A method for efficiently computing the number of codewords of fixed weights in linear codes

Iliya Bouyukliev\textsuperscript{a,}\textsuperscript{*}, Valentin Bakoev\textsuperscript{b}

\textsuperscript{a} Institute of Mathematics and Informatics, Bulgarian Academy of Sciences, P.O. Box 323, 5000 Veliko Tarnovo, Bulgaria
\textsuperscript{b} Department of Mathematics and Informatics, Veliko Tarnovo University, 2 Theodosi Tarnovski Street, 5000 Veliko Tarnovo, Bulgaria

Received 18 September 2006; received in revised form 20 December 2007; accepted 9 January 2008
Available online 4 March 2008

Abstract

The problem of computing the number of codewords of weights not exceeding a given integer in linear codes over a finite field is considered. An efficient method for solving this problem is proposed and discussed in detail. It builds and uses a sequence of different generator matrices, as many as possible, so that the identity matrix takes disjoint places in them. The efficiency of the method is achieved by optimizations in three main directions: (1) the number of the generated codewords, (2) the check whether a given codeword is generated more than once, and (3) the operations for generating and computing these codewords. Since the considered problem generalizes the well-known problems “Weight Distribution” and “Minimum Distance”, their efficient solutions are considered as applications of the algorithms from the method.

Keywords: Linear code; Codewords generating; Weight spectrum algorithm; Minimum distance algorithm

1. Introduction

Many problems in coding theory require an efficient computing of the weight spectrum or the number of codewords of fixed weight (in particular minimal weight) of a given linear code. These data are important for deriving some algebraic properties of the code and its capabilities of detecting and correcting errors.

Let $GF(q)^n$ be the $n$-dimensional vector space over the Galois field $GF(q)$. Every $k$-dimensional subspace $C$ of $GF(q)^n$ is called a linear $[n, k]$ code over $GF(q)$. The parameters $n$ and $k$ are called length and dimension of $C$, respectively, and the vectors in $C$ are called codewords. For an arbitrary codeword $v \in C$ its (Hamming) weight $wt(v)$ is defined as the number of its non-zero coordinates. If $A_i$ is the number of codewords of weight $i$ in $C$, $i = 0, 1, \ldots, n$, then the sequence $S = (A_0, A_1, \ldots, A_n)$ is called a weight spectrum of $C$. The minimum weight $d$ of $C$ is the smallest weight among all its non-zero codewords, and so $C$ is said to be a $[n, k, d]_q$ linear code. Every $k \times n$ matrix $G$, the rows of which are formed by $k$ linearly independent vectors of $C$, is called a generator matrix of the code $C$. If $(u, v) : GF(q)^n \times GF(q)^n \rightarrow GF(q)$ denotes an inner product in the linear space $GF(q)^n$, then

\textsuperscript{*} Corresponding author.
E-mail addresses: iliya@moi.math.bas.bg (I. Bouyukliev), v_bakoev@yahoo.com (V. Bakoev).

0166-218X/ - see front matter © 2008 Elsevier B.V. All rights reserved.
doi:10.1016/j.dam.2008.01.003
Testing the equivalence of codes and/or determining the automorphism group of a given linear code (a proper subset of codewords of fixed weights is used in testing linear codes for isomorphism);
• For some classes of codes (extremal self-dual codes, for example), the weight spectrum is a function of one or two parameters. Finding the number of codewords of fixed weights leads to determining these parameters. By them the whole weight spectrum of the code can be computed quite faster than by using other algorithms;
• The number of codewords of minimal weight is an essential part of the evaluating function in some heuristic algorithms for constructing optimal codes;
• It is proved that the codewords of certain weights in some specific codes can be considered as blocks of combinatorial designs. These codewords are necessary for finding the incidence matrix of the corresponding design.

In this paper we propose and discuss a method for solving the considered problem. Instead of “algorithm” we use the notion “method”, since it integrates three different key ingredients. They are realized by a sequence of algorithms, which differ in their type. The preliminary versions of some of the algorithms in the method are used in [8,9,7,10] for constructing codes with new parameters. The recent version of the method is implemented in the Q-Extension package, used for investigating linear codes [5].

This paper is organized as follows. The whole description of the method for solving the problem is given in Section 2. It begins with deriving some mathematical bases and giving the main idea of the method. Then the ingredients of the method are represented in three subsections. The first one treats optimization of the number of the linear combinations, which have to be generated for solving the problem. It is based on the building and usage of a sequence of different generator matrices, having disjoint information sets, as in the Brouwer–Zimmerman algorithm [3,11]. Some combinatorial properties of these matrices, concerning important parameters in generating codewords, are derived by Theorem 2.3. Finally in this part, the steps of the method are summarized. The minimization of the number of the generated linear combinations implies the possibility some codewords to be generated more than once. So this minimization requires a check whether a given codeword is generated more than once (i.e. a part of the running-time, gained by the minimization, compensates the running-time of the check). This check is given in the second subsection. The third subsection represents some algorithms for an efficient generation of the linear combinations and computing the codewords. They are separated in two groups, concerning binary and non-binary linear codes. The corresponding running-time is derived for each algorithm in the method. The general time-complexity of the method is estimated to \( O \left( k^3(n-k)^2 \sum_{j=1}^{i} (q-1)^{j-1} \binom{k}{j} \right) \), where \( i = O(m) \). Section 3 describes two applications of some algorithms from the method for solving the problems WEIGHT DISTRIBUTION and MINIMUM DISTANCE efficiently. In the last section some conclusions, concerning the effectiveness of the algorithms and their usage are drawn.

2. A method for solving the problem

We assume that \( C \) is a \([n,k]_q\) linear code, given by its generator matrix \( G \). We have to generate and to count all codewords of weights \( \leq m \), where \( m \) is a given integer. Without loss of generality we can consider the code \( C' \), which is obtained from \( C \) by permutation of its coordinates. We can do this because \( C' \) has the same weight spectrum as \( C \), and \( C \) can be obtained from \( C' \) by performing the inverse permutation. We assume that the generator matrix \( G \) of \( C \) is in systematic form, i.e. \( G = (I|A) \), where \( I \) is the identity matrix of rank \( k \), and \( A \) is a \( k \times (n-k) \) matrix (if \( G \) is not in systematic form, we reduce it to such a form by the well-known Gauss–Jordan algorithm, running in a polynomial time \( O(nk^2) \)).

Let the vector \( u \) be the linear combination \( v = \lambda_1 G[i_1] + \lambda_2 G[i_2] + \cdots + \lambda_m G[i_m] \), where \( 1 \leq m \leq k \), \( 1 \leq i_1 < \cdots < i_m \leq k \), \( \lambda_j \in GF(q) \), \( \lambda_j \neq 0 \), and \( G[i_j] \) is the \( (i_j) \)th row of the matrix \( G \), \( j = 1,2,\ldots,m \). Since \( G = (I|A) \), the vector \( u \) consists of two parts. The first \( k \) coordinates of it form the vector \( u_1 \), having coordinates \( \lambda_1, \lambda_2, \ldots, \lambda_m \) in positions \( i_1, i_2, \ldots, i_m \) and zeros in the rest positions. The last \( n-k \) coordinates of \( v \) form the vector \( u_2 = \lambda_1 A[i_1] + \lambda_2 A[i_2] + \cdots + \lambda_m A[i_m] \), where \( A[i_j] \) is the \( (i_j) \)th row of the matrix \( A \), \( j = 1,2,\ldots,m \). Therefore

\[
wt(v) = wt(u_1) + wt(u_2) = m + wt(u_2).
\] (1)

Equality (1) shows that any codeword of weight \( m \) is a linear combination of no more than \( m \) basis vectors. To count all these codewords we must obtain all linear combinations of \( 1,2,\ldots,m \) basis vectors. Therefore, computing the number of codewords of weight \( m \) means solving a more general problem: computing the number of all codewords
of weights smaller or equal to \( m \). When \( m = k \) the problem is actually reduced to computing the weight spectrum of the code.

So, by generating \((q - 1)\binom{k}{1} + (q - 1)^2\binom{k}{2} + \cdots + (q - 1)^m\binom{k}{m}\) linear combinations of the rows of the matrix \( A \), all codewords of weight \( \leq m \) will be obtained. As equality (1) shows, some linear combinations will result in unnecessary codewords of weights \( > m \). If \( q > 2 \) and the vector \( v_1 = A[i_1] + \lambda_2 A[i_2] + \cdots + \lambda_m A[i_m] \) is of weight \( w \), then the vectors \( v_2 = 2v_1, \ldots, v_{q-1} = (q - 1)v_1 \) are proportional to \( v_1 \) and they have the same weight \( w \). So, it will be \((q - 1)\) times more efficient if we generate only these linear combinations, where \( \lambda_1 = 1 \), i.e. only the non-proportional codewords. Their number is

\[
N_1 = \binom{k}{1} + (q - 1)^1\binom{k}{2} + \cdots + (q - 1)^{m-1}\binom{k}{m} = \sum_{j=1}^{m} (q - 1)^{j-1}\binom{k}{j}.
\]

Finally, the number of the obtained codewords of the same weight \( w \) has to be multiplied by \((q - 1)\), for \( w = 1, 2, \ldots, m \).

We use this rough idea in developing the following algorithms. We strive to achieve their efficiency by optimizing the number of:

1. linear combinations, which have to be generated (i.e. to decrease the number of the redundant linear combinations — these, which result in codewords of weight \( > m \));
2. operations in generating and computing these linear combinations.

These two key ingredients are considered and discussed in details in the following subsections.

2.1. Optimizing the number of linear combinations

Let us consider the same problem in the case of self-dual codes — let \( C \) be self-dual code (i.e. \( C = C^\perp \)), given by its generator matrix \( G_1 \) in a systematic form, \( G_1 = (I|A) \). It is well-known, that the matrix \( G_2 = (-A^T|I) \) is a generator matrix of the code \( C^\perp \), and so \( G_2 \) generates the same code \( C \). Let \( v \in C \) be of weight \( m \), \( u_1 \) be the vector, formed by the first \( k \) coordinates of \( v \), and \( u_2 \) — the vector, formed by the last \( k \) coordinates of \( v \). Therefore \( \min\{wt(u_1), wt(u_2)\} \leq \lfloor m/2 \rfloor \). So, it is more efficient to generate all linear combinations of no more than \( \lfloor m/2 \rfloor \) rows of \( G_1 \) and the same combinations of the rows of \( G_2 \), instead of generating all linear combinations of no more than \( m \) rows of \( G_1 \). In other words, for given \( t, 0 < t \leq k \), no more than \( t \) linear combinations of the rows of \( G_1 \) and \( G_2 \) are sufficient to obtain all codewords of weight \( \leq 2t + 1 \). This idea is known. A similar approach is used in the mentioned above algorithm for computing the minimal weight of a given linear code \([3,11]\). We shall develop and apply this idea when solving our problem for arbitrary linear codes.

Let \( C \) be a linear \([n, k]_q \) code, given by its generator matrix \( G \). We assume that \( C \) does not contain redundant zero coordinates. By the following algorithm we get the sequence of generator matrices \( G_1, G_2, \ldots, G_r \) of the code \( C' \), which is obtained by a permutation of the coordinates of \( C \). The matrices in the sequence are of the type:

\[
G_1 = (I_1|A_1), \ldots, G_j = \begin{pmatrix} A_j & I_j \\ O & A_j' \end{pmatrix}, \ldots, G_r = \begin{pmatrix} A_r & I_r \\ O' & O'' \end{pmatrix},
\]

where \( I_j \) is an identity matrix of rank \( k_j \), for \( j = 1, \ldots, r \), and also \( k = k_1 \geq k_2 \geq \cdots \geq k_r \). \( A_j \) is a \( k_j \times (n - k_1 - \cdots - k_{j-1}) \) non-zero matrix, for \( j = 1, \ldots, r - 1 \), and \( A_j' \) is a \( k \times (k_1 + k_2 + \cdots + k_{j-1}) \) matrix of rank \( k \), for \( j = 2, \ldots, r \). \( O' \) and \( O'' \) are zero matrices of appropriate sizes.

**Algorithm GetSGM.** Computes the sequence of generator matrices of the type (3).

**Input.** An arbitrary generator matrix \( G \) of the \([n, k]_q \) linear code \( C \).

**Output.** The sequence of generator matrices \( G_1, G_2, \ldots, G_r \) of the code \( C' \), which are of type (3). Also the sequence \( k_1, k_2, \ldots, k_r \), where \( k_j \) is the rank of the identity matrix \( I_j \) in \( G_j \), \( j = 1, \ldots, r \).

**Procedure:**
(1) Initialization. Put \( i = 0 \) and \( \text{perm}[j] = j \), for \( j = 1, \ldots, n \).

(2) Suppose that after the \( i \)th serial step the matrix \( G_i = \begin{pmatrix} A_i & I_i \\ O' & O'' \end{pmatrix} \) is obtained, where \( A_i' \) is a \( k \times (k_1 + \cdots + k_{i-1}) \) matrix of rank \( k \), \( I_i \) is the identity matrix of rank \( k_i \), \( A_i \) is a \( k_i \times (n - k_1 - \cdots - k_i) \) non-zero matrix, \( O' \) and \( O'' \) are zero matrices. If \( i = 0 \), then \( G = G_0 = (A_0) \).
(3) Apply the Gauss–Jordan elimination algorithm on the matrix \( A_j \) starting from position \( pos \), where \( pos = 1 \) if \( i = 0 \), or \( pos = k_1 + \cdots + k_i + 1 \) otherwise. During the elimination, when a \( j \)th column has to be replaced, choose a column with a greater number — for example the \( l \)th one, \( j < l \). Transpose the \( j \)th and the \( j \)th columns in all matrices \( G_1, \ldots, G_i \). Also put \( \text{perm}[j] = l \) and \( \text{perm}[l] = j \) to store this transposition of columns.

So the matrix \( G_{i+1} = \begin{pmatrix} A_{i+1}' & I_{i+1} \end{pmatrix} \) is obtained and the rank of the identity matrix \( I_{i+1} \) is \( k_{i+1} \).

(4) Put \( pos = pos + k_i + 1 \). If \( pos = n \) (i.e. \( A_{i+1} \) is a \( 0 \times 0 \) matrix) the algorithm finishes. Otherwise put \( i = i + 1 \) and go to step (3).

In step (3) the algorithm permutes columns not only in the last matrix \( G_i \), but in all current matrices \( G_1, \ldots, G_i \). So each matrix \( G_1, \ldots, G_r \) in the output is a generator matrix of the same code \( C' \), and this is an important fact for the following algorithms, as we shall see later. The codewords of \( C \) and these of \( C' \) are bijectively related by the permutation \( \pi \), determined by the array \( \text{perm} \). As we have noted, the both codes have one and the same weight characteristics. In case of necessity the array \( \text{perm} \) can be returned by the algorithm \( \text{GETSGM} \) and the codewords of \( C \) can be obtained by computing and applying the inverse permutation \( \pi^{-1} \) over the codewords of \( C' \).

The time-complexity of the algorithm \( \text{GETSGM} \) can be derived easily, it is \( O(kn^2) \).

For the code \( C' \), having generator matrices \( G_1, G_2, \ldots, G_r \) of the type (3), the following properties are valid:

**Lemma 2.1.** Let \( j \) be an integer, \( 1 \leq j \leq r \) (then \( O' \) is a \( (k - k_j) \times k_j \) zero matrix under the identity matrix \( I_j \) in \( G_j \)). Let \( i \) be an integer such that \( 1 \leq i \leq k \) and \( i - (k - k_j) > 0 \). If the vector \( v \in C' \) has no more than \( i - k + k_j \) non-zero coordinates in the positions from \((k_1 + \cdots + k_{j-1} + 1)\) to \((k_1 + \cdots + k_j)\) (i.e. these, corresponding to the placement of \( I_j \) in \( G_j \)), then \( v \) is a linear combination of no more than \( i \) rows of the generator matrix \( G_j \).

The truth of the assertion is obvious and we omit its proof.

**Theorem 2.2.** Let \( i \) be an integer, \( 1 \leq i \leq k \), and \( t = t(i) \) be the largest integer, such that \( 1 \leq t \leq r \) and \((i + 1 - k + k_i) > 0 \). If \( s = \sum_{j=1}^{t} (i + 1 - k + k_j) \), then any codeword \( v \in C' \) of weight \( \text{wt}(v) < s \), is a linear combination of no more than \( i \) rows of some of the generator matrices \( G_1, \ldots, G_i \) of \( C' \).

**Proof.** Let \( v \) be an arbitrary codeword of \( C' \), such that \( \text{wt}(v) < s \). In the worst case all coordinates of \( v \) in the interval \([k_1 + \cdots + k_i + 1, n]\) are zeros. Even this, according to the Dirichlet Principle, there exists an integer \( j \), \( 1 \leq j \leq t \), such that \( v \) has no more than \( i - k + k_j \) non-zero coordinates in the interval \([k_1 + \cdots + k_{j-1} + 1, k_1 + \cdots + k_j]\). In accordance with Lemma 2.1, \( v \) is a linear combination of no more than \( i \) rows of \( G_j \).

We shall apply Theorem 2.2 in generating all codewords of weight \( \leq m \) in \( C' \). However first we have to determine the integer \( t \) and the number \( i \) of the linear combinations (of the rows of \( G_1, \ldots, G_r \)) in dependence on \( m \). The following simple procedure \( \text{GETLNUM} \) does this. Its input are the integer \( m \) and the one-dimensional array \( \text{rank} \).

It represents the sequence of ranks \( k_1, \ldots, k_r \) (returned by the algorithm \( \text{GETSGM} \)) so that \( \text{rank}[j] = k_j \), for \( j = 1, \ldots, r \), and hence \( \text{rank}[1] \) stores the dimension of the code.

**GetLNUM (m, rank)**

1) \( i = 0; k = \text{rank}[1]; \)
2) \( \text{repeat} \)
3) \( i = i + 1; \)
4) \( t = 1; s = 0; \)
5) \( \text{while}(t \leq r) \text{ and } (i + 1 - k + \text{rank}[t] > 0) \text{ do} \)
6) \( s = s + (i + 1 - k + \text{rank}[t]); \)
7) \( t = t + 1; \)
8) \( \text{if } m < s \text{ then break; } \) \{prevents \( t \) and \( s \) to grow unnecessarily \} \)
9) \( \text{until } m < s; \)
10) \( t = t - 1; \)
11) \( \text{return } (t, i); \)

Obviously, this procedure runs in a polynomial time. By the following example we illustrate the assertion of Theorem 2.2 and the results of execution of procedure \( \text{GETLNUM} \).

**Example.** Let \( C \) be a \([200, 40]\) binary code given by its generator matrix \( G \). Let us assume that when \( G \) is an input of the procedure \( \text{GETSGM} \), it outputs a sequence of seven generator matrices \( G_1, \ldots, G_7 \), and the sequence of ranks,
corresponding to them is 40, 40, 35, 30, 25, 20, 10. When this sequence and \( m = 24 \) are input data of the procedure GetLCN\( \text{UM} \), it returns \( t = 3 \) and \( i = 9 \). Suppose that \( v \) is an arbitrary vector of \( C' \), \( \text{wt}(v) \leq 24 \), and (in the worst case) all non-zero coordinates of \( v \) are in the interval \([1, 115 = k_1+k_2+k_3]\). Since \( 24 = 9+10+5 \), or \( 24 = 10+9+5 \), or \( 24 = 10 + 10 + 4 \), we have:

- if \( v \) has \( \leq 9 = i - k + k_1 \) non-zero coordinates in the interval \([1, 40]\), it is a linear combination of no more than 9 rows of \( G_1 \);
- if \( v \) has \( \leq 9 = i - k + k_2 \) non-zero coordinates in the interval \([41, 80]\), it is a linear combination of no more than 9 rows of \( G_2 \), and
- if \( v \) has \( \leq 4 = i - k + k_3 \) non-zero coordinates in the interval \([81, 115]\), it is a linear combination of no more than 9 rows of \( G_3 \).

So, all linear combinations of no more than 9 rows of \( G_1, G_2 \) and \( G_3 \), are enough to obtain all codewords of \( C' \) of weights \( \leq 24 \).

We note, that the value of \( s \) is \( s = 25 = \sum_{j=1}^{3}(10 - k + k_j) \) and so \( m = s - 1 \). If we take \( m = 25 \), the procedure GetLCN\( \text{UM} \) returns \( t = 3 \) and \( i = 10 \), and so \( s = 28 \). We obtain the same results for \( m = 26 \) and \( m = 27 \). The following theorem shows how to optimize the number of linear combinations in the cases when \( m < s - 1 \).

**Theorem 2.3.** Let \( G_1, G_2, \ldots, G_r \) be a sequence of generator matrices of the code \( C' \) of type (3) and \( m \) be an integer, \( 1 \leq m \leq n \). Let \( t \) and \( i \) be the integers, computed by the procedure GetLCN\( \text{UM} \), \( s = \sum_{j=1}^{r}(i + 1 - k + k_j) \), and so \( m = s - 1 - h \), \( h \geq 0 \). Let \( G_{i_1}, \ldots, G_{i_h} \) be \( h \) generator matrices, chosen arbitrary among \( G_1, \ldots, G_r \). For each of them we put \( i_{j_1} = \cdots = i_{j_h} = i - 1 \), and for the rest matrices among \( G_1, \ldots, G_r \), we put \( i_{j_1} = \cdots = i_{j_{r-h}} = i \).

If we generate all linear combinations of no more than \( i_j \) rows of the matrix \( G_j \), for \( j = 1, 2, \ldots, t \), we obtain all codewords of \( C' \) of weights \( \leq m \).

This theorem generalizes the assertion of the previous one. Its proof is analogous to the proof of Theorem 2.2 and so we omit it.

**Method WeightsLEqM.** Computes the number of codewords of weights \( \leq m \).

**Input.** The integers \( n, k, q, m \) and the generator matrix \( G \) of the linear \([n, k]_q\) code \( C \).

**Output.** The array \( s[j] \) is the number of codewords of weight \( j \), for \( j = 1, 2, \ldots, m \) (since \( s[0] \) is always 1, it is not stored).

**Procedure.**

1. (1) Initialization. Put \( s[j] = 0 \), for \( j = 1, 2, \ldots, n \).
2. (2) Apply the algorithm GETSGM and obtain:
   (a) the sequence \( G_1, \ldots, G_r \) of generator matrices;
   (b) the sequence of ranks \( k_1, \ldots, k_r \) of the identity matrices \( I_1, \ldots, I_r \) in the corresponding generator matrices. Store it in the array \( \text{rank} \), so that \( \text{rank}[j] = k_j \), for \( j = 1, 2, \ldots, r \).
3. (3) Apply the procedure GetLCN\( \text{UM} \) and (in dependence of \( m \) and \( \text{rank} \)) obtain the integers \( t, i \) from Theorem 2.3. Compute the number \( h = \sum_{j=1}^{r}(i + 1 - k + k_j) - 1 - m \). If \( h > 0 \), choose the last \( h \) matrices \( G_{i-h+1}, \ldots, G_i \) to generate no more than \( i - 1 \) linear combinations over each of them (over each of the rest matrices \( G_1, \ldots, G_{i-h} \), no more than \( i \) linear combinations will be generated).
4. (4) Compute the sequence of inverted matrices \( T_i^{-1}, \ldots, T_{i-1}^{-1} \) (as it is shown in the following subsection).
5. (5) for \( l = 1 \) to \( t \) do GenCodewords \( (l) \);

To simplify the following explanations we consider that in step (3) we generate no more than \( i \) linear combinations over each of the generator matrices \( G_1, \ldots, G_i \) (i.e. we follow Theorem 2.2 instead of Theorem 2.3). The optimization in the cases when \( h > 0 \) is important, but it does not change the general type of the time-complexities, computed further.

Step (4) of the method is just outlined, its description is given in the following part.

The so called procedure GenCodewords \( (1) \) in step (5) integrates the performance of three important tasks. It generates all linear combinations of \( 1, 2, \ldots, i \) rows of \( G_i \) and computes each of them. When \( l > 1 \), it checks whether every obtained codeword \( v \) has been already generated by a previous matrix \( G_j \), \( j < l \) (step (4) serves this check). If “Not”, then it increases \( s[\text{wt}(v)] \) by one. So, the computational cost of this check is the “price”, which the method
“pays” for minimizing the number of the generated linear combination. The realizations of these three tasks are discussed in details in the following two subsections.

We note, that the first three steps of the method WEIGHTSLEQM run in a polynomial time, step (4) also does this, as we shall see further. So the efficiency of the method depends on the number and the realization of the operations, included in the procedure GenCodewords. They will be evaluated later.

2.2. The check for a new codeword

Here we describe the check whether a given codeword is a new one or not. We consider the linear combinations of the rows of the matrix \( G_i \), generated on a serial \( l \)th iteration of the cycle in step (5). Let the codeword \( v \) be an arbitrary linear combination of the rows of \( G_i \). When \( l = 1 \) each linear combination generates a new codeword. For \( l > 1 \) we have to check whether \( v \) has already been generated by some previous generator matrix among \( G_1, \ldots, G_{i-1} \). In the general case the sequence of ranks (obtained in step (2)(b) of the method) has the form \( k = k_1 = \cdots = k_j > k_{j+1} \geq \cdots \geq k_t \), where \( 1 \leq j \leq t \). So the sequence of generator matrices consists of two possible subsequences: \( G_1, \ldots, G_j \) and \( G_{j+1}, \ldots, G_i \). They have the following properties: each matrix from the first subsequence does not contain zero matrix under the corresponding identity matrix in it, and vice versa: for any \( h > j \), the zero matrix \( O_j^t \) under \( I_h \) in \( G_h \) contains \( k - k_h \) zero-rows. These properties imply two possible stages of check, concerning the corresponding subsequence. The check on the first stage tests whether \( v \) is a linear combination of the rows of some of the matrices \( G_1, \ldots, G_{i-1} \) when \( 2 \leq l \leq j + 1 \). This check just counts the non-zero coordinates of \( v \) in each of the intervals \([1, k_1], [k_1 + 1, k_2], \ldots, [k_{l-2} + 1, k_{l-1}] \) and compares their number with \( i \). If \( v \) contains \( \leq i \) non-zero coordinates in some of these intervals, then \( v \) is not a new codeword and the check finishes immediately (i.e. \( v \) is a linear combination over the matrix, corresponding to this interval — because \( G_1, G_2, \ldots, G_i \) are generator matrices of one and the same code \( C' \)). Otherwise \( v \) contains more than \( i \) non-zero coordinates in each of these intervals and \( v \) is a new codeword. Then the counting of the rest non-zero coordinates of \( v \) can continue in order the weight of \( v \) to be computed. So, for \( 2 \leq l \leq j + 1 \), the check finishes on the first stage. In the worst case, its computational cost is \( O(k_1 + \cdots + k_j) = O(k_j) \).

When \( j + 1 < l \leq t \), the check concerns both subsequences. It starts with its first stage, for every matrix \( G_1, \ldots, G_j \) (i.e. counting the non-zero coordinates of \( v \) in each interval \([1, k_1], \ldots, [k_{j-1} + 1, k_j] \)), as it has just been described. If \( v \) is not a linear combination of the rows of \( G_1, \ldots, G_j \), then the check continues with its second stage, concerning the second subsequence. We have noted that for \( h = j + 1, \ldots, t \), the zero matrix \( O_j^t \) under \( I_h \) in \( G_h \) contains \( k - k_h \) zero-rows. Some such rows can take part in the linear combination of the rows of \( G_h \) (as Lemma 2.1 shows) and this will not change the number of non-zero coordinates of \( v \) in the corresponding interval \([k_{h-1} + 1, k_h] \). That is why the check of the first stage is not correct here. We recall that \( v \) is a linear combination of the rows of \( G_j \) and let \( v = (v_1, v_2, \ldots, v_k, \ldots, v_n) \). Therefore \( v_1 G_1[1] + v_2 G_1[2] + \cdots + v_k G_1[k] \) is a linear combination of \( v \) over the matrix \( G_1 \), where \( G_1[1], \ldots, G_1[k] \) are its rows. To compute \( v \) as a linear combination over another matrix \( G_h \), for \( j + 1 \leq h \leq l - 1 \), we have to change the basis — a procedure, which is known from Linear Algebra. In other words, we have to express \( v \) as a linear combination of the rows of \( G_h \), for example \( v = u_1 G_h[1] + u_2 G_h[2] + \cdots + u_k G_h[k] = (u_1, u_2, \ldots, u_k, \ldots, u_n) \). Next we have to count the non-zero coordinates of \( v \) among \( u_1, u_2, \ldots, u_k \), i.e. these in the interval \([1, k] \). If their number is \( \leq i \), then \( v \) has been already generated by \( G_h \), otherwise it has not. Let us consider some details of the check in the second stage more precisely. If

\[
G_h = \begin{pmatrix}
G_{1,1} & G_{1,2} & \cdots & G_{1,k} & \cdots & \cdots & G_{1,n} \\
G_{2,1} & G_{2,2} & \cdots & G_{2,k} & \cdots & \cdots & G_{2,n} \\
\vdots & \vdots & & \vdots & \cdots & \cdots & \vdots \\
G_{k,1} & G_{k,2} & \cdots & G_{k,k} & \cdots & \cdots & G_{k,n}
\end{pmatrix}, \quad \text{then} \quad T_h = \begin{pmatrix}
G_{1,1} & G_{2,1} & \cdots & G_{k,1} \\
G_{1,2} & G_{2,2} & \cdots & G_{k,2} \\
\vdots & \vdots & & \vdots \\
G_{1,k} & G_{2,k} & \cdots & G_{k,k}
\end{pmatrix}
\]

is the transition matrix from the basis \( G_1[1], \ldots, G_1[k] \) to the basis \( G_h[1], \ldots, G_h[k] \). So

\[
\begin{pmatrix}
v_1 \\
v_2 \\
\vdots \\
v_k
\end{pmatrix} = T_h \begin{pmatrix}
u_1 \\
u_2 \\
\vdots \\
u_k
\end{pmatrix}, \quad \text{and therefore} \quad T_h = T_h^{-1} \begin{pmatrix}
T_h^{-1} \begin{pmatrix}
T_h^{-1} \begin{pmatrix}

v_1 \\
v_2 \\
\vdots \\
v_k
\end{pmatrix} = T_h^{-1} \begin{pmatrix}

v_1 \\
v_2 \\
\vdots \\
v_k
\end{pmatrix}.
\]
The sequence of transition matrices \( T_{j+1}, \ldots, T_t \) is determined by the matrices \( G_{j+1}, \ldots, G_t \). Their inverse matrices \( T_{j+1}^{-1}, \ldots, T_t^{-1} \) can be computed by a simple algorithm, based on the Gauss–Jordan elimination. This algorithm is also well-known in Linear algebra, it runs in a polynomial time \( O(k^3) \) when reverses a \( k \times k \) matrix. We exploit it to implement step (4) of the method WEIGHTSEQM. So the real check in the second stage is: for every \( h = j + 1, \ldots, l - 1 \) compute the first \( k \) coordinates \( u_1, \ldots, u_k \) of the vector \( v \) by the matrix \( T_h^{-1} \). Count the non-zero coordinates among them. If their number is \( \leq i \), then \( v \) is a linear combination over \( G_1 \) and the check finishes immediately. Otherwise it continues with the next value of \( h \). So, for a given \( h \), this check has a computational cost \( O(k^3) \). For a given vector \( v \) (over \( G_j \)) this check is applied at most \( l - 1 - (j + 1) + 1 = l - j - 1 \) times. In the worst case this is \( O(t - j) \) times, and so the check on the second stage has a computational cost \( O(k^3(t - j)) \). Therefore the running-time of entire check (whether \( v \) is a new codeword or not) is \( O(kj) + O(k^3(t - j)) = O(k^3(t - j)) = O(k^3(n - k)) \), because \( t \leq n - k \) and \( j \geq 1 \). When computing these time-complexities we assume that the addition and the multiplication in \( GF(q) \) are realized by a table in a constant time (as in the following algorithms).

2.3. Optimization of the operations in generating and computing the linear combinations

Here we represent some algorithms for generating and computing all necessary linear combinations, trying to minimize the number of the necessary operations and to implement them effectively. This was our only direction for optimization in the earlier stages of this research. We have developed some algorithms, which use only one generator matrix in a systematic form [6]. Here we represent their new versions and realizations, where the algorithms are a bit modified in accordance with the optimizations given above. They should replace the procedure, called most generally by us GenCodewords.

In our problem \( m < k \) in the general case and it is impossible to exploit the Gray code for generating only the necessary linear combinations. As we have noted, the Gray code is extremely useful in generating all subsets of a given \( k \)-element set, i.e. in computing the whole weight spectrum of the code. It is possible to use a known algorithm for generating all combinations of order \( j \) over a \( k \)-element set with a minimal change, for \( j = 1, 2, \ldots, i \). For obtaining each combination such algorithm replaces one element from the recent combination with another outside it, i.e. it performs two operations — inclusion and exclusion. We shall implement the generating by a single operation (inclusion) on each step. We represent our algorithms in Pascal language, using the following basic types:

```pascal
const maxr = 10; {maximal length of the sequence G_1, G_2, ..., G_r}

type Vector = array [1..n] of integer;
GMatrix = array [1..k] of Vector;
SGMatrices = array [1..maxr] of GMatrix;
```

When \( u \) and \( v \) are variables of type Vector, \( u + v \) denotes a componentwise addition over \( GF(q) \), and \( \alpha v \) denotes a multiplication of the components of \( v \) by the scalar \( \alpha \), as elements of \( GF(q) \). The generator matrix \( G \) can be represented by the type GMatrix, consisting of \( k \) Vector-rows. The type SGMatrices is used for representing the result of algorithm GETSGM — the sequence of generator matrices \( G_1, \ldots, G_r \). The sequence of inverse matrices \( T_{j+1}, \ldots, T_t \) can be represented in a similar way.

For convenience, we consider that the variable (array) \( A \) of type GMatrix represents the generator matrix \( G_i \), corresponding to the parameter \( l \) in the call GenCodewords(l). We use an additional array \( B \) of the same type GMatrix, where we shall obtain and store all resulting vectors of the linear combinations over \( A \). The row \( B[j] \) of the array \( B \) will store the result of any linear combination of exactly \( j \) rows of the array \( A \) (i.e. of \( G_i \)), for \( 1 \leq j \leq i \). For example, we can compute and put:

```pascal
\vdots
```

We use two additional arrays \( c \) and coef, both of type Vector, such that \( c \) stores the numbers of the rows from \( A \) in the linear combinations, and \( coef \) — their coefficients. So \( c[h] = j_h \), and \( coef[h] = \lambda_h \), for \( 1 \leq h \leq i \). Using
them we can obtain each linear combination by an increment of a certain \( c[h] \) or \( \text{coef}[j] \) in some previous linear combination, which result is already in the array \( B \). This idea for generating the linear combinations can be seen in a most clear form in the following two algorithms, called GCBIN and GCQARY. The first algorithm is a particular case of the second one, it runs in the case \( q = 2 \), i.e. for binary linear codes. Then the coefficients in (4) are: \( \lambda_j = 1 \), for \( j = 2, 3, \ldots, i \), and so the array \( \text{coef} \) becomes unnecessary. Algorithm GCBIN generates all linear combinations of no more than \( i \) rows of the array \( A \) in the array \( B \). It does this directly — by using \( i \) embedded cycles, which control the values in \( c[1], c[2], \ldots, c[i] \). The main part of algorithm GCBIN is:

```plaintext
for c[1]:= 1 to k do
begin
    B[1]:= A[c[1]]
    for c[2]:= c[1]+1 to k do
    begin
        for c[3]:= c[2]+1 to k do
        begin
            ....
            for c[i]:= c[i-1]+1 to k do
            B[i]:= B[i-1] XOR A[c[i]]
        ....
        end;
    end;
end;
```

Algorithm GCQARY works when \( q > 2 \), \( q \) is prime. It is built on the base of algorithm GCBIN, where the cycles are complemented with secondary embedded cycles to control the values of \( \text{coef}[j] \), representing \( \lambda_j \) in (4), \( j = 2, 3, \ldots, i \). We recall that always \( \lambda_1 = 1 \) and finally the number of obtained codewords must be multiplied by \( q - 1 \) (because of the proportional codewords). The main part of algorithm GCQARY is:

```plaintext
for c[1]:= 1 to k do
begin
    B[1]:= A[c[1]];
    for c[2]:= c[1]+1 to k do
    for coef[2]:= 1 to q-1 do
    begin
    ....
    for c[i]:= c[i-1]+1 to k do
    for coef[i]:= 1 to q-1 do
    if coef[i]=1 then B[i]:= B[i-1] + A[c[i]]
    else B[i]:= B[i] + A[c[i]];
    ....
end;
```

Obviously, for \( j = 2, \ldots, i \), each assignment \( B[j]:= B[j-1] + A[c[j]] \) corresponds to \( \lambda_j = 1 \), and \( B[j]:= B[j] + A[c[j]] \) — to the increment of \( \lambda_j \) by one in (4). In both algorithms each assignment in some \( B[j] \), \( 1 \leq j \leq i \), denotes that a codeword (new or not) is obtained. Thus we obtain all necessary vectors sequentially, performing at most one addition of two \( n \)-dimensional vectors.

The time-complexity of algorithms GCBIN and GCQARY is proportional (the coefficient of proportionality will be evaluated further) to the number of codewords, which each of them generates for a given matrix \( A = G_l \). Let us compute this number, following Theorem 2.2, the comments after the method WEIGHTSLEQM and the descriptions
of both algorithms. For every matrix $G_l$, $l = 1, 2, \ldots, t$, the algorithm GCBIN generates $\binom{k}{1} + \binom{k}{2} + \cdots + \binom{k}{i}$ codewords. Therefore, for binary linear codes, the number of codewords generated in step (5) of the method WEIGHTSLEQM is:

$$N_{\text{bin}} = t \left( \binom{k}{1} + \binom{k}{2} + \cdots + \binom{k}{i} \right) = t \sum_{j=1}^{i} \binom{k}{j}.$$ 

Similarly, for every matrix $G_l$, $l = 1, 2, \ldots, t$, the algorithm GCQARY generates only non-proportional codewords, which number is $\binom{k}{1} + (q - 1)^1 \binom{k}{2} + \cdots + (q - 1)^{i-1} \binom{k}{i}$. Therefore, when $q$ is prime and $q > 2$, the number of codewords generated by the method WEIGHTSLEQM is:

$$N_{\text{qary}} = t \left( \binom{k}{1} + (q - 1)^1 \binom{k}{2} + \cdots + (q - 1)^{i-1} \binom{k}{i} \right) = t \sum_{j=1}^{i} (q - 1)^{j-1} \binom{k}{j}.$$ 

Although the correctness of both algorithms is obvious, it can be proved strongly by induction on the number of the embedded cycles, i.e. on $i$.

Algorithm GCBIN contains exactly $i$ embedded cycles, and algorithm GCQARY exactly $2i - 1$ embedded cycles. So the algorithms are not appropriate for practical purposes — certain number of cycles have to be added to (or removed from) their programs for any change of $i$. In the following algorithms we eliminate this defect by a recursive and a non-recursive emulation of these embedded cycles.

2.3.1. The case $q = 2$

The addition of vectors in algorithm GCBIN and some other operations are separated in the procedure NextBinVector($j$). When $j > 1$ it adds the vectors $B[j - 1]$ and $A[c[j]]$ over $GF(2)$ and stores the result in $B[j]$ (i.e. it performs a bitwise sum modulo 2, XOR in Pascal), otherwise it puts $B[1] = A[c[1]]$. It calls the function ComputeWeight, which checks whether a given codeword has been already generated (this check was described in Section 2.2). If “Not” the function computes its weight and returns it, otherwise the function returns zero.

```pascal
Procedure NextBinVector (j : integer);
var w : integer;
begin
  if j=1 then B[1]:= A[c[1]]
  else
  w:= ComputeWeight (B[j]);
  if w>0 then Inc (s[w]);
end; {NextBinVector}
```

The first version of algorithm GCBIN is called GCBIN_REC. It emulates the cycles recursively by the following procedure:

```pascal
Procedure Bin_Rec (j : integer);
var r : integer;
begin
  for r:= c[j-1]+1 to k do
    begin
      c[j]:= r;
      NextBinVector (j);
      if j < i then Bin_Rec (j+1);
    end;
end; {Bin_Rec}
```

In the algorithm GCBIN_REC the array c is defined as array [0..n] of integer. Before the first call of the procedure Bin_Rec (1) we initiate $c[0] := 0$ and so we set the initial value for the first cycle, i.e. the most
external cycle, when \( j = 1 \). Obviously, each recursive call of Bin_Rec(\( j \)) executes the sequential iteration of the \( j \)th embedded cycle and the next vector is generated. The external variable \( i \) determines the number of the cycles and controls the depth of the recursive calls.

The next version of algorithm GCBIN is GCBIN_NONREC. It emulates the performance of \( i \) embedded cycles directly, without recursion. The essential part of it is:

```plaintext
Procedure Bin_NonRec (i : integer); { i gives the number of the cycles }
var j : integer;
begin
  j:= 1; c[j]:= 0;
  repeat
    c[j]:= c[j]+1;
    NextBinVector (j);
    if c[j] = k then j:= j-1
    else
      if j < i then
        begin
          j:= j+1; c[j]:= c[j-1];
        end;
  until j=0;
end; {Bin_NonRec}
```

Obviously, the last two algorithms emulate the performance of the embedded cycles in algorithm GCBIN correctly. They are equivalent to it (this fact can be proved strongly by induction on \( i \)). So they have the same time-complexity, which is proportional to the number of generated codewords, i.e. to \( \sum_{j=1}^{i} \binom{k}{j} \). The coefficient of proportionality is just the running-time of the procedure NextBinVector, since all its calls are prepared in a time bounded by a constant. This time is a sum of the following running-times: \( O(n) \) for computing the sequential codeword, \( O(k^3(n - k)) \) for the check whether it is a new one or not, and \( O(n) \) for computing its weight, i.e. it is \( O(k^3(n - k)) \) generally. Therefore both algorithms GCBIN_REC and GCBIN_NONREC have a time-complexity \( O \left( k^3(n - k) \sum_{j=1}^{i} \binom{k}{j} \right) \), when they run over some of the generator matrices \( G_l = A, 1 \leq l \leq t \). So, for binary linear codes, the method WEIGHTSLEQ solves the considered problem with a time-complexity \( O \left( t k^3(n - k) \sum_{j=1}^{i} \binom{k}{j} \right) \), where \( t = O(n - k) \) and \( i = O(m) \).

In comparison, if the algorithms GCBIN_REC and GCBIN_NONREC use a single generator matrix \( G' = (I|A) \) in a systematic form, they will generate \( \binom{k}{1} + \binom{k}{2} + \cdots + \binom{k}{m} = \sum_{j=1}^{m} \binom{k}{j} \) codewords of the code \( C' \) (see equality (2) for \( q = 2 \)). So they solve the considered problem in \( O \left( (n - k) \sum_{j=1}^{m} \binom{k}{j} \right) \) running-time, because all codewords are generated only once and the procedure NextBinVector does not check them. It also generates the linear combinations over the matrix \( A \), in accordance with equality (1) and the notes before it.

2.3.2. The case \( q > 2 \)

We represent the procedure NextVector in three versions before emulating the cycles in algorithm GCQR. Similarly to NextBinVector, NextVector1 works in accordance with equalities (4), but it adds two vectors over \( GF(q) \) when \( q > 2 \) and \( q \) is prime. It also uses the function ComputeWeight to check whether a given codeword has been already generated or not, and it does this in the same way as NextBinVector.

```plaintext
Procedure NextVector1 (j : integer);
var h, r, w : integer;
begin
  0) r:= c[j];
  1) if j=1 then
     2) for h:= 1 to n do B[1][h]:= A[r][h] { or simply B[1]:= A[c[1]] }
  3) else
```
4) if coef[j]=1 then
5) for h:= 1 to n do B[j][h]:= Add [B[j-1][h], A[r][h]]
6) else
7) for h:= 1 to n do B[j][h]:= Add [B[j][h], A[r][h]]
8) w:= ComputeWeight (B[j]);
9) if w>0 then Inc (s[w]);
end; {NextVector1}

The array Add represents addition of integers in the field $GF(q)$, so Add[i,j] is preliminarily set to be $(i + j) \mod q$, for $0 \leq i, j \leq q-1$. Procedure NextVector1 computes the sequential vector in $B[j]$ in dependence of the parameter $j$: when $j = 1$ it puts $B[1] = A[c[1]]$, otherwise it adds (over $GF(q)$) and puts either $B[j] = B[j-1] + A[c[j]]$, or $B[j] = B[j] + A[c[j]]$, in dependence of whether coef[j] = 1, or coef[j] > 1, respectively.

When $q$ is not prime this approach is not correct. More precisely, if $\lambda_j > 1$ then $\lambda_j.A[c[j]]$ is not always equal to $(\lambda_j - 1).A[c[j]] + A[c[j]]$ and the assignment $B[j] = B[j] + \lambda_j.A[c[j]]$ is not correct. So, when $q = p^s, p$ is prime and $s > 1$, we always have to compute (over $GF(p^s)$) and put $B[j] = B[j-1] + \lambda_j.A[c[j]]$. In this case we define and use the array Mult[i,j], representing the multiplication in $GF(p^s)$. For any $i, j \in GF(p^s)$ we compute (in $GF(p^s)$) and put: $Add[i, j] = i + j$, and $Mult[i, j] = i.j$. Then the only change, that we have made in the procedure NextVector1 is in row 7) — replace it by

7) for h:= 1 to n do B[j][h]:= Add [B[j-1][h], Mult [coef[j], A[r][h]]];

Afterward we obtain the procedure NextVector2. It can be simplified by removing the rows 4), 5) and 6). The resulting procedure NextVector3 stays correct, but it works a bit slower than NextVector2.

Procedure NextVector3 (j : integer);
var h, r, w : integer;
begin
  r:= c[j];
  if j=1 then
    for h:= 1 to n do B[1][h]:= A[r][h] { or simply B[1]:= A[c[1]] }
  else
    for h:= 1 to n do B[j][h]:= Add [B[j-1][h], Mult [coef[j], A[r][h]]];
    w:= ComputeWeight (B[j]);
    if w>0 then Inc (s[w]);
end; {NextVector3}

The following two algorithms emulate the embedded cycles in GCQARY. They use NextVector as a common name of all three procedures, given above. So, depending on whether $q$ is prime or not, the corresponding version of the procedure replaces NextVector. The type of $q$ does not need any changes in the algorithms to be made. Here is the code of algorithm GCQARY_REC, which emulates the cycles in GCQARY recursively.

Procedure GCQary_Rec (i : integer); { i gives the number of the cycles }
Procedure Q_Rec (j : integer);
var r, lambda_j : integer;
begin
  for r:= c[j-1]+1 to k do
    begin
      c[j]:= r;
      for lambda_j:= 1 to q-1 do
        begin
          coef[j]:= lambda_j;
          NextVector (j);
          if j < i then Q_Rec (j+1);
        end
    end
begin {GCQary_Rec}
coef[1]:= 1;
for c[1]:= 1 to k do { separates \(\lambda_1=1\) from the other lambdas }
  begin
    NextVector (1);
    if i > 1 then Q_Rec (2);
  end;
end; {GCQary_Rec}

Algorithm GCBIN_REC is a particular case of GCQARY_REC when \(q = 2\). On the contrary of GCBIN_REC, algorithm GCQARY_REC uses the array coef and controls its values during the generation. The assignment \(coef[1]=1\) and the cycle for \(c[1]:= 1\) to \(k\) do... (in the body of the procedure GCQary_Rec) are separated from the rest ones. So \(coef[1] = \lambda_1 = 1\) is not changed and only the non-proportional codewords are generated.

The next version of algorithm GCQARY is GCQARY_NONREC. It emulates the performance of the embedded cycles directly, without recursion.

Procedure GCQary_nonRec (i : integer); { i gives the number of the cycles }
var j, h : integer;
begin
  for j:= 1 to i do coef[j]:= 0; { initialization }
  h:= q-1;
  for c[1]:= 1 to k do { separates \(\lambda_1=1\) from the other lambdas }
    begin
      j:= 1; coef[1]:= 0;
      repeat { emulation of the cycles from the second to the i-th }
        if coef[j] < h then coef[j]:= coef[j]+1
        else
          begin
            c[j]:= c[j]+1; coef[j]:= 1;
            end;
        NextVector (j);
      if c[j] < k then
        begin
          if j < i then
            begin
              j:= j+1; c[j]:= c[j-1]+1;
              if coef[j] = h then coef[j]:= 0;
            end;
          end
        else if coef[j] = h then j:= j-1;
      until j = 1;
    end;
end; {GCQary_NonRec}

Obviously, the last two algorithms are equivalent to the algorithm GCQARY. Their correctness can be proved strongly by induction on \(i\). Analogously to the previous case, both algorithms GCQARY_REC and GCQARY_NONREC have the same time-complexity. As in the algorithm GCQARY, it is proportional to the number of generated codewords, which is \(\binom{k}{1} + (q - 1)^1 \binom{k}{2} + \cdots + (q - 1)^{i-1} \binom{k}{i} = \sum_{j=1}^{i} (q - 1)^{j-1} \binom{k}{j}\). The calls of procedure NextVector are prepared in time, bounded by a constant. So its running-time is
the coefficient of proportionality again. This running-time is computed in the same way, as in the procedure NextBinVector, it has the same components and summands. Hence it is generally $O(k^3(n - k))$. We note, that the multiplications in NextVector2 and NextVector3 do not change the type of this time-complexity, since they (as the additions in $GF(q)$) are realized by a table. In other words, these two procedures perform $O(n)$ table references more than NextVector1 to compute a codeword. So, the running-time of the last two algorithms is $O\left(k^3(n - k)\sum_{j=1}^{n}(q - 1)^{j-1}\left(\begin{array}{c}k \\ j\end{array}\right)\right)$, when they work with a generator matrix $A$ among $G_1, \ldots, G_t$. Therefore, for $q > 2$, the method WEIGTESTEQM solves the considered problem with a time-complexity $O\left(t\cdot k^3(n - k)\sum_{j=1}^{n}(q - 1)^{j-1}\left(\begin{array}{c}k \\ j\end{array}\right)\right)$, where $t = O(n - k)$ and $i = O(m)$ again.

In comparison, let us solve the same problem (for $q > 2$) by using some of the algorithms GCQARY_REC or GCQARY_NONREC, working with a single generator matrix $G' = (I|A)$ in a systematic form (as their earlier versions do this in [6]). We set the parameter $i = m$ and so the algorithms generate all $\sum_{j=1}^{m}(q - 1)^{j-1}\left(\begin{array}{c}k \\ j\end{array}\right)$ non-proportional codewords of the code $C'$. Similarly to the previous case, each linear combination is over the rows of the matrix $A$ and it is generated only once. So the running-time of such procedure NextVector is $O(n - k)$, and for the time-complexity of both algorithms we obtain $O\left((n - k)\sum_{j=1}^{m}(q - 1)^{j-1}\left(\begin{array}{c}k \\ j\end{array}\right)\right)$.

3. Applications

The proposed method and its algorithms can be applied for solving some important problems, discussed at the beginning of this paper and related to the problem, considered here.

3.1. Computing the weight spectrum of the code

We have already mentioned the applications of the $q$-ary Gray codes in solving the problem WEIGHT DISTRIBUTION in [13,15] and also the time-complexities of the corresponding algorithms. We also have mentioned that this problem is a particular case of the problem, considered here, when $m = k$. For computing the weight spectrum of a given linear code we can apply some of the algorithms, given in Sections 2.3.1 and 2.3.2, as follows:

**Algorithm Weight Spectrum.** Computes the weight spectrum of the linear code $C$.

**Input.** The integers $n, k, q$ and the generator matrix $G$ of the linear $[n, k]_q$ code $C$.

**Output.** The array $s$, where $s[j]$ is the number of codewords of weight $j$, for $j = 1, 2, \ldots, n$.

**Procedure.**

1. Initialization:
   a. put $s[j] = 0$, for $j = 1, 2, \ldots, n$;
   b. put $A = G$ (or read the generator matrix $G$ into the array $A$);
   c. for any $x, y \in GF(q)$ compute and put $Add[x, y] = x + y$ in $GF(q)$;
   d. if $q > 2$ and $q$ is not prime, compute and put $Mult[x, y] = xy$ in $GF(q)$, for any $x, y \in GF(q)$.

2. In dependence of $q$ run some of the algorithms:
   a. GCBIN_REC (K) or GCBIN_NONREC (K), if $C$ is a binary code. In the function ComputeWeight (used by the procedure NextBinVector) remove the check whether a given codeword is new or not.
   b. GCQARY_REC (K) or GCQARY_NONREC (K), if $q > 2$. If $q$ is prime use the procedure NextVector1, otherwise use NextVector2 or NextVector3. In the used procedure, in the function ComputeWeight, remove the check for a new codeword.

We note, that the dimension $k$ replaces the formal parameter $i$ in every algorithm, used in step (2). All these algorithms can also run on an arbitrary generator matrix $G$, which is not in a systematic form. Then the algorithm WEIGHTSPECTRUM computes the weight spectrum of the code $C$ with a time-complexity $O\left(n\sum_{j=1}^{k}(q - 1)^{j-1}\left(\begin{array}{c}k \\ j\end{array}\right)\right) = O(nq^{k-1})$. When $G = (I|A)$ is a systematic generator matrix, the generating can be done only over the rows of the matrix $A$. So the time-complexity reduces to $O((n - k)q^{k-1})$. 

3.2. Computing the minimum distance of the code and the number of codewords of minimum weight

This application of the method WEIGHTSLEQM is called CODEWORDSMINW and it computes the minimum distance and the number of codewords of minimum weight of given linear code \( C \). So it solves the problem MINIMUM DISTANCE in particular. We consider that this application is also a method, which represents a modification of WEIGHTSLEQM. It is also based on the generator matrices of the sequence \( G_1, \ldots, G_r \) and their properties, given in Theorem 2.2. From another point of view, this method quite resembles the algorithm MINDIST of A. Brouwer, K.-H. Zimmermann etc. [3,11].

Method CodewordsMinW. Computes the minimum distance and number of codewords of minimum weight of the linear code \( C \).

Input. The integers \( n, k, q \) and the generator matrix \( G \) of the linear \([n,k]_q\) code \( C \).

Output. The minimum distance \( d \) and the number of codewords of minimal weight \( s[d] \).

Procedure.

(1) Initialization. Put \( s[j] = 0 \), for \( j = 1, 2, \ldots, n \).

(2) Apply the algorithm GETSGM and obtain:
   (a) the sequence \( G_1, \ldots, G_r \) of generator matrices;
   (b) the sequence of ranks \( k_1, \ldots, k_r \) of the identity matrices \( I_1, \ldots, I_r \) in the corresponding generator matrices.

   Store the sequence of ranks \( k_1, \ldots, k_r \) in the one-dimensional array \( \text{rank} \), so that \( \text{rank}[j] = k_j \), for \( j = 1, \ldots, r \), and hence \( k[1] \) stores the dimension of the code. Let \( j \) be an integer, such that \( 1 \leq j \leq r \) and \( k_1 = \cdots = k_j > k_{j+1} \geq \cdots \geq k_r \).

(3) Compute the sequence of inversed matrices \( T_{j+1}^{-1}, T_r^{-1} \), as it is described in Section 2.2.

(4) perform the following procedure Computing to compute the minimal distance \( d \) and the number of codewords of minimal weight of the code \( C \).

Procedure Computing;

\[
\begin{array}{l}
\text{var } d, i, j, k, s, t : \text{integer}; \\
\quad \text{IsFound} : \text{boolean}; \\
\text{begin} \\
\quad i := 0; \quad k := \text{rank}[1]; \quad \text{IsFound} := \text{false}; \\
\quad \text{while not IsFound do} \quad \text{begin} \\
\quad \\
\quad \quad i := i + 1; \\
\quad \quad t := 1; \quad \text{sum} := 0; \\
\quad \quad \text{while } i + 1 - k + \text{rank}[t] > 0 \text{ do} \quad \text{begin} \\
\quad \quad \\
\quad \quad \quad \text{sum} := \text{sum} + i + 1 - k + \text{rank}[t]; \quad t := t + 1; \\
\quad \quad \quad \text{end}; \\
\quad \quad t := t - 1; \\
\quad \quad \text{for } j := 1 \text{ to } t \text{ do GenCodewords (j)}; \\
\quad d := 1; \\
\quad \text{while } s[d] = 0 \text{ do } d := d + 1; \\
\quad \text{if } d < \text{sum} \text{ then } \text{IsFound} := \text{true}; \\
\quad \text{end}; \\
\quad s[d] := s[d] * (q - 1); \\
\quad \text{end}; \quad \{\text{Computing}\}
\end{array}
\]

Here only the procedure Computing needs some explanations. For it \( \text{rank} \) and \( s \) are global arrays. The integer \( m \) is not known preliminarily (i.e. the minimum distance \( d \) of the code is not known). Hence \( i \) is an independent variable, it determines the maximal number of the rows in the linear combinations, which have to be generated on each step of the main cycle in row 2. So \( i \) runs through the values 1, 2, 3, etc., and for every such \( i \), the corresponding integers \( t = t(i) \) and \( s = s(i) \) from Theorem 2.2 are determined in rows 5, \ldots, 10 (note that \( s = s(i) \) is represented by the variable \( \text{sum} \), since the array \( s \) represents the spectrum). These rows are taken from the procedure GETLCNUM, or
more precisely, the recent method uses a modification of it. Thereafter, the cycle in row 11 (the same as step (5) of the method \textsc{WeightsLEQM}) generates and computes all linear combinations of 1, 2, \ldots, i rows of each of the generator matrices \( G_1, G_2, \ldots, G_i \). This is done by the procedure \textsc{GenCodewords (3)}, which works in the same way as in the method \textsc{WeightsLEQM}. The minimal weight among all generated codewords is computed in the variable \( d \) by rows 12 and 13. In every step of the main cycle (i.e. for every value of \( i \)), an upper bound of the minimum distance is determined in the variable \( d \), and its lower bound — in the variable \( \text{sum} \). Obviously, when \( i \) grows, then the values of \( \text{sum} \) and \( t \) also grow, whereas these of \( d \) decrease monotonically. When (for some \( i \)) the value of \( d \) becomes smaller than the value of \( \text{sum} \), then the minimum distance \( d \) and the number of codewords of minimal weight are determined. Then the condition in row 14) becomes “true”, the flag \( \text{IsFound} \) is set to “true” and so the main cycle is terminated.

When only the minimum distance of the code has to be determined, rows 12, 13 and 14 become unnecessary. Then \( d \) and \( \text{IsFound} \) should be global variables for the procedures \textsc{GenCodewords} and \textsc{NextVector}. Every time when a new codeword is obtained and its weight \( w \) is computed, \( w \) is compared with the current value of \( \text{sum} \). If \( w < \text{sum} \), then \( \text{IsFound} \) is set to “true”, \( d \) is set to be \( \omega \), and the procedure \textsc{GenCodewords} is terminated.

We have some remarks, concerning the generating of the linear combinations in the procedure \textsc{Computing}. We use the procedure \textsc{GenCodewords (in row 11)} to generate them, and so, most of them are generated more than once — for every \( i \), the number of rows in the linear combinations varies from 1 to \( i \). If the minimum distance is achieved for some \( i = h \), we obtain that \( \sum_{i=1}^{h} \left( \binom{k}{i} + (q-1) \binom{k}{2} + \cdots + (q-1)^{i-1} \binom{k}{i} \right) \) linear combinations are generated. As we have mentioned, another possible choice is to use an algorithm for generating all \( i \)-combinations over a \( k \)-element set with a minimal change (toward the last generated combination). Then each linear combination (of exactly \( i \) rows of each of the matrices \( G_1, G_2, \ldots, G_i \)) will be generated once. But the computational cost of this approach is the performance of two operations for obtaining every next linear combination — exclusion and inclusion of certain elements (rows) to the last one. In fact, this means that every linear combination will be generated twice, i.e. their number will be \( 2 \left( \binom{k}{i} + (q-1) \binom{k}{2} + \cdots + (q-1)^{i-1} \binom{k}{i} \right) \) (since the minimum distance is achieved for \( i = h \)). In the case when \( q > 2 \), \( q \) is prime and only certain \( \lambda_j \) in the last linear combination has to be increased, we can have an inclusion only.

Another approach, which resembles the heuristic technique “hill climbing”, is also possible. We can summarize it as follows: use the method \textsc{WeightsLEQM} and compute the minimal weight among the rows of all generator matrices \( G_1, G_2, \ldots, G_i \) (the probabilistic algorithm of Leon [16] starts in a similar way). If, for example, this minimal weight is \( w \), then put \( m = w \). The method \textsc{WeightsLEQM} is modified preliminarily, so that every time when a codeword of smaller weight \( w' \) is generated, it puts \( m = w' \), recomputes the integers \( i \) and \( t \), and continues (to generate, to compute and to check the codewords) in the known way. When the method finishes, the last value of \( w' \) is the minimum distance of the code \( C' \), and \( s[w'] \) is the number of codewords of minimal weight.

Deriving the time-complexities of these three approaches is not short and simple, and we can obtain only rough estimations for them. So we do not represent them here. The main conclusion, which we can do is: the effectiveness of these approaches depends on the parameters of the code and the generator matrix, given on the input. For some codes (and parameters) one of the approaches is better than the rest ones. For other codes (and parameters) another approach is better. So, the same will be valid for the time-complexity of the entire method \textsc{CodeWordsMinW}. Its estimation can be derived, but it will be very rough — for example of the same type, as the method \textsc{WeightsLEQM} has for \( m = k \). This estimation does not have a practical value, since in the general case the real running-time of the method is many times smaller. The reasons are: on each iteration, the main cycle in row 2) computes some important intermediate parameters, such as \( i, t, \text{sum}, d \). They control the iterations of the main cycle and the secondary cycle (in row 11), but they cannot be estimated precisely. The time-complexity of the algorithm \textsc{Mindist} of Brouwer, K.-H. Zimmermann etc. is also not given in [3] — probably because of the same reasons.

4. Conclusions and some experimental results

Here we have considered the problem of computing the number of codewords of weights not exceeding a given integer \( m \) of a given \([n, k]_q \) linear code \( C \). This problem generalizes both problems \textsc{WeightDistribution} and \textsc{MinimumDistance}, and it is also important in solving some other problems in coding theory. We have considered and represented the method \textsc{WeightsLEQM}, which continues and enlarges our previous investigations for solving this problem efficiently [6]. Some steps of the method are realized by known algorithms, and for the rest steps
some new algorithms and their realizations have been proposed and discussed in details. The time-complexities of these algorithms and their estimations have been derived. Their precise estimations contain some intermediate results (such as the integers \( r, i, t, j \), the sequence \( k_1, k_2, \ldots, k_r \)), since the output of some algorithms is an input for other ones. These results can be bounded from above by the input data only roughly, so the final estimations for the time-complexities of the given algorithms are even rougher. In practice the real running-time of the method is many times better than the computed one. When solving the same problem by algorithms, which work with a single generator matrix (as it has been described in the end of Sections 2.3.1 and 2.3.2), we have derived precise time-complexities.

Realizations of the methods and the algorithms, represented here, are included in the package Q-Extension (written by the first author and accessible in [5]). We have done many experiments with them, using a computer with 1.84 GHz, 32-bits AMD processor, under Red Hat Linux, where Q-Extension is compiled by Kylix 3. For the tests we have used many codes (randomly generated by us), having different parameters. For comparison, we have done experiments for solving some of the same problems by MAGMA 2.11-13 (students version), on the same computer and for the same codes. Table 1 represents the obtained running-times (in seconds) of the used algorithms for 18 codes as follows:

1. the first column contains the parameters \([n, k, d]_q\) of the codes;
2. the second column contains the running-times for solving the problem MINIMUM DISTANCE by Q-Extension (program q_ext_tools.exe), where the method CEDWORDS\textsc{MinW} (p. 2: “Find minimum distance” of the submenu “Weights” — p. 1 in main menu) applies the mentioned above approach “hill climbing”. The eighth column contains the corresponding running-times for solving the same problem by MAGMA (by the function “MinimumWeight”);
3. the columns from the third to the sixth contain the running-times for solving the main problem “computing the number of codewords of weights \( \leq m \)”, for \( m = d \), \( m = d + 1 \), \( m = d + 3 \) and \( m = d + 5 \), respectively, by the method WEIGHTS\textsc{LEQ}M (p. 6: “Find the number of codewords of weights \( \leq w \)” of the submenu “Weights”) in Q-Extension;
4. the seventh column contains the running-times for solving the problem WEIGHT DISTRIBUTION by the algorithm WEIGHT\textsc{Spectrum} (p. 1: “Find the spectrum of linear codes” of the submenu “Weights”) in Q-Extension. The last column contains the corresponding running-times for solving the same problem by MAGMA (by the function “Weight\textsc{Distribution}”);
5. the empty cells denote, that the corresponding running-times are greater than 8 h.

As we can see, the running-times of Q-Extension and MAGMA for solving one and the same problems, for the codes in Table 1, are comparable.

<table>
<thead>
<tr>
<th>([n, k, d]_q)</th>
<th>Q-Extension</th>
<th>MAGMA</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MinDist.</td>
<td>WeightDistr.</td>
</tr>
<tr>
<td>[120, 40, 23]_2</td>
<td>3</td>
<td>57</td>
</tr>
<tr>
<td>[120, 40, 23]_2</td>
<td>4</td>
<td>36</td>
</tr>
<tr>
<td>[100, 25, 30]_2</td>
<td>16</td>
<td>65</td>
</tr>
<tr>
<td>[100, 25, 32]_2</td>
<td>30</td>
<td>80</td>
</tr>
<tr>
<td>[100, 25, 34]_2</td>
<td>42</td>
<td>203</td>
</tr>
<tr>
<td>[100, 33, 20]_2</td>
<td>51</td>
<td>790</td>
</tr>
<tr>
<td>[100, 33, 19]_2</td>
<td>51</td>
<td>790</td>
</tr>
<tr>
<td>[100, 33, 19]_2</td>
<td>30</td>
<td>548</td>
</tr>
<tr>
<td>[150, 28, 39]_2</td>
<td>&lt;1</td>
<td>2</td>
</tr>
<tr>
<td>[200, 28, 60]_2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>[250, 28, 82]_2</td>
<td>6</td>
<td>9</td>
</tr>
<tr>
<td>[150, 18, 64]_3</td>
<td>3</td>
<td>9</td>
</tr>
<tr>
<td>[200, 18, 93]_3</td>
<td>9</td>
<td>19</td>
</tr>
<tr>
<td>[250, 18, 123]_3</td>
<td>22</td>
<td>55</td>
</tr>
<tr>
<td>[150, 15, 74]_4</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>[200, 15, 113]_4</td>
<td>22</td>
<td>82</td>
</tr>
<tr>
<td>[250, 15, 145]_4</td>
<td>37</td>
<td>97</td>
</tr>
</tbody>
</table>
On the base of the experimental results for the given algorithms we draw the following conclusions (Table 1 illustrates some of them):

- when $m$ is close to $d$, the running-time is close to one for computing the minimum distance of the code. Increasing $m$, the corresponding running-times increase exponentially (in accordance with the time-complexities, computed above);
- for most of the codes, the usage of the methods CODEWORDSMinW and WEIGHTSLEQM (especially when $m$ is close to $d$), is quite more efficient in comparison with solving the same problems by computing the whole weight spectrum of the code (the empty cells confirm this conclusion);
- there are codes (for example these with very small $k$ toward $n$ and $d$; see the codes $[250, 18, 23]_3$ and $[250, 15, 140]_4$), where computing the weight spectrum is more efficient in comparison with computing even the minimum distance of these codes. The number $i$ from Theorems 2.2 and 2.3, for such codes, becomes close to (or large than) $k/2$ ($i$ can be estimated roughly as $d/(n/k)$). So the usage of many generator matrices is more complex, time-consuming and not justified, in comparison with the algorithm WEIGHTSPECTRUM, which is quite more simple, uses a single generator matrix and does not check the codewords. The problem here is to choose the appropriate algorithm for solving the desired problem, especially when $d$ is not known preliminarily;
- the recursive and the non-recursive versions of the given algorithms have almost the same running-times;
- the running-times of the algorithms in Q-Extension, for codes with equal parameters and different generator matrices, may be equal, as well as quite different (see the codes $[120, 40, 23]_2$, and $[100, 33, 19]_4$);
- for $q = 2, 3, 4$ we use a bit-wise representation of the generator matrices and the generated vectors as arrays of computer words of maximal length. We have developed effective realizations of vector operations over the corresponding fields $GF(q)$. They make the algorithms to run more than 10 times faster.

As we have shown, the weight spectrum of a given linear code can be computed efficiently by some of the algorithms: GCбин_REC or GCбин_NONREC, when the code is binary, or GCQARY_REC or GCQARY_NONREC, when $q > 2$. Their running-time is not worse than the one of the other known to us algorithms. The represented procedures, implementing these algorithms, are short, clear and easy to use (this was one of the goals of this paper).

By some simple modifications of the method WEIGHTSLEQM we have obtained the method CODEWORDSMinW, which computes the minimum distance and (in a case of necessity) the number of codewords of minimum weight of a given linear code. If we denote by $O(f(n, k, q))$ the time-complexity of the method for computing the minimum distance, the time-complexity for computing the number of codewords of minimal weight will be $\Theta(f(n, k, q))$. When the method CODEWORDSMinW computes the minimum distance of linear codes, it is analogous to the known algorithm MINDIST [3,11].

Acknowledgments

The first author was partially supported by the Bulgarian NSF under Contract MM1304/2003. The second author was partially supported by Scientific Researches Fund of V. Tarnovo University under Contract RD620-02/2006.

References


