Report on the Minimum Spanning Tree problem in Quantum Computing

Colombo Valerio, Deda Valentina, Fontana Tommaso

July 13, 2019

Abstract: After introducing the basics of Quantum Computing, this document will describe how to build a Minimum-Spanning-Tree algorithm which can be implemented on Quantum Computers and finally its feasibility will be analyzed. The proposed implementation has time complexity $O(|V|^3)$, which is worse than the one achieved by classical algorithms, $O(|V|^2 log|V|)$. With future optimizations and architectural changes with respect to current Quantum Computers, this algorithm could reach the theoretical lower-bound of $O(|V|)$.

Contents

List of source codes

MST Algorithms

1.1 The problem: Minimum Spanning Tree

1.1.1 Introduction to the problem

Given an undirected, weighted graph, the Minimum Spanning Tree (MST) is a subset of its edges that connects all the vertices, without any cycles and with the minimum total edge weight.

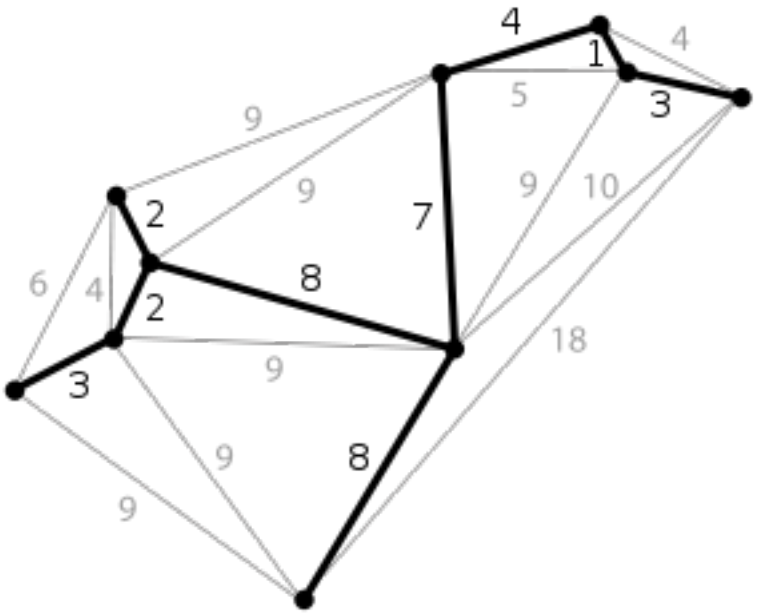


Figure 1.1: A Minimum Spanning Tree Example

In classical computing, three algorithms exist in order to compute the Minimum Spanning Tree of a graph:

- Borůvka's Algorithm $O(|E|log|V|)$
- Prim's Algorithm $O(|E| + |V|log|V|)$ or $O(|V|^2)$
- Kruskal's algorithm $O(|E|log|V|)$

1.2 Algorithms

1.2.1 Borůvka's Algorithm

Borůvka's Algorithm was first published in 1926 by Otakar Borůvka [1]. It proceeds as follows:

1. Create a set of vertices F, called a Forest. Is is initially empty.

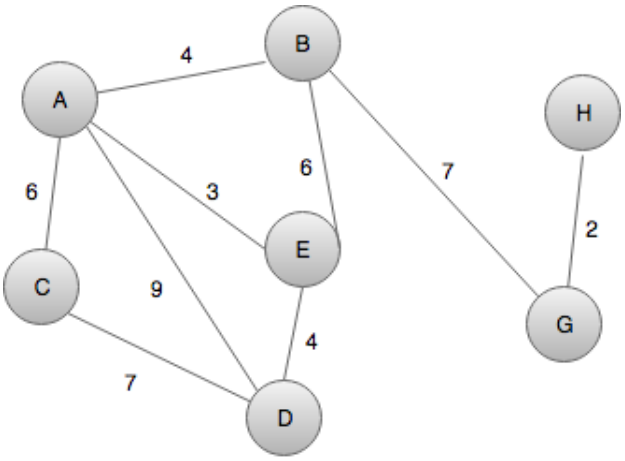


Figure 1.2: Step 1

2. Add all vertices to F. For each vertex, select the incident edge having minimum weight;
3. Group vertices in F which are connected by minimum weight arcs. A set of connected vertices forms a Component.
4. For each Component, select the incident edge having minimum weight;

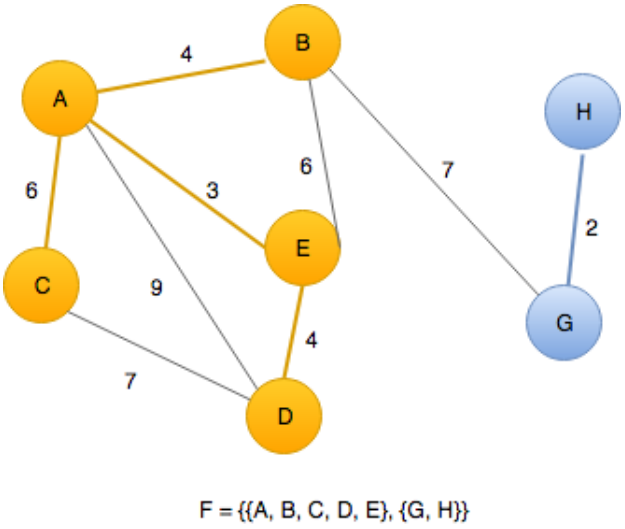


Figure 1.3: Step 2

5. Repeat step 2 until all vertices belong to the same component. The edges that have been selected at this point form the Minimum Spanning Tree.

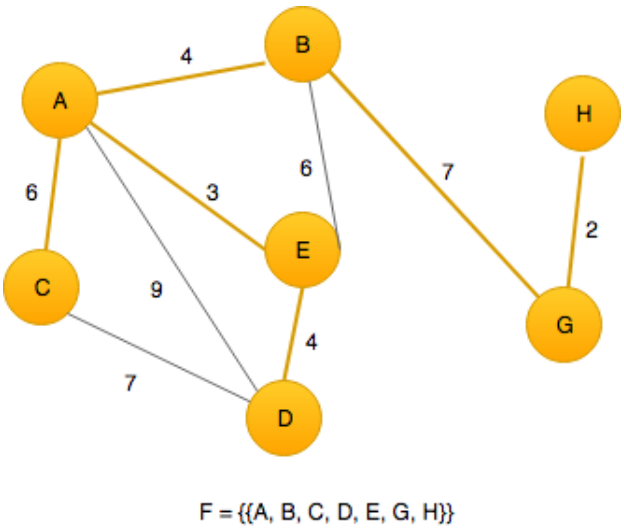


Figure 1.4: Step 3

Algorithm 1 Borůvka's Algorithm

Input: A connected graph $G = (V, E)$ with a positive cost function on the edges
Output: Minimum spanning tree $T = (S, F)$ of G

```
1:  procedure Borůvka
2:      F = {}
3:      for v ∈ Vertices do
4:          F.add(v)
5:      while F.size ≥ 1 do
6:          for C ∈ F do
7:              List= [ ]
8:              for v ∈ C do
9:                  for E ∈ incident-edges(C) do
10:                     for u ∉ C do
11:                         if ∃ edge(v, U) ∧ E.weight < curr-min then
12:                             curr-min = E.weight
13:                             minimum-weight(C).add(curr-min)
14:                 F.add(min(E ∈ List))
```

1.2.2 Prim's Algorithm

The algorithm was developed in 1930 by Czech mathematician Vojtěch Jarník[2]. It proceeds as follows:

1. Initialize a set S with an arbitrary vertex v ∈ V. S will be the set of vertices belonging to the Tree.

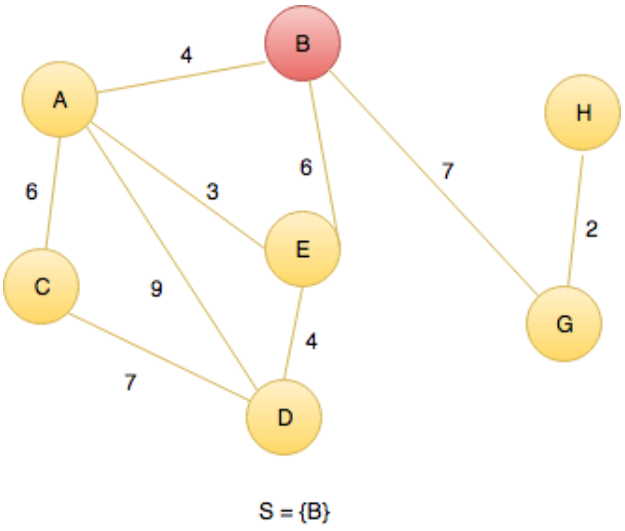


Figure 1.5: Step 1

2. Grow the tree by one edge: among the edges that connect the tree to vertices not yet in the tree, find the minimum-weight edge, and transfer both the edge and the vertex to the tree.

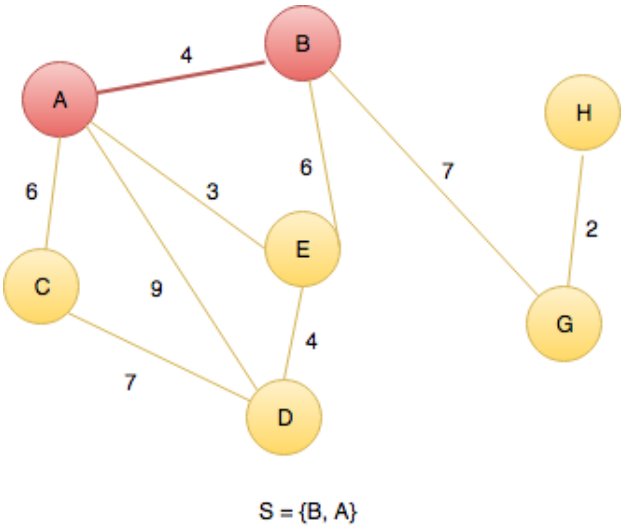


Figure 1.6: Step 2

3. Repeat step 2 until all nodes are in the tree.

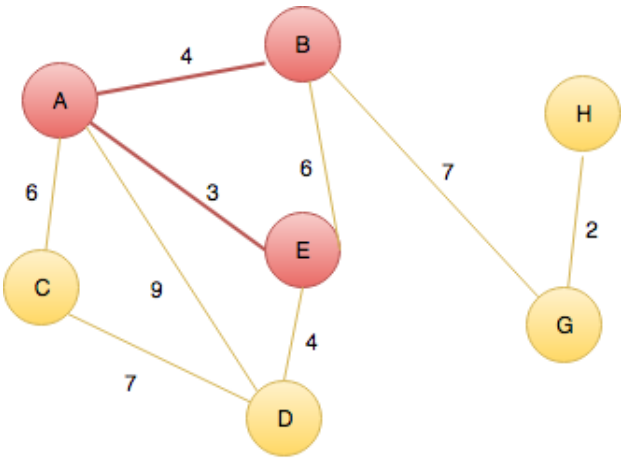


Figure 1.7: Step 3.0

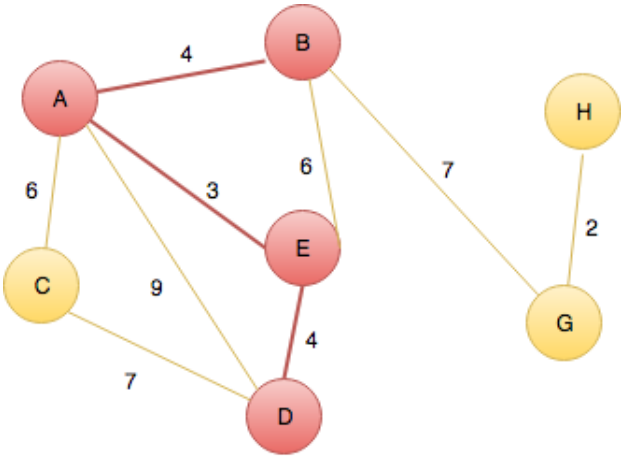


Figure 1.8: Step 3,1

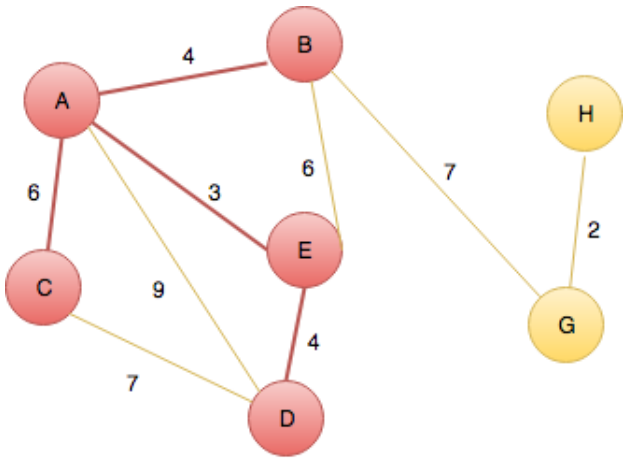


Figure 1.9: Step 3.2

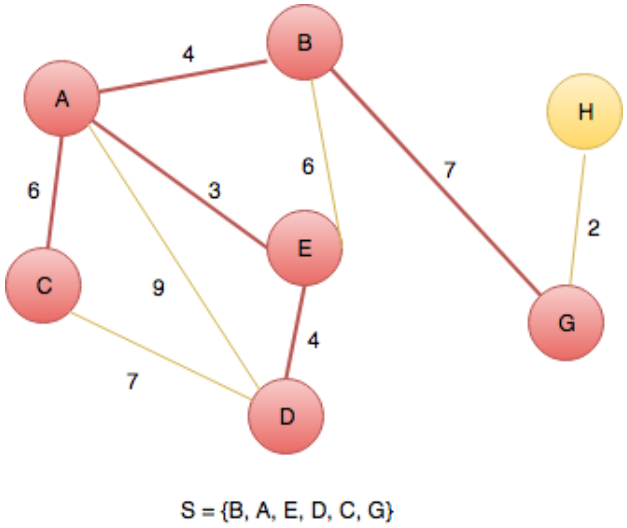


Figure 1.10: Step 3.3

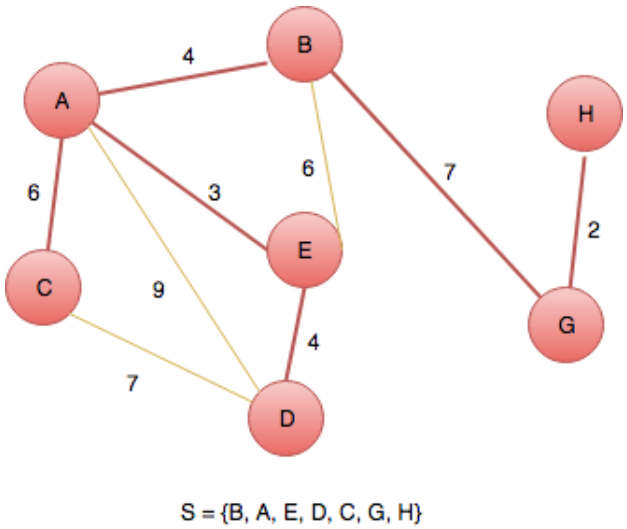


Figure 1.11: Step 3.4

Algorithm 2 Prim's Algorithm

Input: A connected graph $G = (V, E)$ with a positive cost function on the edges
Output: Minimum spanning tree $T = (S, F)$ of G

1: **procedure** PRIM
2: $F = \{\}$
3: $S = \{v\}$ ▷ v is an arbitrary vertex
4: **while** $S \neq F$ **do**
5: $e = (v, u) \mid v \in F \wedge u \notin F \wedge$ e.weight $=$ min
6: $F := F \cup (u, v)$
7: $S := S \cup (\{u, v\} \setminus S)$

1.2.3 Kruskal's algorithm

The algorithm was written by Joseph Kruskal in 1956 [3]. It proceeds as follows:

1. Create a forest F, where each vertex in the graph is a separate tree; Create a set S containing all the edges in the graph;

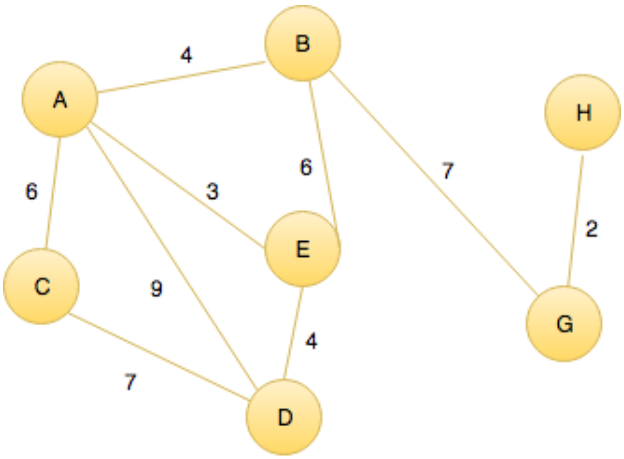


Figure 1.12: Step 1

2. Remove an edge with minimum weight from S. If the removed edge connects two different trees, then add it to the forest F, combining two trees into a single tree;

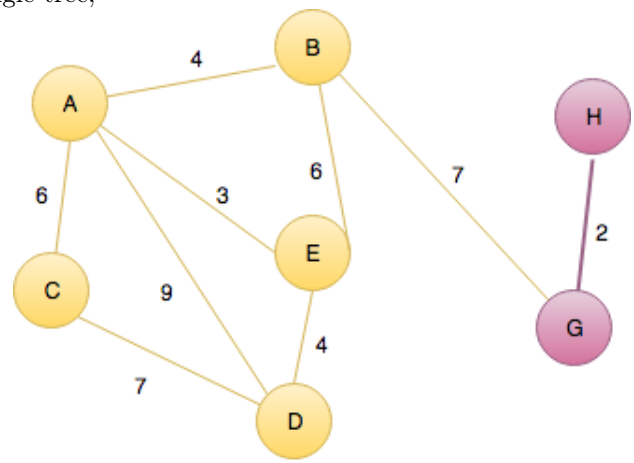


Figure 1.13: Step 2

3. Repeat step 2 until S is empty or F is spanning.

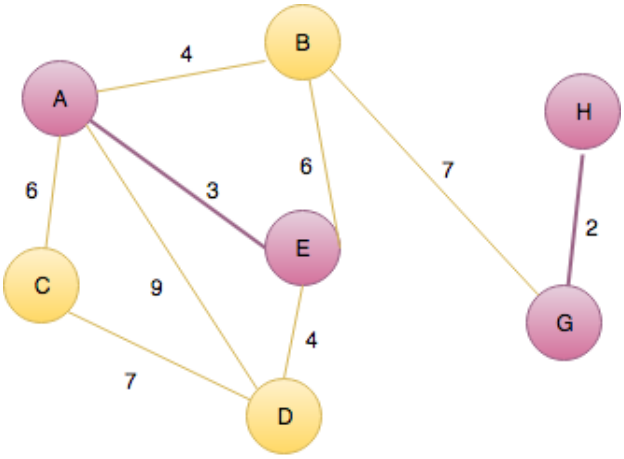


Figure 1.14: Step 3.0

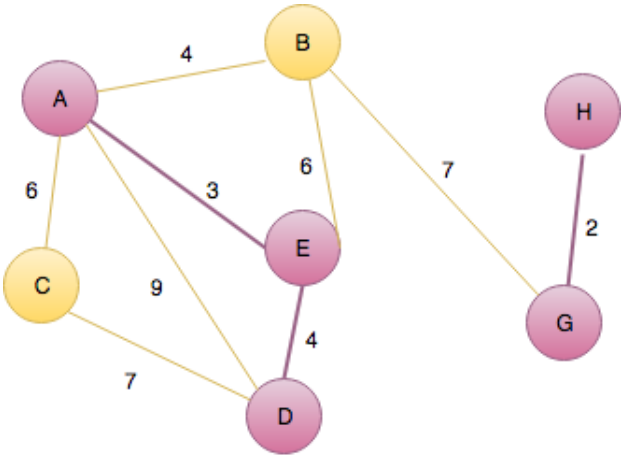


Figure 1.15: Step 3.1

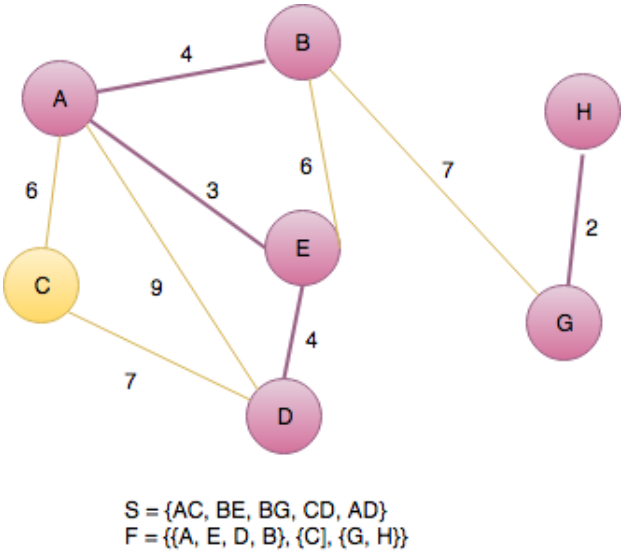


Figure 1.16: Step 3.2

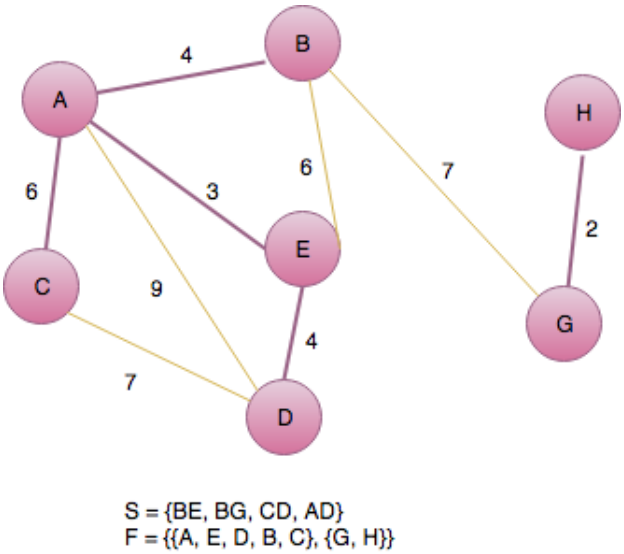


Figure 1.17: Step 3.3

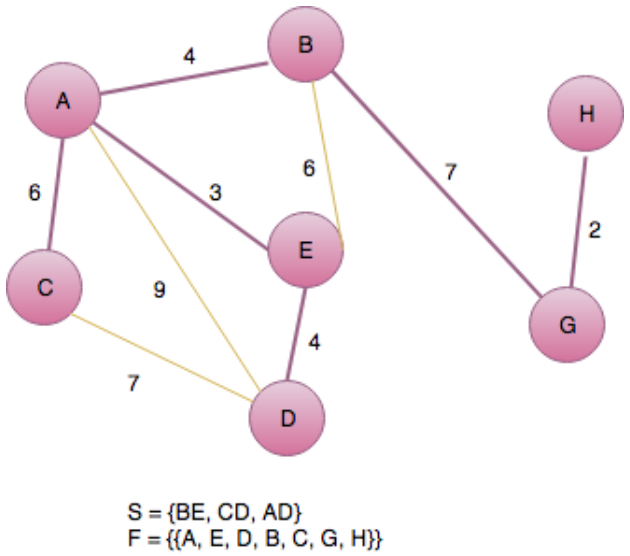


Figure 1.18: Step 3.4

Algorithm 3 Kruskal's Algorithm

Input: A connected graph $G = (V, E)$ with a positive cost function on the edges
Output: Minimum spanning tree $T = (S, F)$ of G

1: **procedure** KRUSKAL
2: $F = \{\}$
3: Sort edges in non-decreasing order by cost.
4: **for** edge e_i, i= 1, 2, ...$|E|$ **do**
5: **if** F $\cup$ e_i has no cycle **then**
6: $F := F \cup e_i$
7: **if** $|F| = |V| - 1$ **then**
8: return F and **Stop**;

Introduction to Quantum Computing

2.1 Quantum world basis

In this chapter, we will build the foundations of our quantum world knowledge. We will start by understanding some basic and very important quantum phenomena which will be a must-know for the following chapters. Then, we will present the notation of quantum computing, its basic concepts and its current state of the art.

2.1.1 Quantum phenomena

In this section, we will discuss the most important quantum mechanics phenomena for the quantum computing world. Before we start, we have to clarify the concept of *quantum state*

Definition 0.1
Quantum state *In quantum physics, quantum state refers to the state of an isolated quantum system. A quantum state provides a probability distribution for the value of each observable item, i.e. for the outcome of each possible measurement on the system.*

Definition 0.2
Superposition *Quantum superposition means that, much like waves in classical physics, any two (or more) quantum states can be added together, superposed, and the result will be another valid quantum state; and conversely, that every quantum state can be represented as a sum of two or more other distinct states.*

Definition 0.3
Entanglement *Quantum entanglement states that pairs or groups of particles could be generated, interact in ways such that quantum state of a single particle can't be described without describing the state of the whole quantum system.*

2.1.2 Qubit

A quantum bit, or qubit for short, is the quantum computing counterpart to the binary digit or bit in classical computing. Just as a bit is the basic unit of information in a classical computer, a qubit is the basic unit of information in a quantum computer. In a classical system, a bit would have to be in one state or in the other. However, quantum mechanics allows the qubit to be in a coherent superposition of both states/levels simultaneously, a property which is fundamental to quantum mechanics and quantum computing. This property gives quantum computing a new tool to unlock a new class of algorithms which are more efficient.

Representation

A qubit can be represented with various notations. Each notation has its own advantages in different scenarios. The main representations are the following.

Standard representation A qubit can be interpreted as a vector whose component are a linear superposition of its two orthonormal basis states. The basis, in our case, are the possible values of the qubit after its measurement: 0 or 1. This quantum states are usually represented with the Dirac/bra-ket notation. The notation uses angle brackets and a vertical bar to denote the scalar product of vectors or the action of a linear function on a vector in a complex vector space. So, for example, the scalar product between v_i and v_j can be expressed as $\langle v_i|v_j\rangle$; the left part $\langle|$ is called *bra* and the right part $|\rangle$ is called *ket*. A ket is a representation of a single vector, so our basis can be represented as

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

This notation is very useful to describe qubits in superposition state. We have said that a qubit can be represented as a linear combination of its basis, thus we can write

$$|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle \quad \alpha, \beta \in \mathbb{C} \tag{2.1}$$

where α and β are probability amplitudes. For Born's rule, the probability of a given qubit to have value $|0\rangle$ is equal to $|\alpha|^2$ and to have value equal to $|1\rangle$ is $|\beta|^2$. Because the absolute squares of the amplitudes equate to probabilities, it follows that α an β must be constrained by the equation

$$|\alpha|^2 + |\beta|^2 = 1 \tag{2.2}$$

This notation can also describe states of multiple qubits. The quantum state of multiple qubits can be represented as tensor product between them. Therefore, for instance for two qubits we have

$$|00\rangle = \begin{bmatrix}1\\0\\0\\0\end{bmatrix}, |01\rangle = \begin{bmatrix}0\\1\\0\\0\end{bmatrix}, |10\rangle = \begin{pmatrix}0\\1\end{pmatrix} \otimes \begin{pmatrix}1\\0\end{pmatrix} = \begin{bmatrix}0\\0\\1\\0\end{bmatrix}, |11\rangle = \begin{bmatrix}0\\0\\0\\1\end{bmatrix}$$

An important clarification has to be done: a qubit cannot have a value between "0" or "1", in fact, when measured, its value can only be $|0\rangle$ or $|1\rangle$. The superposition is given by the intrinsic uncertainty of the qubit and the impossibility to know its value until measuring/collapsing it.

Bloch sphere Another frequently used notation is the Bloch sphere. This notation is based on the fact that (1) has more degrees of freedom than needed, as it has 4 degrees, so we can eliminate some of them. We start from (1)

$$\begin{aligned}
|\psi\rangle &= \alpha|0\rangle + \beta|1\rangle && \textit{standard notation}\\
&= r_0 e^{i\phi_0}|0\rangle + r_1 e^{i\phi_1}|1\rangle && \textit{expand complex number}\\
&= e^{i\phi_0}[r_0|0\rangle + r_1 e^{i(\phi_1-\phi_0))}|1\rangle] && \textit{factoring } e^{i\theta_0}\\
&= r_0|0\rangle + r_1 e^{i(\phi_1-\phi_0))}|1\rangle && e^{i\phi_0} \textit{ is irrelevant}\\
&= r_0|0\rangle + r_1 e^{i\phi}|1\rangle && \textit{rename } \phi_1-\phi_0 \textit{ with } \phi\\
&= \cos\frac{\theta}{2}|0\rangle + \sin\frac{\theta}{2}e^{i\phi}|1\rangle && \textit{apply (2) and substitute } r_0 = \cos\frac{\theta}{2},\quad r_1 = \sin\frac{\theta}{2}
\end{aligned}$$

The last equation has only two variables, θ and ϕ. We can interpret $|\psi\rangle$ as a vector in a three-dimensional space described with polar coordinates. We only need two angles because we know that from (2) $|\psi\rangle$ is a unit vector. For further clarity, we will provide an example. If we have

$$\theta = 0, \quad \phi = 0$$
$$|\psi\rangle = 1|0\rangle + 0|1\rangle = |0\rangle$$

or another example

$$\theta = \frac{\pi}{2}, \quad \phi = 0$$
$$|\psi\rangle = \frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle = |+\rangle$$

Physical realization

In this section, we will provide a little introduction to the physical implementation of a qubit. As a classical bit needs one physical dimension to store information, like voltage on a transistor gate: the same is needed by a qubit. The difference lays in the physical dimension which must be used. Qubits are based on quantum properties like polarization, particle spin, charge or on the presence of a particle like a photon or an electron. For example, we may consider a qubit based on electron spin; The spin of an electron can be in two states, *up* or *down*. Our goal is to measure the electron spin and interpret it as a qubit. In practice, a phosphorous atom is embedded in the p-type part of a classical n-p-n transistor. Then, a large magnetic field is applied to the transistor. The spin of the outermost electron of the phosphorous atom will align with the magnetic field in the lowest energy configuration, so the spin will tend to the down-state. But an issue arises: at room temperature, the electron has enough energy to flip its spin back to higher energy configuration, so the spin would bounce between up and down-state. Therefore, the system must be cooled down to some hundredth of degrees above absolute zero. This way the electron won't have sufficient thermal energy to flip the spin from down to up. After understanding how to bring an electron in a spin down-state, we might discuss the possibility to change its spin at will.

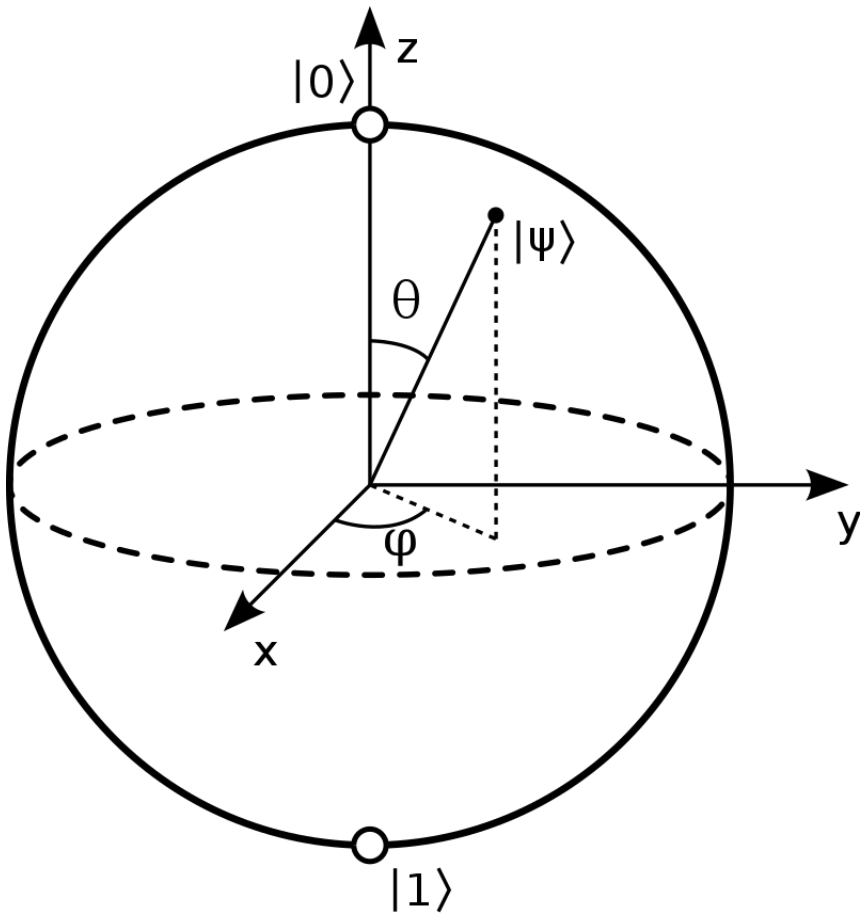


Figure 2.1: Bloch sphere. Basis states are antipodal. All intermediate states on the unitary sphere are a superposition of basis states

To achieve this, we can apply microwaves at a specific frequency, in particular at electron resonance frequency. A better option exists: if we apply the microwave for a shorter time than needed to transition from the down-state to the up-state, we can bring the electron in a superposition state with a specific phase between the two stable states. Now, we know how to set the value of a qubit; the last technique to understand consists in how to perform a measurement. As we have stated before, an electron in the up-state has more energy than when it is in the down-state. When it is more energetic, the phosphorous electron can jump into the silicon, leaving the bare phosphorous nucleus behind. If this happens, the phosphorous nucleus will have a positive charge, which will be reflected in the transistor gate voltage. In order to measure the qubit value it is sufficient to read the transistor gate voltage, determining the electron spin.

2.1.3 Quantum computation

Now that we have a basic understanding of some quantum phenomena and a representation of the minimum quantum information, a qubit, we can discuss how to manipulate them in order to perform some computations. In this document, we will consider a digital approach to quantum computing, which involves quantum logic gates to perform computation. The opposite approach to the digital one is the analogical approach, which will not be discussed in this document. The digital approach consist in manipulating a n-qubit register sequentially through quantum gates.

Logic gates

Gates can be considered as a quantum equivalent of classical logic gates, but they change the quantum state of the qubits by manipulating their probability distribution rather than apply logic functions to the input. Moreover, quantum logic gates have to follow some restrictions that emerge from the nature of qubits. All gates must be reversible: in other words, the gate function must be bijective, so that from the gate output it is possible to infer the input. This property arises from the probabilistic nature of qubits. We know that, for (2), the probability amplitudes must preserve their sum to 1.

Therefore, every gate must conserve a coherent superposition state. To better understand how a gate can achieve that we can consider Bloch sphere. As we have said before, we can interpret a qubit as a unit vector in a three dimensional space. Applying a gate is equivalent to rotating the qubit vector around the origin in the Bloch sphere. Therefore, a quantum logic gate must be a unitary operation in order to preserve the length of the vector. Reversibility is a property of unitary operations.

Quantum logic gates are represented by unitary matrices. The number of qubits in the input and output of the gate must be equal; a gate which acts on n qubits is represented by a 2^nx2^n unitary matrix. The most common quantum gates operate on spaces of one or two qubits, just like common classical logical gates. The quantum states that the gates act upon are vectors in 2^n complex dimensions. The action of the gate on a specific quantum state is found by multiplying the vector $|ab\rangle$, which represents the state, by the matrix U representing the gate.

$$U\,|ab\rangle \tag{2.3}$$

Now that we know which rules a quantum gate must follow, we can present the available gates.

Single qubit gates

- **Hadamard gate (H)** The Hadamard gate acts on a single qubit. Its purpose is to map the basis state $|0\rangle$ and $|1\rangle$ to a superposition state. The Hadamard gate is the combination of two rotations, π about the Z-axis and $\frac{\pi}{2}$ about Y-axis. So its operation matrix is

$$H = \frac{1}{\sqrt{2}}\begin{bmatrix}1 & 1\\ 1 & -1\end{bmatrix} \tag{2.4}$$

We can apply the H gate to the basis states.

$$H\,|0\rangle = \frac{1}{\sqrt{2}}\begin{bmatrix}1 & 1\\ 1 & -1\end{bmatrix}\begin{bmatrix}1\\0\end{bmatrix} = \frac{1}{\sqrt{2}}\begin{bmatrix}1\\1\end{bmatrix} = \frac{1}{\sqrt{2}}\,|0\rangle + \frac{1}{\sqrt{2}}\,|1\rangle \tag{2.5}$$

$$H\,|1\rangle = \frac{1}{\sqrt{2}}\begin{bmatrix}1 & 1\\ 1 & -1\end{bmatrix}\begin{bmatrix}0\\1\end{bmatrix} = \frac{1}{\sqrt{2}}\begin{bmatrix}1\\-1\end{bmatrix} = \frac{1}{\sqrt{2}}\,|0\rangle - \frac{1}{\sqrt{2}}\,|1\rangle \tag{2.6}$$

- **Pauli X gate (X)** The Pauli-X gate acts on a single qubit. It is the quantum equivalent of the NOT gate for classical computers (with respect to the standard basis states $|0\rangle$ and $|1\rangle$). It equates to a rotation around the X-axis of the Bloch sphere by π radians. Its matrix is

$$X = \begin{bmatrix}0 & 1\\ 1 & 0\end{bmatrix} \tag{2.7}$$

If we apply it on the basis states we have

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

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

- **Pauli Y gate (Y)** The Pauli-Y gate acts on a single qubit. It equates to a rotation around the Y-axis of the Bloch sphere by π radians. So its matrix is

$$Y = \begin{bmatrix}0 & -i\\ i & 0\end{bmatrix} \tag{2.8}$$

The effect of the gate is to map $|0\rangle$ to $i\,|1\rangle$ and $|1\rangle$ to $-i\,|0\rangle$. In fact

$$Y\,|0\rangle = \begin{bmatrix}0 & -i\\ i & 0\end{bmatrix}\begin{bmatrix}1\\0\end{bmatrix} = \begin{bmatrix}0\\i\end{bmatrix} = i\,|1\rangle$$

$$Y\,|1\rangle = \begin{bmatrix}0 & -i\\ i & 0\end{bmatrix}\begin{bmatrix}0\\1\end{bmatrix} = \begin{bmatrix}-i\\0\end{bmatrix} = -i\,|0\rangle$$

- **Pauli Z gate (Z)** The Pauli-Z gate acts on a single qubit. It equates to a rotation around the Z-axis of the Bloch sphere by π radians. It's equivalent to a phase shift with $\phi = \pi$. Its matrix is

$$Z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \tag{2.9}$$

$$|A\rangle \xrightarrow{Z} (-1)^A |A\rangle \tag{2.10}$$

Its effect is to leave $|0\rangle$ unchanged and to map $|1\rangle$ to $-|1\rangle$. In fact, we have

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

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

Controlled gate Controlled gates act on 2 or more qubits, where one or more qubits act as a controller for some operations. They are represented by matrices composed by the identity matrix on the top left corner plus an operation matrix $U = \begin{bmatrix} u_{11} & u_{12} \\ u_{21} & u_{22} \end{bmatrix}$ on bottom right corner

$$\begin{bmatrix} 1 & 0 & \dots & 0 & 0 \\ 0 & 1 & \dots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \dots & u_{11} & u_{12} \\ 0 & 0 & \dots & u_{21} & u_{22} \end{bmatrix} \tag{2.11}$$

Here there are some examples

- **Controlled not gate (CNOT)** The controlled NOT gate (or CNOT or cX) acts on 2 qubits, and performs the NOT operation on the second qubit only when the first qubit is $|1\rangle$, and otherwise leaves it unchanged. With respect to the basis $|00\rangle, |01\rangle, |10\rangle, |11\rangle$, it is represented by the matrix:

$$CNOT = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \tag{2.12}$$

$$|A\rangle |B\rangle \xrightarrow{CNOT[A,B]} |A\rangle |A \oplus B\rangle \tag{2.13}$$

So, for example

$$CNOT|10\rangle = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} = |11\rangle$$

- **Controlled phase shift gate (CZ)** The CZ gate acts on 2 qubits, and performs the Z operation on the second qubit only when the first qubit is $|1\rangle$, and otherwise leaves it unchanged. It is represented by the matrix:

$$CZ = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix} \tag{2.14}$$

So for example

$$CZ|11\rangle = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ -1 \end{bmatrix} = -|11\rangle$$

- **CCNOT** The CCNOT gate, also called Toffoli gate or Deutsch gate, is a 3-bit gate which performs the NOT operation on the third qubit only when the first and the second qubits are $|1\rangle$, and otherwise leaves it unchanged. As we have said before controlled, gates are represented by matrices composed by the identity matrix on the top left corner plus an operation matrix, in this case X. So we have

$$CCNOT = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix} \tag{2.15}$$

$$|A\rangle |B\rangle |C\rangle \xrightarrow{CCNOT[A,B,C]} |A\rangle |B\rangle |C \oplus (A \wedge B)\rangle \tag{2.16}$$

We can now realize an example:

$$CCNOT |101\rangle = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} = |101\rangle$$

The controlled gates can have an arbitrary number of qubits so, theoretically, we can construct controlled gates of any dimension. It's worth mentioning that CCNOT gate is a universal gate for classical computing, which means that every classical gate like, AND, OR, XOR, and so on, can be reconstructed from the CCNOT gate. This can be viewed as an evidence of the Turing completeness of quantum machines.

Special gate

- **Measurement** The measurement gate is responsible of the measurement of a qubit. Measurement is irreversible and therefore it is not a quantum gate, because it assigns the observed variable to a single value. Measurement takes a quantum state and projects it on one of the base vectors with a likelihood equal to the square of the amplitude along that base vector collapsing the state of the qubit to a definite state. For this reason, the measurement is usually the last operation.

Quantum speed up

At this point, we can discuss the benefits of quantum computation in comparison to classical computing.
As we have shown in the CCNOT paragraph, the quantum computation is at least Turing complete. So, we can say that every operation that can be carried out by a classical computer can also be performed by a quantum computer. The edges of quantum computers are their abilities to parallelize computation. Consider a function $f : \{0,1\}^n \to \{0,1\}^n$, computable using a circuit consisting of k classical gates. This circuit can be implemented with $\Theta(k)$ CCNOT gates. We can consider the circuit as a single gate named U. Now consider a quantum system composed by a n-qubit register, in an equally weighted superposition:

$$|\psi\rangle = \frac{1}{\sqrt{2^n}}(|0\rangle + |1\rangle + |2\rangle + ... + |2^n - 1\rangle) \tag{2.17}$$

Applying the gate U yelds the state

$$|\psi\rangle = \frac{1}{\sqrt{2^n}}(|f(0)\rangle + |f(1)\rangle + |f(2)\rangle + ... + |f(2^n - 1)\rangle) \tag{2.18}$$

The computation of the function f is done exponentially many times (in the number of bits), consuming an amount of time equal to the one necessary to compute one value.[4] This phenomenon is called quantum parallelism, and this is obviously a huge advantage of quantum computing, although it takes some non-trivial methods to truly exploit it[5]. This is due to the fact that simply measuring the outcome yields just one of the results, which is no better than just computing $f(i)$ for some randomly chosen $i \in \{0, ..., 2^n - 1\}$

Grover

3.1 Grover's algorithm

In this section, we will discuss about Grover's algorithm and its application in the MST problem. Grover's algorithm is defined as follows:

Definition 0.4
Grover's algorithm *Grover's algorithm is a quantum algorithm that finds with high probability the input to a black box function that produces a particular output value[6][7].*

$$f : \{0, 1, ..., N-1\} \to \{0, 1\}, \quad where \quad N = 2^n,\ n \in \mathbb{N}$$

$$f(x) = \begin{cases} 0 & if\ x \neq w \\ 1 & if\ x = w \end{cases}$$

3.1.1 Algorithm characteristics

The applications of the algorithm can be many, but usually, as it was presented in the original paper by Grover, the algorithm is described as "searching in a database". This definition is not precise, as the algorithm cannot be used for actual databases (see [8], *Hint*: constructing an oracle for an unstructured database has a linear complexity; this will be clear in a moment). As suggested by the definition, if we have $y = f(x)$, the algorithm can calculates efficiently x when given y. Inverting a function is related to the research in a database, as we could design a function that produces one particular value of y (1, for instance) if x matches a desired entry in a database, and another value of y (0) for other values of x. So Grover's algorithm can be seen as a research algorithm in an unstructured table. Let's analyse the definition step by step.

Like many quantum algorithms, Grover's algorithm is probabilistic, meaning that it gives the correct answer with a probability smaller than 1. Although there is technically no upper bound on the number of repetitions that might be needed before the correct answer is obtained, the expected number of repetitions is a constant factor which does not grow with N, number of elements. In particular, the number of iterations is a truly significant factor in the correctness of the result. This point will be discussed in depth in the following paragraph. Now that we have the definition of the algorithm, we can analyse its complexity. Classically, a search in an non-ordered table takes $O(N)$ steps, where N is the number of entries. Grover's algorithm takes only $O(\sqrt{N})$ steps to find the solution. This speed-up, even though is not exponential, gives a significant boost in performances to satisfiability problems, which belong to the NP-complete class of complexity.

3.1.2 Algorithm phases

The algorithm is composed by multiple steps:

1. Superposition setup
2. Phase inversion
3. Inversion about the mean

Let's analyse each step.

Superposition setup

The algorithm has multiple variations; the one we use in this document starts with a n qubit set to $|0\rangle$. The first step consist in putting all qubits in a superposition state through Hadamard gates.

Phase inversion

Our objective is to isolate the value we are searching for. In order to do so, we must flip the phase of the searched item. In formulas, we have:

$$f(x^*) = 1 \quad where \quad x^* = w$$
$$\sum_x \alpha_x |x\rangle \xrightarrow{phase\ inversion} \sum_{x \neq x^*} \alpha_x |x\rangle - \alpha_{x^*} |x^*\rangle$$

Now, the problem is how to recognize the right number and flip its phase. In order to do so, we must introduce the concept of Oracle.

Oracle In the classical world, an oracle is a black box which takes as an input an n-bit number x and outputs a function $f(x)$. (Oracles can be reversible, just like any other classical circuits; for instance, it could input x and y and output x and $x \oplus y$). In the quantum case, an oracle is a black box which takes n qubits as an input and performs a unitary transformation U on them. It could take as an input two quantum registers, and transform them this way:

$$U |x\rangle |y\rangle = |x\rangle |y \oplus f(x)\rangle$$

However, in the quantum world other possibilities exist: e.g., we could perform a phase shift based on the value of f(x)

$$U |x\rangle = (-1)^{f(x)} |x\rangle$$

In the quantum world, an oracle is used to encode data into the system, in order to perform actions on it. In fact, for Grover's algorithm, no external data structure exists, as data are encoded in the oracle function $f(x)$.

The first oracle presented in this document is called *boolean oracle*, while the second one is called *phase oracle*. Grover's algorithm requires the use of the phase oracle, as by construction it flips the phase of the desired value.

Inversion about the mean

At this stage our objective is to highlight the phase or amplitude of our value with respect to the others, making its amplitude higher. In order to do so, we can invert all the amplitudes about the overall mean amplitude. In fact, if the amplitude of a value is lower than the mean amplitude, the former will increase, therefore the latter will decrease. Thus, the amplitudes of all $x \neq x^*$ will decrease and the amplitude of x^* will increase. This also explains why we are inverting the phase of x^*. In fact, this way the distance between the mean and the phase of x^* increases, therefore the phase boosting in this last operation will have a greater effect.

$$u = \frac{\sum_{x=0}^{N-1} \alpha_x}{N} \qquad \textit{mean amplitude}$$
$$\alpha_x \to (2u - \alpha_x) = u + (u - \alpha_x) \qquad \textit{amplitudes mapping}$$
$$\sum_x \alpha_x |x\rangle \xrightarrow{inversion\ about\ u} \sum_x (2u - \alpha_x) |x\rangle \qquad \textit{new state after flipping}$$

3.1.3 Number of iteration

As we have seen, applying a phase inversion followed by an inversion about the mean will result in the amplitude of the searched value x^* increasing and in all other amplitudes decreasing. This two operations are called *diffusion step*. In order to obtain a better result, we have to iterate the diffusion step several times. To estimate the number of needed iterations, we can proceed as follows. Suppose that we want to take the x^* amplitude to at least $\widehat{\alpha_{x^*}} = \frac{1}{\sqrt{2}}$, which means 50% of probability to have this result. To do so, we can measure how much the amplitude of x^* increases at each iteration.

$$\#steps = \frac{\widehat{\alpha_{x^*}}}{improvement\ by\ step}$$

When x^* has amplitude equal to $\frac{1}{\sqrt{2}}$, the rest of the amplitude, $\frac{1}{\sqrt{2}}$, has to be distributed among the remaining N-1 values, so that each value will have an amplitude

$$\alpha_x > \frac{1}{\sqrt{2(N-1)}} > \frac{1}{\sqrt{2N}}$$

We can now assume that the mean amplitude is roughly equal to the amplitude of a generic $x \neq x^*$ when N is big enough. When we apply the diffusion step, the amplitude of x^* increases about twice as the value of the mean, which is roughly $\frac{1}{\sqrt{2N}}$. Therefore, the improvement at each step will be

$$improvement\ by\ step = \frac{2}{\sqrt{2N}} = \sqrt{\frac{2}{N}} \tag{3.1}$$

Thus, the number of steps will be around

$$\#steps = \frac{\widehat{\alpha_{x^*}}}{improvement\ by\ step} = \frac{\frac{1}{\sqrt{2}}}{\sqrt{\frac{2}{N}}} = \frac{\sqrt{N}}{2} \sim \sqrt{N} \tag{3.2}$$

This is the reason why the complexity of the algorithm is $O(\sqrt{N})$ In practice, the number of iterations is described by the following formula

$$\#\ iterations = \frac{\pi}{4}\sqrt{N} \tag{3.3}$$

This is proved to be the optimal method to get the highest probability with the minimum number of iterations; moreover, adding more iterations than suggested will lead to failure in the search rather than to a more precise result. This fact is particularly important when the matches in the non-ordered table are more than one.

Multiple matches

So far, we have supposed that there was a single match to be found, but in many applications the search could return multiple results. Grover's algorithm can be generalized to take into account this fact[9]. The number of required iterations changes to

$$\#\ iterations = \frac{\pi}{4}\sqrt{\frac{N}{M}} \quad where\ M\ is\ the\ number\ of\ matches \tag{3.4}$$

As we can see, the number of matches plays an important role in defining the number of iterations required. This could create some problems when the number of matches is unknown, as the number of iteration can't be estimated exactly. Actually, the case with multiple matches is harder to handle. Moreover, the probability of finding a result depends on the amount of matches. Consider a random guesser implemented in a classical way and using Grover's algorithm for one iteration. The classical guesser will have a probability of $\frac{N}{M}$ of finding a result with a single random guess. Now it's useful to compare a classical random guesser with a single iteration of Grover's algorithm.

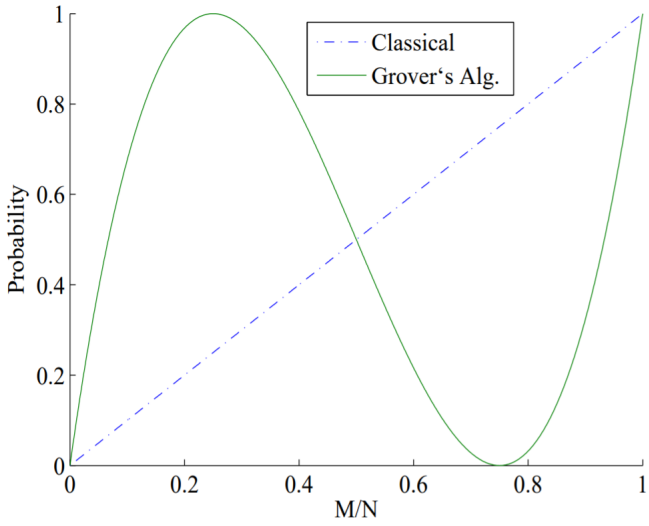


Figure 3.1: Probability of success $P^{Classical}$, P^{Grover} as a function of $\frac{M}{N}$

We can see from figure YYY that Grover's algorithm solves the case where $M = \frac{N}{4}$ with certainty. The probability of success of Grover's algorithm will be below one-half for $M > \frac{N}{2}$ and will fail with certainty for $M = \frac{3}{4}$. The probability of

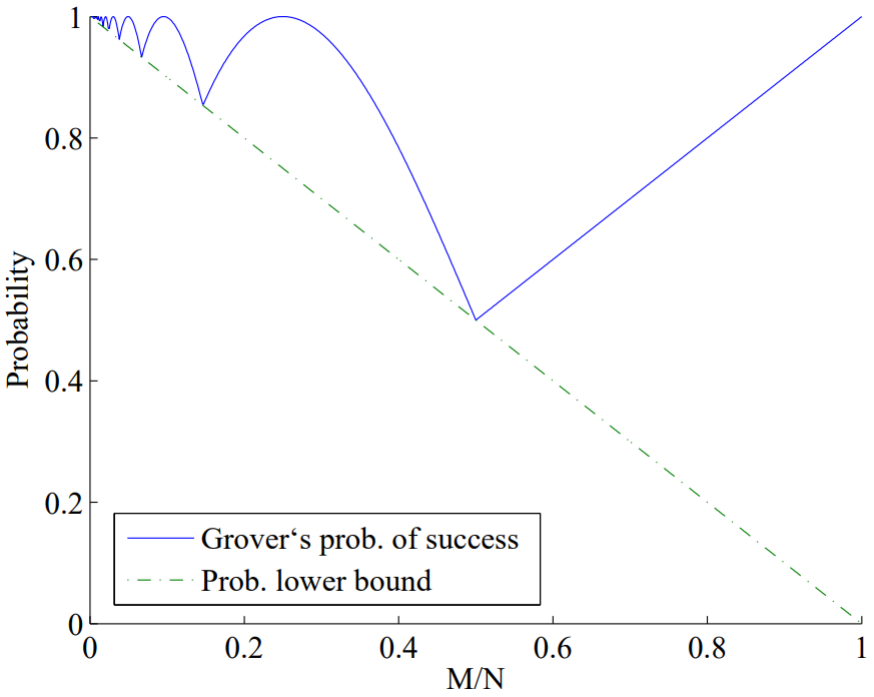


Figure 3.2: Probability of success $P^{Classical}$, P^{Grover} as a function of $\frac{M}{N}$

success of the classical guess technique is always over Grover's for $M > N/2$. Let's focus on how the quantum algorithm behaves when iterating following (4).

As we can see from figure YYY, the algorithm behaves better. The minimum probability of finding the correct result is 0.5 when $\frac{M}{N}$ is equal to 0.5. When $\frac{M}{N}$ is less than 0.25, the algorithm has a high probability of finding a result, but when $\frac{M}{N}$ is greater than 0.5, the algorithm behave as a classical random guesser.

$U|a\rangle = \begin{bmatrix} u_{11}u_{12} \\ u_{21}u_{22} \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \end{bmatrix}$

3.1.4 Phase shift problem

As we have seen in the previous paragraphs, the ability to invert the phase of a particular state of a register is an important skill in the computation of the Grover's algorithm, but the actual implementation of such operation

$$U|x\rangle = (-1)^{f(x)}|x\rangle$$

is not an easy task. Therefore, we have to find another way to implement this by using another combination of simpler operations. We can observe that the Z gate equates to a rotation around the Z-axis of the Bloch sphere by π radians. It's equivalent to a phase shift with $\Phi = \pi$. If we convert this result into an algebraic form we get:

$$|A\rangle \xrightarrow{Z} (-1)^A |A\rangle$$

We should construct a qubit so that by applying to it a Z gate we get an equivalent form of the phase shift. We can start by adding to the circuit a qubit set to $|0\rangle$, beside the qubit $|X\rangle$. We store the result of the oracle function into the qubit $|0\rangle$ and we get

$$|x\rangle|0\rangle \rightarrow |x\rangle|f(x)\rangle$$

then we apply the Z gate, obtaining

$$|x\rangle|f(x)\rangle \xrightarrow{Z} (-1)^{f(x)}|x\rangle|f(x)\rangle$$

By doing so, we have translated the problem from applying a phase shift to applying a boolean function $f(x)$ to the starting register. In the next chapter, we will discuss the generation of boolean function in the quantum world.

Boolean function → Quantum Circuit

4.1 Intro

The goal of this chapter is to provide a set of ready-to-use circuits and methods to automatically translate Boolean functions into quantum circuits.
It contains a number of trivialities, but currently there are not standards nor literature on how to achieve these goals.
To keep the operations as modular as possible, we introduce as a convention to only use **Clean** operations.

Definition 0.5 (Clean operation)
*An operation f is defined **Clean** if and only if it can be described as:*

$$\underbrace{|x\rangle}_{input}\ \underbrace{|0\rangle}_{ancillas}\ \underbrace{|0\rangle}_{result} \xrightarrow{f} |x\rangle\,|0\rangle\,|f(x)\rangle$$

This means that once the operation has been performed, the input should be brought back to the state in which it was before the computation. The ancilla qubits are by default set to 0 and the results is placed on its own reserved qubit.

The clean version of single-qubit gates can be created by using the controlled version of the gates.
The convention to have all ancilla qubits set to 0 helps with the creation of n-qbits ANDs and allows easier reuse of the same ancilla qubits between sub-modules.

Another property of Clean modules is that they allow to create a Clean super-module that combines several sub-modules. In order to reverse the computation to clean ancilla qubits, it's possible to re-apply the sub-modules in reverse order instead of the gates.

4.1.1 Note about Quantum circuits complexity

The "time" complexity is defined equivalently by the depth of the circuit or by the number of gates.[10][11][12]

Theorem 1
Let d being the depth of the circuit and g the number of gates.

$$O(d) = O(g)$$

Proof. Let q be the number of qbits in the circuit.
In the worst case all the gates are on the same qbits.

$$O(d_{\text{worst}}) = O(g)$$

In the best case all the gates are equally spread on the qbits.

$$O(d_{\text{best}}) = O(\frac{g}{q})$$

Therfore the actual complexity of a general circuit must be between the best and the worst cases:

$$O(d_{\text{worst}}) \geq O(d) \geq O(d_{\text{best}})$$

$$O(g) \geq O(d) \geq O(\frac{g}{q})$$

But q is a constant factor therefore $O(\frac{g}{q}) = O(g)$

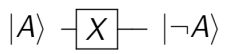
$$O(g) \geq O(d) \geq O(g) \qquad \Rightarrow \qquad O(d) = O(g)$$

□

4.2 Logical ports

4.2.1 NOT

Theorem 2
*The following circuit implements the **NOT** operation:*



Proof. The definition of the X gate is:

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

For the NOT operation it holds that:

$$\neg 1 = 0 \qquad \wedge \qquad \neg 0 = 1$$

Following the standard convention we obtain:

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

And we can verify that:

$$X|0\rangle = |1\rangle \quad \rightarrow \quad \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \qquad \wedge \qquad X|1\rangle = |0\rangle \quad \rightarrow \quad \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

□

4.2.2 CCNOT is a BUG

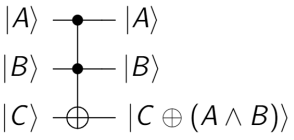
Definition 2.1 (BUG)
A Boolean / Classical Universal Gate (BUG from now on) is a gate that can be used to construct all others logical gates.

Theorem 3
The CCNOT gate is a BUG .

Proof. Since the NAND gate is a well known BUG we just have to show that:

$$\textbf{CCNOT} \Rightarrow \textbf{NAND}$$

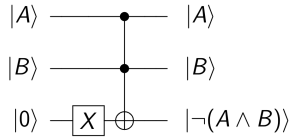
Since the CCNOT flips the target qubits if and only if the two control qubits are set to 1, it has the following logical function:



We can use a qubit set to 1 as C and obtain the NAND operation.

$$C \oplus (A \wedge B) \rightarrow 1 \oplus (A \wedge B) \underbrace{=}_{1 \oplus \alpha = \neg\alpha} \neg(A \wedge B)$$

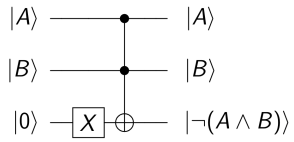
Thus, with the CCNOT Gate we can construct a NAND operation.



Since the NAND is a BUG, it implies that the CCNOT Gate is also a BUG. □

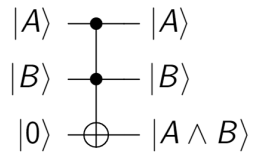
4.2.3 NAND

Corollary 3.1
*From the previous proof, it follows also that the circuit below implements a **NAND** operation:*



4.2.4 AND

Theorem 4
*The circuit below implements an **AND** operation:*



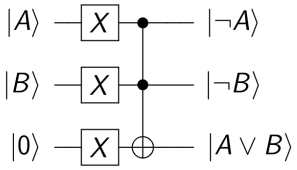
Proof. In a similar fashion to the previous proof, it holds that:

$$C \oplus (A \wedge B) \to 0 \oplus (A \wedge B) \underbrace{=}_{0 \oplus \alpha = \alpha} A \wedge B$$

□

4.2.5 OR

Theorem 5
*The circuit below implements a **OR**:*



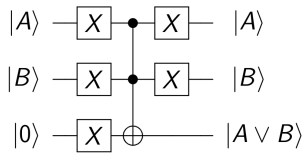
Proof. In a similar fashion to the previous proof, applying De Morgan's Theorems it holds that:

$$C \oplus (A \wedge B) \to 1 \oplus (\neg A \wedge \neg B) \underbrace{=}_{1 \oplus \alpha = \neg \alpha} \neg(\neg A \wedge \neg B) = A \vee B$$

□

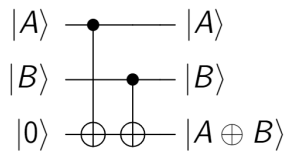
Now, the inputs $|A\rangle, |B\rangle$ are in negated form. We would like to have only "clean" operations which do not "ruin" the input qubits or ancilla qubits.

Theorem 6
*The circuit below implements a **Clean OR**:*

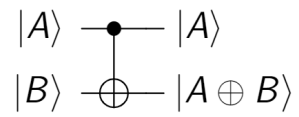


4.2.6 XOR

Theorem 7
*The circuit below implements a **XOR**:*



Proof. Since the CNOT flips the target bit if the control bit is $|1\rangle$, it implements the logical function:



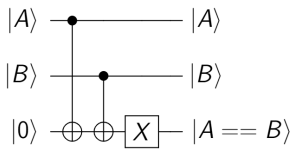
And since the XOR has Identity element 0 ($\alpha \oplus 0 = \alpha$) and it's associative $(\alpha \oplus (\beta \oplus \gamma) = (\alpha \oplus \beta) \oplus \gamma)$ it follows that:

$$0 \oplus A \oplus B = 0 \oplus (A \oplus B) = A \oplus B$$

□

4.2.7 EQ

Theorem 8
*The circuit below implements an **Equivalence**:*



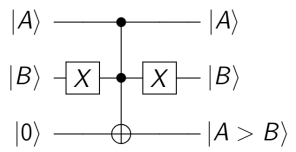
Proof. It's easy to see that the circuit implements the function $\neg(A \oplus B)$, since it's composed by a XOR circuit with a NOT gate on the result flag.

A	B	$A \oplus B$	$\neg(A \oplus B)$	A == B
0	0	0	1	1
0	1	1	0	0
1	0	1	0	0
1	1	0	1	1

□

4.2.8 GR

Theorem 9
The circuit below implements ***A Greater than B****:*



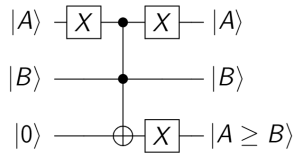
Proof. It's easy to see that the circuit implements the function $A \wedge (\neg B)$, since its is composed by an AND circuit with a NOT gates added on the B qubit.

A	B	$\neg B$	$A \wedge (\neg B)$	A > B
0	0	1	0	0
0	1	0	0	0
1	0	1	1	1
1	1	0	0	0

□

4.2.9 GREQ

Theorem 10
The circuit below implements a ***A Greater than or equal to B****:*

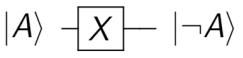
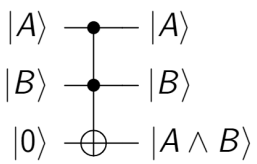
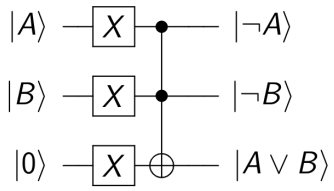
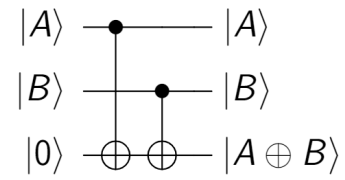
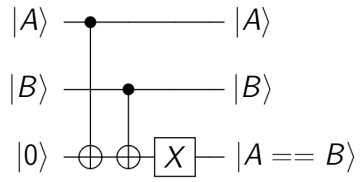
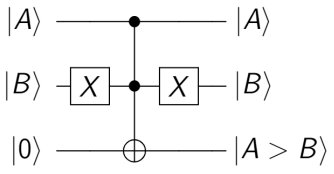
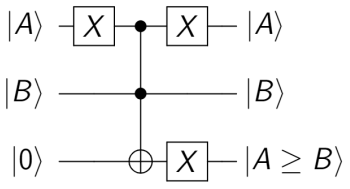


Proof. It's easy to see that the circuit implements the function $A \vee (\neg B)$, since since its is composed by an OR circuit without the NOT gates on the B qubit.

A	B	$\neg B$	$A \vee (\neg B)$	$A \geq B$
0	0	1	1	1
0	1	0	0	0
1	0	1	1	1
1	1	0	1	1

□

4.3 Cheatsheet

Name	Formula	Truth Table	Circuit
NOT	$\neg A$	A | $\neg A$ 0 | 1 1 | 0	
AND	$A \wedge B$	A B | $A \wedge B$ 0 0 | 0 0 1 | 0 1 0 | 0 1 1 | 1	
OR	$A \vee B$	A B | $A \vee B$ 0 0 | 0 0 1 | 1 1 0 | 1 1 1 | 1	
XOR	$A \oplus B$	A B | $A \oplus B$ 0 0 | 0 0 1 | 1 1 0 | 1 1 1 | 0	
A == B	$\neg(A \oplus B)$	A B | $A == B$ 0 0 | 1 0 1 | 0 1 0 | 0 1 1 | 1	
A > B	$A \wedge (\neg B)$	A B | $A > B$ 0 0 | 0 0 1 | 0 1 0 | 1 1 1 | 0	
A ≥ B	$A \vee (\neg B)$	A B | $A \geq B$ 0 0 | 1 0 1 | 0 1 0 | 1 1 1 | 1	

4.4 The nqbits gates problem

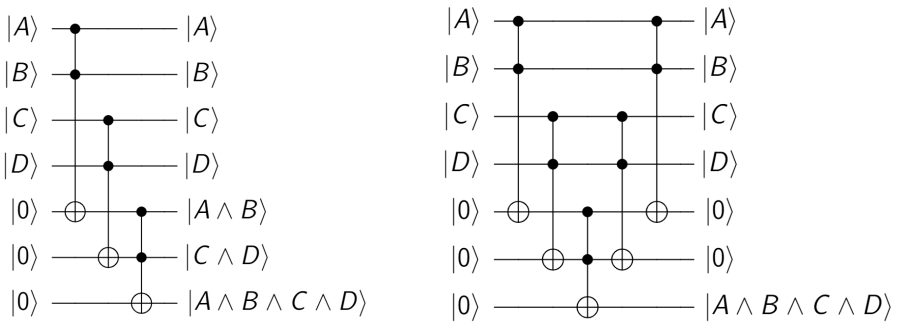
In real world quantum computers, currently, it's not possible to use gates with more than two controlled qubits. Therefore, it's not trivial to perform a "simple" operation such as:

$$A \wedge B \wedge C \wedge D$$

There are equivalent solutions that exploit AND associativity[13].

4.4.1 Tree Solution

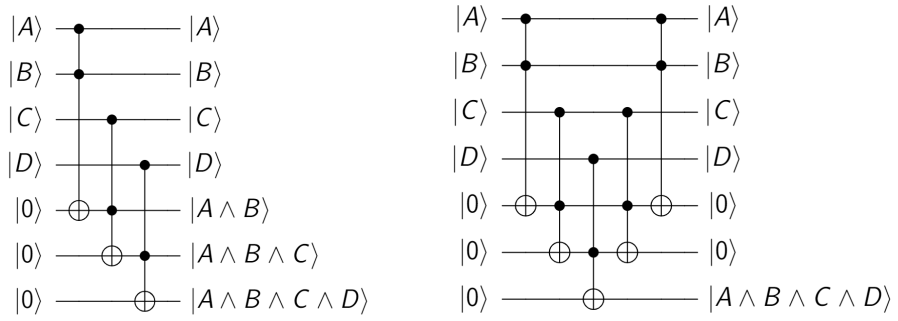
$$A \wedge B \wedge C \wedge D = (A \wedge B) \wedge (C \wedge D)$$



4.4.2 Ladder Solution

This is the technique currently used in **Qiskit-aqua**.

$$A \wedge B \wedge C \wedge D = ((A \wedge B) \wedge C) \wedge D$$



4.4.3 The problem with these solutions

Both the Tree and the Ladder solutions need 2 ancilla qubits. In general to perform an AND on k qubits, the ancilla qubits must be $(k-2) \sim O(k)$.
This implies that in total we will have $2(k-1)$ qubits to simulate.
Since the number of states is $N = 2^n$ we have:

$$2^k \to 2^{2(k-1)} = \left(2^{k-1}\right)^2$$

$$O(2^k) \to O\left(\left(2^k\right)^2\right)$$

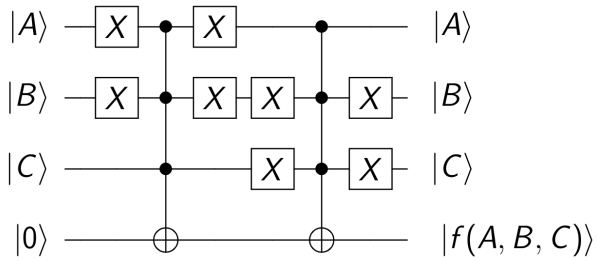
so there's a quadratic slow-down of the simulation.

4.5 The "multiplexer"

Sometimes, it's easier to express a boolean function by using only the combination of inputs that satisfies it.
Example:

$$f(A,B,C) = \begin{cases} 1 & (A,B,C) \in \{(0,0,1),(1,0,0)\} \\ 0 & \text{else} \end{cases}$$

f can be implemented just by "stacking" C^nNOTs with the X gates where the qubit must be set to 0.



Since we assume that the triplets are distinct, the XOR act as an OR:

$$\bigoplus_i (a_{i,0} \wedge a_{i,1} \wedge ...) \rightarrow \bigvee_i (a_{i,0} \wedge a_{i,1} \wedge ...)$$

It's the analogous Disjunctive Normal Form (DNF) of the Boolean function.
So the example translates to:

$$(\neg A \wedge \neg B \wedge C) \vee (A \wedge \neg B \wedge \neg C)$$

In case of a function that has more 1s than 0s, we can just add a gate X to the result qubit and then encode the 0s of the function instead of the 1s.
This implies that given the set of possible input values N with the multiplexer, in the worst case we have to encode $\frac{N}{2}$ values.

4.5.1 Tradeoffs

Being able to build a function from its DNF has the main advantage of being a mechanical procedure and it "skips" the optimization phase.
The main downside is that since we need a C^nNOT for each 1 in the function, this requires a $O(b)$ complexity, where b is the number of 1s in of the function.
This has a linear complexity, while the optimized circuit would require a complexity of $O(ln(b))$.
The issue in performing such an optimizaion consists in it being an NP-HARD problem.
E.g. the Quine Mc Cluskey's method has complexity $O\left(\frac{3^n}{n}\right)$

4.6 Natural numbers representation

The most natural way to encode a natural number is to map each number to a state, which is its binary expansion:

$$0 \to |000\rangle \qquad 1 \to |001\rangle \qquad 2 \to |010\rangle \qquad 3 \to |011\rangle \qquad 4 \to |100\rangle \qquad 5 \to |101\rangle \qquad ...$$

As a convention, the Least Significant Bit is the Upper bit.

This can be generalized to encode a set of natural numbers:

$$\{0,1,2,3\} \to \alpha\,|000\rangle + \beta\,|001\rangle + \gamma\,|010\rangle + \delta\,|011\rangle$$

$$\{0,3\} \to \alpha\,|000\rangle + \beta\,|011\rangle$$

The biggest limitation in this method consists in the difficulty to encode a general set, which is not a trivial task.
There is a type of sets which are easier to encode: sets having cardinality which is a power of two.
These sets can be encoded using just H, X, C^nNOT gates.

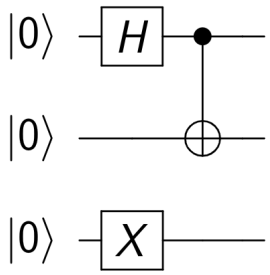


Figure 4.1: Circuit to get the state $\{4,7\} \to \frac{1}{\sqrt{2}}(|100\rangle + |111\rangle)$

The only set which is trivial to create is the universal set (the set with all the natural numbers encodable in the register).
It can be achieved by applying the H gate on all the qubits of the register.

In this framework, we can reinterpret the CCNOT operation:

$$CCNOT(|x\rangle, |0\rangle) = \begin{cases} |x\rangle\,|1\rangle & \text{if } |x\rangle = |11\rangle \\ |x\rangle\,|0\rangle & \text{else} \end{cases}$$

Therefore, the CCNOT, and in general the C^nNOT, allows us to raise a flag only on the state having all ones.
To be able to do it on a general number we just have to add the X gates , before and after the C^nNOT, on the qubits that are to 0 in the binary expansion of the number.
E.G. the value 4 encodes to $|100\rangle$, so it will have an X gates before and after the C^nNOT on the first and second qubit:

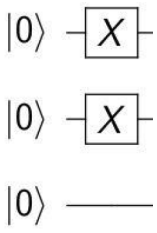


Figure 4.2: The gates to select the number 4 with the C^nNOT

It's possible to create an approximate state of a general set.
The idea is to use the "multiplexer" method to raise a flag on the states that belong to the set, then apply a Z gate.
This can be used as an Oracle. Then, by applying Grover's algorithm we obtain the approximated state.

4.7 Numbers comparisons

In order to be able to compare two k qubit numbers, represented as explained before, we can borrow a comparator circuit from classic literature.

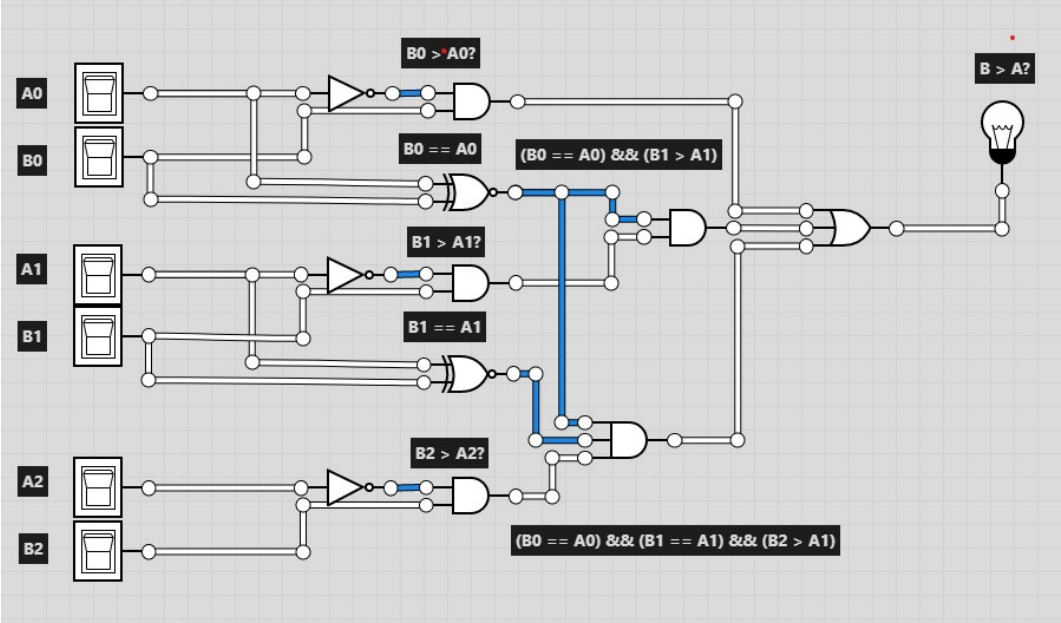


Figure 4.3: Classical 3-bits comparator

The function in general can be expressed as:

$$A \geq B = \bigvee_{i=0}^{n} \left[(b_i \geq a_i) \wedge \bigwedge_{j=0}^{i} b_i = a_i \right]$$

Since in quantum world we can implement at most 2 input gates, we have to "map" operations:

$$\begin{aligned}
A_0 &= (X_0 == J_0) = \neg(X_0 \oplus J_0) \\
A_1 &= (X_0 > J_0) = X_0 \wedge (\neg J_0) \\
A_2 &= (X_1 == J_1) \\
A_3 &= (X_1 > J_1) \\
A_4 &= (X_2 \geq J_2) = X_2 \vee (\neg J_2) = \neg((\neg X_2) \wedge J_2) \\
A_5 &= A_0 \wedge A_3 \\
A_6 &= A_0 \wedge A_2 \\
A_7 &= A_6 \wedge A_4 \\
A_8 &= A_1 \vee A_5 = \neg((\neg A_1) \wedge (\neg A_5)) \\
A_9 &= A_8 \vee A_7 = X \geq J
\end{aligned}$$

Figure 4.4: Operations and qbits mapping

Then, using the Logic-Gates Cheatsheet, the mapping can be translated to a circuit mechanically:

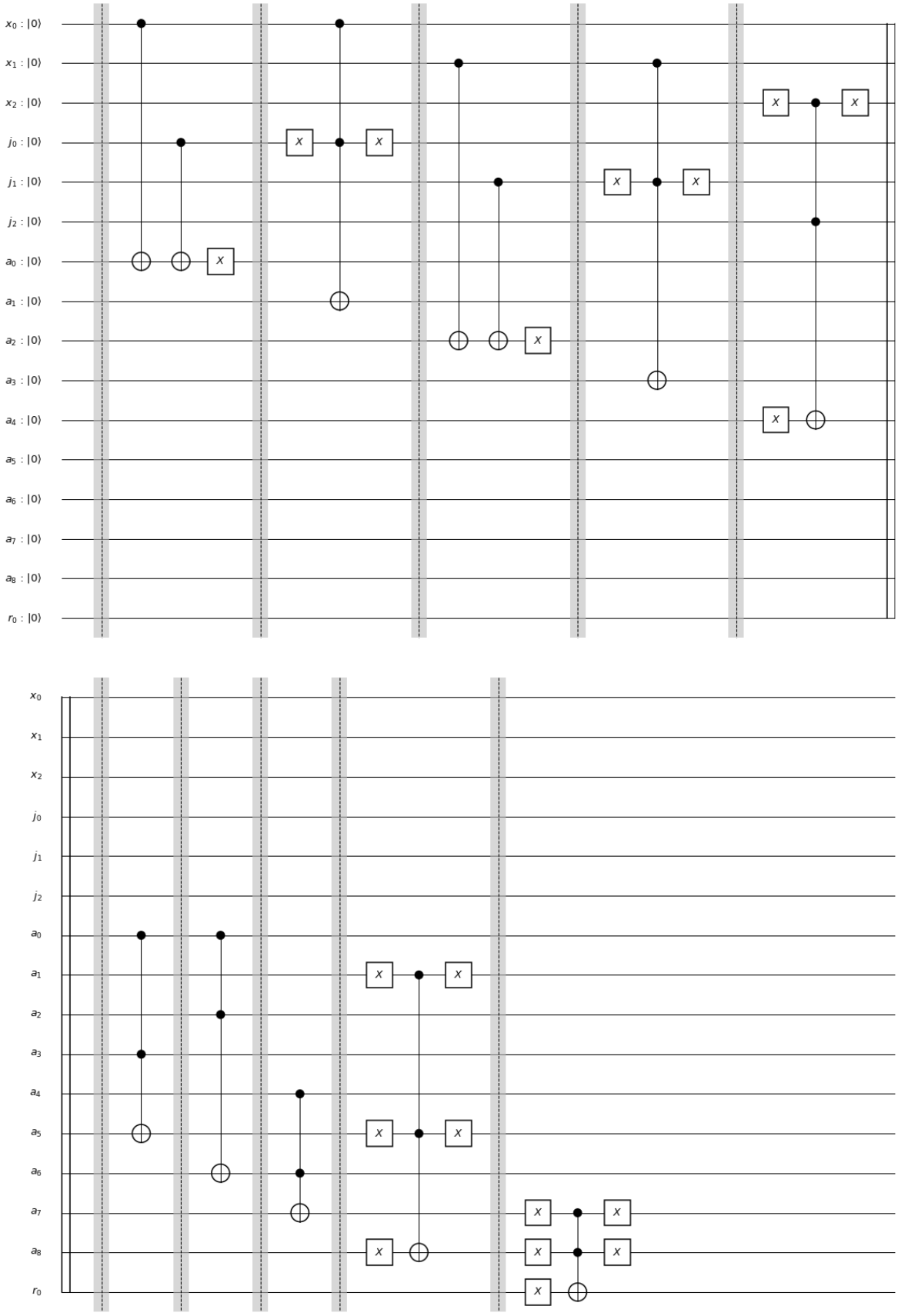


Figure 4.5: Circuito mappato

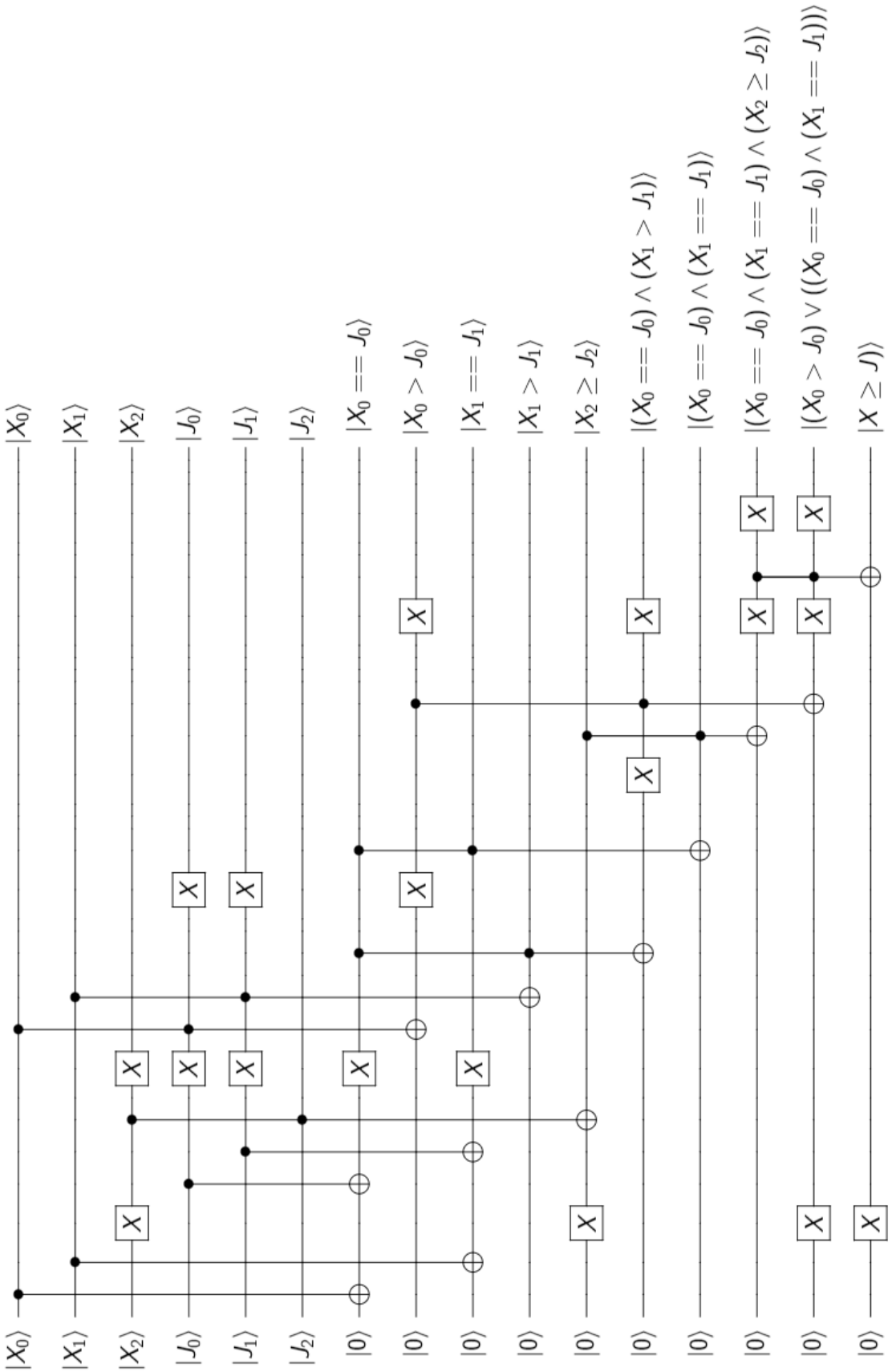


Figure 4.6: Circuito

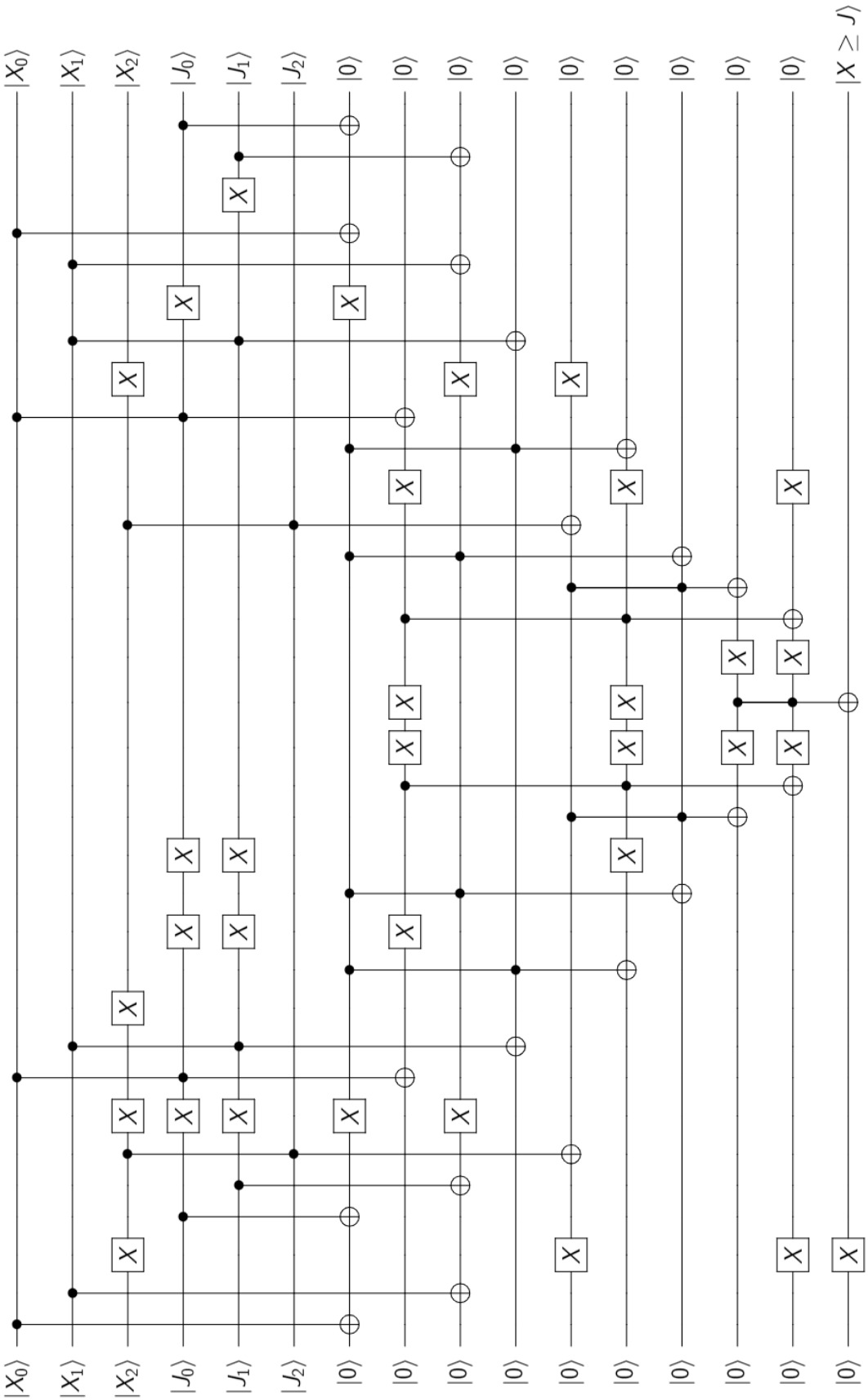


Figure 4.7: Clean circuit

4.7.1 Circuits approximation with Solovay-Kitaev's algorithm

While the techniques explained in this section are general, they are not optimal from a "time" complexity view.
It's proven that it is possible to approximate **any** Quantum Circuits with an error $\epsilon > 0$ in $O(log^c(\frac{1}{\epsilon}))$, where $c \approx 2.71$.
The value c has been improved several times since the first proposed version of this technique. [14].

This could be used to optimize the multiplexer from a linear complexity to the power of a logarithm.

Quantum Kruskal algorithm

5.1 The algorithm

In order to be able to exploit the *speed-up* allowed by quantum computation, we must slightly modify the MST algorithms. Kruskal's algorithm, among proposed algorithms, is the most efficient (having a complexity of $O(|V|log|V|)$) and the easiest to adapt.

As stated above, the classical algorithm works as follows:

Algorithm 4 Kruskal's Algorithm

Input: A connected graph $G = (V, E)$ with a positive cost function on the edges
Output: Minimum spanning tree $T = (S, F)$ of G

```
1: procedure KRUSKAL
2:     F = {}
3:     Sort edges in non-decreasing order by cost.
4:     for edge e_i, i= 1, 2, ...|E| do
5:         if F ∪ e_i has no cycle then
6:             F := F ∪ e_i
7:             if |F| = |V| − 1 then
8:                 return F and Stop;
```

In order to adapt it to quantum computing, it must be modified as shown below:

Algorithm 5 Kruskal's Algorithm modified

Input: A connected graph $G = (V, E)$ with a positive cost function on the edges
Output: Minimum spanning tree $T = (S, F)$ of G

```
1: procedure MODIFIED KRUSKAL
2:     e := min_weight E
3:     MST = {e}
4:     VISITED = {e.start, e.end}
5:     for |V|-2 times do
6:         e := min_weight{x|x ∈ E ∧ x.start ∈ VISITED ∧ e.end ∉ VISITED}
7:         MST := MST ∪ e
8:         VISITED := VISITED ∪ e.end
```

Line 6 is a research or satisfiability problem. Therefore, we might exploit Grover's algorithm to speed it up.

5.2 Oracle Creation

As stated above, in order to simplify the creation of the Oracle, we may transform the phase shift application into a boolean function application. Nevertheless, our specific problem introduces an additional complexity: it contains a function which cannot be expressed as a boolean function, namely the search of the minimum arc. Therefore, the boolean function application cannot be performed immediately. It is possible to circumvent this problem by adapting the research function[15][16].

Algorithm 6 Minimum item search

Input: A set $X \subset \mathbb{N}$
Output: $j = minX$

```
1: procedure SEARCH
2:     j := x ∈ X                          ▷ j chosen randomly from X
3:     while !convergence do
4:         I := {x | x ≤ j ∧ x ∈ X}
5:         j := i ∈ I
```

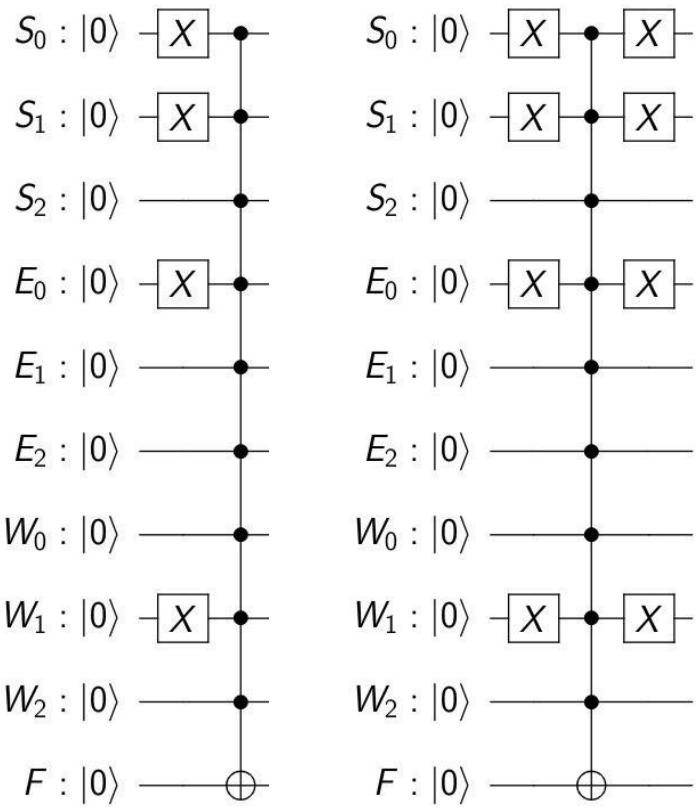
The Oracle can be expressed as follows:

$$Oracle := e \in E \wedge (e.start \in VISITED \wedge e.end \notin VISITED) \wedge e.weight < j$$

The implementation of the three sub-functions is shown below.

5.2.1 Graph membership

After having defined the oracle composition, we may discuss the sub-functions which compose it. As stated above, a graph membership function assigns the value 1 to an ancilla qubit if it is associated to an arc which belongs to the graph. In order to do so, we describe an arc as a tuple (start node, end node, weight). Then, we encode the tuple values into the registers as we have described in the previous chapter. After tuple encoding, we apply a C^nNOT having the registers describing the tuple values as control qubits and the ancilla qubit as the controlled qubit. By doing so, the ancilla qubit is set to $|1\rangle$ if and only if the elaborated arc exists in the graph.



(a) Arc 4 → 6 with weight 5 (b) Complete graph membership

To enable sequential computation, we must clean the output of the function. We apply an X Gate to every qubit to which we have previously applied an X Gate.

5.2.2 The Outgoing Arc problem

The Outgoing arc selection requires the implementation of the set membership operation.
In order to implement it with the methods aforesaid, the Boolean function must be decomposed:

$$e.start \in VISITED \wedge e.end \notin VISITED \quad = \quad \left(\bigvee_{v \in VISITED} e.start = v \right) \wedge \left(\bigvee_{v \in VISITED} e.end \neq v \right)$$

Therefore, it can be implemented with a double multiplexer and a final AND.

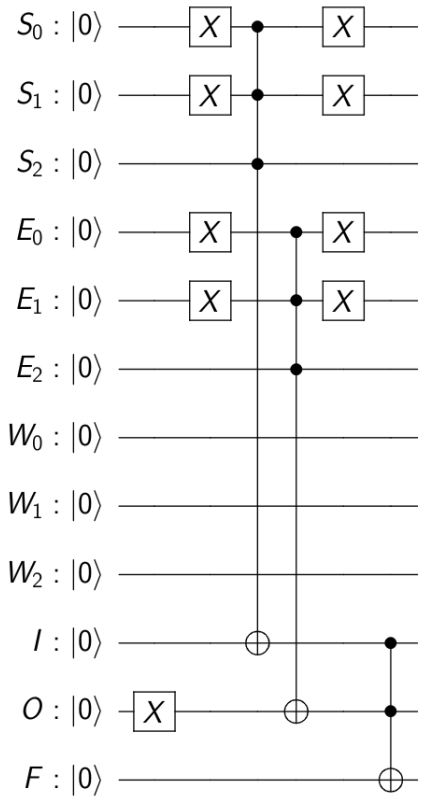


Figure 5.1: Example of Outgoing arc query with $VISITED = \{4\}$

5.2.3 The problem of finding the minimum

In order to find the minimum arc, we may implement the modified research function which we have previously described in **Algorithm 6**.

We begin by initializing a register j, set to a random weight among the weights of the graph edges. This register j will serve as the variable j exploited in **Algorithm 6**. In order to set register j to the correct value j, we must apply an X Gate on all qubits corresponding to value 1 in the binary representation of the value j. Then, by using the previously mentioned register comparator, we can implement the comparison between register j and the weights, thus selecting all weights $\leq$ value j.

Since value j is known before the construction of the Oracle, it would be possible to directly implement a comparison function for each possible value of j. The issue arising from this possible implementation, however, consists in the need to implement this function for each possible value of j, thus leading to the need of a number of circuits > 1. Therefore, we have decided not to use this last method in favor to the method previously described in this section, meaning the implementation of a single general comparator. The method we have chosen requires a higher number of qubits but a single circuit.

5.3 Registers

The arcs can be represented as a triplet (s, e, w) containing the weight, the starting vertex and ending vertex. e.g. An arc from 1 to 4 with weight 3 can be represented as:

$$1 \xrightarrow{3} 4 \quad \rightarrow \quad (1, 3, 4)$$

A non-directed graph can be emulated by implementing it as directed graph with both forward and backward arcs. e.g. An arc from 1 to 3 with weight 5, non-directed:

$$1 \xleftrightarrow{5} 3 \quad \rightarrow \quad \{(1, 5, 3), (5, 1, 3)\}$$

Therefore, the proposed circuit setup needs 3 k qubit registers to represent the graph: one for the start vertex, one for the end vertex and one for the weight.
Assuming the vertices take values in the range $[0, |V| - 1]$, $k = \lceil log|V| \rceil$ it's the size of the 3 registers.

On top of it, a flag register with 4 qubits is needed, one term of the formula:

$$e \in E \wedge (e.start \in VISITED \wedge e.end \notin VISITED) \wedge e.weight < j$$

Moreover, the number of qubits needed for the ancillas are:

$$\approx 3k$$

Therefore, the total number of qubits is:

$$7k + 4 \sim O(k) \sim O(log|V|))$$

At the time of writing, the Quantum Computer with the highest number of qubits has 22 qubits. Therefore, at the current state, this algorithm can be implemented only for graphs with at most 4 vertices.

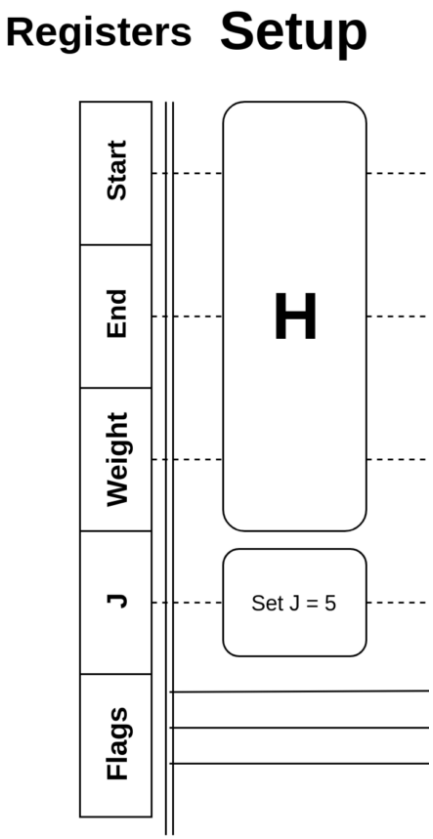


Figure 5.2: Registers and their setup

5.4 Circuit

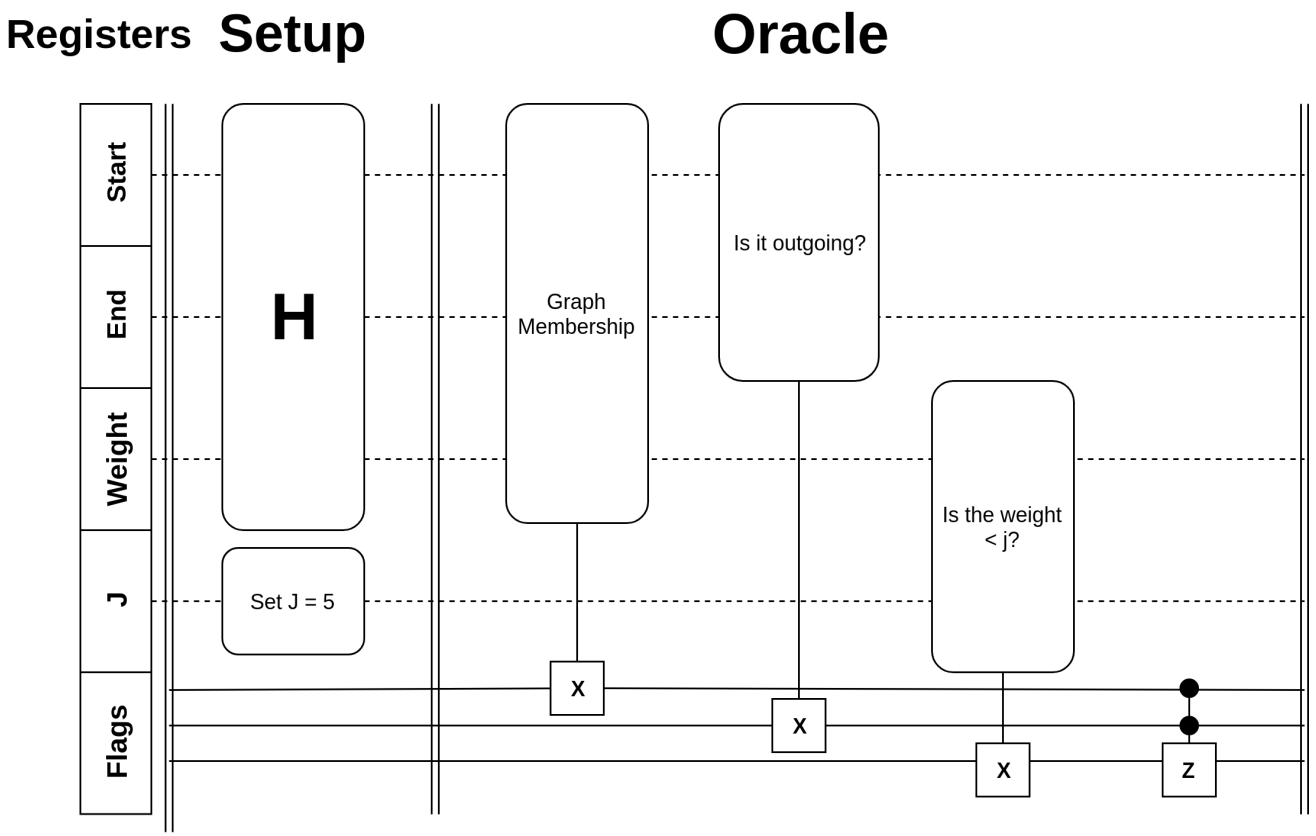


Figure 5.3: The high level description of the circuit

From our experimental results, as the flags are not brought into a superposition, there is a high probability of their correctness after the application of Grover's algorithm. Since, in the proposed setup, the searched arc must have all the flags set to 1, this can be exploited to check for errors.

5.5 The number of iterations

In order to compute the right number of iterations for Grover's algorithm, it is needed to know the number of solutions. This cannot be known a priori in the general case.

$$\#\ iterations = \frac{\pi}{4}\sqrt{\frac{N}{M}} \quad where\ M\ is\ the\ number\ of\ matches$$

There are two solutions for this.
By applying a Quantum Fourier transform after the application of Grover's algorithm, it's possible to obtain the number of solutions[17].
This doubles the complexity of the algorithm, but it doesn't change the asymptotic complexity.

There is also a more sophisticate technique presented in the article "Tight bounds on quantum searching"[18] which is proven to maintain the complexity $O\left(\sqrt{\frac{N}{M}}\right)$.

5.6 Final complexity

Grover's algorithm has a query complexity of $O(\sqrt{|E|}) \sim O(|V|)$. The final Time complexity takes into account the time spent for the creation of the Oracle. The Oracle exploited to solve the problem presented in this document is made up of three functions, having respectively the following Time complexities:

- Arc membership function: $O(|E|) \sim O(|V|^2)$;
- Outgoing arc selection function: $O(|V|)$;
- Weight comparison function: $O(log|V|)$

The final complexity equals to the greater complexity among those belonging to the sub-functions, which is $O(|E|)$. Therefore, the complexity of the full algorithm is [16]:

$$\underbrace{O(\sqrt{|E|})}_{\text{Query Complexity}} \times \underbrace{O(|E|)}_{\text{Oracle Construction}} = \underbrace{O(|E|^{\frac{3}{2}})}_{\text{Time Complexity}}$$

Which corresponds to

$$O(|V|^3)$$

The final complexity is significantly greater than the one of the classical implementation of Kruskal's algorithm.

$$\underbrace{O(|V|^3)}_{\text{the proposed implementation}} >> \underbrace{O(|V|^2 log|V|)}_{\text{classical complexity}}$$

This result suggests that quantum computing, at the state of the art, does not offer any practical benefit to the solution of the MST problem.

5.6.1 Future theoretical speed-ups

Architectural optimization: oracle reuse

The issue concerning the overall complexity may be solved by an eventual change in quantum computer architecture: if future quantum computer will allow to reuse the Oracle, thus making it necessary to only build it once, the overall complexity will significantly decreasing, becoming:

$$\underbrace{O(\sqrt{|E|})}_{\text{Grover}} + \underbrace{O(|E|)}_{\text{Oracle construction}} = O(|E|)$$

However, this cannot be realized on the quantum computers which are available nowadays, nor can be stated with certainty whether they will be implemented one day.

It is not certain that this optimization can be actually realized in real-world quantum computers, due to possible physical limitations. However, this analysis goes beyond the purpose of this document.

Architectural optimization: QRAM

The QRAM Architectural design is based on the bucket-brigade RAM model[19]. This allows to retrieve data from memory in $O(log(n))$:

$$\sum_i \alpha_i |x_i\rangle \xrightarrow{QRAM} \sum_i \alpha_i |x_i\rangle |f(x_i)\rangle$$

This optimization permits to only need to encode the graph in memory once, which leads to a significant improvement in overall complexity.

Combining the Oracle reuse optimization described above and the QRAM optimization, the final complexity obtained would be:

$$\underbrace{O(\sqrt{|E|})}_{\text{Grover}} + \underbrace{O(log(|E|))}_{\text{Oracle construction}} = O(|V|)$$

Since the MST is composed by $|V| - 1$ edges, the best complexity achievable is $O(|V|)$. Therefore, if the aforesaid optimizations become feasible in the future, the MST will be a solved problem.

The classical state of the art

The theoretically achievable linear complexity presented in the previous section does not improve significantly the complexity of the best classical algorithm. Borůvka's and Kruskal's algorithms can be parallelized and optimized to achieve a complexity of:

- Probabilistically: $O(|E|)$
- Deteministically: $O(|E|\alpha(|E|,|V|))$, where α is the inverse of the Ackermann function.

The complexity theoretically obtainable by combining the two quantum optimization presented would introduce a quadratic speed-up in comparison to improved Borůvka's algorithm complexity.

$$\underbrace{O(|E|) \sim O(|V|^2)}_{\text{classical}} \quad > \quad \underbrace{O(|V|)}_{\text{quantum}}$$

The complexities can be summarized:

$$\underbrace{O(|V|^3)}_{\text{the proposed implementation}} \quad >> \quad \underbrace{O(|V|^2 log|V|)}_{\text{classical complexity}} \quad > \quad \underbrace{O(|E|) \sim O(|V|^2)}_{\text{classical state of the art}} \quad > \quad \underbrace{O(|V|)}_{\text{theoretic quantum \& lower bound}}$$

Frameworks Comparison

6.1 IBM qiskit

6.1.1 The Framework

Qiskit is a python library having a simulator written in C++.
The framework is developed by IBM and aims mainly to execute code on their Quantum Computers.
This is the main reason why Qiskit is one of the slowest Quantum Computing Simulators.

6.1.2 Terra

Qiskit Terra provides the foundational roots for our software stack. Within Terra is a set of tools to compose quantum programs at the level of circuits and pulses, optimizing them for the constraints of a particular physical quantum processor, and managing the batched execution of experiments on remote-access back-ends. Terra is constructed modularly, thus simplifying the addition of extensions for circuit optimizations and back-ends.

6.1.3 Aer

Qiskit Aer provides a high performance simulator framework for the Qiskit software stack. It contains optimized C++ simulator back-ends for executing circuits compiled in Qiskit Terra, and tools to construct highly configurable noise models to perform realistic noisy simulations of the errors that occur during the execution on real devices.

6.1.4 Aqua

Qiskit Aqua contains a library of cross-domain quantum algorithms upon which applications for near-term quantum computing can be built. Aqua is designed to be extensible, and employs a pluggable framework where quantum algorithms can easily be added. It currently allows the user to experiment on chemistry, AI, optimization and finance applications for near-term quantum computers.

Aqua's main applications domains are:

- Chemistry
- Artificial Intelligence
- Optimization
- Finance

The most notable algorithm are:

- Grover
- Amplitude Estimation
- Shor
- Simon
- Bernstein-Vazirani
- Deutsch-Jozsa

A great issue is that currently Aqua's algorithms are not compatible with Terra's circuits.

6.1.5 Ignis

Qiskit Ignis is a framework to understand and mitigate noise in quantum circuits and devices. Ignis provides self-contained experiments that include tools to generate the circuits that can be executed on real back-ends via Terra (or on simulators via Aer) and tools to fit the results and analyse the data. The experiments provided in Ignis are grouped into the topics of characterization, verification and mitigation. Characterization experiments, such as T1 (qubit lifetime) and T2 (qubit dephasing), are designed to measure noise parameters in the system. Verification experiments, such as randomized benchmarking and tomography, are designed to verify the performances of gates and small circuits. Mitigation experiments run calibration circuits which are analysed to generate mitigation routines which can be applied to arbitrary sets of results running on the same back-end.

6.1.6 Documentation

The GitHub page of IBM's Qiskit project is full of examples, tutorials, and documentation.

The major problem is that, since the Framework has so many features meant for researchers, the tutorials escalate from trivial to extremely advanced really fast.
Moreover, as it is a relatively "new" framework, often the only way to get information about functions is to read the source code.

6.1.7 Performance & Parallelization

The major downside of having such a low-level and realistic simulation is the abysmal simulation performance.
Moreover, since it lacks the ability to simulate Multiple-Controls-Multiple-Targets gates, it requires to have ancilla qubits. Therefore, the number of necessary qubits is usually the **double** compared to the same implementation on other frameworks.
Since the simulation tends to scale exponentially with regards to the number of qbits, the performance of Qiskit usually have a **quadratic slow-down**.
At the time at which this document has been written, the Qiskit's Aer simulator computes on a single core.
This obviously doesn't scale well, but in the future it might be updated.
On Qiskit Github repository, a pull request of 9.5k lines of code has been opened: it adds Multi-CPU & Multi-GPU support using openCL.
It was open in 05/2019, therefore by the beginning of 2020 we might see it added to Qiskit.

6.2 Cirq

6.2.1 The Framework

Cirq is a framework which allows to create quantum circuits and simulate their execution.

Its alpha version was released in July 2018, thus making Cirq a rather young framework. Its development is still ongoing, as new version are continuously released, implementing an increasing amount of functionalities. No final version has been released by now, and Cirq still appears far from completion. Its short lifespan up to now has caused several consequences, which will be analyzed later on in this section.

It was developed by Google and released as an opensource framework, thus granting the users and developers a certain degree of freedom. It is available as a Python library.

6.2.2 Documentation

The documentation behind Cirq is sparse and incomplete, mostly due to the framework's short lifespan. It can can be read on the platform Read the Docs. It contains a concise description of most methods and of some objects which are exposed by the framework. Only basic items have a detailed, user-friendly description. Most advanced objects and methods are not accompanied by an explanation nor by any examples.

The framework offers a few tutorials, accompanied by some examples which show how to use certain functionalities. However, this examples only belong to two categories: basic examples and extremely advanced examples. No medium-level examples are provided, thus making it difficult to developers to truly master the framework. Moreover, most advanced features are not used in any examples.

A very limited amount of additional examples is available on GitHub. Nevertheless, these few examples are not sufficient to cover the needs of developers, as they expand those presented on Read the Docs only of a small set of features.

6.2.3 Features

The framework offers some advanced features, which can be useful when creating quantum circuits.

- **Controlled gates:** Cirq allows to automatically generate controlled gates; controlled and controller qubits can be arbitrarily chosen, thus allowing a wider set of operations in comparison to frameworks which do not provide this feature. Therefore, it is not needed to implement controlled gates ex-novo, which is a rather complex operation.
- **Power gates:** As gates can be seen as matrices and matrices can be raised to a power, it is possible to apply an exponent to a gate. For instance, it is possible to apply a square root to a gate. Cirq supports this operation, as it offers some classes of *pow* gates, meaning gates which are raised to a power. The exponent can be chosen when creating or applying the gate. The applications of this class of operations is beyond the purpose of this document.
- **Qubits as lists:** Cirq offers an easy, quick way to handle qubits. Qubits are created as lists, and gates can be applied to them simply by specifying on which items of the list they must iterate on. This specification happens when gates are appended to a circuit. Conditional controls can be applied when iterating on the list: therefore, it is for instance possible to apply a gate only to qubits having an even position in the list, simply by using an *if* statement when iterating on the qubit list. This provides a great flexibility when choosing which qubits a gate must be applied to.

6.2.4 Simulator

Cirq allows to simulate the execution of a circuit on a real quantum environment. In order to simulate a likely environment, the framework offers the possibility to add simulated noise to a circuit. It is possible to add noise by randomly applying to the circuit evolutions with different probabilities. For instance, an X Gate may be applied with a 50% probabilty. If the gate is applied, it flips the qubit it is applied to, otherwise it does nothing. As it is applied with a certain probability, it is not possible to foresee whether it will happen or not. Therefore, this successfully emulates some effects of noise on a circuit.

6.2.5 Final considerations

Cirq is certainly a powerful tool which provides many features, both with regards to circuit construction and to simulation. It provides several advanced features, which help in the creation of complex quantum circuits. It also allows to simulate

the computation in a likely environment, granting at the same time some satisfying performances, as will be shown later on in this document. Its main liability is the lack of medium-level documentation and of examples. This framework's development appears to be far from being over: newer versions are continuously released, on an almost monthly basis. It is likely that newer, more stable future versions will attract a higher number of users, which may lead to the development of more examples and documentation. Up to that moment, the framework's potential appears great but unsufficiently exploited.

6.3 Quantum Development Kit

6.3.1 Documentation

The documentation behind QDK is maybe one of the most complete among the various quantum platforms. There are various ways to start interacting with the QDK

- Tutorial on their website.
- Dedicated Github page.

The website explains very precisely, with a step-by-step tutorial, how to install all mandatory tools in order to use properly their platform. The installation of the required tools is done through .NET platform and directly from Microsoft website. On their website, we can also find a very complete reference of their new programming language completely dedicated to quantum computing, Q#, and various examples of basic quantum circuits. On the other hand, on Github we can find more complex examples which teach us how to properly use Q# and its integration in the C# ecosystem.

6.3.2 Features

Q#

The most important feature of QDK is its dedicated programming language, Q#. The model Microsoft is pursuing for quantum computation is to treat a quantum computer as a coprocessor, similarly to GPUs, FPGAs. The primary control logic runs classical code on a classical "host" computer. When appropriate and necessary, the host program can invoke a subroutine that runs on the adjunct processor. When the subroutine completes, the host program gets access to the subroutine results. Q# is a domain-specific programming language used to express quantum algorithms. It is meant to be used to write subroutines that execute on an adjunct quantum processor, under the control of a classical host program and computer running a classical driver written in C#.

The syntax of Q# is rather different from the languages of the other platforms. It closely resembles C# and is more verbose than Python. Its key concept is the "operation" construct. This is a callable routine, which is invoked by the C# driver, with quantum operations. The "operation" is like a function in a classical programming language but it manipulates quantum gates, thus creating a new operation is like building a new quantum gate. This fact imposes to the operation the same restrictions which are applied to quantum gates. This direct transposition enables the birth of new constructs which are closer to the actual representation of quantum circuits. One example is provided by keywords adj(adjointed) and Ctl(controlled), which permit, if an operation is unitary, to define its behaviour when reversed or controlled by a quantum register. It's also worth mentioning that Q# has a type model but permits the compiler to infer the type of the newly initialised variables.

Libraries

QDK environment gives us a very rich ecosystem of libraries, which implement the vast majority of basic gates and algorithms currently known. This library list grows day after day allowing developers to have ready-to-use operators and speeding up development. Some notable mentions are:

- **Microsoft.Quantum.diagnostic** library allows developer to build tests in order to check the state of the machine, when possible. Its structure resembles the one of JUnit, with an assert-like structure.
- **Microsoft.Quantum.Simulation.Simulator** gives access to a wide variety of simulators with different properties. This allows developers to use specialised simulators for some applications. This topic will be covered in depth in following sections.

Visual Studio extension

Another great feature is the ability to use QDK in the Visual Studio environment. This enables us to manage in a more precise way the interactions and the execution of the code. The integration in Visual Studio also allows developers to have a more precise control over the QDK libraries. Another great feature is the debugging ability of VS.

Driver variety

While Q# must be used for every part of the code manipulating quantum states, driver code, meaning the portion of code which invokes Q# operations, could be written in different ways

- C# driver is the most frequent choice. It's also the recommended programming language due to its similarities with Q#.
- Python
- Jupyter notebook

6.3.3 Simulator

As it is not possible to have easy access to a physical quantum computer, simulation covers a very important role in the QDK platform. There are four main simulators in QDK:

- **Full state vector simulator** is the standard and most used simulator. This simulator can be used to execute and debug quantum algorithm. There are two main features: one consists in being able to simulate quantum randomness and the other consists in the exploitation of multithreading to parallelize linear algebra operations required for the simulations. The number of threads and the workload can be decided before the execution of the simulator starts. The limit of qubits for the local simulator is 30, if we run the code from the Azure cloud platform the available qubits raise to 40.
- **Toffoli simulator** is a simplified simulator in which the only available gates are X, CNOT an multi-controlled X. This limitation permits to run up to millions of qubits at the same time in a local simulator.
- **Resources estimator** estimates the resources required to run a given instance of a Q# operation on a quantum computer. It accomplishes this by executing the quantum operations without actually simulating the state of a quantum computer; for this reason, it can estimate resources for Q# operations that use thousands of qubits. The resources estimator is just another type of target machine, thus it can be used to run any Q# operation. The output of the estimator is the number of gates used by the algorithm.
- **Trace-based resource estimator** is similar to the simpler resources estimator, as it is able to execute quantum operations without actually simulating the state of a quantum computer. Iin addition, it can use user information about the probability of certain states during the computation, in order to better understand the behaviour of the simulation. This way, it can return richer information about the resources needed for the computation of the actual algorithm.

6.3.4 Hardware

Unlike the superconducting qubit technology of Rigetti and IBM, Microsoft is betting highly on topological qubits based on Majorana fermions. These particles have recently been discovered and promise long coherence times and other desirable properties, but no functional quantum computer using topological qubits currently exists. So all Q# applications can't be executed on physical machines exploiting this new technology, but can be executed, like Qiskit, on IBMQuantumExperience simply by changing the driver.

6.3.5 Final considerations

QDK is one of the most flexible quantum environments currently available. Its dedicated quantum programming language is one of a kind in the quantum world and gives the developer more freedom considering that Q# has been created specifically for quantum computation. The learning curve could be a little slower with this approach, but we think that in the long term it could bring some advantages.

The only downside of this platform is the absence of a real quantum machine to test the written code on. This is due to the age of the project, which is the youngest among the platforms discussed in this report, and the technology chosen for future physical hardware.

6.4 Framework inter-operability

IBM has created a "standard" language to formalize quantum circuits [20].
This language it's called OpenQasm and it's currently supported by Qiskit and Cirq.
This allows to export and import circuits created in different frameworks.

6.5 Environment setup

In order to keep the environment clean and reproducible, we have built a docker image.

6.5.1 Build & Run

Script to build the container and have the shell on the environment

```
#!/bin/bash
echo "############################################################"
echo "Stopping the istance if there is (This might take a few seconds)"
echo "############################################################"
docker stop performance-test
docker rm performance-test

echo "############################################################"
echo "Build the container"
echo "############################################################"
docker build --file Dockerfile -t quantum-test-env .

echo "############################################################"
echo "Run the container"
echo "############################################################"
docker run -v $PWD/reports:/quantum/reports -it quantum-test-env
```

Listing 1: build_and_run_container.sh

6.5.2 DockerFile

```
FROM ubuntu:18.04
# Install the basic utils we will need
RUN apt-get update && \
  apt-get install -y gcc time vim wget bash neofetch software-properties-common && \
  apt-get upgrade -y
# Create a folder where we will work
RUN mkdir -p /quantum && chmod 777 /quantum
WORKDIR /quantum

# Make the instance killabel with Ctrl-C
STOPSIGNAL 15

# Setup .NET for Q#
RUN wget -q https://packages.microsoft.com/config/ubuntu/18.04/packages-microsoft-prod.deb \
  -O packages-microsoft-prod.deb
RUN dpkg -i packages-microsoft-prod.deb && rm packages-microsoft-prod.deb
RUN add-apt-repository universe
RUN apt-get install -y apt-transport-https
RUN apt-get update
RUN apt-get install -y dotnet-sdk-2.2
# Install Q#
RUN dotnet tool install -g Microsoft.Quantum.IQSharp
# Update Q#
RUN dotnet tool update -g Microsoft.Quantum.IQSharp

# Install python3 with miniconda
RUN wget -q https://repo.anaconda.com/miniconda/Miniconda3-latest-Linux-x86_64.sh -O miniconda.sh
RUN mkdir -p /root
RUN bash miniconda.sh -b
RUN rm miniconda.sh

# Add the executables to path
RUN ln -s /root/miniconda3/bin/python3 /bin/python3
RUN ln -s /root/miniconda3/bin/python3 /bin/python
RUN ln -s /root/miniconda3/bin/pip /bin/pip

# Install the library needed by cirq
RUN apt-get install -y libx11-6
# Install cirq
RUN python -m pip install cirq

# Install qiskit & qiskit-aqua
RUN python -m pip install qiskit qiskit_aqua

# Synchronize the scripts between the folder and the docker
ADD scripts /quantum/scripts

# Add the test script
ADD grid_test.sh /quantum
ADD test_all.sh /quantum

# Clear the apt-get cache
RUN rm -rf /var/lib/apt/lists/*Page
```

Listing 2: Dockerfile

6.5.3 Test All Frameworks

Test all the frameworks with the same parameters

```
#!/bin/bash
printf "################################################################################################\n"
printf "New Batch\n"
printf "################################################################################################\n"

printf "Test Description:\n"
printf "\tTest of the 3 freamwork executing the Grover search of a single number\n"
printf "\tWith $1 searchable qbits and $2 repetitions\n"

printf "\nSystem specifications\n"
neofetch --off --color_blocks off --stdout

printf "\nTesting cirq runtine\n"
/usr/bin/time  --verbose python scripts/cirq/grover_cirq.py $1 $2 > /dev/null

cd scripts/qsharp
# Call the script once before time it so that we don't measure the building
printf "\nTesting Q# runtime\n"
/usr/bin/time  --verbose dotnet run  $1 $2 > /dev/null
cd ../..

printf "\nTesting qiskit runtime without optimizations\n"
/usr/bin/time  --verbose python scripts/qiskit/grover_qiskit.py  $1 $2 > /dev/null

printf "\nTesting qiskit runtime with optimizations\n"
/usr/bin/time  --verbose python scripts/qiskit/grover_qiskit_with_optimizations.py  $1 $2 > /dev/null
```

Listing 3: test_all.sh

6.5.4 Grid Test

Test all the frameworks with different combinations of number of qbits and number of shots

```
#!/bin/bash
# Call the script once before time it so that we don't measure the building
cd scripts/qsharp
dotnet run 5 1 > /dev/null
cd ../..

for I in {1..5}
do
    for N_OF_QBITS in {5..10}
    do
        SHOTS=$(python -c "print(10 ** $I)")
        FILENAME="report"
        FILENAME+=$(printf "%02d" $N_OF_QBITS)
        FILENAME+="_"
        FILENAME+=$SHOTS
        echo $FILENAME
        bash test_all.sh $N_OF_QBITS $SHOTS > reports/$FILENAME 2>&1
    done
done
```

Listing 4: grid_test.sh

6.6 Benchmarks

6.6.1 Benchmarks description

To compare the three frameworks, we need a common task to analyze.
In these benchmarks, the task will be based on Grover's search algorithm.
The 3 scripts must receive the number of qbits and the number of shots as arguments.

<SCRIPT> <NUMBER OF QBITS> <NUMBER OF SHOTS>

The run-time statistics have been collected using GNU implementation of *time*, using the *–verbose* flag.
The output is shown below:

```
$ /usr/bin/time --verbose python scripts/cirq/grover_cirq.py 5 10 > /dev/null
        Command being timed: "python scripts/cirq/grover_cirq.py 5 10"
        User time (seconds): 1.57
        System time (seconds): 0.99
        Percent of CPU this job got: 189%
        Elapsed (wall clock) time (h:mm:ss or m:ss): 0:01.35
        Average shared text size (kbytes): 0
        Average unshared data size (kbytes): 0
        Average stack size (kbytes): 0
        Average total size (kbytes): 0
        Maximum resident set size (kbytes): 103476
        Average resident set size (kbytes): 0
        Major (requiring I/O) page faults: 0
        Minor (reclaiming a frame) page faults: 34590
        Voluntary context switches: 29
        Involuntary context switches: 760
        Swaps: 0
        File system inputs: 0
        File system outputs: 80
        Socket messages sent: 0
        Socket messages received: 0
        Signals delivered: 0
        Page size (bytes): 4096
        Exit status: 0
```

We will exclusively analyze Time (as the sum of *User time, System time*) and Memory (*Maximum resident set size*).

6.6.2 System specs

All the tests were done on the same machine, inside the same container and in the same conditions.
The machine has the following specifications:

```
OS: Ubuntu 18.04.2 LTS x86_64
Host: XPS 15 9570
Kernel: 4.19.49-1-MANJARO
Uptime: 57 mins
Packages: 227
Shell: bash 4.4.19
CPU: Intel i7-8750H (12) @ 4.100GHz
Memory: 2965MiB / 31793MiB
```

6.6.3 Qiskit test program

```
import sys
import qiskit as q
import qiskit.aqua

from math import floor, pi, sqrt

n_of_qbits = int(sys.argv[1])
# Inizzializzazione dei qbit e i bit su cui fare la ricerca
register = q.QuantumRegister(n_of_qbits, name="values")
# Lista dei singoli qbits per averne accesso piu' agevole
qbits = [q for q in register]
# Inizzalizzazione degli ancilla qbits che saranno necessari per il cnx gate
ancillas = q.QuantumRegister(n_of_qbits, name="ancillas")

# Creazione del circuito quantum
circuit = q.QuantumCircuit(register, ancillas)
# Mettiamo in superposizione i qbits
circuit.h(qbits)


# Costruiamo un oracolo d'esempio che cerca il valore 1010 0111
oracle = q.QuantumCircuit(register, ancillas)
# Poniamo un gate X in corrispondenza dei bit uguali a 0 nel valore cercato
oracle.x(qbits[1])
oracle.x(qbits[3])
oracle.x(qbits[4])

# costuriamo un (c^n Z) per poter selezionare il valore
oracle.mcmt(qbits[:-1], ancillas, q.QuantumCircuit.cz, [qbits[-1]])

# Rimettiamo a posto lo stato dei qbits rieseguendo le operazioni al contrario
oracle.x(qbits[4])
oracle.x(qbits[3])
oracle.x(qbits[1])


def construct_diffusion_operator(registers, qbits, ancillas) -> q.QuantumCircuit:
    """Create the diffusion circuit for the registers."""
    diffusion = q.QuantumCircuit(*registers, ancillas)

    # Creiamo un muro di H e X
    for qbit in qbits:
        circuit.h(qbit)
        circuit.x(qbit)

    # Poiche' non esiste il C^n Z su qiskit sfruttiamo ancora HXH = Z
    oracle.mcmt(qbits[:-1], ancillas, q.QuantumCircuit.cz, [qbits[-1]])
```

Listing 5: grover_qiskit.py

```
    # Secondo muro di X e H
    for qbit in qbits:
        diffusion.x(qbit)
        diffusion.h(qbit)

    return diffusion


def construct_grover(circuit : q.QuantumCircuit, oracle : q.QuantumCircuit, registers, qbits, ancillas, numbe
    """Create the circuit to perform a grover search given the orcale"""
    diffusion = construct_diffusion_operator(registers, qbits, ancillas)

    # Calcoliamo il numero di iterazioni previste
    number_of_iterations = (pi / 4)*sqrt((2**len(qbits)) / number_of_expected_results)
    # Arrotondiamo per difetto in quanto sperimentalemnte sembra dare i risultati migliori
    number_of_iterations = floor(number_of_iterations)

    # RIipetiamo oracolo e diffusione il numero calcolato di volte
    for _ in range(number_of_iterations):
        circuit += oracle
        circuit += diffusion

    return circuit

# Applichiamo grover
circuit = construct_grover(circuit, oracle, [register], qbits, ancillas)

# Inizzializziamo il backend del simulatore
backend_sim = q.BasicAer.get_backend('qasm_simulator')

# Setuppiamo il circuito per simularlo

# Inizzializziamo un registro di qbit classici
# I bit normali servono come registi dove il simulatore andra' a salvare il risultato dell'esperimento
cbits = q.ClassicalRegister(n_of_qbits, 'classical_values')

# Ed aggiungiamoli al circuito

circuit.add_register(cbits)
circuit.measure(register, cbits)

# Otteniamo i risultati organizzati come frequenza degli stati misurati
results = q.execute(circuit, backend_sim, shots=int(sys.argv[2])).result().get_counts(circuit)

# Stampiamo a schermo il risultato
print(results)
```

6.6.4 Qiskit with optimizations test program

```
import qiskit as q
import qiskit.aqua

from math import floor, pi, sqrt

# Inizzializzazione dei qbit e i bit su cui fare la ricerca
register = q.QuantumRegister(6, name="values")
# Lista dei singoli qbits per averne accesso piu' agevole
qbits = [q for q in register]
n_of_qbits = len(qbits)
# Inizzalizzazione degli ancilla qbits che saranno necessari per il cnx gate
ancillas = q.QuantumRegister(6, name="ancillas")

# Creazione del circuito quantum
circuit = q.QuantumCircuit(register, ancillas)
# Mettiamo in superposizione i qbits
circuit.h(qbits)


# Costruiamo un oracolo d'esempio che cerca il valore 1010 0111
oracle = q.QuantumCircuit(register, ancillas)
# Poniamo un gate X in corrispondenza dei bit uguali a 0 nel valore cercato
oracle.x(qbits[1])
oracle.x(qbits[3])
oracle.x(qbits[4])

# costuriamo un (c^n Z) per poter selezionare il valore
oracle.mcmt(qbits[:-1], ancillas, q.QuantumCircuit.cz, [qbits[-1]])

# Rimettiamo a posto lo stato dei qbits rieseguendo le operazioni al contrario
oracle.x(qbits[4])
oracle.x(qbits[3])
oracle.x(qbits[1])


def construct_diffusion_operator(registers, qbits, ancillas) -> q.QuantumCircuit:
    """Create the diffusion circuit for the registers."""
    diffusion = q.QuantumCircuit(*registers, ancillas)

    # Creiamo un muro di H e X
    for qbit in qbits:
        circuit.h(qbit)
        circuit.x(qbit)

    # Poiche' non esiste il C^n Z su qiskit sfruttiamo ancora HXH = Z
    oracle.mcmt(qbits[:-1], ancillas, q.QuantumCircuit.cz, [qbits[-1]])

    # Secondo muro di X e H
```

```
    for qbit in qbits:
        diffusion.x(qbit)
        diffusion.h(qbit)

    return diffusion


def construct_grover(circuit : q.QuantumCircuit, oracle : q.QuantumCircuit, registers, qbits, ancillas, numbe
    """Create the circuit to perform a grover search given the orcale"""
    diffusion = construct_diffusion_operator(registers, qbits, ancillas)

    # Calcoliamo il numero di iterazioni previste
    number_of_iterations = (pi / 4)*sqrt((2**len(qbits)) / number_of_expected_results)
    # Arrotondiamo per difetto in quanto sperimentalemnte sembra dare i risultati migliori
    number_of_iterations = floor(number_of_iterations)

    # RIipetiamo oracolo e diffusione il numero calcolato di volte
    for _ in range(number_of_iterations):
        circuit += oracle
        circuit += diffusion

    return circuit

# Applichiamo grover
circuit = construct_grover(circuit, oracle, [register], qbits, ancillas)

# Inizzializziamo il backend del simulatore
backend_sim = q.BasicAer.get_backend('qasm_simulator')
# Optimze the circuit
circuit = q.compiler.transpile(circuit, backend=backend_sim, optimization_level=2)

# Setuppiamo il circuito per simularlo

# Inizzializziamo un registro di qbit classici
# I bit normali servono come registi dove il simulatore andra' a salvare il risultato dell'esperimento
cbits = q.ClassicalRegister(n_of_qbits, 'classical_values')

# Ed aggiungiamoli al circuito

circuit.add_register(cbits)
circuit.measure(register, cbits)

# Si compila il circuito per ottimizzarlo e renderlo eseguibile
# Ed imponiamo che il simulatore faccia 10^4 simulazioni
qobj = q.compiler.assemble(circuit, shots=100, seed_simulator=42)

# Otteniamo i risultati organizzati come frequenza degli stati misurati
results = backend_sim.run(qobj).result().get_counts(circuit)

# Stampiamo a schermo il risultato
print(results)
```

Listing 6: grover_qiskit_with_optimizations.py

6.6.5 Cirq test program

```
import sys
import cirq
from math import floor, pi, sqrt
length = int(sys.argv[1])
number_of_expected_results = 1
qubits = [cirq.GridQubit(i, 0) for i in range(length)]

class Grover():
    def __init__(self, qubits):
        self.qubits = qubits
        self.diffusion = self.make_diffusion()


    def make_diffusion(self):

        circuit = cirq.Circuit()
        circuit.append(cirq.H(q) for q in self.qubits)
        circuit.append(cirq.X(q) for q in self.qubits)
        controllee= []
        for i in range(length - 1):
            controllee.append(self.qubits[i])

        controlled = cirq.control(cirq.Z, controllee)(qubits[-1])

        circuit.append(controlled)

        circuit.append(cirq.X(q) for q in self.qubits)
        circuit.append(cirq.H(q) for q in self.qubits)

        return circuit


    def run(self, circuit, oracle, number_of_expected_results : int = 1):
            number_of_iterations = floor((pi / 4)*sqrt((2**len(self.qubits)) / number_of_expected_results))
            for _ in range(number_of_iterations):
                circuit.append(oracle)
                circuit.append(self.diffusion)
            return circuit

def make_oracle(qubits):
    circuit = cirq.Circuit()
    controllee= []
    for i in range(length - 1):
        controllee.append(qubits[i])
    circuit.append(cirq.X(qubits[i]) for i in [1,3,4])
    circuit.append(cirq.control(cirq.Z, controllee)(qubits[-1]))
    circuit.append(cirq.X(qubits[i]) for i in [1,3,4])
    return circuit
```

```
grover = Grover(qubits)
oracle = make_oracle(qubits)
circuit = cirq.Circuit()
circuit.append(cirq.H(q) for q in qubits)
grover.run(circuit, oracle, number_of_expected_results)
circuit.append(cirq.measure(*qubits))

print(circuit)

from cirq import Simulator
simulator = Simulator()
result = simulator.run(circuit, repetitions=int(sys.argv[2]))

print(result)
```

Listing 7: grover_cirq.py

6.6.6 Q# test program

```
using System;

using Microsoft.Quantum.Simulation.Core;
using Microsoft.Quantum.Simulation.Simulators;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Grover
{
    class Driver
    {
        public static void Pause()
        {
            System.Console.WriteLine("\n\nPress any key to continue...\n\n");
            System.Console.ReadKey();
        }

        static void Main(string[] args)
        {
            // We begin by defining a quantum simulator to be our target
            // machine.
            var sim = new QuantumSimulator(throwOnReleasingQubitsNotInZeroState: true);

            //repeats: number of program execution
            var repeats = Int32.Parse(args[1]);

            //n: number of qubit
            //N: size of "database"
            var numDatabaseQubits = Int32.Parse(args[0]);
            var databaseSize = Math.Pow(2.0, numDatabaseQubits);

            int markedElement = 28;
            int numMatches = 1;

            //number of iteraion of Grover's algorithm
            var numIteration = Convert.ToInt64((Math.PI / 4) * Math.Sqrt(databaseSize / numMatches));
            System.Console.WriteLine("Number of optimal iterations: " + numIteration + "\n\n");

            foreach (var idxAttempt in Enumerable.Range(0, repeats))
            {
                System.Console.WriteLine("Attempt number " + (idxAttempt+1));
                ApplyGroverSearch.Run(sim, markedElement, numIteration, numDatabaseQubits).Wait();
                System.Console.WriteLine("\n\n");
            }
        }
    }
}
```

Listing 8: Driver.cs

```
namespace Grover
{
    open Microsoft.Quantum.Canon;
    open Microsoft.Quantum.Intrinsic;
        open Microsoft.Quantum.Convert;

    operation HelloQ () : Unit {
        Message("Hello quantum world!");
    }

        operation PrepareBitString(bitstring : Bool[], register : Qubit[]) : Unit
        is Adj + Ctl {

                let nQubits = Length(register);
                for (idxQubit in 0..nQubits - 1) {
                  if (bitstring[idxQubit]) {
                     X(register[idxQubit]);
                        }
                }
        }

        operation RegisterSetup(register : Qubit[]) : Unit {
                let registerDB = new Bool[Length(register)-1];
                let ancilla = [true];
                let registerTemplate = registerDB + ancilla;

                PrepareBitString(registerTemplate, register); // |00..001>

                for (q in register) {
                        H(q);
                }
        }

        operation BooleanOracle (markedElement : Int, register : Qubit[]) : Unit {
                let databaseSize = Length(register)-1;
                let boolMarked = IntAsBoolArray(markedElement, databaseSize);

                for(i in 0 .. databaseSize-1) {
                        if(boolMarked[i] == false) {
                                X(register[databaseSize-i-1]);
                        }
                }

                let controls = register[0 .. Length(register)-2];
                let target = register[Length(register)-1];

                Controlled X(controls, target);
```

```
            for(i in 0 .. databaseSize-1) {
                if(boolMarked[i] == false) {
                    X(register[databaseSize-i-1]);
                }
            }
        }

        operation Grover (registerComplete : Qubit[]) : Unit {
            let register = registerComplete[0 .. Length(registerComplete)-2];

            for(q in register) {
                H(q);
                X(q);
            }

            let controls = register[0 .. Length(register)-2];
            let target = register[Length(register)-1];

            Controlled Z(controls, target);

            for(q in register) {
                X(q);
                H(q);
            }
        }

        operation ApplyGroverSearch (markedElement : Int, nIterations : Int, nDatabaseQubits : Int) : Result
            mutable resultSuccess = Zero;

            using (qubits = Qubit[nDatabaseQubits+1]){
                RegisterSetup(qubits);

                for(i in 1 .. nIterations) {
                    BooleanOracle(markedElement, qubits);
                    Grover(qubits);
                }

                for(i in 0 .. nDatabaseQubits-1) {
                    if (M(qubits[i]) == One) {
                        Message("1");
                    }
                    else {
                        Message("0");
                    }
                }

                ResetAll(qubits);
            }

            return (resultSuccess);
        }
}
```

Listing 9: Grover.qs

6.6.7 Time results fixing shots

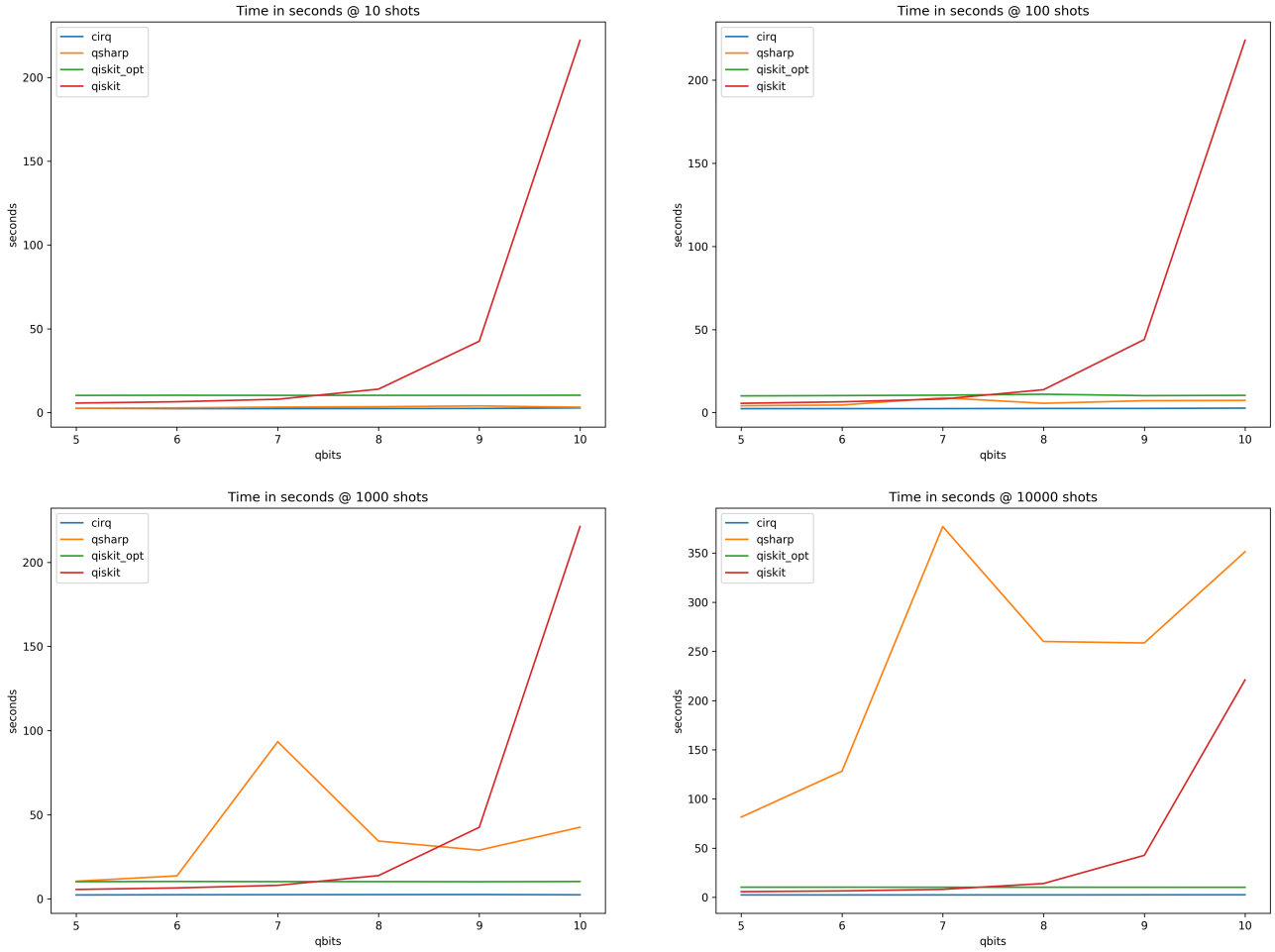


Figure 6.1: Test di tutti i freameworks

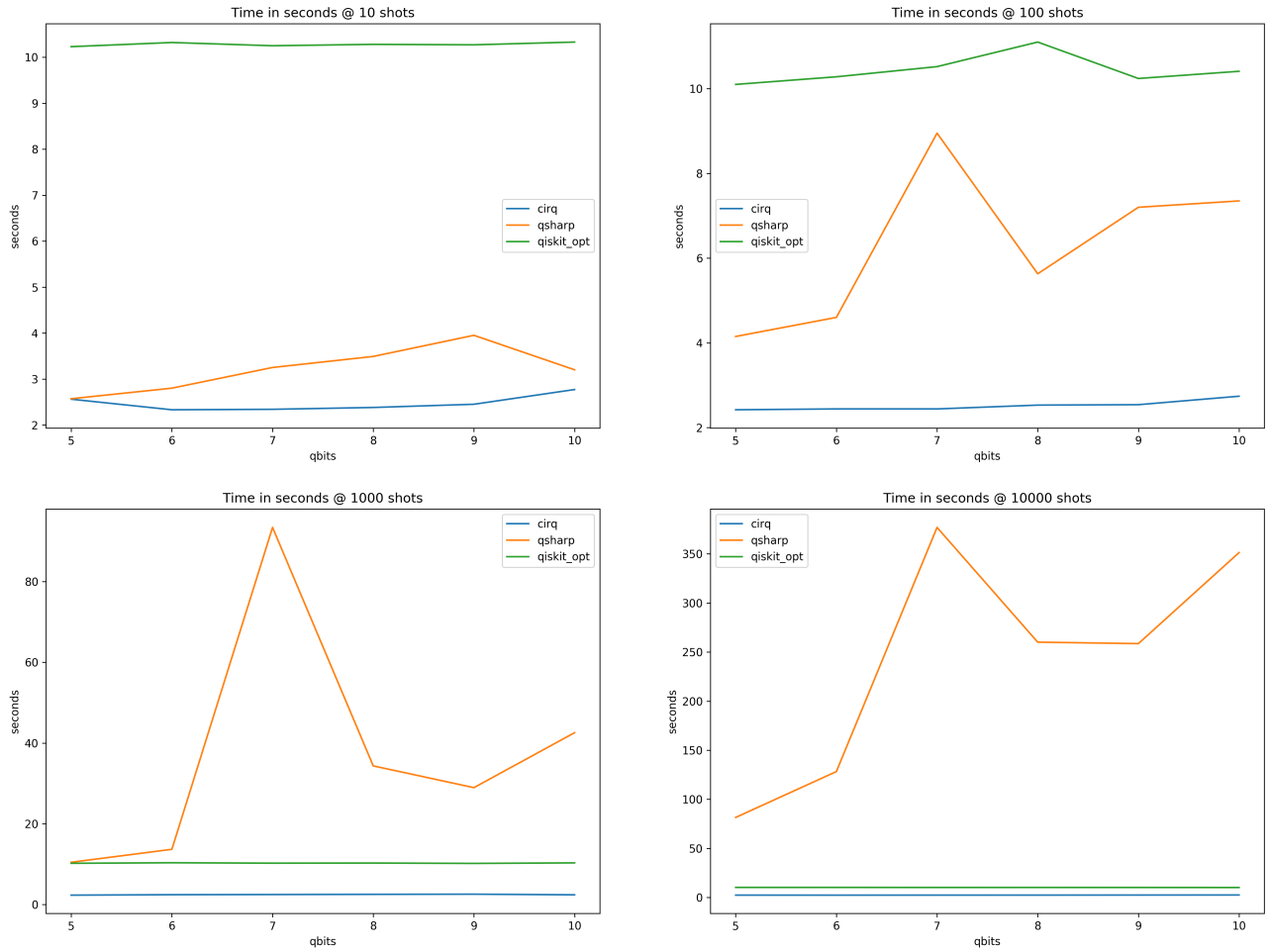


Figure 6.2: Test di tutti i freameworks eccetto qiskit

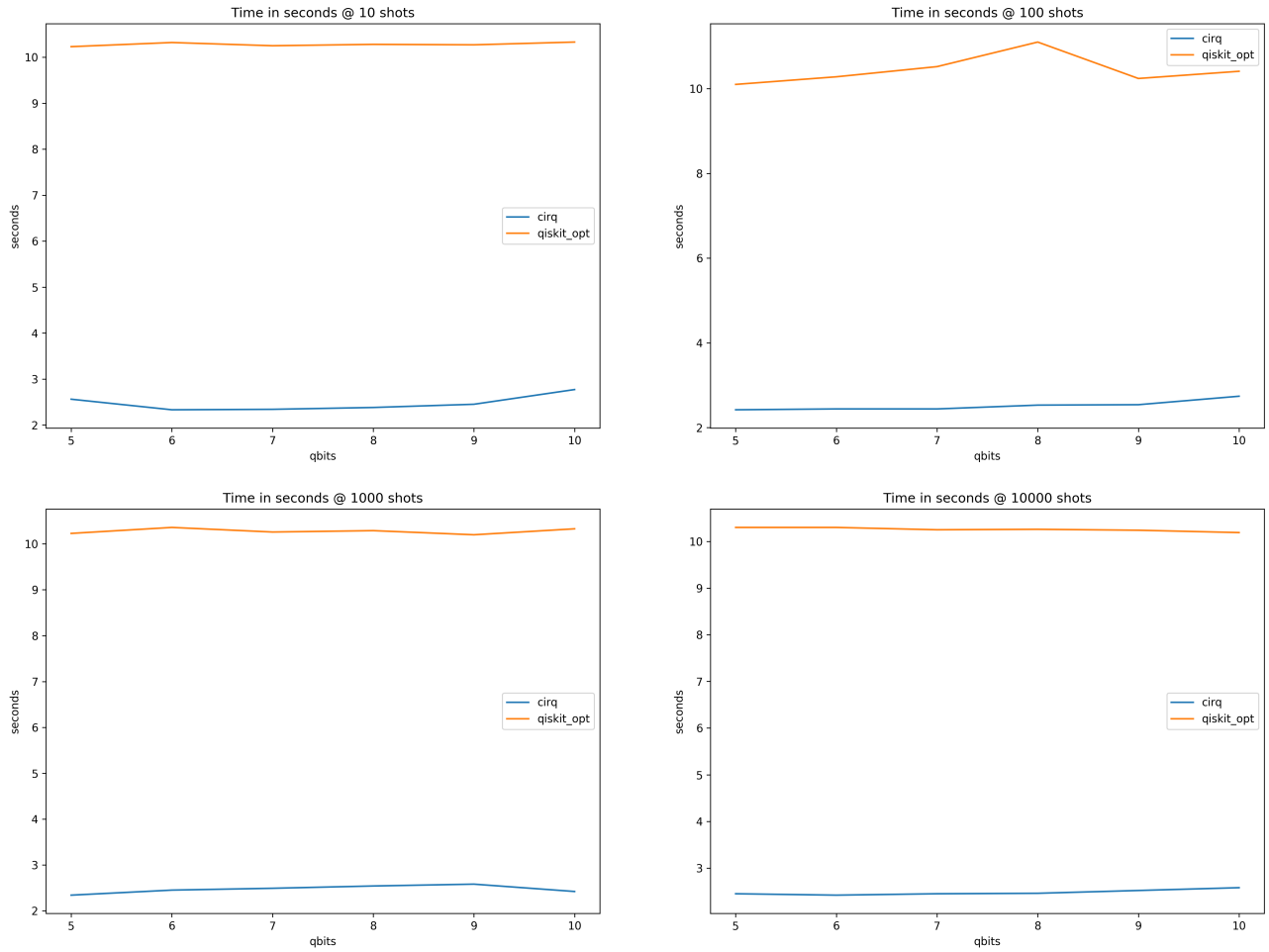


Figure 6.3: Test di tutti i freameworks eccetto i qiskits

6.6.8 Time results fixing the number of qbits

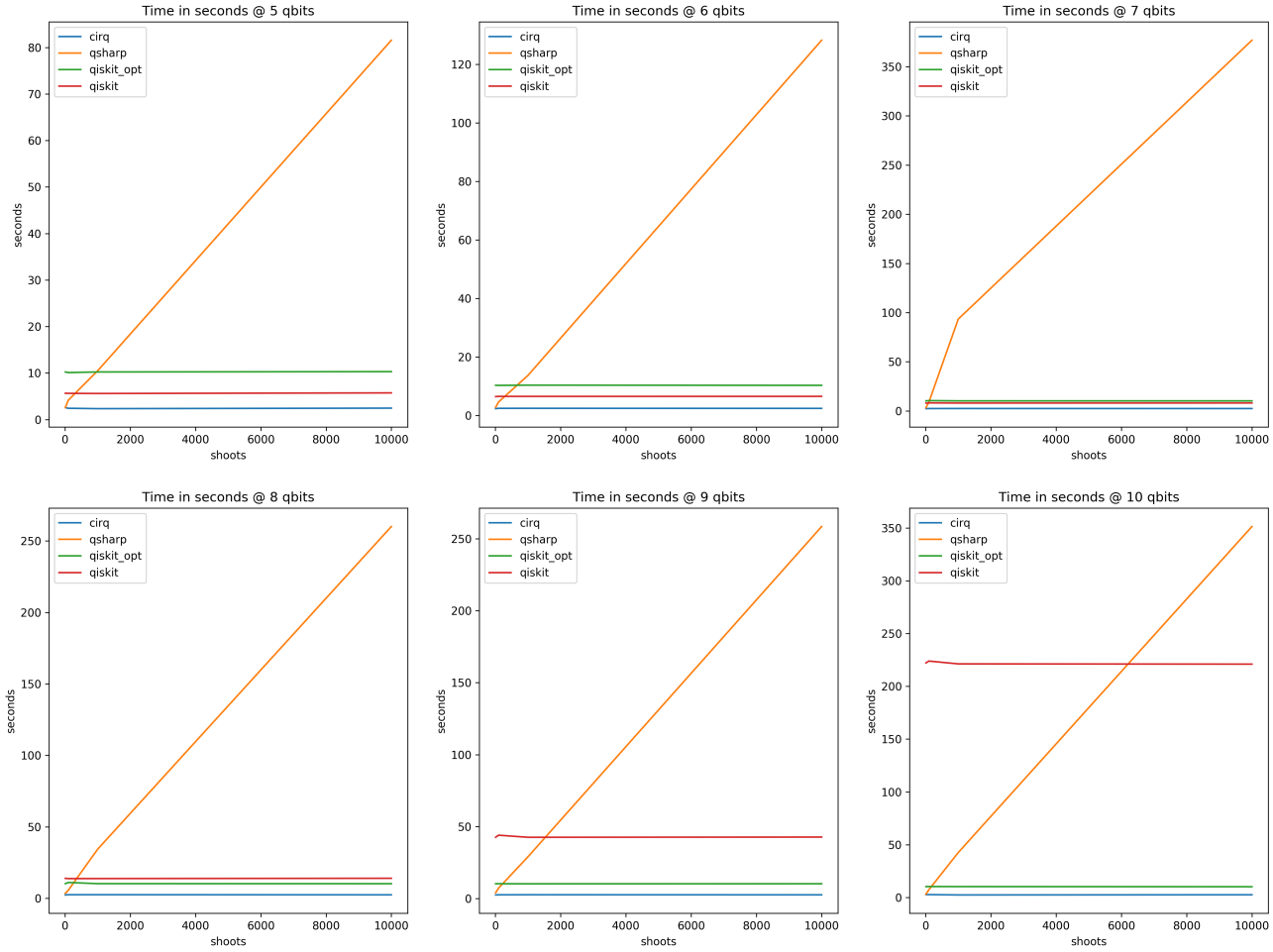


Figure 6.4: Test di tutti i freameworks

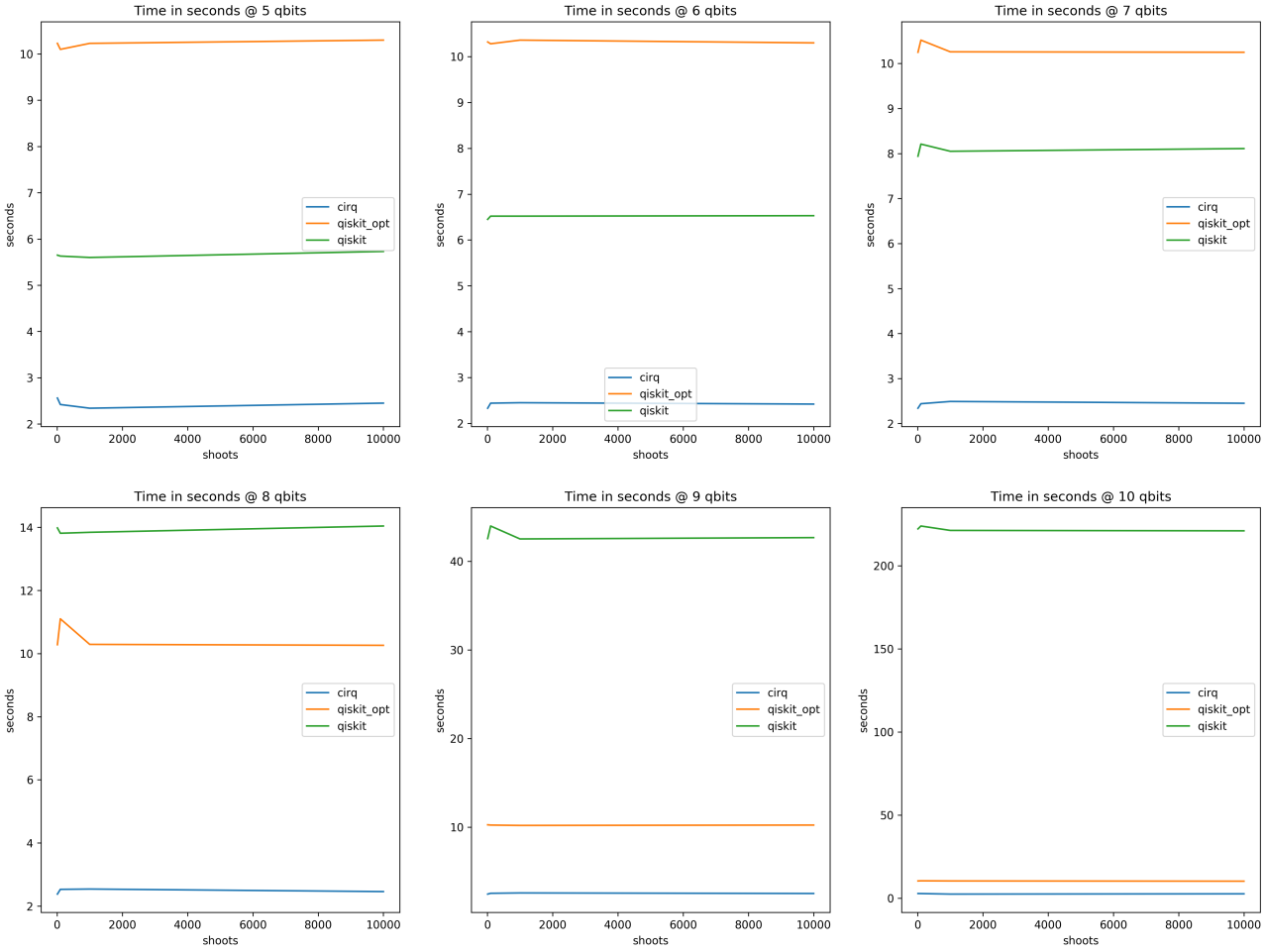


Figure 6.5: Test di tutti i freameworks eccetto Q#

6.6.9 Memory results fixing shots

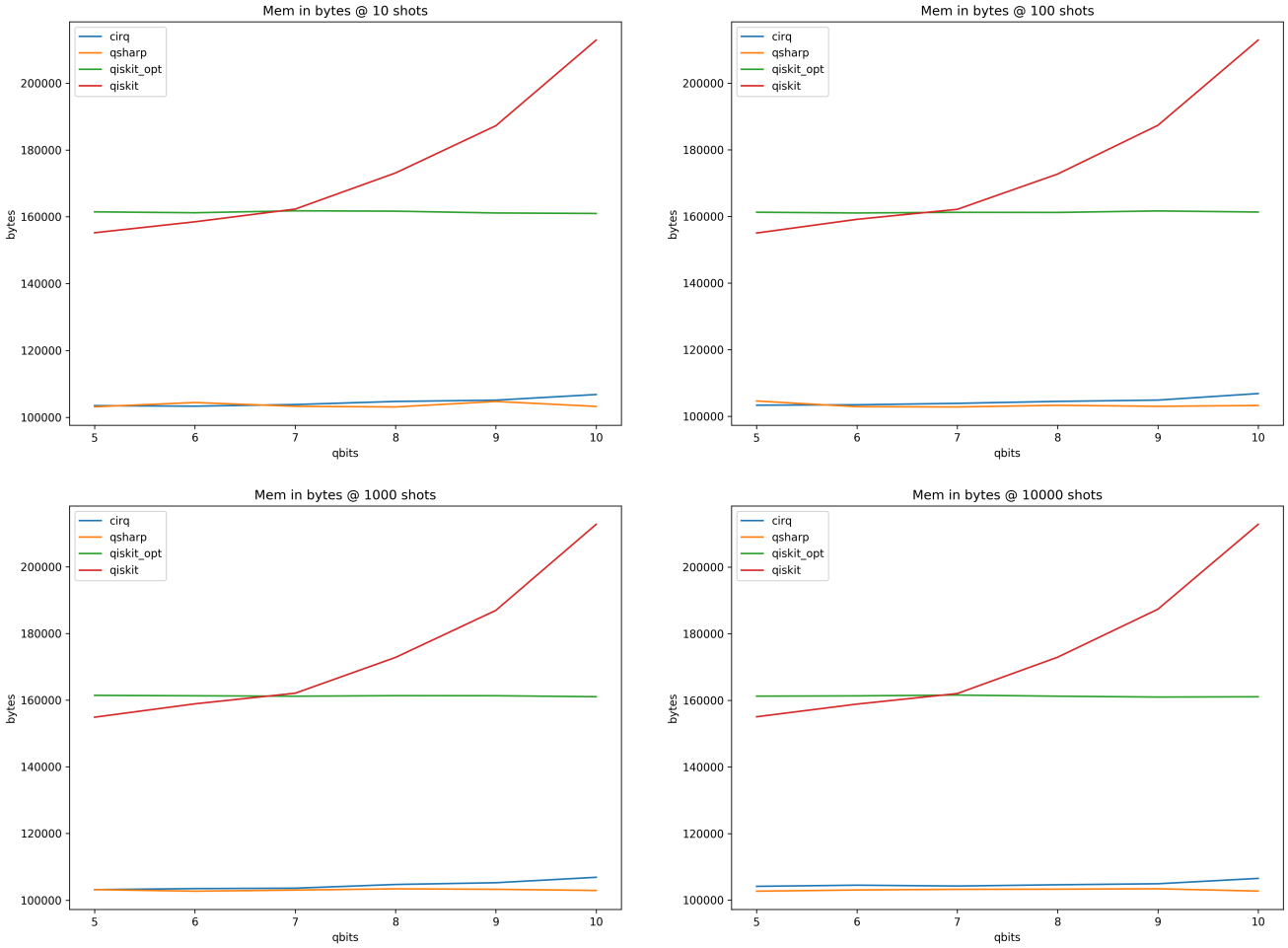


Figure 6.6: Test di tutti i freameworks

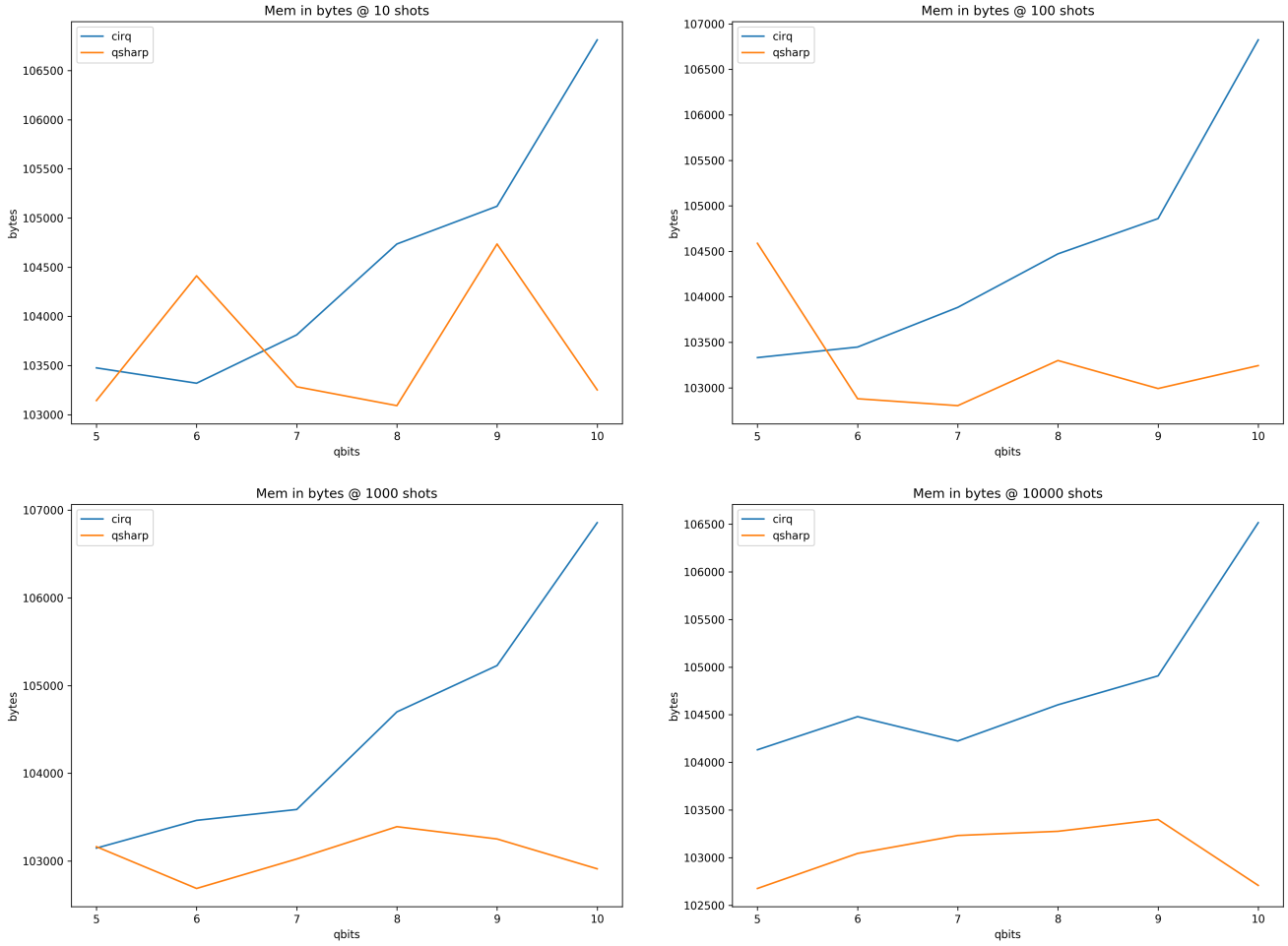


Figure 6.7: Test di tutti i freameworks eccetto i qiskits

6.6.10 Memory results fixing the number of qbits

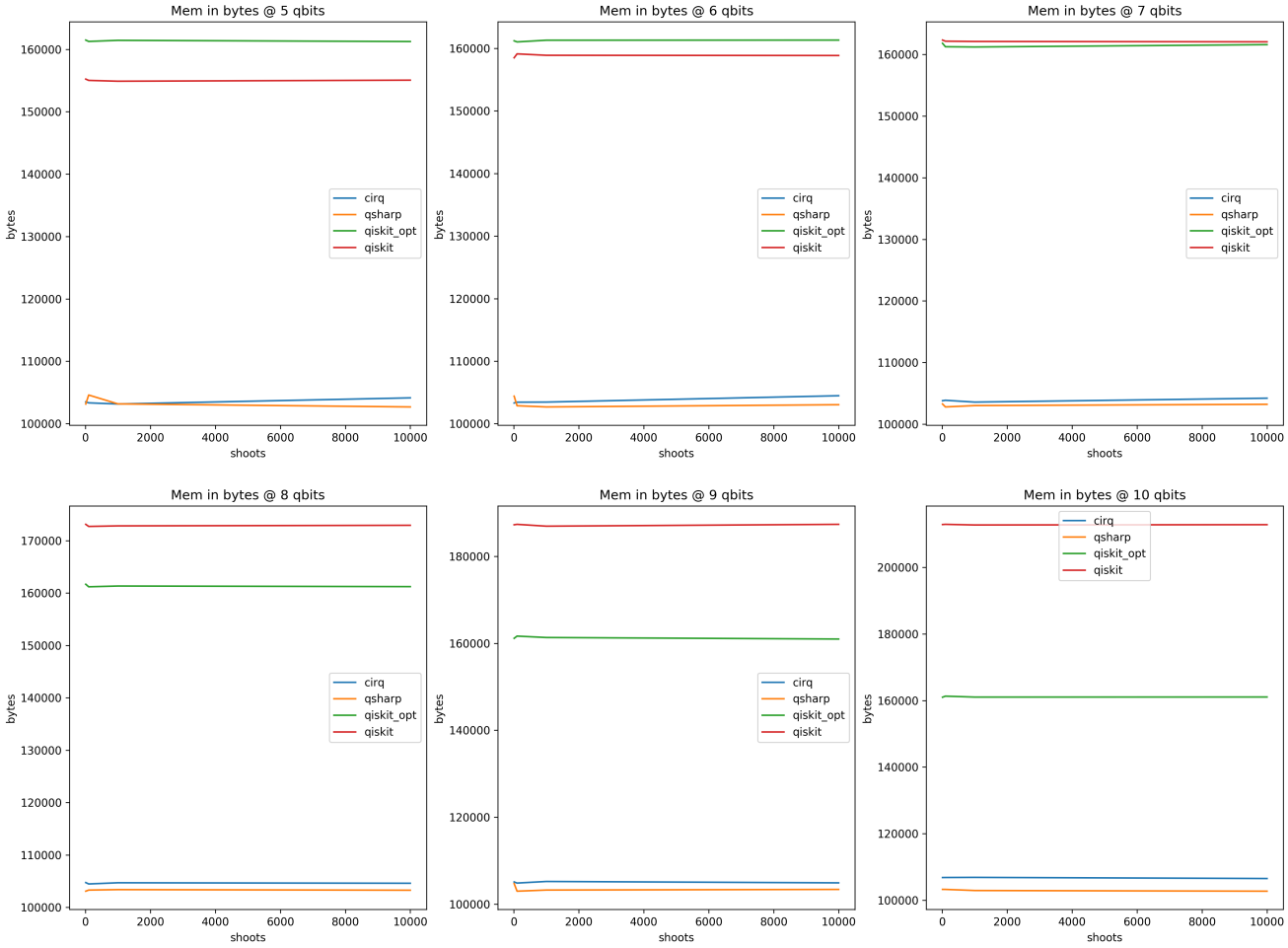


Figure 6.8: Test di tutti i freameworks

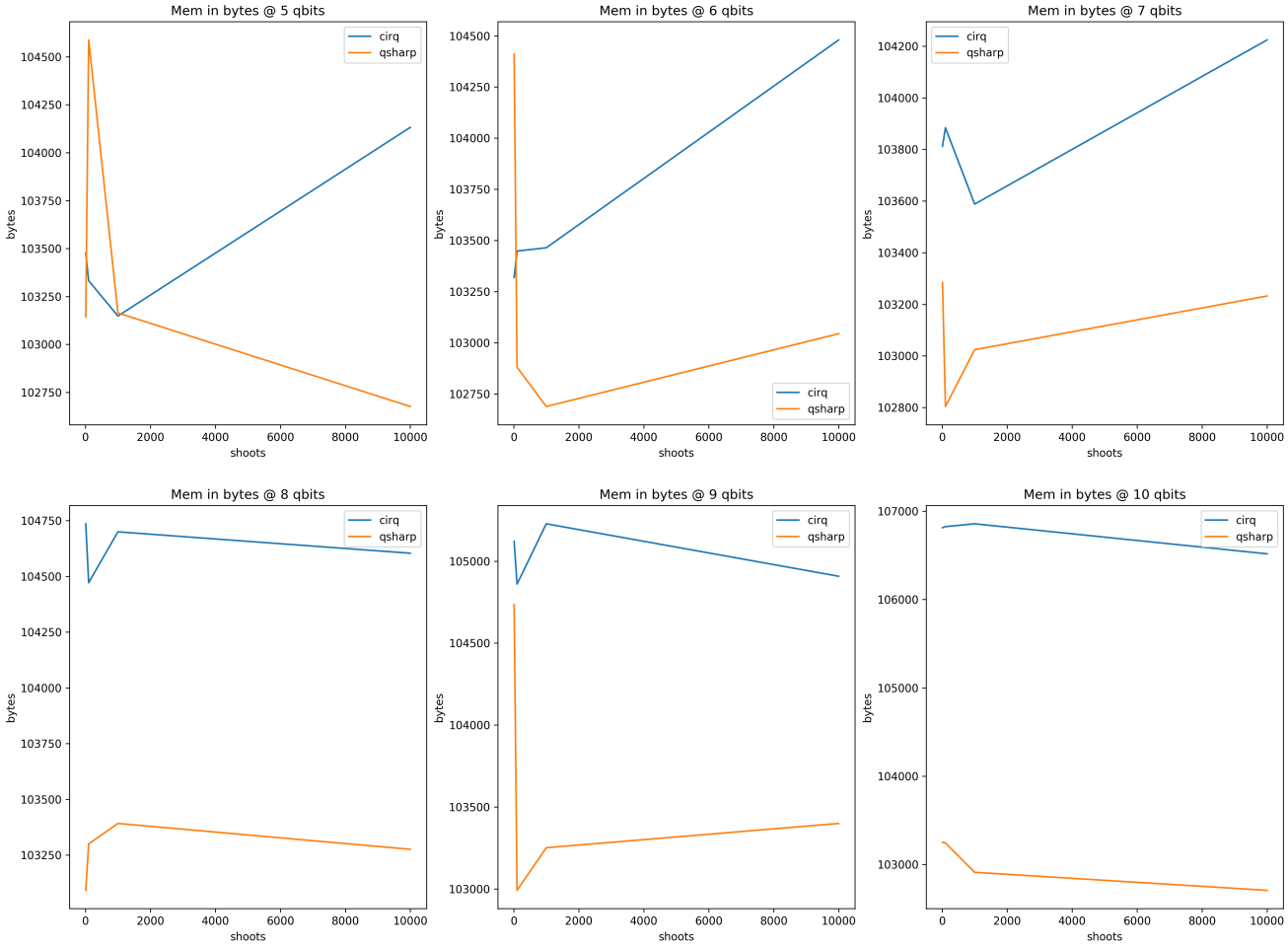


Figure 6.9: Test di tutti i freameworks eccetto i qiskits

Bibliography

[1] Otakar Borůvka. O jistém problému minimálním [about a certain minimal problem]. *Práce Mor. Přírodověd. Spol. V Brn*, III:37–58, 1926.

[2] Vojtěch Jarník. O jistém problému minimálním [about a certain minimal problem]. *Práca Moravské Prírodovedecké Spolecnosti*, 6:57–63, 1930.

[3] Joseph B. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society*, 7(1):48–50, 1956.

[4] David Deutsch and Richard Jozsa. Rapid solution of problems by quantum computation. *Proceedings of the Royal Society of London. Series A: Mathematical and Physical Sciences*, 439(1907):553–558, 1992.

[5] Richard Cleve, Artur Ekert, Chiara Macchiavello, and Michele Mosca. Quantum algorithms revisited. *Proceedings of the Royal Society of London. Series A: Mathematical, Physical and Engineering Sciences*, 454(1969):339–354, 1998.

[6] Lov K. Grover. A fast quantum mechanical algorithm for database search. In *Proceedings of the Twenty-eighth Annual ACM Symposium on Theory of Computing*, STOC '96, pages 212–219, New York, NY, USA, 1996. ACM.

[7] Carlile Lavor, LRU Manssur, and Renato Portugal. Grover's algorithm: quantum database search. *arXiv preprint quant-ph/0301079*, 2003.

[8] George F Viamontes, Igor L Markov, and John P Hayes. Is quantum search practical? *Computing in science & engineering*, 7(3):62–70, 2005.

[9] Gilles Brassard, Peter Hoyer, Michele Mosca, and Alain Tapp. Quantum amplitude amplification and estimation. *Contemporary Mathematics*, 305:53–74, 2002.

[10] Michael A Nielsen and Isaac Chuang. Quantum computation and quantum information, 2002.

[11] John Watrous. Quantum computational complexity. *Encyclopedia of complexity and systems science*, pages 7174–7201, 2009.

[12] Richard Cleve. An introduction to quantum complexity theory. *Collected Papers on Quantum Computation and Quantum Information Theory*, pages 103–127, 2000.

[13] Adriano Barenco, Charles H Bennett, Richard Cleve, David P DiVincenzo, Norman Margolus, Peter Shor, Tycho Sleator, John A Smolin, and Harald Weinfurter. Elementary gates for quantum computation. *Physical review A*, 52(5):3457, 1995.

[14] Christopher M Dawson and Michael A Nielsen. The solovay-kitaev algorithm. *arXiv preprint quant-ph/0505030*, 2005.

[15] Christoph Durr and Peter Hoyer. A quantum algorithm for finding the minimum. *arXiv preprint quant-ph/9607014*, 1996.

[16] Christoph Dürr, Mark Heiligman, Peter HOyer, and Mehdi Mhalla. Quantum query complexity of some graph problems. *SIAM Journal on Computing*, 35(6):1310–1328, 2006.

[17] Gilles Brassard, Peter Høyer, and Alain Tapp. Quantum counting. In *International Colloquium on Automata, Languages, and Programming*, pages 820–831. Springer, 1998.

[18] Michel Boyer, Gilles Brassard, Peter Høyer, and Alain Tapp. Tight bounds on quantum searching. *Fortschritte der Physik: Progress of Physics*, 46(4-5):493–505, 1998.

[19] Vittorio Giovannetti, Seth Lloyd, and Lorenzo Maccone. Quantum random access memory. *Physical review letters*, 100(16):160501, 2008.

[20] Andrew W Cross, Lev S Bishop, John A Smolin, and Jay M Gambetta. Open quantum assembly language. *arXiv preprint arXiv:1707.03429*, 2017.