Università degli Studi di Udine
Dipartimento di Scienze Matematiche, Informatiche e Fisiche
Corso di Laurea in Informatica

Bachelor Thesis

Models for Quantum Computing

Candidate	Supervisor
Riccardo Romanello	Prof. Carla Piazza

Academic Year 2019-2020

Institute Contacts
Dipartimento di Scienze Matematiche, Informatiche e Fisiche
Università degli Studi di Udine
Via delle Scienze, 206
33100 Udine — Italia
+39 0432 558400
http://www.dimi.uniud.it/

To all those who supported me and
to my little and loving nephew, *Ginevra*.

Acknowledgements

First of all, I want to thank my whole family for bearing me during this three years and for beleaving in me no matter what. Moreover, I would like to express my gratitude to all my friends: both the ones I have met in Udine and the ones I have known since childhood. Last, but not the least, my acknowledgements go to my supervisor, Professor Carla Piazza.

Abstract

We will consider Quantum Computation from a theoretical point of view, in particular as the natural evolution of classic and probabilistic computation. We aim to make the reader able to understand how QC works both in a physical and mathematical perspective.
As the Turing Machines have brought deterministic computation to a more practical level, at the same way Quantum Turing Machines have emerged for Quantum Computation. About this matter, we can observe how bonds between TM and QTM exist: these come from several factors, *i.e.* complexity theory and solvable problems. The study of QC introduces two new complexity classes: EQP and BQP. They are related with deterministic and probabilistic computational classes, making more complete and more precise the problems' classification based on execution time.
As a case of study, we introduce *quantum circuits* and based on this we give a possible implementation of Grover Algorithm to solve a *Quantum Search Problem*.

Contents

List of Figures

1 Introduction

Quantum Physics, as General Relativity, is not a really old field in physics. It was born in the 20s when Erwin Schrödinger, Werner Heisenberg, Max Born with their extraordinary theories gave big significance to the discovery that Max Planck made in the 1900 when he discovered *Quanta.* This theory gave us a different description of the subatomic world then the one we had: it asserted that in the 'quantum realm' there are totally different rules (so strange that even Albert Einstein didn't believed in them at the beginning).

This new field didn't gave an improvement only in physics: it has become really important even for computer science. The first one who conjectured a connection between Quantum Mechanics and Computer Science was Richard Feynmann: in [1] he said that classic computers are widely used by physicists to simulate physical experiments (like planet motions) and that Quantum Mechanics was almost impossible to simulate with a classic computer. This because of the large amount of data that are needed to remember all the possible states of a quanta: in fact, given a moment in time, a quanta is not in a precise position, but it can be in an infinite serie of positions where everyone of them has a probability. For this reason, to fully describe and simulate a quanta, we must keep in memory all the informations about its own *cloud of probability.* To achieve that, we require an enormous amount of data: Feynmann proposed to used directly quanta to store them. In this way, all the difficulties that classic computer would have encountered to store all the data could be overcomed by using the nature of quanta.

Starting from this conjecture, a whole new field of computer science was born: its main target was to create a Computer that, using the rules of quantum mechanics, could allow us to solve problems that nowadays are seen as 'untreatable'. The first who formally gave us a description of a Quantum Computers was D. Deutsch in [2]: his paper started a big revolution in Computation Theory, because he gave us a new way of looking at the *Church-Turing principle* making it non-ambiguous in a physical prespective. This change, made him able to describe a total new formalism he called *Quantum Turing Machine* or *QTM.*

Starting from [2], a lot of paper was written, but the most important was the one written by Ethan Bernestein and Umesh Vazirani. In [3], the two authors started by giving a whole new definition of QTMs, creating an easier model than Deutsch's one; they also gave definitions of complexity classes for Quantum Algorithms: EQP and BQP. The definitions they gave are still used to proove the relations between classic and quantum class of complexities.

The purpose of this thesis is to give a generic overview about Quantum Computing, allowing to the ones who want to start studiyng it to have a 'start point' in which are gathered all the basics concepts about this field. Therefore, this thesis is neither going to disprove some old papers nor going to improve the actual state of things in this field, but it just gives a generical base of Maths, Physics, Computer Science and everything that can be useful to approache this subject.

Thus, the structure of this thesis is the one that follows:

- **Chapter 2**: in this chapter it will given all the mathematical tools that are going to be useful during the whole document. The two main topic we will see are complex numbers and vector spaces.
- **Chapter 3**: here it will given a deeper introduction to the differences between a common bit and its counterpart in the quantum world, the Qubit.
- **Chapter 4**: this section is going to clearify which are the main differences between three kind of algorithms: classic, probabilistic and quantum ones.
- **Chapter 5**: before starting to talk about QTMs, it will explain the fundamentals of classic Turing Machines, having an introduction also to complexity theory.
- **Chapter 6**: this is the main chapter of the Thesis: here it will describe carefully the two models that have been explained in [2] and in [3].
- **Chapter 7**: as for classic TMs in chapter 5, we will talk about Complexity theory for QTM in this one.
- **Chapter 8**: in this part we will see how all the theorethical things we explained in all the chapter above can be actually tested in the real world using Quantum Circuits.
- **Chapter 9**: to finish, we will describe and then actually implement with both a circuit and a source code a quantum algorithm: *Grover Algorithm for Quantum search.* We will describe it both in a theoretical and practical way: to achieve this part we will use an Haskel library called *Quipper*

2

A Mathematical introduction

Before starting to explain the true subject of this thesis, let's have a little trip through the mathematics we are going to need during the whole document, such that I'm not going to stop on it anymore.

2.1 Complex numbers

2.1.1 Complex numbers forms

First of all, let's explain what is a complex number. Usually, everytime we talk about complex numbers, we think about the i. The i is, with π and e, one of the most important numbers in the whole math; it takes its authority because of this property:

$$i^2 = -1;$$

All the complex numbers are based on i and they have this form:

$$a + i \cdot b \text{ with } a, b \in \mathbb{R}$$

Where

- the real number a is also known as the real part
- the real number b is also known as the imaginary part

Given a complex number z, its real part is denoted by $\mathrm{Re}(z)$ while its imaginary part by $\mathrm{Im}(z)$

All the complex numbers are gathered togheter in a set named $\mathbb{C}$.

2.1.2 Complex numbers properties and arithmetic operations

Given a complex number $w = a + i \cdot b$, let's give the definition of w's *norm* and *complex conjugate*:

- **norm** : $|w| = \sqrt{a^2 + b^2}$
- **complex conjugate** : $w^* = a - i \cdot b$

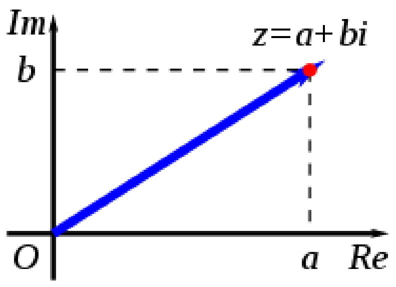


Figure 2.1: Cartesian form of a complex number

Regarding arithmetic operations, complex numbers behave a little different from real numbers or natural numbers. Given two complex numbers $z = a + i \cdot b$ and $w = c + i \cdot d$, then

$$z + w = (a + c) + (b \cdot d)i$$

that is, the sum of the real part is the real part of the sum and the sum of the imaginary part is the imaginary part of the sum. And,

$$z \cdot w = (ac - bd) + (bc + ad) \cdot i$$

where that minus sign is due to $i^2 = -1$.

2.1.3 Complex numbers representations

Complex numbers are usually seen as points (or vectors) in a two-dimensional Cartesian coordinate system where the x axis is used for the real part, while the y axis for the imaginary one.

If the complex number is described as above, referring to its real and imaginary part, then we say that it is described in a *cartesian* form, as in the figure 2.1.3

But, if we look at figure 2.1.3, we can see that a complex number could be expressed also by using the lenght of the vector it describes, and by the angle this vector forms with the x axis (the axis of the real). So, given a complex number $z = a + i \cdot b$, if φ is the angle that z forms with the x axis, and r is the norm of z, then the real part a can be described as $r \cdot \cos(\varphi)$ and the imaginary part is $r \cdot \sin(\varphi)$. So, rewriting $z = a + i \cdot b$ with the just found substitutions, we see that:

$$z = r(\cos(\varphi) + i \cdot \sin(\varphi))$$

(See image 2.1.3)

From this particular form and by using the Euler formula:

$$e^{i\varphi} = \cos(\varphi) + i \cdot \sin(\varphi)$$

we obtain a new representation for complex numbers that is:

$$z = a + i \cdot b = r \cdot e^{i\varphi}$$

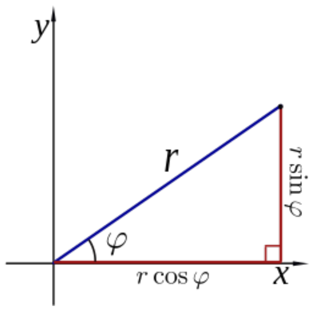


Figure 2.2: Polar form of a complex number

2.2 Vector Space

Roughly speaking, a Vector space is a set of elements, usually called *vectors*, together with two operations to manipulate these vectors.

1. *addition*: given two vectors v_1 and v_2 in a space vector V, the *addition* associates to their sum another vector v in V. Formally speaking,

$$+ : V \times V \to V$$

2. *scalar multiplication*: given a vector v in a space vector V and a complex number a (called *scalar*, and taken from $\mathbb{C}$ because we will always use vector spaces over $\mathbb{C}$) this operation associates to the product $a \cdot v$ another vector v_1 in V. Formally speaking,

$$\cdot : \mathbb{C} \times V \to V$$

The main 'operation' we are going to use in a Vector Space is the linear combination: given a series of n vectors inside a vector space (let's call them $v_1, v_2, \cdots, v_n$) and given n complex numbers (let's call them $\alpha_1, \alpha_2, \cdots, \alpha_n$) combining them creates a new vector w that has this form

$$w = \sum_{i=1}^{n} \alpha_i v_i$$

and w is also known as a linear combination of $v_1, v_2, \cdots, v_n$.

Given a vector space V and a set of k vectors $v_1, v_2, \cdots, v_k$ such that all the vectors in V can be expressed as a linear combination of $v_1, v_2, \cdots, v_k$ then we call $v_1, v_2, \cdots, v_k$ a 'base' of V. If the vectors $v_1, v_2, \cdots, v_k$ are made such that:

$$\forall i \in \{1, 2, \cdots, k\} \text{ the i-th component of } v_i \text{ is 1 and the other components are 0}$$

then $v_1, v_2, \cdots, v_k$ are a special base known as 'canonical base'.

Another operation we will use in this thesis is the *tensorial product*: given two column vector v and

w defined as follows:

$$v = \begin{pmatrix} v_1 \\ \cdots \\ v_n \end{pmatrix} w = \begin{pmatrix} w_1 \\ \cdots \\ w_n \end{pmatrix}$$

Then the tensorial product between v and w is written as $v \otimes w$ and it's defined as:

$$v \otimes w = \begin{pmatrix} v_1 \cdot \begin{pmatrix} w_1 \\ \cdots \\ w_n \end{pmatrix} \\ v_2 \cdot \begin{pmatrix} w_1 \\ \cdots \\ w_n \end{pmatrix} \\ \cdots \\ v_n \cdot \begin{pmatrix} w_1 \\ \cdots \\ w_n \end{pmatrix} \end{pmatrix}$$

2.2.1 Linear operators

Given two vector spaces V, W, we say that $f : V \to W$ is a *linear operator* if it satisfies the following:

1. $f(x + y) = f(x) + f(y), \forall x \in V, \forall y \in W$
2. $f(ax) = af(x), \forall x \in V, \forall a \in \mathbb{N}$

Each linear operator can be represented as a matrix. If the linear operator $f : V \to W$ is described with the matrix F, then, given a vector $v \in V$ we have that:

$$f(v) = F \cdot v$$

2.2.2 Hilbert Spaces

Let V denote a vector space over $\mathbb{C}$. An *inner space* over V is a complex function $(\cdot\ ,\ \cdot)$; defined on $V \times V \to \mathbb{C}$ which satisfies the following:

1. $\forall x, y, z \in V, (\alpha x + \beta y, z) = \alpha(x, z) + \beta(y, z)$
2. $\forall x \in V, (x, x) \geq 0$ Moreover, $(x, x) = 0 \leftrightarrow x = 0$
3. $\forall x, y \in V, (x, y) = (y, x)^*$

An *Inner product space* is a vector space V togheter with an inner product $(\cdot\ ,\ \cdot)$; An *Hilbert space* is an inner product space complete under the norm induced by $\|\cdot\|$. With 'complete under the norm' we mean that all the Cauchy sequences of vectors in V focus to a limit in V: this proprerty is meaningfull with infinite-dimension spaces, because with finite-dimension spaces it is always true. In quantum computing, all the vector spaces have a finite number of dimensions: so, for our purposes, the term 'Hilber Space' is equal to 'Inner product Space'.

2.2.3 The $\mathbb{C}^2$ space

The main vector space we are going to use during this thesis is the $\mathbb{C}^2$ space. It's the set of all the column vector with this form:

$$w = \begin{pmatrix} \alpha \\ \beta \end{pmatrix}$$

where $\alpha, \beta \in \mathbb{C}$.

Given a form as just seen above, we can obtain its *norm* and its *complex conjugate*:

- **norm** : $\|w\| = \sqrt{|a|^2 + |b|^2}$ (where the $|\cdot|$ is the absolute value of a complex number)
- **complex conjugate** : given w, its complex conjugate is the row vector $w^\dagger = (\alpha^*, \beta^*)$

Given two vectors

$$w_1 = \begin{pmatrix} \alpha_1 \\ \beta_1 \end{pmatrix} w_2 = \begin{pmatrix} \alpha_2 \\ \beta_2 \end{pmatrix}$$

both $\in \mathbb{C}^2$, we define *inner product* of w_1 and w_2 like this:

$$(w_1, w_2) = w_1^\dagger w_2 = (\alpha_1^*, \beta_1^*) \begin{pmatrix} \alpha_2 \\ \beta_2 \end{pmatrix} = \alpha_1^* \alpha_2 + \beta_1^* \beta_2$$

The $\mathbb{C}^2$ space with the inner product as defined above is also known as **two-dimnesional Hilbert Space**. The two main vectors of $\mathbb{C}^2$ we're going to use are these two:

$$\begin{pmatrix} 1 \\ 0 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

they are also known as 'canonical base' for the $\mathbb{C}^2$ space.

2.2.4 Dirac notation

To represent the elements of a complex vector space it's really suitable to use a notation called 'Dirac Notanion' (Named from the physicist who invented that). This is the standard notation in quantum mechanics, and it has few rules.

1. the generic column vector $v \in \mathbb{C}^2$ is written as $|v\rangle$ (it's read '*ket*').
2. the generic row vector $w = (\alpha_1^*, \alpha_2^*)$ is written as $\langle w|$ (it's read '*bra*')
3. the inner product between v and w is written as $\langle w|v\rangle$ (it's read '*braket*')
4. Given a linear operator U (on some vector space V) then applying U to a Qubit $|v\rangle$ is written as $U|v\rangle$
5. When using this notation: $|q_1 q_2 q_3 \cdots q_n\rangle$ then we will refer to the Qubit formed like this: $|q_1\rangle \otimes |q_2\rangle \otimes \cdots |q_n\rangle$. For example, writing $|00\rangle$ we refer to the Qubit $|0\rangle \otimes |0\rangle$.

2.2.5 Eigenvalues and Eigenvectors

An *eigenvector* of a linear operator L (on some vector space V) is a non-zero vector ($|v\rangle \in V$) such that:

$$L\,|v\rangle = v\,|v\rangle$$

where v is a complex number, also called *eigenvalue* of L for $|v\rangle$.

3 From bit to Qubit

In common computing every single information, algorithm, videogame, website is built from a very small base. The bit. A bit is a really simple structure that can be in two different state either 0 or 1 (or on/off, toggled/untoggled depending on the context). Every single data we store in a computer, smartphone or in the cloud is built from this simple structure. When 8 bit are gathered togheter and seen as a single thing, they are called a 'Byte', and they are used as mesure unit of the storage (every file we have has a weight, usually described in bytes). The fact that a bit can be in only two states, is sometimes a slowdown factor: let's imagine we have to guess an n-character length password where each char is either 0 or 1. Trying to guess that password takes us $O(2^n)$ attempts: we must generate all the possible n-character length sequences of 0s and 1s and then try them one by one. It's an incredibly long work (luckily for us, otherwise our passwords would be useless): but what if we can find some kind of structure that's in not just in a 0 OR 1 state, but that can be both 0 and 1 togheter, and it collapses in one of the two just when we measure it? In the very beginning of 20th century, a physicist named Max Planck, discovered that there is a lower bound to the amount of energy that a photon releases: a quanta. After this discovery, there was other: the wave-particle behaviour, the uncertanty principle, ecc and all of this created a new field of physics that, in honor to Planck and its milestone discovery, was named 'Quantum mechanincs'. Quantum mechanics nowadays is the best model we have to describe the world of the endlessly small, even if it has some problem to be united with the other main theory of our universe: theory of relativity, that describes the behaviour of endlessly big things. But, what we concerne the most about quantum mechanics is the uncertainty principle: we cannot know both position and speed of a particle, but just one. (an easy experiment can show that empirically but this is no physics paper). This principle allows us to say that 'we can know where a particle is but not how fast it goes, and viceversa'. One other conseguence of the uncertainty principle is that, if we measure the speed of a particle, we loose every information about its location, and viceversa. If we apply the uncertainty principle to a quantum system with two states (0 or 1), we create a new idea of bit, the *quantum bit*, aka *qubit*.

The *Qubit* can be seen as a switch that is neither on nor off: it's both of them togheter, until we do not look at it (this 'look' will become 'measure' whene we are going to get deeper in the subject). In fact, a Qubit is both 0 and 1 until we do not measure it: but, when we do that, how it's decided if we get 0 or 1? Well, to answer this question we must say that all Quantum Mechanics is based in some solid principles:

- Probability
- Quantum Interference: we will talk about it in the next chapter
- Quantum Entaglement: we will see it in chapter 8

The entire evolution of a quantum system is not described as a sure list of events, but it's like 'with this probability α we will get this result': so, also reading the value of a Qubit is driven by probability. But, let's give a Qubit its mathematical description to formalize those concepts.

3.1 A mathematical description of Qubit

So, as we just saw, a Qubit is an abstract mathematical object that has some properties, first of all the fact that, until we do not measure it, it has both 0 and 1 value. But how can we describe it, in a mathematical way?

We told, in the previous chapter that there is an operation in vector space called 'linear combination': we can use it to make the state 0 and 1 live togheter. The last thing we have to decide is how to describe the state 0 and the state 1: we will use the canocical base of $\mathbb{C}^2$

$$\begin{pmatrix} 1 \\ 0 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

The first to the left describes the 0, and using the dirac's notation is going to be called $|0\rangle$, while the rightmost is the 1, and we'll call it $|1\rangle$.

Defined the 0 and the 1 for quantum model, we can define a generic Qubit as a their linear combination:

$$|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle = \alpha \begin{pmatrix} 1 \\ 0 \end{pmatrix} + \beta \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

where α and β are two complex numbers such that:

$$|\alpha|^2 + |\beta|^2 = 1$$

Geometrically, this means that the states of a Qubit are normalized vector which length one: this is a main feature, because each operation applied to a Qubit must preserve this property. (we may refer to this concept as *preserving Euclidean Length*)

Now that we have seen this kind of notation we can introduce the last (but not the least) concept of Qubit: the *superpostion*. In fact, the fact that a Qubit is both in 0 and 1 state togheter is usually sad as 'the Qubit is in a state of superpostion of 0 and 1'.

Introducing this notation allows us to answer to another question we left: a Qubit is a superpostion, but when do we get 0 and when 1 after measuring a Qubit? The answer is inside the two values α, β that we used before to describe the generic Qubit. In fact, they are not simple complex numbers but they tell us that, given a generic qubit as defined above, after we measure it we get the state $|0\rangle$ with a $|\alpha|^2$ probability, and we get $|1\rangle$ with a $|\beta|^2$ probability. For this particular reason, α and β are called *probability amplitude* and it must be true that $|\alpha|^2 + |\beta|^2 = 1$

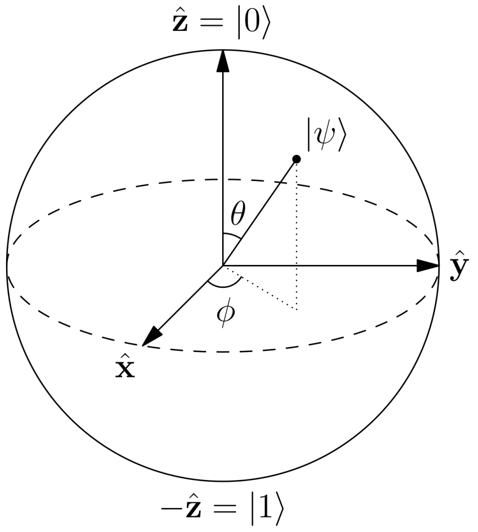


Figure 3.1: Each point in the surface of the Bloch Sphere is a correct state for a Qubit

To finish this little introduction about Qubit we can say that, a single Qubit can be in a number of state that is infinitely bigger than the two state in which a bit can be: we know that each 'correct' state for a Qubit is a superposition

$$\alpha\,|0\rangle + \beta\,|1\rangle \;\text{ with } \alpha, \beta \in \mathbb{C}$$

where must be true that $|\alpha|^2 + |\beta|^2 = 1$. Let's assume

$$\alpha = a + i \cdot b$$
$$\beta = c + i \cdot d$$

if we take a look to the formula of *norm* we see that we can rewrite the left part of the condition about $\alpha \& \beta$.

$$\begin{aligned} |\alpha|^2 + |\beta|^2 &= |a + i \cdot b|^2 + |c + i \cdot d|^2 \\ &= \left(\sqrt{(a^2 + b^2)}\right)^2 + \left(\sqrt{(c^2 + d^2)}\right)^2 \\ &= a^2 + b^2 + c^2 + d^2 \end{aligned}$$

And then, the new condition we obtain is:

$$a^2 + b^2 + c^2 + d^2 = 1$$

This particular equation describes a particular sphere that in Quantum Mechanics is called *Bloch Sphere* (figure 3.1) : it's true then that, each point in the surface of a Bloch sphere is a correct state for a Qubit.

Even if a Qubit can be in a number of state that is infinitely bigger than a bit, when we measure a Qubit we are always going to get only one result: either 0 or 1, because of the natural behaviour of quanta.

4 Classic, Probabilistic and Quantum Algorithms

Before defining the differences between a classic, a Probabilistic and a quantum algorithm, let's see what is an algorithm. There are plenty of definitions, but the one we will use is:

An algorithm is a finite serie of instructions that, given some data in input, it transforms them in a output result

So, an algorithm is just like a function that, starting from a domain (input data), gives us a value inside a codomain (output data). The easiest algorithm is the one that creates the identity function: given a number $x \in \{0, 1\}$ it gives us the value of x as result.

$$f : \{0, 1\} \to \{0, 1\}$$

We can think that any function can be computed with algorithm: actually, it has been proved in 1936 that not all the functions that we can think can be computed with an algorithm: for example, there is no algorithm that takes in input one other algorithm and tells us if the algorithm ends or no. We'll see that the set of the *computable* functions has well known bounds.

But, how many types of algorithm can we write? Before Quantum Computing there were two types:

1. **Deterministic algorithms**: the algorithm has one 'way' to solve a problem and the solution is correct in 100% of cases. A classic algorithm follows a finite sequence of instructions that gives us the result. If we describe every instruction of the algorithm as a vertex, the execution of classic algorithm could be plotted as a one-way connected list of vertex: every vertex has one incoming edge (except for the Input vertex) and one outcoming edge (except for the Output Vertex)

2. **Probabilistic algorithms**: the algorithm has many paths it can follow, and each possible path has a probability to be followed. The result is not always right, we will see when a Probabilistic algorithm is defined as 'good' and when no. As before, let's imagine every instruction of the algorithm as a vertex but in this case, every vertex is not connected to just one another vertex, but it can be connected to many. So, each vertex can have many incoming edges (except for the input vertex) and can have many outcoming edges (except for the output vertex). Each vertex

has a weight: if the vertex e goes from u to v, then the weight of e is the probability that, once the algorithm has executed the instruction described by node u, it executes, as next operation the instruction decribed by node v. And so, referring to a classic mathematical structure, the execution of a Probabilistic algorithm can be seen as a weighted graph.

When talking about a classic algorithm we know that, if we reach a result, it is correct in the 100% of cases, in the case of a Probabilistic algorithm the probability that we get the right result is given by probaility theory.

Let's take a generic Probabilistic algorithm, let's call it A. This algorithm has n different paths that it can take, but just m of them give us a correct result. Each of these paths is made by some edges, where each edge has a weight
We can know extend the definition of weigth: we will call *weight of a path* the product of the weights of all the edges that form the path (this is just the probability of following the entire path edge by edge). Given a path P, we will refer to its weight as $W(P)$.
So, we know that, in the A algorithm we have m paths that, during the execution can lead us to a correct output. What is the probability that we *actually* get a correct output? Let's enumerate all the m paths as $P_1, P_2, ..., P_m$, then the probability that the algorithm A gives us a correct answer is given by this formula:

$$\sum_{i=1}^{m} W(P_i) \tag{4.1}$$

It may look trivial, but, if at least one of the m path has a non-zero weight, then we get a non-zero probability of having the right output. Obviosly, before things gets creepy, it's important to say that:

In a probabilistic algorithm, there are many paths that can be followed, but during the execution just one of them is followed

Now that we introduced classic and probabilistic algorithm, let's see which changes quantum computing has brought to this matter. The execution of a Quantum algorithm can be seen (graphically) as the execution of a probabilistic one (in what concern probabilities, edges and vertex). But, there are some differences:

1. the weight of each edge is not a probability but it's a *probability amplitude* and, as we saw before, probability amplitudes are complex numbers. So, the function that associates to each edge a weight has a new codomine: $\mathbb{C}$.
   We will see that this change of codomine will give us an unexpected result because of minus sign that products between complex numbers generate.

2. while the *weight of a path* is always calculated as a product (in this case of probability amplitudes instead of simple probabilities) but the probability that we got with 4.1 changes its formula (we use the same hypotesis as for 4.1) and becomes:

$$\left| \sum_{i=1}^{m} W(P_i) \right|^2 \tag{4.2}$$

3. Last (but not least) difference is that, whereas in a probabilistic algorithm, between all the paths only one is actually followed, in a quantum algorithm (because of quantum mechanincs rules) all the paths are followed and together executed: just in the end, when we watch the result we make the system collapse in one single result, but until we don't measure it all the possible results exist together!

.

Maybe, those three differences, taken one by one, could make you think that, beside some maths, there is no difference between a probabilistic and a quantum algorithm: but take a look to this 3 togheter. The consequence is a strange phenomena called 'quantum interference': while in a probabilistic algorithm if even only one of the probabilities P_i in 4.1 is non-zero then we have a non-zero probability to get the right output, in a quantum algorithm that's not true! and the reason is that sometimes the probabilities in 4.2 cancel out each other. This is due to the fact that for quantum algorithm we do not talk about simple probabilities, but we are using complex numbers and, multiply complex numbers generate minus signs that make separated paths to intefere with others. So, we can get a quantum algorithm where we have two paths (P_1, P_2) with non-zero probability to get to the output $(W(P_1), W(P_2) > 0)$, but 4.2 gives us a 0 result. This, because we have

$$W(P_1) = c$$

And

$$W(P_2) = -c$$

and if we put $W(P_1), W(P_2)$ in 4.2 we get

$$\begin{aligned}\left|\sum_{i=1}^{m} W(P_i)\right|^2 &= \text{with } m = 2\\ &= \left|W(P_1) + W(P_2)\right|^2\\ &= \left|c + (-c)\right|^2\\ &= |0|^2\\ &= 0\end{aligned}$$

Let's give an example to better understand this critical concept. Looking at picture 4.1, we can see a simple quantum machine with its *probability amplitudes.* We will use this notation:

c_{ij} is the probability amplitude to get to output j with input i.

In this case we have:

$$c_{00} = i/2, c_{01} = 1/2, c_{10} = 1/2, c_{11} = i/2$$

In this particular case, the machine at picture 4.1 is also known as *Random bit* because given in input a bit, it gives us in output a bit that is 0 or 1 with the same probability, so it's an example of true randomness in the world of Computer Science (true randomness cannot be achieved in classic computation).

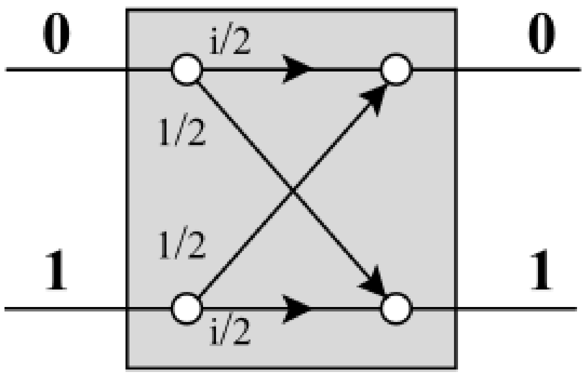


Figure 4.1: A simple Quantum machine decribed as a graph

So, let's imagine to put two copies of this machine in serie (connecting the output of the first as input for the second). Now, we want to compute the probability of getting a 0 output giving 0 as input in this new machine. If we look at it, we have two paths that can lead us to the wanted result:

1. $0 \to 0 \to 0$, we will call it P_1
2. $0 \to 1 \to 0$, we will call it P_2

So, the m we are gonna use for (4.2) is 2 (becase of two good paths).
Let's calculate P_1 and P_2.
P_1 is made of two edges: the one that brings from 0 to 0, and one other from 0 to 0, so, it's *weight* is

$$W(P_1) = c_{00} \cdot c_{00} = i/2 \cdot i/2 = -1/4$$

P_2 is made of two edges too: the one that brings from 0 to 1, and then the one that brings from 1 to 0

$$W(P_2) = c_{01} \cdot c_{10} = 1/2 \cdot 1/2 = 1/4$$

Now, applying (4.2) we get that, the probability of going from 0 input to 0 output is

$$\begin{aligned}\left|\sum_{i=1}^{m} W(P_i)\right|^2 &= \text{with } m = 2 \\ &= \left|W(P_1) + W(P_2)\right|^2 \\ &= \left|-1/4 + 1/4\right|^2 \\ &= |0|^2 \\ &= 0\end{aligned}$$

And, as we wanted to show, there are case in which, non-zero weights for single paths bring us to zero probability after applying (4.2).

5

Turing Machines

In 1936, Alan Turing (1912-1954), published a paper that will start a whole new field in mathematics: computer science. In this article, Alan Turing gave us a definition of a brand new mathematical model of computation: the Turing Machine. Turing's idea was to create a machine, with an infinite tape (divided in *cells*) where the machine can *read* or *write* symbols (where each symbol is taken from an *alphabet*). The way the machine uses to interact with the tape is a *head*: every moment, the head can read/write only one symbol in one cell of the tape. The 'brain' of the Machine is a *FSA* (Finite State Automata), that decides in every moment if:

- the head has to write a new symbol in the cell
- the head has to move one cell to left or to the right
- the FSA has to change its internal state

Inside the alphabet of the Machine there must a be a symbol called *blank* (we will use the #) that is used to describe which cells of the tape are empty, or where the input begins/ends, ecc
From a mathematical point of view, a TM (Turing Machines) is a triplet (Σ, Q, δ) where:

- Σ is the finite alhabet $\{s_1, s_2, ..., s_n\}$ where there is at least one symbol s_i such that $s_i = \#$
- Q is the FSA's finite set of states. There must be one state q_0 called *initial state* and one state q_f called *final state.*
- δ is the *Transiction Function*:
  $$\delta : Q \times \Sigma \rightarrow \Sigma \times Q \times L, R$$
  A function that, taken as input a couple containing: the FSA current state and the current read symbol, gives us a triplet that contains: the new FSA state, the symbol that has to be written on the tape and the direction in which the head must move (L = Left, R = Right).

From the idea of Turing Machine, Turing developed a 'special' Turing Machine called *Universal Turing machine.* Taken for granted a sort for the Turing Machines, and known that each Turing Machine can be described with an index (in $\mathbb{N}$), we define the Universal Turing machine U as a Turing Machine that takes two parameters:

1. the index m of a Turing Machine
2. a number n

and U, taken the index m, generates the corresponding Turing Machine and executes it on the input n.

Introduced the idea of Universal Turing Machine, we can give the limitations of what is computable and what is not, according to Turing. He said that, the bounds of what can be computed are not imposed neither by the design of the computing machines nor by our ingenuity in constructing models for computaion, but are universal. This is called 'Church - Turing hypotesis' (because both Turing and Church got to the same result together but in two different ways):

Every 'function which would naturally be regarded as computable' cane be computed by the universal Turing machine

5.1 Complexity Theory

Now that we have defined the definition of a Turing Machine, we can use it to introduce the idea of *Complexity Theory.* Before introducing the definition of Time and Space complexity, we have to say that each problem P that can be solved with a TM can be seen as a *Language*: this *Language* is the set of all the solutions to P. For example, if P is 'Given a number x find if x is prime': then the *language* of P is the set of all the prime numbers.

Definition 1. *Given a TM M and an input x we say the time needed to M for x is the number of steps that M needs, with input x to halt. Let $f : \mathbb{N} \to \mathbb{N}$ be a total function. We say that a TM M works in time $f(n)$ if for each input x the time needed to M for x is less or equal to $f(|x|)$.*

Definition 2. *A language $\mathscr{L} \in \Sigma^*$ is decided by a TM M if for each $x \in \Sigma^*$ satisfies the follows:*

- *if $x \in \mathscr{L}$ then M with input x ends with ouput 1*
- *if $x \notin \mathscr{L}$ then M with input x ends with ouput 0*

If $\mathscr{L}$ is decided by a TM M and M *works in time* $f(n)$ then $\mathscr{L} \in \text{TIME}(f(n))$

$\text{TIME}(f(n))$ as defined above is a *Time complexity class.*

With these definitions, we can now introduce on the most important complexity classes: **P**. It is also known as the class of the problem easily solvable, and is defined as:

Definition 3. *A language $\mathscr{L}$ is in **P** iff exists a TM M such that:*

- *M works in polynomial time for all the inputs*
- *$\forall x \in \mathscr{L}$ M halts with ouput 1*
- *$\forall x \notin \mathscr{L}$ M halts with output 0*

To introduce the class **NP**, we first have to define wath a *Non-Deterministic* TM is.

Definition 4. *Given a TM $M = (Q, \Sigma, q_0, P)$ we say that it is a Non-Deterministic TM (ND-TM) if P is a relation:*

$$P \subseteq (Q \times \Sigma) \times ((Q \cup \{h, 1, 0\}) \times \Sigma \times \{L, R, F\})$$

Barely, every state has not only one successor, but it has many. (all the other definitions we gave for Deterministic TMs are good also for Non-deterministic ones).

Definition 5. *A language $\mathscr{L}$ is in **NP** iff exists a TM M such that:*

- *M works is polynomial time for all the inputs*
- *$\forall x \in \mathscr{L}$ exists one computation of M with input x that halts with output 1*
- *$\forall x \notin \mathscr{L}$ all the computation of M with input x halts with output 0*

6

Quantum Turing Machine

6.1 Overcoming the Turing Machine capabilities

The Turing model of computation and algorithm is, still today, the model at the base of every device we use: smartphones, pc, tablets, ecc But, in the last decade, all the big companies like Facebook, Google, ecc has started to invest a lot of money on a new field of research: creating a quantum computer. Actually, quantum computers already exists: both Google, and IBM has created one, but these computers can do exactly wath a common computer can do, at the same speed, giving no improvements. The true challenge is to create a quantum computer that, actually taking advantage of the QM's laws, could give to our computer an exponential speed up. But, if the Computer as we know it, is based on Turing Machine, then also the Quantum Computers are based on something similar called *Quantum Turing Machine.*

The first physicist who sayd that a Quantum Computer was able to simulate things that a normal computer could not was Richard Feynmann. In a lecture, Feynmann described how physicists use computers to do simulations: sometimes they want to do tests before trying the experiment for real, or maybe they don't have access to the system so they fake it using computers simulation. But, can a normal computer simulate every physical system? Feynmann supposed that the amount of memory and computing capability may be enough to simulate planets, or the motion of galaxies: but what about quantum systems? When talking about quantum realm, the main feature is the *Superposition Principle*: for example, a quantum system of electrons or protons can exist in a superpostion of observable states before the measure. So we must keep track of all the probabilities: usually, with a classical computers, to keep track of n particles in a quantum system we need an amount of memory that is in the order of 2^n and these number become really big in a really short time. The solution Feynmann gave was that, if the problem is to simulate quantum systems, why don't we use them as computing power? Qubits can keep track of the probabilities better than normal bits, because it's their nature to behave like quantum particles. So, if we suppose that each Qubit keep track of the probability of one quantum particle in the quantum system we described before, then we should need just n Qubit (one per particle).

So, theorical physicists and computer scientists started to work on a model of Turing Machines but based on Quantum Mechanics: *Quantum Turing Machine*

6.2 The first complete model: Deutsch's QTM

In 1985, David Deutsch published an article in which he gave the first formalization of a fully operative Quantum Turing Machine. He started by saying that the conventional non-physical view of the Church Turing hypotesis interprets it as the quasi-mathematical conjecture that all possible formalizations of the intuitive mathematical notion of 'algorithm' and 'computation' are equivalent to each other. But, looking at the Church-Turing hypotesis in a physical way, made him create something he called 'Church-Turing *principle*'. He suggested a new way of defining Turing's 'functions which would naturally be computed', as the functions which may in be in principle computed by a real physical system. After that definition, he gave us the definition of '*perfect simulation*':

a computing machine $\mathscr{M}$ is capable of perfectly simulating a physical system $\mathscr{L}$, under giving labelling of their inputs and outputs (using same rules to give labels to input and output) if there exists a program $\pi(\mathscr{L})$ for $\mathscr{M}$ that renders $\mathscr{M}$ computionally equivalent to $\mathscr{L}$ under that labelling.

In other words, what we just sayid can be seen as '$\pi(\mathscr{L})$ converts $\mathscr{M}$ into a black box functionally indistinguishable from $\mathscr{L}$. And then, after giving this definition, he finally defined the *'Church-Turing Principle'*:

Every finitely realizable physical system can be perfectly simulated by a universal model computing machine operating by finite means.

In this way, the Church-Turing principle refers only to terms like 'measurement', 'preparation' and 'physical system' which are already defined in measurement theory, insted of using terminology like 'would naturally be regarded' that does not fit inside the world of physics.

One thing we do not have to forget is that, each operation that is appliead to a Qubit must preserve the unitary feature:

One of the main things that made Deutcsh's model different from all the others is that, according to him, there is no a *priori* reason why physical laws should respect the limitations of the mathematical process we call 'algorithm'.

6.2.1 The QTM

As a Turing Machine, a Quantum Turing Machine consists of two components:

1. a finite *processor*
2. an infinite *memory* (which only a portion is ever used)

Every step of the computation has a fixed duration T and during each step only the processor a finite part of the memory interact.

Before defining what the processor and the memory actually are, let's define what's an observable: in physics, an observable is any kind of size that can be measured directly or analytically.

Then, the *processor* of the QTM consists of M 2-state observable:

$$\{\hat{n}_i\}\ \{i \in \mathbb{N}_M\}$$

where $\mathbb{N}_M$ is the set of all integers from 0 to M - 1.
The memory consists of an infinite sequence

$$\{\hat{m}_i\}\ \{i \in \mathbb{N}\}$$

of 2-state observable (it's the equivalent of Turing Machine's infinite tape). We will refer to the set of $\hat{n}_i$ as $\mathbf{\hat{n}}$ and to set of $\hat{m}_i$ as $\mathbf{\hat{m}}$

Corresponding to Turing's 'tape position', there is another observable $\hat{x}$ which has the whole $\mathbb{N}$ has its spectrum. The observable $\hat{x}$ is the 'address' number of the currently scanned tape location.
Thus, the state of a QTM is a unit vector in the space $\mathscr{H}$:

$$|x; \boldsymbol{n}; \boldsymbol{m}\rangle = |x; n_0, n_1, ..., n_{M-1}; ...m_{-1}, m_0, m1, ...\rangle \tag{6.1}$$

of $\hat{x}$, $\mathbf{\hat{n}}$, $\mathbf{\hat{m}}$, labelled by the corresponding eigenvalues x, $\boldsymbol{n}$, $\boldsymbol{m}$.

We will refer to 6.1 as the *'Computational basis states'*.
The dynamics of a QTM is defined by an unitary operator U on $\mathscr{H}$: U specifies the evolution of any state $|\psi(t)\rangle \in \mathscr{H}$ during a single computation step:

$$|\psi(nT)\rangle = U^n\,|\psi(0)\rangle\ \ (n \in \mathbb{N}^+)$$

At time $t = 0$, $\hat{x}$ and $\mathbf{\hat{n}}$ are prepared with the value zero, while the state of a finite number of cells in $\mathbf{\hat{m}}$ is prepared as the 'program' and the 'input'. (Obviosly, the initial state and all the states in which the QTM is have to respect the unitary property of Qubits).

Turing machines are said to *'halt'* when two consecutively states are identical. A 'valid' program is one that causes the machine to halt after a finite number of steps. But, because of Quantum Mechanics rules, a QTM must not be observed before computation has ended since this would alterate the QTM state: therefore, QTMs need to signal actively that they have halted. For this purpose, on of the processor's internal bits, let's say $\hat{n_0}$, must be set for this purpose. Every valid program for a QTM set $\hat{n_0}$ to 1 when it terminates but do not interact with $\hat{n_0}$ anymore: in this way, $\hat{n_0}$ can be periodically checked without affecting the QTM's operator.

6.3 After the model of Deutsch: Bernstein and Vazirani

After Deutsch defined a model of QTM's, a lot of papers has improved his work. One of the most important that came out was the Bernstein and Vazirani one.

6.3.1 The QTM

Before defining the QTM in this new way, we shall define the set $\tilde{C}$ of $\alpha \in \mathbb{C}$ such that there is a deterministic algorithm that computes the real and imaginary parts of α to within 2^{-n} in time polynomial in n.

Then, a *QTM M* is defined by a triplet (Σ, Q, δ), where

- Σ is a finite alphabet with a 'blank symbol' #.
- Q is a finite set of states with an initial state q_0 and a final state q_1 (with $q_0 \neq q_1$).
- δ, the *quantum transiction function*, is a function:

$$\delta : Q \times \Sigma \rightarrow \tilde{\mathbf{C}}^{\Sigma \times Q \times \{L,R\}}$$

  To be clear, giving in input to δ a state $q \in Q$ and a symbol $s \in \Sigma$, it gives us a function g. By giving in input to g a state $q^{'} \in Q$, a symbol $s^{'} \in \Sigma$ and direction $d \in \{L, R\}$ it gives us a complex number $\alpha \in \tilde{\mathbf{C}}$ such that α describes the probability amplitude that from the configuration $\{q, s\}$ we get into the configuration $\{q^{'}, s^{'}, d\}$

Let S be the inner-product space of finite complex linear combinations of connfigurations of M. We call each element $\phi \in S$ a *superposition* of M.
The time evolution of the QTM M, is a linear operator $U_M : S \rightarrow S$, such that:
if M starts in configuration c, with current state q and scanned symbol s, then after one step M will be in a superposition of configurations:

$$\psi = \sum_i \alpha_i c_i$$

where:

- each non-zero α_i corresponds to a transition $\delta(q, s)(q^{'}, s^{'}, d)$
- each c_i is the new configuration we get after applying δ to c (the new couple $(state, symbol)$ we obtain, AKA $q^{'}, s^{'}$)

Expanding this definition to all S, then we get the linear operator U_M.

6.3.2 Properties of a QTM

We will give now a serie of definitions (and properties) that will be useful in the next section when we will talk about the programming primitives for QTMs.

Definition 6 (Well formed QTM). *a QTM M is well formed if its evuolution operator U_M preservers the unitary property of the Qubits.*

In other words, the time evolution operator of a QTM must satisfy the condition that in each successive superposition, *the sum of the probabilities of all possible configurations must be 1*

Definition 7 (Well behaved QTM). *A QTM M is called well behaved if it halts on all input strings in a final superposition where each configuration has the tape head in the same cell.*
If this cell is always the start, then we call the QTM stationary.

Definition 8 (Unidirectional QTM). *A QTM M is called Unidirectional if each state can be entered from only one direction: in other words if $\delta(p_1, \sigma_1, \tau_1, q, d_1)$ and $\delta(p_2, \sigma_2, \tau_2, q, d_2)$ are both nonzero, then $d_1 = d_2$*

Definition 9 (Normal form QTM). *A QTM $M = (\Sigma, Q, \delta)$ is in normal form iff:*

$$\forall \sigma \in \Sigma \ \delta(q_f, \sigma) = |\sigma\rangle |q_0\rangle |R\rangle$$

We can also define that it is normal form if, no matter which symbol δ take in input, if the input state is the $final state$, then the QTM always do these three things:

1. *the symbol remain unchanged (is always σ)*
2. *the state becomes the $initial state q_0$*
3. *the tape head is always moving to the right (R)*

Definition 10 (Reversible TM). *A reversible TM is a deterministic TM for which each configuration has at most one predecessor.*

Now, we can tie the definition of reversible TM with, QTMs using this theorem:

Theorem 1. *Any reversible TM is also a well-formed QTM*

Proof. The transiction function δ of a deterministic TM maps the current state and symbol to an update triple. If we think to it as instead giving the unit superposition with amplitude 1 for that triple and 0 elsewhere, then δ is also a quantum transition function and we have a QTM.
The time evolution matrix corresponding to this QTM contains only entries 1 and 0: if the TM is reversible then there must be at most one 1 in each row. Therefore, any superposition of configurations $\sum_i \alpha_i |c_i\rangle$ is mapped by this time evolution matrix to some other superposition of configuratioons $\sum_i \alpha_i \left|c_i^{'}\right\rangle$.
So, the time evolution preserves the length, and the QTM is well formed. □

Measuring a QTM

When a QTM M is in a superposition

$$\psi = \sum_i \alpha_i c_i$$

and it is observed or measured, a generic configuration c_i is seen with probability $|\alpha_i|^2$.
Let's remember that, for $Wellformedness$ property, it must be true that:

$$\sum_i |\alpha_i|^2 = 1$$

When a QTM halts

A *Final Configuration* of a QTM M is any configuration in state q_f.
If when QTM M is run with input x, at the time T the superposition

$$\psi = \sum_i \alpha_i c_i$$

contains only final configurations, and at any time less than then T the superposition contains no final configuration, then M *halts* with running time T on input x.
The superposition of M at time T is called the $final superposition$ of M run on input x. A *polynomial−time* QTM is a well formed QTM which on every input x halts in time polynomial in the length of x.

6.3.3 Programming Primitives for QTMs

In these section, we will see 4 programming primitives that allow us to create more powerful QTMs; theire names are:

1. **Dovetail**
2. **Branch**
3. **Reverse**
4. **Loop**

Lemma 1 (Dovetailing lemma)**.** *If M_1 and M_2 are well-behaved, normal form QTMs (reversible TMs) with the same alphabet, then there is a normal form QTM (reversible TM) M which carries out the computation of M_1 followed by the computation of M_2.*

Lemma 2 (Branching Lemma)**.** *If M_1 and M_2 are well-behaved, normal form QTMs with the same alphabet, then there is a well-behaved, normal form QTM M such that if the second track is empty, M runs M_1 on its first track and leaves it second track empty, and if the second track has a 1 in the start cell (and all other cells blank), M runs M_2 on its first track and leaves the 1 where its tape head ends up. In either case, M takes exactly foru more time steps than the approriate M_i.*

Lemma 3 (Reversal lemma)**.** *If M is a normal form, Unidirectional QTM, then there is a normal form QTM M' that reverses the computation of M while taking two extra time steps.*

Lemma 4 (Looping lemma)**.** *There is a stationary, normal form, reversible TM M and a costant c with the following properties.*
On input any positive integer k written in binary, M runs for time $O(k \log^c k)$ and halts with its tape unchanged. Moreover, M has a special state q^* such that on input k, M visits state q^* exaclty k times, each time with its tape head back in the start cell.

7 Complexity for QTMs

7.1 BPP

Giving for good the definitions of P, NP, PSpace, NPSpace, then we shall define the class of problems known as BPP. A language L is in BPP iff exists a probabilistic TM M such that:

- M works in polynomial time in all the inputs
- $\forall x \in L$, then M accepts x with probability bigger than 2/3
- $\forall x \notin L$, then M accepts x with probabilty smaller than 1/3

It's not known if $P = BPP$

7.2 New classes for Quantum computing: EQP and BQP

7.2.1 Accepting a language in QTM

Before defining the new complexity class, let's define what, for a QTM 'accepts' means.

Let M be a stationary, normal form, multitrack (k different tapes to write/read) whose last track has alphabet $\{\#, 0, 1\}$. If we run M with string x on the first track and the empty string elsewhere, wait until M halts, and then we observe the start cell of the last track, we will see a 1 with some probability p.
We will say that M accepts x with probability p and rejects x with probability 1 - p.

We will say that M *exaclty accepts* a language L if M accepts every string $x \in L$ with probability 1 and rejects every string $x \notin L$ with probability 1.

We can now define the first class of complexity for Quantum Computing: *EQP*

7.2.2 EQP

We define the class **EQP** (error-free quantum polynomial time) as:

the set of languages which are exactly accepted by some polynomial time QTM

More generally we can define the class **EQTime**$T(n)$ as the set of languages which are exactly accepted by some QTM whose running time on any input of length n is bounded by $T(n)$ This class, is the equivalent of P for QTMs.

7.2.3 BQP

We say that a QTM M *accepts* a language L with probability p if M accepts with probability at least p every string $x \in L$ and rejects with probability at least p every string $x \notin L$ (The last part can be rewritten as 'accepts with probability at least 1-p all the strings $x \in L$').

Then, we define the class **BQP** as:

the set of languages that are accepted with probability 2/3 by some polynomial time QTM.

More generally we can define the class **BQTime**$T(n)$ as the set of languages that are accepted with probability 2/3 by some QTM whose running time on any input of length n is bounded by $T(n)$.

7.2.4 Results on these complexities

The results we get in positioning these two new classes inside the already defined hierarchy are:

- $\mathbf{P} \subseteq \mathbf{EQP}$
- $\mathbf{BPP} \subseteq \mathbf{BQP}$
- $\mathbf{BQP} \subseteq \mathbf{PSPACE}$

Of the three relations we wrote above, we will give a proof of the second one (using the formalism described in [3]:

Theorem 2. *BPP* $\subseteq$ *BQP*

Proof. Let $\mathscr{L}$ be a language in **BPP**. Then there must be a polynomial $p(n)$ and a polynomial time deterministic TM M with 0,1 output which satisfies the following: for any string x of length n, if we call S_x the set of $2^{p(n)}$ bits computed by M on input $x; y$ with $y \in \{0,1\}^{p(n)}$, then the proportion of 1's in S_x is at least $\frac{2}{3}$ when $x \in \mathscr{L}$ and at most $\frac{1}{3}$ otherwise.
We can use a QTM to decide whether a string x is in the language $\mathscr{L}$ by first breaking into a superposition split equally among all $|x\rangle |y\rangle$ and then running this deterministic algorithm. First we dovetail a stationary, normal form QTM that takes in input x to output $x; 0^{p(n)}$ with a stationary, normal form QTM that applies a Fourier transform to the content of its second track. This gives us a stationary, normal form QTM which on input x produces the superposition:

$$\sum_{y \in \{0,1\}^{p(n)}} \frac{1}{2^{p(n)/2}} |x\rangle |y\rangle$$

Then, by dovetailing this with a normal form of M we get a polynomial time QTM that on input x produces a final superposition

$$\sum_{y \in \{0,1\}^{p(n)}} \frac{1}{2^{p(n)/2}} |x\rangle |y\rangle \big|M(x; y)\big\rangle$$

Since the portion of 1's in S_x is at least $\frac{2}{3}$ if $x \in \mathscr{L}$ and at most $\frac{1}{3}$ otherwise, observing the bit on the third track will give the proper classification for string x with probability at least $\frac{2}{3}$.
Therefore, this is a **BQP** machine accepting the language $\mathscr{L}$ $\square$

8

Quantum Circuits

Similarly to classic computers, a Quantum Computer is made of quantum circuits built from elementar quantum logic gates. In the classic case, there exists only one (non-trivial) one bit gate: the NOT gate that implements the logic operation of negation through a truth table in which $0 \to 1$ and $1 \to 0$. To define the same operation on a Qubit, we cannot define its behaviour only on the states $|0\rangle$ and $|1\rangle$ but we must specify how a Qubit that is in superposition is affected by the gate. The NOT gate should transform a general Qubit in this way:

$$(NOT)(\alpha|0\rangle + \beta|1\rangle) = \beta|0\rangle + \alpha|1\rangle$$

We will see that the operation that implements this kind of transformation is a *linear operator*. In Quantum Circuits, the best way to describe quantum gates is through matrices.

8.1 One Qubit Quantum gates: X, Y, Z

The matrix that describes the Quantum NOT is called X and is defined as:

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

We will show that, if we apply this operation to a general Qubit $\alpha|0\rangle + \beta|1\rangle$ will give us the Qubit $\beta|0\rangle + \alpha|1\rangle$. Let's take a general Qubit $\psi = \alpha|0\rangle + \beta|1\rangle$ and let's write it in its vectorial form:

$$\psi = \alpha \begin{bmatrix} 1 \\ 0 \end{bmatrix} + \beta \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} \alpha \\ \beta \end{bmatrix}$$

To apply X to ψ we will do:

$$X \begin{bmatrix} \alpha \\ \beta \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} \alpha \\ \beta \end{bmatrix} = \begin{bmatrix} 0\alpha + 1\beta \\ 1\alpha + 0\beta \end{bmatrix} = \begin{bmatrix} \beta \\ \alpha \end{bmatrix} = \beta \begin{bmatrix} 1 \\ 0 \end{bmatrix} + \alpha \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

But, knowing that

$$|0\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \text{ and } |1\rangle = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

then we got:

$$\beta \begin{bmatrix} 1 \\ 0 \end{bmatrix} + \alpha \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \beta |0\rangle + \alpha |1\rangle$$

So we correctly get the NOT of the Qubit ψ.

Generally, an operation on a single Qubit can be described as a 2×2 matrix. Nevertheless not all 2×2 matrices define legal operations on Qubit: we must remembre that the *unitary* property of Qubit force that

$$\left|\alpha^2 + \beta^2\right| = 1 \quad \text{For all Quantum states } \alpha |0\rangle + \beta |1\rangle$$

This condition must be true in the Qubit we obtain after applying the gate too.

The matrix property that guarantees the transformation from a unitary vector to another one is that the matrix is unitary: a matrix is said unitary if, multiplied by its inverse, gives the unit matrix.

While in the classic case we have one one bit gate, in Quantum Circuits we have also the gates Z and H. The Z gate acts on a Qubit by switching the sign of the component $|1\rangle$ and its matrix is:

$$Z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$$

After this one, we have the *Hadamard* gate (known as H):

$$H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

that as the effect of transforming a base stat in a superposition that after a measure can result $|0\rangle$ or $|1\rangle$ with the same probability. The effect of H can be seen as a NOT not finished in a way such that the result is neither 0 nor 1 but a superposition of the two. For this reason, usually H is also known as 'square root of NOT'.

The fourth one Qubit gate is Y:

$$Y = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}$$

The triplet of matrix X, Z, H are known as the *Matrices of Pauli.*

8.2 Quantum Gates with multiple inputs

Operations on quantum registers (just like a classic register, but every cell is made by a Qubit) are necessary to describe transformations of non-trivial states. Operations on Qubit registers are described,

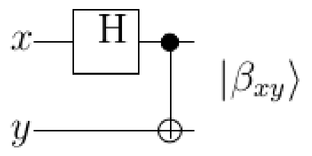


Figure 8.1: Bell states circuit

like operation with only one Qubit, by *linear operations*. The main gates on two Qubits are: *AND, OR, XOR, NAND, NOR*; The set {*NOT, AND*} is called *universal* because every boolean function can be described with these two gates.

The analogous of XOR in quantum circuits is the gate *CNOT* (Controlled-NOT) that works on two Qubits: the first one is called *control* and the second one is the *target*. If the *control* is zero then the *target* remains unchanged, while, if the *control* is one, the *target* is negated:

$$|00\rangle \Rightarrow |00\rangle, |01\rangle \Rightarrow |01\rangle, |10\rangle \Rightarrow |11\rangle, |11\rangle \Rightarrow |10\rangle$$

Where the first Qubit from the left is the *control* and the other is the *target*. Also, the *CNOT* tranformation can be seen as:

$$|A, B\rangle \Rightarrow |A, B \oplus A\rangle$$

The unitary matrix that describes the CNOT is:

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

The *CNOT* gate and the one Qubit gates are the base of all the quantum gates (together they are *universal* like *AND*, *NOT* for classic computing).

8.3 Entagled states

An important property of $n - bit$ quantum registers is that it's not always possible to decompose them into single Qubits. States like that are call 'Entagled' and they have properties that cannot be found in any object in classic physics: the members of an entagled collection don't have an individual state but only the entire collection has a state (entagled states behave like they are strictly connected to each other no matter the distance between them). For example measuring one of the two states of a entagled couple gives us information also about the other state that has not been measured.

8.4 Quantum Circuit: Bell states

The circuit in the figure 7.1 generates, for each state of the computational base

$$|00\rangle, |01\rangle, |10\rangle, |11\rangle$$

a particular entagled state. These states, we will refer to with $\beta_{00}, \beta_{01}, \beta_{10}, \beta_{11}$ are known as 'Bell states' or 'EPR' From Bell, Einstein, Podolsky and Rosen that discovered theri extraordinary properties. The circuit 7.1 transforms the first Qubit (using an Hadamard gate) in a superposition that is used as *control* for a CNOT gate. The input vector are transformed as follows:

$$|00\rangle \Rightarrow |\beta_{00}\rangle \equiv (|00\rangle + |11\rangle / \sqrt{2})\, |01\rangle \Rightarrow |\beta_{01}\rangle \equiv (|01\rangle + |10\rangle / \sqrt{2})\, |10\rangle \Rightarrow |\beta_{10}\rangle \equiv (|00\rangle - |11\rangle / \sqrt{2})\, |11\rangle \Rightarrow |\beta_{11}\rangle \equiv (|01\rangle -$$

8.5 Quantum Parallelism

On a Quantum Computer (based on Quantum Gates) we can evaluate a function $f(x)$ on different values of x all togheter. This fact is known as *Quantum Parallelism* and it is one of the main property of quantum circuits. Let's consider the boolean function:

$$f(x) : \{0, 1\} \to \{0, 1\}$$

To compute $f(x)$ using a quantum computation we have to define the transformation $f(x)$ like an unitary transformation, let's call it U_f. We can do that by applying a serie of quantum gates to the data register (the input) $|x, y\rangle$: this serie of gates will transform $|x, y\rangle$ into $\big|x, y \oplus f(x)\big\rangle$ (the output), and if $y = 0$ then the second Qubit of the output will contain exactly $f(x)$.

For example, if whe set the values:

$$x = \frac{|0\rangle + |1\rangle}{\sqrt{2}} (\text{can be obtain by applying H to a } |0\rangle \text{ Qubit}) y = |0\rangle$$

and we use it as input for U_f we will obtain as result:

$$\frac{\big|0, f(0)\big\rangle + \big|1, f(1)\big\rangle}{\sqrt{2}}$$

In this way, we have evaluated / gathered information about both $f(0)$ and $f(1)$: this kind of Parallelism is different from the classic one where different circuits (everyone of them evaluate $f(x)$ for a different x) are runned together.
This procedure can be generalized to compute function on a arbitrary number of bit using a genralization of the Hadamard gate known as *Walsh-Hadamard transform.*
This transformation is based on n Hadamard gates that work together on n Qubits: for example, with $n = 2$, the Walsh-Hadamard transformation is written as $H^{\otimes 2} = H \otimes H$ and applyied to two Qubits in the $|0\rangle$ state gives as result:

$$\frac{|00\rangle + |01\rangle + |10\rangle + |11\rangle}{2}$$

In General, the result of $H^{\otimes n}$ applyied to n Qubit in the $|0\rangle$ state is:

$$\frac{1}{\sqrt{2^n}} \sum_x |x\rangle$$

Where x i the binary representation of all the numbers between 0 and $2^n - 1$.
So, the Walsh-Hadamard transformation produces an equiprobable superposition of all the states of the n-bit computation base. (let's see that to get a superposition of 2^n states we only need n Hadamard gates).
So, the parallel evaluation of a function $f(x)$ with $|x| = n$bits and 1 bit as output can be achieved by prepairing n bit in the state $|0\rangle^{\otimes n}$ (the input) and one bit in the state $|0\rangle$ (the output) and giving them as input to the U_f circuit. The result will then be:

$$\frac{1}{\sqrt{2^n}} \sum_x |x\rangle \left|f(x)\right\rangle$$

The only problem that we have with Quantum Parallelism is that we cannot obtain all the values that have computed with just one measure: measuring the state $\frac{1}{\sqrt{2^n}} \sum_x |x\rangle \left|f(x)\right\rangle$ will give us the value of $f(x)$ for just one x. But, we can, for example, define if a Function is *balanced* or *constant.*

9 Grover Algorithm

In this last chapter we see a quantum algorithm that allows us to solve a problem with quadratic speedup compared to a deterministic solution. The description of this problem has been taken from [4]. The problem we will handle is a *search problem*: suppose you have a set S of N elements, and you want to find the one which has some specific features. Suppose you have a function $f : S \to \{0, 1\}$ defined as follow:

$$f(x) = \begin{cases} 0, & \text{if } x \text{ hasn't the feature we are looking for} \\ 1, & \text{otherwise} \end{cases}$$

Then, a deterministic algorithm to find a solution to this problem could be:

```
function SEARCH($S, f$)
    for all $x \in S$ do
        if $f(x) == 1$ then
            return $x$
        end if
    end for
end function
```

This solution has a time complexity that is $O(|S|) = O(N)$ because can happen that the solution is the $N-th$ element of S.
Remarkably, there is a *Quantum Search Algorithm*, known as *Grover Algorithm* which allows us to find a solution to this problem with a *quadratic* speedup, with a $O(\sqrt{N})$ complexity.

9.1 The oracle

Before looking at the procedure to solve this problem, we need to see how the function f we mentioned above can be actually implemented. In this case, we are going to use a *Quantum Oracle*.

Suppose that we want to search through a space of N elements. Let's concentrate not in the actual number but on its index, that is a number between 0 and $N-1$. Then, we can assume that the problem has not just one solution, but M different ones ($M \leq N$).
A particular instance of the problem can be described by a function f as defined above, where we just change the domine from S to $\{0, \cdots N-1\}$. Suppose we are supplied with a quantum *oracle* - a black

box - with the ability to recognize solutions to the problem. This recognition is signalled by making use of an *oracle qubit.* More precisely, the oracle is an unitary operator O, defined by its action on the computational basis:

$$|x\rangle |q\rangle \xrightarrow{O} |x\rangle \left|q \oplus f(x)\right\rangle$$

where x is the index register, and the oracle qubit $|q\rangle$ is a single qubit which is flipped if $f(x) = 1$, and is unchanged otherwise. We can check whether x is a solution or not by preparing $|x\rangle |0\rangle$, applying the oracle, and checking to see if the oracle qubit has been flipped to $|1\rangle$.

9.2 Grover Operator

The algorithm makes use of a single n qubit register. The goal of the algorithm is to find a solution to the search problem we explained above, using the smallest possible number of applications of the oracle. The algorithm begins with the state $|0\rangle^{\otimes n}$: this transformation, as we saw in the previous chapter is used to put the computer in the state

$$|\phi\rangle = \frac{1}{\sqrt{N}} \sum_{x=0}^{N-1} |x\rangle$$

The Grover quantum search algorithm then consists of repeated application of a Quantum *operator* G, also known as the *Grover operator.* The operations that G does are:

1. Apply the oracle O
2. Apply the Hadamard transform $H^{\otimes n}$
3. Perform a conditional phase shift on the computer, with every computational basis state except $|0\rangle$ receiving a phase shift of 1. So:

$$Phase(|x\rangle) = \begin{cases} |0\rangle\,, & \text{if } x = 0 \\ -|x\rangle\,, & \text{otherwise} \end{cases}$$

4. Apply the Hadamard transform $H^{\otimes n}$

9.3 Complete Algorithm

now that we have seen how the *Grover operator G* works, we can now give a complete procedure for the quantum search algorithm: The algorithm takes two inputs:

1. An Oracle O, which performs the transformation

$$O|x\rangle |q\rangle = |x\rangle \left|q \oplus f(x)\right\rangle$$

where assuming x_0 is the solution of the problem, then $f(x) : \{0, \cdots, N-1\} \rightarrow \{0, 1\}$ is defined

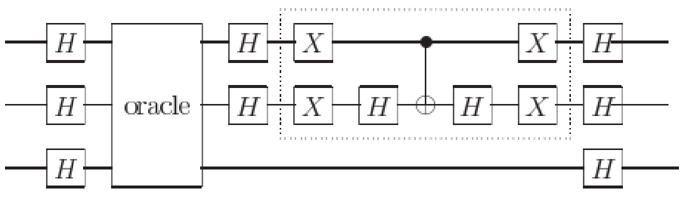


Figure 9.1: Circuit for Grover Algorithm with N = 4 and M = 1

as:

$$f(x) = \begin{cases} 1, & \text{if } x = x_0 \\ 0, & \text{otherwise} \end{cases}$$

2. $n + 1$ qubits in the state $|0\rangle$

The operations that the algorithm does are:

1. apply $H^{\otimes n}$ to the first n qubits, and HX to the last one
2. Apply the Grover operator G for $\approx \left\lceil \pi \frac{\sqrt{N}}{4} \right\rceil$ times
3. Measure the first n qubits to obtain x_0

This algorithm gives us the result with an $O(1)$ probability, with a complexity that is $O(\sqrt{N})$

9.4 Solving Grover algorithm with N = 4

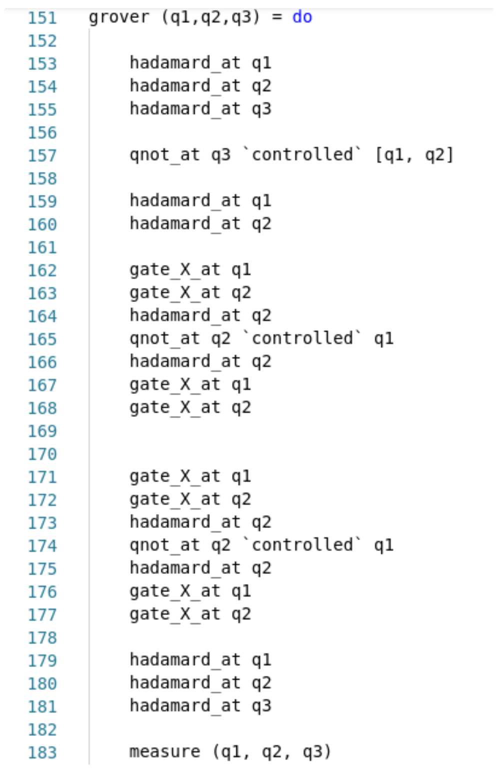
We will now use the definitions of Quantum Circuit we gave in chapter 8 to show an actual implementation of Grover Algorithm in the case where possible solutions space dimension is 4 ($N = 4 = 2^2$) and M is 1. In this case, the function $f : 0, 1, 2, 3 \rightarrow 0, 1$ is defined as follows:

$$f(x) = \begin{cases} 1, & \text{if } x = 3 \\ 0, & \text{otherwise} \end{cases}$$

So the solution we are looking for is $x_0 = 3$. The circuit that implements Grover Algorithm under this hypotesis is in figure 9.1(taken from [4]).

Generally, until $N \leq 4$ the solution to Grover Algorithm can be got by applying just one time the Grover operator: in this case we exactly apply it 1 time and then by measuring the first 2 qubits we will get the answer to the problem.
The oracle, to choose exactly $x_0 = 3$ as solution can be implemented as a CNOT circuit where the third Qubit (the one in the bottom) is the *target* and the other two qubits are used as *controls* (the third Qubit is negated only if both the first and the second qubit are $|1\rangle$).

For the circuit above, the two upper qubits are set to $|0\rangle$, while the bottom one is set to $|1\rangle$. After consulting the oracle, the circuit inside the dotted box implements the condition phase shift we mentioned before.



```
151  grover (q1,q2,q3) = do
152
153      hadamard_at q1
154      hadamard_at q2
155      hadamard_at q3
156
157      qnot_at q3 `controlled` [q1, q2]
158
159      hadamard_at q1
160      hadamard_at q2
161
162      gate_X_at q1
163      gate_X_at q2
164      hadamard_at q2
165      qnot_at q2 `controlled` q1
166      hadamard_at q2
167      gate_X_at q1
168      gate_X_at q2
169
170
171      gate_X_at q1
172      gate_X_at q2
173      hadamard_at q2
174      qnot_at q2 `controlled` q1
175      hadamard_at q2
176      gate_X_at q1
177      gate_X_at q2
178
179      hadamard_at q1
180      hadamard_at q2
181      hadamard_at q3
182
183      measure (q1, q2, q3)
```

Figure 9.2: Grover Quipper code

9.4.1 Implementing Grover with N = 4 with Quipper

In this last section we will see the source code of Grover with $N = 4$ using an Haskell library called Quipper that allows us to create Quantum Circuits and simulate them in a deterministic device.

As we can see in 9.4.1, the source code has been implemented to follow slavishly the circuit in the image 9.1 (the only difference is that we apply 2 times the Grover Operator insted of one).

In 9.1, the Grover Operator G is the circuit inside the dot box.

The 3 Qubits that the circuit takes in input are set in this state

$$|q_1\rangle \otimes |q_2\rangle \otimes |q_3\rangle = |000\rangle$$

As first operation, we apply Hadamard to all the three Qubits: the result of this operation is passed to the Oracle: in 9.4.1 the oracle is implemented by the instruction at line 157: we apply a $CNOT$ gate with q_3 as *target* and with q_1 & q_2 as *controls*.

Then, following the structure of the Grover Algorithm, we need to apply the Grover Operator for $\left\lceil \pi \frac{\sqrt{N}}{4} \right\rceil = 2$ times. The application of the Grover Operator, in the code, is made (the first time) from line 162 to 168, and the second time from line 171 to 177.

To end the algorithm, we apply Hadamard to the three Qubits and then we measure them.

Summary

We saw, starting from a mathematical point of view, a little part of quantum computing. Starting from a merely Computer Science / Physical point of view, we then had a little introduction to classical TM and Complexity, to then pass to the same arguments but in a Quantum point of view: for this purpose, we talked about two different theorical models (Deutsch and Bernestein & Vazirani) showing their properties and features. We focused on the second one because it allowed us to achieve several objectives:

1. We could create a bond from QTMs and TMs using definitions and Lemmas
2. We have been able to see Programming Primitives that allow us to create more complex and powerfult QTMs
3. We could introduce Complexity for Quantum Computing, having a look also to which are the bonds between Quantum and Classic Complexity

After that, we passed to Quantum Circuits, an instrument that can allow us to actually show the power of quantum computing: in fact, using Quantum Circuit we could explain a Quantum Algorithm (Grover) both from a theorical point of view and from a practical one; we also implemented it in a Program Language (Haskell) using a library that allows us to simulate Quantum Circuits (Quipper).

In this Thesis we didn't talked about complexity for Quantum Circuits because it would have required an huge introduction about classical circuits and about their complexity; It's true that we did it for TMs, but the discussion about circuits would have been really heavier than the one about TMs. (For the ones which are interested about circuit complexity, I advice to read [5]). Moreover, we didn't talk about a third model of QTM (that I studied too), exposed by 3 italian scientists in [6]: I decided not to talk about that model because, as an introduction, the one described in [3] was simplier but, at the same time, useful and complete to understand all the facets of Quantum Computing.

Bibliography

[1] R. Feynmann, *Simulating physics with computers*, Internat. J. Theoret. Phys., 21(1982), pp. 467-488.

[2] D. Deutsch, *Quantum theory, the Church-Turing principle and the universal quantum computer*, in Proc. Roy. Soc. London Ser. A, 400 (1985), pp. 97-117

[3] E. Bernstein, U. Vazirani, *Quantum complexity theory* SIAM J. Compu., 26 (1997), pp. 1411-1473

[4] M. A. Nielsen, Isaac L. Chuang, *Quantum Computation and Quantum Information* Cambridge University Press, 2010

[5] A. Yao, *Quantum Circuit Complexity*, in Proc. 34th Annual IEEE Symposium on Foundations of Computer Science, IEEE press, Piscataway, NJ, 1993, pp. 352-361

[6] S. Guerrini, S. Martini, A. Masini, *Towards a theory of quantum computability*, CoRR, abs/1054.02817, 2015