An Introduction to Quantum Computing for Computer Scientists, with SAT

Francesco Piro

April 5, 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 $e_1, ..., e_m$ and $f_1, ..., f_n$ 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 $\otimes : V \times W \to V \otimes W$. The vector space $V \otimes W$ has basis $e_i \otimes f_j \forall i = 1, ..., m, j = 1, ..., n$.*

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.i. $\langle\psi| = |\psi\rangle$. 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 represents 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:*

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

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 $\mathcal{U}$ 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. $\alpha_0 = 1,\ \alpha_1 = 0$

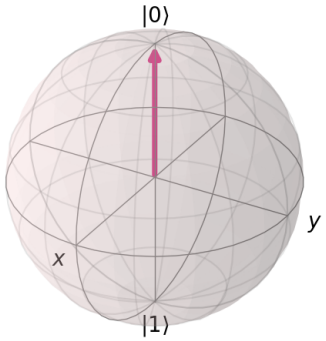

Figure 1: $|\psi\rangle = |0\rangle$

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

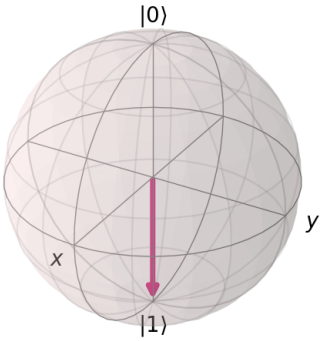

Figure 2: $|\psi\rangle = |1\rangle$

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

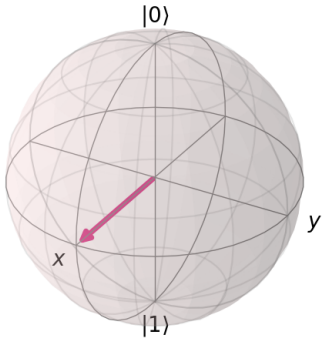

Figure 3: $|\psi\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$

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

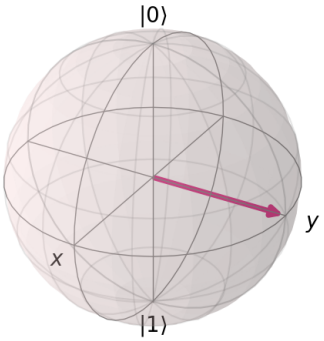

Figure 4: $|\psi\rangle = \frac{1}{\sqrt{2}}(|0\rangle + i|1\rangle)$

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 hyper-sphere (check the plot_bloch_multivector method in the qiskit library). As in classical computer we haver 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.*

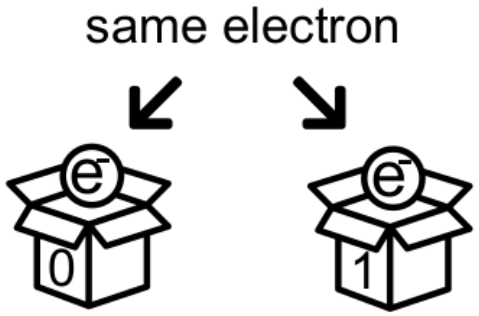

Figure 5: Superposition of the electron

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 se 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* $|\psi_0\rangle \otimes |\psi_1\rangle \otimes \cdots \otimes |\psi_n\rangle$ *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.

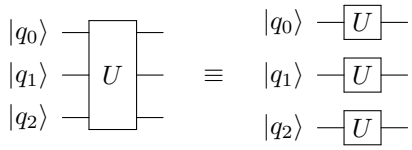

Figure 6: Trivial equivalence of quantum circuits

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.

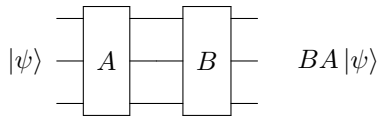

Figure 7: Quantum circuit interpretation

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:*

$$I = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \qquad X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

$$Y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix} \qquad Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$

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 $Z|1\rangle = -1$.
- 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:

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

  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.

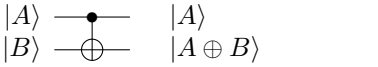

Figure 8: The CNOT gate

- 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.

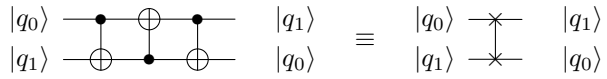

Figure 9: The SWAP gate

- 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.

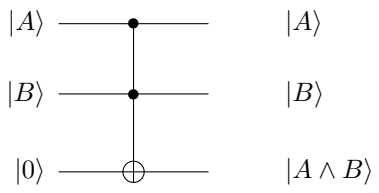

Figure 10: The CCNOT gate

- 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:*

$$C_0 = \{\neg X_1, X_2, X_3\}$$
$$C_1 = \{X_1, \neg X_2, X_3\}$$

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))

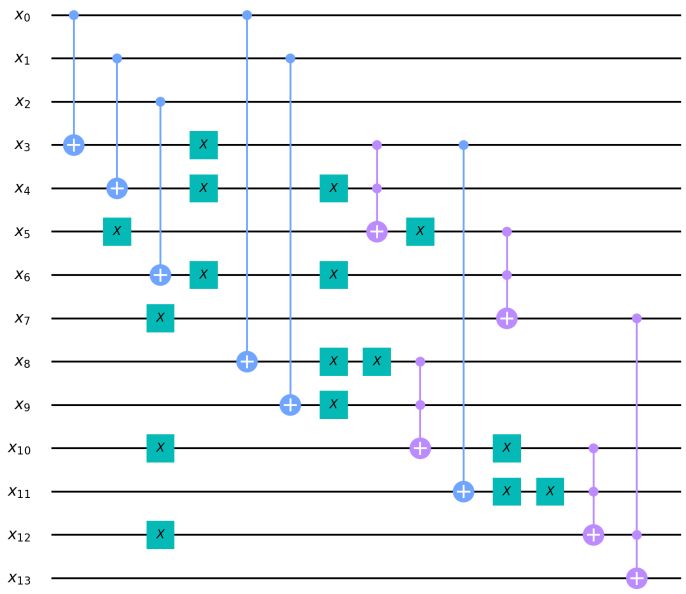

Figure 11: CNF circuit

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 posses 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 ([4]), 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 $f : \{0,1\}^n \to \{0,1\}$, 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 $diag(-1, 1, \cdots, 1)$ of size 2^n. This expression trivially shows that " is unitary in fact $\bigotimes^n \mathcal{H}$ 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 $W = (-\bigotimes^n \mathcal{H}) D (\bigotimes^n \mathcal{H})$ 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 biggest the instance it is, the highest 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 3.2 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 loosing 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:

$$P \propto P_1, \forall P \in \mathcal{NP} \wedge P_1 \propto P_2 \overset{transitivity}{\implies} P \propto P_2, \forall P \in \mathcal{NP}$$

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 ones 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 $X \equiv \{x_1, x_2, \cdots, x_n\}$ be a set. Then x_k and its negations $\overline{x}_k$ $(k = 1, 2, \cdots, n)$ are called literals and the set of all such literals is denoted by $X^{'} = \{x_1, \overline{x}_1, \cdots, x_n, \overline{x}_n\}$. 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

$$C\ is\ satisfiable \iff t(C) = 1$$

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 $(j = 1, 2, \cdots, m)$ 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* $X \equiv \{x_1, x_2, \cdots, x_n\}$ *and a set* $\mathcal{C} = C_1, C_2, \cdots, C_m$ *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 $\wedge_{j=1}^{m} C_j$ over n Boolean variables $x_1, x_2, \cdots, x_n$ with 3 literals per clause $C_1, C_2, \cdots, C_m$

  OUTPUT: Does there exist an assignment $x_1, x_2, \cdots, x_n$ such that every clause $C_1, C_2, \cdots, C_m$ 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 [11, 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 $\mathcal{NP} \nsubseteq BQP$ 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

We have finally reached the heart of this paper. Having now all the necessary fundamental concepts of quantum computing and of the problem we want to study, it is possible to put everything together and make the comparison we have been talking until now. To understand how to compare a classical algorithm with its respective quantum counterpart we will need one last definition and some further considerations on the complexities that are now achieved by the most efficient classical algorithms for the problem of solving the SAT. In this section we will start in the classical (3.2) part to describe these complexities by showing in the end the characteristics of the algorithm we decided to use in order to realize the comparison. Then in the quantum part (3.3) in the same way we describe how the complexities are enhanced thanks to the implementation we realized using Grover's search and then which are the elements of the algorithm we realized for the comparison. In the end, thanks to all the information obtained from the two implementations we will draw the comparison.

As we mentioned, before going deep inside the implementation of the algorithms we need to define a concept that may result confusing for what we have said until now about computational complexity. However, we needed what studied in section 2 to identify the main issues of complexity in both a classical and quantum world, but to conclude our work and realize a comparison between the SAT solvers we need to consider another type of complexity in particular related to all those algorithms that belong to the family of **searching**. As we already said, thanks to Grover's algorithm we are able to perform a search of our solution to the SAT problem by exploiting quantum physics features and to compare the quadratic speedup obtained we need to consider the following very important conclusion.

Conclusion 8: *the complexity of an algorithm that belongs to the search class is determined only in terms of the number of the calls to the function f. This model is known as query complexity, because it defines the complexity of an algorithm as the number of queries to a given function (in this case, f). Query complexity is used as a model to answer important theoretical questions. There are many quantum algorithms that yield speedups under the query complexity model, but some others, e.g., Shor's algorithm, are faster than (known) classical algorithms under the more traditional computational complexity model, i.e., number of basic operations.*

3.1 SAT Instances

3.2 Classical

The SAT problem, thanks to the importance it provided to the field of computational complexity when Cooks demonstrated its belonging to the $\mathcal{NP}$-complete class, became the subject of several research studies. The researches followed all the same objective, that is the one of obtaining a polynomial solver for any instance of the SAT problem in order to prove the millennial problem of P vs NP. However nowadays we are not still able to find such an algorithm and thus a solution to the famous problem.

Every year it is held a competition where the most competent computer scientists in the world compete in order to implement the best SAT solver in terms of computational complexity. Thanks to this event and to the huge research behind this problem, starting from deterministic algorithms forward to nondeterministic ones surprising results were obtained. The exciting part of these exponential time algorithms is their ability to reduce always more the basis of the general starting point of every search algorithm: 2^n. As we see in the following list for the 3-SAT problem of deterministic and nondeterministic algorithms different approaches allow to reduce the 2 until the best result reached so far of: $\mathcal{O}(1.30704^n)$.

- [8, Kutzkov K., Scheder D.] $\mathcal{O}(1.439^n)$ **[Deterministic]**
- [10, Makino K., Tamaki S., Yamamoto M] $\mathcal{O}(1.3303^n)$ **[Derandomized]**
- [7, Iwama K., Seto K., Takai T., Tamaki S.] $\mathcal{O}(1.32113^n)$ **[Randomized]**
- [6, Hertli T., Moser R. A., Scheder D.] $\mathcal{O}(1.32065^n)$ **[Randomized]**
- [5, Hertli T.] $\mathcal{O}(1.30704^n)$ **[Randomized]**

3.2.1 Solver

We can now introduce the version of the solver considered to make the comparison between the classical and the quantum implementations. The aim was to find the best solver for the comparison, but in order to realize one as efficient

as the those listed above would have required mathematical skills not relevant for our study. Hence I decided to start with a solver that is based on *Knuth* SAT0 (`https://www-cs-faculty.stanford.edu/~knuth/programs/sat0.w`) solution in order to have a first comparison to test some particular instances of the k-SAT problem (with k ranging from 2 to 4). The solver implementation is available at `https://github.com/sahands/simple-sat`, even if based on the first of Knuth's SAT-solvers series it exploits some particular features that allow to handle in the best way the algorithm resolution over a general k-SAT formulation.

The algorithm is a watch-list based backtracking algorithm based on the following features that allow to obtain an efficient version in terms of both time and space complexities:

(i) To encode the formulation of a general k-SAT instance we need to represent it with the meaning of a CNF. Thus a set of clauses each containing variables, negated or not, in conjunction. To do so we may think to realize a list of all the clauses, but this does not help when in the algorithm we want to look for the variables in order to find a satisfying formula. Instead, unique number is assigned to each variable once it is encountered, starting from 0 and counting up, using a dictionary to keep track of the mapping. Then the positive variables encoded as number x will be stored at position $2x$ and the negated one will be present at $2x + 1$.

(ii) It is a backtracking algorithm in which we try to assign true or false values to all variables, starting from the one at position 0 up to the last $n - 1$ one. The entire search space is of size 2^n but by pruning, only the best branches of the search tree will be explored.

(iii) Watch-lists allow to manage easily the backtracking mechanism. For each clause to be satisfied we need in fact one literal to be satisfied. As such, we can make each clause watch one of its literals, and ensure that the following **invariant** is maintained throughout the algorithm: *All watched literals are either not assigned yet, or they have been assigned true.* Then we proceed assigning true or false values to variables, starting from 0 up to $n - 1$. If we successfully assign true or false to every variable while maintaining the above variant, then we have an assignment that satisfies every clause.

(iv) Every time we assign a value to a variable we have to ensure that the watch-list is updated accordingly. To do so efficiently, we need to keep a list of clauses that are currently watching a given literal. This is done thanks a double-ended queue which provides the best performances.

(v) During a backtracking step, assignments only go from 0 or 1 to None, the watch-list does not to be updated at all to maintain the invariant. This is why the watch-list is the best approach and it means that the backtracking step will simply be changing the assignment of a variable back to None.

(vi) Both recursive and iterative versions are present, we used the iterative one for out tests mainly for what Knuth says about his dislike to recursion. The SAT0 is in fact implemented in an iterative way.

3.2.2 Significant Results

3.3 Quantum

In this section we finally provide the description of the solvers we considered to realize the comparison. In order to face the implementation of a solver for the SAT problem we go through three different approaches that allow to understand how the final algorithm has been implemented. First with a jupyter notebook it is possible to understand the comparison from the most general formulation of the problem, thus a k-SAT instance. In the notebook we used qiskit's implementations to solve a big instance of the problem in order to see how it performed in a very complicate example. The decision version implementation instead is used to make some practice, in particular for the problem of realizing manually multiple qubit gates that the library does not support. In the end the exactly 1 k-SAT solver is the most general implementation we were able to realize, generalizing the example provided by [11, Nannicini G.]. The following subsections describe precisely how these programs work.

3.3.1 Qiskit's Solver

In the jupyter notebook provided in the repository at Code/Quantum we can find the complete description of how qiskit is used in order to solve an instance of a 4-SAT problem composed of 6 clauses. This complex example has been used to check if the algorithm realized by qiskit was able to solve in an efficient way problems with more than 3 variables. It has been the first approach to the implementation and it has helped us to conclude the following considerations that we used to realize the solvers presented in the following sections.

Thanks to the open source code we were able to study how the grover class is implemented in qiskit. It is important to remark that ideally the code provided is able to solve every instance of a k-SAT with m clauses. What we were interested to understand from this implementation was first how to achieve such a general algorithm in order to use it for our comparison and second how many iterations of grover's step were used in order to find a solution; as we saw in the description of grover's algorithm, in fact, it is very important to choose the correct number of cycles not to compromise the result of the algorithm. Hence we understood that:

- Qiskit's grover solver is based on [2, Boyer M., Brassard G., Hoyer P. and Tapp A.] where it is explained that a good number to achieve almost the certainty of finding a solution, if there were only one, is $\frac{\pi}{4}\sqrt{N}$ (where N is the number of the variables). This led us to understand that typically

one only iteration suffices to find the optimal result of the problem, and that increasing the number N we will need to do more iterations.

- Grover class is not the only one needed to solve the general k-SAT problem, several components are correlated in order to exploit also an optimal counting of the solution by using techniques achieved with some of the Shor's basic components. The high coupling of classes shows that it requires a more complete knowledge than the one described so far to define a solver as complete as the one of the library. This led us first to start with the very simple decision version implementation and then the exactly 1 k-SAT solver, not that fare from the most general version but still very noisy in the solutions.

However to conclude with this first approach to the SAT solver it can be interesting to include the histogram for the results of the instance described in the notebook and draw some considerations on the actual possibility to run this algorithm on a real quantum computer. To obtain significant results it is important at least to consider the transpiled version of the circuit that realizes the implementation of the solver to understand if it can be considered in reality or the results are just theoretical. The first notebook considered a very complex example where 10 clauses were considered to define the 4-SAT problem and theoretically we reached 5 clauses that satisfied the CNF. However the other possibilities were still very noisy and it was difficult to distinguish which were the correct results so that we needed to check the correct solutions with the classical solver that we described before. In addition the transpiled version could not be realized because the actual technology supported by the IBMQ experience allows to build at most 16 qubits circuits. Hence we reduced the number of clauses to 6 and we obtained the following two important results:

1. Three solutions are found in the 6 clauses formulation of the 4-SAT problem and we can clearly distinguish them thanks to the high probability associated to the coefficients related to each of the satisfying formulae found by the algorithm (check in the notebook the complete description of the solutions).

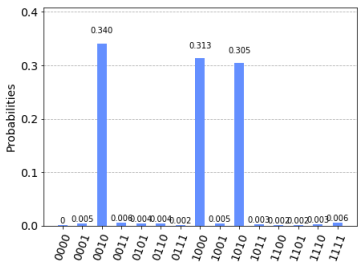

Figure 13: 4-SAT solutions from the jupyter notebook

2. From the transpiled version we were able to retrieve that:

```
gates = OrderedDict([('cx', 523),
                     ('u3', 184),
                     ('u1', 102),
                     ('u2', 50),
                     ('measure', 4),
                     ('barrier', 2)])
depth = 481
```

   The results make evidence of the fact that is not possible to realize physically a quantum device with 523 CNOT gates, 184 U_3 gates... The number of gates in fact is far above the limits regarding decoherence time of the current near-term quantum computer. This is another challenge that we managed in our implementation where we achieved an implementable circuit thanks to a simplified version of the problem.

3.3.2 Decision Version

This implementation was the first independent code I implemented in order to begin working seriously with qiskit. Thus I started to realize this program as a toy example thanks to the study provided in [9, Ohya M. and Masuda N.] and it turned out to be very important conclusion we later considered in the exactly 1 implementation.

The paper cited above shows the quadratic speedup obtained by the quantum version of the implementation of the *decision version* for the SAT. The decision version of the SAT is the problem of determining whether a given formulation can be satisfied or not, without providing the formulae that make the instance satisfiable. The circuit realized by the code is nothing more than a quantum version of the CNF for the circuit. Thanks to the correspondence between the basic operations of the quantum device and the ones of a classical one (described in section 1.1.5) we realized the disjunction of the literals composing the clauses in the end conjuncted all together. Thus with one measurement only on the last qubit on which the result of the formula is stored we are able to determine whether the instance of the SAT provided is satisfiable or not. The algorithm can be implemented for a general instance of every k-SAT problem, but as we can see from the following example it suffices a formulation of a 3-SAT with 4 clauses to obtain an exponential growth in the number of the qubits needed to find a solution. This is due both to the number of variables and to the number of clauses, in fact:

1. The number of variables increases the OR operations to realize the disjunctive form of each clause. Each OR requires 3 qubits to be realized and an additional one to make the disjunction with the previous result.

2. In the same way the number of clauses increases the AND operations to realize the conjunction of all the final disjunctive clauses.

Considering the following formulation of a 3-SAT with 4 clauses:

$$C_1 = \{x_1\}$$
$$C_2 = \{x_2, x_3\}$$
$$C_3 = \{x_1, \neg x_3\}$$
$$C_4 = \{\neg x_1, \neg x_2, x_3\}$$

Corresponding to the Conjunctive Normal Form:

$$CNF = (x_1) \wedge (x_2 \vee x_3) \wedge (x_1 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_2 \vee x_3)$$

Resulted in the very very big circuit that follows which needed several ancillary qubits in order to realize all the AND and OR operations identified. Thanks

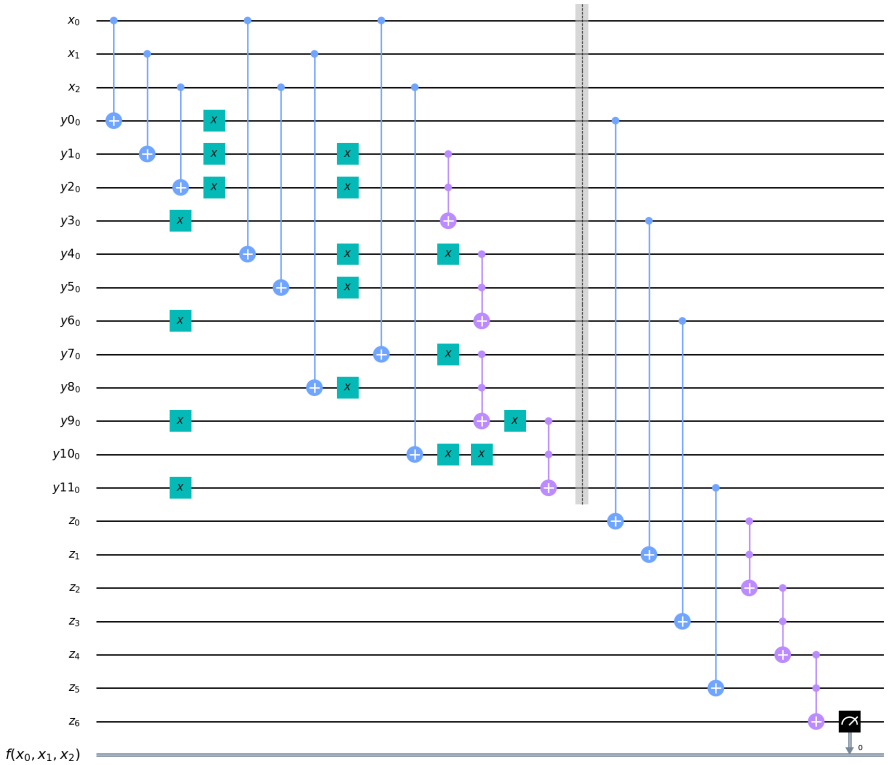

Figure 14: Decision version circuit

to the observation for the exponential grow-up of the ancillary qubits needed to realize multiple gates in a circuit we understood that it was necessary to find a way to reuse the qubits. Considering the important no-cloning principle described in section 1.2 we will se in the next section how we were able to realize a solver that uses less 10 qubits in order to solve the exactly 1 problem for a 3-SAT problem.

3.3.3 Exactly 1 k-SAT

We have finally arrived to the most complete implementation we developed for this paper in order to have an algorithm that was able to solve first a 3-SAT instance and then its generalization to the k case. Before describing how we implemented the steps of Grover's algorithm to solve the problem let's remark the definition of an exactly 1 k-SAT problem.

Definition (Exactly-1-k-SAT): *determining a satisfying assignment containing one true literal per clause.*

This definition helps us to reduce the complexity in which a general k-SAT solver works (see qiskit's implementation at 3.3.1) and allows to realize an algorithm that performs iterations of Grover's algorithm to find the solutions. In the following we will describe precisely how the steps composing Grover's algorithm are implemented as the phenomenon of *amplitude amplification* allows to modify the coefficients of the possible solutions for a given problem and find the best ones. In the end we will draw additional conclusions also for this case as we will have sufficient information to draw at best the comparison between all the algorithms we have described so far.

As described in [11, G.Nannicini, 2020] three steps are needed to implement the solver, and the number of iterations of these steps is fundamental to find solutions that are significant. The following list describes precisely how we implemented the steps in the generalized version of a solver for a general *exactly-1-k-SAT* problem with an arbitrary number of clauses:

1. **Initialization:**
   Grover's algorithm applies to a function with an n-qubit input and a single qubit output. We call `f_in` the input register of the encoded function U_f, of size n, and `f_out` the output register of U_f, of size 1. The initialization of a quantum algorithm, as we have discussed in the conclusion 1.2, consists of applying the Hadamard gate to the quantum state composed of n-qubits in order to obtain the uniform superposition to work with. Thus the piece of code that corresponds to the method:

   ```
   def input_state(circuit, f_in, f_out):
   ```

   Is equivalent to the following circuit:

   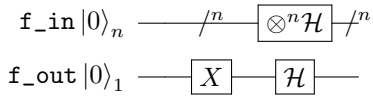

   Figure 15: Initialization step of Grover's algorithm

2. **Black-box function U_f**:
Implementing U_f is the most complex part of the algorithm, in particular now that we want to generalize the case allowing the possibility to have more than 3 variables only. The problem of computing U_f is in fact decomposed for each clause by introducing m auxiliary qubits, one per each clause. And for each clause we build a circuit that bit-flips the auxiliary qubit of the corresponding auxiliary qubit if and only if the clause has exactly one true literal. Finally, we apply a bit-flip on the output register of U_f if and only if all m auxiliary qubits are 1. In figure 16 we can see the example of the circuit that we want to implement in order to flip the last qubit if the clause $x_1 \vee \neg x_2 \vee x_3 \vee \neg x_4$ is satisfied by exactly one literal. The circuit contains NOT gates to encode the negated variables and then to define the double implication of flipping $|y\rangle$ we first use 4 CNOTS to bring each qubit on the last one. Thus we now have $|y\rangle = |y \oplus x_1 \oplus \neg x_2 \oplus x_3 \oplus \neg x_4\rangle$, and with the quadruply-controlled gate leads the auxiliary qubit of the clause to $|y\rangle = |y \oplus x_1 \oplus \neg x_2 \oplus x_3 \oplus \neg x_4 \oplus (x_1 \wedge \neg x_2 \wedge x_3 \wedge \neg x_4)\rangle$. This sets $|y\rangle = |1\rangle$ if and only if exactly one literal is satisfied because as we know from the initialization of every new qubit we always have the starting value of $|0\rangle$. To implement this circuit in Qiskit there is a problem: only double con-

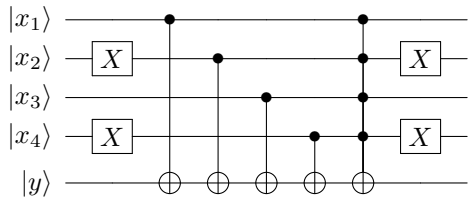

Figure 16: Circuit for the clause $x_1 \vee \neg x_2 \vee x_3 \vee \neg x_4$

trolled gates are implemented in the library. To overcome this issue we can easily concatenate CCNOT gates adding an ancillary qubit for each of the additional variables. In this way we will compute the partial results of the CCNOT between two variables and by storing this value in the first auxiliary qubit we continue with another CCNOT with the next variable. In the end we also perform another CCNOT with all the variables which targeted the auxiliary qubits to reset their state to the initial $|0\rangle$ and reuse them for the circuit of the following clauses. To understand better the mechanism of generalization we can see how the circuit is implemented in our algorithm in figure 17. In the circuit we represent how to realize a quadruply-controlled NOT gate with the reset of the auxiliary qubits initialized as $|0\rangle$. This fragment of the code is the one that substitutes the last gate that we can see above in the real implementation. As we can see we added two ancillary qubits a_1 and a_2 (initialized as $|0\rangle$) to "bring" the final result of the CNOT gate between the four variables x_1, x_2, x_3, x_4. Finally we are able to implement the U_f function trivially by looping over all the clauses and implementing for each of these the circuit of figure

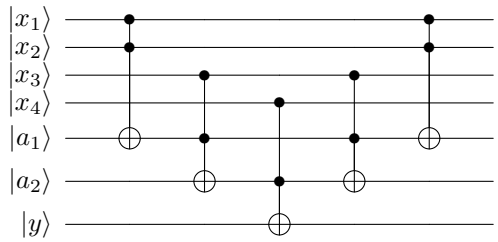

Figure 17: Quadruply-controlled NOT gate

16, setting the auxiliary qubit `aux[k]` to 1 if clause C_{k+1} is satisfied. In the end the method that allows to perform an iteration of the black-box function U_f is:

```
def black_box_u_f(circuit, f_in, f_out, aux, n, exactly_1_k_sat_formula):
```

3. **Inversion about the average:**
   The last operation we need to variate the coefficients of the possible quantum states of the solution is the inversion about the average. As precisely described in section 1.3.2 we can realize this operation with a unitary matrix that we called W. As we see from the definition (remember how to read the algebraic relation of a quantum circuit) we need first to perform Hadamard gates on the entire state, followed by the diagonal matrix D and in the end negated Hadamard gates are performed once more on the entire quantum state. The implementation of W is realized by the method:

   ```
   def inversion_about_average(circuit, f_in, n):
       /* ... */
       def n_controlled_Z(circuit, controls, target):
       /* ... */
   ```

   To generalize the D matrix we need to implement a $C^{n-1}Z$ gate (realized by `n_controlled_Z` defined above). Also in this case for more than 2 controls we need to realize multiple controlled NOT gates. In fact to realize the gate we round the controlled NOT gate of all the controls passed as argument of the method `n_controlled_Z`, with two Hadamard gates. Considering how we dealt with the problem of multiply-controlled NOT gates we can see in figure 18 how the diagonal matrix D is implemented for the case of 4 control qubits. In the actual implementation also in this case we reset the state of the ancillary qubits as they will be reused at the next iteration of the amplitude amplification phase.

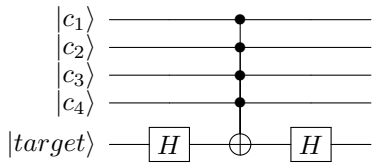

Figure 18: Theoretical $C^{n-1}Z$ gate

Now that we have the complete implementation of all the steps that compose the amplitude amplification in grover's algorithm we have to choose how many time to iterate them in order to solve the problem. As it is done in [11, G.Nannicini, 2020] we decided to do the same by iterating twice the bit-flip and the inversion about the average steps. As we can see in the actual implementation, once the problem has been encoded in the quantum circuit we run the `grover` method in order to solve the instance. The definition of Grover's algorithm with two iterations is the following:

```
def grover(self):
    self.__input_state()
    self.__black_box_u_f()
    self.__inversion_about_average()
    self.__black_box_u_f()
    self.__inversion_about_average()
```

3.3.4 Significant Results

3.4 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$\cdots$

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, notX_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 notX_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 notX_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] Hoyer P. Boyer M., Brassard G. and Tapp A. Tight bounds on quantum searching. 1996. https://arxiv.org/abs/quant-ph/9605034.

[3] Stephen A. Cook. The complexity of theorem-proving procedures. In *IN STOC*, pages 151–158. ACM, 1971.

[4] Lov K. Grover. A fast quantum mechanical algorithm for database search. In *ANNUAL ACM SYMPOSIUM ON THEORY OF COMPUTING*, pages 212–219. ACM, 1996.

[5] T. Hertli. 3-sat faster and simpler - unique-sat bounds for ppsz hold in general. In *2011 IEEE 52nd Annual Symposium on Foundations of Computer Science*, pages 277–284, 2011.

[6] Timon Hertli, Robin A. Moser, and Dominik Scheder. Improving PPSZ for 3-sat using crtitical variables. *CoRR*, abs/1009.4830, 2010.

[7] Takai T. Tamaki S. Iwama K., Seto K. Improved randomized algorithms for 3-sat. 2010. In: Cheong O., Chwa KY., Park K. (eds) Algorithms and Computation. ISAAC 2010. Lecture Notes in Computer Science, vol 6506. Springer, Berlin, Heidelberg.

[8] Konstantin Kutzkov and Dominik Scheder. Using CSP to improve deterministic 3-sat. *CoRR*, abs/1007.1166, 2010.

[9] Ohya M. and Masuda N. Np problem in quantum algorithm. 2000. Open Systems and Information Dynamics 7, 33–39. https://doi.org/10.1023/A:1009651417615.

[10] Tamaki S. Makino K. and Yamamoto M. Derandomizing the hssw algorithm for 3-sat. 2013. Algorithmica 67, 112–124. https://doi.org/10.1007/s00453-012-9741-4.

[11] Giacomo Nannicini. An introduction to quantum computing, without the physics. 08 2017.

[12] Feynman R.P. Quantum mechanical computers. 1986. Found Phys 16, 507–531. https://doi.org/10.1007/BF01886518.