Application of Association Schemes in Calculating and Implementing Linear Programming Bounds for Codes in SageMath

Charalampos Kokkalis

Jesus College

University of Oxford

A thesis submitted for the degree of

BA Computer Science

Trinity Term 2021

Abstract

Coding theory is an area of mathematics that examines the reliable and efficient transmission of information over noisy channels. To achieve this, mathematical constructions named codes are used. An important goal of coding theory is to study the relationship between the length and the specification of a code, and its ability to transmit information accurately and efficiently. This project aims to investigate how the theory of association schemes can be applied in order to formulate bounds on the size of codes and how these bounds can then be implemented for an open-source library, SageMath, so that they can be used for further research in a great variety of fields. The starting point for these is the derivation of the Linear Programming bound, as it was put forth first by Delsarte [8]. This bound is then enhanced with further constraints, obtaining 40 improved bounds on the size of binary constant weight codes. Finally, we generalize the bound to other schemes, demonstrating its application on the association scheme of Hermitian dual polar graphs.

Acknowledgements

I would like to deeply thank my supervisor Dr. Pasechnik for guiding me through this process, facilitating my introduction to a variety of interesting fields of mathematics, helping me become familiar with SageMath, and kindly offering previous unpublished work of his, in the form of both code and papers, to assist me in completing the project.

I would also like to express my gratitude towards my college tutor Professor Živný for his continuous support throughout the years of my undergraduate degree, going above and beyond to ensure that all of us had an all-around smooth learning experience.

Last but not least, I want to thank my family and friends for their unfaltering support and encouragement, always being there for me in difficult moments.

Contents

1 Introduction

1.1 Motivation

In the age of the internet, huge amounts of information are constantly being transmitted around us, encoded using some sort of a coding system. The distance and the mediums that information has to go through causes loss of information to be a common issue. Using more technical terminology, information channels are usually noisy. Coding theory is an area of mathematics that deals with the reliable and efficient transmission of information over noisy channels. The constructions we use to achieve this are called codes. An obvious criterion to judge the practicality of a code is its size, meaning the number of distinct codewords it contains, as this depicts the amount of information we can encode using it. Therefore, it is important to be able to reason about bounds on the sizes of codes. This constitutes the first aim of this project, while the second goal is to implement the bounds discussed above in SageMath. SageMath is a free, open-source mathematics software system, which covers a great variety of fields. Its mission is to "create a viable free open source alternative to Magma, Maple, Mathematica and Matlab."[21]. It is built using Python, and it allows coding in Python as well as in a language (sage) which is an extension of Python.

1.2 Problem Statement and Related Work

The main theme of the project, which is then improved and extended, is Delsarte's Linear Programming Bound. First derived in P.Delsarte's 1973 paper "An algebraic approach to the association schemes of coding theory"[8], the connection between association schemes and certain codes has already been used in a variety of papers, along with numerous improvements, aiming to provide tighter bounds to the size of codes. Two classes of such codes are the Hamming Codes and the Binary Constant Weight Codes, with the corresponding association schemes being the Hamming scheme and the Johnson scheme. SageMath already contained work for the former, therefore the first problem solved by this project is the derivation and implementation of the Linear Programming bound for Binary Constant Weight Codes. The first question that immediately arises is where else this bound could be applied. For this reason, we provide a more generic formulation of the Linear Programming bound, which can be used to calculate bounds on Q-polynomial association schemes. We proceed by employing this generic function to the Q-polynomial association scheme of Hermitian dual polar graphs.

1.3 Project Structure

We begin by providing some theoretical background on coding theory, which is essential in order to introduce the different types of codes and, subsequently, their properties. Next, we introduce the theory of association schemes, and we use their properties to derive the Linear Programming bound.

The first application of the bound is on Binary Constant Weight Codes, which correspond to the Johnson association scheme. We provide an enhanced bound by considering an Integer Linear Program, since we can reason that the domain of the variables of our objective function consists only of integers. Having implemented the established bounds, we can then test our results compared to the best known bounds in the relevant bibliography. This yields an improved bound in 40 cases, as demonstrated in section 4.7.

In section 5 we generalize this result further, providing an implementation that calculates similar bounds given the parameters of any Q-polynomial association scheme, and in section 6 we demonstrate its application to the association scheme of Hermitian dual polar graphs.

2 Coding Theory Foundations

2.1 Key Definitions

Before presenting the main results of this project, it is important to provide some basic definitions. A *field* is a set on which addition and multiplication behave similarly to how they behave for rational and real numbers. Denote $\mathbb{F}_q$ to be a field, the elements of which create an alphabet of size $q = p^k$, for p a prime number. We then define a *word* to be an element $v \in \mathbb{F}_q^n$, where n is a fixed natural number. A *code* C is a subset of $\mathbb{F}_q^n$. Every word $v \in C$ (for come code $C \subseteq \mathbb{F}_q^n$) is called a *codeword* of that code. The *weight* of a codeword v, denoted as $w(v)$, is the number of nonzero entries of v. It is useful to define a *distance function* between any two codewords. One of the most widely used distance functions is the *Hamming distance* ($d_H(u, v)$ for some codewords u, v), which measures the number of positions in which two words differ: $d_H(u, v) = |\{i \mid u_i \neq v_i\}|$. [22]

2.2 Further Properties

One important property of codes is their minimum distance. We define the *minimum distance* $d_{\min}(C)$ of a code C to be the minimum distance between any pair of distinct codewords in C. To appreciate why this property is useful, we can consider the transmission of a codeword v that belongs in some code C. As discussed in the introduction, it is important for codes to be reliable, therefore any errors in their transmission should be recognizable and we should be able to recover the original codeword, assuming a reasonable number of transmission errors. In the case where v is transmitted but $v' \neq v$ is received, we should have a method to try and reconstruct the codeword that was intended to be sent.

A typical approach for achieving that is by using the method of Nearest Neighbour decoding. In our example, the method would calculate the distances between v' and all $c \in C$, and return the codeword c' that is closest to v'. We can easily see that the number of errors that are detectable and correctable are somehow related to the minimum distance of the code [17]. As mentioned earlier, our goal is to establish upper bounds on the size of codes.

We define $A(n,d)$ to be the maximum size of a code (which is a subset of some $\mathbb{F}_q^n$) of minimum distance at least d, and $A(n,d,w)$ to be the maximum size of such a code with the additional property of each codeword having weight w.

Before proceeding to specific types of codes and their bounds, it is also important to introduce the theory of association schemes and their connection with some codes. This is what we are going to use later to derive the linear programs, which will give us the bounds on the size of the codes.

3 Association Schemes

3.1 Definition

An *association scheme* with d classes consists of a finite set $\mathcal{X}$ together with a set $\mathcal{R}$ of $d+1$ relations $R_0, R_1, ..., R_d$ on $\mathcal{X}$ with the following properties:

(i) $\mathcal{R}$ partitions $\mathcal{X}^2$

(ii) R_0 is defined as $R_0 = \{(x,x)|x \in \mathcal{X}\}$

(iii) if $(x,y) \in R_i$, then also $(y,x) \in R_i$ for all $x, y \in \mathcal{X}$ and $i \in \{0, ..., d\}$, which is equivalent to saying that for any $R_i \in \mathcal{R}$ the inverse relation R_i^{-1} also belongs to $\mathcal{R}$

(iv) For any $i, j, k \in \{0, 1, ..., d\}$, there exists a number $p^k_{i,j} = p^k_{j,i}$ such that for all $(x,y) \in R_k$:

$$|\{z \in \mathcal{X}|(x,z) \in R_i, (z,y) \in R_j\}| = p^k_{i,j}$$

The numbers $p^k_{i,j}$ are called the intersection numbers of the association scheme. It is then convenient to define the intersection matrices $L_0, ...L_d$ which satisfy $(L_i)_{kj} = p^k_{ij}$. [5]

3.2 The Bose-Mesner Algebra

It is convenient to describe the members of $\mathcal{R}$ by their adjacency matrices:

$$(A_i)_{xy} = \begin{cases} 1 & \text{if } (x,y) \in R_i \\ 0 & \text{otherwise} \end{cases}$$

We can now rewrite the axioms of association schemes in the following way:

(i) $\sum_{i=0}^{d} A_i = J$, where J is the matrix of all 1s

(ii) $A_0 = I$, where I is the identity matrix

(iii) $A_i = A_i^T$, for all $i \in \{0, 1, ..., d\}$

(iv) $A_i A_j = \sum_k p_{i,j}^k A_k$, for all $i, j, k \in \{0, ..., d\}$

We can see that the matrices A_i are linearly independent from axiom (i). To proceed, it is useful to define the concept of an algebra over a field. An *algebra* is a vector space together with a bilinear product operation (which combines elements of two vector spaces to return an element of a third vector space, being linear in all its arguments). Furthermore, we see that the above axioms (ii)-(iv) can generate a commutative (since axiom (iv) and the fact that $p_{i,j}^k = p_{j,i}^k$ yield that $A_i A_j = A_j A_i$) $(d+1)$-dimensional algebra $\mathcal{A}$ of symmetric matrices (from axiom (iii)) with constant diagonal, since A_0's diagonal consists only of 1s, and all other A_is' diagonals consist only of 0s. This particular algebra was first studied by Bose and Mesner and it is called the *Bose-Mesner algebra.* [5]

Since the A_i matrices commute, they can be diagonalized simultaneously [9], that is, there exists a matrix S such that for each $A_i \in \mathcal{A}$, $S^{-1} A_i S$ is a diagonal matrix. By the spectral theorem of linear algebra[15], $\mathcal{A}$ has a unique basis of minimal idempotents $E_0, ..., E_d$, which satisfy:

$$E_i E_j = \delta_{ij} E_i, \ \sum_{i=0}^{d} E_i = I$$

3.3 The matrices P and Q

Let P and $\frac{1}{n}Q$ be the matrices which relate the two bases for $\mathcal{A}$ as follows:

$$A_j = \sum_{i=0}^{d} P_{ij} E_i, \ E_j = \frac{1}{n} \sum_{i=0}^{d} Q_{ij} A_i$$

or equivalently in matrix form:

$$A = EP, \ E = \frac{1}{n} AQ$$

Then:

$$PQ = (E^{-1}A)(nA^{-1}E) = nI$$

and similarly:

$$QP = nI$$

Furthermore, we note that

$$A_jE_i = (\sum_{i=0}^{d} P_{ij}E_i)E_i = \sum_{k=0}^{d} P_{kj}E_kE_i = \sum_{k=0}^{d} P_{kj}\delta_{ki}E_i = P_{ij}E_i$$

which shows that P_{ij} are the eigenvalues of A_j and that the columns of E_i are the corresponding eigenvectors. We can compute the matrices P and Q from the intersection numbers of the scheme. To achieve this, we first show another property of the above-defined P matrix as follows:
We know that $A_jA_k = \sum_{i=0}^{d} p_{i,j}^{k} A_k$. Using the definition of P_{ij} we then have:

$$A_jA_k = \sum_{l=0}^{d} p_{j,k}^{l} A_l \Leftrightarrow A_jA_kE_i = (\sum_{l=0}^{d} p_{j,k}^{l} A_l)E_i \Leftrightarrow A_jP_{ik}E_i = \sum_{l=0}^{d} p_{j,k}^{l} A_lE_i$$

$$\Leftrightarrow P_{ik}A_jE_i = \sum_{l=0}^{d} p_{j,k}^{l} P_{il}E_i \Leftrightarrow P_{ij}P_{ik} = \sum_{l=0}^{d} p_{j,k}^{l} P_{il}$$

We can now show that the intersection matrix L_j has eigenvalues P_{ij}:

$$(PL_jP^{-1})_{im} = \sum_{k=0}^{d}\sum_{l=0}^{d} P_{il}(L_j)_{lk}(P^{-1})_{km} = \sum_{k=0}^{d}\sum_{l=0}^{d} P_{il}p_{j,k}^{l}(P^{-1})_{km} =$$

$$= \sum_{k=0}^{d}(\sum_{l=0}^{d} P_{il}p_{j,k}^{l})(P^{-1})_{km} = \sum_{k=0}^{d} P_{ij}P_{ik}(P^-1)_{km} = P_{ij}\sum_{k=0}^{d} P_{ik}(P^{-1})_{km} = \delta_{im}P_{ij}$$

This means that $PL_jP^{-1} = \text{diag}(P_{0j}, ..., P_{dj})$, so the P_{ij} are the eigenvalues of L_j. Hence, we can easily now compute P and $Q = \frac{1}{n}P^{-1}$. We also notice that the rows of P are left eigenvectors and the columns of Q are right eigenvectors.

3.4 The Linear Programming Bound

We have introduced association schemes as a resource to be used to derive upper bounds for the size of substructures. What we describe next is the derivation of Delsarte's Linear Programming Bound.[8]

First, define Y to be some nonempty subset of X, and its inner distribution to be the vector a given by $a_i = \frac{|(Y \times Y) \cap R_i|}{|Y|}$. This can be thought of as the average number of elements of Y that are related in R_i to some other (given) element in Y. Let y be the characteristic vector of Y. Then, we can rewrite $a_i = \frac{1}{|Y|} y^T A_i y$. Delsarte's theorem states that $aQ \geq 0$. To prove this, consider the quantity $|Y|(aQ)_j$. This is nonnegative if and only if $aQ \geq 0$, since $|Y|$ is just a positive constant. Then, we have:

$$|Y|(aQ)_j = |Y| \sum_{j=0}^{d} a_i Q_{ij} = y^T \left(\sum_{j=0}^{d} Q_{ij} A_i \right) y = n y^T E_j y \geq 0$$

since E_j is by definition positive semidefinite. Therefore, we have proved that $aQ \geq 0$. To relate this to codes, we can think of the relations of the association scheme to be $R_i = \{(x, y) | x, y \in C \text{ and } d(x, y) = i\}$, where d is some distance metric and C is the code we are considering. Let a be the distribution vector of the code. Then, if the code has minimum distance r, it should satisfy $|C| \leq \max(\sum_{i=0}^{d} a_i)$, where the maximum is taken over all possible $\{a_0, ..., a_d\}$ that satisfy:

- $a_0 = 1$ (every word has distance 0 only from itself)
- $a_i = 0$ for $1 \leq i \leq r$ (since r is defined to be the minimum distance)
- $a_i \geq 0$ for all i
- $aQ \geq 0$ (by Delsarte's theorem)

We will now proceed to apply this to a specific class of codes, that of Binary Constant Weight Codes.

4 Bound for Binary Constant Weight Codes

4.1 Definition of Binary Constant Weight Codes

An (n, d) *binary code* is a subset of $\mathcal{F}_2^n$, where $\mathcal{F} = \{0, 1\}$, whose codewords have minimum distance d. An (n, d, w) *constant weight binary code* is a code satisfying the above, with the additional property that all its codewords have the same weight (w), which in our case means that each codeword has w ones and $n - w$ zeros. We are looking to calculate bounds on the maximum size of the code, which we denote as $A(n, d, w)$.

4.2 The Johnson Scheme

Earlier, we defined the Hamming distance to be the number of entries in which two codewords differ. In the case of binary constant weight codes, we can realize that this distance will always be even. We define the *Johnson distance* (named after S.M.Johnson, who first considered these codes) to be $d_J(x, y) = \frac{1}{2} d_H(x, y)$ for all $x, y \in \mathcal{X}$.[12] Now we can define the relations for the association scheme as $R = \{R_0, R_1, ..., R_n\}$ where:

$$R_i = \{(x, y) \in \mathcal{X}^2 \mid d_J(x, y) = i\}$$

which, as we can easily notice, form a partition of $\mathcal{X}^2$. Therefore, this yields an association scheme with n classes, which we will use to reason about the bounds of binary constant weight codes. It is also useful to notice the valence of each of the R_i, which is equal to the following:

$$v_i = \binom{w}{i} \binom{n - w}{i}$$

4.3 Properties of Binary Constant Weight Codes

Before proceeding to build the LP, it is convenient to introduce some useful properties of Binary Constant Weight Codes [16], which we can then turn into constraints for our LP, to achieve a tighter bound:

(i) $A(n, d, w) = A(n, d + 1, w)$, if d is odd

(ii) $A(n, d, w) = A(n, d, n - w)$

(iii) $A(n, 2, w) = \binom{n}{w}$

(iv) $A(n, 2w, w) = \lfloor \frac{n}{w} \rfloor$

(v) $A(n, d, w) = 1$ if $2w < d$

We can then always assume (using (i) and (iii)) that d is even and $d \geq 4$. Furthermore, using (ii),(iv),(v), we can assume that $d < 2w \leq n$. [13]

4.4 Deriving the Linear Program

To apply the theory discussed in section 4, we have to find formulas for the eigenmatrices P and Q. These can be computed using a result in combinatorics that produces the Eberlein polynomial (or dual Hahn polynomial), which is defined in the following way:

$$E_k(x) = \sum_{j=0}^{k} (-1)^j \binom{x}{j} \binom{w-x}{k-j} \binom{n-w-x}{k-j}$$

Then, as proved by Delsarte in [8]:

$$P_k(i) = E_k(i), \quad Q_i(k) = \mu_i v_k^{-1} E_k(i)$$

where

$$\mu_i = \binom{n}{i} - \binom{n}{i-1} = \frac{n-2i+1}{n-i+1} \binom{n}{i}$$

We can now construct the Linear Program as described in the last section. This will be similar to what was derived in section 4, with the additional constraints of section 5.3. The result is the following LP:

$$\begin{array}{lll}
\text{maximize} & \displaystyle\sum_{i=0}^{n} A_i & \\
\text{subject to} & A_0 = 1 & \\
 & A_j = 0 & \text{for } j \text{ s.t. } 0 < j < d \\
 & A_j = 0 & \text{for } j \text{ s.t. } 2w < j \\
 & A_j = 0 & \text{for odd } j \\
 & \sum_{j=0}^{n} Q_k(j) A_j \geq 0 & \text{for } k = 1, ..., w \\
 & A_j \geq 0 & \text{for all } j \geq d/2
\end{array}$$

In order to make implementation easier and more efficient, it is important to simplify the LP by reducing the number of variables and constraints. We can eliminate the first five constraints by simply rewriting the objective function, yielding the following LP:

$$\begin{array}{lll}
\text{maximize} & \displaystyle\left(\sum_{i=d/2}^{w} A_{2i}\right) + 1 & \\
\text{subject to} & \sum_{j=d/2}^{w} Q_k(j) A_{2j} \geq 0 & \text{for } k = 1, ..., w \\
 & A_{2j} \geq 0 & \text{for } d/2 \leq j \leq w
\end{array}$$

which now has $w - \frac{d}{2} + 1$ variables and $(w - \frac{d}{2} + 1)(w + 1)$ constraints. This is significantly more efficient than the original version which had $n + 1$ variables and $d + (n - 2w) + \frac{n}{2} + w(n + 1) + n - \frac{d}{2} + 1$ constraints.

We can now proceed to implement this for SageMath.

4.5 Implementation in SageMath

SageMath already contained code for calculating Delsarte bounds for Hamming codes [20], so it was clear where in the codebase the new additions belonged, as well as what the coding conventions and paradigms followed by SageMath were.

The first function to implement was the one defining the Eberlein polynomials:

```
def eberlein(n,k,l,x):
  from sage.arith.all import binomial
  from sage.arith.srange import srange
  return sum([(-1)**j*binomial(x,j)*binomial(n-x,k-j)
    *binomial(l-n-x,k-j) for j in srange(0,k+1)])
```

Of course, the above is a simplified version of the actual code, which includes sanity checks on the values of the arguments as well as many lines of documentation, which contain examples, tests and references justifying the correctness of what is being implemented. Notice that we are using other tools from SageMath at the same time, such as *binomial* and *srange*, in order to achieve consistency and potentially more efficient code.

We are now ready to build the linear program, before we use a prebuilt solver to get a solution. The version of the LP that we implement is based on Delsarte's theory as we discussed, and a simplification of the formulation to make it more suitable for coding by Kang [13]. We build the LP using the following helper function:

```
def _delsarte_cwc_LP_building(n,d,w,solver,isinteger):
  from sage.numerical.mip import MixedIntegerLinearProgram
  from sage.arith.all import binomial
  p = MixedIntegerLinearProgram(maximization=True, solver=solver)
  A = p.new_variable(integer=isinteger, nonnegative=True)
  p.set_objective(sum([A[2*r] for r in range(d//2,w+1)])+1)
  def _q(k,i):
    v_i = binomial(w,i) * binomial(n-w,i)
    return eberlein(w,i,n,k)/v_i
  for k in range(1,w+1):
    p.add_constraint(sum([A[2*i]*_q(k,i) for i in range(d//2,w+1)]),
      min=-1)
  return A, p
```

In the above code, we take advantage of SageMath's *MixedIntegerLinearProgram* class to build the LP. Following the documentation, one can see that the *solver* and *isinteger* arguments that are passed to the function are used to specify whether the variables of the LP can be restricted to integers (in which case we have an Integer Linear Program, which is an NP-Hard problem) or not, and what solver we would like to use. Finally, we create the main function that gets exported, namely *delsarte_ bound_ constant_ weight_ code*. This simply calls the above helper function with some default values for *solver* and *isinteger*, and makes sure the correct exceptions are thrown when they should, for example in case of invalid input. The details of all three functions can be found in Appendix I.

4.6 Results and Comparison to Prior Art

Having implemented the code to build the Linear Program, we make use of SageMath's sophisticated LP solvers to calculate the bounds. To evaluate the outcome of our approach, we compare our results to the best previously known upper bounds. To this end, the tables by A.E.Brouwer [3] proved very helpful. In order to scrape and clean the data from the tables (as they contained both lower and upper bounds as well some references to the papers establishing them), a Python script was written, which can be found in Appendix II. Having extracted the data, we can create a script that automatically tests our approach, taking advantage of the fact that it is possible to import our local version of SageMath (including the above changes) in any Python file by running a sage shell built using the local version of Sage. The testing module iterates over possible values for n and w and in the cases where the bounds are defined, it checks how the new derived bound compares to the previously known one, keeping track of the number of better, equal, and worse bounds achieved, as well as of the cases where the new bound is an improvement. Running this showed that we match the best known bound in 40 cases, and we get an overestimation in the remaining 32 cases.

4.7 Improving the Bound

It is important to realize that the variables of the optimization problem should be integers. That is because what these variables represent is the number of words at a specific distance from a fixed word. Therefore, instead of using an LP solver for the optimization problem, we could use an Integer Linear Programming solver. However, as ILP is an NP-Hard problem [6], this is going to be computationally challenging. Testing using the ILP on a personal computer produced a solution for certain n, d, w. More specifically, in the cases where $d = \{4, 6, 8\}$, the processes had to be manually interrupted after some point before continuing with the next value for d. This means that the results of solving the ILP remain unknown for some of these cases. The interesting outcome is that in 40 out of the cases in which the algorithm terminated, it produced an improved bound (see table of improvements on the next page).

n	**d**	**w**	**previous**	**improved**
17	4	3	44	43
23	4	3	83	81
20	6	9	1363	1335
20	6	10	1420	1413
21	6	10	2685	2604
23	10	10	116	115
23	10	11	135	134
24	10	10	170	168
24	10	11	222	217
25	10	9	157	154
25	10	10	262	261
25	10	11	379	377
26	10	9	213	208
28	10	12	1977	1967
28	10	13	2438	2425
28	10	14	2628	2626
29	10	12	3091	3981
27	12	12	139	138
27	12	13	155	154
28	12	11	147	146
28	12	12	198	196
29	12	11	197	193
29	12	12	298	296
29	12	14	492	486
30	12	10	159	157
30	12	12	492	486
31	12	10	229	194
31	12	11	415	375
31	12	12	679	678
32	12	10	300	243
32	12	11	573	568
32	12	14	2140	2133
32	12	15	2641	2496
32	12	16	2870	2642
29	14	12	47	46
31	14	14	167	165
31	14	15	183	182
32	14	13	183	181
32	14	14	233	232
32	14	15	277	274

Table 1: List of Improved Bounds for Binary Constant Weight Codes

5 P- and Q- Polynomial Association Schemes

Next, we look to generalize these bounds to other types of codes. To do this, we will consider a specific class of association schemes, namely the $P-$ and $Q-$ polynomial association schemes.

5.1 The Krein parameters

The Bose-Mesner algebra $\mathcal{A}$ is also closed under componentwise (Hadamard) matrix multiplication (denoted as $\circ$). We can then define

$$E_i \circ E_j = \frac{1}{n} \sum_{k=0}^{d} q_{ij}^k E_k$$

where the numbers q_{ij}^k are called the *Krein parameters*. An important property of the Krein parameters is that for all $i, j, k \in \{0, ..., d\}$ they satisfy $q_{ij}^k \geq 0$. This shows a dual behaviour between on the one side ordinary multiplication, the intersection numbers p_{ij}^k and the matrices A_i and P and on the other side Hadamard multiplication, the Krein parameters q_{ij}^k and the matrices E_i and Q. [5]

We are now able to define $P-$ and $Q-$ polynomial association schemes as we discussed earlier.

5.2 Definition

Association schemes in which the relations R_i represent two elements having distance i are called *metric*. Metric schemes are characterized by the fact that p_{ij}^k is zero whenever one of i, j, k is larger than the sum of the other two (we can think of this as a form of a triangle inequality), and nonzero in the case where $i = j + k$. As we would expect, a cometric scheme is defined dually, by $q_{ij}^k = 0$ when $i > j + k$ and $q_{jk}^i > 0$ when $i = j + k$.

An association scheme is called *$P-$polynomial* if there exist polynomials f_k of degree k with real coefficients, and real numbers z_i such that $P_{ik} = f_k(z_i)$. Dually, an association scheme is called Q*−polynomial* when the above holds for Q instead of P.

There is an interesting relation between metric association schemes and a notion that we will introduce in section 6.2, that of *distance-regular* graphs. Namely, both represent the same object. An important theorem which can be found in [5] (Theorem 11.6.1) states that an association scheme is metric (resp. cometric), and thus can be thought of as a distance-regular graph, if and only if it is P-polynomial (resp. Q-polynomial).

5.3 Generalization of LP Bound to Q-polynomial schemes

In section 3.4, we showed how to derive a Linear Programming Bound given an association scheme. The resulting LP depends only in the Q-matrix of the scheme as well as its minimum distance. Therefore, it makes sense to implement a more general function for SageMath, which given the Q-matrix and the minimum distance d returns the LP bound on the corresponding association scheme. We can then test this function by producing the Q-matrix, using the Q-polynomial of the Q-polynomial scheme. The resulting code to build the LP is similar to that for Binary Constant Weight codes:

```
def _delsarte_Q_LP_building(q,d,solver,isinteger):
  from sage.numerical.mip import MixedIntegerLinearProgram
  n, _ = q.dimensions()
  p = MixedIntegerLinearProgram(maximization=True,solver=solver)
  A = p.new_variable(integer=isinteger,nonnegative=True)
  p.set_objective(sum([A[i] for i in range(n)]))
  p.add_constraint(A[0]==1)
  try:
    for i in range(1,d):
      p.add_constraint(A[i]==0)
```

```
except:
  for i in d:
    p.add_constraint(A[i]==0)


for k in range(n):
  p.add_constraint(sum([q[k][i]*A[i] for i in range(n)]),min=0)
return A, p
```

An additional feature of the above code is that it allows for the option of having a list of indices such that $A[i]$s will be set to 0 in the LP, which proves to be useful for some association schemes. We use the try-except block to achieve this form of polymorphism (d can be either an integer - the minimum distance - or a list of integers), which is consistent with Python's conventions.

Indeed, we verify that this procedure works by testing it against the ones already implemented in SageMath, the one for the Hamming scheme [20] and the one we just derived for the Johnson scheme. The code producing the Q matrices for both schemes can be found in Appendix IV.

In the next chapter, we are going to show how this function can be applied to other problems that have the structure of an association scheme, such as the Hermitian Dual Polar graphs.

6 Application to Hermitian Dual Polar Graphs

We begin by defining some basic terms that are essential to establishing what Hermitian Dual Polar Graphs are.

6.1 Definitions

Let $V(n,q)$ denote the vector space of dimension n over the field $\mathcal{F}_q$. Consider also a *Hermitian form*, which we can think of as an inner product, defined as a function $h : V \times V \to \mathbb{C}$ such that $h(w,z) = \overline{h(z,w)}$. A subspace U of $V(n,q)$ is called *totally isotropic* if the restriction to U of the Hermitian form above is identically zero, meaning that $h(z,w) = 0$ for all $z, w \in U$. Then, a *Hermitian polar space* $\mathcal{P}$ is the collection of all totally isotropic subspaces of $V(n,q)$. Define the *Witt index* d of $V(n,q)$ to be the dimension of the largest totally isotropic subspace of $V(n,q)$, which we often call the *rank* of $\mathcal{P}$. All maximal isotropic subspaces are called *generators* of the polar space. We can now define the *Hermitian Dual Polar Graph* on $\mathcal{P}$ to be the graph whose vertices are the generators of $\mathcal{P}$ and where two generators are adjacent if and only if their intersection has dimension $d-1$. [1]

6.2 Deriving the matrix Q

We are looking for bounds on the constant distance codes of generators on Hermitian polar spaces of type $H(2d-1,q^2)$. This means that the vector space in the definition of the specific Hermitian polar space has dimension $2d-1$ and the size of the field is q^2, as it also includes complex numbers. For generators of Hermitian polar spaces, constant distance codes are sets of subspaces which pairwise intersect in codimension i (so in dimension $d-i$ where d is the Witt index of the polar space). We define *partial spreads* as constant distance codes with $i = d$. [11] Let X be the set of all the above Hermitian forms and define R_i relations as follows:

$$(x,y) \in R_i \Leftrightarrow (x,y) \text{ intersect in codimension } i$$

Then, $Y = (X, \{R_i\}_{0 \leq i \leq d})$ is a (P and Q)-polynomial association scheme.[2] The following are the intersection numbers of the distance-regular [1] graph $\Gamma = (X, R_1)$:

$$b_i = b_i(d,q) = \frac{q^{2d} - q^{2i}}{q+1}$$

$$c_i = c_i(d,q) = \frac{q^{i-1}(q^i - (-1)^i)}{q+1}$$

$$a_i = a_i(d,q) = \frac{q^{2i} - q^{i-1}(q^i - (-1)^i) - 1}{q+1}$$

It is important to consider the meaning of these intersection numbers. Let the partition $V_0, V_1, ..., V_d$ of the set of vertices X (of graph Γ) with respect to the distances from some $v \in X$. Define $k_i = |V_i|$. Now, the b_j is the number of edges going from $v_j \in V_j$ to V_{j+1} and c_j is the number of edges going from $v_j \in V_j$ to V_{j-1}. Due to distance-regularity, the subgraph G_j of edges between V_j and V_{j+1} is biregular (V_j, V_{j+1} partition the vertices in two sets the vertices of which are not connected with each other, making G_j a bipartite subgraph), for any vertex of G_j in V_j the number of edges on it is b_j and for any vertex of G_j in V_{j+1} this number is c_{j+1}. Therefore, counting the number of edges of G_j in two different ways we get:

$$b_j|V_j| = c_{j+1}|V_{j+1}| \Rightarrow b_j k_j = c_{j+1} k_{j+1} \Rightarrow k_{j+1} = \frac{b_j}{c_{j+1}} k_j$$

Note that $k_0 = 1$, since only one vertex has distance 0 from v (itself), thus we can compute k_{j+1} recursively, as we already know the b_j and c_{j+1}. This is implemented in the function *Hdpgk* (returning the value of k for Hermitian dual polar graph) below:

```
def Hdpgk(p,f,d,j):
    q = p^f
```

[1]A connected graph Γ is called *distance-regular* if it is regular of valency (degree of vertices) k and for any two points $\gamma, \delta \in \Gamma$ at distance $i = d(\gamma, \delta)$, there are precisely c_i neighbours of δ in $\Gamma_{i-1}(\gamma)$ and b_i neighbours of δ in $\Gamma_{i+1}(\gamma)$. The array of integers ($\{b_0, b_1, ..., b_{d-1}; c_1, c_2, ..., c_d\}$, where d is the diameter of Γ) characterizing a distance-regular graph is known as its intersection array.[4]

```
    def b(i):
        return q^(2*i)*(q^(2*(d-i))-1)/(q+1)
    def c(i):
        return q^(i-1)*(q^i-(-1)^i)/(q+1)
    if j==0: return 1
    return b(j-1)*Hdpgk(p,f,d,j-1)/c(j)
```

There was preexisting work from my supervisor in implementing the Q-matrix for Hermitian dual polar graphs, which with the addition of the above calculation of the values of k gives a complete implementation of theorem 9.4.3 in [4]. The resulting sage code producing the Q-matrix which we are going to use as input to the procedure defined in section 5.3 can be found in Appendix V.

Having completed this, it remains to test it for different values of q and d and compare it to the best known bounds so far.

6.3 Results and Comparison to Prior Art

In F.Ihringer's paper "A new upper bound for constant distance codes of generators on Hermitian polar spaces of type $H(2d-1,q^2)$"[10], a table of bounds on the maximum size of partial spreads in $H(2d-1,q^2)$ is given, depending on the values of d (where d is even) and q. As discussed in the paper, it is not known whether these bounds are sharp or not, so it would be worth testing our approach in deriving them. We can then test the generalized procedure that was defined in section 5.3 for all Q-polynomial schemes, using as input the matrix Q derived in the previous section. The code for testing this and comparing with the best known bounds for each case can be found in Appendix VI.

All the bounds that were produced by the code were less tight than the best known ones discussed in [10]. However, this still verifies that our construction is correct and indeed provides an upper bound in all cases. This could be revisited when further improvements to the bound on association schemes are implemented, as we will discuss later in the future work section of the conclusion.

Similarly to the case of binary constant weight codes, in which we only got improvements in a limited number of cases, here also we see that there are other approaches that outperform our method. This is logical, because most of these approaches further exploit the structure of the specific problem, whereas ours applies a generic method using only the Q-polynomiality of the schemes. However, the ability of our method to generalize proves to be very useful and worth exploring, since a potential breakthrough in bounds for Q-polynomial association schemes could instantly lead to improvements on a variety of problems.

7 Conclusion

7.1 Evaluation

This project served two purposes; firstly, providing rigorous mathematical proofs to derive the bounds, and, secondly, discussing their implementation for SageMath and the results of their testing. To this end, we formulated the Linear Programming Bound, using properties of the structure of Association Schemes, around which the core of the project revolved.

This bound was first implemented specifically for the case of Binary Constant Weight Codes, for which the Integer Linear Program produced improvements to the previously best known bounds in 40 cases, using the fact that the variables of the objective function all represent integers. Consequently, we generalized the function to be applicable to all types of Q-polynomial association schemes, which, after being deployed in SageMath - an open-source framework-, will facilitate research in a variety of topics. Even in the specific project, other than the field of coding theory, there were applications to graph theory, information theory, geometry, and combinatorics. There is even an interesting instance of a quantum information theory problem to which similar bounds can be applied, discussed in [14].

Finally, we demonstrated how this generalized code could be applied to a different setting, by considering the association scheme corresponding to Hermitian Dual Polar Graphs. After deriving the Q-matrix for these, we compared the upper bounds of our implementation to the best known ones, which unfortunately in this case did not show any improvements, but still verified the fact that the upper bounds are correct (they do indeed provide a close overestimation).

7.2 Future Work

An obvious way to apply the results of this project would be to use the generic function on other Q-polynomial association schemes in the literature, similarly to what we did for Hermitian dual polar graphs. An example of that could be partial ovoids [7], such as the Ree-Tits octagon $O(2^t)$ [11]. The book "Algebraic Combinatorics I"[2] provides an extensive list of other examples of P- and Q- polynomial association schemes in chapter 3.6.

Another opportunity for extension of this project would be in deriving even tighter bounds. There has already been work on that, with an example of such an improvement being the use of Semidefinite Programming (cf. Schrijver's paper "New Code Upper Bounds From the Terwillger Algebra and Semidefinite Programming"[19]). Clearly, an improvement on that could yield tighter bounds in all of the above-discussed association schemes. Furthermore, an improvement to the more specific bound for binary constant weight codes is discussed in the paper "Delsarte's Linear Programming Bound for Constant-Weight Codes"[13] in section III.B, and another one was given by Polak in "Semidefinite programming bounds for constant weight codes" [18], based on the work of Schrijver.

References

[1] R. F. Bailey and P. Spiga. Metric dimension of dual polar graphs, 2017.

[2] E. Bannai and I. Tatsuro. *Algebraic combinatorics.* Benjamin/Cummings, 1984.

[3] A. E. Brouwer. Bounds for binary constant weight codes, Mar 2021.

[4] A. E. Brouwer, A. M. Cohen, and A. Neumaier. *Distance-regular graphs.* Springer, 1989.

[5] A. E. Brouwer and W. H. Haemers. *Spectra of Graphs.* Springer, Amsterdam, The Netherlands, 2011.

[6] S. Dasgupta, C. H. Papadimitriou, and U. V. Vazirani. *Algorithms.* McGraw-Hill Education (India), 2016.

[7] J. De Beule, A. Klein, and K. Metsch. *Substructures of finite classical polar spaces*, pages 35–61. 01 2012.

[8] P. Delsarte. An algebraic approach to the association schemes of coding theory. *Philips Research Reports Supplements, No.10*, 1973.

[9] R. A. Horn and C. R. Johnson. *Matrix Analysis.* Cambridge University Press, 2017.

[10] F. Ihringer. A new upper bound for constant distance codes of generators on Hermitian polar spaces of type $H(2d-1, q^2)$. *Journal of Geometry*, 105(3):457–464, 2014.

[11] F. Ihringer, P. Sin, and Q. Xiang. New bounds for partial spreads of $H(2d-1, q^2)$ and partial ovoids of the ree–tits octagon. *Journal of Combinatorial Theory, Series A*, 153:46–53, 2018.

[12] S. Johnson. Upper bounds for constant weight error correcting codes. *Discrete Mathematics*, 3(1):109–124, 1972.

[13] B. G. Kang, H. K. Kim, and P. T. Toan. Delsarte's linear programming bound for constant-weight codes. *IEEE Transactions on Information Theory*, 58(9):5956–5962, 2012.

[14] E. D. Klerk and D. Pasechnik. A note on the stability number of an orthogonality graph. *European Journal of Combinatorics*, 28(7):1971–1979, 2007.

[15] S. Lang. *Linear algebra.* Springer, 2011.

[16] F. J. MacWilliams and N. J. A. Sloane. *The Theory of Error-Correcting Codes.* North Holland, Amsterdam, The Netherlands, 1976.

[17] S. McKinley. The hamming codes and delsarte's linear programming bound. Master's thesis, Portland State University, May 2003.

[18] S. Polak. Semidefinite Programming Bounds for Constant Weight Codes. *IEEE Transactions on Information Theory*, 65(1):28–38, 2019.

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

[20] The Sage Development Team. Delsarte (or linear programming) bounds. `https://doc.sagemath.org/html/en/reference/coding/sage/coding/delsarte_bounds.html`, 2015.

[21] The Sage Development Team. Sagemath homepage. `https://www.sagemath.org/`, 2021.

[22] D. J. A. Welsh. *Codes and Cryptography.* Oxford University Press, Oxford, UK, 1988.

8 Appendices

8.1 Appendix I: Code for Binary Constant Weight Codes

```
def eberlein(n, w, k, u, check=True):
    r"""
    Compute ``E^{n,l}_k(x)``, the Eberlein polynomial.
    Equal to
    .. MATH::
        E^{w,n}_k(u)=\sum_{j=0}^k (-1)^j \binom{u}{j} \binom{w-u}{k-j}
        \binom{n-w-u}{k-j},
    INPUT:
    - ``w, k, x`` -- arbitrary numbers
    - ``n`` -- a nonnegative integer
    - ``check`` -- check the input for correctness. ``True`` by
      default. Otherwise, pass it as it is. Use ``check=False`` at
      your own risk.
    EXAMPLES:
        sage: codes.bounds.eberlein(24,10,2,6)
        -9
    """
    from sage.arith.all import binomial
    from sage.arith.srange import srange
    if check:
        from sage.rings.integer_ring import ZZ
        n0 = ZZ(n)
        if n0 != n or n0 < 0:
            raise ValueError('l must be a nonnegative integer')
        n = n0
    if 2*w>n:
        return eberlein(n,n-w,k,u)
    return sum([(-1)**j*binomial(u,j)*binomial(w-u,k-j)
        *binomial(n-w-u,k-j) for j in srange(0,k+1)])
def _delsarte_cwc_LP_building(n, d, w, solver, isinteger):
    """
    LP builder for Delsarte's LP for constant weight codes, used in
    delsarte_bound_constant_weight_code; not exported.
    INPUT:
    - ``n`` -- the code length
    - ``w`` -- the weight of the code
    - ``d`` -- the (lower bound on) minimal distance of the code
    - ``solver`` -- the LP/ILP solver to be used. Defaults to
      ``PPL``. It is arbitrary precision, thus there will be no
      rounding errors. With other solvers (see
      :class:`MixedIntegerLinearProgram` for the list), you are on
      your own!
    - ``isinteger`` -- if ``True``, uses an integer programming solver
      (ILP), rather that an LP solver. Can be very slow if set to
      ``True``.
    """
    from sage.numerical.mip import MixedIntegerLinearProgram
    from sage.arith.all import binomial
    p = MixedIntegerLinearProgram(maximization=True, solver=solver)
    A = p.new_variable(integer=isinteger, nonnegative=True)
    p.set_objective(sum([A[2*r] for r in range(d//2,w+1)])+1)
    def _q(k,i):
        v_i = binomial(w,i)*binomial(n-w,i)
        return eberlein(n,w,i,k)/v_i

    for k in range(1,w+1):

        p.add_constraint(sum([A[2*i]*_q(k,i) for i in range(d//2,w+1)]),min=-1)

    return A, p

def delsarte_bound_constant_weight_code(n, d, w, return_data=False,

    solver="PPL", isinteger=False):

    """

    Find the Delsarte bound on a constant weight code of weight ``w``,

    length ``n``, lower bound on minimal distance ``d``

    INPUT:

    - ``n`` -- the code length

    - ``d`` -- the (lower bound on) minimal distance of the code

    - ``w`` -- the weight of the code

    - ``return_data`` -- if ``True``, return a triple

      ``(W,LP,bound)``, where ``W`` is a weights vector, and ``LP``

      the Delsarte upper bound LP; both of them are Sage LP data.

      ``W`` need not be a weight distribution of a code.

    - ``solver`` -- the LP/ILP solver to be used. Defaults to

      ``PPL``. It is arbitrary precision, thus there will be no

      rounding errors. With other solvers (see

      :class:`MixedIntegerLinearProgram` for the list), you are on

      your own!

    - ``isinteger`` -- if ``True``, uses an integer programming solver

      (ILP), rather that an LP solver. Can be very slow if set to

      ``True``.

    """

    from sage.numerical.mip import MIPSolverException

    if d<4:
        raise ValueError("Violated constraint d>=4 for
          Binary Constant Weight Codes")
    if d>=2*w or 2*w>n:
        raise ValueError("Violated constraint d<2w<=n for
          Binary Constant Weight Codes")
    # minimum distance is even => if there is an odd lower bound on d we can
    # increase it by 1
    if d%2: d+=1
    A, p = _delsarte_cwc_LP_building(n, d, w, solver, isinteger)
    try:
        bd = p.solve()
    except MIPSolverException as exc:
        print("Solver exception: {}".format(exc))
        if return_data:
            return A,p,False
        return False
    if return_data:
        return A,p,bd
    else:
        return int(bd)
```

8.2 Appendix II: Testing for Binary Constant Weight Codes

Code for scraping Brouwer's page to get the bounds into csv files in order to automate testing:

```
import pandas as pd
import numpy as np
import json
tables = pd.read_html('https://www.win.tue.nl/~aeb/codes/Andw.html#d4')
```

```python
d_tables = {
        4:tables[1], 6:tables[4], 8:tables[8], 10:tables[15],
            12:tables[19], 14:tables[23], 16:tables[26],
            18:tables[28]
        }
d_vals = [4,6,8,10,12,14,16,18]
def format_data(x):
    # check for nan
    if pd.isna(x):
        return None
    # get upper bounds only (eliminate -)
    if '-' in str(x):
        x = x[(x.index('-')+1):]
    # get rid of references etc
    if not str(x).isdigit():
        i = 0
        while i<len(str(x)):
            if not str(x)[i].isdigit():
                x = str(x)[:i]
                break
            i += 1
    # cases where only lower bound given
    if x == '': return None
    return int(x)
# make first col index, drop last row, apply format_data, create csv
for key in d_vals:
    d_tables[key] = d_tables[key].set_index('n\w')[:-1].applymap(format_data)
    d_tables[key].to_csv(f'd{key}.csv')
```

Code for testing (both LP and ILP):

```
from sage.all import *

def cwc_tests(ilp=False):

    print('Testing')

    import pandas as pd

    d_vals = [4,6,8,10,12,14,16,18]

    # uncomment below accordingly to get computationally feasible subproblems

    # if ilp:d_vals=[16,18]

    d_tables = {i:pd.read_csv(f'd{i}.csv',index_col=0) for i in d_vals}

    for d in d_vals:

        print(20*'*')

        print(f'd={d}')

        print(20*'*')

        curr = d_tables[d]

        leq = eq = geq = 0

        improved = []

        for n in curr.index:

            for w in list(curr):

                b = curr.at[n,w]

                if not pd.isna(b):

                    print(f'n={n}, d={d}, w={w}, exp={curr.at[n,w]}')

                    if not ilp:

                        exp = codes.bounds.delsarte_bound_constant_weight_code(

                          int(n),int(d),int(w))

                    else:

                        exp = codes.bounds.delsarte_bound_constant_weight_code(

                          int(n),int(d),int(w),isinteger=True)

                    print(f'Got:{exp}')
                    if exp<b:
                        print('IMP')
                        leq+=1
                        improved.append(f'A({n},{d},{w})={exp} --- was {b}')
                    elif exp>b:geq+=1
                    else:eq+=1
        print('RESULTS')
        print(f'leq={leq},eq={eq},geq={geq}')
        print('******IMPROVED*****')
        for imp in improved:print(imp)
if __name__=="__main__":
    cwc_tests()
```

8.3 Appendix III: Code for Q-matrix LP Building

```
def _delsarte_Q_LP_building(q,d,solver,isinteger):
    from sage.numerical.mip import MixedIntegerLinearProgram
    n, _ = q.dimensions() # Q is a square matrix
    p = MixedIntegerLinearProgram(maximization=True, solver=solver)
    A = p.new_variable(integer=isinteger, nonnegative=True)
    p.set_objective(sum([A[i] for i in range(n)]))
    p.add_constraint(A[0]==1)
    try:
        for i in range(1,d):
            p.add_constraint(A[i]==0)
    except:
        for i in d:
            p.add_constraint(A[i]==0)
    for k in range(1,n):
        p.add_constraint(sum([q[k][i]*A[i] for i in range(n)]),min=0)
    return A, p


def delsarte_bound_Q_matrix(q,d,return_data=False, solver="PPL",
  isinteger=False):
    """
    Find the Delsarte bound on a code with Q matrix ``q`` and lower
    bound on minimal distance ``d``.
    INPUT:
    - ``q`` -- the Q matrix
    - ``d`` -- the (lower bound on) minimal distance of the code
    - ``return_data`` -- if ``True``, return a triple
      ``(W,LP,bound)``, where ``W`` is a weights vector, and ``LP``
      the Delsarte upper bound LP; both of them are Sage LP data.
      ``W`` need not be a weight distribution of a code.
    - ``solver`` -- the LP/ILP solver to be used. Defaults to
      ``PPL``. It is arbitrary precision, thus there will be no
      rounding errors. With other solvers (see
      :class:`MixedIntegerLinearProgram` for the list), you are on
      your own!
    - ``isinteger`` -- if ``True``, uses an integer programming solver
      (ILP), rather that an LP solver. Can be very slow if set to
      ``True``.
    """
    from sage.structure.element import is_Matrix
    from sage.numerical.mip import MIPSolverException
    if not is_Matrix(q):
        raise ValueError("Input to delsarte_bound_Q_matrix should be
        a sage Matrix()")
    A, p = _delsarte_Q_LP_building(q, d, solver, isinteger)
    try:
        bd=p.solve()
    except MIPSolverException as exc:
        print("Solver exception: {}".format(exc))
        if return_data:
            return A,p,False
        return False
    if return_data:
        return A,p,bd
    else:
        return bd
```

8.4 Appendix IV: Testing Code for Q-matrix LP Building

```
from sage.all import *
def hamming_scheme_q_matrix(n,q):
    """
    n length, q alphabet size, returns sage Matrix() representing the Q matrix
    """
    rows = []
    for i in range(n+1):
        row = []
        for j in range(n+1):
            row.append(codes.bounds.krawtchouk(n,q,i,j))
        rows.append(row)
    q_matrix = Matrix(rows)
    return q_matrix
```

```
def johnson_scheme_q_matrix(n,w):
    rows = []
    for i in range(w+1):
        row = []
        for j in range(w+1):
            row.append(codes.bounds.eberlein(n,w,i,j,inef=True))
        rows.append(row)
    p_matrix = Matrix(rows)
    q_matrix = binomial(n,w)*p_matrix.inverse()
    return q_matrix
```

8.5 Appendix V: Hermitian dual polar graphs Q-matrix generation

```
# (a_1,a_2,...,a_k;q)_n:=prod_{j=1}^k(a_j;q)_n
def brlist_n(q,n,*a):
    def br(x): return prod([1-x*q**k for k in range(n)])
    return prod([br(aj) for aj in a])
def q_hyper(q,z,a,b,ulim=oo):
    s = len(a)
    if s-1 == len(b):
        if ulim==oo: # do a symbolic summation
            var('kkk')
            return sum(z**kkk*brlist_n(q,kkk,*a)/
              brlist_n(q,kkk,*(b+[q])), kkk, 0, ulim)
        else:
            return reduce(lambda x,y: x+y, [z**k*brlist_n(q,k,*a)/
              brlist_n(q,k,*(b+[q])) for k in range(ulim+1)])
# P and Q for the Hermitian dual polar graph of dimension d
# over the field GF(p^f)
# multiplicites - j's multiplicity, starting from 0, e=1/2
# implement [BCN] Thm 9.4.3
def Hdpgm(p,f,d,j):
    q=p^(2*f)
    return q^j*prod([(q^(d-t+1)-1)/(q^t-1) for t in range(1,j+1)]) * \
      (1+q^(d-2*j)*p^f) * prod([(1+q^(d-i)*p^f)/(1+q^i/p^f)
      for i in range(1,j+1)])/(1+q^(d-j)*p^f)
# Calculation of valencies
def Hdpgk(p,f,d,j):
    q = p^f
    def b(i):
        return q^(2*i)*(q^(2*(d-i))-1)/(q+1)
    def c(i):
        return q^(i-1)*(q^i-(-1)^i)/(q+1)
    if j==0: return 1
    return b(j-1)*Hdpgk(p,f,d,j-1)/c(j)
# (1st ordering), returns Q-matrix for Hdpg
def Hdpg(p,f,d):
    q=p^(2*f)
    def ff(i,j):
       return  Hdpgk(p,f,d,j)*q_hyper(q,q,[q^-i,q^-j,-p^-f*q^(j-d)],
        [q^-d,0],ulim=i)
    return matrix(d+1,d+1,ff)
```

8.6 Appendix VI: Hermitian dual polar graphs Testing

```
def hdpg_test():
    improvements = 0
```

```
prime_powers = [2,3,4,5,7,8,9,11,13,16,17,19,23,25,27,29,31,32,37,41,
        43,47,49,53,59,61,64,67,71,73,79,81,83,89,97,101,103,107,109,
        113,121,125,127,128,131,137,139,149,151,157,163,167,169]
# tests for d = 2
d = 2
for q in prime_powers[1:]:
    Q = Hdpg(q,1,2*d-1)
    A,p,bd = codes.bounds.delsarte_bound_Q_matrix(Q.transpose(),2*d-1,
      return_data=True,isinteger=True)
    best = (q^3+q+2)/2
    if bd<best: improvements += 1
    print(f"q={q},d={d},new bound = {int(bd)}, best known = {best},{bd>best}")
# tests for d = 4
d = 4
for q in prime_powers:
    Q = Hdpg(q,1,2*d-1)
    A,p,bd = codes.bounds.delsarte_bound_Q_matrix(Q.transpose(),2*d-1,
      return_data=True,isinteger=True)
    best = q^(2*d-1)-q^(3*d/2)*(q^(0.5)-1)
    if bd<best: improvements += 1
    print(f"q={q},d={d},new bound = {int(bd)}, best known = {int(best)},
      {bd>best}")
# tests for d>4
for d in [6,8]:
    for q in prime_powers:
        Q = Hdpg(q,1,2*d-1)
        A,p,bd = codes.bounds.delsarte_bound_Q_matrix(Q.transpose(),2*d-1,
          return_data=True,isinteger=True)
        best = q^(2*d-1)-q*(q^(2*d-2)-1)/(q+1)
        if bd<best: improvements += 1
        print(f"q={q},d={d},new bound = {int(bd)}, best known = {int(best)},
          {bd>best}")
print(f"Total improvements: {improvements}")
```