UNIVERSITY OF TARTU
Institute of Computer Science
Computer Science Curriculum

Mart Simisker

Study of Optimal Linear Batch Codes

Bachelor's Thesis (9 ECTS)

Supervisor: Vitaly Skachek, PhD

Tartu 2017

Study of Optimal Linear Batch Codes

Abstract:

Linear batch codes can be used for load balancing in distributed storage systems. In order to obtain efficient performance, it is important to have codes with optimized parameters, which is a complicated mathematical problem.

Specifically, in this thesis, algorithms and software for searching for linear batch codes are presented. Two upper bounds for systematic linear batch codes are derived. The shortest lengths of systematic linear batch codes, which have been found with the help of the software, are compared to known upper and lower bounds.

Keywords: Coding theory, linear batch codes, bounds on the code parameters, distributed data storage

CERCS: P175 Informatics, systems theory

Optimaalsete lineaarsete partiikoodide uuring

Lühikokkuvõte:

Lineaarseid partiikoode saab kasutada koormuse ühtlustamiseks hajusandmetalletussüsteemides. Selleks et tagada efektiivne sooritusvõime on tarvis optimeeritud parameetritega koode. Selliste koodide leidmine on aga keerukas matemaatiline probleem.

Selles töös esitatakse algoritme ja tarkvara, mille abil on võimalik uurida lineaarseid partiikoode. Tuletatakse kaks uut ülemtõket lineaarsetele partiikoodidele. Lõpuks võrreldakse tarkvara abil leitud lühimaid süstemaatiliste lineaarsete partiikoodide pikkuseid seni teadaolevate tõketega.

Võtmesõnad: Kodeerimisteooria, lineaarsed partiikoodid, tõkked koodi parameetritele, hajus andmetalletus.

CERCS: P175 Informaatika, süsteemiteooria

Contents

1 Introduction

1.1 General background

In a large data storage system, where the data is distributed between multiple servers, there are multiple concurrent requests. To ensure efficient service, load balancing is required. For this purpose, batch codes were proposed by Ishai *et al.* in [1].

Batch codes help to design a system, that ensures that any t requests can be made at the same time, and the system will know how to handle the requests in parallel, without overloading some of the servers.

A special case of batch codes are linear batch codes. In that case, the coding is a linear mapping function. A common way of defining linear batch codes is with the help of a generator matrix. The generator matrix is a $k \times n$ matrix, where k is the number of information symbols and n is the number of coded symbols that are stored.

Linear batch codes were studied in [2]. Additional bounds were obtained in [3] and [4]. In [5], the authors study a special case of batch codes called "binary switch codes." A related family of codes called locally repairable codes is considered in [6].

Another related code family is codes for private information retrieval (PIR). These codes can be used in the distributed databases, where the user wants to keep the requested symbols secret from the server [7].

The following questions arise in this context:

1. What is the smallest amount of servers for handling t requests at the same time?
2. What is the best method for forming reconstruction sets?
3. How to generate good batch codes?

The aim of the thesis is to study constructions of batch codes. In Chapter 1, the definitions and notation are presented. Chapter 2 presents algorithms, which are used for checking whether a batch code with given parameters exists. Chapter 3 discusses two upper bounds. Chapter 4 covers results of experimental search for efficient linear batch codes.

The software described in Chapter 2 can be used for searching for generator matrices of batch codes, but it is not limited only to batch codes. It can also be used for PIR codes. By using this software, in Chapter 4 minimum lengths of possible batch codes are presented.

1.2 Definitions

We use $\mathbb{N}$ for a set of natural numbers. In this work, $\mathbf{g}_i$ denotes the i-th column of a matrix G. Moreover, $[m]$ denotes a set $\{1, 2, ..., m\}$. A vector $\mathbf{e}_i$ denotes a column vector, that contains a single '1' in the i-th row and contains zeros elsewhere. This section begins with the definition of code.

Definition 1. [8] Let Σ be a finite alphabet. Let $\mathbf{x} = (x_1, x_2, ..., x_k) \in \Sigma^k$ be an information vector. A code is a set of coded vectors (codewords) $\{\mathbf{y} = (y_1, y_2, ..., y_n) = C(\mathbf{x}) : \mathbf{x} \in \Sigma^k\} \subseteq \Sigma^n$, where $C : \Sigma^k \to \Sigma^n$ is a bijection for some $n \in \mathbb{N}$.

The specific codes proposed for load balancing in information retrieval are called batch codes and are defined as follows based on [8]:

Definition 2. An $(k, n, t, r)_\Sigma$ batch code C over a finite alphabet Σ is defined by an encoding mapping $C : \Sigma^k \to \Sigma^n$ and a decoding mapping $D : \Sigma^n \times [k]^t \to \Sigma^t$, such that:

1. for any $\mathbf{x} \in \Sigma^k$ and $i_1, i_2, ...i_t \in [k]$, $D(C(\mathbf{x}), i_1, i_2, ...i_t) = (x_{i_1}, x_{i_2}, ..., x_{i_t})$.
2. each x_{i_j}, $1 \leq j \leq t$, can be reconstructed from a set of at most r symbols of $C(\mathbf{x})$, where these sets are disjoint for x_{i_j}, x_{i_l}, $j \neq l$.

In this thesis, we focus on linear batch codes, which are defined as follows:

Definition 3. [2] We say that an $(k, n, t, r)_q$ batch code is linear over a finite field $\mathbb{F}_q$, if every symbol in the codeword is a linear combination over $\mathbb{F}_q$ of the original symbols from the information vector.

Definition 4. A generator matrix is a $k \times n$ matrix, where k is equal to the number of information symbols and n is the number of coded symbols, such that $C(\mathbf{x}) = \mathbf{x}G$.

In this paper, the binary field $\mathbb{F}_2$ is used instead of Σ.

Definition 5. [9, p. 6] Hamming distance between two vectors is the number of coordinates on which these two vectors differ. The minimum distance of a code is the minimum Hamming distance between any two different codewords in the code.

Definition 6. [9, p. 6] Let $\mathbb{F}^n$ be a set of vectors of length n over a finite field $\mathbb{F}$. The Hamming weight of $\mathbf{e} \in \mathbb{F}^n$ is the number of nonzero entries in $\mathbf{e}$.

Definition 7. A linear batch code is called systematic if the generator matrix contains a $k \times k$ identity matrix as a sub-matrix.

The request can be viewed as a vector $(x_{i_1}, x_{i_2}, ..., x_{i_t}) \in (\mathbb{F}_2)^t$. Here t denotes the size of the request.

Definition 8. Recovery is the process of reconstructing original information symbols from a codeword.

The following theorem, describes the requirements for retrieving t symbols from a codeword of a linear batch code.

Theorem 1. *[2] Let C be an $(n, k, t, r)_q$ batch code. It is possible to retrieve $x_{i_1}, x_{i_2}, ..., x_{i_t}$ simultaneously if and only if there exist t non-intersecting sets $T_1, T_2, ..., T_t$ of size at most r each containing indices of columns in G, and for T_l there exists a linear combination of columns of G indexed by that set, which equals to the column vector $e_{i_l}^T$, for all $l \in [t]$.*

Private information retrieval codes [7] differ from batch codes as following: a PIR code with the parameters (k, n, t, r) allows the retrieval of t identical symbols. In other words, all symbols in a request are the same original symbol. A batch code allows for the retrieval of any sequence of original symbols. Therefore it can be seen, that any linear batch code is, in particular, a PIR code.

To describe the algorithms in Chapter 2, the following definition is required.

Definition 9. Multicombination is a set where each element can occur multiple number of times.

1.3 Examples

Example 1. Consider a code C defined by a generator matrix G.

$$G = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \end{pmatrix} \tag{1}$$

Here, the codeword $\mathbf{y}$ can be obtained by multiplying the information vector $\mathbf{x}$ by the generator matrix. Let the information vector be $\mathbf{x} = (1, 0, 1, 1) \in (\mathbb{F}_2)^t$. We have that $\mathbf{y} = (1, 0, 1, 1, 1, 0, 0, 1) = \mathbf{x}G$. The symbols in the codeword can be written also as $y_1 = x_1$, $y_2 = x_2$, $y_3 = x_3$, $y_4 = x_4$, $y_5 = x_1 + x_2$, $y_6 = x_3 + x_4$, $y_7 = x_1 + x_3$, $y_8 = x_2 + x_4$.

This code can support any request of size $t = 3$ with $r = 2$. For example, given the request (x_2, x_2, x_2) the information symbols can be retrieved using elements (y_2), (y_1, y_5), (y_7, y_4), namely, $x_2 = y_2$, $x_2 = y_1 + y_5$, $x_2 = y_4 + y_7$. Since similar recovery equations can be written down for any request $(x_{i_1}, x_{i_2}, x_{i_3})$, the corresponding code is a $(4, 8, 3, 2)_2$ systematic linear batch code.

Example 2. Consider a code C, which is defined by a generator matrix G.

$$G = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 \end{pmatrix} \tag{2}$$

Here, the codeword $\mathbf{y}$ can be found by multiplying the information vector $\mathbf{x}$ by the generator matrix. Let the information vector be $\mathbf{x} = (1, 0, 1, 1, 0) \in (\mathbb{F}_2)^t$. We have that $\mathbf{y} = (1, 0, 1, 1, 0, 0, 0, 0, 0, 1) = \mathbf{x}G$. The symbols in the codeword can be written also as $y_1 = x_1$, $y_2 = x_2$, $y_3 = x_3$, $y_4 = x_4$, $y_5 = x_5$, $y_6 = x_1 + x_2 + x_3$, $y_7 = x_3 + x_4 + x_5$, $y_8 = x_1 + x_2 + x_4$, $y_9 = x_1 + x_3 + x_5$ and $y_{10} = x_2 + x_4 + x_5$.

Some recovery sets for each symbol are:

- for x_1, $T_1 = y_1$, $T_2 = y_6 + y_2 + y_3$, $T_3 = y_8 + y_2 + y_4$ and $T_4 = y_9 + y_3 + y_5$.
- for x_2, $T_5 = y_2$, $T_6 = y_6 + y_1 + y_3$, $T_7 = y_8 + y_1 + y_4$ and $T_8 = y_{10} + y_4 + y_5$.
- for x_3, $T_9 = y_3$, $T_{10} = y_6 + y_1 + y_2$, $T_{11} = y_7 + y_4 + y_5$ and $T_{12} = y_9 + y_1 + y_5$.
- for x_4, $T_{13} = y_4$, $T_{14} = y_7 + y_3 + y_5$, $T_{15} = y_8 + y_1 + y_2$ and $T_{16} = y_{10} + y_2 + y_5$.
- for x_5, $T_{17} = y_5$, $T_{18} = y_7 + y_3 + y_4$, $T_{19} = y_9 + y_1 + y_3$, $T_{20} = y_{10} + y_2 + y_4$, $T_{21} = y_8 + y_{10} + y_1$, $T_{22} = y_9 + y_6 + y_2$.

This code can support any request of size $t = 4$ with $r = 3$. For example, if the request is (x_3, x_5, x_5, x_5), the information symbols can be retrieved using recovery sets T_9, T_{17}, T_{21}, T_{22}. The used columns are $y_3; y_5; y_8, y_{10}, y_1; y_2, y_6, y_9$.

This is a $(5, 10, 4, 3)_2$ systematic linear batch code. It is shown in Section 4 that if $r = 2$, then there is no linear batch code with parameters $(5, 10, 4, 2)_2$.

Example 3. During execution of the algorithm in Chapter 2, some interesting examples of linear batch codes were found. For example, the matrix G_1 is a generator matrix of a $(6, 10, 3, 4)_2$ systematic linear batch code.

$$G_1 = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 \end{pmatrix} \tag{3}$$

The matrix G_2 is the generator matrix of a $(4, 11, 5, 3)_2$ systematic linear batch code. For parameters $k = 4$, $t = 5$ and $r = 2$, the shortest known length of a linear batch code is $n = 12$, as shown in Chapter 4, Table 1.

$$G_2 = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 1 \end{pmatrix} \tag{4}$$

The matrix G_3 is the generator matrix of a $(5, 13, 5, 3)_2$ systematic linear batch code.

$$G_3 = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 \end{pmatrix} \tag{5}$$

1.4 Known results

There have been several upper and lower bounds on the code parameters presented in the literature. Specifically, we fix k, r and t and estimate the smallest value of n such that there exists an (k, n, t, r) linear batch code.

In [6], Theorem 2, the following lower bound on the length n of the code is obtained:

$$n \geq d_{min}(C) + k + \left\lceil \frac{(t-1)(k-1)+1}{(t-1)(r-1)+1} \right\rceil - 2 \tag{6}$$

In [3], the following upper bounds were shown:

$$\text{if } t = 2 \text{ and } r \geq 2,\ n \leq \left\lceil \frac{k}{r} \right\rceil + k \tag{7}$$

$$\text{if } t \geq 2 \text{ and } r = 2,\ n \leq k + \left\lceil (t-1)\frac{k}{r} \right\rceil \tag{8}$$

In Table 2 of the same paper, the following special cases of equations (7) and (8) were shown.

$$\text{if } t = 2 \text{ and } r = 2,\ n \leq k + \left\lceil \frac{k}{2} \right\rceil \tag{9}$$

$$\text{if } t = 3,\ r = 2 \text{ and } k > 2,\ n \leq 2k \tag{10}$$

$$\text{if } t = 4,\ r = 2 \text{ and } k > 2,\ n \leq k + \left\lceil \frac{3k}{2} \right\rceil \tag{11}$$

$$\text{if } t = 2 \text{ and } r \geq 3,\ n \leq k + \left\lceil \frac{k}{r} \right\rceil \tag{12}$$

In [4], the following upper bound was derived.

$$\zeta = \max\left\{\tfrac{k}{r}, r\right\} \qquad s = k \mod r$$
$$\tau = \min\left\{r - s, \left\lfloor \tfrac{k}{r} \right\rfloor\right\} \qquad \eta = \min\left\{r - 1, \left\lfloor \tfrac{k}{r} \right\rfloor\right\}$$
$$\gamma = \min\left\{r, \left\lfloor \tfrac{k}{r} \right\rfloor\right\}$$

$$\text{if } t = 3 \text{ and } r \geq 3,\ n \leq \begin{cases} (r+1)\frac{k}{r} + \zeta & \text{if } r|k \\ (r+1)\left\lfloor \frac{k}{r} \right\rfloor + 2s + 1 + \left\lceil \frac{(k-s)-\tau-\eta s}{\gamma} \right\rceil & \text{if } \neg r|k \end{cases} \tag{13}$$

In Chapter 4, the upper and lower bounds are compared to the values calculated by the algorithms described in Chapter 2.

2 Software

2.1 Introduction

In this chapter, the software for efficient searching for optimal batch codes is described. The chapter also contains information about the algorithms proposed to solve this task.

The program was implemented using C++ 11 with memory efficiency in mind. The MinGW g++ compiler was used to ensure cross-platform usability. A copy of the code is accessible on GitHub [10].

2.2 Algorithms

2.2.1 Request fill algorithm

The algorithm receives a request of size t and returns a vector of the same size containing values of the requested symbols. The recovery process has access to code parameters, the codeword and the generator matrix used in the generation of this code. In the implementation, the accessible variables are class variables and the algorithm is a class function.

According to Theorem 1, in order to recover t symbols of information, t non-intersecting sets of column indices are required. The construction of recovery sets and recovery of symbols can be solved recursively or with a cycle. Algorithm 1 generates the required sets using recursion and is presented in this section. An alternative algorithm for the same task, which uses *while* cycle, is presented in Appendix.

The process of filling in the request uses recursion with a step representing consecutive processing of the requested symbol indices. In the implementation of the algorithm, the whole algorithm works on the same arrays with memory efficiency in mind.

1. At any given step of recursion, subsets of indices of increasing size are generated. The subsets contain the generator matrix column indices. The size of the subsets grows from 1 to r.

Algorithm 1: Answering a request

Input: $\text{type}_{\text{numeric}}$ *step*, $\text{type}_{\text{array}}$ *usedColumns*, $\text{type}_{\text{array}}$ *answer*
Result: $\text{type}_{\text{array}}$ *answer*, $\text{type}_{\text{boolean}}$ *completed*

```
1  foreach column in generatorMatrix do
2      if not (column used) and (column weight == 1) and (column answers
         requested symbol) then
3          setColumnAsUsed;
4          addToAnswer;
5          completed = Answering a request (increase step, usedColumns,
             answer);
6          if completed then
7              return (answer, completed);
8          else
9              remove column from used;
10             revert changes to answer;
11     else
12         continue;
13 foreach combinationsize from 2 to r do
14     combination[combinationsize]
15     while hasNextCombination(combination) do
16         if ColumnsInCombinationUnused and
             combinationAnswersRequestedSymbol then
17             setcolumnIdsInCombinationAsUsed;
18             addToAnswer;
19             completed = Answering a request (increase step, usedColumns,
                 answer);
20             if completed then
21                 return (answer, true);
22             else
23                 remove columnIdsInCombination from used;
24                 revert changes to answer;
25         else
26             continue;
27 return (answer, false);
```

2. Every subset is tested to determine, whether it is a recovery set for the symbol in question. The number of times a column in this subset is previously used,

is also checked at this step. The algorithm allows for a column to be used a predefined number of times. For example, in linear batch codes under consideration every symbol in codeword is allowed to be read only once.

3. If Step 2 is successful, the recursion continues until the last symbol is retrieved.

   (a) If the final symbol is recovered, the vector containing the values of requested symbols is returned.

   (b) If the last tested subset at the current level of recursion fails to recover the symbol in question, recursion returns a fail message, and the cycle in the previous level continues.

The following functions are used. The overview of these functions is given as follows.

- *setColumnAsUsed* - the column index in question is marked as a used column to prevent reading a symbol in the codeword more times than it is allowed.
- *addToAnswer* - a given column or columns are used to calculate the value of the requested symbol, which is then added to the *answer* variable.
- *Answering a request* - a recursive call to the same function.
- *revert changes to answer* - the values written to the *answer* variable at this step of recursion are reverted.
- *hasNextCombination* - a function, which takes a variable containing some combination, and tries to write the next combination in the given variable. Returns *TRUE* or *FALSE* value depending on whether the writing is successful.
- *ColumnsInCombinationUnused* - a function which checks that column indices in a combination are unused.
- *combinationAnswersRequestedSybmols* - a function, which checks if a set of column indices can be used to recover the requested symbol.
- *setColumnIdsInCombinationAsUsed* - similar to *setColumnsAsUsed*. Has multiple column indices to be marked.
- *remove columIdsInCombination from used* - similar to *remove column from used*. Reverts the changes made to the variable containing information about the used symbols.

2.2.2 Code testing algorithm

The code testing algorithm, which appears as Algorithm 2, uses a generator matrix and a code generated using that matrix. The function checks every possible request of size t for that code and decides if the generator matrix can be used to generate a code or not.

All possible requests contain all possible multicombinations of indices of information symbols and their permutations. If a request can be satisfied, then another request containing the same symbols in another order is also satisfied. Therefore the permutations of the requested symbols can be ignored.

Algorithm 2: Testing if a matrix is batch code

Input: $\mathsf{type}_{\mathsf{arrayofnumeric}}$ *request*
Result: $\mathsf{type}_{\mathsf{boolean}}$ isBatchCode

```
1 generatesACodeFromGivenGeneratorMatrix;
2 type_array combination[request_size t]
3 while hasNextMultiCombination(combination) do
4 |   answer = answerRequest(combination)
5 |   if failed to answer or wrong answer then
6 |   |   return false;
7 |   else
8 |   |   continue;
9 return true;
```

1. All possible requests of size t are generated up to their permutations.
2. For every request, *answer the request* function is run.
3. The output of the function "*Answer the request*" is checked. If the request was satisfied, the cycle continues. If it fails, the cycle breaks.
   (a) If the final request has been checked and found satisfiable, it has been proven that the generator matrix belongs to a code with given parameters.
   (b) If at any iteration of the cycle a request fails, it proves that the generator matrix does not belong to a code with given parameters.

The following functions are used. The overview of these functions is given as follows.

- *generatesACodeFromGivenGeneratorMatrix* - a function which finds a codeword for some random information vector.
- *hasNextMultiCombination* - given a variable containing some combination, makes changes to the variable to find the next multicombination (see Definition 9).
- *answerRequest* - calls a function, which answers the request, for example *Answer a Request* (see Algorithm 1).

2.2.3 Checking if a code with given parameters exists

To prove that a code with given parameters exists, at least one example of such code is required. To prove the opposite, all possible codes must be checked, and every check must fail. The checking process can be reduced to the space of all possible generator matrices which can not be constructed by permuting the columns of another matrix in the set.

First, all possible columns are fixed. Then the following algorithm, called Algorithm 3, is carried out.

1. All possible multicombinations of size n, where n is the length of the codeword, are generated from the indices of all possible columns. Every such vector containing indices of columns represents a possible generator matrix.
2. For every possible generator matrix, the following checks are carried out:
   (a) The Hamming weight of each row of the matrix must be at least t, the size of the request.
   (b) The indices of columns are checked against previously generated and tested matrices to make sure that this is not a permutation of a previously checked matrix.
3. If the previous step succeeded, all possible requests shall be tested against a code constructed using this possible generator matrix.
   (a) If the testing step succeeded, the generator matrix is saved and it is proven that there exists a code for given parameters.
   (b) If the testing step failed, another generator matrix will have to be tested. The algorithm continues.

Algorithm 3: Generating a generator matrix for batch code

Input: k, n, t, r, *systematic*

Result: Generator matrices for batch codes or special message that no such code exists for given parameters

```
1  fixAllPossibleColumns;
2  type_array combination[code Size n];
3  if systematic then
4  |  set first k columns
5  else
6  |  continue;
7  while hasNextMultiCombination(combination) do
8  |  if contains at least t ones in every row then
9  |  |  answer = testIfIsBatchCode(combination);
10 |  |  if is batch code then
11 |  |  |  save generator matrix;
12 |  |  else
13 |  |  |  continue;
14 |  else
15 |  |  continue;
```

4. If the final subset has been tested and failed, then it is proved that no code exists for given parameters.

The following functions are used. The overview of these functions is given as follows.

- *fixAllPossibleColumns* - enumerates all possible columns.
- *set first k columns* - sets the first k columns of the generator matrix to $k \times k$ identity matrix.
- *hasNextMultiCombination* - given a variable containing some combination, makes changes to the variable to find the next multicombination (see Definition 9).
- *contains at lest t ones in every row* - checks that the generator matrix contains t ones at every row.
- *testIfIsBatchCode* - runs all tests on the found generator matrix and code parameters, as it is shown in Algorithm 2.

- *save generator matrix* - saves the found matrix.

In this implementation of the algorithm, the program is not terminated upon finding one generator matrix, but it continues until it finds all possible generator matrices up to permutations of columns.

2.3 Instructions for use

The implementation of these algorithms has been done using object-oriented language C++, thus the classes have been used. Answering a request is implemented in the *Codesys* class, running all tests is implemeted in the *Tester* class and finally, checking if a code with given parameters exists is implemented in the *CodeFinder* class.

An implementation of the algorithm is provided in a file called *program.cpp*. It receives parameters in the given order

1. *Program method*:
    - 0 - for finding batch codes.
    - 1 - for finding batch codes with fixed columns in the generator matrix.
    - 2 - for finding PIR codes
2. k - the number of symbols in the information vector.
3. n - the length of the codeword.
4. t - the size of the request.
5. r - the maximal size of the recovery set.
6. *is systematic* - 1 or 0.
7. s - the number of times a symbol in the codeword can be read.

2.4 Possible improvements

It is possible to run some parts of algorithms in parallel using multi threading in order to reduce the total running time of the algorithm. This could cause issues with multiple threads accessing the same generator matrix at the same time.

It would also be possible to implement the algorithms in the GPU (graphical processing unit).

Currently, the program is built with memory efficiency in mind. If the user were to have sufficient memory, the algorithm could be optimized using techniques from dynamic programming. Then, running multiple requests could be optimized to generate all recovery sets for every information symbol first, and to combine them together later.

3 Constructions

A variety of bounds on the code parameters appear in the literature. In this chapter, we assume that r and t are fixed, and k is an arbitrary parameter. Two upper bounds are constructed for linear batch codes with information vector of size 2 and 3 for requests of any size. These upper bounds are obtained by using a concatenation of generator matrices.

The construction uses some ideas from [5].

3.1 Upper bound for systematic linear batch code with $k = 2$, $r = 2$ and any t

From [1], the shortest length linear batch code for $k = 2$, $r = 2$, $t = 2$ is $n = 3$. The generator matrix has the following form:

$$G = \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \end{pmatrix}$$

To fill a request of any size t, there are two possibilities: $t \mod 2 = 0$ and $t \mod 2 = 1$.

First, if $t \mod 2 = 0$, then a generator matrix M for a linear batch code $(2, n, t, 2)_2$ can be constructed by appending $\frac{t}{2}$ copies of the matrix G:

$$M = \begin{pmatrix} G & | & G & | & ... & | & G \end{pmatrix}$$

Then the request can be divided into pairs of symbols, and any two symbols can be retrieved using one sub-matrix G in M.

If $t \mod 2 = 1$, then the generator matrix M can be constructed as follows:

$$M = \begin{pmatrix} G & |...| & G & | & \mathbf{g}_1 & | & \mathbf{g}_2 \end{pmatrix}$$

where there are $\left\lfloor \frac{t}{2} \right\rfloor$ instances of G and two additional columns, $\mathbf{g}_1$ and $\mathbf{g}_2$. When dividing the request into pairs of symbols, the $t-1$ symbols can be retrieved from the sub-matrices G in M. To recover the last symbol, two columns $\mathbf{g}_1$ and $\mathbf{g}_2$ must be used.

If t is divisible by 2, then for every group of three columns, any 2 requested symbols can be recovered. Therefore, the maximum value of n is given by equation (14).

$$n = \frac{3}{2}t \tag{14}$$

If the request size t is not divisible by two, then the number of columns to retrieve the first $t-1$ symbols, can be found using equation (14). To retrieve the last symbol, two additional columns are necessary. Therefore, the required number of columns is given in equation (15).

$$n = \frac{3(t-1)}{2} + 2 \tag{15}$$

Equation (15) can be simplified as shown in equation (16).

$$n = \frac{3(t-1)}{2} + 2 = \frac{3t-3+4}{2} = \frac{3t+1}{2} \tag{16}$$

Therefore, we obtained the following upper bound shown in equation (17).

$$n = \left\lceil \frac{3}{2}t \right\rceil = \begin{cases} \frac{3}{2}t & \text{if } t \text{ is even} \\ \frac{3t+1}{2} & \text{if } t \text{ is odd} \end{cases} \tag{17}$$

3.2 Upper bound for systematic linear batch code with $k = 3$, $r = 2$ and any t

In [5], a special case of linear batch codes was constructed, when

$$n = 2^k - 1 \tag{18}$$

$$t = 2^{k-1} \tag{19}$$

$r = 2$, for any value of k. These codes can be used as building blocks in a more general construction, where the value of t varies.

For $k = 3$, the corresponding parameters are $n = 7$ and $t = 4$ (from equations (18) and (19)). That means, with 7 columns, any request of size 4 can be satisfied.

Let us take the binary 3×7 matrix G as follows:

$$G = \begin{pmatrix} 1 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 \end{pmatrix}$$

It was shown in [5] that this matrix corresponds to a batch code with $n = 7$, $k = 3$, $t = 4$, $r = 2$. Next we use G to construct more general batch codes for $k = 3$ and any t.

If $t \mod 4 = 0$, we use groups of 7 different columns to satisfy any request of size 4. Then, a generator matrix M for a linear batch code $(3, n, t, 2)_2$ can be constructed by appending $\frac{t}{4}$ copies of the matrix G:

$$M = \begin{pmatrix} G & | & G & | & ... & | & G \end{pmatrix}$$

If $t \mod 4 = 1$, then the generator matrix M_1 can be constructed as follows:

$$M_1 = \begin{pmatrix} M & | & \mathbf{g}_1 & | & \mathbf{g}_2 & | & \mathbf{g}_3 \end{pmatrix}$$

where there are $\left\lfloor \frac{t}{4} \right\rfloor$ instances of G denoted as M and three additional columns, $\mathbf{g}_1$, $\mathbf{g}_2$ and $\mathbf{g}_3$. When dividing the request into groups of symbols by 4, the $t-1$ symbols can be retrieved from the sub-matrices G in M, the last symbol from $\mathbf{g}_1$, $\mathbf{g}_2$ or $\mathbf{g}_3$.

If $t \mod 4 = 2$, then the generator matrix M_2 can be constructed as follows:

$$M_2 = \begin{pmatrix} M_1 & | & \mathbf{g}_4 & | & \mathbf{g}_5 \end{pmatrix}$$

As it is shown above, for $t \mod 4 = 1$, t symbols can be retrieved from the generator matrix of form M_1. If $t \mod 4 = 2$, then $t-1$ symbols can be retrieved using M_1. For the last symbol, columns $\mathbf{g}_4$ and $\mathbf{g}_5$ are needed. If $x_{i_{t-1}} = x_{i_t}$, then x_{i_t} can be recovered using one of the other e_i columns and one of the added columns $\mathbf{g}_4$ or $\mathbf{g}_5$ based on which column returns a column equal to e_{i_t}.

If $t \mod 4 = 3$, $t-1$ symbols can be requested using a generator matrix described in the previous step. The generator matrix M_3 can be constructed by constructing a generator matrix M_2 and adding column $\mathbf{g}_6$.

$$M_3 = \begin{pmatrix} M_2 & | & \mathbf{g}_6 \end{pmatrix}$$

From these 4 types of generator matrices, 4 upper bounds can be derived. For a request of size t, n can be found with equation (20).

$$n = \begin{cases} \frac{7}{4}t & \text{if } t \mod 4 = 0 \\ \frac{7(t-1)}{4} + 3 & \text{if } t \mod 4 = 1 \\ \frac{7(t-2)}{4} + 5 & \text{if } t \mod 4 = 2 \\ \frac{7(t-3)}{4} + 6 & \text{if } t \mod 4 = 3 \end{cases} \tag{20}$$

These 4 cases can be used to calculate the minimum value of n for any t.

4 Experimental results

With the help of the program described in Chapter 2, the following tables were filled in. All the tested codes are over $\mathbb{F}_2$.

4.1 Tables for batch codes

	$k=2$	$k=3$	$k=4$	$k=5$	$k=6$
$t=2$	(3, 3, 3)	(5, 5, 5)	(6, 6, 6)	(8, 8, 8)	(9, 9, 9)
$t=3$	(4, 5, 5)	(6, 6, 6)	(8, 8 8)	(9, 10, 10)	(11, >10, 12)
$t=4$	(5, 6, 6)	(7, 7, 8)	(9, 10, 10)	(11, >11, 13)	(12, ... , 15)
$t=5$	(6, 8, ...)	(8, 10, ...)	(10, 12, ...)	(12, >12, ...)	(14, ... , ...)
$t=6$	(7, 9, ...)	(9, 12, ...)	(11, >12, ...)	(13, ... , ...)	(15, ... , ...)
$t=7$	(8, 11, ...)	(10, 13, ...)	(12, $\leq$ 14, ...)	(14, ... , ...)	(16, ... , ...)
$t=8$	(9, 12, ...)	(11, 14, ...)	(13, $\leq$ 15, ...)	(15, ... , ...)	(17, ... , ...)
$t=9$	(10, 14, ...)	(12, >15, ...)	(14, ... , ...)	(16, ... , ...)	(18 ... , ...)
$t=10$	(11, 15, ...)	(13, ... , ...)	(15, ... , ...)	(17, ... , ...)	(19, ... , ...)
$t=11$	(12, 17, ...)	(14, ... , ...)	(16, ... , ...)	(18, ... , ...)	(20, ... , ...)
$t=12$	(13, 18, ...)	(15, ... , ...)	(17, ... , ...)	(19, ... , ...)	(21, ... , ...)

Table 1: Lower bound, calculated optimal length of systematic linear batch code and upper bound for various value of k and t where $r = 2$

Tables 1 and 3 - 5 present triples of numbers (a, b, c). These numbers represent shortest lengths n of a systematic linear batch codes with given parameters k, t and r. Here, a is a lower bound, b is an actual value and c is an upper bound. The notation ... means, that the value is unknown. The notation $>$ means that the value is larger than the one that appears in the table.

The lower bound was calculated using equation (6). The actual value was computed using a program described in Chapter 2.

In Table 1, r is 2. In the $t = 2$ row, the real value, lower and upper bounds are equal. Upper bound was found with equation (9). In the $t = 3$ row, the upper bound was found with equation (10) except for column $k = 2$, where the upper bound was found with equation (17). In the $t = 4$ row, the upper bound was found with equation (11) except for column $k = 2$, where the upper bound was found with equation (17).

	$k=2$	$k=3$
$t=2$	(3, 3)	(5, 5)
$t=3$	(5, 5)	(6, 6)
$t=4$	(6, 6)	(7, 7)
$t=5$	(8, 8)	(10, 10)
$t=6$	(9, 9)	(12, 12)
$t=7$	(11, 11)	(13, 13)
$t=8$	(12, 12)	(14, 14)
$t=9$	(14, 14)	(>15, 17)
$t=10$	(15, 15)	(... , 19)
$t=11$	(17, 17)	(... , 20)
$t=12$	(18, 18)	(... , 21)

Table 2: Calculated optimal length of systematic linear batch code and upper bound for t with equation (17) for $k=2$ and equation (20) for $k=3$ when $r=2$

	$k=3$	$k=4$	$k=5$	$k=6$
$t=2$	(4, 4, 4)	(6, 6, 6)	(7, 7, 7)	(8, 8, 8)
$t=3$	(5, 6, 7)	(7, 8, 8)	(8, 9, 9)	(10, 10, 11)
$t=4$	(6, 7, ...)	(8, 9, ...)	(9, 10, ...)	(11, >11, ...)
$t=5$	(7, 10, ...)	(9, 11, ...)	(10, 13, ...)	(12, ... , ...)
$t=6$	(8, 11, ...)	(10, 12, ...)	(11, >13, ...)	(13, ... , ...)
$t=7$	(9, 13, ...)	(11, 14, ...)	(12, ... , ...)	(14, ... , ...)
$t=8$	(10, 14, ...)	(12, 15, ...)	(13, ... , ...)	(15, ... , ...)

Table 3: Lower bound, calculated optimal length of systematic linear batch code, upper bound for various value of k and t where $r=3$

Table 2 displays the values calculated with the program compared to the upper bounds proposed in this paper.

In Table 3, r is 3. In the row $t=2$, the upper and lower bounds are equal to the real value found by the program. The upper bound was found with equation (12). In the row $t=3$, the upper bound found with equation (13).

In Table 4, r is 4. In row $t=2$, the value calculated with the program matches both the upper and lower bounds. The upper bound was calculated with equation (12). In the row $t=3$ the upper bound was calculated with equation (13).

	$k = 4$	$k = 5$	$k = 6$
$t = 2$	(5, 5, 5)	(7, 7, 7)	(8, 8, 8)
$t = 3$	(6, 8, 9)	(8, 9, 10)	(9, 10, 11)
$t = 4$	(7, 9, ...)	(9, 10, ...)	(10, ... , ...)
$t = 5$	(8, 11, ...)	(10, 13, ...)	(11, ... , ...)
$t = 6$	(9, 12, ...)	(11, ... , ...)	(12, ... , ...)
$t = 7$	(10, 14, ...)	(12, ... , ...)	(13, ... , ...)
$t = 8$	(11, 15, ...)	(13, ... , ...)	(14, ... , ...)

Table 4: Lower bound, calculated optimal length of systematic linear batch code, upper bound for various value of k and t where $r = 4$

	$k = 2$	$k = 3$	$k = 4$	$k = 5$	$k = 6$
$t = 2$	(3)	(5, 4)	(6, 6, 5)	(8, 7, 7)	(9, 8, 8)
$t = 3$	(5)	(6, 6)	(8, 8, 8)	(10, 9, 9)	(>10, 10, 10)
$t = 4$	(6)	(7, 7)	(10, 9, 9)	(>11, 10, 10)	(... , >11)
$t = 5$	(8)	(10, 10)	(12, 11, 11)	(>12, 13, 13)	(...)
$t = 6$	(9)	(12, 11)	(>12, 12, 12)	(... , >13)	(...)
$t = 7$	(11)	(13, 13)	($\leq$ 14, 14, 14)	(...)	(...)
$t = 8$	(12)	(14, 14)	($\leq$ 15, 15, 15)	(...)	(...)

Table 5: Comparison of calculated optimal length of systematic linear batch codes for various values of k and t. The first value in each entry is the minimum length n for $r = 2$, and the subsequent values represent minimum length n for $r = 3$, $r = 4$, etc. The maximum value of r under consideration is $r = k$.

Table 5 contains entries from previous tables. In any cell, the first value is the optimal length n of a linear batch code with given parameters and $r = 2$. The subsequent values represent minimum length n for $r = 3$, $r = 4$, etc. It should be noted that only cases where $r \leq k$ are of interest. If $r > k$ then not all r symbols are necessary for recovery. We observe that increasing values of r yield shorter codes for fixed values of k and t.

4.2 Comparison of PIR and batch codes

The question arises, whether for the same parameters PIR codes are shorter that batch codes. The experiment tested PIR codes, where the request size was taken as in the shortest known batch codes, reduced by one. The following values were tested:

- $r = 2$, $k = 3$ and $k = 4$, $t \in [5, 8]$
- $r = 3$, $k = 3$ and $k = 4$, $t \in [5, 8]$; $k = 5$ and $t = 5$
- $r = 4$, $k = 4$, $t \in [5, 6]$

From the experimentations, no PIR codes with shorter length than linear systematic batch codes with the same parameters were found.

5 Conclusions

In this thesis, algorithms and software for finding generator matrices of linear batch codes, running tests and answering requests are presented. The algorithms were modified and improved during the process of this work.

To optimize the parameters of linear batch codes, equations (17) and (20) for upper bounds on systematic linear batch codes are proposed. The first one follows from [1]. The second bound builds upon the equations in [5]. As it was mentioned, similar tight upper bounds for linear batch codes can be obtained by calculating optimal values of n for the first requests up to size t.

Finally, the optimal lengths of linear batch codes found with the software are compared to lower and upper bounds from [6], [3] and [4].

There are some questions that are left open.

1. In this thesis, the optimal values of n were calculated for relatively small codes. However, the search space is large, therefore it would be interesting to compute values of n for larger parameters. This could be possible with improvements to the algorithm or by using parallelization and high performance computers.

2. Currently, the bounds are not tight. It would be interesting to further tighten the bounds. Also, it would be interesting to compare bounds for different variations of batch codes. For example, one could consider generalizations of a batch code which allows sampling every coded symbol a constant number of times, and derive bonds on its parameters.

References

[1] Ishai Y, Kushilevitz E, Ostrovsky R, Sahai A. Batch codes and their applications. In: Proceedings of the thirty-sixth annual ACM symposium on Theory of computing. ACM; 2004. p. 262–271.

[2] Lipmaa H, Skachek V. Linear batch codes. In: Coding Theory and Applications. Springer; 2015. p. 245–253.

[3] Thomas EK, Skachek V. Explicit Constructions and Bounds for Batch Codes with Restricted Size of Reconstruction Sets; 2017. `http://arxiv.org/abs/1701.07579v1/`.

[4] Zhang H, Skachek V. Bounds for batch codes with restricted query size. In: Information Theory (ISIT), 2016 IEEE International Symposium on. IEEE; 2016. p. 1192–1196.

[5] Wang Z, Kiah HM, Cassuto Y. Optimal binary switch codes with small query size. In: Information Theory (ISIT), 2015 IEEE International Symposium on. IEEE; 2015. p. 636–640.

[6] Rawat AS, Papailiopoulos DS, Dimakis AG, Vishwanath S. Locality and availability in distributed storage. IEEE Transactions on Information Theory. 2016;62(8):4481–4493.

[7] Fazeli A, Vardy A, Yaakobi E. PIR with low storage overhead: coding instead of replication; 2015. arXiv preprint arXiv:1505.06241.

[8] Skachek V. Batch and PIR Codes and Their Connections to Locally-Repairable Codes; 2016. arXiv preprint arXiv:1611.09914.

[9] Roth R. Introduction to coding theory. Cambridge University Press; 2006.

[10] Simisker M. Linear Batch Code Finder; 2017. `https://github.com/Martsim/Linear-Batch-Code-Finder`.

Appendix A: Alternative algorithm for filling a request

To generate reconstruction sets with *while* loop, variables for the list of recovery sets, a list of recovery set sizes, a working set on which possible recovery sets are generated, a list containing used column indices and a list containing the answer are required.

The algorithm continues until an answer is returned or all possible cases have been tested.

The following functions are used. The overview of these functions is given as follows.

- *unused(column/columns)* - a function, which returns the truth value, whether the columns have not been marked as used.
- *answersSymbolAt(columns, symbol)* - function for determining whether the given columns will sum up to answer the requested symbol. If true, writes the answer of requested symbol to the correct position of *answer* variable.
- *setColumnsToUsed* - the column indices in question are marked as a used column to prevent reading one symbol in the codeword more times than allowed.
- *generateNextCombination* - a function, which takes a variable containing some combination, and tries to write the next combination in the given variable. Returns *TRUE* or *FALSE* value depending on the success of the process.
- *setTheFirstCombination(variable)* - sets the variable to contain the first possible combination.

At any iteration the following operations are performed:

1. Generate a new combination of column indices from the generator matrix with length equal to the size of the currently generated recovery set, starting with the sets of size one and finishing with the sets of size r.
2. Check that the set contains unused column indices and that it allows to recover the requested symbol.

Algorithm 4: Answering a request

Input: $\mathsf{type}_{\mathsf{arrayofnumeric}}$ *request*
Result: $\mathsf{type}_{\mathsf{array}}$ *answer*, $\mathsf{type}_{\mathsf{boolean}}$ *completed*

```
1  type_array answer
2  type_numeric position
3  while position ≥ 0 do
4      if unused(recoverySets[position]) and answersSymbolAt(
         recoverySets[position], request[position]) then
5          if position == requestLength then
6              return (answer, true);
7          else
8              setColumnsToUsed;
9              position++;
10             setSizes[position] = 1;
11     else
12         type_boolean breaked = false;
13         while generateNextCombination(recoverySets[position],
             setSizes[position]) do
14             if unused(recoverySets[position]) then
15                 breaked = true;
16                 break;
17             else
18                 continue;
19         if not breaked then
20             if setSizes[position] <r then
21                 setSizes[position]++;
22                 setTheFirstCombination(recoverySets[position]);
23             else
24                 position−;
25         else
26             continue;
27 return (answer, false);
```

(a) If the set meets these requirements, increase the variable that points to the set that is being constructed.

(b) If the set does not meet the requirements, try another set until all combinations have been tested. If no tested set allows for symbol recovery, increase the set size by one and repeat the procedure. If the set size is r, decrease the pointer and continue to the previous level.

3. If the *position* variable points to the last index of the request and the generated set allows for the recovery of the requested symbol, return the values of the requested symbols.

4. If the last combination of maximum size set at the first pointer level has reached the end without satisfying all the requests, the algorithm outputs a failure message.

When the function execution is finished, the requested symbols are either recovered or it is verified, that such a sequence of symbol can not be retrieved from a code constructed with the given generator matrix. The cyclic algorithm gives more control to the programmer, as it is possible to save the state of the program and continue from the previous state.

Non-exclusive licence to reproduce thesis and make thesis public

I, Mart Simisker (date of birth: 15th of August 1995),

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to:

1.1 reproduce, for the purpose of preservation and making available to the public, including for addition to the DSpace digital archives until expiry of the term of validity of the copyright, and

1.2 make available to the public via the web environment of the University of Tartu, including via the DSpace digital archives until expiry of the term of validity of the copyright,

Study of Optimal Linear Batch Codes

supervised by Vitaly Skachek

2. I am aware of the fact that the author retains these rights.

3. I certify that granting the non-exclusive licence does not infringe the intellectual property rights or rights arising from the Personal Data Protection Act.

Tartu, 11.05.2017