University of Latvia
Institute of Mathematics and Computer Science

Rubens Agadžanjans

A Complexity of Quantum Query Algorithms

Doctoral Thesis

Area: Computer Science
Sub-Area: Mathematical Foundations of Computer Science

Scientific Advisor:
Dr. habil. math., Prof.
Rūsiņš Freivalds

Riga 2010

Abstract

The query algorithms are a very convenient model for quantum complexity studies. In the thesis we study query algorithms for functions based on full Hamming codes and Reed-Solomon codes.

We show a way to construct exact quantum query algorithms for the both sets of functions and compare their complexity to the complexity of the most efficient deterministic algorithms.

Our algorithm for a Hamming code function of $m = 2^n - 1$ arguments needs $3/4 \cdot m$ queries to return the value of the function, which is 25% less than the classical complexity.

We achieve even better complexity improvement for the functions based on Reed-Solomon code. We show how to construct exact quantum query algorithm which needs only $m/2$ queries when the function has an even number m of arguments. This is a 50% improvement against the classical complexity and this repeats the best known improvement by exact quantum query algorithms.

We also prove a polynomial lower bound for the both sets of functions. For Hamming code this bound is 2^{n-2}, for Reed-Solomon code it is $n \cdot 2^{n-2}$.

We find an optimal adversary lower bound of Hamming code functions of three and seven arguments. This gives us a tight lower bound of two queries for the former function and a lower bound of three queries for the latter. Both these lower bounds are higher then the respective polynomial lower bounds.

Anotācija

Vaicājošie algoritmi ir viens no ērtākiem modeļiem kvantu sarežģītības pētīšanai. Promocijas darbā ir pētīti vaicājošie algoritmi funkcijām, kuras ir bāzētas uz pilniem Heminga un Rīda-Solomona kodiem.

Mēs rādām kā uzkonstruēt eksaktos kvantu vaicājošos algoritmus abām funkciju kopām un salīdzinām viņu sarežģītību ar visefektīvāko determinēto algoritmu sarežģītību.

Heminga koda funkcijai no $m = 2^n - 1$ argumentiem mūsu algoritmam pietiek ar $3/4 \cdot m$ kvantu vaicājumiem lai atgrieztu funkcijas vērtību, kas ir par 25% mazāk nekā klasiskā sarežģītība.

Mēs sasniedzam vēl lielāku sarežģītības uzlabojumu funkcijām bāzētām uz Rīda-Solomona kodiem. Mēs parādam kā uzkonstruēt eksakto kvantu vaicājošo algoritmu, kuram pietiek ar $m/2$ vaicājumiem kad funkcijai ir pāra skaits m argumentu. Tas ir 50% uzlabojums salīdzinājumā ar klasisko sarežģītību un tas atkārto vislabāko zināmo uzlabojumu eksaktiem kvantu vaicājošiem algoritmiem.

Mēs pierādām arī polinomiālo apakšējo novērtējumu abām funkciju kopām. Heminga kodiem šis novērtējums ir 2^{n-2}, Rīda-Solomona kodiem – $n \cdot 2^{n-2}$.

Mēs atrodam optimālo apakšējo novērtējumu ar "adversary" metodi Heminga koda funkcijām no trim un septiņiem argumentiem. Pirmajai funkcijai tas dod ciešo apakšējo novērtējumu, ka ir nepieciešami divi vaicājumi. Apakšējais novērtējums otrai funkcijai ir trīs vaicājumi. Abi šie novērtējumi ir labāki, nekā attiecīgie novērtējumi ar polinomiālo metodi.

Preface

This thesis assembles the research performed by the author and reflected in the following publications:

1. Rubens Agadžanjans, Rūsiņš Freivalds and Juris Smotrovs. An Exact Quantum Query Algorithm for a Hamming Code Function. *Proceedings of EQIS 2005: ERATO Conference on Quantum Information Science,* pp. 197-198, 2005.

2. Rubens Agadžanjans and Juris Smotrovs. Efficient Quantum Query Algorithms Detecting Hamming and Reed-Solomon Codes. *Proceedings of SOFSEM 2006: Theory and Practice of Computer Science, 32nd Conference on Current Trends in Theory and Practice of Computer Science* pp. 64-73, 2006.

3. Māris Ozols, Laura Mančinska, Ilze Dzelme-Bērziņa, Rubens Agadžanjans and Ansis Rosmanis. Principles of Optimal Probabilistic Decision Tree Construction. *FCS 2006: 2006 International Conference of Foundations of Computer Science*, CSREA Press, pp. 214-218, 2006.

4. Rubens Agadžanjans. Query algorithms for detecting Hamming and Reed-Solomon codes. *Proceedings of CiE 2008: Logic and*

Theory of Algorithms, Fourth Conference on Computability in Europe, 2008.

5. Rubens Agadžanjans, Rūsiņš Freivalds. Finite state transducers with intuition. *Unconventional Computation 2010*, Lecture Notes in Computer Science vol. 6079, pp. 11-20, 2010.

6. Abuzer Yakaryılmaz, Rūsiņš Freivalds, A. C. Cem Say and Rubens Agadžanjans. Quantum computation with devices whose contents are never read. *Unconventional Computation 2010*, Lecture Notes in Computer Science vol. 6079, pp. 164-174, 2010.

The results of the thesis were presented at the following international conferences and workshops:

1. 5th ERATO Conference on Quantum Information Systems (EQIS 2005), Tokyo, Japan, August 24-31, 2005. Poster "An Exact Quantum Query Algorithm for a Hamming Code Function".

2. 32nd International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM 2006). Merin, Czech Republic, January 21-27, 2006. Poster "Efficient Quantum Query Algorithms Detecting Hamming and Reed-Solomon Codes".

3. The 2006 International Conference on Foundations of Computer Science (FCS 2006). Las Vegas, Nevada, USA, June 26-29, 2006. Presentation "Principles of Optimal Probabilistic Decision Tree Construction".

4. Logic and Theory of Algorithms, Fourth Conference on Computability in Europe (Cie 2008). Athens, Greece, June 15-20,

2008. Presentation "Query algorithms for detecting Hamming and Reed-Solomon codes".

5. 9th International Conference on Unconventional Computation (UC 2010). Tokyo, Japan, June 21-25, 2010. Presentation "Finite state transducers with intuition".

6. 9th International Conference on Unconventional Computation (UC 2010). Tokyo, Japan, June 21-25, 2010. Presentation "Quantum computation with devices whose contents are never read".

The research was supported by the European Social Fund.

Acknowledgements

I gratefully thank my supervisor Prof. Rūsiņš Freivalds, whose support and ideas were one of the key factors for me to pursue and complete the research.

Thanks to Andris Ambainis who by his example encouraged me to dive into the area of quantum computation. Thanks to Juris Smotrovs for help in proving the polynomial lower bounds. Many thanks to Andrew Childs for familiarization with spectral adversary methods and advises in calculating adversary lower bounds for Hamming codes.

My gratitude to Alina Vasilieva for helping me with some administrative tasks and to Aleksandrs Rivosh and Nikolajs Nahimovs for sharing interesting ideas and concepts.

I greatfully thank my parents and family whose support was critical for writing up the thesis.

Also my gratitude to other colleagues, friends, and relatives who in one or other way helped me while working on this research.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

We will be researching quantum algorithms and their complexity. Almost all known quantum algorithms can be expressed in a query model where the input is given by a black box which answers queries in a certain form.

Classical algorithms also can be expressed in a query form. This gives us a possibility to compare the two models and see whether we can get any improvements by allowing quantum operations.

Query algorithms may be of different types. The main classification is in probability of returning a correct result. There are exact and probabilistic algorithms. Exact algorithms always return correct result. Probabilistic algorithms, in contrast, either return a result which is correct with some probability or always return a correct result, but there is a probability of returning "unknown".

The best improvements were achieved with probabilistic algorithms. This is actually one of the main reasons why quantum computation become popular and got a lot of support. The other reason is cryptography and secure communication, but this is out of scope of this

research.

The most attractive and most referenced improvements were presented by Peter Shor [Sh 97] and Lov Grover [Gr 96]. We're not going to talk much about these algorithms, but I still think that it's worth to explain what problems they are solving.

Shor's algorithm solves integer factorization problem in a polynomial time of integer's size. You may already know that this is an exponential improvement over a best known classical algorithm. So, it's practically impossible to factorize a large integer by a classical computer and modern cryptography heavily relies on this fact. The Shor's algorithm gained its popularity by the fact that it can break current cryptography model and make our digital communication insecure. Quantum cryptography addresses this issue, but again, I will not go into much details as it is out our scope.

Now consider a problem of searching. The setup is that we have an unsorted database of N entries and want to find a specific entry in this database. Classically, in the worst case, we need to check value of each element before we find the one we are looking for. Grover found an algorithm which will use just $O(\sqrt{N})$ queries to solve this problem in a quantum setting.

As I already mentioned, the algorithms above are of a probabilistic nature, most of the time they will return correct result, but there will be a probability of error.

In the thesis we are going to concentrate on the exact algorithms. There are surprising results in this area also.

The most interesting and attractive result was achieved for the following algorithmic problem introduced by Deutsch [De 85]. Informally it is a very simple problem of guessing whether a given coin is genuine

(with head on one side and tail on the other) or fake (with both sides the same). The question is how many times we need to look at the coin to find out which case it is. In the classical world twice, to both sides. In the quantum world only once, but to a quantum superposition (of both sides).

Cleve, Ekert, Macchiavello and Mosca [CE$^+$98] were the first who presented how to solve the above problem with one quantum query. We will use their result for constructing our algorithms.

This was the first result which achieved a 50% improvement for an exact algorithm. You may notice that this improvement is not as big as for probabilistic algorithms. So the question is whether we can do better? The reality is that since that time nobody provided a better gap, and there are not many algorithms achieving even the same result.

You may ask if it possible at all? At least this is the question which motivates this thesis. Nobody has the answer. There are several methods to calculate lower bounds for specific functions: polynomial [BB$^+$01], adversary [Am 02], and different variations of adversary [BSS 03, HLS 07]. For many functions even the first polynomial method provides high enough lower bounds being above the threshold of 50%, but for some functions lower bounds are below it. The Deutsch's problem is one of the former functions, so it is not interesting any more to try to improve its algorithm.

Obviously we are interested in the latter functions when trying to find an algorithm with a better improvement. In fact there is not so many of them known (see [NS 94, Am 03, NW 95, AF 03] for the examples).

We will introduce functions based on Hamming and Reed-Solomon error correcting codes. We will construct exact query algorithms for

them and compare complexities between quantum and classical versions. We will find lower bounds by polynomials and for some special cases also adversary lower bounds. We'll see the calculated lower bounds are below 50% threshold and so the functions are promising in the sense of existence of faster algorithm for processing them.

We will get 25% complexity improvement for Hamming code functions and 50% for Reed-Solomon functions. The latter repeats the result of solving the Deutsch's problem and still leaves a possibility of further improvement.

Chapter 2

Preliminaries

In this chapter we consider notations, definitions and well-known or elementary facts, referenced directly or indirectly further in the thesis. We refer to [Am 04] for the wording of most definitions in sections 2.1, 2.2 and 2.3.

2.1 Notation

$[N]$ denotes the set $\{1, \ldots, N\}$.

We use $\oplus$ to denote XOR (exclusive OR). If $x_1 \in \{0,1\}, x_2 \in \{0,1\}$, $x_1 \oplus x_2$ denotes XOR of bits x_1, x_2. If $x_1 \in \{0,1\}^n, x_2 \in \{0,1\}^n$, $x1 \oplus x_2$ denotes bitwise XOR of n-bit strings x_1 and x_2. We use $+$ to denote the usual addition of integers.

We use the O and Θ notation [CLR 90] standard in computer science. Let $f(N)$ and $g(N)$ be functions defined on positive integers N and taking positive values. We say that $f = O(g)$ if there exists a constant c such that $f(N) \leq cg(N)$. We say that $f = o(g)$ if, for any $c > 0$, there exists N_0 such that $f(N) \leq cg(N)$ for all $N > N_0$. We say

that $f = \Omega(g)$ if there exists $c > 0$ and N_0 such that $f(N) \geq cg(N)$ for all $N > N_0$.

2.2 Quantum Computing

We introduce the basic model of quantum computing. For more details, see textbooks by Gruska [Gr 99] and Nielsen and Chuang [NC 00].

2.2.1 Quantum States

We consider finite dimensional quantum systems. An n-dimensional pure quantum state is a vector $|\psi\rangle \in \mathbb{C}_n$ of norm 1. We use $|0\rangle$, $|1\rangle$, ..., $|n-1\rangle$ to denote an orthonormal basis for $\mathbb{C}^n$. Then, any state can be expressed as $|\psi\rangle = \sum_{i=0}^{n-1} a_i |i\rangle$ for some $a_0 \in \mathbb{C}, a_1 \in \mathbb{C}, \ldots, a_{n-1} \in \mathbb{C}$. Since the norm of $|\psi\rangle$ is 1, $\sum_{i=0}^{n-1} |a_i|^2 = 1$.

We call the states $|0\rangle, \ldots, |n-1\rangle$ *basis states*. Any state of the form $\sum_{i=0}^{n-1} a_i |i\rangle$ is called a *superposition* of $|0\rangle, \ldots, |n-1\rangle$. The coefficient a_i is called *amplitude* of $|i\rangle$. A quantum system can undergo two basic operations: a unitary evolution and a measurement.

2.2.2 Unitary Evolution

A unitary transformation U is a linear transformation on $\mathbb{C}^k$ that preserves the l_2 norm (i.e., maps vectors of unit norm to vectors of unit norm). If, before applying U, the system was in a state $|\psi\rangle$, then the state after the transformation is $U|\psi\rangle$.

2.2.3 Measurements

In the thesis, we just use the simplest case of quantum measurement. It is the full measurement in the computational basis. Performing this measurement on a state $|\psi\rangle = a_0 |0\rangle + \ldots + a_{n-1} |n-1\rangle$ gives the outcome i with probability $|a_i|^2$. The measurement changes the state of the system to $|i\rangle$. Notice that the measurement destroys the original state $|\psi\rangle$ and repeating the measurement gives the same i with probability 1 (because the state after the first measurement is $|i\rangle$).

2.3 Query Models

Query algorithms are used for computing functions of form $f : \{0,1\}^n \to \{0,1\}^m$. We will mostly consider boolean functions, that is those functions where $m = 1$.

In this model the boolean function is known, but arguments are kept in "black box". The aim is to compute the value of function making as less queries to "black box" as possible.

In the classical computation query algorithms are usually referenced as decision trees [Pa 94, BW 02]. The decision trees notation is not widely used for quantum algorithms as they don't have an explicit tree structure in most cases.

2.3.1 Deterministic Decision Tree

A *deterministic decision tree* is a rooted ordered binary tree T. Each internal node of T is labeled with a variable x_i and each leaf is labeled with a value 0 or 1. For given input $x \in \{0,1\}^n$ the evaluation starts at the root. If the current node is a leaf then the evaluation stops.

Otherwise the variable x_i that labels the current node is queried. If $x_i = 0$, then left subtree will be recursively evaluated, if $x_i = 1$ then the right one. The output of the tree is the value (0 or 1) of the leaf that is eventually reached. A deterministic decision tree computes f if its output equals $f(x)$, for every $x \in \{0,1\}^n$. The complexity of the decision tree is its depth, i.e., the number of queries made on the worst case input. It usually coincides with the number of arguments of function f and it never exceeds this number.

Definition 2.1. *The decision tree computes Boolean function* $f : \{0,1\}^n \to \{0,1\}$ *if for each input tuple* $X = (x_0, x_1, ..., x_{n-1})$, *computation goes to accepting leaf if* $f(X) = 1$ *and the computation ends in the rejecting leaf, if* $f(X) = 0$.

Definition 2.2. *Decision tree computes Boolean function* $f(X)$ *with complexity* k *if* k *is the number of oracle queries in the worst case (*k *is the depth of decision tree).*

Definition 2.3. *The decision tree complexity* $D(F)$ *of the Boolean function* $f(X)$ *is complexity of the best decision tree that computes* $f(X)$.

2.3.2 Quantum Query Model

There are two ways to define the query box in the quantum model. The first is an extension of the classical query. It has two inputs i, consisting of $\lceil \log N \rceil$ bits and b consisting of 1 bit. If the input to the query box is a basis state $|i\rangle\,|b\rangle$, the output is $|i\rangle\,|b \oplus x_i\rangle$. If the input is a superposition $\sum_{i,b} a_{i,b}\,|i\rangle\,|b\rangle$, the output is $\sum_{i,b} a_{i,b}\,|i\rangle\,|b \oplus x_i\rangle$. Notice that this definition applies both to the case when the x_i are binary and

to the case when they are k-valued. In the k-valued case, we just make b consist of $\lceil log_2 k \rceil$ bits and take $b \oplus x_i$ to be bitwise XOR of b and x_i.

In the second form of quantum query (which only applies to problems with $\{0,1\}$-valued x_i), the black box has just one input i. If the input is a state $\sum_i a_i |i\rangle$, the output is $\sum_i a_i(-1)x^i |i\rangle$. While this form is less intuitive, it is very convenient for use in quantum algorithms.

We will mostly use this second form in the thesis. This is possible to do because a query of the second type can be simulated by a query of the first type [Gr 96]. Conversely, an oracle of the first type can be simulated by a generalization of the sign oracle which receives $\sum_i a_{i,b} |i\rangle |b\rangle$ as an input and outputs $\sum_i a_i(-1)^{b \cdot x_i} |i\rangle |b\rangle$.

A quantum query algorithm with T queries is just a sequence of unitary transformations

$$U_0 \to O \to U_1 \to O \to \ldots \to U_{T-1} \to O \to U_T$$

on some finite-dimensional space $\mathbb{C}^k$. $U_0, U_1, \ldots, U_T$ can be any unitary transformations that do not depend on the bits $x_1, \ldots, x_N$ inside the black box. O are query transformations that consist of applying the black box to the first $\log N + 1$ bits of the state. That is, we represent basis states of $\mathbb{C}^k$ as $|i, b, z\rangle$. Then, O maps $|i, b, z\rangle$ to $|i, b \oplus x_i, z\rangle$. We use O_x to denote the query transformation corresponding to an input $x = (x_1, \ldots, x_N)$.

The computation starts with the state $|0\rangle$. Then, we apply $U_0, O_x, \ldots, O_x, U_T$ and measure the final state. The result of the computation is the rightmost bit of the state obtained by the measurement (or several bits if we are considering a problem where the answer has more than two values).

The quantum algorithm computes a function $f(x_1, \ldots, x_N)$ if, for every $x = (x_1, \ldots, x_N)$ for which f is defined, the probability that the rightmost bit of $U_T O_x U_{T-1} \ldots O_x U_0 |0\rangle$ equals $f(x_1, \ldots, x_N)$ is at least $1 - \epsilon$ for some fixed $\epsilon < 1/2$. The exact quantum algorithm computes a function with probability 1, i.e. $\epsilon = 0$.

The complexity of the quantum algorithm that computes f is a number of queries used by the algorithm. The quantum query complexity of function f is the complexity of the best quantum algorithm that computes f. We denote it by $Q(f)$. We denote the exact quantum query complexity by $Q_E(f)$.

2.4 The Deutsch Algorithm

Our quantum query algorithms employ the Deutsch algorithm [De 85, CE$^+$98] solving the XOR problem in a single query, where any classical algorithm makes at least two. This algorithm is exact. It can be described by the one qubit scheme $\rightarrow H \rightarrow O(x_1, x_2) \rightarrow H \rightarrow$ where H is the Hadamard gate, and $O(x_1, x_2)$ is the (only) oracle query encoding the answer in the phase:

$$H = \begin{pmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & \frac{-1}{\sqrt{2}} \end{pmatrix}, \quad O(x_1, x_2) = \begin{pmatrix} (-1)^{x_1} & 0 \\ 0 & (-1)^{x_2} \end{pmatrix}$$

If given $|0\rangle$ in the input, this scheme produces $|x_1 \oplus x_2\rangle$ in the output with probability 1.

2.5 Lower Bound Methods

We consider the both main techniques for proving lower bound on quantum query complexity.

The first technique is the polynomial method introduced in [BB$^+$01]. This approach is algebraic and follows earlier very successful work on classical lower bound by polynomials [Be 93, Re 97].

The other technique is the adversary method originally developed by Ambainis [Am 02] with roots in the hybrid method of [BB$^+$97]. It has proven to be a versatile technique, with formulations given by various authors in terms of spectral norms of matrices [BSS 03], weight schemes [Am 03, Zh 04], and Kolmogorov complexity [LM 04]. Spalek and Szegedy showed that all these versions of the adversary method are in fact equivalent [SS 05]. In 2007, Hoyer, Lee, and Spalek developed a new version of the adversary method using negative weights which is always at least as powerful as the standard adversary method, and can sometimes give better lower bounds [HLS 07].

We will use the spectral formulation of the adversary bound as this version best expresses similarity between the standard and negative adversary methods.

2.5.1 Lower Bound by Polynomials

The quantum complexity of f is related to *representing Boolean functions by polynomials.*

For any Boolean function $f(x_1, \ldots, x_N)$, there is a unique multilinear *polynomial* $p(x_1, \ldots, x_N)$ such that $f(x_1, \ldots, x_N) = p(x_1, \ldots, x_N)$ for all $x_1, \ldots, x_N \in \{0, 1\}$. For example, the function $f(x_1, x_2) =$

x_1 *OR* x_2 has polynomial $x_1 + x_2 - x_1x_2$. *The exact degree* of f is just the degree of the corresponding polynomial p. We denote it by $deg(f)$.

The degree can be used to determine a lower bound of f. It's proved that $D(f) \geq deg(f)$ and $Q_E(f) \geq deg(f)/2$ [BB$^+$01].

2.5.2 Spectral Adversary

We refer to the method's definition in [CL 08].

In this formulation the value of the adversary method for a function f is given by

$$ADV(f) := \max_{\substack{\Gamma \geq 0 \\ \Gamma \neq 0}} \frac{||\Gamma||}{max_i||\Gamma \circ D_i||}, \tag{2.1}$$

where Γ is a square matrix with rows and columns indexed by the possible inputs $x \in S \subseteq \{0,1\}^n$, constrained to satisfy $\Gamma[x,y] = 0$ if $f(x) = f(y)$; D_i is a zero/one matrix with $D_i[x,y] = 1$ if $x_i \neq y_i$ and 0 otherwise; $A \circ B$ denotes the Hadamard (i.e. entrywise) product of matrices A and B; and $\Gamma \geq 0$ means that the matrix Γ is entrywise non-negative.

The negative adversary method is of the same form, but removes the restriction to non-negative matrices in the maximization. Thus the value of the negative adversary method for a function f is given by

$$ADV^{\pm}(f) := \max_{\Gamma \neq 0} \frac{||\Gamma||}{max_i||\Gamma \circ D_i||}. \tag{2.2}$$

The relation of these adversary bounds to exact quantum query complexity is the following: $Q_E(f) \geq \frac{1}{2}ADV^{\pm}(f) \geq \frac{1}{2}ADV(f)$.

2.6 Galois Fields

We refer to [Bl 83] for the definitions of Galois Fields, associated terms and facts.

Definition 2.4. *A field is a set F together with two operations over this set - addition and multiplication, denoted accordingly by $+$ and $\cdot$, such that the following axioms hold:*

- *Closure of F under addition and multiplication:*

$$\forall a, b \in F : a + b \in F \ \& \ a \cdot b \in F$$

- *Associativity of addition and multiplication:*

$$\forall a, b, c \in F : a + (b + c) = (a + b) + c \ \& \ a \cdot (b \cdot c) = (a \cdot b) \cdot c$$

- *Commutativity of addition and multiplication:*

$$\forall a, b \in F : a + b = b + a \ \& \ a \cdot b = b \cdot a$$

- *Additive and multiplicative identity:*

$$\exists!\ 0 \in F \forall a \in F : a + 0 = a$$
$$\exists!\ 1 \in F \forall a \in F : a \cdot 1 = a$$

- *Additive and multiplicative inverses:*

$$\forall a \in F \exists -a \in F : a + (-a) = 0$$
$$\forall a \neq 0 \in F \exists a^{-1} \in F : a \cdot a^{-1} = 1$$

+	0	1
0	0	1
1	1	0

·	0	1
0	0	0
1	0	1

Table 2.1: GF(2).

+	0	1	2
0	0	1	2
1	1	2	0
2	2	0	1

·	0	1	2
0	0	0	0
1	0	1	2
2	0	2	1

Table 2.2: GF(3).

- *Distributivity of multiplication over addition:*

$$\forall a, b, c \in F : a \cdot (b + c) = (a \cdot b) + (a \cdot c)$$

The most common field examples are:

- Real numbers
- Rational numbers
- Complex numbers

Definition 2.5. *Finite or Galois field is a field with finite number of elements. If the number of elements in the field is q then the field is denoted by $GF(q)$.*

See tables 2.1, 2.2, 2.3 for examples of Galois fields.

Notice that in the GF(4) addition is not by modulus 4, and multiplication is also not by modulus 4.

+	0	1	2	3
0	0	1	2	3
1	1	0	3	2
2	2	3	0	1
3	3	2	1	0

·	0	1	2	3
0	0	0	0	0
1	0	1	2	3
2	0	2	3	1
3	0	3	1	2

Table 2.3: GF(4).

In order to construct Galois fields we need to know how to construct addition and multiplication tables. First we can begin with the definition of a polynomial over a Galois field and associated definitions.

Definition 2.6. *Polynomial over $GF(q)$ is a mathematical expression $f(x) = f_{n-1}x^{n-1} + f_{n-2}x^{n-2} + \ldots + f_1x + f_0$, where x is a free variable, coefficients belong to $GF(q)$, indices and powers are integers.*

Zero polynomial is a polynomial $f(x) = 0$.

Reduced polynomial is a polynomial with higher coefficient f_{n-1} is equal to 1.

Two polynomials are equal if all their coefficients f_i are equal.

Definition 2.7. *Sum of polynomials is defined as*

$$f(x) + g(x) = \sum_{i=0}^{\infty} (f_i + g_i)x^i$$

Definition 2.8. *Multiplication of polynomials is defined as*

$$f(x)g(x) = \sum_{i} (\sum_{j=0}^{i} f_j g_{i-j})x^i$$

Definition 2.9. *Primary polynomial is a reduced polynomial which can be divided only by itself or by 1.*

Degree	Primary polynomial	Degree	Primary polynomial
2	$x^2 + x + 1$	14	$x^{14} + x^{10} + x^6 + x + 1$
3	$x^3 + x + 1$	15	$x^{15} + x + 1$
4	$x^4 + x + 1$	16	$x^{16} + x^{12} + x3 + x + 1$
5	$x^5 + x^2 + 1$	17	$x_{17} + x^3 + 1$
6	$x^6 + x + 1$	18	$x^{18} + x^7 + 1$
7	$x^7 + x^3 + 1$	19	$x^{19} + x^5 + x^2 + x + 1$
8	$x^8 + x^4 + x^3 + x^2 + 1$	20	$x^{20} + x^3 + 1$
9	$x^9 + x^4 + 1$	21	$x^{21} + x^2 + 1$
10	$x^{10} + x^3 + 1$	22	$x^{22} + x + 1$
11	$x^{11} + x^2 + 1$	23	$x^{23} + x^6 + 1$
12	$x^{12} + x^6 + x^4 + x + 1$	24	$x^{24} + x^7 + x^2 + x + 1$

Table 2.4: The primary polynomials over $GF(2)$. All the polynomials are primitive.

Table 2.4 lists primary polynomials over GF(2). These polynomials can be used to construct larger fields $GF(2^m)$, where m is the degree of a polynomial.

To get Galois field from the table 2.4 it's needed to take a polynomial of degree m (let's denote it by p_m) and to make a set consisting of polynomials by modulus p_m.

All the polynomials $p(x)$ from the table 2.4 are primitive - that means that x can be used as a primitive element in a field constructed as a polynomials by modulus $p(x)$. We define a primitive element below.

Definition 2.10. *$GF(q)$ primitive element is such an element α that all field's elements except zero can be expressed as a power of α.*

For instance, 2 is a primitive element of $GF(5)$: $2^1 = 2$, $2^2 = 4$, $2^3 = 3$, $2^4 = 1$.

Another example of a primitive element x (now it's in a polynomial form) for field $GF(4)$, constructed as polynomials by modulus x^2+x+1. The calculations below are by modulus $x^2 + x + 1$:

$$x^1 = x$$
$$x^2 = x + 1$$
$$x^3 = x(x + 1) = 1$$

Definition 2.11. *Let $GF(Q)$ be an extension of $GF(q)$ and $\beta \in GF(Q)$. The polynomial $f(x)$ of a lowest degree over $GF(q)$ with property that $f(\beta) = 0$ is called an element's β minimal polynomial over $GF(q)$.*

The main properties of Galois fields:

1. Elements count in any Galois field is a power of prime number.

2. For any prime number p and positive integer m the smallest $GF(p^m)$ subfield is the field $GF(p)$. Elements of $GF(p)$ are called a field's $GF(p^m)$ integers, and p - a characteristic.

3. For Galois fields of characteristic 2, it is true for each element β that $-\beta = \beta$.

4. For any prime number p and positive integer m always exists a Galois field of p^m elements.

5. Every Galois field contains at least one primitive element.

6. There is always at least one primitive polynomial of any positive power over each Galois field.

7. Each primitive element has a primary minimal polynomial over any subfield.

8. Two Galois fields with the same number of elements are isomorphic.

9. for any q which is a power of a prime number, and any positive integer m field $GF(q)$ is a subfield in $GF(q^m)$, but $GF(q^m)$ is an extension of field $GF(q)$.

10. If n is not dividing m then $GF(q^n)$ is not a subfield of $GF(q^m)$

11. For any element of $GF(q^m)$ the power of minimal polynomial over $GF(q)$ divides m.

Chapter 3

Hamming Codes

Hamming codes were introduced in [Ha 50] and very soon they became common knowledge for everybody in computer science. They contain many symmetries and other good properties, and so are used widely in the theoretical computer science.

Hamming distance between two n-bit vectors a and b is the number of the positions at which these vectors differ. *Hamming weight* of a vector is defined as the number of 1's in it.

In this thesis we will use *Hamming codes correcting one error*. They can be defined as 0-1 N-vectors where N equals $2^n - 1$ for some integer $n > 1$, with the following property: $i_1 \oplus i_2 \oplus \ldots \oplus i_m = 0$ where $i_1, i_2, \ldots, i_m \in [2^n - 1]$ are the positions (or indices) of 1's in the vector, and $\oplus$ is the bitwise addition modulo 2 of the binary representations of numbers from $[2^n - 1]$. In total there are $2^{2^n - n - 1}$ such code vectors. See tables 3.1 and 3.2 for the valid Hamming codewords of size three and seven.

Another way of checking whether a 0-1 N-vector $(x_1, \ldots, x_N)$ is a correct Hamming code is to verify whether n checksums are equal to

X_1	X_2	X_3
0	0	0
1	1	1

Table 3.1: The valid Hamming codewords of size three.

X_1	X_2	X_3	X_4	X_5	X_6	X_7
0	0	0	0	0	0	0
1	1	1	0	0	0	0
1	0	0	1	1	0	0
0	1	1	1	1	0	0
0	1	0	1	0	1	0
1	0	1	1	0	1	0
1	1	0	0	1	1	0
0	0	1	0	1	1	0
1	1	0	1	0	0	1
0	0	1	1	0	0	1
0	1	0	0	1	0	1
1	0	1	0	1	0	1
1	0	0	0	0	1	1
0	1	1	0	0	1	1
0	0	0	1	1	1	1
1	1	1	1	1	1	1

Table 3.2: The valid Hamming codewords of size seven.

0. Let $\overline{j_n j_{n-1} \dots j_1}$ be the binary representation of a variable index j. The variable x_j is present in the m-th checksum iff $j_m = 1$. This gives us the following checksums:

$$x_1 \oplus x_3 \oplus \dots \oplus x_{2^n-3} \oplus x_{2^n-1} = 0 - the\ first\ checksum$$
$$x_2 \oplus x_3 \oplus \dots \oplus x_{2^n-2} \oplus x_{2^n-1} = 0 - the\ second\ checksum$$
$$\dots$$
$$x_{2^{n-1}} \oplus x_{2^{n-1}+1} \oplus \dots \oplus x_{2^n-2} \oplus x_{2^n-1} = 0 - the\ n^{th}\ checksum$$

We will need one more representation of the same verification schema to prove lower bound by polynomials of Reed-Solomon code functions in the section 4.4.

Lemma 3.1. *Hamming code can be verified by multiplying the input variable vector* $[x_1 \dots x_N]$ *by a checking matrix* Λ *of* N *rows and* n *columns. The code is correct if the multiplication's result is equal to zero-vector.*

Moreover, each row of the matrix Λ *is unique and the rows are all possible* n*-bit vectors except zero-vector.*

Proof. We construct the checking matrix based on the variable indexes. It has N rows and n columns. Its j^{th} row corresponds to an index j by representing it in a binary form in such a way that the value in column m equals to j_m from the binary representation of j. In other words it can be written as $\Lambda_{j,m} = j_m$, where $j = \overline{j_n j_{n-1} \dots j_1}, j > 0$. Multiplying a code-word vector by Λ and verifying that the result is a zero-vector is exactly the same as verifying the checksums in the verification method above.

Since each row of Λ represents a distinct j all the rows are unique. Presence of all possible n-bit vectors follows from the fact that $N = 2^n - 1$, which is exactly the number of distinct n-bit non-zero vectors. □

The following is a sample checking matrix for Hamming code of seven arguments:

$$\Lambda = \begin{bmatrix} 100 \\ 010 \\ 110 \\ 001 \\ 101 \\ 011 \\ 111 \end{bmatrix}$$

3.1 Deterministic Query Algorithm for Hamming Codes

In this and the following sections we denote $N = 2^n - 1$.

We define a Boolean function $f(b_1, \ldots, b_N)$ equal to 1 on Hamming code vectors and equal to 0 otherwise.

Theorem 3.2. $D(f) = N$.

Proof. The all-zero vector is a correct Hamming code (all checksums are 0), while any vector containing exactly one 1 is not a Hamming code (the checksums containing the only 1 are equal to 1). Thus, receiving only zeroes in answers to the queries for variable values, the

deterministic algorithm needs to query all N variables before producing result 1: to ascertain that no argument is equal to 1. □

The visualization of the deterministic algorithm for the function of seven arguments can be seen on figure 3.1.

3.2 Quantum Query Algorithm for Hamming Codes

Theorem 3.3. *There is an exact quantum query algorithm with complexity* $3 \cdot 2^{n-2} - 1$ *for the function* f.

Proof. We use verification of checksums to construct the algorithm. Each checksum consists of 2^{n-1} variables. So we need 2^{n-2} queries to compute it using the Deutsch 1-query algorithm for XOR of two variables. For example, we query $x_1 \oplus x_3, x_5 \oplus x_7, \ldots, x_{N-2} \oplus x_N$ to compute the first checksum $x_1 \oplus x_3 \oplus \ldots \oplus x_N$.

While calculating checksums, we query for XOR of pairs of variables. Some pairs will occur in more than one checksum. We will save several queries by asking such pairs only once and by remembering results of these queries. It remains to calculate how many queries we save in such way.

Let us analyze checksums starting with the second one, i.e. the checksums which control variables whose index binary representation contains 1 in positions $2, 3, \ldots$.

If a variable with even index $2i$ belongs to such checksum, then the variable with index $2i + 1$ also belongs to this checksum (because the binary representations of the indices of these variables differ only in the last bit).

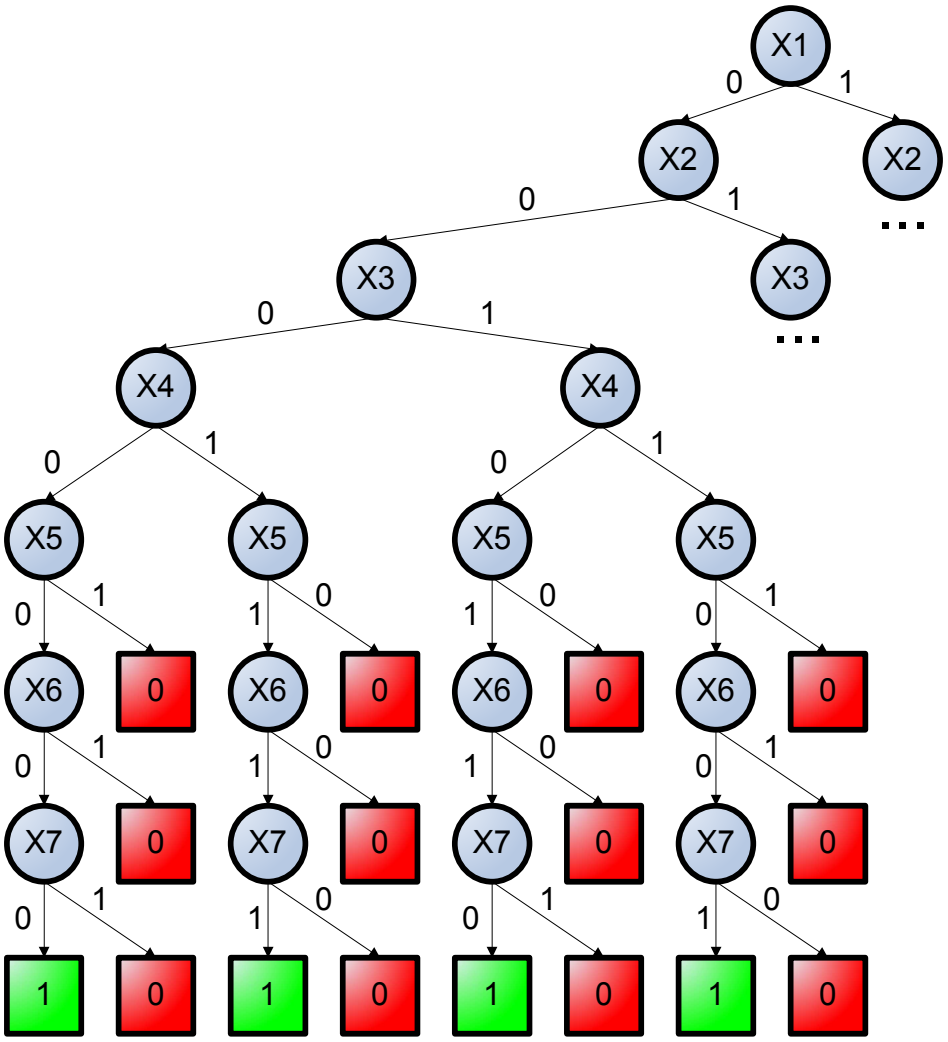


Figure 3.1: A deterministic query algorithm for the Hamming code function of seven arguments.

In total we have $(2^n - 2)/2 = 2^{n-1} - 1$ such pairs ($2i$ and $2i + 1$). Thus we can compute XOR of all these pairs with $2^{n-1} - 1$ queries, and knowing their XOR we can verify all checksums starting from the second one. Additionally we make 2^{n-2} queries to verify the first checksum. So we make $2^{n-1} - 1 + 2^{n-2} = 3 \cdot 2^{n-2} - 1$ queries to verify all checksums and thus evaluate the function.

As an example, let us construct such algorithm for the function $f(x_1, \ldots, x_7)$. Valid keywords are:

0000000 1110000 1001100 0111100
0101010 1011010 1100110 0010110
1101001 0011001 0100101 1010101
1000011 0110011 0001111 1111111

Checksums to be verified:

$$x_1 \oplus x_3 \oplus x_5 \oplus x_7 = 0$$
$$x_2 \oplus x_3 \oplus \mathbf{x_6} \oplus \mathbf{x_7} = 0$$
$$x_4 \oplus x_5 \oplus \mathbf{x_6} \oplus \mathbf{x_7} = 0$$

Variable pairs to be asked: $x_2 \oplus x_3$; $x_4 \oplus x_5$; $x_6 \oplus x_7$; $x_1 \oplus x_3$; $x_5 \oplus x_7$. □

Figure 3.2 visualizes the quantum algorithm for the Hamming code of size seven.

3.3 Lower Bound by Polynomials

Now we will find the degree of f which helps us to prove the lower bound by polynomials.

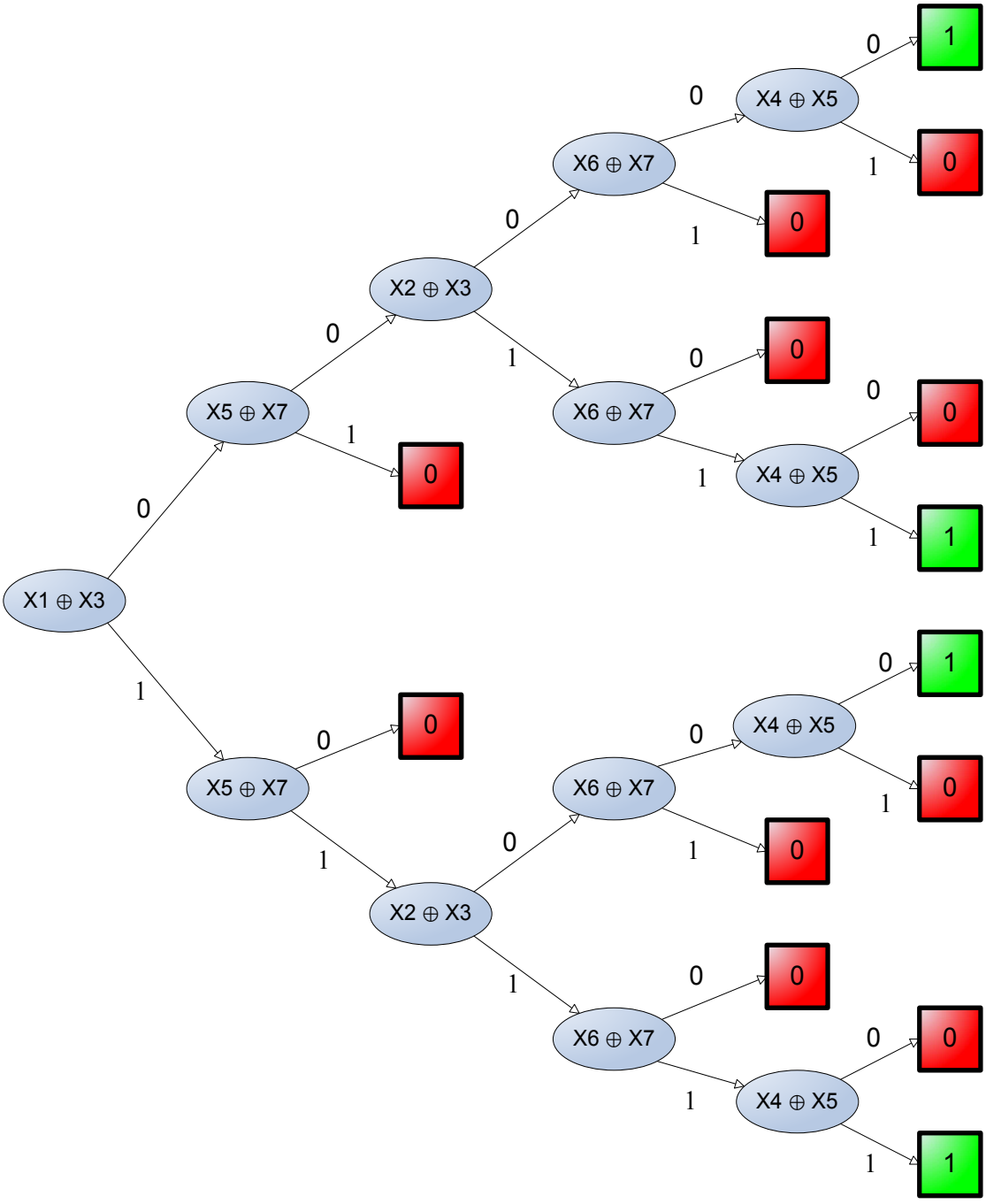


Figure 3.2: A quantum query algorithm for the Hamming code function of seven arguments.

Theorem 3.4. $deg(f) = 2^{n-1}$.

Proof. The polynomial of f can be expressed as $p(x_1, \ldots, x_N) = \sum_{b:f(b)=1} \prod_{i:b_i=1} x_i \prod_{i:b_i=0} (1 - x_i)$. In the following we will refer to the elements of this sum as "the summands". Thus each summand corresponds to a particular Hamming code vector b.

Now, let us open all the parentheses $(1 - x_i)$ expressing each summand as a sum of one or more monomials (referred so in further) of the kind $\prod x_i$ with (non-zero) coefficients. Let us investigate in which summands can a particular monomial occur. If a monomial from a summand *does not* contain a particular variable x_j, it means that the summand contained a factor $(1-x_j)$ (otherwise all its monomials would contain x_j). And, vice versa, the presence of the factor $(1 - x_j)$ in the summand means that the summand's monomials can be split into pairs which differ only in the presence/absence of the variable x_j.

Thus, for $I \subseteq [N]$, the monomial $\prod_{i \in I} x_i$ occurs exactly in the summands with such b that $\forall j \in [N] \setminus I$ $(b_j = 0)$, and the coefficient of the monomial in each occurrence is 1 or -1.

Suppose that $card(I) > 2^{n-1}$. We will show that, in the whole sum, the monomial has coefficient 1 exactly as many times as coefficient -1, resulting in the canceling of the monomial. How is the sign of a monomial obtained? When opening parentheses in a summand, -1 appears only when we pick the $-x_i$ from a factor $(1 - x_i)$ to obtain the monomial. If there is an even number of such $-x_i$ picked, the coefficient of the monomial is 1, otherwise it is -1. We must pick $-x_i$ from $(1 - x_i)$ iff $i \in I$. Thus, coefficient is -1 iff the number of zeroes among b_i, $i \in I$ is odd. Since, as we established, $b_i = 0$ for $i \notin I$, two possible monomial coefficients 1 and -1 correspond to two opposite

parities of the number of 1's in the code vector of the summand.

Thus, it is enough to prove that among the Hamming codes with 0's in all the positions not from I, the number of codes with even Hamming weight must coincide with the number of codes with odd Hamming weight. It is enough to prove that among these there is a code c with Hamming weight 3, since then, due to the linearity of Hamming codes, all these codes can be split into pairs $(b, b \oplus c)$, with opposite Hamming weight parities. By definition, a 0-1 vector with weight 3 and 1's in positions i, j, k is a Hamming code iff $i \oplus j \oplus k = 0$. Select an $i \in I$. Now all the positions from $[N] \setminus \{i\}$ are split into $2^{n-1} - 1$ pairs (j, k) with the property $i \oplus j \oplus k = 0$. Since $card([N] \setminus I) \leq 2^{n-1} - 2$, at least one of these pairs (j, k) gives a triple $i, j, k \in I$ with $i \oplus j \oplus k = 0$. Upper bound proved.

It remains to show that there is a monomial with $card(I) = 2^{n-1}$ and non-zero coefficient. A Hamming code b must satisfy $b_1 \oplus b_3 \oplus \ldots \oplus b_N = 0$. Hence the number of 1's in the positions from $I = \{1, 3, \ldots, N\}$ must be even. Since $card(I) = 2^{n-1}$ is even, also the number of 0's in positions from I is even, and the corresponding monomial occurs in the sum only with coefficients $+1$ and thus cannot cancel (and it occurs at least once - for the all-zero Hamming code). □

Corollary 3.5. *The lower bound by polynomials for Hamming code function is:* $Q_E(f) \geq 2^{n-2}$.

We continue the research with finding adversary bounds for Hamming codes.

3.4 Adversary Bounds for Hamming Codes

We will be following the principles from [CL 08] when applying the adversary method to Hamming codes.

Finding the value of the adversary method is as an optimization problem. To analyze the adversary bound for hamming codes, we will use symmetry to simplify this problem. The same simplification applies to both the standard and negative adversary bounds, so we treat the two cases simultaneously.

Suppose we are trying to design a good adversary matrix Γ, and are deciding what weight to assign the (x, y) entry. Intuitively, it seems that if (x, y) and (x', y') are related by an automorphism, then they should look the same to an adversary, and hence should be given the same weight. The automorphism principle states that there is an optimal adversary matrix with this property. Although this principle does not provide any advice about what weight to give a particular pair (x, y), it can vastly reduce the optimization space by indicating that the adversary matrix should possess certain symmetries.

Definition 3.6. *([HLS 07]) Let G be a group of automorphisms for a function f. We say that G is* f-transitive *if for every x, y such that $f(x) = f(y)$, there is $\pi \in G$ such that $\pi(x) = y$.*

Theorem 3.7. *(Automorphism principle [HLS 07]) Let G be a group of automorphisms of f. There is an optimal adversary matrix Γ for which $\Gamma[x, y] = \Gamma[\pi(x), \pi(y)]$ for all $\pi \in G$ and x, y. Furthermore, if G is f-transitive then Γ has a principal eigenvector β for which $\beta[x] = \beta[y]$ whenever $f(x) = f(y)$.*

There are two types of automorphisms in the automorphism group

of Hamming code function:

1. "Linear" automorphism. Linearity of the Hamming codes means that we can add one Hamming codeword to another and the result will still be a valid Hamming codeword. Each valid Hamming codeword h forms an automorphism which maps vector v to another vector $v \oplus h$. If v is a valid Hamming code, then and only then $v \oplus h$ is also a valid Hamming code.

2. "Cyclic" automorphism. It's well known that the Hamming codes are cyclic codes ([Bl 83]). This means that there is a cyclic operator ϕ which maps a Hamming code h of length N to another Hamming code $\phi(h)$ and by applying this operator N times we get the h again: $\phi^N(h) = h$.

Theorem 3.8. *The group G of "linear" and "cyclic" automorphisms of a Hamming code function of N arguments is f-transitive.*

Proof. We get $N+1$ distinct groups of code vectors. The first group F_0 consists of all valid Hamming codewords. This group can be generated by applying each of the "linear" automorphisms to codeword of all-zeros. Obviously, we can transform a codeword $s \in F_0$ to another codeword $t \in F_0$ by applying sequentially two automorphisms which are accordingly based on s and t: $s \oplus s \oplus t = t$.

The rest N groups F_i are generated by applying the "linear" automorphisms to code vector containing just one "1" in position i and all zeros in other positions. These groups are distinct as they are based on distinct starting vectors and we can't get another vector with hamming weight 1 by applying the "linear" automorphisms to the starting

vector (just because there is no any valid Hamming codeword of weight 2).

Since we have a "cyclic" automorphism then for each i, j we can transform elements from F_i into elements of F_j by applying a "shift" automorphism necessary number of times.

This means that we can transform any element $f_i \in F_i$ into element $f_j \in F_j$ which together with the transitivity of F_0 elements is the condition for the automorphism group G to be f-transitive. □

By following the automorphism principle 3.7 we can assume without loss of generality that the number of distinct entries in matrix Γ is at most the number of elements in F_0, which is number of distinct valid codewords. Hamming code of $N = 2^n - 1$ arguments has $2^n - n - 1$ valid codewords. Let's denote this number by $size(N)$. So we will have at most $size(N)$ distinct entries in Γ.

Let's take a look into the structure of Γ.

Lemma 3.9. *The row $\Gamma[z,]$, corresponding to all-zeros codeword, consists of $size(N)$ zeros and $size(N)$ distinct elements each of them appearing exactly N times.*

Proof. The row z has $size(N)$ zeros in the columns corresponding to valid codewords, because by definition Γ has a constraint that if $f(x) = f(y)$, then $\Gamma[x, y] = 0$.

Some of the columns correspond to cyclically isomorphic code vectors. By cyclically isomorphic code vectors I mean such code vectors which can be got from one another by applying the "cyclic" automorphism.

Entries corresponding to columns of cyclically isomorphic code vectors will be equal between one another. We can see this by having

$x = 0 \dots 0$ and π equal to "cyclic shift" operator when applying the equation from the automorphism principle:

$$\Gamma[0 \dots 0, y] = \Gamma[shift(0 \dots 0), shift(y)] = \Gamma[0 \dots 0, shift(y)]$$

□

Lemma 3.10. *The row $\Gamma[v,]$, corresponding to a valid codeword v, consists of exactly the same elements as the row $\Gamma[z,]$, permuted across columns.*

Proof. We can use the "linear" automorphisms to see the structure for the other rows corresponding to valid codewords. For a valid codeword v and corresponding "linear" automorphism we have the following equation: $\Gamma[v, x] = \Gamma[v \oplus v, x \oplus v] = \Gamma[z, x \oplus v]$. □

Lemma 3.11. *The rows of Γ corresponding to not valid code vectors consist of $N \cdot size(N)$ zeros and $size(N)$ distinct elements.*

Proof. These rows will have zeros in all columns corresponding to not valid code vectors. The columns which correspond to valid codewords will have $size(n)$ distinct entries. □

Theorem 3.12. *If u is not a valid code vector of a Hamming code of N arguments, then sum of the entries in the u^{th} row of Γ is equal to $\sigma = \sum_{i \in S} \Gamma[u, i] = \sum_{j \in H} \gamma_j$, where S is a set of all possible code vectors; H is a set of all valid codewords.*

For the valid codeword v the sum of the entries in the v^{th} row of Γ is equal to $\sigma \cdot N$.

Proof. The proof follows from lemmas 3.9, 3.10, and 3.11. □

Following is the sample Γ matrix for the Hamming code of size three:

$$\Gamma = \begin{array}{c} \begin{matrix} 000 & 001 & 010 & 011 & 100 & 101 & 110 & 111 \end{matrix} \\ \begin{bmatrix} 0 & \gamma_1 & \gamma_1 & \gamma_2 & \gamma_1 & \gamma_2 & \gamma_2 & 0 \\ \gamma_1 & 0 & 0 & 0 & 0 & 0 & 0 & \gamma_2 \\ \gamma_1 & 0 & 0 & 0 & 0 & 0 & 0 & \gamma_2 \\ \gamma_2 & 0 & 0 & 0 & 0 & 0 & 0 & \gamma_1 \\ \gamma_1 & 0 & 0 & 0 & 0 & 0 & 0 & \gamma_2 \\ \gamma_2 & 0 & 0 & 0 & 0 & 0 & 0 & \gamma_1 \\ \gamma_2 & 0 & 0 & 0 & 0 & 0 & 0 & \gamma_1 \\ 0 & \gamma_2 & \gamma_2 & \gamma_1 & \gamma_2 & \gamma_1 & \gamma_1 & 0 \end{bmatrix} \end{array} \begin{matrix} \\ 000 \\ 001 \\ 010 \\ 011 \\ 100 \\ 101 \\ 110 \\ 111 \end{matrix}$$

Theorem 3.13. *Matrix Γ of a Hamming code function of N arguments has a principal eigenvector β for which $\beta[x] = \sqrt{N}$ whenever $f(x) = 1$, and $\beta[x] = 1$ whenever $f(x) = 0$.*

Proof. Since we proved theorem 3.8 we can follow the second part of the automorphism principle 3.7 and construct the principal eigenvector β. It will have distinct entries a and b corresponding accordingly to valid and invalid Hamming code vectors.

By the definition of eigenvector it needs to satisfy the following equality:

$$\Gamma\beta = \lambda\beta, \tag{3.1}$$

where λ is the corresponding eigenvalue.

We have

$$\Gamma\beta = \Gamma \begin{bmatrix} a \\ b \\ \vdots \\ b \\ a \end{bmatrix} = \begin{bmatrix} b\sigma N \\ \sigma a \\ \vdots \\ \sigma a \\ b\sigma N \end{bmatrix} = \sigma \begin{bmatrix} bN \\ a \\ \vdots \\ a \\ bN \end{bmatrix} \tag{3.2}$$

and

$$\lambda\beta = \lambda \begin{bmatrix} a \\ b \\ \vdots \\ b \\ a \end{bmatrix} \tag{3.3}$$

We get the following dependency for a and b if we put 3.2 and 3.3 into 3.1:

$$\begin{cases} \sigma bN = \lambda a \\ \sigma a = \lambda b \end{cases} \Rightarrow \begin{cases} \sigma bN = \frac{\lambda^2 b}{\sigma} \\ a = \frac{\lambda b}{\sigma} \end{cases} \Rightarrow \begin{cases} \sigma^2 bN = \lambda^2 b \\ a = \frac{\lambda b}{\sigma} \end{cases} \Rightarrow \begin{cases} b(\sigma^2 N - \lambda^2) = 0 \\ a = \frac{\lambda b}{\sigma} \end{cases}$$

$$\lambda = \pm\sigma\sqrt{N} \tag{3.4}$$

$$a = \pm b\sqrt{N}$$

Since in this case sign of entries doesn't affect modulus of λ we can chose both a and b to have the same sign:

$$a = b\sqrt{N}$$

We construct β, by having $a = \sqrt{N}, b = 1$. □

Theorem 3.14. $||\Gamma|| = |\sigma\sqrt{N}|$.

Proof. The proof follows from the fact that Γ is symmetric by design: $\Gamma[x, y] = \Gamma[y, x]$.

Since Γ is symmetric, its norm is equal to the modulus of eigenvalue of a principal eigenvector, which we already expressed in the equation 3.4. □

Now we know the value of the numerator in formula 2.2. Let's now take a look on the denominator $max_i||\Gamma \circ D_i||$.

Transitivity of the automorphism group implies that all matrices $\Gamma \circ D_i$ have the same norm ([HLS 07]), so it is sufficient to consider $\Gamma \circ D_1$. Considering the example of $N = 3$, we have

$$\Gamma \circ D_1 = \begin{bmatrix} 0 & 0 & 0 & 0 & \gamma_1 & \gamma_2 & \gamma_2 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & \gamma_2 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & \gamma_2 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & \gamma_1 \\ \gamma_1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \gamma_2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \gamma_2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & \gamma_2 & \gamma_2 & \gamma_1 & 0 & 0 & 0 & 0 \end{bmatrix}$$

This matrix consists of two disjoint, symmetrical blocks, so its spectral norm is simply the spectral norm of one of those blocks.

3.4.1 Optimal Adversary Bounds for Hamming Code of Three Arguments

We can find the norm of the $\Gamma \circ D_1$ for the case when $N = 3$.

Theorem 3.15. *For the Hamming code of three arguments ($N = 3$) the norm of* $\Gamma \circ D_1 = \sqrt{\gamma_1^2 + 2\gamma_2^2}$

Proof. First we can reverse the order of columns. This operation doesn't change the norm, but it makes the matrix to be symmetric. For symmetric matrix the norm is just the biggest eigenvalue (by absolute value).

In order to find the eigenvalues we need to solve the the following equation:

$$\begin{aligned}
\begin{vmatrix} -\lambda & \gamma_2 & \gamma_2 & \gamma_1 \\ \gamma_2 & -\lambda & 0 & 0 \\ \gamma_2 & 0 & -\lambda & 0 \\ \gamma_1 & 0 & 0 & -\lambda \end{vmatrix} &= -\gamma_2 \begin{vmatrix} \gamma_2 & 0 & 0 \\ \gamma_2 & -\lambda & 0 \\ \gamma_1 & 0 & -\lambda \end{vmatrix} - \lambda \begin{vmatrix} -\lambda & \gamma_2 & \gamma_1 \\ \gamma_2 & -\lambda & 0 \\ \gamma_1 & 0 & -\lambda \end{vmatrix} \\
&= -\gamma_2^2\lambda^2 + \lambda^4 - \lambda^2\gamma_1^2 - \lambda^2\gamma_2^2 \\
&= \lambda^4 - 2\lambda^2\gamma_2^2 - \lambda^2\gamma_1^2 = 0
\end{aligned}$$

This equation has four solutions. Two of them are $\lambda_{1,2} = 0$. The other two solutions are the following:

$$\lambda_{3,4} = \pm\sqrt{\gamma_1^2 + 2\gamma_2^2}$$

Which means that the norm of the matrix is $\sqrt{\gamma_1^2 + 2\gamma_2^2}$. □

Theorem 3.16. *The optimal adversary and optimal negative adversary bounds of Hamming code function of three arguments are equal to*

$\frac{3}{\sqrt{2}}$:

$$ADV^{\pm}(Hamming(3)) = ADV(Hamming(3)) = \frac{3}{\sqrt{2}}$$

Proof. The formula for the standard adversary bound is (2.1):

$$ADV(f) := \max_{\substack{\Gamma \geq 0 \\ \Gamma \neq 0}} \frac{||\Gamma||}{max_i ||\Gamma \circ D_i||},$$

From the transitivity of automorphism group and from theorems 3.14 and 3.15 it follows that without loss of generality the formula can be reduced to a more simple form:

$$ADV(Hamming(3)) = \max_{\substack{\Gamma \geq 0 \\ \Gamma \neq 0}} \frac{(\gamma_1 + \gamma_2)\sqrt{3}}{||\Gamma \circ D_1||} = \max_{\substack{\Gamma \geq 0 \\ \Gamma \neq 0}} \frac{(\gamma_1 + \gamma_2)\sqrt{3}}{\sqrt{\gamma_1^2 + 2\gamma_2^2}}.$$

In this case we cannot get any improvement from allowing negative entries in Γ, so we can safely assume that $ADV^{\pm} = ADV$.

The formula reaches it's maximum when $\gamma_1 = 2\gamma_2$:

$$ADV^{\pm}(Hamming(3)) = ADV(Hamming(3)) = \frac{(2\gamma_2 + \gamma_2)\sqrt{3}}{\sqrt{(2\gamma_2)^2 + 2\gamma_2^2}} = \frac{3}{\sqrt{2}}.$$

□

The same result can be achieved by solving a positive semidefinite optimization problem of maximizing $\gamma_1 + \gamma_2$ subject to constraint that $I - \Gamma \circ D_1$ and $I + \Gamma \circ D_1$ are positive semidefinite.

Now, knowing the adversary bound we can tell the lower bound for quantum exact algorithm.

Corollary 3.17.

$$Q_E(Hamming(3)) = \frac{ADV(Hamming(3))}{2} = \frac{3}{2\sqrt{2}} \approx 1.06...$$

This means that minimal number of queries for an exact algorith is 2, so our algorithm for Hamming code of three arguments is optimal.

3.4.2 Optimal Adversary Bounds for Hamming Code of Seven Arguments

Γ for the Hamming code of seven arguments consists of 16 distinct non-zero entries $\gamma_1 \dots \gamma_{16}$. This fact follows from lemmas 3.9, 3.10, and 3.11.

So the optimization problem for finding the adversary bounds can be formulated as following:

maximize $\sum_{i\in\{1,\dots,16\}} \gamma_i$
subject to constraint that $I - \Gamma \circ D_1$ *and* $I + \Gamma \circ D_1$ *are positive semidefinite, and* $\gamma_i \geq 0$

Let's try to simplify the problem. First of all because of its symmetric structure, if $\Gamma \circ D_1$ has an eigenvalue λ, then it also has an eigenvalue $-\lambda$. This means that we don't need both conditions $I - \Gamma \circ D_1$ and $I + \Gamma \circ D_1$. It's enough to have just one of them. If one of them is true, then the other one is also true.

As a second simplification we can reduce the number of distinct entries by proving that some of γ_i are equal. We can prove that there is an automorphism which can permute bits in a code vector in such

a way that we can get all code vectors of the same weight just by applying a permutation to one of them.

We can permute parity bits and according data bits of a valid codeword and still get another valid codeword. If we permute two parity bits then we also need to permute those data bits which are checked by the two parity bits. This gives us the new automorphism group with the following permutations:

$$abucvwx \to baucwvx$$
$$abucvwx \to cbwavux$$
$$abucvwx \to acvbuwx$$
$$abucvwx \to cavbwux$$
$$abucvwx \to bcwauvx$$

It's just a matter of checking case by case to see that with these automorphisms it's possible to get any not valid code vector of the same size from any other not valid vector of the same size. The same is true for valid codewords, but we already knew it before as the code is cyclic. This property means that if hammingWeight(i) = hammingWeight(j) then $\Gamma[0, i] = \Gamma[0, j]$. Since there are only 6 distinct weights of code vectors, we can reduce number of distinct entries of Γ to six.

Now the optimization problem is reduced to the following:

maximize $\gamma_1 + 3\gamma_2 + 4\gamma_3 + 4\gamma_4 + 3\gamma_5 + \gamma_6$

subject to constraint that $I - \Gamma \circ D_1$ *is positive semidefinite, and* $\gamma_i \geq 0$

For the negative adversary the constraint for entries to be not negative is removed:

maximize $\gamma_1 + 3\gamma_2 + 4\gamma_3 + 4\gamma_4 + 3\gamma_5 + \gamma_6$
subject to constraint that $I - \Gamma \circ D_1$ *is positive semidefinite*

We will find the solutions for the both problems by using Matlab together with Sedumi package. The program's source code is attached to the thesis in appendix A.

We get the following results for standard adversary:

$$\gamma_1 = 2/\sqrt{10}$$
$$\gamma_2 = 1/\sqrt{10}$$
$$\gamma_3 = \gamma_4 = \gamma_5 = \gamma_6 = 0$$

The norm of $\Gamma \circ D_1$ is equal to 1 when applying the calculated values.

We can find the optimal standard adversary by using the calculated values:

$$\begin{aligned} ADV(Hamming(7)) &= (\gamma_1 + 3\gamma_2 + 4\gamma_3 + 4\gamma_4 + 3\gamma_5 + \gamma_6)\sqrt{7} \\ &= (2/\sqrt{10} + 3/\sqrt{10})\sqrt{7} \\ &= (5/\sqrt{10})\sqrt{7} \\ &\approx 4.18... \end{aligned}$$

This gives the lower bound of quantum exact complexity of $Q_E \geq 2.09$

The values for negative adversary are also calculated, they are not

in so nice form though:

$$\begin{aligned}
\gamma_1 &\approx 0.6412 \\
\gamma_2 &\approx -0.0042 \\
\gamma_3 &\approx -0.0283 \\
\gamma_4 &\approx 0.2572 \\
\gamma_5 &\approx 0.0902 \\
\gamma_6 &\approx -0.0287
\end{aligned}$$

The optimal negative adversary is a little bit bigger than the standard one:

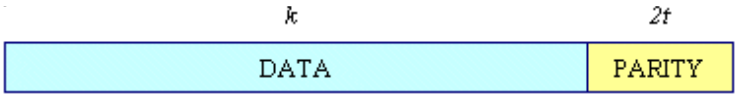
$$\begin{aligned}
ADV^{\pm}(Hamming(7)) &= (\gamma_1 + 3\gamma_2 + 4\gamma_3 + 4\gamma_4 + 3\gamma_5 + \gamma_6)\sqrt{7} \\
&\approx (0.6412 - 4 * 0.0042 - 3 * 0.0283 + 3 * 0.2572 \\
&\qquad + 4 * 0.0902 - 0.0287)\sqrt{7} \\
&= 1.6432\sqrt{7} \\
&\approx 4.3475
\end{aligned}$$

It means that $Q_E \geq 2.17$.

Chapter 4

Reed-Solomon Codes

Reed-Solomon codes were introduced in [RS 60]. These codes are being used widely in communication, storage devices (CD, DVD) etc. Reed-Solomon codes are a subset of BCH codes and are linear block codes [BC 60, Ho 59]. A Reed-Solomon code is specified as $RS(l, k)$ with n-bit symbols. Here $l = 2^n - 1$. $RS(l, k)$ codeword consists of k data symbols and $l - k$ parity symbols (all symbols consist of n bits):



A Reed-Solomon decoder can correct up to t erroneous symbols per codeword, where $2t = l - k$.

The simplest example of such code is the $RS(7, 5)$ code. It has $l = 7, k = 5, n = 3, t = 1$.

In the thesis we will investigate functions based on *Reed-Solomon codes correcting one error*. It means that $t = 1$.

4.1 Constructing a codeword

In this section we will learn how to construct Reed-Solomon codewords. Assuming we have data to encode, how do we calculate parity symbols?

The short answer is that parity symbols are obtained by getting remainder of dividing data symbols with the generating polynomial of Galois field $GF(2^n)$. GF arithmetics is used for dividing. Let's go ahead and learn it in more details.

Each symbol in the code is considered as an element from Galois Field. Any word $A_1A_2A_3A_4 \dots A_n$ can be expressed as a polynomial $A_1X^{n-1}A_2X^{n-2} \dots A_n$, where X_i are just formal multipliers to denote different types of variables which cannot be added to one another. Polynomial operations, such as addition, multiplication and deletion happen the same way as in case of normal polynomials, the only difference is that coefficients are added and multiplied following the Galois Fields rules.

If we have word D and we want to get a codeword in Reed-Solomon $RS(n, k)$ encoding then we will do the following:

1. Add $r = n - k$ zeros to the word D from the right side. In the terms of polynomials, we are multiplying the polynomial by X^r. So as the result of this step we get a new polynomial DX^r.

2. Divide the polynomial DX^r by generating polynomial G and get a remainder R, such that $DX^r = GQ + R$, where Q is a quotient which we will ignore as we are interested only in the remainder.

3. Add remainder R to the initial word D. As a result we get a codeword C which data symbols are stored separately from parity symbols R.

Power of primitive element	Polynomial	Binary represent.	Decimal represent.	Minimal polynomial
0	0	000	0	
α^0	1	001	1	$x+1$
α^1	z	010	2	x^3+x+1
α^2	z^2	100	4	x^3+x+1
α^3	$z+1$	011	3	x^3+x^2+1
α^4	z^2+z	110	6	x^3+x+1
α^5	z^2+z+1	111	7	x^3+x+1
α^6	z^2+1	101	5	x^3+x^2+1

Table 4.1: The summary of GF(8).

The Galois Fields arithmetics is explained in section 2.6.

Let's construct the field $GF(2^3)$ and use it for getting sample codeword of code $RS(7,5)$.

4.1.1 The Field $GF(2^3)$

The field $GF(2^3)$ is a superfield of GF(2), which means that it's characteristic is two and for each β it's true that $-\beta = \beta$. We will use the primitive polynomial of degree three from table 2.4: x^3+x+1. We will use x as a primitive element.

In [Bl 83] there is a very convenient way of representing the properties of Galois fields . We will use the same representation here. It is shown in table 4.1

The table is constructed in the following way. We get polynomials by raising z to respective power by modulus z^3+z+1. Binary representation is obtained from the polynomials as a concatenation of its coefficients. Decimal representation is just an equivalent of binary representation in decimal system. Finally, minimal polynomial is taken in

such a way that by applying it to according polynomial its value is 0 (by modulus $z^3 + z + 1$).

For instance element $z+1$ has a minimal polynomial x^3+x^2+1. We get the following result when evaluating the polynomial by substituting x with $z + 1$. The calculation done is by modulus $z^3 + z + 1$:

$$\begin{aligned}(z+1)^3 + (z+1)^2 + 1 &= (z+1)(z^2+1) + (z^2+1) + 1 \\ &= z^3 + z^2 + z + 1 + z^2 + 1 + 1 \\ &= z^3 + z + 1 = 0\end{aligned}$$

When having such a representation, the multiplication and deletion can be done as easy as:

$$\alpha^i \alpha^j = \alpha i + j (mod 7)$$
$$\alpha^i / \alpha^j = \alpha i - j (mod 7)$$

Addition is equivalent to XOR.

Multiplication can also be represented as a vector multiplication by matrix ([Ma 89]). The list of multiplication matrices is shown in table 4.2.

For instance multiplication $011 \cdot 101$ can be performed in the following way:

$$\begin{bmatrix}100\end{bmatrix} \begin{bmatrix}0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 1\end{bmatrix} = \begin{bmatrix}100\end{bmatrix}$$

$\cdot\,[000]$	$\cdot\,[001]\ (\alpha^0)$	$\cdot\,[010]\ (\alpha^1)$	$\cdot\,[011]\ (\alpha^3)$
$\begin{bmatrix}0&0&0\\0&0&0\\0&0&0\end{bmatrix}$	$\begin{bmatrix}1&0&0\\0&1&0\\0&0&1\end{bmatrix}$	$\begin{bmatrix}0&1&1\\1&0&0\\0&1&0\end{bmatrix}$	$\begin{bmatrix}1&1&1\\1&1&0\\0&1&1\end{bmatrix}$
$\cdot\,[100]\ (\alpha^2)$	$\cdot\,[101]\ (\alpha^6)$	$\cdot\,[110]\ (\alpha^4)$	$\cdot\,[111]\ (\alpha^5)$
$\begin{bmatrix}1&1&0\\0&1&1\\1&0&0\end{bmatrix}$	$\begin{bmatrix}0&1&0\\0&0&1\\1&0&1\end{bmatrix}$	$\begin{bmatrix}1&0&1\\1&1&1\\1&1&0\end{bmatrix}$	$\begin{bmatrix}0&0&1\\1&0&1\\1&1&1\end{bmatrix}$

Table 4.2: Multiplication matrices of GF(8)

4.1.2 Sample Codewords

Before continuing further investigation, let's consider two examples of Reed-Solomon codewords. The first example will be for the code $RS(7,5)$, the second - for the code $RS(7,3)$.

Let's take a data to be 12345. Now we will find the codeword for $RS(7,5)$ code. To get the parity symbols we need to divide the data symbols by the generating polynomial of GF(8). The generating polynomial of GF(8) is $x^2 + 6x + 3$. The codeword will be in format $12345XX$, where XX is the remainder of diving 1234500 by 163. We will use the long division procedure to find the remainder. For easier application of the division we will use the addition and multiplication tables of GF(8) (see tables 4.3 and 4.4 accordingly).

```
1 2 3 4 5 0 0 |1 6 3
1 6 3         |1 4 5 0 1
  4 0 4
```

+	0	1	2	3	4	5	6	7
0	0	1	2	3	4	5	6	7
1	1	0	3	2	4	5	7	6
2	2	3	0	1	6	7	4	5
3	3	2	1	0	7	6	5	4
4	4	5	6	7	0	1	2	3
5	5	4	7	6	1	0	3	2
6	6	7	4	5	2	3	0	1
7	7	6	5	4	3	2	1	0

Table 4.3: The addition table of GF(8).

·	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7
2	0	2	4	6	3	1	7	5
3	0	3	6	5	7	4	1	2
4	0	4	3	7	6	2	5	1
5	0	5	1	4	2	7	3	6
6	0	6	7	1	5	3	2	4
7	0	7	5	2	1	6	4	3

Table 4.4: The multiplication table of GF(8).

```
4 5 7
  5 3 5
  5 3 4
    0 1 0 0
      1 6 3
        6 3
```

We got the remainder 63. So the codeword is 1234563.

Now let's take a word 123 and find a codeword for the $RS(7,3)$ code. We do it by getting the remainder of dividing 1230000 by 13123:

```
1 2 3 0 0 0 0 |1 3 1 2 3
1 3 1 2 3     |1 1 1
  1 2 2 3 0
  1 3 1 2 3
    1 3 1 3 0
    1 3 1 2 3
          1 3
```

We got the remainder 13 which means that the codeword is 1230013.

4.2 Deterministic Query Algorithm for Reed-Solomon Codes

In this and the following sections we denote $N = n(2^n - 1)$.

We define a Boolean function $f(b_1, \ldots, b_N)$ equal to 1 on Reed-Solomon code vectors and equal to 0 otherwise.

Theorem 4.1. $D(f) = N$.

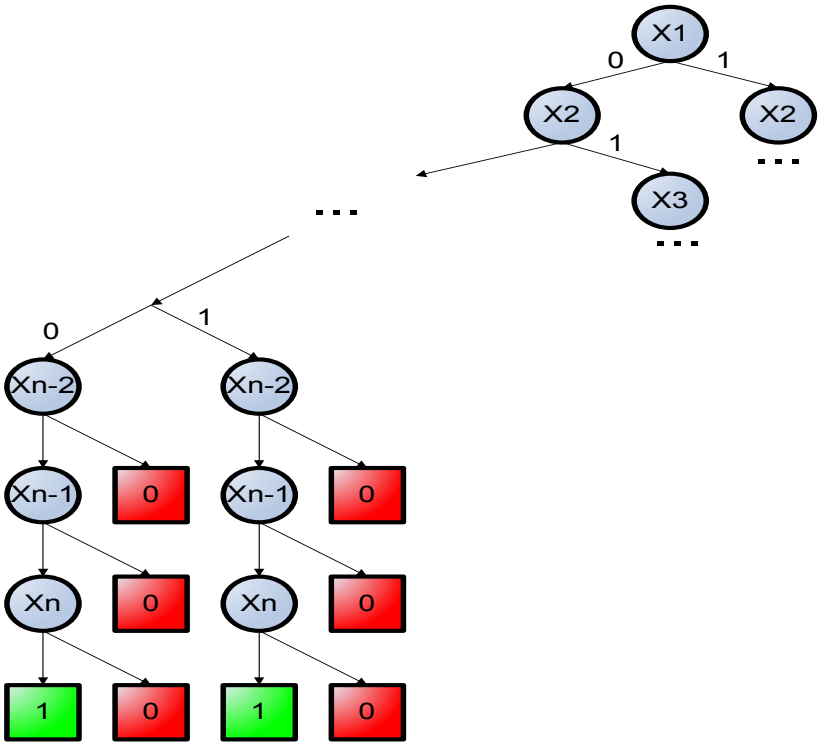


Figure 4.1: A deterministic query algorithm for the Reed-Solomon code function $RS(7, 5)$ of 21 arguments.

Proof. The proof is identical to the proof of Theorem 3.2. □

The visualization of the deterministic algorithm for the function of 21 arguments can be seen on figure 4.1

4.3 Quantum Query Algorithm for Reed-Solomon Codes

Lemma 4.2. *Reed-Solomon code can be verified multiplying the input variable vector* $[x_1 \dots x_N]$ *by matrix* Λ *of* N *rows and* n *columns. The code is correct if the multiplication's result is equal to zero-vector.*

Proof. We can find out from [Pe 60], that syndrome can be used to verify Reed-Solomon code. This syndrome's formula is: $S = X_1\alpha^{2^n-2} + \ldots + X_{2^n-1}\alpha^0$ where X_i are codeword's n-bit symbols, α is GF's primitive element.

The code is correct if $S = 0$.

To prove the lemma we can represent the syndrome as a multiplication of matrices:

$$\begin{bmatrix} X_1 & \ldots & X_{2^n-1} \end{bmatrix} \times \begin{bmatrix} \alpha^{2^n-2} \\ \vdots \\ \alpha^0 \end{bmatrix}$$

Here we can replace X_i with its binary representation. It is known from [Ma 89] that multiplication in $GF(2^n)$ can be represented as a multiplication with matrix (See table 4.2 for the list of multiplication matrices in $GF(8)$). So we can also replace α^i with multiplication matrix $n \times n$ for this α^i. As a result we get matrix Λ which consists of multiplication matrices for each element of GF. Here is the multiplication we obtain after all replacements:

$$\begin{bmatrix} x_1 \ldots x_n & \ldots & x_{N-n+1} \ldots x_N \end{bmatrix} \times \begin{bmatrix} \lambda_N \\ \vdots \\ \lambda_{N-n+1} \\ \vdots \\ \lambda_n \\ \vdots \\ \lambda_1 \end{bmatrix}$$

If a result of this multiplication is zero-vector then the syndrome $S = 0$ and so $[x_1 \ldots x_N]$ is a correct Reed-Solomon codeword. □

Lemma 4.3. *The rows of the matrix Λ are all possible n-bit vectors except zero-vector, moreover each vector occurs exactly n times.*

Proof. If we take the rows with indices $1, n+1, 2n+1, \ldots, N-n+1$ of the matrix Λ we will meet between them each vector from [00...01] to [11...11]. It is implied by the fact that if we multiply [10...00] with all elements of GF, we should get the same elements exactly once as a result. The same is true for each row sequence $i, n+i, \ldots, N-n+i$ where $i \in [n]$ □

For example, let's take Reed-Solomon code $RS(7,5)$. Here $n = 3$, $l = 7$ and $k = 5$. Each codeword consist of $l \times s = 7 \times 3 = 21$ bits (function f is of 21 variables). Matrix Λ will be of 21 rows and 3 columns. Rows of the matrix are all possible 3-bit vectors except zero-vector, and each vector occurs exactly 3 times:

$$\Lambda = \begin{bmatrix} 0 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}^T$$

Theorem 4.4. *There is an exact quantum query algorithm for function f with complexity $N/2$ for even n.*

Proof. We will use the method described in Lemma 4.2 and Lemma 4.3 for constructing the algorithm. We have a codeword and the matrix Λ for Reed-Solomon code of N variables. First we take multiplication of matrices

$$[x_1 \ldots x_N] \times \begin{bmatrix} \lambda_N \\ \vdots \\ \lambda_1 \end{bmatrix},$$

then we permute rows of the matrix Λ (and accordingly the codeword's bits x_i) in the following way (it's possible because of Lemma 4.3):

$$\begin{bmatrix} 0\ldots01 \\ \vdots \\ 0\ldots01 \\ \vdots \\ 1\ldots11 \\ \vdots \\ 1\ldots11 \end{bmatrix} \begin{matrix} \\ \left.\begin{matrix} \\ \\ \end{matrix}\right\} n \\ \\ \left.\begin{matrix} \\ \\ \end{matrix}\right\} n \end{matrix}$$

Now the task is to multiply the matrices and to check if we have zero-vector as a result. It is equivalent to verifying the following checksums:

$$\begin{aligned} y_1 &= \textstyle\sum_{\lambda_i[1]=1} x_i \\ &\vdots \\ y_n &= \textstyle\sum_{\lambda_i[n]=1} x_i \end{aligned}$$

$\lambda_i[j]$ denotes the j-th element in vector λ_i. The sums are calculated modulo 2.

If each y_i is equal to 0, then the result of the multiplication is zero-vector.

Let's evaluate the complexity (i.e. the number of queries) for even n.

For any i, we take input variables x_{2i-1} and x_{2i} and calculate $x_{2i-1} \oplus x_{2i}$. We use this result in those checksums where this pair occurs. It is easy to see that if one variable of the pair belongs to some checksum then also the other variable belongs to the same checksum. So we need

$N/2$ queries to evaluate the function.

□

Theorem 4.5. *There is an exact quantum query algorithm for function f with complexity $n \cdot (2^n - 1)/2 + (2^{n-1} - 1)/2 - 2^{n-2}$ for odd n.*

Proof. In the case of odd n we begin like we did when n was even. So we represent the task as multiplication of matrices and then permute rows of a matrix Λ in the following way:

$$\left[\begin{array}{l} \left.\begin{array}{l} 0\ldots01 \\ \vdots \\ 0\ldots01 \end{array}\right\} n-1 \\ \vdots \\ \left.\begin{array}{l} 1\ldots11 \\ \vdots \\ 1\ldots11 \end{array}\right\} n-1 \end{array}\right\} A$$
$$\left[\begin{array}{l} 0\ldots01 \\ \vdots \\ 1\ldots11 \end{array}\right\} B$$

We use the algorithm from the previous theorem to calculate those parts of the checksums which contain the first $(n-1)(2^n-1)$ variables (see part A above) of the codeword. The remaining variables represent subsums equivalent to the Hamming code checksums (see part B above) which can be calculated with $2^n - 1 - 2^{n-2}$ queries. In total we have $(n-1)(2^n-1)/2 + 2^n - 1 - 2^{n-2} = n \cdot (2^n-1)/2 + (2^{n-1}-1)/2 - 2^{n-2}$ queries for odd n.

□

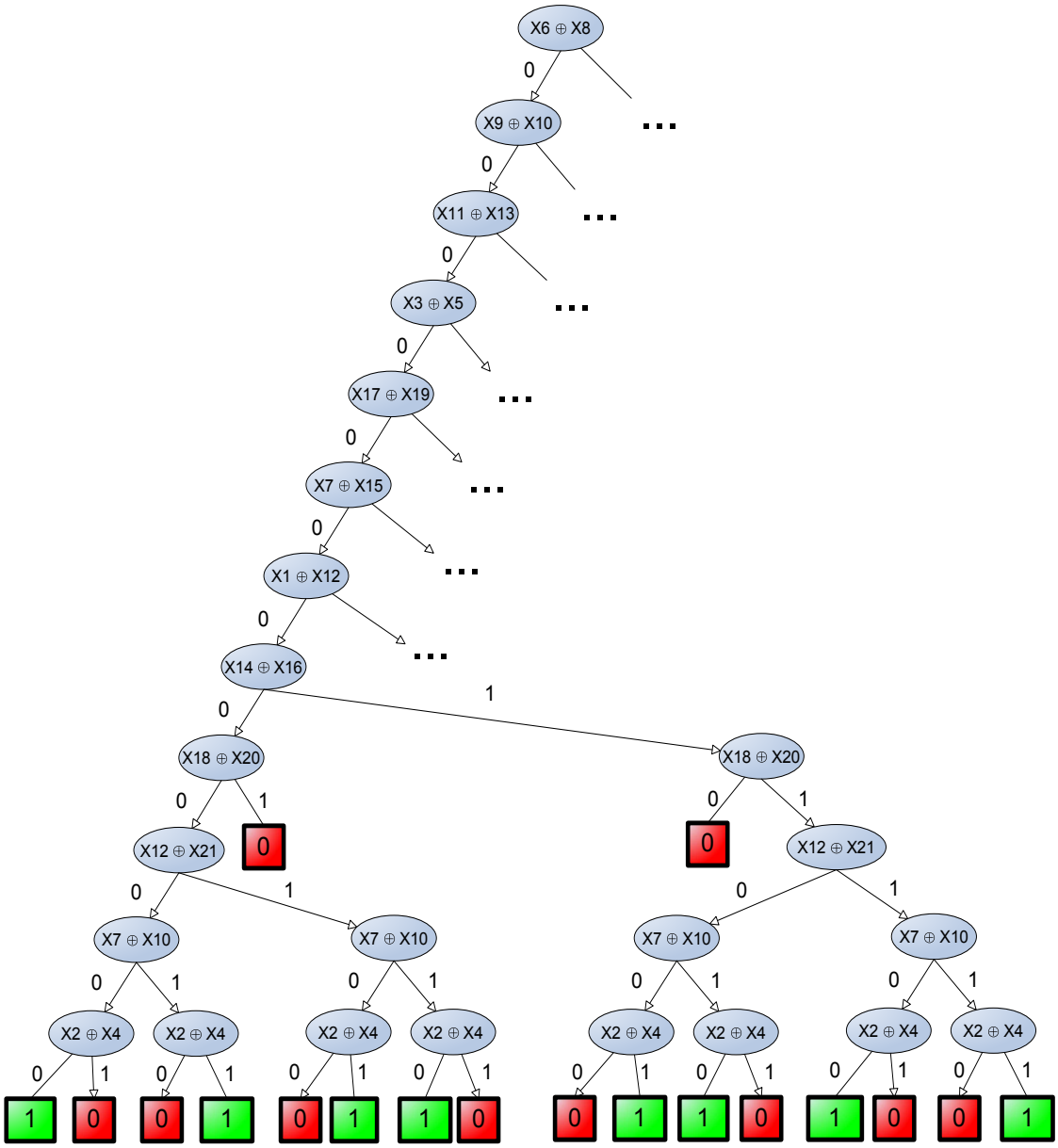


Figure 4.2: A quantum query algorithm for the Reed-Solomon code function $RS(7, 5)$ of 21 arguments.

Figure 4.2 visualizes the quantum algorithm for the Reed-Solomon code of 21 arguments.

4.4 Lower Bound by Polynomials

We will begin with finding the degree of f which helps us to prove the lower bound by polynomials.

Theorem 4.6. $deg(f) = n \cdot 2^{n-1}$.

Proof. We prove this fact using Dirichlet principle and by constructing Reed-Solomon code from several Hamming codes. It is similar to proof for degree of Hamming code function. It is based on linearity of Reed-Solomon codes.

Similarly as in the proof of Theorem 3.4 we will express the polynomial of f as $p(x_1, \ldots, x_N) = \sum_{b:f(b)=1} \prod_{i:b_i=1} x_i \prod_{i:b_i=0} (1 - x_i)$. In this formula each summand corresponds to a particular Reed-Solomon code vector b.

Similarly as for the Hamming code proof we open all the parentheses $(1-x_i)$ to get sum of monomials and investigate in which summands a particular monomial occurs. First we need to prove that for $I \subseteq [N]$ the same monomial $\prod_{i \in I} x_i$ of size more than $n \cdot 2^{n-1}$ appears the same number of times with positive sign as with negative sign. This reduces to the prove that among the Reed-Solomon codes with 0's in all the positions not from I, the number of codes with even Hamming weight coincides with the number of codes with odd Hamming weight. Reed-Solomon codes are also linear, so similarly like we did for Hamming codes it is enough to prove that among the above codewords there is a code c with Hamming weight 3.

We proved in Lemma 4.3 that rows of the matrix Λ are all possible n-bit vectors except zero-vector and each vector occurs exactly n times. This means that we can split all these rows into n sets of size n in such a way that each set contains all possible n-bit vectors. According to Lemma 3.1 each of the sets represents a checking matrix for verifying a Hamming code.

We can divide all the arguments of the Reed-Solomon code into groups which correspond to the Hamming code checking matrices. We denote this set of groups by G. Since there are only n distinct groups and $I > n \cdot 2^{n-1}$ then by Dirichlet principle one of the groups will have at least $2^{n-1} + 1$ representatives in I. We already proved in Theorem 3.4 that there is a codeword with Hamming weight 3 in such a group.

Now we similarly as in Theorem 3.4 need to prove that there is a monomial with $card(I) = n \cdot 2^{n-1}$ and non-zero coefficient. We use the same grouping G as above. In each of the groups $g \in G$ we can find such set of indices J_g of size 2^{n-1}, where if h is a valid Hamming code then $\oplus_{j \in J} h_j = 0$. Now we make the set $I = \cup_{g \in G} J_g$. The number of 0's and 1's in positions from I is even because they need to satisfy the checksum $\oplus_{i \in I} b_j = 0$ for b to be a valid Reed-Solomon code and because $card(I) = n \cdot 2^{n-1}$ is an even number. Monomial corresponding to this set of indices appears only with coefficients +1 and thus cannot cancel (and it occurs at least once - for the all-zero Reed-Solomon code).

□

Corollary 4.7. *The lower bound by polynomials for Reed-Solomon code function is:* $Q_E(f) \geq n \cdot 2^{n-2}$.

Chapter 5

Conclusion

In the conclusion we will summarize all the items which were discussed in the thesis.

We investigated possibilities to implement quantum algorithms for the Hamming and Reed-Solomon code functions. We investigated complexity of the algorithms and their possible implementations.

Several results were achieved in this area. I think the most important result is that we found a better quantum algorithm for evaluating Hamming code functions. The algorithm reduces number of queries by 25% compared to the best possible classical algorithm. It's not the world best achievement for reducing classical complexity by an exact quantum algorithm. The best known achievement is an algorithm for XOR (Deutsch's) function which reduces the number of queries by one half. From the other hand there is not so many algorithms known which achieve improvement similar to what we got for the Hamming codes functions.

For the Reed-Solomon functions we were able to reduce queries count by one half. This is the same improvement as for the XOR. I

think that the main value of the investigation is in showing how to reduce Reed-Solomon function to use XOR as a subroutine.

We have found a polynomial lower bound for the both function classes. The bound is not tight, but it is tight enough to see that the classical deterministic algorithms can be improved only by a constant factor.

We also found a lower bound by spectral adversary methods for Hamming code function of seven arguments. Positive adversary gave us lower bound of 2.09 and negative adversary increased it to 2.17. Both these numbers mean that at least three queries are required for an exact algorithm to evaluate them. This lower bound is higher than the polynomial one for the same function.

Obviously there is an additional work possible in this area. The immediate desire is to apply adversary method to Hamming code functions of more arguments and also for Reed-Solomon code functions.

Some investigations still possible to minimize the gap between lower bounds and algorithms complexity.

There is a related work done for other error correcting codes in [Va 09]. It may be of interest to find a lower bound for the functions in that work by applying the techniques from this research.

Another activity could be to extend this approach to a broader class of error correcting codes and define which of them can be reduced to using XOR as a subroutine.

All these activities may lead either to improvement of lower bound estimations or even provide some idea for implementing more efficient exact quantum algorithms.

Bibliography

[AF 03] A. Ambainis, R. Freivalds. Boolean function with a low polynomial degree. *in Proc. of the Latvian Academy of sciences*, vol. 57, pp. 74–77, 2003.

[Am 02] A. Ambainis. Quantum lower bounds by quantum arguments. *Journal of Computer and System Sciences*, 64:750–767, 2001.

[Am 03] A. Ambainis. Polynomial degree vs. quantum query complexity. *in Proc. of the 44th IEEE FOCS*, 2003.

[Am 04] A. Ambainis. Quantum query algorithms and lower bounds (survey article). *Proceedings of FOTFS III, Trends on Logic*, vol. 23, pp. 15–32, 2004.

[Be 93] R. Beigel. The polynomial method in circuit complexity. *in Proc. of the 8th Annual Structure in Complexity Theory Conference, IEEE Computer Society Press*, pp. 82–95, 1993.

[BB$^+$97] C. H. Bennett, E. Bernstein, G. Brassard, and U. Vazirani. Strengths and weaknesses of quantum computing. *SIAM Journal on Computing 26*, pp. 1510–1523, 1997.

[BB+01] R. Beals, H. Buhrman, R. Cleve, M. Mosca, and R. de Wolf. Quantum lower bounds by polynomials. *Journal of the ACM*, 48(4):778–797, 2001.

[Bl 83] R. Blahut. *Theory and Practice of Error Control Codes*. Mass.: Addison-Wesley, 1983.

[BC 60] R. C. Bose, D. K. Ray-Chaudhuri On a class of error correcting binary group codes. *Inf. and Contr.*, v.3, p. 68–79, 1960.

[BSS 03] H. Barnum, M. Saks, and M. Szegedy. Quantum query complexity and semi-definite programming. *in Proc. of the 18th IEEE Conference on Computational Complexity*, pp. 179–193, 2003.

[BW 02] H. Buhrman, R. de Wolf. Complexity measures and decision tree complexity: a survey. *Theoretical Computer Science*, 288:21–43, 2002.

[CL 08] A. M. Childs and T. Lee. Optimal Quantum Adversary Lower Bounds for Ordered Search. *Lecture Notes in Computer Sciense*, v.5125, pp. 869-880, 2008.

[CE+98] R. Cleve, A. Ekert, C. Macchiavello, M. Mosca. Quantum Algorithms Revisited. *in proc. of the Royal Society, London*, A454:339–354, 1998.

[CLR 90] T. Cormen, C. Leiserson, R. Rivest. Introduction to Algorithms. *MIT Press*, 1990

[De 85] D. Deutsch. Quantum theory, the Church-Turing principle and the universal quantum computer. *in proc. of the Royal Society, London*, A400:97–117, 1985.

[Gr 96] L. K. Grover. A fast quantum mechanical algorithm for database search. *In Proc. of 28th ACM STOC*, pp. 212–219, 1996.

[Gr 99] J. Gruska. Quantum Computing, *McGraw Hill*, 439 p, 1999

[Ha 50] R. W. Hamming. Error detection and error correcting codes. *Bell System Technical Journal*, 26(2):147–160, 1950.

[Ho 59] A. Hochquenghem. Codes correcteurs d'erreurs. *Chiffres*, t.2., p. 147–156, 1959.

[HLS 07] P. Hoyer, T. Lee, R. Spalek. Negative weights make adversaries stronger. *in Proc. of 39th ACM Symposium on Theory of Computing*, 2007.

[LM 04] S. Laplante and F. Magniez. Lower bounds for randomized and quantum query complexity using Kolmogorov arguments. *in Proc. of 19th IEEE Conference on Computational Complexity*, pp. 294–304, 2004.

[Ma 89] E. D. Mastrovito. VLSI Designs for Multiplication over Finite Fields $GF(2^m)$ *Lecture Notes in Computer Science* 357, p. 297–309, 1989.

[NC 00] M. Nielsen, I. Chuang. Quantum Computation and Quantum Information. *Cambridge University Press*, 675 p.,2000.

[NS 94] N. Nisan, M. Szegedy. On the degree of Boolean functions as real polynomials. *Computational Complexity*, 4, pp. 301–313, 1994.

[NW 95] N. Nisan, A. Wigdersonn. On rank vs. Communication complexity. *Combinatorica*, 15, p. 557–565, 1995.

[Pa 94] C. Papadimitriou. Computational Complexity. *Addison-Wesley, Reading*, 500pp., 1994.

[Pe 60] W. W. Peterson. Encoding and error-correction procedures for the Bose-Chaudhuri codes. *IEEE Trans. Inf. Theor.* v.IT-6, p.459–470, 1960.

[Re 97] K. Regan. Polynomials and combinatorial definitions of languages. *In Complexity Theory Retrospective II, Springer-Verlag*, pp 261–293, 1997.

[RS 60] I. S. Reed,G. Solomon. Polynomial codes over certain finite fields. *J. Soc. Indust. Appl. Math.*, v.8, p.300–304, 1960.

[Sh 97] P. W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Journal of Computing*, 26(5):1484–1509, 1997.

[SS 05] R. Spalek, M. Szegedy. All quantum adversary methods are equivalent. *In Proceedings of 32nd International Colloquium on Automata, Languages and Programming , Lecture Notes in Computer Science* 3580, pp 1299–1311, 2005.

[Va 09] A. Vasilieva. Exact Quantum Query Algorithm for Error Detection Code Verification. *in Proc. of MEMICS,* 2009.

[Zh 04] S. Zhang. On the power of Ambainis's lower bounds. *in Proc. of 31st International Colloquium on Automata, Languages and Programming, Lecture Notes in Computer Science* 3142, pp 1238–1250, 2004.

Appendix A

The Matlab Program

Following is the source code of a Matlab program calculating the optimal adversary lower bound of a Hamming code function. The program uses sedumi package for solving semidefinite programming problems. Visit
`http://sedumi.ie.lehigh.edu` for more information about the package.

```
function x = hammingAdversary(N, positive)

% N - length of the code vectors
% positive - if true then the positive adversary will be returned,
% otherwise will return the negative adversary

% count of variables to be optimized.
% We have one variable for each set of code vectors of the
% same size.
% This is based on conjecture that all not valid code vectors
% of the same size
```

```
% can be get from one another by a permutation of bits.
% This conjecture is proved for the code of size seven
variablesCount = N-1;

% entry (of not reversed) D(i,j) is equal to entry
% Gamma(i,j+indent);
indent = 2^(N-1);

% columns count in D
colsCount = 2^(N-1);



% supporting structure for constructing input of sedumi
rows = [];
cols = [];
s = [];
b = [];

% by D we denote Gamma * D1, where "*" is entrywise
% multiplication
D = zeros(N,N);
for i=1:colsCount
  for j=1:colsCount
    iG = i; %row index in Gamma
    jG = (indent + j); %column index in Gamma
    if (isHamming(iG-1) == isHamming(jG-1))
      % reversing columns order of D, we need it to have
      % all zeros on the diagonal
```

```
      % this operation doesn't change the norm
      D(i,colsCount-j+1) = 0;
    else
      codeVector = bitxor(j-1,(indent + i)-1);
      dec2bin(codeVector);
      % reversing columns order of D, we need it to have
      % all zeros on the diagonal
      % this operation doesn't change the norm
      D(i,colsCount-j+1) = sum(dec2bin(codeVector) == '1');
    end
  end
end

% I-D>0 - the constraint that I-D is positive semidefinite
for i=1:colsCount
  for j=1:colsCount
    if (D(i,j)==0)
      rows = [rows (i-1)*colsCount+j];
      cols = [cols (i-1)*colsCount+j+variablesCount];
      s = [s 1];
      if (i == j)
        b = [b 1];
      else
        b = [b 0];
      end
    else
      rows = [rows (i-1)*colsCount+j (i-1)*colsCount+j];
      cols = [cols D(i,j) (i-1)*colsCount+j+variablesCount];
```

```
      s = [s 1 1];
      b = [b 0];
    end
  end
end

A=sparse(rows, cols, s);

weightsDistribution = wordsDistributionByWeight(N);

c = -1*ones(1,variablesCount);
for i=1:variablesCount
  % optimizing the sum of free variables
  c(i) = -1 * (nchoosek(N,i)-weightsDistribution(i+1))/N;
end

c = [c zeros(1,colsCount*colsCount)];

if(positive)
  K.l = variablesCount;
else
  K.f = variablesCount;
end
K.s = colsCount;

x = sedumi(A, b, c, K);
```

The function for detecting valid hamming codewords. The code words are reversed, but it doesn't have impact on the result.

```
function x = isHamming(codeVector)

codeVectorLength = length(dec2bin(codeVector));
result = 0;
for i=1:codeVectorLength
  if (bitget(codeVector,i) == 1)
    result = bitxor(result,i);
  end
end
x = (result == 0);
```

The function for determining valid codewords distribution by weight.

```
function x = wordsDistributionByWeight(N)

result = zeros(1,N+1);
totalCodeVectors = 2^N;
for i=0:totalCodeVectors-1
  if(isHamming(i))
    weight = sum(dec2bin(i) == '1');
    result(weight+1) = result(weight+1) + 1;
  end
end

x = result;
```