

An Introduction to Quantum Computing for Computer Scientists, with SAT

Francesco Piro

March 30, 2020

Abstract

This is the paper I have produced during my study of quantum computing for solving the SAT problem...

Introduction

Quantum computation is a wide area involving several disciplines that is having always more success in nowadays applications thanks in particular to the development of the technology and its incredible results. Quantum physics' principles are the fundamentals on which the entire theory is based: thanks to their properties, new architectures allow to define devices that can solve classical problems in surprisingly reduced time and space complexities. However we always have a trade-off to consider, in particular now that these technology are still emerging.

This paper aims at giving a description for computer scientists of what a quantum computer is and which are its real impacts and advantages with respect to the classical ones. Hence I will try to provide a description of the state of the art of quantum computing with an approach that allows to understand how from the basic principles of quantum mechanics we are able to have an algorithm that is faster with respect to its classic counterpart. In order to do so in the first chapter (1) I will start with the basic linear algebra needed to study the quantum physics' principles we need to define a quantum computer. The definition of the *quantum computer* is fundamental both to understand how and algorithm is executed but also to have a comparison with the classic Turing machine that allows us to determine conclusions on computational complexity (chapter 2). In the first chapter we will also have practical examples realized with the **qiskit library** in order to clarify also with some lines of code the concepts. Once the background on quantum computation and computational theory are well consolidated by the reader, the last chapters provide a practical example that is used to prove the speedup for the particular **satisfiability problem**. I have used an efficient classical solver for the *k-SAT* as I could compare it with my quantum implementation, both realized in python. In the

end I provide some important conclusions for quantum computing that I was able to conclude thanks to my study, in particular in the papers listed in the references.

1 Quantum Computing

This chapter aims at providing first the fundamentals needed in order to deal with quantum mechanics and second at defining a quantum computer thanks to the principles previously identified. With the quantum device we will be able to make a comparison with the classical one to understand with examples how the basic operations are realized in order to use them to implement complete algorithms. Further in the chapter, a section is completely dedicated to the main quantum search algorithm that is fundamental to solve the SAT with a quantum algorithm. Also here we will start from a classical version to compare it with its quantum counterpart. All the arguments related in this chapter, together with the ones in the next (2) are fundamental to give a significant interpretation to the comparison between the classic and quantum implementation of the algorithm able to solve the **satisfiability problem** (2.4).

1.1 Fundamentals

The study of quantum computers requires the knowledge of the decimal and binary representation of integers, probability notions and in particular linear algebra fundamental definitions like the ones of: *vectors, spaces, bases, linear systems, tensor product....* In this section are presented the basic concepts needed to face the quantum physics principles that we need in order to realize our quantum computer.

1.1.1 Linear Algebra

Basic principles of linear algebra are assumed to be well known by the reader, I want now to remark only the most important operators and definitions that we need to face the definition of the following quantum mechanical theorems we need to define a quantum device. The most important notions we need to acquire from this section are: *tensor product, Hilbert space, bra-ket notation*.

In order to realize a quantum computer we need understand how to define a state that is able to contain information that can be used to obtain a certain objective. As we will see in the next section, the state of a quantum device is a quantum state, thus a mathematical model that lives into a specific *vector state* whose dimension depends on the amount of information it needs to take care of. Typically, significant states contain information that results from the composition of several spaces combined together thanks to a particular operator called tensor product.

Definition (Tensor Product): Given two vector spaces V and W over a field K with bases e_1, \dots, e_m and f_1, \dots, f_n respectively, the tensor product $V \otimes W$ is another vector space over K of dimension mn . The tensor product space is equipped with a bilinear operation $\otimes : V \times W \rightarrow V \otimes W$. The vector space $V \otimes W$ has basis $e_i \otimes f_j \forall i = 1, \dots, m, j = 1, \dots, n$.

Typically we are going to work with complex Euclidean vector spaces of the form \mathbb{C}^n and, by choosing the standard basis in the origin vector spaces, then the tensor product is nothing more than the Kronecker product.

Definition (Kronecker Product): Given $A \in \mathbb{C}^{m \times n}$, $B \in \mathbb{C}^{p \times q}$, the Kronecker product $A \otimes B$ is the matrix $D \in \mathbb{C}^{mp \times nq}$ defined as:

$$D = A \otimes B = \begin{pmatrix} a_{11}B & \cdots & a_{1n}B \\ a_{21}B & \cdots & a_{2n}B \\ \vdots & \vdots & \vdots \\ a_{m1}B & \cdots & a_{mn}B \end{pmatrix}$$

Now that we know the notions of vector state and Tensor product, we can use them to define an important space (in particular for the SAT problem we are going to study later) called the **Hilbert space** and denoted with \mathcal{H} .

Definition (Hilbert Space): Given the complex space \mathbb{C} we define the Hilbert space \mathcal{H} as the $(n + 1)$ -tuple tensor product:

$$\mathcal{H} := \bigotimes_1^{n+1} \mathbb{C}^2$$

As we said, in the Hilbert space we will carry the discussion of the SAT problem but to understand what the result of this tensor product actually defines we still need to give probably the most important definition. In order to represent a quantum state we use the so called *bra-ket notation* introduced in 1939 by Paul Dirac.

Definition (Dirac/bra-ket Notation): Given a complex Euclidean space $\mathbb{S} \equiv \mathbb{C}^n$, $|\psi\rangle \in \mathbb{S}$ denotes a column vector, and $\langle\psi| \in \mathbb{S}^*$ denotes a row vector that is the conjugate transpose of $|\psi\rangle$, i.e. $\langle\psi| = |\psi\rangle^*$. The vector $|\psi\rangle$ is also called a *ket*, while the vector $\langle\psi|$ is also called a *bra*.

The bra-ket notation allows us to define a quantum state, hence a vector that lives into a particular vector space. Its definition intrinsically defines also the result of combining two states living in the same state with the **inner product**, straightforwardly obtained from what we have just said: $\langle\psi|\phi\rangle$. This result is fundamental to define spaces that are higher than one only dimension as the Hilbert state presented before. To understand what \mathcal{H} is we can now use two examples where we define the basis of the first two results obtained by doing the tensor product of the complex space \mathbb{C}^2 with itself. Remember that the basis

of a space is the smallest set of linearly independent vectors that can be used to represent all the vectors that belong to that space.

Example 1: *Considering the basic case of \mathbb{C}^2 the basis can be trivially identified as:*

$$|0\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \quad |1\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

Example 2: *Considering a single product, thus $\bigotimes_1^1 \mathbb{C}^2 = \mathbb{C}^2 \otimes \mathbb{C}^2$, we obtain the basis by multiplying in all possible ways the vectors of the basis of the previous example. We now have kets of dimension 2, thus vectors with 4 lines and 1 column:*

$$|00\rangle = |0\rangle \otimes |0\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \otimes \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \begin{pmatrix} 1 \\ 0 \end{pmatrix} \\ 0 \begin{pmatrix} 1 \\ 0 \end{pmatrix} \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

$$|01\rangle = |0\rangle \otimes |1\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \otimes \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \begin{pmatrix} 0 \\ 1 \end{pmatrix} \\ 0 \begin{pmatrix} 0 \\ 1 \end{pmatrix} \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix}$$

$$|10\rangle = |1\rangle \otimes |0\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \otimes \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \begin{pmatrix} 1 \\ 0 \end{pmatrix} \\ 1 \begin{pmatrix} 1 \\ 0 \end{pmatrix} \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix}$$

$$|11\rangle = |1\rangle \otimes |1\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \otimes \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \begin{pmatrix} 0 \\ 1 \end{pmatrix} \\ 1 \begin{pmatrix} 0 \\ 1 \end{pmatrix} \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}$$

Thanks to these two examples we can generalize to the $n + 1$ case and obtain the definition of the Hilbert space we gave before.

Before starting with the quantum physics section where we will start by defining the smallest unit element we use to represent information, the so called **qubit**, we need to give one further definition. In order to perform operations on qubits we will consider (section 1.1.5) only a particular family of matrices called unitary matrices. These matrices allow to perform operations on qubits without modifying the basic properties of the quantum state and are defined as follows.

Definition (Unitary Matrix): A complex square matrix U is unitary if $U^*U = UU^* = I$.

Unitary matrices have significant importance in quantum mechanics because they are **norm-preserving**, this will be fundamental to identify the two main features of quantum operations that can now be introduced as: *apply a unitary matrix on a quantum state, thus a vector whose norm will be preserved.*

1.1.2 Quantum Physics

Quantum mechanics is a fundamental theory in physics describing the properties of nature. We do not need entirely its entire power for our purpose but, starting from some basic concepts we will exploit some conclusions that are useful to design an algorithm that is faster than its classical counterpart. As we do when we start studying computer science, we want to identify the smallest, most basic element that allows us to represent the information. From a classical point of view we have the **bit** whose values can be either 1 or 0. From a quantum point of view, instead, we have the **qubit**, complex variables that can assume values ranging from 0 to 1 (in modulus), identified in a complex space over a surface called the *Block Sphere*.

Definition (Qubit): The qubit is the smallest unit of measurement used to quantify information in quantum computing. It identifies the bit in a superposition, hence both 0 and 1 values are considered. Formally it is a vector of the space \mathbb{C}^2 represented as a linear combination of the elements contained in its basis (1.1.1). A qubit ψ , with $\alpha_0, \alpha_1 \in \mathbb{C}$, is defined as:

$$|\psi\rangle = \alpha_0|0\rangle + \alpha_1|1\rangle = \alpha_0 \begin{pmatrix} 1 \\ 0 \end{pmatrix} + \alpha_1 \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

To understand better the definition of the qubit we now provide its representation in a tridimensional space whose directions are obtained from basic linear algebra principles that are not now relevant. We now just need to understand that, thanks to the coefficients α_0 and α_1 belonging to the complex space \mathbb{C} we are able to consider the basic element of our computation as one of the infinite points that live over the surface of a sphere. We can grasp in this concept a first hint in the advantages that quantum states provide with respect their classic counterpart.

Definition (Block Sphere): The Block Sphere is the geometrical representation of the pure state space of a two-level quantum mechanical system. In other words it represents all the possible vectors that can be obtained by combining the vectors of the basis for a quantum register of 1 qubit.

Consider the following examples to understand how vectors are represented in the Bloch sphere; we present the trivial cases for both $|0\rangle$ and $|1\rangle$ and the respective orthogonal vectors identifying the x and y axes.

Example 3: Consider the qubit ψ , with $\alpha_0, \alpha_1 \in \mathbb{C}$, such that: $|\psi\rangle = \alpha_0|0\rangle + \alpha_1|1\rangle$. (Check code at 1)

1. $\alpha_0 = 1, \alpha_1 = 0$

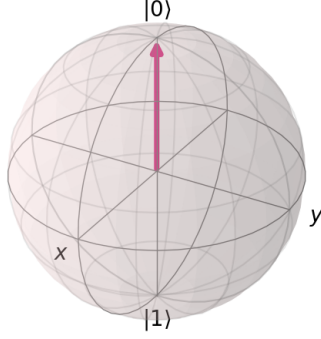


Figure 1: $|\psi\rangle = |0\rangle$

2. $\alpha_0 = 0, \alpha_1 = 1$

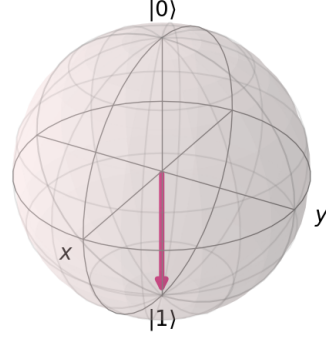


Figure 2: $|\psi\rangle = |1\rangle$

3. $\alpha_0 = \frac{1}{\sqrt{2}}, \alpha_1 = \frac{1}{\sqrt{2}}$

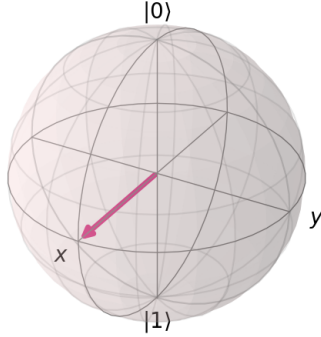


Figure 3: $|\psi\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$

4. $\alpha_0 = \frac{1}{\sqrt{2}}, \alpha_1 = \frac{i}{\sqrt{2}}$

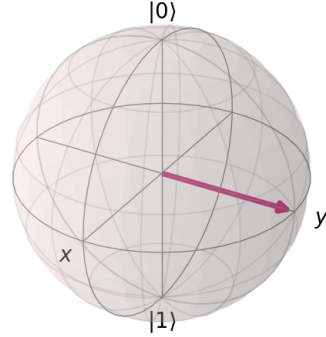


Figure 4: $|\psi\rangle = \frac{1}{\sqrt{2}}(|0\rangle + i|1\rangle)$

Obviously we can also represent vectors that belong to spaces with a higher dimension than 2, in this case we will have their representation over a hypersphere (check the `plot_bloch_multivector` method in the qiskit library). As in classical computer we have registers, defined as a sequence of bits, in quantum computing we have *quantum-registers* composed of a sequence of qubits. Typically a quantum computer has a single quantum-register made up of qubits (see

section 1.2). It is now important to remark the first important conclusion on quantum computing, that derives from the definition of quantum-register that we have just provided. We see that $\bigotimes_1^{n+1} \mathbb{C}^2$ is a 2^n dimensional space. This is sharp in contrast with what happens for classical registers: given n classical bits, their state is a binary string in $\{0,1\}^n$, thus an n -dimensional space. In other words we arrive to the first important conclusion.

Conclusion 1: *the dimension of the state space of quantum registers grows exponentially in the number of qubits, whereas the dimension of the state space of classical registers grows linearly in the number of bits.*

1.1.3 Superposition

The mathematical definition of the qubit we have just seen is useful to understand the first difference between the representation of a state either in classical or quantum fields. We want now to define from a more physical point of view what means for a qubit to assume both the value 0 and 1. The basic physical principle behind this property is the *Heisenberg indetermination principle*, but without going too deep in the details let's try to understand it with a simple example.

Example 4: *We have seen that bits can assume at a certain time instant one and one value only. Considering now a qubit we could say that it assumed both the values 1 and 0 because its definition is based on Heisenberg's principle, basically stating that particles can assume at the same time different positions. This is a physics principle: electrons can be at the same time in different positions. That is why considering the position 0 and the position 1 we can say that a qubit is situated in both of them. The sad reality is that we do not know the real state of the qubit until we perform a specific operation on it, called measurement, which makes it collapse either to a 0 or a 1 "boring" bit. The following picture tries to illustrate the principle of an electron whose real position is not deterministic until this operation.*

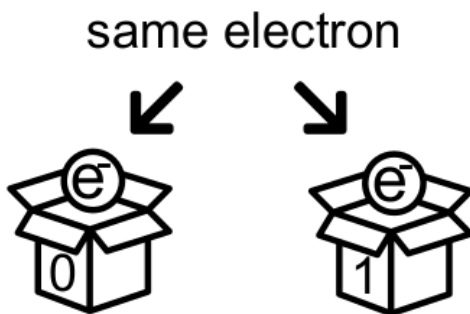


Figure 5: Superposition of the electron

As we may have understood superposition is a very interesting property, also because it can be generalized to the case of n qubits rather than just one. We will see in the implementation of the SAT algorithm how operations on multiple qubits are able to exploit superposition in order to compute the solution of the problem. With this property we can start guessing how algorithms will be run on our quantum computer: from a certain initial state we will perform operations that exploit the indetermination principle over the qubits, thus considering several states at the same time, until the end when performing the measurement we will make them collapse to a single string of 1s and 0s which is our result.

1.1.4 Entanglement

Entanglement is the second main feature of quantum computers that, together with superposition, differentiates them from quantum computers. These are in fact the two main features that are exploited in algorithms to find the solution of a problem.

To understand entanglement we start by answering the following question:

What do we gain by moving from single qubits spaces to multiple qubits spaces?

The answer has two motivations, the first regarding the states representation the second the property of entanglement:

- (i) As we described before, also in quantum computing we will need to define registers that contain more than 1 qubit. To do so it suffices to compose a vector of n qubits which identify the state obtained by the result of their quantum product.
- (ii) Linear algebra definitions, in particular for what concerns the tensor products, show that not every state that we consider can be represented as the tensor product of n -qubits. Whenever we have to deal with a state of that kind we will say that the quantum state is entangled.

More formally the definition of entanglement is the following:

Definition (Entanglement): A quantum state $|\psi\rangle \in \bigotimes_1^{n+1} \mathbb{C}^2$ is a product state if it can be expressed as a tensor product $|\psi_0\rangle \otimes |\psi_1\rangle \otimes \dots \otimes |\psi_n\rangle$ of n 1-qubit states. Otherwise, it is entangled.

To understand better the definition let's consider the simplest example possible: considering a state living in the \mathbb{C}^4 complex space we want to check if it is a product state or entangled by looking for two 1-qubit states whose tensor product is the state that we are dealing with. We will see next how the entanglement feature can be used, in particular what still happens when 2 qubits are entangled together.

Example 5: Consider the following 2-qubit state:

$$\frac{1}{2}|00\rangle + \frac{1}{2}|01\rangle + \frac{1}{2}|10\rangle + \frac{1}{2}|11\rangle$$

This is a product state because we can find two states whose tensor product is the starting one: $\frac{1}{2}(|0\rangle + |1\rangle) \otimes \frac{1}{2}(|0\rangle + |1\rangle)$. By contrast, the 2-qubit state:

$$\frac{1}{\sqrt{2}}|00\rangle + \frac{1}{\sqrt{2}}|11\rangle$$

is an entangled state, because it can not be expressed as the tensor product of two 1-qubit states.

The actual meaning of entanglement can be fully appreciated only once we have defined the measurement, but we can start grasping its importance thanks to the algebraic definition that we have just provided. Having an entangled state means that we can not find two qubits whose tensor product result is the one that we are considering, this because there are not two "linearly independent" states satisfying such property. In conclusion the qubits that compose an entangled state are in some way related one with the other, in other words: *when two or more qubits are entangled, they affect each other, and measuring one qubit changes the probability distribution for the other qubits.*

1.1.5 Operations on Qubits

The definition of qubit should be clear now, moreover we have also understood that a quantum computer is composed of a quantum register where multiple qubits are used in order to move from a state to another. We want now to understand how to perform operations on qubits (without breaking the basic properties of the quantum state) so that we can implement algorithms that allow to solve problems incrementally going through different quantum states. First of all we have to give the definition of a quantum operation on n qubits:

Definition (Quantum Gates): An operation performed by a quantum computer with n qubits, also called a gate, is a unitary matrix in $\mathbb{C}^{2^n \times 2^n}$.

Thus, for an n-qubit system, the quantum state is a unit vector $|\psi\rangle \in \mathbb{C}^{2^n}$, while a quantum operation is a matrix $U \in \mathbb{C}^{2^n \times 2^n}$, and the application of U onto the state $|\psi\rangle$ is the unit vector $U|\psi\rangle \in \mathbb{C}^{2^n}$. This leads to the following important features of quantum gates:

- Quantum operations are **linear**
- Quantum operations are **reversible**

Every significant operation on a quantum state must be represented as a unitary matrix, this may seem very restrictive but it has been proved that these two features do not remove any power to the quantum computer we want to design. We can now give the following important conclusion.

Conclusion 2: *A universal quantum computer is Turing-complete.*

Now that we know how quantum operations are formally defined we have to show first how they operate over a set of qubits and second which are the most important gates that we need to implement quantum algorithms. A quantum algorithm is implemented with a quantum circuit: a quantum circuit is represented by indicating which operations are performed on each qubit or group of qubits. For a quantum computer with n qubits, we represent n qubits lines and operations as blocks taking as input a set of qubits and with output the same input lines. Consider the following picture as the first trivial representation where a general unitary matrix U is applied over all the qubits of the quantum computer.

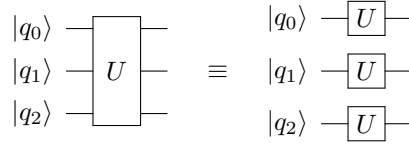


Figure 6: Trivial equivalence of quantum circuits

In the picture above we have used a slight abuse of notation for what concerns the multigate U applied over the 3-qubit state. In mathematical terms, in fact, the equivalence holds when we consider the gate $U \otimes U \otimes U$ applied to the state composed of 3 qubits. Hence, the gate of the first circuit has to be interpreted as the unitary matrix obtained from the tensor product of 3 U unitary matrices, thus living in the \mathbb{C}^8 space.

Before going to study the fundamental gates we need to implement quantum algorithms it is very important to understand how to interpret the representation of a quantum circuit. Circuit diagrams are read from left to right, but because each gate corresponds to applying a matrix to the quantum state, the matrices corresponding to the gates should be written from right to left in the mathematical representation. In the following picture, for example, the result of the circuit is the state $BA|\psi\rangle$, obtained by first applying gate A and then B .

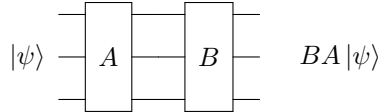


Figure 7: Quantum circuit interpretation

So far it seems that every kind of unitary gate is allowed to be used in a quantum circuit, but as we are not allowed in classical algorithms to define every kind of function also in quantum computing we will define operations as unitary gates by combining a set of nice matrices which are efficiently specifiable and implementable. The only set of nice matrices we will consider in our study

(which suffices to implement the SAT algorithm) are called the *Pauli Operators* and they are defined as follows.

Definition (Pauli Operators): *The Pauli operators are four single-qubit unitary matrices I, X, Y, Z forming a basis for $\mathbb{C}^{2 \times 2}$ such that: $XYZ = iI$. The four matrices are:*

$$\begin{aligned} I &= \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} & X &= \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \\ Y &= \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix} & Z &= \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \end{aligned}$$

With the definition it can be checked trivially that all I, X, Y, Z are unitary.

Now that we know the Pauli set we can start to give the list of the most important operators we need to implement quantum algorithms by comparing them with the respective counterpart operations in classic computation:

- The X gate is the equivalent to the NOT gate in classical computers. Thus we have: $X|0\rangle = |1\rangle$ and $X|1\rangle = |0\rangle$.
- The Z gate has no equivalent in classical computers because it performs a phase-flip on the target qubit. Thus we have: $Z|0\rangle = |0\rangle$ and $Z|1\rangle = -|1\rangle$.
- Another single-qubit fundamental gate is the Hadamard gate H . H is still a unitary matrix that belongs to the class of the **Clifford** gates, the characteristic property of a Clifford gate is to transform a Pauli operator in another Pauli operator. The Hadamard gate is defined as:

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

Applying a Hadamard to a qubit brings it to a superposition, we will see later that this is the fundamental initialization of several quantum algorithms. In fact: $H|0\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ and $H|1\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$.

So far we have considered only single-qubit gates, let's continue our list with multiple qubit gates, starting from the basic ones and continuing with those obtained by computing a tensor product of the single qubit gates just seen.

- The CNOT gate, also called "controlled NOT", acts on a 2-qubit state. The two qubit are called *control* and *target* qubits, and the CNOT gate works as follows: the target qubit is inverted if and only if the control qubit value is $|1\rangle$. The unitary matrix representing the CNOT gate is defined as:

$$CNOT := \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

As we may have guessed we have a clear analogy with the XOR gate in classical computers, in fact, as we see in the circuit representation the result is nothing more than the XOR between the control and the target qubit.

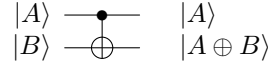


Figure 8: The CNOT gate

- The SWAP gate is used to swap the control with the target qubit and it can be obtained by using a sequence of three CNOT gates. The SWAP operation on a quantum state maps it to a new quantum state in which every basis state has its i-th and j-th digit permuted. The circuit definition is provided as an equivalence with the sequence of CNOT gates that allow to implement it.

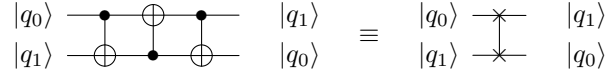


Figure 9: The SWAP gate

- The CNOT gate can be extended to more than two qubits only. If we consider two bits as control bits and one as target we obtain the "double controlled NOT" gate also known as the Toffoli gate. The generalized meaning of the CCNOT gate is: the control qubit is flipped if and only if both the control qubits values are $|1\rangle$. The unitary matrix representing the CCNOT gate is defined as:

$$CCNOT := \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

If we consider as target qubit $|0\rangle$ we clearly see the analogy between the CCNOT gate and the AND classical gate. The result of applying it to two general control qubits $|A\rangle$ and $|B\rangle$ yields in fact to the logical formula shown in the circuit representation.

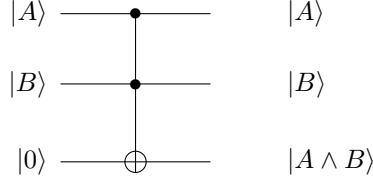


Figure 10: The CCNOT gate

- In general, as we saw at the beginning of this section we can build quantum gates by computing the tensor product of a set of nice unitary matrix, so to obtain another unitary matrix that maintains the basic properties of the quantum state on which we are acting. The most important example is to compute the Hadamard gate on all the n qubits of which a quantum computer is composed. This operations allows to bring the initial state in a *superposition* so that we can continue performing quantum gates to implement our quantum algorithm. The formal definition of an n -Hadamard gate is:

$$\bigotimes^n \mathcal{H} = \frac{1}{\sqrt{2}} \begin{pmatrix} \bigotimes^{n-1} \mathcal{H} & \bigotimes^{n-1} \mathcal{H} \\ \bigotimes^{n-1} \mathcal{H} & -\bigotimes^{n-1} \mathcal{H} \end{pmatrix}$$

Now that we know the basic quantum gates and their relative counterpart in classical computers we can start to play with them in an example. Knowing in fact how to define a NOT and an AND gate we can do everything by exploiting the De Morgan laws.

Example 6: Consider the following clauses defining a 3-SAT problem over the variables X_1, X_2, X_3 :

$$\begin{aligned} C_0 &= \{\neg X_1, X_2, X_3\} \\ C_1 &= \{X_1, \neg X_2, X_3\} \end{aligned}$$

The problem is trivially satisfiable, but for now we are interested in its representation by using the gates that we have defined so far, knowing that the definition of satisfiability yields to check the possibility to assign the true value to the CNF of the clauses defining the problem. In our case the CNF becomes:

$$CNF = (\neg X_1 \vee X_2 \vee X_3) \wedge (X_1 \vee \neg X_2 \vee X_3)$$

And the quantum circuit corresponding to this instance can be obtained with the implementation available at the path: `/CODE/QUANTUM/DECISIONVERSION`. (Check the snippet of code for this example at (2))

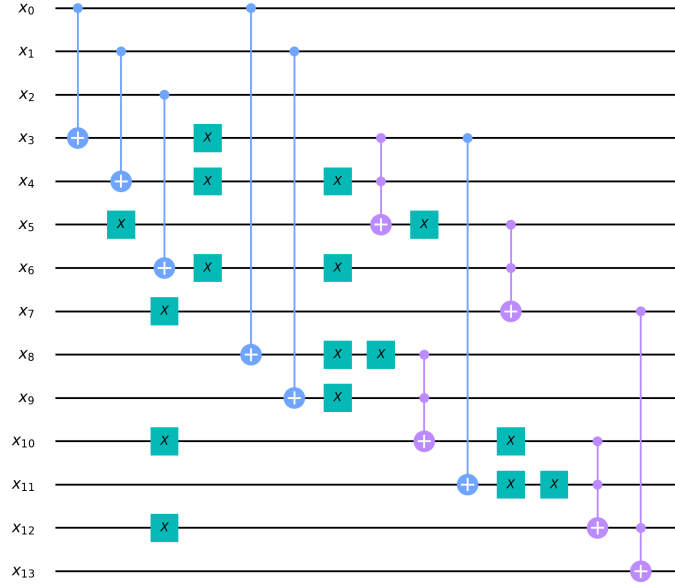


Figure 11: CNF circuit

To conclude with this section we still need to define the fundamental operation that allows us to retrieve the information from a qubit. As we have already said for qubits, while our algorithm is running, they possess the superposition property which makes them not to be at a certain instant in an exact place. To access the qubit information we want to understand if at that instance it is placed in either the 0 or the 1 position. The operation that allows to bring the information carried by a qubit to a result, thus something interpretable is called *measurement*. Measurements will be implemented on the relevant qubits of our quantum computer so that we can see at the end of the algorithm which is the real quantum state in which we are arrived. By measuring a qubit we are asking which is the highest probability of the qubit for being 1 or 0. Measurement is represented with the following block in a quantum circuit:

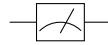


Figure 12: The measurement gate

It is important to understand that the outcome of a measurement will then be brought on a classical register where we will collect all the partial results measured from the relevant qubits of our circuit. As we saw in the introduction, the sad part of the story is that once we have measured a qubit it collapses to what we have called a "boring" unitary bit. This concept needs to be remembered also for what concerns the problem of copying a qubit that we will see later. Hence it is important to pose another important conclusion to the list.

Conclusion 3: *The state of the quantum system after a measurement collapses to a linear combination of only those basis states that are consistent with the outcome of the measurement. The original quantum state is no longer recoverable.*

1.2 The Quantum Computer

We now have all the necessary to define the quantum device, and thanks to this formalism we will be also able to define further important conclusions, obtained by putting all together what we have seen so far.

A quantum computer is not that different from how we consider a classical one in a general point of view. The device has a state and it evolves in other states by performing operations. The model of computation that is considered to formalize it is the quantum circuit model, which works as follows:

- The quantum computer has a *state* that is stored in a quantum register, initialized in a certain way at the beginning of the computation
- *Quantum operations* applied on a state allow the quantum computer to evolve from a state to another
- At the end of the computation the information stored in the quantum register, thus the final state, contains the result

Now it should be clear what to implement a quantum algorithm means and how the computation of a quantum algorithm is carried out on a quantum computer. Quantum algorithms are implemented on a quantum computer that provides a certain number of qubits to store the state. It is the programmer, knowing the quantum operations that we described in the previous section, who will make the states evolve in order to reach the one that contains the result. The best the quantum computer is realized the smaller the noise in the circuit will be. This means that the state will evolve with higher probability to the following expected one and that the measurements on the final state will return the expected result.

To conclude with this part and start to study our first relevant quantum algorithm, we still have to conclude two more important features related properly to quantum algorithms. We have seen that measuring a quantum state makes it collapse so that it can not be reused. It seems natural hence to look for a way to copy a quantum state, for example to continue with another computation. However, in particular because of the property of quantum gates to be unitary matrices it turns out that cloning a quantum state is not possible. This yields to another important conclusion that we have always to keep in mind while defining quantum algorithms.

Conclusion 4: *It is impossible to clone quantum states.*

Hence, whenever we run a circuit that produces an output quantum state, in general we can reproduce the output quantum state only by repeating all the steps of the algorithm. Another important aspect concerns the initialization of the quantum computer before starting to compute the algorithm. As we mentioned when we defined the Hadamard gate, by applying multiple Hadamards

on the entire quantum state of the quantum computer we set the state to a uniform superposition. It is here that we can feel another time the extreme power of quantum computing with respect its classical counterpart.

Conclusion 5: *Applying operations on a quantum device whose state is in a uniform superposition allows to apply them simultaneously to all possible binary strings thanks to linearity.*

1.3 Grover's Algorithm

Being now able to understand what a quantum algorithm is and how it can be implemented on a quantum circuit, we can start to consider one of the most important algorithms that is used in quantum computing. Thanks to Grover's algorithm we will be able to appreciate how a real quantum algorithm is implemented and why we actually obtain a speedup with respect to its classical counterpart. This algorithm, as it is described in the paper by Grover ([6]), is a search algorithm in particular focused on looking for a certain element in a database. Hence the most important feature we will conclude is a **quadratic** speedup with respect to the most efficient search algorithm that we can implement classically. To exploit this feature we will use Grover's algorithm to solve the SAT problem and compare the computation complexity with a very efficient classical algorithm found on the web.

Basic Idea: *start with the uniform superposition of all basis states, and iteratively increase the coefficients of basis states that correspond to binary strings for which the unknown function gives output 1.*

This means that iteratively we will perform a set of operations that allow to increment the coefficient of the correct solution that will be retrieved at the end thanks to the measurement operation. The algorithm requires $q = n + 1$ qubits to perform the basic operations but we will see in our implementation, as in many generalized cases happens, that some more qubits will be needed in order to perform intermediate steps that allow to generalize the algorithm. The qubits used to perform intermediate operations are typically called *ancilla qubits* and it is very important to manage them as best as we can to prevent their exponential growth, caring of the *no-cloning* conclusion we defined before.

Outline: *Grover's algorithm starts with the uniform superposition of all basis states on n qubits. The last $n + 1$ qubit is already an auxiliary qubit and it is initialized as $H|1\rangle$. Thus we obtain the initial quantum state $|\psi_0\rangle$ and the following are iteratively repeated:*

1. *Flip the sign of the vectors for which U_f gives output 1.*
2. *Invert all the coefficients of the quantum state around the average coefficient.*

*A full cycle of these iterations increases the coefficients of the vectors that have been flipped at the first step. Continuing to iterate this cycles allows to make the coefficient get always nearer to 1 until we perform the measurement to obtain the result that is the closest to 1. This phenomenon is known as **Amplitude Amplification**, the basis of Grover's algorithm but also the most important thing to take care of (doing too many iterations may lead to solutions that are completely wrong!).*

Considering now the function we are interested in to implement Grover's algorithm we first consider how a classical search would allow us to find a solution and then we define the 3 steps that are necessary to implement the quantum algorithm. Let $f : \{0,1\}^n \rightarrow \{0,1\}$, and assume that there exists a unique vector $\underline{s} \in \{0,1\}^n : f(\underline{s}) = 1$, i.e., there is a unique element in the domain of the function that yields output 1. We want to determine \underline{s} . The algorithm will run assuming that the function f is encoded by a unitary matrix that we will consider as \mathcal{U}_f .

1.3.1 Classical Grover

Now that we know how the search problem is defined, given the function f introduced above, we have clear in mind that classical search cannot do better than $\mathcal{O}(2^n)$ operations. Indeed, as we will precisely describe at the beginning of the SAT implementation section (3), we can design classical algorithms in both deterministic or nondeterministic paradigms. Nondeterminism is obtained with randomization and it provides in general more efficient algorithms but very complex to be designed and debugged. Any deterministic classical algorithm may need to explore all 2^n possible input values before finding \underline{s} : given any deterministic classical algorithm, there exists a permutation π of $\{0,1\}^n$ that represents the longest execution path of such algorithm. Then, if $\underline{s} = \pi(\underline{1})$ the algorithm will require 2^n **queries** (important to understand query complexity described at 3) to determine the answer, which is clearly the worst case. At the same time, a randomized algorithm requires $\mathcal{O}(2^n)$ function calls to have at least a constant positive probability to determine \underline{s} ; the expected number of function calls to determine the answer is approximately $2^{2^{-1}}$.

1.3.2 Quantum Grover

In the beginning of the section we have provided all the elements that we need to realize Grover's algorithm, as we said there are 3 steps that are iteratively repeated in order to obtain the best solution possible. We now want to give some more details about these steps:

1. **Initialization:** The algorithm is initialized by applying the operation $\bigotimes^{n+1} \mathcal{H}(\bigotimes^n I \otimes X)$ onto the state $|\underline{0}\rangle_{n+1}$. This brings the basis state in uniform superposition and the initial coefficients of the state $|\psi\rangle$ are real numbers.

2. **Sign Flip:** To flip the sign of the target state $|\underline{s}\rangle_n \otimes \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$, we apply \mathcal{U}_f to $|\psi\rangle$. In fact we can always think of the $n + 1$ qubit as being in the state $\frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$ and *unentangled* from the rest of the qubits, with the sign flip afflicting only the first n qubits. Therefore, the state we obtain by applying \mathcal{U}_f to $|\psi\rangle$ is the same as $|\psi\rangle$ except that the sign of $|\underline{s}\rangle_n \otimes \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$ has been flipped.
3. **Inversion about the average:** This is the last step we need to modify the coefficients of our state in order to find the solution. This is a unitary operation that can be expressed by the following matrix:

$$W = \begin{pmatrix} \frac{2}{2^n} - 1 & \frac{2}{2^n} & \cdots & \frac{2}{2^n} \\ \frac{2}{2^n} & \frac{2}{2^n} - 1 & \cdots & \frac{2}{2^n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{2}{2^n} & \frac{2}{2^n} & \cdots & \frac{2}{2^n} - 1 \end{pmatrix} = \begin{pmatrix} \frac{2}{2^n} & \frac{2}{2^n} & \cdots & \frac{2}{2^n} \\ \frac{2}{2^n} & \frac{2}{2^n} & \cdots & \frac{2}{2^n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{2}{2^n} & \frac{2}{2^n} & \cdots & \frac{2}{2^n} \end{pmatrix} - \otimes^n I$$

where the denominator $\frac{1}{2^n}$ computes the average coefficient, the numerator 2 of the fraction takes twice the average, and finally we subtract the identity to subtract each individual coefficient from twice the average. Thanks to the definition of the Hadamard gate we can show that W can be written as:

$$W = (-\otimes^n \mathcal{H})D(\otimes^n \mathcal{H})$$

where D is a diagonal matrix $\text{diag}(-1, 1, \dots, 1)$ of size 2^n . This expression trivially shows that W is unitary in fact $\otimes^n \mathcal{H}$ is unitary and D is diagonal with ones on its diagonal. Thus we can summarize the analysis of the inversion of the average by concluding that it can be performed by applying $W = (-\otimes^n \mathcal{H})D(\otimes^n \mathcal{H})$ to the n qubits of interest.

2 Computational Theory

It is important to give a brief introduction also about computational theory, in particular to understand how we are able to compare the execution of a classical with a quantum algorithm but also to grasp why the SAT has been chosen as the reference algorithm to carry on the entire paper. In this section we want to address quickly the basic computational definitions we need to evaluate the complexity of a classical algorithm and understand how these definitions are applied in quantum algorithms. We will see that the quadratic speedup we achieved thanks to Grover's algorithm provides enhancements in a slight different computation complexity definition with respect to the one we are used to consider.

In the classical computational theory we are interested in determining two major types of issues:

1. Evaluate the *complexity* of a given algorithm A to solve a problem P
2. Evaluate the *inherent difficulty* of a given problem P

2.1 Algorithm Complexity

As we have studied, being computer scientist engineers, one of the most important things to solve a problem is to choose the best algorithm in terms of performances to solve each of the possible instances of the problem. But how do we evaluate the performances of an algorithm?

Definition (Computing Time): *The computing time of an algorithm is evaluated in terms of elementary operations needed to solve a given instance I . In this definition we assume that all elementary operations require one unit of time.*

It is clear that the number of elementary operations depends on the size of the instance I . The bigger the instance it is, the higher the computing time will be. The **size** of an instance can be considered as the number of bits needed to encode that specific instance. There are two features of complexity that influence the choice of an algorithm:

- **Time Complexity:** we want to identify a function $f(n)$ such that, for every instance I of size at most n the number of elementary operations to solve the instance is smaller or equal to $f(n)$.
- **Space Complexity:** we want to identify a function $g(n)$ such that, for every instance I of size n the number of bits needed to encode it is smaller or equal to $g(n)$.

In the comparison between the classical and the quantum solver for the SAT we will be mainly interested in time complexity, this is why we are going to focus on it in the next sections. However we do not have to neglect space complexity, as we saw at the beginning of the paper it is still one of the main issues in quantum computing, in particular because of its young technology; this is also why we prefer to focus on time complexity where we can see in practice the quadratic speedup that we were mentioning.

As we know, the function $f(n)$ is expressed in asymptotic terms using the *big-O notation*, and thanks to it we are able to make a very important distinction in classical algorithms.

Definition (Polynomial Algorithm): *an algorithm is polynomial if it requires, in the worst case, a number of elementary operations $f(n) = \mathcal{O}(n^k)$ where n is the size of the instance and k a constant.*

Definition (Exponential Algorithm): *an algorithm is exponential if it requires, in the worst case, a number of elementary operations $f(n) = \mathcal{O}(2^n)$ where n is the size of the instance.*

We always look for polynomial algorithms to solve our problems but for some interesting case we haven't still found a better solution than an exponential algorithm. As we will see in section 3.1 very efficient algorithms have been studied to reduce the complexity of the solver for the SAT problem but the best we can achieve is to decrease the base of the exponential until $\mathcal{O}(1.30704^n)$ (randomized classical algorithm). This is already an extreme enhancement decreasing almost to 1 which would lead the algorithm to be constant time; now considering the quadratic speedup we can achieve with quantum computing we can grasp once more its advantages.

2.2 Inherent Difficulty

In this section we are going to formalize the theory that studies which of the algorithms is the best to solve a specific problem. Intuitively we are looking for the complexity of the most efficient algorithm that could ever be designed for that problem.

Definition (Simple Problem): *an algorithm P is polynomially solvable ("easy"), if there is a polynomial-time algorithm providing an optimal solution for every of its instances.*

Thus which are difficult problem? Actually we will give a more precise definition by using the *hard* word, in fact it does not suffice to have an algorithm that solves an algorithm in exponential time to say that it is difficult. It may be that we have not still found the algorithm able to solve it in polynomial

time! There are several problems that seem to be "difficult", for example: *the Travelling Salesman Problem, the SAT, graphs colouring...*

2.3 NP-completeness Theory

To continue the study of algorithms in particular with the definitions that we now know about their performances, we want to reach the definitions of the complexity classes P and NP . With this classification we will finally be able to understand why we decided to choose the SAT as the problem to carry on this paper. To simplify the discussion we will consider the *recognition version* of a problem without losing any kind of generality.

Definition (Recognition Version): *a recognition problem is a problem whose solution is either "yes" or "no".*

To every optimization problem we can associate its recognition version, always considering the following assumption: any optimization problem is at least as difficult as its recognition version. This shows why concluding that a recognition problem is "difficult" implies that its original version is "difficult" too. Now that we know the definition of recognition problems we can provide the first very important definition of the complexity class in which we hope to find the algorithm that solves our problem.

Definition (Complexity P-class): \mathcal{P} denotes the class of all recognition problems that can be solved in polynomial time.

An important conclusion shows that \mathcal{P} can be formally defined in terms of deterministic Turing machines. This is very interesting for us, in particular recalling what we concluded at 1.1.5. The other complexity class we do not hope to retrieve our algorithm is a superclass of the one we have just presented and it is defined as follows.

Definition (Complexity NP-class): \mathcal{NP} denotes the class of all recognition problems such that, for each instance with "yes" answer, there exists a concise proof which allows to verify in polynomial time that the answer is "yes".

A trivial example is the 3-SAT recognition problem: given a clause we are able in polynomial time to establish whether the problem is satisfiable or not. From the definitions we have just provided it is clear that the \mathcal{NP} includes the \mathcal{P} , but nothing more can be said among these two classes. It is in fact one of the *millennial problems* the one to establish whether \mathcal{NP} coincides with \mathcal{P} or that the inclusion is strict.

To understand the decision of choosing the SAT as the reference algorithm we still need some definitions. In particular we will see that thanks to the SAT

we are able to classify several algorithms in the classes we have just presented. In order to do so we need a criterion that allows to classify an algorithm thanks to its intrinsic difficulty, identifying the most difficult ones in \mathcal{NP} .

Definition (Polynomial Time Reduction): Let $\mathcal{P}_1, \mathcal{P}_2 \in \mathcal{NP}$, then \mathcal{P}_1 reduces in polynomial time to \mathcal{P}_2 ($\mathcal{P}_1 \propto \mathcal{P}_2$) if there exists an algorithm to solve \mathcal{P}_1 which:

- (i) uses (once or several times) a hypothetical algorithm for \mathcal{P}_2 as a subroutine
- (ii) the algorithm for \mathcal{P}_1 runs in polynomial time if we assume that the algorithm for \mathcal{P}_2 runs in constant time

Thanks to this first definition we can draw a first conclusion that allows us to classify an algorithm in the \mathcal{P} class: if $\mathcal{P}_1 \propto \mathcal{P}_2$ and \mathcal{P}_2 admits a polynomial-time algorithm, then also \mathcal{P}_1 can be solved in polynomial time. In formulas:

$$(\mathcal{P}_1 \propto \mathcal{P}_2) \wedge (\mathcal{P}_2 \in \mathcal{P}) \implies \mathcal{P}_1 \in \mathcal{P}$$

Now that we know how to classify polynomial time algorithms we need a similar criterion also for the problems that belong to the \mathcal{NP} class. The discussion is now more delicate, in particular we need another definition in order to identify a very special subclass containing the most relevant difficult problems contained in \mathcal{NP} .

Definition (\mathcal{NP} -complete Problems): a problem P is \mathcal{NP} -complete if and only if:

- (i) \mathcal{P} belongs to \mathcal{NP}
- (ii) every other problem $P' \in \mathcal{NP}$ can be reduced to P in polynomial time ($P' \propto P$)

This definition has a very important consequence related to the millennial problem we defined before. If there existed any polynomial-time algorithm for any \mathcal{NP} -complete problem, then all problems in \mathcal{NP} can be solved in polynomial time. Thus we would have demonstrated that: $\mathcal{P} = \mathcal{NP}$. Many studies show that the equality is very unlikely, thus it is conceived that the relation between the two classes is a strict inclusion. However, this allows us to say that \mathcal{NP} -completeness provides a strong evidence that a problem is *inherently difficult*. Now, to show that $P_2 \in \mathcal{NP}$ is \mathcal{NP} -complete it suffices to show that an \mathcal{NP} -complete problem P_1 reduces in polynomial time to P_2 . More formally:

$$P \propto P_1, \forall P \in \mathcal{NP} \wedge P_1 \propto P_2 \xRightarrow{\text{transitivity}} P \propto P_2, \forall P \in \mathcal{NP}$$

But does there exist an \mathcal{NP} -complete algorithm that allows us to exploit this implication? The answer was provided by Stephen Arthur Cook in 1971 when he proved that the SAT is \mathcal{NP} -complete ([3]). This conclusion has brought

enormous advantages to the complexity theory, an example are the 21 discrete optimization problems shown to be \mathcal{NP} -complete by Richard Karp in 1974. To conclude this section thanks to the definition of \mathcal{NP} -complete problems we can give a further classification.

Definition (\mathcal{NP} -Hard Problem): *a problem is \mathcal{NP} -Hard if every problem in \mathcal{NP} can be reduced to it in polynomial time (even if it does not belong to \mathcal{NP}).*

This last definition allows us to formulate a very important observation that relates one more a problem with its recognition version: all optimization problems with an \mathcal{NP} -complete recognition version are \mathcal{NP} -Hard.

It should be now clear why we decided to choose the SAT as the example to carry on the entire comparison between how its classical solver performs with respect to its quantum counterpart. The conceptual part of this paper is now arrived to its epilogue; we are now ready to start the real comparison. The next two subsections are used to introduce first the formal definition of the SAT problem and then to understand how quantum algorithms impact onto the \mathcal{NP} class.

2.4 SAT Problem

Let $X \equiv \{x_1, x_2, \dots, x_n\}$ be a set. Then x_k and its negations \bar{x}_k ($k = 1, 2, \dots, n$) are called literals and the set of all such literals is denoted by $X' = \{x_1, \bar{x}_1, \dots, x_n, \bar{x}_n\}$. The set of all subsets of X' is denoted by $\mathcal{F}(X')$ and an element $C \in \mathcal{F}(X')$ is called a clause. We consider a truth assignment to all variables x_k . If we can assign the truth value to at least one element of C , then C is called satisfiable. When C is satisfiable, the truth value $t(C)$ of C is regarded as true, otherwise, that of C is false. Take the truth values as true "1" while false "0". Then

$$C \text{ is satisfiable} \iff t(C) = 1$$

Let $L = \{0, 1\}$ be a Boolean lattice with usual join \vee and meet \wedge , and $t(x)$ be the truth value of a literal x in X . Then the truth value of a clause C is written as

$$t(C) = \vee_{x \in C} t(x)$$

Further a set \mathcal{C} of all clauses C_j ($j = 1, 2, \dots, m$) is called *satisfiable* if and only if the meet of all truth values of C_j is 1:

$$t(\mathcal{C}) = \wedge_{j=1}^m t(C_j) = 1$$

Finally the formal definition of the most general formulation of the SAT problem is written as follows.

Definition (SAT Problem): *given a set $X \equiv \{x_1, x_2, \dots, x_n\}$ and a set $\mathcal{C} = C_1, C_2, \dots, C_m$ of clauses, determine whether \mathcal{C} is satisfiable or not.*

Hence it is the problem to determine whether it exists a truth assignment to make \mathcal{C} satisfiable. As we have already mentioned the satisfiability problem can also be described as the problem of asking if there exists an assignment to the literals of its clauses that allows to make the *Conjunctive Normal Form* to have value 1. The CNF also called *Product of Sums* (POS) is obtained by the conjunction of the clauses written as the disjunction of their literals.

Now that we now the know the formal definition of the SAT it is important to understand some of its specific formulations, in particular the ones that we considered during this study and that I decided to implement for the comparison. As it is reported in almost every paper, the SAT is in fact denoted as k-SAT. The k indicates the number of variables of the SAT instance that is considered (n in the definition above); if we look at the example 1.1.5 we have a 3-SAT problem with 2 clauses. The precise definition of the implementations of the SAT that we considered are presented in the last section (3), here it is the list of all its formulations and some important features of each in particular with respect their complexity:

- At first it is important to understand that the exponential growth of the instance in modern solvers for an SAT instance is caused by the number of variables chosen. Thus the highest the k will be the more time will be required by our solver to solve the problem. The number of clauses instead do not influence the time complexity while we will need more space to encode every additional clause; but let's focus always on time complexity
- The recognition version of every k-SAT instance can be solved in polynomial time. Thus we are able to design an algorithm that is able to answer "yes" when a given clause makes the formulation satisfiable or "no" otherwise, in polynomial time.
- It has been proved that the general versions of th 1-SAT and the 2-SAT versions can be solved in polynomial time. The solvers we implemented are generalized to the case of k clauses as we could compare also the polynomial time solutions; we will see that while in this case the enhancement provided by the quantum version is not that significant we perceive problems for the exponential growth of the size of the instance of the quantum device!
- The 3-SAT version is the simplest and significant case from which we can start the comparison. As we can see in the literature it is the most studied case and all the efficient solvers that have been implemented consider it as reference. It would be in fact sufficient to find an algorithm that solves it in polynomial time and generalize it for its further formulations.

- The most general quantum solver of an SAT formulation combines classical efficient algorithms with Grover's search and qiskit version is already the best. We considered it to make the general comparison between k-SAT problems while we studied a simplified version to see in action the specific quantum steps of Grover's algorithm.
- The exactly-1-3-SAT problem is formalized as follows:

INPUT: SAT formula in conjunctive normal form $\bigwedge_{j=1}^m C_j$ over n Boolean variables x_1, x_2, \dots, x_n with 3 literals per clause C_1, C_2, \dots, C_m

OUTPUT: Does there exist an assignment x_1, x_2, \dots, x_n such that every clause C_1, C_2, \dots, C_m has exactly one *true* literal?

The exactly-1-3-SAT has been proved to be an \mathcal{NP} -Hard problem.

2.5 Can we solve NP-hard problems?

In [10, G.Nannicini, 2020] is highlighted an important remark that we have now to consider, finally competing with both the quantum computing world and the computational complexity theory. The answer to this section can be included in the important conclusions we are highlighting in this paper, in fact we can formulate it as follows.

Conclusion 6: *even if we can easily create a uniform superposition of all basis states, the rules of measurement imply that using just this easily-obtained superposition does not allow us satisfactorily solve \mathcal{NP} -complete problems, such as, for example, SAT.*

As we explained in Grover's algorithm to solve the SAT problem (section 1.3), considering a circuit of $q = n + 1$ qubits and performing the measurement of this state will return a binary string that satisfies the SAT formula if and only if the last qubit has value 1 after the measurement. This happens with a probability that depends on the number of binary assignments that satisfy the formula. If the SAT problem at hand is solved by exactly ϵ assignments out of 2^n possible assignments, then the probability of finding the solution after one measurement is $\frac{\epsilon}{2^n}$: we have just randomly sampled a binary string hoping that it satisfies the SAT formula. Clearly, this is not a good algorithm. In fact, we can from this draw another important conclusion to be considered with respect to the complexity classes that we identified.

Conclusion 7: *in general solving \mathcal{NP} -Hard problems in polynomial time with quantum computers is not believed to be possible.*

The literature shows that the BQP class, the class of problems solvable in polynomial time by a quantum computer with bounded error probability,

does not contain the class \mathcal{NP} . Of course the proof is not available yet, because showing that $\mathcal{NP} \not\subseteq BQP$ would resolve the millennial problem **P vs NP**. Even if we cannot solve all "difficult" problems in polynomial time using a quantum computer, we will see with the implementation of our SAT solver a significant speedup from every other classical implementation. The basic principle, as we already anticipated while describing Grover's algorithm, is to start with a uniform superposition of basis states, then apply operations that make the basis states interact with each other so that the modulus of the coefficients for some basis states increase, which implies that the other coefficients decrease. Performing a measurement will then reveal the solution to the problem at hand, or some useful information about the solution, with high probability. As we will see in the results of our generalized implementation the result is very noisy but still significant as we have verified that for several runs it still provides with the highest probability always the same satisfiable formula.

3 SAT Implementation

Conclusion n:

3.1 Classical

3.2 Quantum

3.3 Classical vs. Quantum

4 Conclusions

Appendices

A Qiskit

Qiskit is an open-source quantum computing software development framework for leveraging today's quantum processors. In the literature we find several different frameworks that allow to work with quantum computers, characterized by different implementation languages but from the quantum algorithm designer point of view, with different interfaces to the quantum computers and different simulators. In this paper I decided to use qiskit because it is the most spread one, with a very good documentation and a wide community that updates every day the issues. Moreover, being realized by IBM it provides an experience called *IBM quantum experience*(IBMQ) which allows to run the algorithms directly on a real quantum computer rather than on a simulator. Qiskit is realized in python and this may suffice to explain why it has been chosen among the other solutions...

Thanks to its textbook [1] it provides a very wide and complete description on how to approach the area of quantum computing. Starting from the introductory subjects needed to face the basic principles of quantum physics it increasingly introduces how the library realizes the quantum operations that a user can define to implement its algorithms. Several examples are provided with jupyter notebooks where each step can be analysed in detail understanding how and what each instruction from the library does. I completely went through the entire notebook for my first approach to quantum computing and there I found the example I was looking for to produce this paper which is meant to simplify even more the introduction to this area, using a concrete example decomposed in different steps each introducing a particular aspect of how to use the library. Thanks to this notebook I realised the notebook visible in the repository at CODE/QUANTUM/QUANTUMSAT.IPYNB where I replicated the solution of the SAT problem by using the grover solver implemented in the library. Thanks to the quantumSat.ipynb notebook I understood how difficult it is to realize the real implementation of a solver, interfacing directly with the quantum device, this helped me a lot to find a good way to realize an implementation of the same algorithm in a simpler case to solve the SAT problem. All the notes and some additional examples I realized during this first approach are present in the notebook.

In conclusion it is very important to remark that qiskit is an open source software where a great community is very available to help whoever wants to start studying quantum computing. Very interesting videos were also realized to give a further way to approach the library, thanks to them I learnt how to become a contributor of an open-source project and I became a contributor by solving an issue that can be visible at the following url: <https://github.com/Qiskit/qiskit-terra/pull/3751>.

B Code with Qiskit

This section contains the list of the snippets of code needed to represent the examples used in the paper. To replicate them consider to import the following libraries:

```
from matplotlib import pyplot as plt
import numpy as np
from qiskit import *
from qiskit.visualization import *
```

```
1. figure1 = plot_bloch_vector([1, 0, 0], title="zeroket")
   figure1.savefig('zeroket.png')

   figure2 = plot_bloch_vector([-1, 0, 0], title="oneket")
   figure2.savefig('oneket.png')

   figure3 = plot_bloch_vector([0, 1, 0], title="xket")
   figure3.savefig('xket.png')

   figure4 = plot_bloch_vector([0, 0, y], title="yket")
   figure4.savefig('yket.png')

2. # First I define the dimensions of the quantum circuit:
   # 3 qubits for the variables +
   # + 5 ancilla qubits for each clause
   # + 1 = 14 qubits
   qc = QuantumCircuit(QuantumRegister(14, 'x'))

   # A quantum circuit once initialized has all qubits set to 0
   # Hence we need to invert them when storing the clauses

   # Encoding of the first clause: C_0 = notX_1, X_2, X_3
   qc.cx(0, 3)
   qc.cx(1, 4)
   qc.x(4)
   qc.x(3)
   qc.x(4)
   qc.x(5)
   qc.ccx(3, 4, 5) # notX_1 or X_2 now encoded on qubit 5
   qc.cx(2, 6)
   qc.x(6)
   qc.x(5)
   qc.x(6)
   qc.x(7)
   qc.ccx(5, 6, 7) # notX_1 or X_2 or X_3 now encoded on qubit 7
```

```

# Encoding of the second clause:  $X_1$ ,  $\text{not}X_2$ ,  $X_3$ 
qc.cx(0, 8)
qc.cx(1, 9)
qc.x(8)
qc.x(8)
qc.x(9)
qc.x(10)
qc.ccx(8, 9, 10) #  $X_1$  or  $\text{not}X_2$  now encoded on qubit 10
qc.cx(3, 11)
qc.x(11)
qc.x(10)
qc.x(11)
qc.x(12)
qc.ccx(10, 11, 12) #  $X_1$  or  $\text{not}X_2$  or  $X_3$  now encoded on qubit 12

# Now that we have both partial results
# we can compute the and of the two clauses
qc.ccx(7, 12, 13)

```

References

- [1] Qiskit textbook. <https://qiskit.org/textbook/>.
- [2] Scott Aaronson. Np-complete problems and physical reality. 2005.
- [3] Stephen A. Cook. The complexity of theorem-proving procedures. 1971.
- [4] V. Kreinovich E. Dantsin and A. Wolpert. On quantum versions of record-breaking algorithms for sat. 2005.
- [5] Richard P. Feynman. Quantum mechanical computers. 1986.
- [6] Lov K. Grover. A fast quantum mechanical algorithm for database search. 1996.
- [7] T. Hertli and R. A. Moser. Improving ppsz for 3-sat using critical variables. 2018.
- [8] S. Tamaki K. Makino and M. Yamamoto. Derandomizing hssw algorithm for 3-sat. 2011.
- [9] K. Kutzkov and D. Scheder. *Using Constraint Satisfaction To Improve Deterministic 3-SAT*. 2018.
- [10] Giacomo Nannicini. An introduction to quantum computing, without the physics. 2020.
- [11] K. Y. Chwa O. Cheong and K. Park. *Algorithms and Computations*. Springer, 2010.
- [12] M. Ohya and N. Masuda. Np problem in quantum algorithm. 2018.
- [13] M. Ohya and Igor V. Volovich. New quantum algorithm for studying np-complete problems. 2018.