rangle \\ \text{s.t.} & 4x_1+x_2+x_3=8 & \text{s.t.} & \langle A,\chi\rangle = 8 \\ & 2x_2-x_3\le 9 & & \langle B,\chi\rangle \le 9 \\ & x_1,x_2,x_3\ge 0 & & \langle E_{12},\chi\rangle = \langle E_{13},\chi\rangle = \langle E_{23},\chi\rangle = 0 \quad , \end{array} \tag{3.6}$$

where

$$\begin{gathered} \chi = \begin{bmatrix} x_{11} & x_{12} & x_{13} \\ x_{12} & x_{22} & x_{23} \\ x_{13} & x_{23} & x_{33} \end{bmatrix} \succeq 0 \\ C = \begin{bmatrix} 3 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 2 \end{bmatrix} A = \begin{bmatrix} 4 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} B = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & -1 \end{bmatrix} \\ E_{12} = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} E_{13} = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} E_{23} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix} \quad . \end{gathered} \tag{3.7}$$

In the above example, the left side is the formulation of LP problem and the right side is its corresponding SDP formulation. To force the semidefinite variable be a diagonal matrix, we introduce three linear constraints $\langle E_{ij}, \chi\rangle = 0$ for $i \neq j$.
Because all the eigenvalues are real, we let $x_{11} = x_1$, $x_{22} = x_2$, $x_{33} = x_3$, and $x_{ij} = 0$, $\forall i \neq j$
then we have

$$\chi = \begin{bmatrix} x_1 & 0 & 0 \\ 0 & x_2 & 0 \\ 0 & 0 & x_3 \end{bmatrix} . \tag{3.8}$$

This example indicates how any LP can be formulated as an SDP, but the class of SDPs is much larger.
There is a fact in semidefinite matrices that every principal submatrix of a positive semidefinite matrix is positive semidefinite. To see the application of this fact, let's look at another example. Given a matrix

$$\chi = \left[\begin{array}{cc|cc} x_1 & 0 & 0 & 0 \\ 0 & x_2 & 0 & 0 \\ \hline 0 & 0 & \alpha & \beta \\ 0 & 0 & \beta & \gamma \end{array}\right] \succeq 0 \tag{3.9}$$

from the fact, we have to make every principal submatrix of matrix χ semidefinite. Choose subset S of rows and the same subset of columns, when $S = \{1, 4\}$, we can get the submatrix

$$\chi|_S = \begin{bmatrix} x & 0 \\ 0 & \gamma \end{bmatrix} \succeq 0$$

then we will have a constraint that $x\gamma \geq 0$.

When $S = \{3, 4\}$, we can get the submatrix

$$\chi|_S = \begin{bmatrix} \alpha & \beta \\ \beta & \gamma \end{bmatrix} \succeq 0$$

then we will have a constraint that $det\,(\chi|_S) = \alpha\gamma - \beta^2 \geq 0$ to force $\beta^2 - \alpha\gamma \leq 0$.

Let's look at a more specific example. Given the following semidefinite program:

$$\begin{aligned}
\max \quad & \left\langle \begin{bmatrix} 2 & 0 & 0 & 0 \\ 0 & 5 & 0 & 0 \\ 0 & 0 & 1 & -1 \\ 0 & 0 & -1 & 4 \end{bmatrix}, \chi \right\rangle \\
\text{s.t.} \quad & \left\langle \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix}, \chi \right\rangle = 10 \\
& \langle E_{ij}, \chi \rangle = 0 \\
& \chi \succeq 0
\end{aligned}$$

It is equivalent to the following non-linear program:

$$\begin{aligned}
\max \quad & 2x_1 + 5x_2 + \alpha - 2\beta + 4\gamma \\
\text{s.t.} \quad & x_1 + x_2 + \alpha + 2\beta + \gamma = 10 \\
& x_1 \geq 0, x_2 \geq 0, \alpha \geq 0, \gamma \geq 0 \\
& \beta^2 \leq \alpha\gamma
\end{aligned} \tag{3.10}$$

3.3 General form of dual semidefinite program

In this project we aim at the dual solutions of a semidefinite program. So let us look at the general form of a semidefinite program and its corresponding dual program. Considering the following general form of a semidefinite program:

$$\begin{aligned} \max \quad & \langle C, \chi \rangle \\ \text{s.t.} \quad & \langle A_i, \chi \rangle = b_i \qquad (1 \leq i \leq m) \\ & \langle B_j, \chi \rangle \leq d_j \qquad (1 \leq j \leq k) \\ & \chi \succeq 0 \end{aligned} \tag{3.11}$$

Then the general form of its corresponding dual problem is defined as the following form:

$$\begin{aligned} \min \quad & b^\intercal y + d^\intercal t \\ \text{s.t.} \quad & A^\intercal(y) + B^\intercal(t) \succeq C \\ & t \geq 0 \\ & \text{where } A^\intercal(y) = \textstyle\sum_{i=1}^{m} y_i A_i \\ & \text{ and } B^\intercal(t) = \textstyle\sum_{j=1}^{k} t_j B_j \end{aligned} \tag{3.12}$$

The transformation between the primal semidefinite program and its corresponding dual problem is very similar to the relationship between the linear program and its corresponding dual problem.

Now let's look at a special semidefinite program and its dual problem. Consider the following optimization problem. To adapt the general definition of semidefinite programming and its dual problem to our special case, we define the special case of

P-SDP and D-SDP as following:

$$\begin{aligned} \max \quad & \textstyle\sum_\alpha u_\alpha x_\alpha \\ \text{s.t.} \quad & \textstyle\sum_\alpha x_\alpha B_\alpha \succeq C \\ & x_\alpha \geq 0 \end{aligned} \tag{3.13}$$

Then its corresponding dual problem is in the following form:

$$\begin{aligned} \min \quad & \langle C, \chi \rangle \\ & \langle B_\alpha, \chi \rangle \leq -u_\alpha \\ & \chi \succeq 0 \end{aligned} \tag{3.14}$$

In general, we always have the following theorems for a semidefinite program:

Theorem 3.1. (Weak Duality Theorem) If X is feasible for the SDP and (y, t, Z) is feasible for the D-SDP, then $\langle C, X \rangle \leq y^\intercal a + t^\intercal b$.

Proof. First it is easy to check that for X and (y, t, Z) that are feasible, then $X \succeq 0$ and $Z \succeq 0$ means that $\langle Z, X \rangle \geq 0$.
By substituting C with the constraint in D-SDP in $\langle C, X \rangle$

$$\langle C, X \rangle = \langle \sum_{i=1}^{k} y_i A_i + \sum_{i=1}^{l} t_i b_i - Z, X \rangle = \sum_{i=1}^{k} y_i \langle A_i, X \rangle + \sum_{i=1}^{l} t_i \langle B_i, X \rangle - \langle Z, X \rangle$$

Since X and (y, t, Z) are both feasible, then the constraints hold, which means

$$\langle A_i, X \rangle = a \text{ and } \langle B_i, X \rangle \leq b$$

So we have

$$\langle C, X\rangle = \sum_{i=1}^{k} y_i \langle A_i, X\rangle + \sum_{i=1}^{l} t_i \langle B_i, X\rangle - \langle Z, X\rangle \le \sum_{i=1}^{k} y_i a_i + \sum_{i=1}^{l} t_i b_i - \langle Z, X\rangle$$

Additionally, we have $\langle Z, X\rangle \ge 0$, so we can conclude that:

$$\sum_{i=1}^{k} y_i a_i + \sum_{i=1}^{l} t_i b_i - \langle Z, X\rangle \le \sum_{i=1}^{k} y_i a_i + \sum_{i=1}^{l} t_i b_i = y^\intercal a + t^\intercal b$$

hence the theorem holds. $\square$

Theorem 3.2. (Complementary Slackness Theorem) If X is optimal for SDP, (y, t, Z) is optimal for D-SDP and $\langle C, X\rangle = y^\intercal a + t^\intercal b$ then

1. $Tr(ZX) = 0$
2. For every i, $1 \le i \le l$, $t_i = 0$ or $\langle B_i, X\rangle = b_i$

Proof. By substituting C with the constraint in D-SDP in $\langle C, X\rangle$

$$\langle C, X\rangle = \sum_{i=1}^{k} y_i \langle A_i, X, +\rangle \sum_{i=1}^{l} t_i \langle B_i, X\rangle - \langle Z, X\rangle$$

after some rearrangement, we can get

$$\langle Z, X\rangle = \sum_{i=1}^{k} y_i \langle A_i, X, +\rangle \sum_{i=1}^{l} t_i \langle B_i, X\rangle - \langle C, X\rangle$$

Since $\langle A_i, X\rangle = a_i$ and $\langle B_i, X\rangle \le b_i$, then we can get

$$\langle Z, X\rangle \le \sum_{i=1}^{k} y_i a_i + \sum_{i=1}^{l} t_i b_i - \langle C, X\rangle = y^\intercal a + t^\intercal b - \langle C, X\rangle.$$

Since the assumption that X and (y, t, Z) are optimal and $\langle C, X\rangle = y^\intercal a + t^\intercal b$, then

$\langle Z, X\rangle \leq y^{\intercal}a + t^{\intercal}b - \langle C, X\rangle = 0$, which means that $\langle Z, X\rangle \leq 0$.

Since $Z \succeq 0$ and $X \succeq 0$, then $\langle Z, X\rangle \geq 0$.

So $\langle Z, X\rangle = Tr(ZX) = 0$ holds.

Next since $\langle Z, X\rangle = 0$, we will have

$$\langle C, X\rangle = \sum_{i=1}^{k} y_i \langle A_i, X\rangle + \sum_{i=1}^{l} t_i \langle B_i, X\rangle$$

Since $B(X) \leq b$, then

$$\langle C, X\rangle = \sum_{i=1}^{k} y_i \langle A_i, X\rangle + \sum_{i=1}^{l} t_i \langle B_i, X\rangle \leq \sum_{i=1}^{k} y_i a_i + \sum_{i=1}^{l} t_i b_i = y^{\intercal}a + t^{\intercal}b,$$

which means the following equation must hold:

$$\sum_{i=1}^{k} y_i \langle A_i, X\rangle + \sum_{i=1}^{l} t_i \langle B_i, X\rangle = \sum_{i=1}^{k} y_i a_i + \sum_{i=1}^{l} t_i b_i$$

Since $\langle A_i, X\rangle = a_i$, it is obvious that

$$\sum_{i=1}^{k} y_i \langle A_i, X\rangle = \sum_{i=1}^{k} y_i a_i,$$

which gives us

$$\sum_{i=1}^{l} t_i \langle B_i, X\rangle = \sum_{i=1}^{l} t_i b_i,$$

from the constraints, we have $\langle B_i, X\rangle \leq b_i$, which gives us two cases:

1. if $\langle B_i, X\rangle < b_i$, then we must have $t_i = 0$ for all i to make the equality hold.

2. if $\langle B_i, X\rangle = b_i$, then the equality holds.

So the claim holds. □

Theorem 3.3. If both primal P-SDP and dual D-SDP have nonempty interiors, we also have *Strong Duality*: there exist feasible solutions to P-SDP and D-SDP satisfying equality in Theorem 3.1.

Chapter 4

Semidefinite programming bound for codes

4.1 Basis for Terwilliger algebra $\mathcal{T}$

Recall that we define the *Hamming weight* of codeword x as the Hamming distance between x and the zero vector, represented by $wt(x)$. Let α be a 4-tuple $(\alpha_0, \alpha_1, \alpha_2, \alpha_3)$ and L_α be a $F^n \times F^n$ matrix with

$$(L_\alpha)_{x,y} = \begin{cases} 1 & \text{if } wt(x) = \alpha_2 + \alpha_3, wt(y) = \alpha_1 + \alpha_3, \partial(x, y) = \alpha_1 + \alpha_2 \\ 0 & o.w. \end{cases} \tag{4.1}$$

for $x, y \in F^n$ and $\alpha \vdash n$, where a $F^n \times F^n$ matrix means the rows and columns of this matrix are both indexed by the elements of F^n, and $\alpha \vdash n$ means $\sum_{i=0}^{3} \alpha_i = n$ as well as $\alpha_i \geq 0$. More generally, a $U \times V$ matrix means the rows and columns of this matrix are indexed by the elements of U and V. α is isomorphic to a 3-tuple (i, j, t), where $wt(x) = i$, $wt(y) = j$ and $wt(x \oplus y) = t$. Obviously $\partial(x, y) = i + j - 2t$ and there will always exists an α in the triangle constituted by x, y and the zero

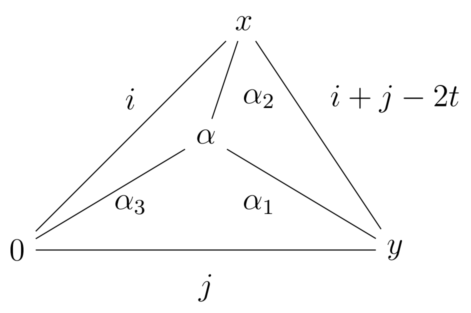

Figure 4.1: Relation between (i, j, t) and α

vector, satisfying the relationship shown in Figure 4.1. It is also obvious that L_α is a Hermitian matrix. Let $\mathcal{T}_n$ be the set of matrices:

$$\sum_\alpha x_\alpha L_\alpha, \quad \forall \alpha \vdash n \tag{4.2}$$

then $\mathcal{T}_n$ is a C^*-algebra because it is closed under addition, scalar and matrix multiplication, and taking the adjoint. Then linear space (4.2) is called the *Terwilliger algebra* of the Hamming scheme. The dimension of $\mathcal{T}_n$ is

$$dim(\mathcal{T}_n) = \binom{n+3}{3} \tag{4.3}$$

since we need to make $(L_\alpha)_{x,y} = 1$, and then we must have $\alpha \vdash n$. It equals to the number of ways we split an integer n into 4 integers, which is isomorphic to we change 3 balls into bars from $n + 3$ balls.

4.2 Block diagonalization

As $\mathcal{T}_n$ is a $C*$-algebra, and contains the identity matrix, there exists a unitary $F^n \times F^n$ matrix U and positive integers $p_0, q_0, \ldots p_m, q_m$ such that $U^\intercal \mathcal{T}_n U$ is equal to the collection of all block diagonal matrices

$$\begin{bmatrix} C_0 & 0 & \cdots & 0 \\ 0 & C_1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & C_m \end{bmatrix} \tag{4.4}$$

in which C_k is a block-diagonal matrix with q_k repeated, identical blocks of order p_k that

$$C_k = \begin{bmatrix} B_k & 0 & \cdots & 0 \\ 0 & B_k & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & B_k \end{bmatrix} \tag{4.5}$$

in which B_k is a $p_k \times p_k$ matrix. As $\mathcal{T}_n \cong \mathbb{C}^{p_0 \times p_0} \oplus \cdots \oplus \mathbb{C}^{p_m \times p_m}$, we define a mapping $\varphi : \mathcal{T}_n \to \bigoplus_k \mathbb{C}^{P_k \times P_k}$ where $P_k = \{x \in F^n : wt(x) = k\}$. Finally, by deleting repetitive of the block matrices, we will obtain an

$$\varphi(\mathcal{T}_n) \mapsto \begin{bmatrix} B_0 & 0 & \cdots & 0 \\ 0 & B_1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & B_m \end{bmatrix} \tag{4.6}$$

By giving the unitary matrix U required properties, we can obtain the specified values of the parameters for $p_k = n + 1 - 2k$, and $q_k = \binom{n}{j} - \binom{n}{k-1}$. $\sum_{j=0}^{m} p_j^2 = \dim(\mathcal{T}_n) = \binom{n+3}{3}$. To see the entries of the isomorphic block diagonal matrix to

(4.1), define

$$\beta_\alpha^k = \sum_{u=0}^{n} (-1)^{u-\alpha_3} \binom{u}{\alpha_3} \binom{n-2k}{u-k} \binom{n-k-u}{\alpha_2+\alpha_3-u} \binom{n-k-u}{\alpha_1+\alpha_3-u}. \tag{4.7}$$

then we will get the kth block matrix B_k in the image of matrix (4.6) is a $(n-2k+1) \times (n-2k+1)$ matrix: (4.1) is a $(n-2k+1) \times (n-2k+1)$ matrix:

$$\left(\sum_{\alpha \vdash n} \binom{n-2k}{\alpha_2+\alpha_3-k}^{-\frac{1}{2}} \binom{n-2k}{\alpha_1+\alpha_3-k}^{-\frac{1}{2}} \beta_\alpha^k x_\alpha \right)_{\alpha_2+\alpha_3=k, \alpha_1+\alpha_3=k}^{n-k} \tag{4.8}$$

4.3 Application to code size bound

First, let us define some mappings. Let $\alpha = (\alpha_0, \alpha_1, \alpha_2, \alpha_3)$, then define:

$$\begin{aligned} \check{\alpha}(\alpha) &= (\alpha_0+\alpha_2, 0, \alpha_1+\alpha_3, 0) \\ \acute{\alpha}(\alpha) &= (\alpha_0+\alpha_1, 0, \alpha_2+\alpha_3, 0) \\ \hat{\alpha}(\alpha) &= (\alpha_0+\alpha_3, 0, \alpha_1+\alpha_2, 0) \end{aligned} \tag{4.9}$$

Let Π be the set of all the automorphisms of F^n, Π_0 be the set of automorphisms π of F^n with $\emptyset \in \pi(C)$ and Π_1 be the set of automorphisms π of F^n with $\emptyset \notin \pi(C)$, where C is any code with $C \subseteq F^n$. Then we will have

$$\begin{aligned} \Pi_0 &= \{\pi \in \Pi : 0 \in \pi(C)\} \\ \Pi_1 &= \{\pi \in \Pi : 0 \notin \pi(C)\} \\ R &= \frac{1}{|\Pi_0|} \sum_{\pi \in \Pi_0} \chi_{\pi(C)} \chi_{\pi(C)}^{\mathsf{T}} \\ R' &= \frac{1}{|\Pi_1|} \sum_{\pi \in \Pi_1} \chi_{\pi(C)} \chi_{\pi(C)}^{\mathsf{T}} \end{aligned}$$

where for all $\pi \in \Pi$, $\pi(C) = \{\pi(c) : c \in C\}$.

For all R and R' the following properties hold:

1. R and R' are positive semidefinite matrices;
2. $R, R' \in \mathcal{T}_n$ where $\mathcal{T}_n = \{M \in \mathbb{C}^{2^n \times 2^n} | \forall \pi \in S_n (MP_\pi = P_\pi M)\}$.

following proposition holds:

Proposition 4.1. Let λ_α be the number of triples $(x, y, z) \in C^3$, where $\partial(x, y) = \alpha_2 + \alpha_3$, $\partial(x, z) = \alpha_1 + \alpha_3$, and $\partial(y, z) = \alpha_1 + \alpha_2$, define:

$$x_\alpha = \frac{1}{|C|\binom{n}{\alpha_0,\alpha_1,\alpha_2,\alpha_3}}\lambda_\alpha, \tag{4.10}$$

and $x_\alpha = 0$ when $\binom{n}{\alpha_0,\alpha_1,\alpha_2,\alpha_3} = 0$,

$$R = \sum x_\alpha L_\alpha \tag{4.11}$$

and

$$R' = \frac{|C|}{2^n - |C|}\sum_\alpha (x_{\hat{\alpha}} - x_\alpha) L_\alpha \tag{4.12}$$

Now we have got enough background to formulate the semidefinite program for obtaining the upper bound on the binary codes of code length n. Let $\alpha(0) = (n, 0, 0, 0)$. Then our objective function is

$$\max \quad |C| = \sum_\alpha \binom{n}{i} x_{\acute{\alpha}}, \tag{4.13}$$

since $|C|^2 = \sum_{\alpha \vdash n} \lambda_{\acute{\alpha}}, \forall \alpha \vdash n$. From (4.8) and the above equations (4.11) and (4.12) we can get the equivalent matrices:

$$\left(\sum_{\alpha \vdash n} \beta_\alpha x_\alpha \right)^{n-m}_{\alpha_2+\alpha_3=m, \alpha_1+\alpha_3=m}$$

and

$$\left(\sum_{\alpha \vdash n} \beta_\alpha (x_{\hat{\alpha}} - x_\alpha) \right)^{n-m}_{\alpha_2+\alpha_3=m, \alpha_1+\alpha_3=m} \tag{4.14}$$

are positive semidefinite, and along with the following constraints:

$$\begin{aligned}
&\text{I} \quad x_{\alpha(0)} = 1 \\
&\text{II} \quad 0 \leq x_\alpha \leq x_{\acute{\alpha}}, \forall \alpha \in \mathcal{T} \text{ and } x_{\acute{\alpha}} + x_{\breve{\alpha}} \leq 1 + x_\alpha, \forall \alpha \in \mathcal{T} \\
&\text{III} \quad x_\alpha = x_{\alpha'} \text{ if } (\alpha_1, \alpha_2, \alpha_3) \text{ is a permutation of } (\alpha_1', \alpha_2', \alpha_3') \\
&\text{IV} \quad x_\alpha = 0 \text{ if } \{\alpha_1+\alpha_2, \alpha_2+\alpha_3, \alpha_1+\alpha_3\} \cap \{1, \ldots, d-1\} \neq \emptyset
\end{aligned} \tag{4.15}$$

4.4 Implementating for the semidefinite programming bound

So far we have obtained all the information we need to formulate our semidefinite programming to obtain the upper bound for binary codes with length n. In this chapter let us introduce our implementation of the primal problem.

We use CVX, (please find [4] and [3]), connecting with MATLAB to formulate our semidefinite program. We will first introduce some basic idea of the CVX package and show you a simple demo which takes advantage of CVX package to model and solve a semidefinite programming problem. We also use the duality theorem of semidefinite programming to model the dual problem of our simple demo, which will show you the geometrical meaning of the dual solution from the dual variable, which is corresponding to the semidefinite cone of the primal problem.

4.4.1 Environment setup

First, we need to download the CVX package which is available from the official website (`http://cvxr.com/cvx/download`). We use Windows 64-bit operating system, so we choose the "cvx-w64.zip". Installation of the CVX package is fairly easy: after unzipping "cvx-w64.zip" to any directory of the file system, open MATLAB, and navigate to that directory. Type "cvx_setup" command to finish the setup process. The built-in default solver of CVX package is SDPT3, while it also supports some commercial solvers like Gurobi and MOSEK, which requires a professional license of CVX. In our experiments, we only use the default solver SDPT3. To change the solver being used, one can type the command "cvx_solver solver_name" in the MATLAB command line window. For example, if one has the professional license for CVX, by typing the command "cvx_solver mosek" could make the solver being used change to MOSEK.

4.4.2 Demo using MATLAB and CVX

Before introducing our implementation of the SDP bound, we would like to first give a demo of using MATLAB and CVX to model and solve a semidefinite programming problem, which could make the structure of a program solving SDP problem very clear.

Consider the following quadratic programming problem:

$$\begin{aligned} \max \quad & 2x_1 + x_2 \\ \text{s.t.} \quad & x_1 + x_2 \leq 10 \\ & x_1 \leq 9 \\ & x_1^2 + x_2 \leq 16 \\ & x_1, x_2 \geq 0 \end{aligned} \tag{4.16}$$

This can be transformed into a semidefinite programming problem taking the following form:

$$\begin{aligned} \max \quad & \langle C, X \rangle \\ \text{s.t.} \quad & \langle A_i, X \rangle \leq a_i, \quad i = 1, 2, 3, 4 \\ & \langle B_i, X \rangle = b_j, \quad j = 1, 2, 3, 4 \\ & X \succeq 0 \end{aligned} \tag{4.17}$$

where

$$A_1 = \begin{bmatrix} 0 & 0 & \frac{1}{2} \\ 0 & 1 & 0 \\ \frac{1}{2} & 0 & 0 \end{bmatrix} \quad A_2 = \begin{bmatrix} 0 & 0 & \frac{1}{2} \\ 0 & 0 & 0 \\ \frac{1}{2} & 0 & 0 \end{bmatrix} \quad A_3 = \begin{bmatrix} 0 & 0 & -\frac{1}{2} \\ 0 & 0 & 0 \\ -\frac{1}{2} & 0 & 0 \end{bmatrix} \quad A_4 = \begin{bmatrix} 0 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

$$B_1 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad B_2 = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad B_3 = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix} \quad B_4 = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$C = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix} \quad X = \begin{bmatrix} 1 & 0 & x_1 \\ 0 & x_2 & 0 \\ x_1 & 0 & 16 - x_2 \end{bmatrix}$$

and

$$\begin{aligned} a &= (10, 9, 0, 0) \\ b &= (1, 0, 0, 16). \end{aligned} \tag{4.18}$$

In MATLAB we first define the above matrices and vectors, and then begin to model the SDP problem by using the *cvx_begin sdp* keyword. To finish modeling we use the *cvx_end* keyword. Once the script hits the *cvx_end* keyword, the solver will be intrigued to solve the model. If the problem has feasible solution, it will output the optimal value and the dual optimal value. We can check the optimal solution in the MATLAB workspace environment. For example, our demo problem is formulated with the following MATLAB code:

```
cvx_begin sdp
  variable X(3,3) hermitian;
  maximize(trace(C'*X));
    dual variable Q;
    for i = 1:size(a,2)
      trace(A(:,:,i)'*X) <= a(i);
    end
    for j = 1:size(b,2)
      trace(B(:,:,j)'*X) == b(j);
    end
    X >= 0 : Q;
cvx_end
```

where A and B are the matrices defined as above, and X is the semidefinite cone. Q is the dual variable which could be obtained once the formulation is solved with a feasible solution. From the workspace of MATLAB, we can obtain the solution for X which is

$$X = \begin{bmatrix} 1 & 0 & 3 \\ 0 & 7 & 0 \\ 3 & 0 & 9 \end{bmatrix}.$$

Hence we can conclude that our solution is $x_1 = 3$ and $x_2 = 7$. To check the ence we can conclude that our solution is $x_1 = 3$ and $x_2 = 7$. To check the shown in Figure 4.2. It is pretty clear that point $(3, 7)$ is our optimal solution for this problem.

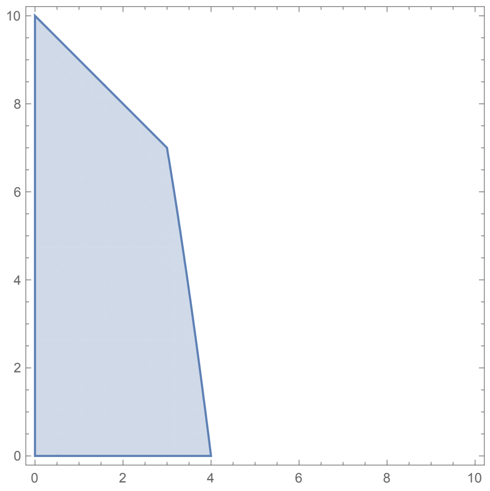

Figure 4.2: Feasible region of SDP demo

Now let us model the dual problem to check the correctness of the primal solution obtained as X from the primal problem. From (3.11) and (3.12) we can get the dual problem of our demo SDP as the following:

$$\begin{aligned} \min \quad & b^\intercal t + a^\intercal y \\ \text{s.t.} \quad & \textstyle\sum_{i=1}^{4} A_i y_i + \sum_{j=1}^{4} B_j t_j - C \succeq 0 \\ & y_i, t_j \geq 0 \qquad \text{for all } 1 \leq i, j \leq 4 \end{aligned} \tag{4.19}$$

where A, B, C, a and b are the same as the primal problem. Using the following MATLAB code we can model the dual semideifinite problem of our demo problem:

```
cvx_begin sdp
```

```
  variables y(4) t(4);
    expression obj;
    dual variable Q;
    for i = 1:size(a,2)
      obj = obj + a(i)*y(i);
    end
    for j = 1:size(b,2)
      obj = obj + b(j)*t(j);
    end
  minimize(obj);
    expression Z(3,3);
  subject to
      for i = 1:size(a,2)
        Z = Z+A(:,:,i)*y(i);
      end
      for j = 1:size(b,2)
        Z = Z+B(:,:,j)*t(j);
      end
      Z-C >= 0 : Q;
      Z == Z';
      y >= 0;
      t >= 0;
cvx_end
```

What differs from our code for primal SDP problem is that we use the keyword *expression* to represent the objective function and our semidefinite cone, since they are constituted iteratively by general variables. By solving the D-SDP, we can get the optimal solution which is 8, satisfying the weak duality theorem stating that the value of primal SDP is at least the value of the dual SDP. We can simply get the dual variable Q from the MATLAB workspace, which gives us that

$$Q = \begin{bmatrix} 1 & 0 & 4 \\ 0 & 0 & 0 \\ 4 & 0 & 0 \end{bmatrix}.$$

Comparing the dual variable of dual solution which gives us $x_1 = 4$ and $x_2 = 0$ with our primal solution where

$$X = \begin{bmatrix} 1 & 0 & 3 \\ 0 & 7 & 0 \\ 3 & 0 & 9 \end{bmatrix}.$$

which gives us the solution as $x_1 = 3$ and $x_2 = 7$, and check the region in Figure 4.2. We can know that the dual variable of our dual problem found the optimal solution as point $(4, 0)$. By substituting the (x_1, x_2) with $(4, 0)$ in the objective function of the primal problem, we can get our optimal value 8 of the dual problem. From Figure 4.2 we can know that there is some geometrical meaning in the dual solution, that is where we will look at the dual solution of the semidefinite programming for code size bound problem.

So far we have learned the tools we need to study the dual solutions of the semidefinite program of the code theory problem. Let us first study the primal solutions of this semidefinite program, which is obtained from (4.13) to (4.15). For full MATLAB implemetation please find Appendix B, while we will introduce the basic ideas behind it.

To simplify our semidefinite programming model, we only declare the distinct varibles and nonzero variables. So the constraint III and IV of (4.15) are evaluated during the modeling process. We will obtain the number of different variables, n_{var}. We assign all the variables $x_\alpha = 0$ the same index 1, and $x_{\alpha(0)}$ index n_{var}. So in our implementation, there are two equality constraints which are

$$x_1 = 0$$
$$x_{n_{var}} = 1 \tag{4.20}$$

All the other constraints with respect to II in (4.15) are generated iteratively.

Figure 4.3 shows our implementation of the SDP bound in MATLAB. Now let us

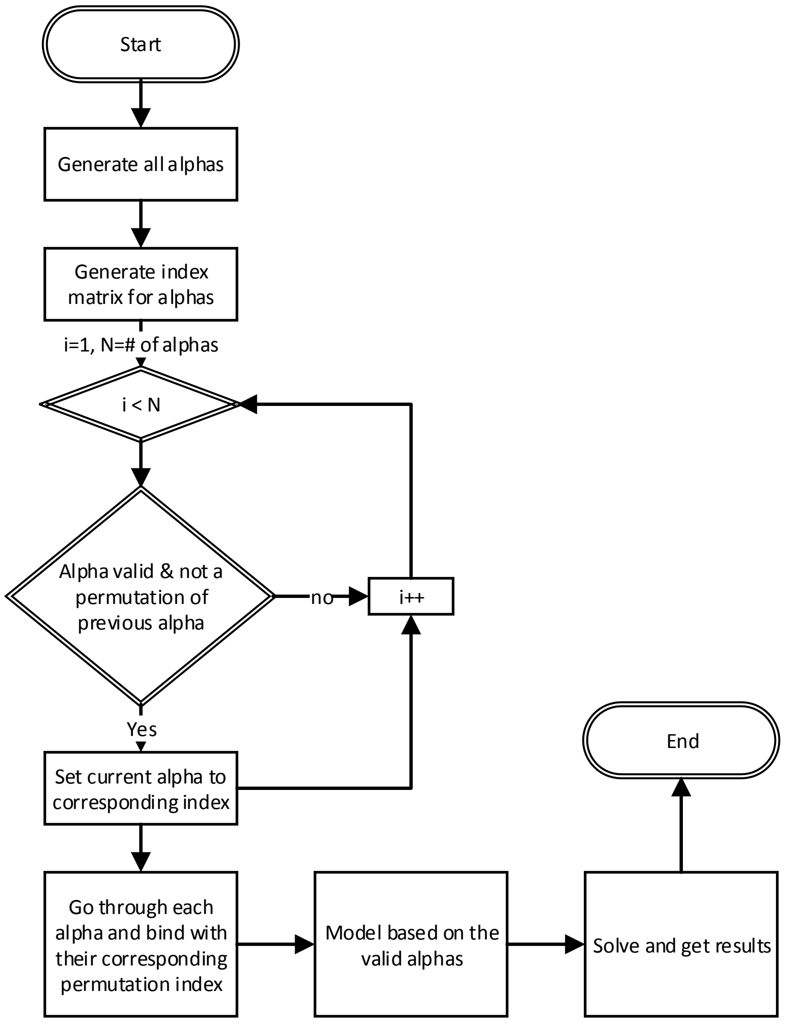
Start
Generate all alphas
Generate index matrix for alphas
i=1, N=# of alphas
i < N
Alpha valid & not a permutation of previous alpha
no
i++
Yes
Set current alpha to corresponding index
Go through each alpha and bind with their corresponding permutation index
Model based on the valid alphas
Solve and get results
End

Figure 4.3: Flowchart of SDP bound program

go through the process of implementing the SDP bound with MATLAB and CVX more in detail. As we can see in Figure 4.3, we start by generating all the possible αs. By using $\alpha_1, \alpha_2, \alpha_3$ and α_4 as the index we build up a index table for all the αs, and the entry refers to the name of the corresponding variable, say x_α. Next, we check each α to get rid of those that are not allowed due to IV in (4.15) as well as combine the αs according to the permutation constraint III in (4.15). Once we get all the valid αs, we can start building up our constraints and objective function following (4.13) and II in (4.15) iteratively. Our program takes two parameters as the initial state, n and d representing the length of codewords and the minimum Hamming distance. Now let us examine a small instance of the above semidefinite program, for $n = 4$ and minimum distance $d = 2$. The solution give us the optimal value is 8 and there are 5 xs equals to 1 and all the other variables equals to 0. For all the xs equals to 1 the corresponding λ_αs are shown below:

αs	x_α	$\lambda\alpha$
$(0,0,0,4)$	1	8
$(0,0,2,2)$	1	48
$(1,1,1,1)$	1	192
$(2,0,0,2)$	1	48
$(4,0,0,0)$	1	8

Table 4.1: Result of SDP bound for $n = 4$ and $d = 2$

Since $|C|^2 = \sum_{\alpha \vdash n} \lambda_{\acute{\alpha}}$, we can use this table to simply check that, $|C|^2 = \lambda_{(0,4,0,0)} + \lambda_{(4,0,0,0)} + \lambda_{(2,2,0,0)}$, which equals to 64. Hence our optimal value for the code upper bound is 8.

4.5 Applying SDP to the orthogonality graph coloring problem

In de Klerk and Pasechnik's work [5], Schrijver's semidefinite programming was applied to the orthogonality graph coloring problem; they obtained a very impressive bounds for that problem. By little modification of the MATLAB code of Schrijver's semidefinite programming for the code bound problem, we can obtain the formulation for the orthogonality graph coloring problem.

4.6 Reformulation

As they showed, the formulation only differed with the constraints of (4.15) in IV. De Klerk and Pasechnik change IV into

$$x_\alpha = 0 \text{ if } \{\alpha_1+\alpha_3, \alpha_2+\alpha_3, \alpha_1+\alpha_2\} \cap \{\frac{1}{2}n\} \neq \emptyset$$

Hence the constraints of the orthogonality graph problem would be

$$\begin{array}{rl} \text{I} & x_{\alpha(0)} = 1 \\ \text{II} & 0 \leq x_\alpha \leq x_{\acute{\alpha}}, \forall \alpha \in \mathcal{T} \text{ and } x_{\acute{\alpha}} + x_{\check{\alpha}} \leq 1 + x_\alpha, \forall \alpha \in \mathcal{T} \\ \text{III} & x_\alpha = x_{\alpha'} \text{ if } (\alpha_1,\alpha_2,\alpha_3) \text{ is a permutation of } (\alpha'_1,\alpha'_2,\alpha'_3) \\ \text{IV} & x_\alpha = 0 \text{ if } \{\alpha_1+\alpha_2, \alpha_2+\alpha_3, \alpha_1+\alpha_3\} \cap \{\frac{1}{2}n\} \neq \emptyset. \end{array} \tag{4.21}$$

The MATLAB code for modeling the orthogonality graph coloring problem is very similar to that for modeling the code size problem. Now let us go through the process of the MATLAB code for this problem. Same as the MATLAB code to solve the code size problem, we still start from creating the index matrix for all the αs. What differs from with before is that, after building up the index matrix, we don't check the minimum Hamming distance as an evaluation for the permission of an α. Instead we check IV in (4.21), which means we check whether the Hamming distance is exactly equal to $\frac{1}{2}n$. Then the permutation checking is same as before. After the index matrix is built up, we use the same process to formulate the constraints iteratively as we did in the code size problem. Table 4.2 shows some computational results produced by our program, which matches de Klerk and Pasechnik's results shown in [5]. Because of the computational limitation of MATLAB and CVX, we cannot get results for some relatively large n, say $n > 20$. But our purpose is to produce results for relatively small instances and try finding the pattern behind them. Let's see some results as shown in following table, in which $\alpha(n)$ denotes the upper bound of the size of set of orthogonality graphs:

n	$\alpha(n)$
8	32
12	268
16	2304
20	20167

Table 4.2: Result of SDP bound for $\Omega(16)$ and $\Omega(20)$

Chapter 5

Dual SDP for the graph coloring problem

What we are really interested is finding the difference between the optimal solution of the primal SDP formulation and the its dual problem the for this graph coloring problem. Now we have introduced everything we need to formulate the dual SDP formulation for the problem. Let's start with looking at a small instance for the coloring problem say coloring $\Omega(4)$. Then we will go into generate the dual SDP problem from the primal SDP problem automatically, since to study the pattern of the difference between the optimal solution for the primal problem and its dual, we need to look at some larger scale problems. Since the complextiy of this SDP formulation, it is necessary to generate the dual problem automatically to avoid any possible errors to produce correct solutions.

5.1 Dual problem of coloring $\Omega(4)$

Recall that we have seen the P-SDP and D-SDP formulations in chapter 3 and a special form to our special case. THe P-SDP of the graph coloring problem is shown as follows:

$$\begin{aligned} \max\quad & \textstyle\sum_\alpha u_\alpha x_\alpha \\ \text{s.t.}\quad & \textstyle\sum_\alpha x_\alpha B_\alpha \succeq C \\ & x_\alpha \geq 0. \end{aligned}$$

Then its corresponding dual problem is in the following form:

$$\begin{aligned} \min\quad & \langle C, \chi\rangle \\ & \langle B_\alpha, \chi\rangle \leq -u_\alpha \\ & \chi \succeq 0. \end{aligned}$$

To simplify the work, we need to change our formulation into a single positive semidefinite constraint. In our case, we have a positive semidefinite constraint as well as some linear constraints, so we append the linear constraints to the end of the semidefinite matrix as a single element block matrix. After obtaining our semidefinite constraints, by splitting the matrix into the sum of variables multiplied by their corresponding coefficient matrices, we can get all the elements we need to build up the dual. First let's formulate the semidefinite condition. From (4.14), we can get the block diagonal matrices B and B' for $\Omega(4)$. The blocks are

$$
\begin{aligned}
B_1 &= \begin{bmatrix} x_6 & 4x_5 & 6x_1 & 4x_4 & x_2 \\ 4x_5 & 12x_1+4x_5 & 24x_1 & 4x_3+12x_1 & 4x_3 \\ 6x_1 & 24x_1 & 36x_1 & 24x_1 & 6x_1 \\ 4x_4 & 4x_3+12x_1 & 24x_1 & 12x_1+4x_4 & 4x_3 \\ x_2 & 4x_3 & 6x_1 & 4x_3 & x_2 \end{bmatrix} \\
B_2 &= \begin{bmatrix} -x_1+x_5 & 0 & -x_3+x_1 \\ 0 & 0 & 0 \\ -x_3+x_1 & 0 & -x_1+x_4 \end{bmatrix} \\
B_3 &= \begin{bmatrix} 0 \end{bmatrix} \\
B'_1 &= \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 4x_6-4x_5 & 12x_4+12x_5-24x_1 & 4x_2-4x_3 & 4x_4-4x_3 \\ 0 & 12x_4+12x_5-24x_1 & 6x_2+6x_6-12x_1 & 12x_4+12x_5-24x_1 & 0 \\ 0 & 4x_2-4x_3 & 12x_4+12x_5-24x_1 & 4x_6-4x_4 & 4x_5-4x_3 \\ 0 & 4x_4-4x_3 & 0 & 4x_5-4x_3 & x_6-x_2 \end{bmatrix} \\
B'_2 &= \begin{bmatrix} x_6-x_5 & 2x_5-2x_4 & x_3-x_2 \\ 2x_5-2x_4 & 2x_6-2x_2 & 2x_5-2x_4 \\ x_3-x_2 & 2x_5-2x_4 & x_6-x_4 \end{bmatrix} \\
B'_3 &= \begin{bmatrix} x_2+x_6-2x_1 \end{bmatrix}.
\end{aligned}
\tag{5.1}
$$

From (4.13) we can obtain our objective function:

$$\max \qquad x_6+4x_5+6x_1+4x_4+x_2 \tag{5.2}$$

where $x_2 = x_{(0,0,0,4)}$, $x_3 = x_{(0,0,1,3)}$, $x_4 = x_{(1,0,0,3)}$, $x_5 = x_{(3,0,0,1)}$, $x_6 = x_{(4,0,0,0)}$ and x_1 represents all the other x_αs, which are all equal to zero. From x_2 to x_6, they also represents the x_αs if the α has the permutation on α_1, α_2 and α_3 to the corresponding α. From II of (4.21), we can obtain a bunch of linear constraints, each of which we append to our semidefinite cone as a diagonal matrix . There are 13 different linear constraints in total, so together with B and B', they constitute a semidefinite cone of dimension 31×31.

By substituting $x_6 = 1$ and $x_1 = 0$ into our semidefinite cone, we can get our block diagonal matrices B and B' as

$$\begin{aligned}
B_1 &= \begin{bmatrix} 1 & 4x_5 & 0 & 4x_4 & x_2 \\ 4x_5 & 4x_5 & 0 & 4x_3 & 4x_3 \\ 0 & 0 & 0 & 0 & 0 \\ 4x_4 & 4x_3 & 0 & 4x_4 & 4x_3 \\ x_2 & 4x_3 & 0 & 4x_3 & x_2 \end{bmatrix} \\
B_2 &= \begin{bmatrix} -x_5 & 0 & -x_3 \\ 0 & 0 & 0 \\ -x_3 & 0 & -x_4 \end{bmatrix} \\
B_3 &= \begin{bmatrix} 0 \end{bmatrix} \\
B'_1 &= \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 4-4x_5 & 12x_4+12x_5 & 4x_2-4x_3 & 4x_4-4x_3 \\ 0 & 12x_4+12x_5 & 6x_2+6 & 12x_4+12x_5 & 0 \\ 0 & 4x_2-4x_3 & 12x_4+12x_5 & 4-4x_4 & 4x_5-4x_3 \\ 0 & 4x_4-4x_3 & 0 & 4x_5-4x_3 & 1-x_2 \end{bmatrix} \\
B'_2 &= \begin{bmatrix} 1-x_5 & 2x_5-2x_4 & x_3-x_2 \\ 2x_5-2x_4 & 2-2x_2 & 2x_5-2x_4 \\ x_3-x_2 & 2x_5-2x_4 & 1-x_4 \end{bmatrix} \\
B'_3 &= \begin{bmatrix} x_2+1 \end{bmatrix}
\end{aligned} . \tag{5.3}$$

$$L_d = \left[\begin{array}{ccccccccccccc}
x_2 \\
& 1-x_2 \\
& & x_3 \\
& & & x_2-x_3 \\
& & & & 1+x_3-x_2-x_4 \\
& & & & & 1+x_3-x_2-x_5 \\
& & & & & & x_4-x_3 \\
& & & & & & & 1+x_3-x_4-x_5 \\
& & & & & & & & x_5-x_3 \\
& & & & & & & & & x_4 \\
& & & & & & & & & & 1-x_4 \\
& & & & & & & & & & & x_5 \\
& & & & & & & & & & & & 1-x_5
\end{array}\right] \tag{5.4}$$

Now it is fairly easy to split the semidefinite cone into summation of several matrices with the variables as the coefficients. By (3.13) and (3.14) we formulated the dual semidefinite programming formulation of this $\Omega(4)$ coloring problem, which is shown as the following MATLAB code:

```
cvx_begin sdp
  variable X(31,31) hermitian;
  minimize(trace(C'*X));
    dual variable Q;
    for i = 1:size(u,2)
      trace(A(:,:,i)'*X) <= -u(i);
    end
    X >= 0 : Q;
cvx_end
```

Here A is a 4-dimensional 31×31 matrix vector, and each of the elements is the coefficient of an x_i, where $i = \{2, 3, 4, 5\}$. By solving this dual problem of the $\Omega(4)$ coloring problem, we can get the dual optimal solution which is 3, since we hard coded the variable x_6 as $x_6 = 1$, this means we obtained the same optimal value as the primal SDP problem. By examining the dual variable Q in the above code, we can obtain the same solution as shown in the solution of the primal SDP problem.

5.2 Generate the dual SDP automatically

After examine the small instance of the graph coloring problem, we are very interested in some larger scale problems, such as $\Omega(8)$. But as we have seen, generating the dual SDP problem by hand is very difficult and it is very easy to make errors, so we wrote a program to parse the outputs of the primal SDP problem and generate the dual SDP problem with MATLAB script.
We wrote the parser in C++, which parses the text files representing the objective function, semidefinite cone and the linear constraints respectively. Our output files are written in the format defined by ourselves which could be easily recognized by our parser. During the process of building the primal SDP formulations we output the information needed and then the parser collect all the information needed for the dual SDP formulation and output it to files. Finally a simple MATLAB script parses the files outputed from the C++ parser and build the SDP problem. Please find Appendix C and D for the C++ parser and the dual SDP builder.

Chapter 6

Summary and future work

From our software suite, we can obtain the optimal solution for both the primal SDP and its dual problem for the graph coloring problem correctly. This results could be useful for future study on the difference of the optimal solution between the primal and dual problem. In future study, we will focus on finding the pattern of difference between the optimal solution of the primal SDP and its dual. Once we make an assumption based on our observation, we will try to prove it for infinite large n analytically. This could probably tighten the gap between the upper bound of and the lower bound of code size.

Bibliography

[1] Raj Chandra Bose and Dale M Mesner. On linear associative algebras corresponding to association schemes of partially balanced designs. *The Annals of Mathematical Statistics*, pages 21–38, 1959.

[2] Philippe Delsarte. *An algebraic approach to the association schemes of coding theory*. PhD thesis, Universite Catholique de Louvain., 1973.

[3] Michael Grant and Stephen Boyd. Graph implementations for nonsmooth convex programs. In V. Blondel, S. Boyd, and H. Kimura, editors, *Recent Advances in Learning and Control*, Lecture Notes in Control and Information Sciences, pages 95–110. Springer-Verlag Limited, 2008. `http://stanford.edu/~boyd/graph_dcp.html`.

[4] Michael Grant and Stephen Boyd. CVX: Matlab software for disciplined convex programming, version 2.1. `http://cvxr.com/cvx`, March 2014.

[5] E de Klerk and DV Pasechnik. A note on the stability number of an orthogonality graph. Technical report, 2005.

[6] Florence Jessie MacWilliams and Neil James Alexander Sloane. *The theory of error-correcting codes*, volume 16. Elsevier, 1977.

[7] Alex Samorodnitsky. On the optimum of delsarte's linear program. *Journal of Combinatorial Theory, Series A*, 96(2):261–287, 2001.

[8] Alexander Schrijver. New code upper bounds from the terwilliger algebra and semidefinite programming. *Information Theory, IEEE Transactions on*, 51(8):2859–2866, 2005.

Appendices

Appendix A

C++ source code for LP bound

```
// DesarteLPBound.h
#pragma once
#include <ilcplex/cplex.h>
#include <ilcplex/ilocplex.h>
#include <cmath>
#include <vector>
#include <iostream>
using namespace std;
class DelsarteLPBound
{
public:
    DelsarteLPBound(int nn, int d);
    ˜DelsarteLPBound(void);
    void Formulate();
    void GenerateEigenmatrix();
    void Solve();
    void Process();
private:
    int nchoosek(int n, int k);
    int KrawtchoukPolynomial(int i, int j);
    int nn;
    int dd;
    vector<vector<int> > Q;
    IloEnv env;
    IloModel model;
    IloNumVarArray var_x;
    IloObjective obj;
    IloCplex cplex;

};

// DelsarteLPBound.cpp
#include "DelsarteLPBound.h"


DelsarteLPBound::DelsarteLPBound(int n, int d):
    nn(n),
    dd(d),
    Q(nn+1)
{
    for (int i = 0; i < Q.size(); i++)
    {
        Q[i] = vector<int>(nn+1, 0);
    }
}



DelsarteLPBound::~DelsarteLPBound(void)
{
}

int DelsarteLPBound::nchoosek(int n, int k)
{
    if (n < 0)
    {
        return 1;
    }
    if (k < 0)
    {
        return 0;
    }
    if (k > n)
    {
        return 0;
    }
    vector<vector<int> > pt(n+1);
    for (int i = 0; i < pt.size(); i++)
    {
        pt[i] = vector<int>(i+1,1);
    }
    for (int i = 2; i < pt.size(); i++)
    {
        for (int j = 1; j < pt[i].size()-1; j++)
        {
            pt[i][j] = pt[i-1][j-1]+pt[i-1][j];
        }
    }
    return pt[n][k];
}

inline int DelsarteLPBound::KrawtchoukPolynomial(int i, int j)
{
    int result = 0;
    for (int k = 0; k <= j; k++)
    {
        result += pow(-1,k)*nchoosek(i,k)*nchoosek(nn-i,j-k);
    }
    return result;
}

void DelsarteLPBound::GenerateEigenmatrix()
{
    for (int i = 0; i < Q.size(); i++)
    {
        for (int j = 0; j < Q[i].size(); j++)
        {
            Q[i][j] = KrawtchoukPolynomial(i,j);
        }
    }
}

void DelsarteLPBound::Formulate()
{
    env = IloEnv();
    model = IloModel(env);
    var_x = IloNumVarArray(env);
    for (int i = 0; i <= nn; i++)
    {
        string name("a");
        name.append(std::to_string(i));
        IloNumVar r(env, 0, IloInfinity, IloNumVar::Float, name.c_str());
        var_x.add(r);
    }
    obj = IloMaximize(env, IloSum(var_x));
    model.add(obj);
    GenerateEigenmatrix();
    for (int j = 1; j <= nn; j++)
    {
        IloExpr expr(env);
        for (int i = 0; i <= nn; i++)
        {
            expr += var_x[i]*Q[i][j];
        }
        model.add(expr >= 0);
    }
    model.add(var_x[0] == 1);
    for (int i = 1; i < dd; i++)
    {
        model.add(var_x[i] == 0);
    }
}

void DelsarteLPBound::Solve()
{
    cplex = IloCplex(env);
    cplex.extract(model);
    //cplex.exportModel("model.lp");
    cplex.setOut(env.getNullStream());
    if (cplex.solve())
    {
        int optval = cplex.getObjValue();
        printf("Optimal␣value␣for␣n=%d,␣d=%d␣is␣%d.\n", nn, dd, optval);
        printf("Variables␣of␣optimal␣solutions␣are␣\n");
        for (int i = 0; i <= nn; i++)
        {
            float a_i = cplex.getValue(var_x[i]);
            printf("a[%d]␣=␣%e\t", i, a_i);
        }
        cout << endl;
    }
}

void DelsarteLPBound::Process()
{
    Formulate();
    Solve();
}

int binomial(int n, int k)
{
    if (n < 0 || k < 0)
    {
        return 0;
    }
    if (k > n)
    {
        return 0;
    }
    vector<vector<int> > pt(n+1);
    for (int i = 0; i < pt.size(); i++)
    {
        pt[i] = vector<int>(i+1,1);
    }
    for (int i = 2; i < pt.size(); i++)
    {
        for (int j = 1; j < pt[i].size()-1; j++)
        {
            pt[i][j] = pt[i-1][j-1]+pt[i-1][j];
        }
    }
    return pt[n][k];
}
int main(int argc, char* argv[])
{
    if (argc != 2)
    {
        printf("Usage:\n");
        printf("lpbound.exe␣n\n");
        printf("where␣n␣is␣an␣integer.");
        return -1;
    }
    int MAX_N = atoi(argv[1]);
    for (int n = 4; n <= MAX_N; n++)
    {
        for (int d = 2; d <= n; d++)
        {
            DelsarteLPBound del(n,d);
            del.Process();
        }
    }
    return 0;
}
```

Appendix B

MATLAB source code for SDP bound

```
function b=active(alpha,d,n)
% active: Answer "1" (yes) if this is a shape for length n which is allowed to
% be non-zero.  Otherwise answer "0" (no).
    if ( (alpha(1)+alpha(2)+alpha(3)+alpha(4) < n) ) ||...
     ( (alpha(1)+alpha(2)+alpha(3)+alpha(4) > n) ) ||...
     ( (0 < alpha(2)+alpha(3)) && (alpha(2)+alpha(3) < d) ) ||...
     ( (0 < alpha(2)+alpha(4)) && (alpha(2)+alpha(4) < d) ) ||...
     ( (0 < alpha(3)+alpha(4)) && (alpha(3)+alpha(4) < d) )
        b = 0;
    else
        b = 1;
    end
end

function b = beta(alpha, omega)
  n = sum(alpha);
    b = 0;
    r = alpha(3) + alpha(4);
    s = alpha(2) + alpha(4);
    for u = 0 : n
        if (u >= alpha(4)) && ...
           (r-u >= 0)      && ...
           (u-omega >= 0) && ...
           (s-u >= 0)
            b = b + (-1)^(u-alpha(4))*nchoosek(u, (alpha(4)))*...
            nchoosek((n-2*omega), (u-omega))*nchoosek((n-...
            omega-u), (r-u))*nchoosek((n-omega-u), (s-u));
        end
    end
end

function beta=checkalpha(alpha)
% Matlab function for Schrijver SDP converts alpha to (alpha0+alpha3, 0 ,
% alpha1+alpha2 , 0) as in (19) and in Schrijver
  beta=[alpha(1)+alpha(3) 0 alpha(2)+alpha(4) 0];
end

function beta=primealpha(alpha)
% Matlab function for Schrijver SDP converts alpha to (alpha0+alpha3, 0 ,
% alpha1+alpha2 , 0) as in (19) and in Schrijver
  beta=[alpha(1)+alpha(2) 0 alpha(3)+alpha(4) 0];
end

function sh=repshape(alpha)
% map alpha to equivalent alpha' with i>=j>=k
    if  (alpha(2) <= alpha(3)) && (alpha(3) <= alpha(4))
        sh = [alpha(1) alpha(2) alpha(3) alpha(4)];
    end
    if  (alpha(2) <= alpha(4)) && (alpha(4) <= alpha(3))
        sh = [alpha(1) alpha(2) alpha(4) alpha(3)];
    end
    if  (alpha(3) <= alpha(2)) && (alpha(2) <= alpha(4))
        sh = [alpha(1) alpha(3) alpha(2) alpha(4)];
    end
    if  (alpha(3) <= alpha(4)) && (alpha(4) <= alpha(2))
        sh = [alpha(1) alpha(3) alpha(4) alpha(2)];
    end
    if  (alpha(4) <= alpha(2)) && (alpha(2) <= alpha(3))
        sh = [alpha(1) alpha(4) alpha(2) alpha(3)];
    end
    if  (alpha(4) <= alpha(3)) && (alpha(3) <= alpha(2))
        sh = [alpha(1) alpha(4) alpha(3) alpha(2)];
    end
end

function beta=zalpha(alpha)
% Matlab function for Schrijver SDP converts alpha to (alpha0+alpha3, 0 ,
% alpha1+alpha2 , 0) as in (19) and in Schrijver
    beta=[alpha(1)+alpha(4) 0 alpha(2)+alpha(3) 0];
end

  %% initial arguments
  n = 4;          % length of code
  d = 3;         % minimum Hamming distance
  m = floor(n/2);    % total number of eigenspace

  %% Generate all the alphas:
  allalpha = {};
  for alpha0 = 0 : n
    for alpha1 = 0 : n-alpha0
      for alpha2 = 0 : n-alpha0-alpha1
        new_row = [alpha0  alpha1  alpha2  n-alpha0-alpha1-alpha2];
        allalpha = [allalpha ; new_row];
      end
    end
  end
  %for i = 1 : length(allalpha)
  % disp(allalpha{i});
  %end

  N = length(allalpha);
  % uint8 only allows 255 variables, which may be small.
  indx = uint16(ones(n+1,n+1,n+1,n+1));

  numvars = 1;   % Need x(1)=0 for all inactive shapes.
  for shape = 1 : N  % Include trivial shape. But this variable is set to 1.
    alpha = allalpha{shape};
    if (active(alpha,d,n)) &&...
      % Only consider if a1 <= a2 <= a3, i.e. i >= j >= k
      isequal(alpha,repshape(alpha))
      numvars = numvars+1;  % One variable for each unordered triple (i,j,k)
      indx(1+alpha(1), 1+alpha(2), 1+alpha(3),...
      1+alpha(4))=numvars; %This will be the corr. variable name.
      disp(sprintf('x(%d)=[%d␣%d␣%d␣%d]', numvars, alpha(1), alpha(2),...
      alpha(3), alpha(4)));
    end
  end

  for shape = 1 : N  % Now go through and index each alpha to its variable.
    alpha = allalpha{shape};
    gamma = repshape(alpha);    % Cf. (20)(iii)
    indx(1+alpha(1),1+alpha(2),1+alpha(3),1+alpha(4))...
    = indx(1+gamma(1),1+gamma(2),1+gamma(3),1+gamma(4));
  end
```

```
%% start modeling
cvx_begin sdp

variables x(numvars);
Bls = {};
disp('Begin␣to␣generate␣B␣and␣B"...');
for omega = 0 : m
  expression Bl(n-omega+1,n-omega+1);   % Matrix defined in (19) from R
  expression Blp(n-omega+1,n-omega+1);  % Matrix defined in (19) from R'
  for i = omega : (n-omega)
    for j = omega : (n-omega)
      for t = 0 : min(i,j)
        if (n+t-i-j >= 0)
          gamma = [n+t-i-j,j-t,i-t,t]; % Here is the shape determined by i,j,t
          delta = zalpha(gamma);
          index = double(indx(1+gamma(1),1+gamma(2),1+gamma(3),1+gamma(4)));
          b = double(beta(gamma, omega));
          Bl(i+1,j+1) = Bl(i+1,j+1)+b*x(index);
          %disp(sprintf('Bl[%d,%d]+=%d*x(%d)', i, j, b, index));
          Blp(i+1,j+1) = Blp(i+1,j+1)+...
            double(beta(gamma, omega))*...
            (x(double(indx(1+delta(1),1+delta(2),1+delta(3),1+delta(4))))-...
            x(double(indx(1+gamma(1),1+gamma(2),1+gamma(3),1+gamma(4)))));
          %disp(sprintf('Blp[%d,%d]+=%d*(x(%d)-x(%d))', i, j, b, ...
          %indx(1+delta(1),1+delta(2),1+delta(3),1+delta(4)),...
          %indx(1+gamma(1),1+gamma(2),1+gamma(3),1+gamma(4))));
        end
      end
    end
  end
  Bl = Bl((omega+1):end,(omega+1):end);
  Blp = Blp((omega+1):end,(omega+1):end);
  Bls(end+1) = {Bl};
  Bls(end+1) = {Blp};
  if omega == 0
    Bl(1,1) = 1;
  end
end

expression B;
B = blkdiag(Bls{1:length(Bls)});
disp('B␣and␣B"␣generating␣finished.');
matrix_dim = size(B,1);

expression ObjFunc;
for i = 0 : n
  alpha = [n-i 0 i 0];
  ObjFunc=ObjFunc+double(mychoose(n,i))*...
    x(indx(alpha(1)+1,alpha(2)+1,alpha(3)+1,alpha(4)+1));
  %disp(sprintf('ObjFun+=%d*x[%d]',double(mychoose(n,i)), ...
  %indx(alpha(1)+1,alpha(2)+1,alpha(3)+1,alpha(4)+1)));
end

dual variables y{3*(N-1)};
constraints = {};
disp('Begin␣to␣add␣consstraints...');
maximize(ObjFunc);
subject to
x(1) == 0;    % This kills off all inactive shapes alpha
%x(numvars) == 1;        % (20)(i)
nineqconstr = 0;
```

```
for h = 1 : N-1           % Ignore trivial partition
  alpha = allalpha{h};
  if(active(alpha, d, n))
    % We only created variables when they can be nonzero
    l = indx(alpha(1)+1,alpha(2)+1,alpha(3)+1,alpha(4)+1);
    % Variable subscript for alpha=(i,j,t)
    zal = primealpha(alpha);
    li = indx(zal(1)+1,zal(2)+1,zal(3)+1,zal(4)+1);
    % Variable subscript for (i,0,0) Cf. (20)(ii)
    zal = checkalpha(alpha);
    lj = indx(zal(1)+1,zal(2)+1,zal(3)+1,zal(4)+1);
    % same for (j,0,0) Cf. (20)(ii)
    x(l) >= 0 : y{(h-1)*3+1};        % (20)(ii), ineq. 1
    constraints{(h-1)*3+1} = sprintf('x(%d)␣>=␣0', l);
    nineqconstr = nineqconstr + 1;
    x(l) - x( li ) <= 0 : y{(h-1)*3+2};    % (20)(ii), ineq. 2
    constraints{(h-1)*3+2} = sprintf('x(%d)␣-␣x(%d)␣<=␣0', l, li);
    nineqconstr = nineqconstr + 1;
    %disp(sprintf('x(%d)<=x(%d)', l, li));
    x(li)+x(lj)-1-x(l) <= 0 : y{(h-1)*3+3}; % (20)(ii), ineq. 3
    constraints{(h-1)*3+3} = sprintf('x(%d)+x(%d)-1-x(%d)␣<=␣0', li, lj, l);
    nineqconstr = nineqconstr + 1;
    %disp(sprintf('x(%d)+x(%d)<=1+x(%d)',li,lj,l));
  end
end
disp(sprintf('Totally␣%d␣inequality␣constraints␣added...',nineqconstr));
B == semidefinite(matrix_dim);
B == transpose(B);
disp('All␣constraints␣has␣been␣added␣in␣to␣the␣model');
cvx_end
```

```
  yy = {};
  cc = {};
  %% parsing dual solutions
  for i = 1:size(y,1)
    if ~isempty(y{i})
      if abs(y{i}(1)) > 0.0001
        yy(end+1) = {y{i}};
        cc(end+1) = {constraints{i}};
      end
    end
  end
  disp('The␣primal␣variables␣are:');
  disp(x);
  disp('The␣dual␣variables␣are:');
  disp(yy);
  yy = yy';
  cc = cc';
  out = cell(length(yy),2);
  for i = 1:length(yy)
    out{i,1} =  yy{i};
    out{i,2} =  cc{i};
  end
% Ty = cell2table(yy, 'VariableNames', {'y'});
% Tc = cell2table(cc, 'VariableNames', {'c'});
  Tout = cell2table(out, 'VariableNames', {'y' 'c'});
% writetable(Ty, ['y',num2str(n),',',num2str(d)]);
% writetable(Tc, ['c',num2str(n),',',num2str(d)]);
  filename = ['out',num2str(n),',',num2str(d),'.csv'];
  writetable(Tout, filename, 'Delimiter', ',');
```

Appendix C

C++ code for parsing primal SDP outputs

```
#pragma once
#include <string>
#include <iostream>
#include <cmath>
#include <algorithm>
#include <vector>
#include <fstream>
using namespace std;
typedef vector<vector<int> > MatInt;
typedef vector<vector<string> > MatStr;
class FileParser
{
public:
    FileParser(int n, int nv);
    ˜FileParser();
    void ReadFromFile();
    void ParseInputs();
    void Process();
    void PrintInputs();
    void InitializeMatrices();
    void PrintOutputs();
    void OutputMatrices();
private:
    int N;
    int m;
    int nVar;
    int mdim;
    vector<MatStr> inputs;
    vector<MatInt> As;
    MatInt C;
    vector<int> offsets;
    vector<int> coefs;
    inline int get_x_index(string& x);
    inline int get_coef(string& c);
    inline bool isPlusorMinus(string& s);
    void Tokenize(const string& str,
        vector<string>& tokens,
        const string& delimiters);
};

#include "FileParser.h"


void FileParser::Tokenize(const string& str,
            vector<string>& tokens,
            const string& delimiters = "␣")
{
  // Skip delimiters at beginning.
  string::size_type lastPos = str.find_first_not_of(delimiters, 0);
  // Find first "non-delimiter".
  string::size_type pos  = str.find_first_of(delimiters, lastPos);

  while (string::npos != pos || string::npos != lastPos)
  {
    // Found a token, add it to the vector.
    tokens.push_back(str.substr(lastPos, pos - lastPos));
    // Skip delimiters.  Note the "not_of"
    lastPos = str.find_first_not_of(delimiters, pos);
    // Find next "non-delimiter"
    pos = str.find_first_of(delimiters, lastPos);
  }
}

FileParser::FileParser(int n, int nv):
N(n),
nVar(nv),
m(n/2),
As(nv),
inputs(n+3)
{
}


FileParser::~FileParser()
{
}


void FileParser::ReadFromFile()
{
  char buffer[500];
```

```
vector<string> files;
for (auto i = 0; i <= m; i++)
{
  sprintf_s(buffer,
    "D:/Chao/Dropbox/Workspace/Matlab/deklerk_sdp_code/Bls%d.txt",
    i + 1);
  string filename(buffer);
  files.push_back(filename);
  sprintf_s(buffer,
    "D:/Chao/Dropbox/Workspace/Matlab/deklerk_sdp_code/Blps%d.txt",
    i + 1);
  filename = string(buffer);
  files.push_back(filename);
}
files.push_back(string(
  "D:/Chao/Dropbox/Workspace/Matlab/deklerk_sdp_code/linear.txt"
  ));
for (auto i = 0; i < files.size(); i++)
{
  ifstream fin(files[i]);
  string line;
  vector<string> dirs;
  Tokenize(files[i], dirs, "/");
  string filename = dirs[dirs.size() - 1];
  getline(fin, line);
  printf("%s\n", filename.c_str());
  while (getline(fin, line))
  {
    vector<string> entries;
    Tokenize(line, entries, ",␣");
    inputs[i].push_back(entries);
```

```
      }
    }
    offsets.push_back(0);
    for (auto i = 0; i < inputs.size(); i++)
    {
      offsets.push_back(inputs[i].size()+offsets[i]);
    }
    PrintInputs();
    ifstream fin("D:/Chao/Dropbox/Workspace/Matlab/deklerk_sdp_code/objcoef.txt");
    string line;
    getline(fin, line);
    getline(fin, line);
    vector<string> entries;
    Tokenize(line, entries, ",␣");
    for (size_t i = 0; i < entries.size(); i++)
    {
      coefs.push_back(stoi(entries[i]));
    }
  }

  bool FileParser::isPlusorMinus(string& s)
  {
    return s == "+" || s == "-" ? true : false;
  }

  int FileParser::get_x_index(string& x)
  {
    return stoi(x.substr(1, x.size() - 1));
  }

  int FileParser::get_coef(string& c)
```

```
{
  return stoi(c);
}

void FileParser::ParseInputs()
{
  for (auto i = 0; i < inputs.size()-1; i++)
  {
    int offset = offsets[i];
    for (auto j = 0; j < inputs[i].size(); j++)
    {
      for (auto k = 0; k < inputs[i][j].size(); k++)
      {
        string tmp = inputs[i][j][k];
        // Skip delimiters at beginning.
        string::size_type lastPos = tmp.find_first_not_of("*", 0);
        // Find first "non-delimiter".
        string::size_type pos  = tmp.find_first_of("*", lastPos);

        while (string::npos != pos)
        {
          string::size_type tmpPos = pos + 1;
          if (tmp.substr(tmpPos, 1) == "x")
          {
            int xpos = tmpPos;
            while (!isPlusorMinus(tmp.substr(tmpPos, 1))
              && tmpPos < tmp.size())
            {
              tmpPos++;
            }
            string x_ind(tmp.substr(xpos, tmpPos - xpos));
            int ind = get_x_index(x_ind)-2;
            tmpPos = xpos - 2;
            xpos--;
            while (!isPlusorMinus(tmp.substr(tmpPos, 1))
              && tmpPos >= 0)
            {
              tmpPos--;
            }
            int num = get_coef(tmp.substr(tmpPos, xpos-tmpPos));
            As[ind][offset + j][offset + k] += num;
          }
          else if (tmp.substr(tmpPos, 1) != "0")
          {
            int cpos = tmpPos;
            while (!isPlusorMinus(tmp.substr(tmpPos, 1))
              && tmpPos < tmp.size())
            {
              tmpPos++;
            }
            int num1 = get_coef(tmp.substr(cpos, tmpPos - cpos));
            tmpPos = cpos - 2;
            cpos--;
            while (!isPlusorMinus(tmp.substr(tmpPos, 1))
              && tmpPos >= 0)
            {
              tmpPos--;
            }
            int num = get_coef(tmp.substr(tmpPos, cpos - tmpPos));
            C[offset + j][offset + k] += num*num1;
          }
          // Skip delimiters.  Note the "not_of"
```

```
        lastPos = tmp.find_first_not_of("*", pos);
        // Find next "non-delimiter"
        pos = tmp.find_first_of("*", lastPos);
      }
    }
  }
  printf("Finished␣%d␣files!\n", i + 1);
}
auto i = inputs.size() - 1;
auto j = inputs[i].size()-1;
int offset = offsets[offsets.size() - 2];
for (auto k = 0; k < inputs[i][j].size(); k++)
{
  string tmp = inputs[i][j][k];
  // Skip delimiters at beginning.
  string::size_type lastPos = tmp.find_first_not_of("/", 0);
  // Find first "non-delimiter".
  string::size_type pos = tmp.find_first_of("/", lastPos);

  while (string::npos != pos)
  {
    int tmpPos = pos - 1;
    if (tmpPos == 0 ||
      (stoi(tmp.substr(tmpPos, 1)) == 1)
      && isPlusorMinus(tmp.substr(tmpPos-1, 1))
      )
    {
      int num = stoi(tmp.substr(lastPos, pos - lastPos));
      C[offset + k][offset + k] += num;
    }
    else
    {
      while (tmp.substr(tmpPos, 1) != "x")
      {
        tmpPos--;
      }
      int xpos = tmpPos;
      while (!isPlusorMinus(tmp.substr(tmpPos, 1))
        && tmpPos < tmp.size())
      {
        tmpPos++;
      }
      string x_ind(tmp.substr(xpos, tmpPos - xpos));
      int ind = get_x_index(x_ind)-2;
      do
      {
        tmpPos--;
      } while (!isPlusorMinus(tmp.substr(tmpPos, 1)));
      if (tmp.substr(tmpPos, 1) == "+")
      {
        As[ind][offset + k][offset + k] += 1;
      }
      else
      {
        As[ind][offset + k][offset + k] += -1;
      }
    }
    // Skip delimiters.  Note the "not_of"
    lastPos = tmp.find_first_not_of("/", pos);
    // Find next "non-delimiter"
    pos = tmp.find_first_of("/", lastPos);
  }
```

```
    printf("linear␣constraint␣%d␣finished.\n", k);
  }
  cout << "Finished␣linear␣constraints!" << endl;
}



void FileParser::InitializeMatrices()
{
  mdim = 0;
  for (int i = 0; i < inputs.size() - 1; i++)
  {
    mdim += inputs[i].size();
  }
  mdim += inputs[inputs.size() - 1][0].size();
  printf("Matrices␣are␣initialized␣as␣%dx%d␣dimensions.\n", mdim, mdim);
  C = MatInt(mdim);
  for (auto i = 0; i < C.size(); i++)
  {
    C[i] = vector<int>(mdim, 0);
  }
  for (auto i = 0; i < As.size(); i++)
  {
    As[i] = MatInt(mdim);
    for (auto j = 0; j < As[i].size(); j++)
    {
      As[i][j] = vector<int>(mdim, 0);
    }
  }
}

void FileParser::Process()
```

```
{
  ReadFromFile();
  InitializeMatrices();
  ParseInputs();
  //PrintOutputs();
  OutputMatrices();
  printf("Matrices'␣dimension␣is␣%dx%d", mdim, mdim);
}

void FileParser::OutputMatrices()
{
  for (size_t i = 0; i < As.size(); i++)
  {
    char buffer[500];
    sprintf_s(buffer, "A%d", i);
    string file(buffer);
    ofstream fout(file, std::ofstream::out);
    for (size_t j = 0; j < As[i].size(); j++)
    {
      for (size_t k = 0; k < As[i][j].size(); k++)
      {
        fout << As[i][j][k] << "\t";
      }
      fout << endl;
    }
    fout.close();
  }
  char buffer[500];
  sprintf_s(buffer, "Co");
  string file(buffer);
  ofstream fout(file, std::ofstream::out);
  for (size_t i = 0; i < C.size(); i++)
  {
    for (size_t j = 0; j < C[i].size(); j++)
    {
      fout << C[i][j] << "\t";
    }
    fout << endl;
  }
  fout.close();
  file = string("Obj");
  fout.open(file, std::ofstream::out);
  for (size_t i = 0; i < coefs.size(); i++)
  {
    fout << coefs[i] << "\t";
  }
  fout.close();
}

void FileParser::PrintOutputs()
{
  for (auto i = 0; i < As.size(); i++)
  {
    printf("Coefficient␣matrix␣%d.\n", i + 1);
    for (auto j = 0; j < As[i].size(); j++)
    {
      for (auto k = 0; k < As[i][j].size(); k++)
      {
        cout << As[i][j][k] << "\t";
      }
      cout << endl;
    }
    cout << endl << endl;
  }
  printf("Constant␣matrix.\n");
  for (auto i = 0; i < C.size(); i++)
  {
    for (auto j = 0; j < C.size(); j++)
    {
      cout << C[i][j] << "\t";
    }
    cout << endl;
  }
}


void FileParser::PrintInputs()
{
  for (auto i = 0; i < inputs.size(); i++)
  {
    for (auto j = 0; j < inputs[i].size(); j++)
    {
      for (auto k = 0; k < inputs[i][j].size(); k++)
      {
        cout << inputs[i][j][k] << '\t';
      }
      cout << endl;
    }
    cout << endl << endl;
  }
}

int main(void)
```

```
{
  //FileParser fp(8, 26);
  FileParser fp(4, 4);
  fp.Process();
  return 0;
}
```

Appendix D

MATLAB source code for generate the dual SDP problem

```
%n = 26; dim = 230;
n = 4; dim=43;
A = zeros(dim, dim, n);
for i = 1:n
  file = sprintf('A%d', i-1);
  A(:, :, i) = importdata(file, '\t', 0);
end
file = sprintf('Co');
C = importdata(file, '\t', 0);
file = sprintf('Obj');
u = importdata(file, '\t', 0);
cvx_begin sdp
    variable X(dim,dim) hermitian;
    minimize(trace(C'*X));
    dual variable Q;
    for i = 2:size(u,2)-1
      trace(A(:,:,i-1)'*X) <= -u(i);
    end
    X >= 0 : Q;
cvx_end
OutQ = array2table(full(Q));
writetable(OutQ, 'OutQ');
```