Efficient quantum circuits for classical decision problems

Minimizing the number of Toffoli gates

We go over the basics of quantum computing and error correcting codes; we focus our attention on classical computations and show that these can be implemented on quantum computers using only NOT, CNOT and *Toffoli* gates, with the latter being particularly challenging to implement in a fault tolerant manner. Consequently, we set the goal of minimizing the number of *Toffoli* gates required to implement a classical decision problem. We leverage general properties of boolean functions to establish upper bounds and develop a greedy algorithm that manipulates algebraic expressions representing the targeted decision problem, with the same goal.

Bachelor's Degree
Department of Physics
University of Trento
A.Y. 2023-2024

Supervisor:
Alessandro Roggero

Candidate:
Davide Zanin

Contents

CHAPTER 1

Introduction & relevance

1. Basics of quantum computing

1.1 Qubits

The smallest unit of information in quantum computing is the *qubit*. This term refers to the state of a physical two-level system. The two basis states are denoted by $|0\rangle$ and $|1\rangle$ and together constitute the *computational basis*. The typical realizations of a qubit are the spin state of an electron and the polarization of a photon. The basis states $|0\rangle$ and $|1\rangle$ are mathematically represented by vectors in a complex vector space equipped with an hermitian inner product, denoted by $\langle \cdot | \cdot \rangle$. Such a vector space is called a *Hilbert space* and with respect to its inner product the *computational basis* is orthonormal.

A qubit can take more values than it's classical counter part, in fact all unitary vectors of the before mentioned Hilbert space are valid states for a qubit. In other words, given any two complex numbers a and b such that $|a|^2 + |b|^2 = 1$, the vector $a|0\rangle + b|1\rangle$ represents a valid state. This fact that unitary linear combinations of basis states are valid states is called *superposition* and is a key property of all quantum systems. Applying a common phase factor $e^{i\theta}$ to a and b changes the vector but not the physical system represented by it[5].

As most vector spaces, this space has many bases. Each way of measuring the value of the qubit is related to a specific orthonormal basis. With this postulate we can now better define $|0\rangle$ and $|1\rangle$ as the basis vectors associated with our measuring procedure. The act of measuring a qubit in the quantum state $|\Psi\rangle$ destroys the quantum state of the qubit which immediately turns into $|0\rangle$ with probability $|\langle 0|\Psi\rangle|^2$ or into $|1\rangle$ with probability $|\langle 1|\Psi\rangle|^2$. The result of the measurement is a classical bit with value 0 if the state collapsed into $|0\rangle$ and 1 if it collapsed into $|1\rangle$. This is the only *nondeterministic* part of

the theory. This effect implies that the quantum system stores more information than that we can extract by performing a measurement on it. To know part of the hidden information, one needs to be able to precisely replicate the experiment many times.

I.II More qubits

As for classical computers, a quantum device often needs more than one qubit. Luckily the math doesn't change much[5] when dealing with a *multi-qubit* system, in fact the state of n qubits is still represented by a single unitary vector. That said, the Hilbert space hosting the vectors becomes the tensor product of the Hilbert spaces where the states of every single qubit live. The dimension of the tensor product of several Hilbert spaces is the product of their dimensions, therefore an n-qubit state lives inside a 2^n-dimensional space. Given a basis for each single qubit Hilbert space, one can construct a basis for the tensor product space with all the possibile products of single qubit basis states. We will indicate the tensor product of vectors with the juxtaposition of the terms and use a common notation where $|a\rangle|b\rangle\cdots$ can be written as $|ab\dots\rangle$. For example a basis of a *two-qubit* system is formed by the vectors $|00\rangle$, $|01\rangle$, $|10\rangle$ and $|11\rangle$.

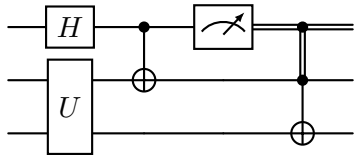
Systems of more than one qubit can exhibit a peculiar property, called *entanglement*. A state is said to be entangled when it is not possible to express it as a product of single qubit states, possibly superposed. When a system of qubits is entangled the measurement of one of them can effect the probabilities for the outcomes of the measurement of the others. For example $|00\rangle + |01\rangle$ normalized is not an entangled state because it can be factored in a pair of single qubit states $|0\rangle\,(|0\rangle + |1\rangle)$. On the contrary $|00\rangle + |11\rangle$ normalized is an entangled state because it cannot be factored in single qubit states. We can see that in this case the measurement of one qubit determines the state of both: if the measurement resulted in the classical bit 0 the quantum state must have collapsed to $|00\rangle$; if the result was 1 then the final state must be $|11\rangle$.

The kinds of manipulations one can do on qubits are vast but limited. In fact only linear unitary operators can act on a quantum state. The requirement of the operators being linear is at the core of the theory of quantum mechanics and the unitarity is needed to guarantee that state vectors keep having length one. There's no other constraint on what kind of operation can be performed on qubits.

I.III Quantum gates

A quantum computer is a device that can manipulate qubits and measure their state. Similarly to how classical computers function, the quantum computation is performed consecutively applying small operations on the qubits. The entities performing the steps are called *quantum gates* and can operate on any number of qubits. Every operation must produce an output by evaluating the action of a linear unitary operation on its input. Unitary operators satisfy the property $U^\dagger U = \mathbb{I}$, where $\mathbb{I}$ is the identity and $U^\dagger$ is the hermitian adjoint of U. As a consequence every computation performed by quantum circuits without measurement devices is *reversible*. State vectors and quantum gates can be expressed as column vectors and matrices using the coordinates induced by the basis chosen for the Hilbert space. A n-qubit state vector is represented by a column vector of 2^n complex numbers and a quantum gate operating on m qubits would be represented by a $2^m \times 2^m$ complex matrix.

A quantum circuit is often depicted using a diagram like the one you see below.



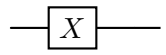
The diagram is read from left to right, following the passage of time. Every qubit is assigned to a different "wire", represented by a single horizontal line. Most boxes represent the action of a quantum gate on the intersecting qubits and the letters inside the boxes identify the kind of quantum gates applied. The boxed gauge symbol at the end of the first line represents a measuring device. The double line represents a classical bit. There are also some common gates that earned a special notation. In Focus 1 we present four gates of great relevance: the NOT, *Hadamard*, CNOT and *Toffoli* gates.

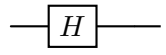
I.IV Things a quantum gate cannot do

As we said quantum gates must act on qubits as linear unitary operators. This constraint has some important consequences. To have a reversible operation it is necessary that the number of qubits entering the gate equals the number qubits exiting the gate. It is therefore

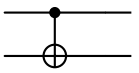
Common quantum gates

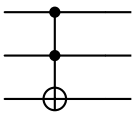
A simple *single-qubit* gate is the NOT gate. This gate maps $|0\rangle$ to $|1\rangle$ and $|1\rangle$ to $|0\rangle$. Another notable example is the *Hadamard gate* which is capable of mapping non superposed states to superposed ones.

NOT  $X \overset{\text{R}}{=} \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$

HADAMARD  $H \overset{\text{R}}{=} \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$

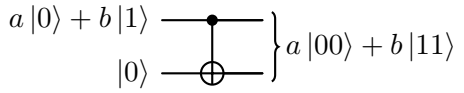
The symbol $\overset{\text{R}}{=}$ relates the operator and its coordinate representation in the canonical basis. One way of constructing a reversible *two-qubit* gate is taking any *single-qubit* gate U and turning it in its controlled version CU. When CU acts on a basis state the U gate is applied on the second qubit if and only if the first qubit is one. A controlled gate is depicted with a black dot on the control qubit, connected to the gate it is controlling. The controlled version of the NOT gate is the CNOT whose controlled version is the CCNOT, also know as *Toffoli* gate. For reasons that will become clear later, the NOT gate is depicted with an $\oplus$ symbol when controlled.

CNOT  $U_{\text{CNOT}} \overset{\text{R}}{=} \begin{bmatrix} \mathbb{I}_2 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}$

TOFFOLI (CCNOT)  $U_{\text{CCNOT}} \overset{\text{R}}{=} \begin{bmatrix} \mathbb{I}_6 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}$

Focus 1: *Some of the most common quantum gates.*

impossibile to create a quantum gate taking one qubit in input and giving two qubits in output both in the original state. To bypass this inconvenience, one might think about introducing an extra qubit in input, which is known to be prepared in the state $|0\rangle$, and apply a CNOT between the qubit to copy and this extra qubit. This indeed would map $|00\rangle$ to $|00\rangle$ and $|10\rangle$ to $|11\rangle$ but a general state $|\Psi\rangle|0\rangle = a\,|00\rangle + b\,|10\rangle$ would be sent to $a\,|00\rangle + b\,|11\rangle$, which is not only entangled but also different from the desired output $|\Psi\rangle|\Psi\rangle$.



This is a manifestation of the so called *No-cloning theorem* which we now state and prove[2].

THEOREM 1: *(Statement)* Let H be the Hilbert space where all *single qubit* states live. There is no choice of a unitary linear operator $U : H \otimes H \to H \otimes H$ and a state vector $|u\rangle \in H$ such that for all vectors $|\Psi\rangle \in H$,

$$U(|\Psi\rangle|u\rangle) = |\Psi\rangle|\Psi\rangle\,.$$

(Proof) Let's suppose by contradiction that such a choice of U and $|u\rangle$ exists. Take any two vectors $|\Psi\rangle$ and $|\Phi\rangle$ then

$$\begin{cases} U(|\Psi\rangle|u\rangle) = |\Psi\rangle|\Psi\rangle \\ U(|\Phi\rangle|u\rangle) = |\Phi\rangle|\Phi\rangle \end{cases} \quad \Rightarrow \quad \langle\Psi|\Phi\rangle = \langle\Psi|\Phi\rangle^2.$$

Here we have equated the inner product of the left terms with the inner product of the right terms, exploited the unitarity of U and used the fact that $\langle u|u\rangle = 1$. The last equation implies $\langle\Psi|\Phi\rangle \in \{0, 1\}$ which cannot be true in general. Q.E.D

Proving the no-cloning theorem told us that it is possible to build a quantum gate capable of cloning the states of a orthonormal basis but that gate won't work for any other state.

II. A CLASSICAL PROBLEM ON A QUANTUM COMPUTER

Given a system of qubits, the state vectors forming the computational basis are said to be *classical states*. For example $|0\rangle$ and $|101\rangle$ are classical states of a one and three qubits system respectively. A quantum

gate is said to be classical if its action maps classical states to classical states reversibly. This property is easily checked looking at the matrix representing the gate: a quantum gate is classical if and only if the representing matrix is obtained by permutation of the columns of the identity matrix. Referring to the gates described in Focus 1 we can verify that the NOT, CNOT and *Toffoli* gates are classical whereas the *Hadamard* gate isn't. A system of m qubits admits $2^m!$ classical gates. The only two classical gates for a *one-qubit* system are the identity and the NOT gate. The number of classical gates grows incredibly fast with the number of qubits, in fact a system of eight qubits already admits more then 10^{500} classical gates.

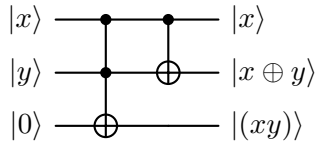


Figure 1: *One qubit half-adder.*

As a first example of a classical computation we present the *one-qubit half-adder*, whose circuit implementation is depicted in Figure 1. This circuit operates on three qubits and clearly shows how the CNOT and *Toffoli* gates work. The third qubit is an *ancilla* qubit, that is a qubit whose initial state is fixed. When this qubit is correctly prepared in the state $|0\rangle$ and the adder is fed with classical states, the second qubit is overwritten with the quantum state corresponding to the logical XOR of the first two input qubits and the third qubit is overwritten with their logical AND. The XOR binary operation between bits is denoted by the $\oplus$ symbol while the AND binary operation is denoted by juxtaposition of bits, see Table 2 for their definition.

III. Quantum error correction

III.1 Error correcting codes

As a consequence of the interaction between the environment and the physical realization of any quantum computer, noise is a significant concern. We adopt a model for noisy quantum wires: we treat them as independent channels that transmit the quantum state untouched most of the time, but occasionally act on the transported state causing it to change.

There's only one kind of error that can happen on a classical bit, namely the *bit flip*: 1 becomes 0 and 0 becomes 1. On the contrary, quantum wires can exhibit a continuous spectrum of single qubit errors, as $|0\rangle$ and $|1\rangle$ can be sent to any superposed state. Nevertheless the space of all possible single qubit operators, and therefore errors, is generated by the identity $\mathbb{I}$ together with the operators represented by the three Pauli matrices X, Y and Z.

$$\mathbb{I} \overset{\mathrm{R}}{=} \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \quad X \overset{\mathrm{R}}{=} \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \quad Y \overset{\mathrm{R}}{=} i \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} \quad Z \overset{\mathrm{R}}{=} \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$$

We shall see in a moment that this permits a system that can recover from an error of type X, Y and Z to also recover from any other kind of single qubit error $E = a\mathbb{I} + bX + cY + dZ$, if designed correctly.

As with classical devices, adding redundancy is often the key to make recovering from errors possible. Without redundancy, a recovery system cannot distinguish between a correctly transmitted state and one that suffers from errors. Still, even with redundant information, if an arbitrary number of errors can occur on the transmission lines, it's impossible to guarantee a correct decoding. We therefore proceed with the study of *error correcting codes* that protect a state when no more then one error occurs on the transmitted qubits. Let p be the probability that an error will occur on a qubit being transmitted and let n be the number of qubits the error correcting code needs to encode a single qubit state. By assumption, the probability of a recovery failure P is not greater than the probability of more than one error occurring, that is $P(p) \leq 1 - (1-p)^n - np(1-p)^{n-1}$. The *error threshold* is defined as the probability $\bar{p} \in (0, 1)$ such that $P(\bar{p}) = \bar{p}$; in other words $\bar{p}$ is the probability of error on a physical qubit below which using the error correcting code makes sense. In the case $n = 3$, we have $P \leq 3p^2 - 2p^3$ and $\bar{p} \geq 1/2$. In the case $n = 9$ we have approximately $\bar{p} \geq 1/36$[6].

Let's start from a *three-qubit* code that enables the recovery from up to one *bit-flip* error, that is an error of type X. Firstly, we encode the state $|\Psi\rangle = a\,|0\rangle + b\,|1\rangle$ with the *three-qubit* state $a\,|000\rangle + b\,|111\rangle$. Then we send all three qubits through independent noisy channels and suppose that no more than one *single qubit bit-flip* error can occur. At this point comes an error detection procedure. As showed by Table 1, the values of the observables $Z_1Z_2 := Z \otimes Z \otimes \mathbb{I}$ and $Z_2Z_3 = \mathbb{I} \otimes Z \otimes Z$ on the received state are well defined and can determine if an error occurred and on which qubit. Since $(Z_1Z_2)^2 =$

E	Received state	Z_1Z_2	Z_2Z_3	S
$\mathbb{I}$	$a\|000\rangle + b\|111\rangle$	$+1$	$+1$	00
X_1	$a\|100\rangle + b\|011\rangle$	-1	$+1$	10
X_2	$a\|010\rangle + b\|101\rangle$	-1	-1	11
X_3	$a\|001\rangle + b\|110\rangle$	$+1$	-1	01

Table 1: *Every error is uniquely determined by the values of the observables Z_1Z_2 and Z_2Z_3 on the received three-qubit state.*

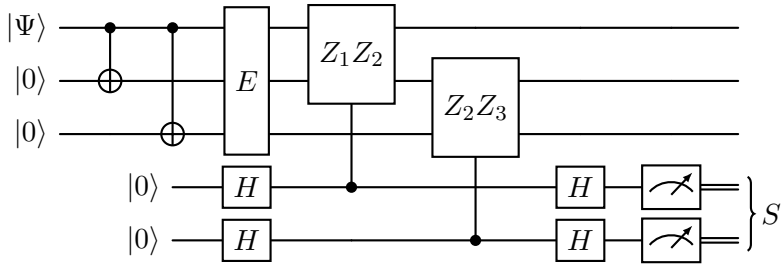


Figure 2: *Encoding and error detection in the three-qubit bit-flip code. S is a pair of classical bits containing the error syndrome.*

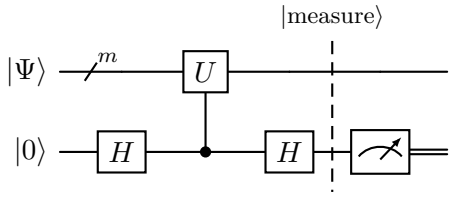
$(Z_2Z_3)^2 = \mathbb{I}$, the measurements for the error syndrome S can be performed with the procedure described in Focus 2. In this case, the received state is guaranteed to be a simultaneous eigenstate of Z_1Z_2 and Z_2Z_3, therefore the measurements don't cause a collapse of the state. Moreover, the order in which the measurements are performed doesn't matter because Z_1Z_2 and Z_2Z_3 commute. The circuit implementation up to this point is given by Figure 2. The final stage is the recovery operation: the classical information contained in the error syndrome is used to apply an X gate on the flipped qubit and recover the original state.

Now that we know how to correct for errors of type X, we should think about errors of type Z. We call these *phase-flip* errors. On closer inspection, this problem is identical to the *bit-flip* error problem. In fact, the action of X on the canonical basis $\{|0\rangle, |1\rangle\}$ is identical to the action of Z on the basis $\{|+\rangle, |-\rangle\}$.

$$|+\rangle := H|0\rangle \qquad Z|+\rangle = |-\rangle \qquad X|+\rangle = (+1)|+\rangle$$
$$|-\rangle := H|1\rangle \qquad Z|-\rangle = |+\rangle \qquad X|-\rangle = (-1)|-\rangle$$

Performing measurements

There's a general procedure to perform measurements of observables that have spectrum $\{-1, 1\}$, for example those hermitian operators that satisfy $U^2 = \mathbb{I}$.



It's easy to see that the measurement happens on the state

$$|\text{measure}\rangle = \frac{1}{2}\big(\,|\Psi\rangle + U\,|\Psi\rangle\,\big)\,|0\rangle + \frac{1}{2}\big(\,|\Psi\rangle - U\,|\Psi\rangle\,\big)\,|1\rangle\,.$$

Given that U is hermitian, its two eigenspaces are in direct sum. This implies that a generic state $|\Psi\rangle$ is a linear combination of one eigenstate per eigenspace. Let $|\uparrow\rangle$ and $|\downarrow\rangle$ be the two normalized eigenstates of U with eigenvalues 1 e -1 such that $|\Psi\rangle = a\,|\uparrow\rangle + b\,|\downarrow\rangle$. By substitution in the equation above, we get

$$|\text{measure}\rangle = a\,|\uparrow\rangle|0\rangle + b\,|\downarrow\rangle|1\rangle\,.$$

We see that the qubit to measure is entangled with the rest of the system in such a way that its measurement is equivalent to a measurement of U on the state $a\,|\uparrow\rangle + b\,|\downarrow\rangle$, which is exactly $|\Psi\rangle$.

Focus 2: *Performing a measurement on a state $|\Psi\rangle$ of m qubits with respect to an operator U with eigenvalues ± 1.*

Mutatis mutandis, the two error correcting codes are the same. We just need to encode the state with $a\,|{+}{+}{+}\rangle + b\,|{-}{-}{-}\rangle$ and get the error syndrome by measuring X_1X_2 and X_2X_3. To recover the original state, we apply a Z gate on the qubit where the error happened.

At this point, we have everything we need to understand *Shor's code*[6]: a *nine-qubit* code that can correct from any single qubit error. The general scheme is the same: encode a one qubit state with a redundant *many-qubits* state; send the qubits via independent channels; measure a set of observables to determine the error syndrome; finally use this information to recover the original state. Shor's code represents the state $|\Psi\rangle = a\,|0\rangle + b\,|1\rangle$ with the *nine-qubits* state:

$$|\Psi\rangle_L = a\left(\frac{|000\rangle + |111\rangle}{\sqrt{2}}\right)^3 + b\left(\frac{|000\rangle - |111\rangle}{\sqrt{2}}\right)^3.$$

In this context, exponentiation refers to the repeated application of the tensor product. This encoding is obtained by first applying the *phase-flip* code to $|\Psi\rangle$ and then applying the *bit-flip* code to all three qubits used for the *phase-flip* encoding. We need eight measurements to determine the error syndrome. The observables

$$Z_1Z_2, Z_2Z_3,\ Z_4Z_5, Z_5Z_6,\ Z_7Z_8, Z_8Z_9$$

determine the presence of any bit flip error with the same logic used for the *single qubit bit-flip* code. The observables

$$(X_1X_2X_3)(X_4X_5X_6), (X_4X_5X_6)(X_7X_8X_9)$$

are used to determine if any phase-flip error happened. Notice that the action of Z_1 on $|\Psi\rangle_L$ is equivalent to the action of Z_2 and Z_3. The same goes for Z_4, Z_5, Z_6 and Z_7, Z_8, Z_9. This is why there are only $4^4 = 2^8$ possible error syndromes. This time the received state $E\,|\Psi\rangle_L$ is not guaranteed to have well defined values for all the observables that we want to measure. In fact E can be a superposition of different error types. Nevertheless, since the error syndrome contains enough information to determine the type of error that occurred, the measurements must collapse the state $E\,|\Psi\rangle_L$ to a state with "*definite error*". We can recover the original state by applying X and Z gates where needed, therefore the system can recover from any single qubit error. We can recover from an error of type Y using gates of type X and Z because $Y = iXZ$.

III.II Fault tolerant computation

We desire to make entire computations reliable. To satisfy this request, we must consider that pretty much every part of a quantum computation is subject to noise: quantum gates, ancilla preparation, measurement devices, the quantum wires themselves, and so on. To perform fault tolerant computations people work directly on encoded states because redundancy is key, as always. As a consequence, for every quantum gate, we must design a circuit that implements it reliably on the bundle of quantum wires that represent the ingoing states. Clearly, every error correcting code requires different implementations. While performing fault tolerant computations, it's crucial to prevent errors from propagating. Gates are designed to guarantee that if any single component fails the output of the gate has at most one error per encoding bundle. As a consequence, applying the error correcting procedure on all outgoing bundles will restore the correct state.

Given an error correcting code of choice there's a class of gates that are easy to implement fault tolerantly. These are the *transversal* gates[2]: gates whose implementation on the encoding bundle is by definition as simple as applying the gate on all qubits. *Many-qubits* gates can be transversal just as single qubit gates can. For example, the CNOT gate is transversal if its implementation is given by a sequence of CNOT gates applied on all pairs of corresponding qubits in the two incoming bundles. Transversal gates are automatically fault tolerant, as the failure of one gate or one quantum wire cannot affect more than one qubit per bundle.

We can verify that the CNOT gate is transversal in Shor's code. On the contrary, the *Toffoli* gate is not transversal in Shor's code, nor in other common error correcting codes such as Steane's code[2, 8]. Fault tolerant *Toffoli* gate implementations for these codes exist[3] but require more complex circuits, which can allow errors to propagate more easily, therefore increasing the probability of failure.

III.III Stating the goal

We have seen that when using common error correcting codes the CNOT gate is transversal while the *Toffoli* gate isn't. This makes computations that require *Toffoli* gates noisier. We also know that the same computation can be performed by many quantum circuits whose design is different but all having the same describing matrix. As a silly example, applying a CNOT twice does nothing, so we could

take any circuit, add a pair of consecutive CNOTs and obtain an equivalent but different design. More interestingly, equivalent circuits with different numbers of *Toffoli* gates exist. Most of the time, the implementation with the lowest count of *Toffoli* gates is preferable, as it will perform more reliably. While for classical circuits a good measure of complexity is the total number of gates[7], for quantum circuits counting the number of *Toffoli* gates is more appropriate.

Solving the general problem of minimizing the number of *Toffoli* gates in a circuit, given a set of allowed gates, is far to ambitious for us; so we put ourselves in the simpler case of *classical decision problems* where there may be many input *non-ancilla* qubits but only one *non-ancilla* output qubit. We restrict the allowed gates to CNOT and *Toffoli*. The formalism developed in the next chapter will make obvious that any classical computation can be performed with only these two gates. We permit the use of an arbitrary number of *ancilla* qubits. Then, our goal is to study methods that given a desired classical decision problem try to find circuit implementations that use as few *Toffoli* gates as possible.

CHAPTER 2

Formal analysis

1. The Galois field GF(2)

Let $\mathbb{F}_2 := \{0, 1\}$ where 0 and 1 are symbols. The elements of $\mathbb{F}_2$ constitute the *binary alphabet*. Variables taking values in $\mathbb{F}_2$ are said to be *binary variables* and hold a *bit*. We endow $\mathbb{F}_2$ with three binary operations: AND($\cdot$), OR($+$) and XOR($\oplus$); whose definition in given by the truth Table 2. All three operations are commutative and associative by inspection. The AND operation is distributive both over the OR and the XOR operations. We can define the usual negation as $\bar{x} := x \oplus 1$. All this properties can be easily verified by manual inspection of all possible cases.

A	B	AB	$A + B$	$A \oplus B$
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Table 2: *Truth tables of the binary operations* AND, OR *and* XOR.

The structure $\mathrm{GF}(2) := (\mathbb{F}_2, \oplus, \cdot)$ is a Galois field[7], that is a field of finite cardinality. The XOR operation is the sum of the field, whose neutral element is 0. The opposite of $x \in \mathbb{F}_2$ is x, in fact $x \oplus x = 0$. The AND operation is the product of the field, whose neutral element is 1. The inverse of $x \in \mathbb{F}_2 \setminus \{0\}$ is 1, in fact $1 \cdot 1 = 1$. Note that $(\mathbb{F}_2, +, \cdot)$ is not a field since $0 + 1 = 1 + 1$ and therefore OR doesn't

have an inverse operations. It is possible to prove that GF(2) is the only *two-element* field and that no *one-element* field can exist.

I.I Bit strings

Elements of $\mathbb{F}_2^n$, where exponentiation denotes the repeated cartesian product, are said to be *bit strings*. Variables taking values in $\mathbb{F}_2^n$ will be written in bold, like $\boldsymbol{x} \in \mathbb{F}_2^n$, while their constituting bits will be denoted with a subscript like $\boldsymbol{x} =: (x_0, x_1, \ldots, x_{n-1})$ with $x_i \in \mathbb{F}_2$. Fixed a value of $n \in \mathbb{N}$, every bit string is uniquely associated with a natural number by the function $\mathrm{Bin} : \mathbb{F}_2^n \to [0, 2^n) \cap \mathbb{N}$ defined below.

$$\mathrm{Bin}(\boldsymbol{x}) := \sum_{k:x_k=1} 2^k$$

This map gives $\mathbb{F}_2^n$ a natural *strict total order relation* where $\boldsymbol{x} > \boldsymbol{y} \Leftrightarrow \mathrm{Bin}(\boldsymbol{x}) > \mathrm{Bin}(\boldsymbol{y})$. The inverse function $\mathrm{Bin}^{-1} : \mathbb{N} \cap [0, 2^n) \to \mathbb{F}_2^n$ exists and gives the bit string representation of the natural numbers from 0 to $2^n - 1$. Given that n is sufficiently large, the *bold version* of a natural number denotes the bit string obtained by the action of Bin^{-1} on it; for example $\mathbf{42} = \mathrm{Bin}^{-1}(42)$. We can define an operation $\mathbb{F}_2^n \times \mathbb{F}_2^n \to \mathbb{F}_2$ as follows and use the exponentiation notation for it.

$$\boldsymbol{a}^{\boldsymbol{b}} := \prod_{i:b_i=1} a_i \in \mathbb{F}_2, \quad \boldsymbol{a}, \boldsymbol{b} \in \mathbb{F}_2^n$$

The result of the operation is one if and only if the condition $(b_i = 1 \Rightarrow a_i = 1)$ is met. The contrapositive of this requirement is the equivalent $(\bar{a}_i = 1 \Rightarrow \bar{b}_i = 1)$ which implies the property $\boldsymbol{a}^{\boldsymbol{b}} = \bar{\boldsymbol{b}}^{\bar{\boldsymbol{a}}}$. The equivalence of two bit strings $\boldsymbol{a}$ and $\boldsymbol{b}$ can be written as $\boldsymbol{a}^{\boldsymbol{b}}\boldsymbol{b}^{\boldsymbol{a}} = 1$ since the equivalent logic condition is $(b_i = 1 \Leftrightarrow a_i = 1)$.

I.II Boolean functions

Boolean functions[7] are maps from bit strings to bit strings. We define $\mathbb{B}_{nm}$ as the set of all boolean functions of the kind $\mathbb{F}_2^n \to \mathbb{F}_2^m$. There are 2^n possible inputs and 2^m possible output values for every input therefore there are $(2^m)^{2^n}$ distinct boolean functions inside $\mathbb{B}_{nm}$. As a shorthand we denote $\mathbb{B}_{n1}$ with $\mathbb{B}_n$. A function $f \in \mathbb{B}_n$ is completely defined by its support $\mathrm{supp}(f)$, that is the set of all bit strings in $\mathbb{F}_2^n$ such that the evaluation of f on them gives 1.

$$\mathrm{supp}(f) := \{\boldsymbol{x} \in \mathbb{F}_2^n : f(\boldsymbol{x}) = 1\}$$

For example, if we choose $n = 3$, the function $f(\boldsymbol{x}) = x_0x_1x_2$ is uniquely determined by $\mathrm{supp}(f) = \{(1,1,1)\} = \{\mathbf{7}\}$.

The four operations we have defined on $\mathbb{F}_2$ induce four corresponding operations on $\mathbb{B}_n$. Let $f, g \in \mathbb{B}_n$ then

$$\begin{aligned}
(fg)(\boldsymbol{x}) &:= f(\boldsymbol{x})g(\boldsymbol{x}),\\
(f \oplus g)(\boldsymbol{x}) &:= f(\boldsymbol{x}) \oplus g(\boldsymbol{x}),\\
(f + g)(\boldsymbol{x}) &:= f(\boldsymbol{x}) + g(\boldsymbol{x}),\\
\bar{f}(\boldsymbol{x}) &:= \overline{f(\boldsymbol{x})}.
\end{aligned}$$

This four operations on boolean functions have equivalent operations on their set representations.

$$\begin{aligned}
&\mathrm{supp}(fg) = \mathrm{supp}(f) \cap \mathrm{supp}(g),\\
&\mathrm{supp}(f \oplus g) = \mathrm{supp}(f) \cup \mathrm{supp}(g) \setminus \mathrm{supp}(f) \cap \mathrm{supp}(g),\\
&\mathrm{supp}(f + g) = \mathrm{supp}(f) \cup \mathrm{supp}(g),\\
&\mathrm{supp}(\bar{f}) = \mathbb{F}_2^n \setminus \mathrm{supp}(f).
\end{aligned}$$

II. Normal forms

Boolean functions of $\mathbb{B}_n$ can be expressed algebraically in many different ways. A normal form[7] is a particular prescription that allows to choose a unique expression for every function in $\mathbb{B}_n$.

II.I Disjunctive Normal Form

The *Disjunctive Normal Form*, also known as *Sum of min-terms*, of a function $f \in \mathbb{B}_n$ is defined by the following expression.

$$f(\boldsymbol{x}) = \sum_{\boldsymbol{k} \in \mathbb{F}_2^n} f(\boldsymbol{k}) \prod_{i=0}^{n-1} (x_i \oplus \bar{k}_i) \tag{1}$$

We can prove this formula is correct by noting that the repeated product inside the summation is 1 if and only if $\boldsymbol{x} = \boldsymbol{k}$. It follows that the summation can have a maximum of one non-zero term which evaluates to $f(\boldsymbol{x})$. As a consequence of the fact that the summation has at most one non-zero term, the OR-summation can be replaced by a XOR-summation.

$$f(\boldsymbol{x}) = \bigoplus_{\boldsymbol{k} \in \mathbb{F}_2^n} f(\boldsymbol{k}) \prod_{i=0}^{n-1} (x_i \oplus \bar{k}_i) \tag{2}$$

We can also exploit the property ($\boldsymbol{x} = \boldsymbol{k} \Leftrightarrow \boldsymbol{x}^{\boldsymbol{k}}\boldsymbol{k}^{\boldsymbol{x}} = 1$) and the relation $\boldsymbol{k}^{\boldsymbol{x}} = \bar{\boldsymbol{x}}^{\boldsymbol{k}}$ to get the following more compact expression.

$$f(\boldsymbol{x}) = \bigoplus_{\boldsymbol{k}\in\mathbb{F}_2^n} f(\boldsymbol{k})\boldsymbol{x}^{\boldsymbol{k}}\bar{\boldsymbol{x}}^{\bar{\boldsymbol{k}}}$$

II.II Algebraic Normal Form

The *Algebraic Normal Form*, also known as *Ring Sum Expansion*[7], can be informally defined by the following procedure: take (2), flatten the expression by carrying out all products and cancel like terms in the XOR-summation. The result of this process is a polynomial expression in the indeterminates x_i. Considering that repeated multiplication does nothing $x_i x_i = x_i$, all possible monomials are of the form $\boldsymbol{x}^{\boldsymbol{k}}$ and all of them can in principle appear in the algebraic normal form. If we call $\tilde{f}(\boldsymbol{k})$ the coefficients in front of the monomials $\boldsymbol{x}^{\boldsymbol{k}}$ we get the following expression for the algebraic normal form of the function f.

$$f(\boldsymbol{x}) = \bigoplus_{\boldsymbol{k}\in\mathbb{F}_2^n} \tilde{f}(\boldsymbol{k})\boldsymbol{x}^{\boldsymbol{k}} \tag{3}$$

We can prove formally that such a function $\tilde{f} \in \mathbb{B}_n$ exists for every function $f \in \mathbb{B}_n$ by giving $\tilde{f}$ explicitly.

$$\tilde{f}(\boldsymbol{k}) = \bigoplus_{\boldsymbol{x}\in\mathbb{F}_2^n} f(\boldsymbol{x})\boldsymbol{k}^{\boldsymbol{x}} \tag{4}$$

We will call $\Theta[f] = \tilde{f}$ the *Möbius transform* of f. In order for equation (3) to hold, we see that the function f must it self be the Möbius transform of the function $\tilde{f}$. We give a proof of this in Focus 3. This is sufficient to guarantee that $\tilde{f}$ exists and that (3) is correct. It is easy to see that the Möbius transform is linear, that is $\Theta[\alpha f + \beta g] = \alpha\Theta[f] + \beta\Theta[g]$ given $f, g \in \mathbb{B}_n$ and $\alpha, \beta \in \mathbb{F}_2$.

III. Upper bounds to the number of Toffoli gates

As described in greater detail in section 1.3.3, our goal is to find quantum circuits made of CNOT and *Toffoli* gates that implement a given classical decision problem, that is a map from n bits to one bit, with as few *Toffoli* gates as possible. The mathematical structures defined in this chapter are a valuable tool that we can harness to tackle the problem. As showed by the half-adder of Figure 1 the

Inverse of the Möbius transform

THEOREM 2: *(Statement)* The Möbius transform (4) has an inverse and that inverse is the Möbius transform itself.

(Proof) We can show explicitly that $\tilde{\tilde{f}} = f, \forall f \in \mathbb{B}_n$.

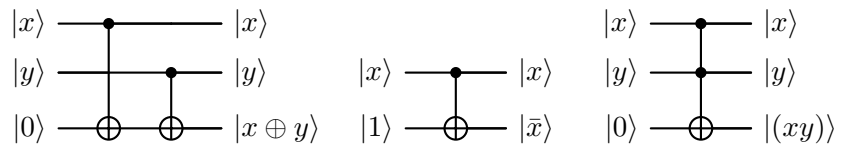
$$\tilde{\tilde{f}}(\boldsymbol{x}) = \bigoplus_{\boldsymbol{k}\in\mathbb{F}_2^n}\left(\bigoplus_{\boldsymbol{h}\in\mathbb{F}_2^n} f(\boldsymbol{h})\boldsymbol{k}^{\boldsymbol{h}}\right)\boldsymbol{x}^{\boldsymbol{k}} = \bigoplus_{\boldsymbol{h}\in\mathbb{F}_2^n} f(\boldsymbol{h})\left(\bigoplus_{\boldsymbol{k}\in\mathbb{F}_2^n}\boldsymbol{x}^{\boldsymbol{k}}\boldsymbol{k}^{\boldsymbol{h}}\right)$$

The last expression looks similar to the disjunctive normal form of f. To prove the equality holds, it's easier to start from the DNF and reach this form. We take equation (2) and expand the products.

$$f(\boldsymbol{x}) = \bigoplus_{\boldsymbol{h}\in\mathbb{F}_2^n} f(\boldsymbol{h})\prod_{i=0}^{n-1}(x_i \oplus \bar{h}_i) = \bigoplus_{\boldsymbol{h}\in\mathbb{F}_2^n} f(\boldsymbol{h})\left(\bigoplus_{\boldsymbol{k}\in\mathbb{F}_2^n}\boldsymbol{x}^{\boldsymbol{k}}\bar{\boldsymbol{h}}^{\bar{\boldsymbol{k}}}\right)$$

Note that the XOR-summation over $\boldsymbol{k}$ goes over all possible ways of choosing between the terms x_i or $\bar{h}_i$ while doing the expansion. Now, we make use of the previously proven property $\bar{\boldsymbol{k}}^{\bar{\boldsymbol{h}}} = \boldsymbol{h}^{\boldsymbol{k}}$ to obtain that $\Theta[\Theta[f]]$ is equal to f. This implies, by definition of inverse transform, that $\Theta^{-1} = \Theta$.

Focus 3: *Proving that the Möbius transform is its own inverse.*



(a) XOR (b) NOT (c) AND

Figure 3: *How to implement the logical* XOR*, the logical* NOT *and logical* AND *of classical states without changing them.*

action of the CNOT gate on a basis state is to overwrite the qubit on the target wire with the XOR of the two qubits and the action of the *Toffoli* gate is to XOR-sum the AND-product of the controlling qubits onto the target qubit. This implies that with the help of an ancilla qubit we can take the XOR and AND of two qubits without changing their state. Negation is even easier, in fact applying a CNOT gate on $|x\rangle|1\rangle$ gives $|x\rangle|\bar{x}\rangle$. This, together with the implementations of Figure 3, demonstrates that, given any boolean expression containing only S XOR-sums, N negations, P products and a desired number of ones, zeros and parenthesis, its implementation requires no more then: $(2S + N)$ CNOT gates, P *Toffoli* gates and $(S + N + P + 2)$ ancilla qubits. Given any $f \in \mathbb{B}_n$ the minimum number of *Toffoli* gates required to implement it will be denoted with $\mathrm{Tof}(f)$. Different expressions of f can give different upper bounds to $\mathrm{Tof}(f)$.

III.I Upper bounds given by the DNF

We consider equation (2) to get a first upper bound. The expression contains $(n-1)$ products for every bit string in the support of f. Since the only function whose support has cardinality 2^n is the constant function 1, which we know can be done with no products, the upper bound to the number of *Toffoli* gates provided by the expression (2) can be lowered from $(n-1)2^n$ to $(n-1)(2^n-1)$. Taking into consideration that $f = \bar{f} \oplus 1$ we conclude that implementing f requires no more *Toffoli* gates than implementing $\bar{f}$.

$$\mathrm{Tof}(f) \leq (n-1)\min\{|\operatorname{supp}(f)|, 2^n - |\operatorname{supp}(f)|\}$$
$$\mathrm{Tof}(f) \leq (n-1)2^{n-1}$$

The two vertical bars denote the cardinality of the set. The first upper bound is function dependent while the second requires no notion over the function; both are interesting.

The average number of products in the DNF is $(n-1)2^{n-1}$, half of the maximum. This is because all functions can be paired such that the two DNF*s* contain a total of $(n-1)2^n$ products.

III.II Upper bounds given by the ANF

We now consider equation (3) in the hope of finding tighter upper bounds to the number of *Toffoli* gates required. Given a bit string $\boldsymbol{k}$ the expression $\boldsymbol{x}^{\boldsymbol{k}}$ is a product of as many terms as there are ones in $\boldsymbol{k}$. Since the summation goes over all possible 2^n values of $\boldsymbol{k}$, the total number of ones is $n2^{n-1}$, half of the total number of bits. We have to account for the fact that multiplying m terms requires $(m-1)$ products when $m > 0$ and none otherwise. There's only one bit string with no ones therefore we get the following upper bound.

$$\mathrm{Tof}(f) \leq n2^{n-1} - (2^n - 1) = (n-2)2^{n-1} + 1$$

In this case, considering $f = \bar{f} \oplus 1$ does not reduce the upper bound, in fact the algebraic normal form of f and $\bar{f}$ differ only in the free term. This new upper bound is lower than the one found in section 2.3.1 but they both scale as $\mathcal{O}(n2^{n-1})$.

The average number of products in the ANF is half of the upper bound, as all functions can be ordered in pairs with complementary algebraic normal forms.

III.III Exponential upper bound

The distributive property of the AND operation over the XOR operation allows the collection of terms inside expressions. The best upper bound we have discussed up until now is that given by the ANF, which is a flat expression and therefore one that doesn't make use of the distributive property to reduce the number of products. Collecting terms always reduces the number of products in the expression therefore it's clear that better upper bounds can be found. The power of term collection is encapsulated by the following general property.

$$f(\boldsymbol{x}) = f(x_{j\neq i}, x_i = 0) \oplus x_i\big(f(x_{j\neq i}, x_i = 0) \oplus f(x_{j\neq i}, x_i = 1)\big) \quad (5)$$

This expression exposes the dependency of f from x_i and can be verified by inspection of the only two possible cases: $x_i = 0$ and $x_i = 1$. Both $f(x_{j\neq i}, x_i = 0)$ and $f(x_{j\neq i}, x_i = 0) \oplus f(x_{j\neq i}, x_i = 1)$ are functions of $\mathbb{B}_{n-1}$; this implies

$$\mathrm{Tof}(n) \leq 1 + 2\,\mathrm{Tof}(n-1).$$

Putting ourselves in the worst case scenario, where the equality holds for every $n > 0$, we get a *linear difference equation* that, given the initial condition $\mathrm{Tof}(1) = 0$, has the unique solution $2^{n-1} - 1$.

$$\mathrm{Tof}(n) \leq 2^{n-1} - 1 \tag{6}$$

This is an exponential upper bound so it scales better than all the ones previously found.

CHAPTER 3

Algorithms

In chapter 2 we discussed general upper bounds to the number of Toffoli gates required to implement any function of $\mathbb{B}_n$. Now, we look into algorithms that can lower the upper bound for a given function by explicitly providing an expression that uses fewer products.

I. Fast Möbius Transform

The first thing we need is an efficient way to compute the Möbius transform of a function. If we were to calculate the truth table of $\Theta[f]$ naïvely, we would loop through all 2^n possible values of $\boldsymbol{x} \in \mathbb{F}_2^n$ in equation (4) for all 2^n possible values of $\boldsymbol{k}$, therefore the total expense would be of order $\mathcal{O}(2^n \cdot 2^n)$. By doing so we would read from the truth table of f a total of $2^n \cdot 2^n$ times.

Better algorithms for computing the truth table of $\Theta[f]$ exist. In fact, we can exploit the linearity of Θ and the property (5) to reduce the problem of finding the Möbius transform of a function in $\mathbb{B}_n$ to finding that of two functions in $\mathbb{B}_{n-1}$.

$$\begin{aligned}\Theta[f(\boldsymbol{x})] = \Theta[f(x_{i<n-1}, 0)] \oplus \\ \Theta[x_{n-1} f(x_{i<n-1}, 0)] \oplus \\ \Theta[x_{n-1} f(x_{i<n-1}, 1)]\end{aligned} \tag{7}$$

Note that $\Theta[x_{n-1} f(x_{i<n-1}, 0)]$ and $\Theta[x_{n-1} f(x_{i<n-1}, 1)]$ are still functions of $\mathbb{B}_n$ but their truth tables are trivial known the truth tables of $\Theta[f(x_{i<n-1}, 0)]$ and $\Theta[f(x_{i<n-1}, 1)]$, which are functions of $\mathbb{B}_{n-1}$ as anticipated. It is important to observe that the truth table of the XOR-sum of boolean functions is the bit by bit XOR-sum of the truth tables of the original functions. As a consequence, we can turn

this property of boolean functions into an equivalent relation on their truth tables. Figure 4 provides a visual representation of this fact.

By recursively applying equation (7), we can determine the truth table of $\Theta[f(\boldsymbol{x})]$. The inductive base is the truth table of the Möbius transform of a constant function which is a sequence of 2^n zeros, in the case $f(\boldsymbol{x}) \equiv 0$, and a single one followed by $(2^n - 1)$ zeros in the case $f(\boldsymbol{x}) \equiv 1$. This *divide et impera* algorithm is called *Fast Möbius Transform*[1]. The algorithm is recursive by design, nevertheless it's easy to see that the truth table of two functions of $\mathbb{B}_{n-1}$ combined have the same length as the truth table of a function in $\mathbb{B}_n$. This means that, by saving only the truth tables of $\Theta[f(x_{i<n-1}, 0)]$ and $\Theta[f(x_{i<n-1}, 1)]$, the entire operation can be implemented without allocating new memory at every step of the recursion. With this in mind, the recursion can be easily removed, making the algorithm an iterative one, as shown in Figure 5.

The complexity of the Fast Möbius Transform scales as $\mathcal{O}(n2^n)$, since there are n steps each of which requires $2^n/2$ XOR operations. If we express the complexity in terms of the input size, which is $N = 2^n$, we get $\mathcal{O}(N \log_2(N))$. This is a clear improvement on the naïve implementation, which scales as $\mathcal{O}(2^n \cdot 2^n) = \mathcal{O}(N^2)$.

1.1 An example

We decide to use Figure 5 as an example. We are in the case $n = 3$. Let's denote $\boldsymbol{x}$ with (x, y, z). The function, whose Möbius transform we want to calculate, is

$$f(x, y, z) = x\bar{y}\bar{z} \oplus \bar{x}y\bar{z} \oplus x\bar{y}z.$$

We have written f in it's disjunctive normal form, so the truth table of f is manifest in this expression. The truth table of f is the input of the algorithm and in Figure 5 is written in the topmost row. We see that every one in that row corresponds to a min-term that appears in the disjunctive normal form as implied by equation (2). We can reach the algebraic normal form of f by hand, applying the property $\bar{a} = a \oplus 1$ and carrying out the products wherever possible in the expression above.

$$\begin{aligned} f(x, y, z) &= x(y \oplus 1)(z \oplus 1) \oplus (x \oplus 1)y(z \oplus 1) \oplus x(y \oplus 1)z \\ &= xyz \oplus xy \oplus xz \oplus x \oplus yxz \oplus xy \oplus yz \oplus y \oplus xyz \oplus xz \\ &= x \oplus y \oplus yz \oplus xyz \end{aligned}$$

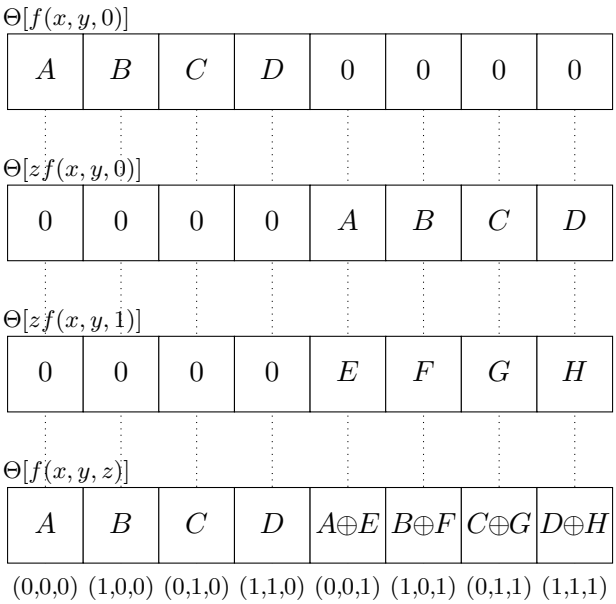


Figure 4: *The decomposition (7) visualized using truth tables.*

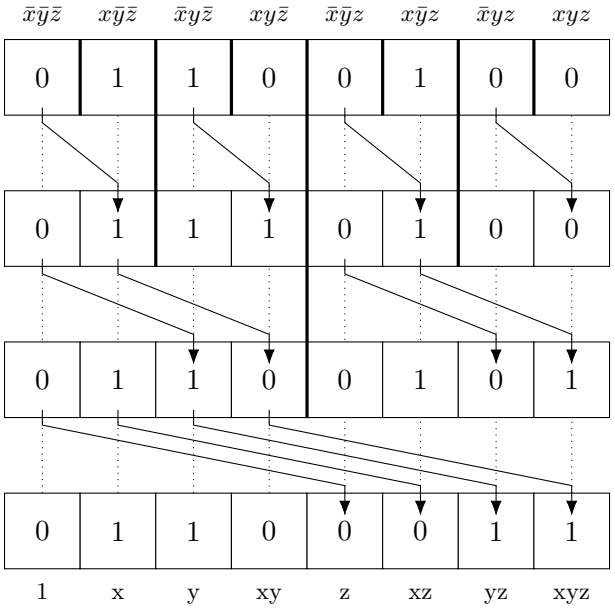


Figure 5: *Iterative implementation of the FMT algorithm visualized. The input truth table of f is the first row while the output truth table of $\Theta[f]$ is the last row.*

Now we have written f in it's algebraic normal form, so the truth table of $\Theta[f]$ is manifest in this expression. The truth table of $\Theta[f]$ is the output of the FMT and in Figure 5 is written in the last row. In fact, we see that every one in that row corresponds to a monomial that appears in the above equation as implied by (3).

II. Greedily collecting terms

Thanks to the FMT we are now able to efficiently compute the ANF of any desired function $f \in \mathbb{B}_n$. We shall consider this the first step of our algorithm.

The second step is to recursively collect terms. We shall notice that, although term collection cannot increase the number of products, different ways of collecting terms can lead to expressions for f that contain products in different amounts. On long expressions, the number of ways in which we can collect terms can be very large, therefore we focus on a subset of these. We choose to collect only one variable at a time and to make our algorithm greedy: at every step it will collect the variable that appears more often in the ANF of the function and it will collect it from all terms where it appears.

These choices bring us back to equation (5), which describes exactly the kind of term collection we decided to perform. We now see that computing the algebraic normal form of the function is in fact needed in order to know which variable to collect. One could skip the FMT and apply equation (5) directly, but he would probably have to think about some other heuristic to choose a good variable to collect. Trying all options is not feasible for big ns, in fact there are $[n!]^{2^0}[(n-1)!]^{2^1}\cdots[1!]^{2^{n-1}}$ ways of recursively applying (5).

II.I Greedily choosing a variable to collect

To count the number of times c_v the variable x_v appears in the ANF of the function $g \in \mathbb{B}_m$ we must consider the truth table of $\theta[g]$. As showed in Figure 6, the terms of the ANF containing the variable x_v are gathered in alternating groups of length 2^v; *even* groups do not contain the variable while *odd* groups do. If we loop over the *odd* groups with a summation over i, and over all consecutive terms in the group with a summation over j, we can count the total number of

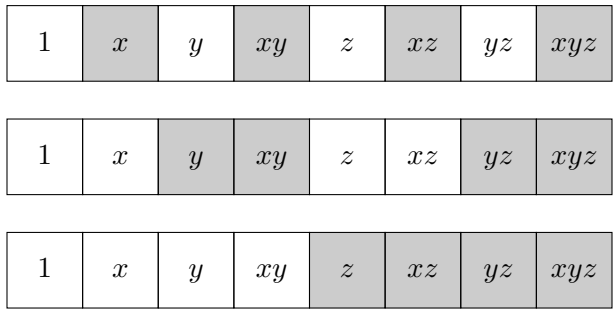


Figure 6: *The visual pattern governing where the monomials containing a given variable appear in the ANF of a function $f(x, y, z)$. In order from top to bottom: the terms with x, y and z respectively.*

times x_v appears with the following expression:

$$c_v = \sum_{i=1}^{2^{m-v-1}} \sum_{j=1}^{2^v} \tilde{g}(\mathrm{Bin}^{-1}((2i-1)2^v + j - 1)),$$

where the values 0 and 1 of $\mathbb{F}_2$ are implicitly mapped to the natural numbers 0 and 1 such that the summations are over the naturals. The collected variable is x_w such that $c_w = \max\{c_v \in \mathbb{N} : v \in [0, m) \cap \mathbb{N}\}$. If more than one w satisfies the above condition then the minimum of those is chosen. Evaluating all the elements of $\{c_v\}$ has a cost of order $\mathcal{O}(m2^{m-1})$ since the cardinality of $\{c_v\}$ is m and every c_v is obtained by summing $2^{m-v-1} \cdot 2^v$ values. Finding the maximum is of linear complexity in m, so the scaling does not change.

II.II Collecting a variable

A routine which collects a variable x_i from the ANF of $g \in \mathbb{B}_m$ has the goal of turning the truth table of $\theta[g]$ into two smaller truth tables: the truth table of $\theta[g(x_i = 0; x_{j\neq i})]$ and the truth table of $\theta[g(x_i = 0; x_{j\neq i}) \oplus g(x_i = 1; x_{j\neq i})]$, which represent the algebraic normal forms of the functions in $\mathbb{B}_{m-1}$ that appear respectively outside and inside the parenthesis of equation (5). By doing so we get one ANF as input and give two ANF*s* as output.

As showed by Figure 7, performing this operation is just a matter of reordering groups of bits. First, we identify which bits do and do not correspond to a monomial containing x_i, then move those which do not on the first half of the bit-string and move those which do on the second half, all while maintaining the relative order between

terms of the same type. Determining the new position of every element of the *bit-string* requires a calculation performable in constant time, therefore the computational expense of collecting a variable is of order $\mathcal{O}(2^m)$.

II.III Total cost

Let's suppose the input function is $f \in \mathbb{B}_n$. The total cost of the greedy algorithm is dominated by the *terms-counting* operations so the total computational expense is of order

$$\mathcal{O}\left(\sum_{m=1}^{n} 2^{n-m} \cdot m 2^{m-1}\right) = \mathcal{O}\left(2^{n-1} \sum_{m=1}^{n} m\right) = \mathcal{O}(n^2 2^n).$$

This order of complexity is higher than that of the Möbius transform, therefore the cost of going from the truth table of f to the binary tree of Figure 8 is of order $\mathcal{O}(n^2 2^n)$.

II.IV Outputting human readable equations

The output of the algorithm is a binary tree whose nodes represent the act of collecting a variable and whose leaves determine whether a term is present or not in the expression. Since humans have a hard time reading this output, it's useful to turn the tree into a proper mathematical expression. For example, the tree of Figure 8 gives the string `x+y(1+z(1+x))`. To turn the tree into a string, we traverse it with a *postorder depth first* recursion[4]. Evaluating a node results in a string containing the mathematical expression of the corresponding part of the equation.

III. Results

To evaluate the efficiency of the greedy algorithm in minimizing the number of products, we compare the results with those obtained using the disjunctive and algebraic normal forms directly, as well as with the method of sequentially collecting the variables x_i from $i = 0$ to $i = (n - 1)$, independently of the number of times they appear in the expression. The averages for the disjunctive and algebraic normal forms are known exactly, as discussed in sections 2.3.1 and 2.3.2. The averages for the greedy and "in order" methods are calculated exactly up to $n = 4$, whereas for $n > 4$ we sampled only a subset of $\mathbb{B}_n$. We plot the averages as functions of n in Figure 9 and transcript in Table 3

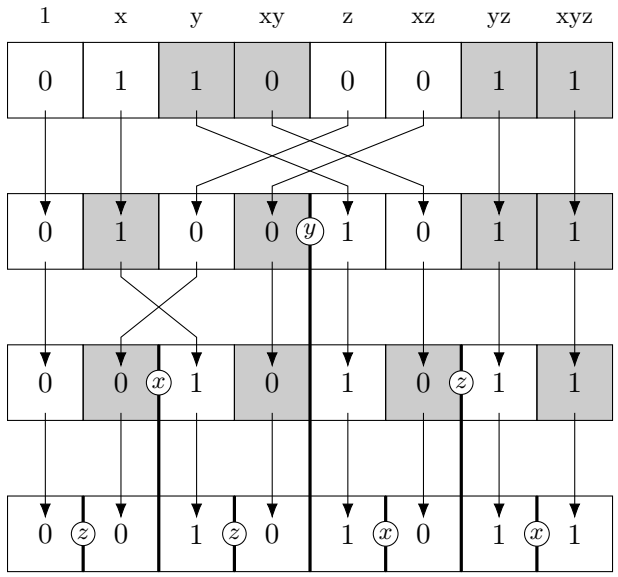


Figure 7: *Greedy algorithm visualized. The first row contains the truth table of the input Möbius transform. The last row contains the leaves of the output binary tree. The letters surrounded by circles represent variables that where collected: on their right go the terms inside the parenthesis, on the left the others.*

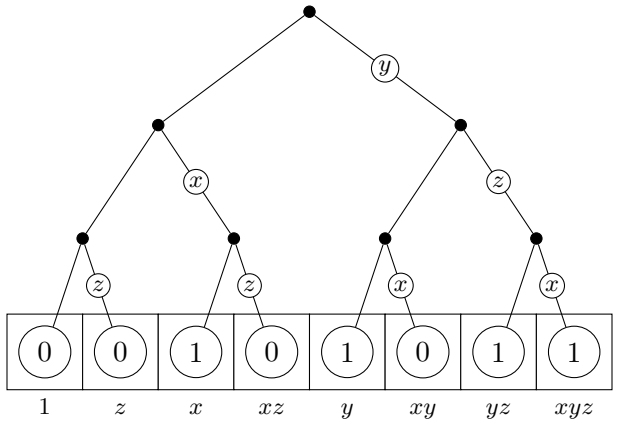


Figure 8: *Example of tree structure given by the greedy algorithm. This binary tree corresponds to the expression* $x \oplus y(1 \oplus z(1 \oplus x))$*, which contains exactly two products.*

n	Average number of products			
	DNF	ANF	In order	Greedy
2	2.00	0.50	0.50	0.50
4	24.00	8.50	4.74	3.92
8*	896.1 ± 0.2	384.24 ± 0.08	90.85 ± 0.02	70.79 ± 0.01
16*	$491\,518 \pm 6$	$229\,376 \pm 3$	$23\,519.3 \pm 0.2$	$18\,066.6 \pm 0.2$

Table 3: *Table showing how our greedy algorithm performs on average with respect to: using the DNF, using the ANF or collecting the variables in order. Exact values are rounded to the second decimal digit.* **The averages are estimated statistically on a random set of 10^5 truth tables.*

Method	Average	Upper bound	Complexity
DNF	$(n-1)2^{n-1}$	$(n-1)2^n$	free
ANF	$(n-2)2^{n-2}+1/2$	$(n-2)2^{n-1}+1$	$\mathcal{O}(n2^n)$
In order*	$0.36 \cdot 2^{1.00n} - 0.05$	$2^{n-1}-1$	$\mathcal{O}(n2^n)$
Greedy*	$0.28 \cdot 2^{1.00n} - 1.00$	$2^{n-1}-1$	$\mathcal{O}(n^2 2^n)$

Table 4: *Comparing the scaling of the average performance, worst case scenario and complexity of all discussed methods.* **The scaling of the average is obtained by fitting the model $a2^{bn}+c$ on the range $8 \leq n \leq 16$ to the averages estimated with a uniform sample of functions; fitted values are rounded to the second decimal digit.*

the values obtained computationally for $n \in \{2, 4, 8, 16\}$. We decided to plot on Figure 10 the entire distribution of number of products for the greedy and "in order" methods. For every value of n, the lowest average is that given by the greedy algorithm. We decided to fit the model $a2^{bn} + c$ on the previously estimated averages given both by the greedy and by the "in order" algorithm, for n in the range $[8, 16]$. The results are presented in Table 4 and plotted in Figure 9.

Testing the algorithm on generic functions is not enough, as often useful decision problems have structure in them. For this reason, we test the different approaches also on three common problems: computing equality between $\boldsymbol{x}$ and $\boldsymbol{y}$ with $\boldsymbol{x}^{\boldsymbol{y}}\bar{\boldsymbol{x}}^{\bar{\boldsymbol{y}}}$, comparing bit strings

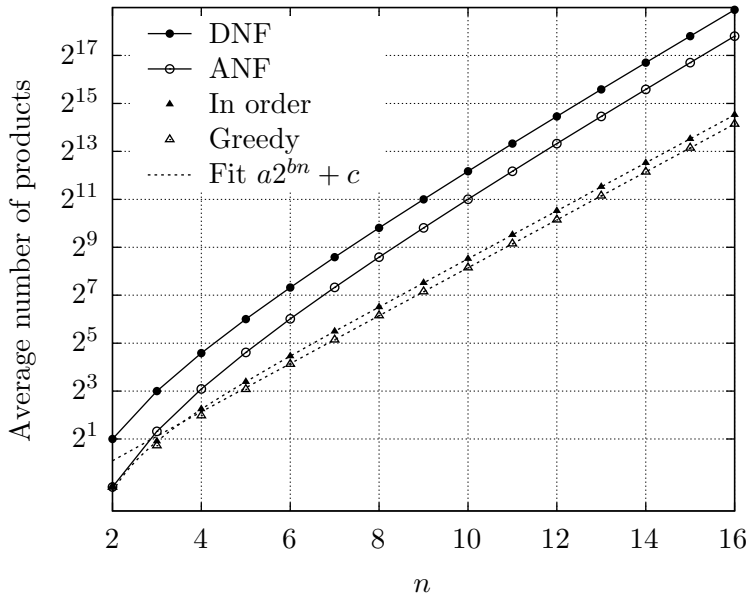


Figure 9: *The values for "DNF" and "ANF" are calculated exactly. The values for "Greedy" and "In order" are obtained statistically with enough accuracy for the error to be negligible.*

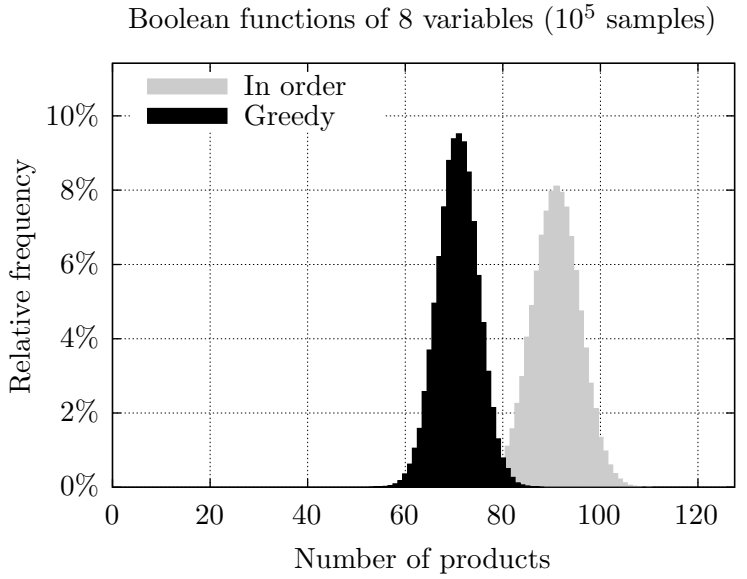


Figure 10: *Histogram of the number of products in the final expression from a uniform sample of 10^5 functions in $\mathbb{B}_8$. The error bars are too small to be displayed.*

with $\boldsymbol{x} > \boldsymbol{y}$ and evaluating $\boldsymbol{x}^{\boldsymbol{y}}$. The results are summarized in Table 5. In all scenarios the least number of products is achieved by the greedy algorithm. Compared to the average function, these require significantly less products. Nevertheless, a closer inspection of the results reveals a limitation of our approach. In fact, counting the number of products in the expression is not really representative of the number of *Toffoli* gates required to implement a function: we should instead count unique products. If for example we take a look at the final expression for $f(\boldsymbol{x}, \boldsymbol{y}) = \boldsymbol{x}^{\boldsymbol{y}}\bar{\boldsymbol{x}}^{\bar{\boldsymbol{y}}}$ we see that the same calculations appear multiple times. Let $n = 8$ and $(\boldsymbol{x}, \boldsymbol{y}) = (a, b, c, d, e, f, g, h)$.

$$\begin{aligned}\boldsymbol{x}^{\boldsymbol{y}}\bar{\boldsymbol{x}}^{\bar{\boldsymbol{y}}} = {} & 1 \oplus h \oplus d \oplus g(1 \oplus h \oplus d) \oplus c(1 \oplus h \oplus d) \oplus \\ & f(1 \oplus h \oplus d \oplus g(1 \oplus h \oplus d) \oplus c(1 \oplus h \oplus d)) \oplus \\ & b(1 \oplus h \oplus d \oplus g(1 \oplus h \oplus d) \oplus c(1 \oplus h \oplus d)) \oplus \\ & e(1 \oplus h \oplus d \oplus g(1 \oplus h \oplus d) \oplus c(1 \oplus h \oplus d) \oplus \\ & f(1 \oplus h \oplus d \oplus g(1 \oplus h \oplus d) \oplus c(1 \oplus h \oplus d)) \oplus \\ & b(1 \oplus h \oplus d \oplus g(1 \oplus h \oplus d) \oplus c(1 \oplus h \oplus d))) \oplus \\ & a(1 \oplus h \oplus d \oplus g(1 \oplus h \oplus d) \oplus c(1 \oplus h \oplus d) \oplus \\ & f(1 \oplus h \oplus d \oplus g(1 \oplus h \oplus d) \oplus c(1 \oplus h \oplus d)) \oplus \\ & b(1 \oplus h \oplus d \oplus g(1 \oplus h \oplus d) \oplus c(1 \oplus h \oplus d)))\end{aligned}$$

We count a total of 26 products but only 6 of these are unique. We can take advantage of this fact to manually compile this formula into an implementing circuit with only six *Toffoli* gates. Studying new and existing techniques to make this step automatic may be the goal of a future work. We can improve the expression further if we allow ourselves to collect entire expressions instead of single variables, as we have done up until now. So, we obtain the following formula with only three products starting from the previous one and collecting repeated expressions wherever possible.

$$\boldsymbol{x}^{\boldsymbol{y}}\bar{\boldsymbol{x}}^{\bar{\boldsymbol{y}}} = (a \oplus e \oplus 1)(b \oplus f \oplus 1)(c \oplus g \oplus 1)(d \oplus h \oplus 1).$$

Supposing that the pattern continues, Table 5 shows that the final number of products in the expression given by the greedy algorithm is $\mathcal{O}(3^{n/2})$ while the number of products in the generalization of the expression above is $\mathcal{O}(n)$.

In summary, although the greedy algorithm performs better than the other discussed methods, both on average and on the considered structured functions, manual inspection shows that the latter can be done more efficiently.

$f(\boldsymbol{x}, \boldsymbol{y})$	n	Products		
		DNF	ANF	Greedy
$\boldsymbol{x}^{\boldsymbol{y}}\bar{\boldsymbol{x}}^{\bar{\boldsymbol{y}}}$	2	2	0	$0 = 3^0 - 1$
	4	12	4	$2 = 3^1 - 1$
	8	112	136	$26 = 3^3 - 1$
	16	3840	28432	$2186 = 3^7 - 1$
$\boldsymbol{x} > \boldsymbol{y}$	2	1	1	$1 = 3^0$
	4	18	8	$3 = 3^1$
	8	849	176	$27 = 3^3$
	16	489600	31712	$2187 = 3^7$
$\boldsymbol{x}^{\boldsymbol{y}}$	2	3	1	1
	4	27	10	4
	8	567	244	40
	16	98415	45928	3280

Table 5: *Table showing how our greedy algorithm performs with structured functions such as the greater-than function* $(\boldsymbol{x} > \boldsymbol{y})$*, the equals-to function* $\boldsymbol{x}^{\boldsymbol{y}}\boldsymbol{y}^{\boldsymbol{x}}$ *and exponentiation* $\boldsymbol{x}^{\boldsymbol{y}}$*. Here* $(\boldsymbol{x}, \boldsymbol{y}) \in \mathbb{F}_2^n$*.*

Considering that collecting one variable at a time gave us an upper bound to the number of products that is of order $\mathcal{O}(2^n)$ and that the same is true both for the average of the greedy and the "in order" algorithm, we argue that probably even using the optimal sequence of *single variable* collecting moves would give an average number of products that grows as $\mathcal{O}(2^n)$. On the other hand, we think that allowing for the collection of entire expressions would change the scaling of the average number of products. Future work is needed in order to determine whether collecting arbitrary expressions can be performed efficiently by giving names to expressions and iterating our *variable collecting* algorithm. For example, the final expression of the greedy algorithm reveals that $\boldsymbol{x}^{\boldsymbol{y}}\bar{\boldsymbol{x}}^{\bar{\boldsymbol{y}}}$ can be written as a function of the seven variables $(a, b, c, e, f, g, 1 \oplus h \oplus d)$.

Conclusions

We started our discussion talking about how fault tolerant computation is achieved on quantum computers despite the presence of noise. We presented Shor's code and showed that transversal gates are easier to implement fault tolerantly. As the CNOT gate is transversal in most common error correcting codes while the *Toffoli* gate isn't, we concluded that fault tolerant *Toffoli* gates are more expensive in terms of noise. We set the goal of reducing the number of *Toffoli* gates required to implement a classical decision problem on a quantum computer that can utilize NOT, CNOT, and *Toffoli* gates, and with an infinite supply of *ancilla qubits*.

Then we showed that finding an expression which contains P products for a function $f \in \mathbb{B}_n$ is a direct way of proving that the corresponding decision problem can be implemented with no more than P *Toffoli* gates. This is why we looked into normals forms: these gave us upper bounds to the number of *Toffoli* gates of the order $\mathcal{O}(n2^n)$, where n is the number of input qubits. Afterwards we managed to improve the theoretical upper bound by leveraging the distributive property of the logical AND over the logical XOR. What we got was the upper bound $\text{Tof}(n) \leq 2^{n-1} - 1$.

We than looked into general algorithms that can lower the upper bound for a given function. We built on top of the Fast Möbius Transform and obtained a greedy algorithm that recursively collects terms with complexity scaling as $\mathcal{O}(n^2 2^n)$. As showed by Table 3, we demonstrated statistically that the greedy approach performs better on average than using the normal forms directly or collecting variables in a fixed order. We also showed the limitations of working on algebraic expressions instead of circuits by considering structured functions which result in high counts of products most of which repeat.

C implementations of all discussed algorithms are available online at `https://github.com/ZaninDavide/bachelor_dissertation`.

Bibliography

[1] Morgan Barbier, Hayat Cheballah, and Jean-Marie Le Bars. "On the computation of the Möbius transform". In: *Theoretical Computer Science* 809 (2020), pp. 171–188. ISSN: 0304-3975. DOI: https://doi.org/10.1016/j.tcs.2019.12.005.

[2] Michael A. Nielsen & Isaac L. Chuang. *Quantum Computation and Quantum Information*. Cambridge University Press, 2010. ISBN: 9781107002173.

[3] Daniel Gottesman. *Stabilizer Codes and Quantum Error Correction*. 1997. eprint: quant-ph/9705052 (quant-ph). URL: https://arxiv.org/abs/quant-ph/9705052.

[4] Alan Bertossi & Alberto Montresor. "Algoritmi e strutture di dati - 3ª Ed." In: Torino: Cittá Studi Edizioni, 2014. Chap. 5, 10.

[5] Jun John Sakurai & Jim Napolitano. *Modern Quantum Mechanics*. Cambridge University Press, 2017. ISBN: 9781108645928.

[6] Peter W Shor. "Scheme for reducing decoherence in quantum computer memory". In: *Physical review A* 52.4 (1995), R2493.

[7] Ingo Wegener. *The Complexity of Boolean Functions*. Springer Berlin Heidelberg, 1987. ISBN: 0471915556.

[8] Bei Zeng, Andrew Cross, and Isaac L. Chuang. *Transversality versus Universality for Additive Quantum Codes*. 2007. eprint: 0706.1382 (quant-ph). URL: https://arxiv.org/abs/0706.1382.