UCLouvain

École polytechnique de Louvain

Discrete Optimization in Quantum Computing

Authors: **Jean-Baptiste Hontoir, Arnaud Servais**
Supervisor: **Jean-Charles Delvenne**
Readers: **Geovani Nunes Grapiglia, Yves Deville**
Academic year 2023–2024
Master [120] in Data Sciences Engineering
Master [120] in Mathematical Engineering

Acknowledgments

We would like to thank Pr. Jean-Charles Delvenne for his time and advices that guided us all along this year of research and redaction.

We also would like to thank the Professors Geovani Nunes Grapiglia and Yves Deville for accepting reading and assessing our work.

Contents

List of Figures

List of Tables

Chapter 1

Introduction to Quantum Computing

In this chapter, we delve into the fundamentals of Quantum Computing, a field that leverages the principles of quantum mechanics to perform computation. Unlike classical computing, which relies on bits as the smallest unit of data, quantum computing uses quantum bits, or qubits. Qubits have unique properties such as superposition and entanglement, which allow quantum computers to solve certain problems more efficiently than classical computers. It is on these properties and accelerations that we will base our discrete optimization algorithms in later chapters.

But before diving into the heart of our work, let's review the basics of quantum computing, based on [**1**], [**2**] and [**3**].

1.1 Qubits and Quantum States

In classical computing, the smallest element that stores information is the *bit*. This physical element has the ability to be either *ON* or *OFF*. It is also said to store the value 1 or 0.

Quantum computers, meanwhile, have their equivalent: the *quantum bit* or *qubit*. Its special feature is that it can be either in the *ON* state, in the *OFF* state or in a superposition of the *ON* and *OFF* states.

This quantum property, which allows multiple states to be superposed, calls for a special notation to describe a qubit: Dirac notation.

1.1.1 Dirac Notation

In Dirac's convention, the *ON* state is written $|1\rangle$ and the *OFF* state $|0\rangle$. The information contained in a *qubit* $|q\rangle$ is therefore noted:

$$|q\rangle = \alpha\,|0\rangle + \beta\,|1\rangle$$

with the normalization condition:

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

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

You could compare a *qubit* to a coin when you toss it in the air. When the coin is in the air, it is both half in the *tail* state and half in the *head* state. When it falls, the coin is equally likely to be observed in either the *head* or *tail* state.

If we take a specific rigged coin, when it is flipping in the air, it can be considered to have a probability $\frac{1}{3}$ of being in the *head* state and a probability $\frac{2}{3}$ of being in the *tail* state, for example.

It is very similar for a *qubit* which can be in a superposition of $|0\rangle$ and $|1\rangle$ states such that it has a probability $|\alpha|^2 = \frac{1}{3}$ of being observed[1] in the $|0\rangle$ state and a probability $|\beta|^2 = \frac{2}{3}$ of being observed in the $|1\rangle$ state.

If we want to describe a *qubit* in a more formal and mathematical framework, it can be done with a Hilbert space of basis $\{|0\rangle, |1\rangle\}$ such that:

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

Therefore, the information a *qubit* stores can be written as a vector:

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

Moreover, the normalization condition $|\alpha|^2 + |\beta|^2 = 1$ ensures $|q\rangle$ has a norm equal to 1:

$$\| \, |q\rangle \, \|^2 = |\alpha|^2 + |\beta|^2 = 1$$

[1] We will come back to the concept of observation later in the chapter.

This is why a *qubit* can also be seen as a vector whose extremities are the origin and a point on a sphere of radius 1, centered at the origin of this same Hilbert space.

Finally, the last concept to be discussed in this section is the phase of a qubit. Since α and β are complex numbers, they can be written in their polar form that will always be of the form:

$$\alpha = |\alpha|e^{j\theta}$$
$$\beta = |\beta|e^{j(\psi+\theta)}$$

with j being the imaginary unit, $\theta \in [0; 2\pi]$ and $\psi + \theta \in [0; 2\pi]$. The phase of the *qubit* $|q\rangle$ is defined as the common angle θ in the arguments of the polar representations of α and β. Although the phase of a *qubit* has no physically observable consequences, it will be useful in the next chapters.

1.2 Qubits Systems

In a quantum computer program, a qubit is rarely used on its own. To store more information and allow one to create more complex programs, a combination of *qubits* is needed.

A system of n *qubits* is denoted $|q\rangle_n$. This system, also called *quantum register*, can also be described as vector in a Hilbert space of dimension 2^n: $H^{\otimes n}$. A quantum register can in fact represented as the tensor product between every single *qubits* representation it contains.

The tensor product we will use is the following. For two vectors:

$$\alpha = \begin{pmatrix} \alpha_1 \\ \vdots \\ \alpha_n \end{pmatrix}$$
$$\beta = \begin{pmatrix} \beta_1 \\ \vdots \\ \beta_n \end{pmatrix}$$

the tensor product is defined as:

$$\alpha \otimes \beta = \begin{pmatrix} \alpha_1\beta_1 \\ \vdots \\ \alpha_1\beta_m \\ \alpha_2\beta_1 \\ \vdots \\ \alpha_2\beta_m \\ \vdots \\ \alpha_n\beta_1 \\ \vdots \\ \alpha_n\beta_m \end{pmatrix}$$

Let's take the example of a 2-*qubit* register $|q\rangle_2$. Using the previous tensor product definition, the new basis for the Hilbert space in which it can be represented is $\{|00\rangle, |01\rangle, |10\rangle, |11\rangle\}$:

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

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

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

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

The information stored in a register $|q\rangle_2$ can therefore be described as the tensor product of its first *qubit*:

$$|\phi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle$$

with its second *qubit*:

$$|\psi\rangle = \gamma\,|0\rangle + \delta\,|1\rangle$$

Therefore:

$$|q\rangle_2 = |\phi\rangle \otimes |\psi\rangle = \alpha\gamma\,|00\rangle + \alpha\delta\,|01\rangle + \beta\gamma\,|10\rangle + \beta\delta\,|11\rangle = \begin{pmatrix} \alpha\gamma \\ \alpha\delta \\ \beta\gamma \\ \beta\delta \end{pmatrix}$$

Once again, we note that a quantum register of 2 *qubits* can contain the superposition of 4 states at the same time:

- the state *OFF* - *OFF*: given by $|00\rangle$ where both *qubits* store the value 0,
- the state *OFF* - *ON*: given by $|01\rangle$ where the first *qubit* stores 0 and the second 1,
- the state *ON* - *OFF*: given by $|10\rangle$ where the first *qubit* stores 1 and the second 0,
- the state *ON* - *ON*: given by $|11\rangle$ where both *qubits* store the value 1.

Finally, we note that the normalization condition is always verified:

$$\begin{aligned} \| \, |q\rangle_2 \, \|^2 &= |\alpha\gamma|^2 + |\alpha\delta|^2 + |\beta\gamma|^2 + |\beta\delta|^2 \\ &= |\alpha|^2|\gamma|^2 + |\alpha|^2|\delta|^2 + |\beta|^2|\gamma|^2 + |\beta|^2|\delta|^2 \\ &= \left(|\alpha|^2 + |\beta|^2\right)\left(|\gamma|^2 + |\delta|^2\right) \\ &= 1 \end{aligned}$$

The same reasoning can be applied to a register of n *qubits*. Thus, such a register can contain a superposition of up to 2^n different states, represented by the base $\{|0\rangle, |1\rangle\}^{\otimes n}$.

1.2.1 Operations on Registers

As the information stored in a quantum register can be expressed as a vector of a Hilbert space, one can use the usual operations on them, such as the matrix product.

This matrix product has a physical meaning because it represents an operation that modifies the states of a quantum register. As usual, the matrix product has to involve a matrix $U \in \mathcal{H}^{2^n \times 2^n}$ and multiply it with the vector $|q\rangle_n \in \mathcal{H}^{2^n}$.

As we want the output of this product to be a valid superposition verifying the normalization condition $\| \, |q\rangle_n \, \|^2 = 1$, we need U to be unitary. Indeed, unitary matrices are such that:

$$\|U \, |q\rangle_n \, \|^2 = \| \, |q\rangle_n \, \| = 1$$

Moreover, the unitary matrices have the following property:
U is unitary if and only if:

$$U^*U = UU^* = 1$$

which means they are invertible and their inverse is their transpose conjugate matrix. This is an important property as it shows operations on *qubits* have to be invertible.

1.3 Entanglement and Separability

In quantum mechanics, according to the principle of superposition, the same quantum state can have several values for a certain observable quantity (spin, position, momentum, etc.).

In quantum computing, this means that one or more *qubits* can be in several states at once, with an associated probability. These states and probabilities are given by the orthogonal decomposition on the basis of the Hilbert space associated with the quantum register. The states being the vectors of the basis with associated non-zero coefficients, and the probabilities the coefficients themselves.

Sometimes *qubits* interact with each other via the operations applied to them. These interactions can have the consequence of creating an *entanglement* of the *qubits* states. This means that the states of several *qubits* become inseparable from each other.

Mathematically, we can translate this sentence by saying that, an entangled state is a state that cannot be written as the tensor product of single *qubits*. In other words, $|p\rangle_n$ is entangled if:

$$\nexists\, |p_1\rangle_1, \ldots, |p_n\rangle_1 : |p\rangle_n = |p_1\rangle_1 \otimes \ldots \otimes |p_n\rangle_1$$

Here's an example of an entangled state in a 2 *qubits* register:

$$|q\rangle_2 = \frac{1}{\sqrt{2}} (|00\rangle + |11\rangle)$$

Indeed, one can check that there are no values of $\alpha, \beta, \gamma, \delta \in \mathbb{C}$ such that:

$$\begin{aligned} \frac{1}{\sqrt{2}} (|00\rangle + |11\rangle) &= (\alpha |0\rangle + \beta |1\rangle) \otimes (\gamma |0\rangle + \delta |1\rangle) \\ &= \alpha\gamma |00\rangle + \alpha\delta |01\rangle + \beta\gamma |10\rangle + \beta\delta |11\rangle \end{aligned}$$

Conversely, states linked to *qubits* that don't interact with each other are called *separable* states.

Mathematically, we can translate this sentence by saying that, a separable state is a state that can be written as the tensor product of single *qubits*. In other words, $|p\rangle_n$ is separable if:

$$\exists\, |p_1\rangle_1, \ldots, |p_n\rangle_1 : |p\rangle_n = |p_1\rangle_1 \otimes \ldots \otimes |p_n\rangle_1$$

An example of an entangled state in a 2 qubit register:

$$|q\rangle_2 = \frac{1}{2}\left(|00\rangle + |01\rangle + |10\rangle + |11\rangle\right)$$

Indeed, this state can be written as a tensor product of the *qubit* states $\left(\frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|0\rangle\right)$ and $\left(\frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|0\rangle\right)$:

$$\left(\frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle\right) \otimes \left(\frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle\right) = \frac{1}{2}\left(|00\rangle + |01\rangle + |10\rangle + |11\rangle\right)$$

1.4 Qubit Measurement

Thanks to state superpositions, *qubits* allow us to perform computer operations on an exponentially larger number of data items at the same time, which seems providential. Unfortunately, there's a catch. If we try to observe a quantum particle to find out what states it's composed of, it irreversibly collapses into one of its superposition states. It is therefore impossible to know all the states of a register. If we measure a register of *qubits*, we'll only know one of the states they possessed; the one into which the *qubits* collapsed.

For example, if we measure the two *qubits* of the quantum register:

$$|q\rangle_2 = \left(\frac{3}{4}|00\rangle + \frac{1}{4}|01\rangle + \frac{1}{2}|10\rangle + \frac{1}{2\sqrt{2}}|11\rangle\right)$$

we will have:

- a probability of $|\frac{3}{4}|^2 = \frac{9}{16}$ of measuring the state $|00\rangle$
- a probability of $|\frac{1}{4}|^2 = \frac{1}{16}$ of measuring the state $|01\rangle$
- a probability of $|\frac{1}{2}|^2 = \frac{1}{4}$ of measuring the state $|10\rangle$

- a probability of $|\frac{1}{2\sqrt{2}}|^2 = \frac{1}{8}$ of measuring the state $|11\rangle$

To sum up, a measurement of an whole quantum register will return one of the states of the superposition with a probability equal to the square of the absolute value of the associated coefficient.

1.4.1 Measuring a Part of a Register

In some cases, we may wish to measure only certain *qubits* of a quantum register. This works in the same way as above, except that if one of the *qubits* we are measuring is entangled with other *qubits* in the register (not necessarily the ones we're measuring), then all the entangled *qubits* will collapse to one of their states at the same time (and not just the ones we've measured). Therefore a measure of an entangled *qubits* has effects on all the entangled *qubits* and not only the one measured.

For example let's say we have a 2 entangled *qubits* register:

$$|q\rangle_2 = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$$

We wish to measure the first *qubit* only. The measure will have the following effect:

- We will measure the state $|0\rangle$ with probability $\frac{1}{2}$. As the *qubits* are entangled, they will collapse to the state corresponding to the one of the first *qubit* measure: $|00\rangle$
- We will measure the state $|1\rangle$ with probability $\frac{1}{2}$. As the *qubits* are entangled, they will collapse to the state corresponding to the one of the first *qubit* measure: $|11\rangle$

We therefore have to be careful with the way we measure *qubits* as it collapse the superposition and it can have effect on the whole register.

1.5 Quantum Gates and Circuit

Now that we've introduced the essentials for understanding *qubits* and their systems, we can finally get to the heart of quantum computing.

Unlike classical computing, quantum computing programs are coded in the form of line diagrams. Each line corresponds to a *qubit* used in the program, to which gates

are applied. The gates are the unitary operations we mentioned earlier, whose purpose is to transform the states stored in the register. Here's an example of such diagram:

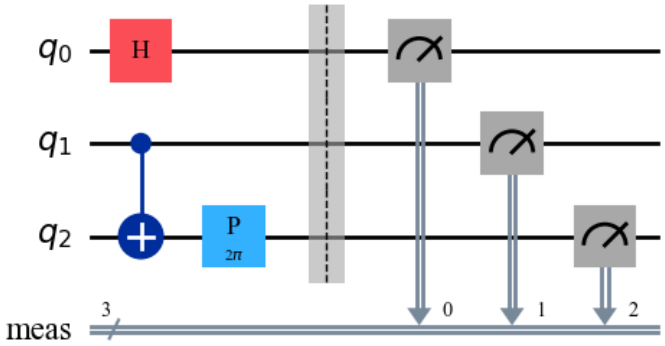

Figure 1.1: Example Circuit of a 3-*qubits* Register

This circuit represents 3 *qubits*: q_0, q_1 and q_2, each with a black instruction line. Each of these *qubits* undergoes unitary operations called gates [2] located on each line in a given order. After the vertical dotted line, the 3 *qubits* are measured and the result stored in a conventional register represented by the double *meas* line.

This diagram was created from *Qiskit*. *Qiskit* is an open-source quantum computing software development framework created by IBM. It provides tools for working with quantum computers at multiple levels of programming and research, enabling users to develop and run quantum algorithms on both simulators and real quantum hardware.

For this thesis, all of our *qubits* simulations and code will be done using *Qiskit*.

Moreover, sometimes, we will say that we ran a circuit. It means that we executed the circuit with the help of the *Qiskit qubits* simulator.

We will now introduce the few gates that will be useful in our programs.

[2] We'll present the most important ones in this section.

1.5.1 X gate

The X gate corresponds to a *qubit* state flip. It has the following effect on the possible states of a *qubit*:

$$|0\rangle \longrightarrow |1\rangle$$
$$|1\rangle \longrightarrow |0\rangle$$

The matrix representation of this gate is the following:

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

Indeed, we see that the matrix product has the desired effect:

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

Obviously, the inverse transformation of the X gate is the X gate itself. Indeed, flipping the value of *qubit* twice in a row returns it to its original value.

This postulate can be verified in matrix form:

$$XX^* = X^*X = XX = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} = I$$

Finally, below is the *Qiskit* diagram of an X gate:

Figure 1.2: X Gate *Qiskit* Diagram

1.5.2 H gate

The H gate is a gate that creates an equally distributed superpositon with a *qubit*. It has the following effect on the possible states of a *qubit*:

$$|0\rangle \longrightarrow \frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle$$
$$|1\rangle \longrightarrow \frac{1}{\sqrt{2}}|0\rangle - \frac{1}{\sqrt{2}}|1\rangle$$

The "$-$" in $-\frac{1}{\sqrt{2}}|1\rangle$ shows that the phase of this state is flipped ($\theta = -180°$).

The matrix representation of this gate is the following:

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

Indeed, we see that the matrix product has the desired effect:

$$H|0\rangle = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{pmatrix} = \frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle$$
$$H|1\rangle = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} \frac{1}{\sqrt{2}} \\ -\frac{1}{\sqrt{2}} \end{pmatrix} = \frac{1}{\sqrt{2}}|0\rangle - \frac{1}{\sqrt{2}}|1\rangle$$

Again in this case, the H gate is its self inverse. Indeed, it can be verified in matrix form:

$$HH^* = H^*H = HH = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} = \frac{1}{2} \begin{pmatrix} 2 & 0 \\ 0 & 2 \end{pmatrix} = I$$

Below is the *Qiskit* diagram of an H gate:

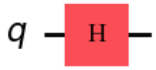

Figure 1.3: H Gate *Qiskit* Diagram

1.5.3 Z gate

The Z gate corresponds to a *qubit* phase flip for each *ON* states. In other words, the phase of the *ON* state of a qubit undergoes an increase of 180°. It has the following effect on the possible states of a *qubit*:

$$|0\rangle \longrightarrow |0\rangle$$
$$|1\rangle \longrightarrow -|1\rangle$$

The matrix representation of this gate is the following:

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

Indeed, we see that the matrix product has the desired effect:

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

Clearly, the inverse transformation of the Z gate is the Z gate itself. Indeed, reversing the phase of a state twice in a row is equivalent to not acting on it at all. This postulate can be verified in matrix form:

$$ZZ^* = Z^*Z = ZZ = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} = I$$

Finally, below is the *Qiskit* diagram of a Z gate:

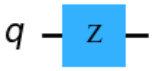

Figure 1.4: Z Gate *Qiskit* Diagram

It is also important to add that if we apply a Z gate to one of the *qubits* of a state, then the gate acts on the integrality of the state. For example, if we apply a Z gate on the first *qubit* of those states, we will get:

$$|01011\rangle \longrightarrow |01011\rangle$$
$$|11011\rangle \longrightarrow -|11011\rangle$$

1.5.4 P gate

The P gate (or Phase gate) is a generalization of the Z gate. This gate adds an angle θ (in rad.) to the phase of a state if it is *ON*. It has the following effect on the possible states of a *qubit*:

$$|0\rangle \longrightarrow |0\rangle$$
$$|1\rangle \longrightarrow e^{j\theta}|1\rangle$$

with j being the imaginary unit. We can definitely see that the Z gate is a special case of P gate where $\theta = \pi$.

The matrix representation of this gate is the following:

$$P(\theta) = \begin{pmatrix} 1 & 0 \\ 0 & e^{j\theta} \end{pmatrix}$$

Indeed, we see that the matrix product has the desired effect:

$$P(\theta)\,|0\rangle = \begin{pmatrix} 1 & 0 \\ 0 & e^{j\theta} \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix} = |0\rangle$$
$$P(\theta)\,|1\rangle = \begin{pmatrix} 1 & 0 \\ 0 & e^{j\theta} \end{pmatrix} \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ e^{j\theta} \end{pmatrix} = e^{j\theta}\,|1\rangle$$

Clearly, the inverse transformation of the Z gate is the Z gate itself. Indeed, reversing the phase of a state twice in a row is equivalent to not acting on it at all.

This postulate can be verified in matrix form:

$$P(\theta)P(\theta)^* = P(\theta)P(-\theta) = \begin{pmatrix} 1 & 0 \\ 0 & e^{j\theta} \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & e^{-j\theta} \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & e^{0} \end{pmatrix} = I$$

$$P(\theta)^*P(\theta) = P(-\theta)P(\theta) = \begin{pmatrix} 1 & 0 \\ 0 & e^{-j\theta} \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & e^{j\theta} \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & e^{0} \end{pmatrix} = I$$

For example, below is the *Qiskit* diagram of an $P(\theta = \pi)$ gate:

Figure 1.5: $P(\theta = \pi)$ Gate *Qiskit* Diagram

It is also important to add that if we apply a P gate to one of the *qubits* of a state, then the gate acts on the integrality of the state. For example, if we apply a P gate on the first *qubit* of those states, we will get:

$$|01011\rangle \longrightarrow |01011\rangle$$
$$|11011\rangle \longrightarrow e^{j\theta}\,|11011\rangle$$

1.5.5 Controlled gate

A controlled gate is a gate that acts on several *qubits*. It operates on the state of a target *qubit* if and only if the value of a control qubit in that same state is 1.

Let us take the example of a controlled X gate C_1X_2 on a 2 *qubits* register. The 1 in the C_1 index indicates that the control *qubit* is the first *qubit* in the register. The 2 in the X_2 index indicates that the target *qubit* is the second *qubit* in the register.

Therefore a C_1X_2 gate has the following effects on the possible states of a 2 *qubits* register:

$$\begin{aligned}|00\rangle &\longrightarrow |00\rangle \\ |01\rangle &\longrightarrow |01\rangle \\ |10\rangle &\longrightarrow |11\rangle \\ |11\rangle &\longrightarrow |10\rangle\end{aligned}$$

One can verify the matrix representation of such a gate is:

$$C_1X_2 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

Another example of a controlled gate is the controlled Z gate C_2Z_1. Notice how we swapped the control and the target *qubit* this time.

As we would expect the C_2Z_1 gate has the following effects on the possible states of a 2 *qubits* register:

$$\begin{aligned}|00\rangle &\longrightarrow |00\rangle \\ |01\rangle &\longrightarrow |01\rangle \\ |10\rangle &\longrightarrow |10\rangle \\ |11\rangle &\longrightarrow -|11\rangle\end{aligned}$$

Again, one can verify the matrix representation of such a gate is:

$$C_2Z_1 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{pmatrix}$$

For information, below are the *Qiskit* diagrams of, respectively, a C_1X_2 gate, a C_2Z_A gate and a $C_1P_2(\theta)$ gate on the same register:

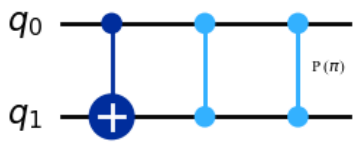

Figure 1.6: Controlled Gates *Qiskit* Diagram

We can also use multi controlled gates. They work in exactly the same way, except that there are now a greater number of control *qubits*. For example, the gate $C_{1,2}X_3$, performs an operation X on a state of the third *qubit* of the register, if and only if the first and second *qubits* of the register are *ON* in the same state.

Therefore, a $C_{1,2}X_3$ has the following effects on the possible states of a 3 *qubits* register:

$$
\begin{aligned}
|000\rangle &\longrightarrow |000\rangle \\
|001\rangle &\longrightarrow |001\rangle \\
|010\rangle &\longrightarrow |010\rangle \\
|011\rangle &\longrightarrow |011\rangle \\
|100\rangle &\longrightarrow |100\rangle \\
|101\rangle &\longrightarrow |101\rangle \\
|110\rangle &\longrightarrow |111\rangle \\
|111\rangle &\longrightarrow |110\rangle
\end{aligned}
$$

Below is the *Qiskit* diagram of this $C_{1,2}X_3$ gate:

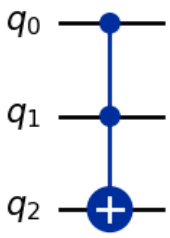

Figure 1.7: $C_{1,2}X_3$ Gate *Qiskit* Diagram

1.5.6 Custom Gates

As one can imagine, increasingly complex circuits can be created with the various gates we have introduced. It's even possible to create customized gates as a combi-

nation of existing gates. To do this, we introduce symbolic notation.

A new gate can be defined as a unitary transformation product. For example, the circuit shown in figure 1.1 (before the measurement), can be renamed G and used as a gate un another bigger circuit. The symbolic notation we will use to define G is the following:

$$G = (I \otimes I \otimes P(2\pi))(H \otimes C_2X_3)$$

with I being the identity gate. This gate is the empty gate has it has no effect on the *qubits* states. Its matrix representation is the identity matrix.

Note that, in the symbolic representation, the order of the circuit gates is read from right to left. This is necessary to preserve the matrix representation and the matrix product. Indeed, the matrix operator representing the circuit's action on a quantum register $|q\rangle_3$ is given by exactly the same expression $G = (I \otimes I \otimes P(2\pi)(H \otimes C_2X_3)$.

To compress the notation even more by getting rid of the I gates, we can index the target qubit of a gate to implicitly express the "$\otimes$" and the "I". G becomes:

$$G = (H_1C_2X_3)P(2\pi)_3$$

Lastly, this same G gate can again be used in another circuit in the same way as the previous ones. If used in a register containing more than 3 *qubits*, the index will indicate the starting *qubit* where G is applied, and the exponent the ending *qubit*. For example, the circuit:

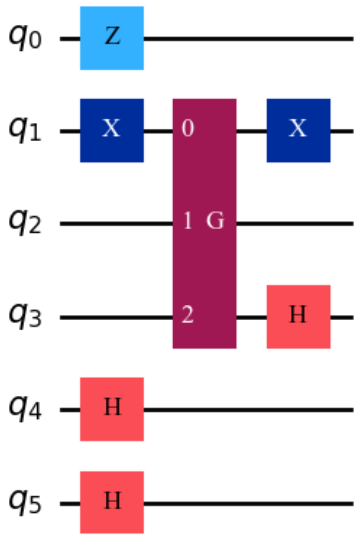

Figure 1.8: G Custom Gate in a 6-*Qubits* Register

is denoted as:

$$\text{Circuit } = (X_2 H_4) G_2^4 (Z_1 X_2 H_5 H_6)$$

1.5.7 Inverse of a Circuit

The concept of a circuit and its associated matrix representation calls for the notion of the inverse of a circuit.

Obtaining the inverse of a circuit involves results from the theory of unitary matrices. First, we note that since all circuits and gates are obtained by unitary transformations, they are also unitary circuit and gates.

Let's consider the gate U constituted by 2 other unitary gates U_1 and U_2:

$$U = U_2 U_1$$

Then, U is also unitary. Indeed:

$$\begin{aligned} UU^* &= (U_2 U_1)(U_2 U_1)^* \\ &= U_2 U_1 U_1^* U_2^* \\ &= U_2 I U_2^* \\ &= U_2 U_2^* \\ &= I \end{aligned}$$

and:

$$\begin{aligned} U^* U &= (U_2 U_1)^* (U_2 U_1) \\ &= U_1^* U_2^* U_2 U_1 \\ &= U_1^* I U_1 \\ &= U_1^* U_1 \\ &= I \end{aligned}$$

Since every circuit is also a unitary transformation, its inverse circuit is given by the circuit corresponding to the conjugate transpose of the cicuit matrix.
Taking the G gate as an example, its inverse G^{-1} is given by:

$$\begin{aligned} G^{-1} &= G^* \\ &= ((I \otimes I \otimes P(2\pi))(H \otimes C_2 X_3))^* \\ &= (H \otimes C_2 X_3)^* (I \otimes I \otimes P(2\pi))^* \\ &= (H^* \otimes C_2 X_3^*)(I^* \otimes I^* \otimes P(2\pi)^*) \\ &= (H \otimes C_2 X_3)(I \otimes I \otimes P(-2\pi)) \end{aligned}$$

There are many lessons to be learned from this example. It shows that physically, the inverse of a circuit is the same circuit traversed in the opposite direction with the inverse gates. Here's a diagram of the inverse of the circuit shown in figure 1.1 as an example:

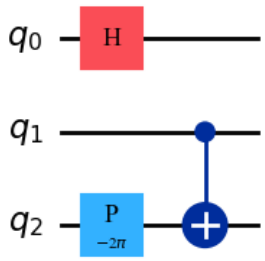

Figure 1.9: Inverse of the Example 3-*Qubits* Circuit

1.6 Binary Representation

Quantum computing, unlike classical computing, doesn't yet have sophisticated languages with routines for handling real numbers. As we have seen, *Qiskit* only allows one to code operations on *qubits* and not on *float* directly. However, as we'll be tackling optimization on real numbers, we introduce a binary notation that represents the reals via *qubits* states in this section.

1.6.1 Integer Representation

Every integer i can be written as an unique binary sequence of length n: $(b_0, b_1, ..., b_{n-1})$ such that:

$$i = \sum_{i=0}^{n-1} 2^i b_{n-1-i}$$

For example, let's have a look at the binary representation of the number 41 in binary:

$$\begin{aligned} 41 &= 1 \times 32 + 0 \times 16 + 1 \times 8 + 0 \times 4 + 0 \times 2 + 1 \times 1 \\ &= 1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 \\ 41 &= 101001_2 \end{aligned}$$

1.6.2 Positive Real numbers Representation

For the representation of positive real numbers, we have chosen a rather simple form that will make operations on real numbers simple in the algorithms.

The binary sequence then consists of two parts: A first sequence of n *bits* that represents the integer part of the number and then, an m *bits* representing the decimal part. This representation is very similar to the integer representation, except that we also use m negative powers of 2 for the decimal part. Therefore, a positive real number r can be written as an unique binary sequence of length $n+m$: $(b_0, b_1, ..., b_{n+m-1})$ such that:

$$r = \sum_{i=-m}^{m+n-1} 2^i b_{n-1-i}$$

For example, let's see the binary representation of 3.375 with $n = 2$ and $m = 3$:

$$\begin{aligned} 3.375 &= 3.375 \\ &= 1 \times 2 + 1 \times 1 + 0 \times 0.5 + 1 \times 0.25 + 1 \times 0.125 \\ &= 1 \times 2^1 + 1 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2} + 1 \times 2^{-3} \\ &= 11011_2 \\ 3.375 &= 11.011_2 \end{aligned}$$

Not all decimal numbers can be exactly represented with a limited number of *bits* for the decimal part. However, it is possible to associate each number with its closest decimal representation (given m). By the closest, we mean we will round the number to the nearest binary number of our $m+n$ *bits* representation.

For example, let's find the binary approximation of 2.888.

$$\begin{aligned} 2.888 &\approx 2.875_{m=3} \\ &= 1 \times 2^1 + 0 \times 2^0 + 1 \times 2^{-1} + 1 \times 2^{-2} + 1 \times 2^{-3} \\ &= 10.111_2 \\ 2.888 &\approx 10.111_2 \end{aligned}$$

The great advantage of this representation is that binary addition of real numbers is exactly the same as classical binary addition:

$$\begin{aligned} 1.500 + 1.875 &= 01.100_2 + 01.111_2 \\ 3.375 &= 11.011_2 \end{aligned}$$

1.6.3 Real numbers Representation

Finally, we can look at the representation that includes negative numbers. In order to preserve the usual operation of addition on real numbers, we use the negative

representation of *two's complement*.

This representation requires the addition of an extra *bit* in the first place of the sequence. The number is translated in the same way as before, with the difference that now you need to substract 2^n in your number computation if the first *bit* has the value of 1.

Therefore, a real number r can be written as an unique binary sequence of length $n+m+1$: $(s, b_0, b_1, ..., b_{n+m-1})$ such that:

$$r = -2^n s + \sum_{i=-m}^{m+n-1} 2^i b_{n-1-i}$$

For example, the 6-*bits* representation of -3.375 with $n = 2$ and $m = 3$ is the following:

$$\begin{aligned}
-3.375 &= -3.375 \\
&= 0 \times 2 + 0 \times 1 + 1 \times 0.5 + 0 \times 0.25 + 1 \times 0.125 - 1 \times 4 \\
&= 0 \times 2^1 + 0 \times 2^0 + 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} - 1 \times 2^2 \\
&= 100.101_2 \\
-3.375 &= 100.101_2
\end{aligned}$$

We can also use a simple example to check that adding real numbers (positive and/or negative) still works:

$$\begin{aligned}
1.500 + (-1.875) &= 001.100_2 + 110.001_2 \\
-0.375 &= 111.101_2
\end{aligned}$$

Finally, we'll note that such an $m+n+1$ *bits* representation can be used to describe real numbers x between:

$$-2^{n-1} \leq x < 2^{n-1}$$

and rounded to the nearest binary number with a maximal error margin of $\epsilon = (\frac{1}{2})^{m+1}$.

1.6.4 Binary Qubits Notation

Now that we've introduced binary notations for reals and integers, we can finally create programs that act on registers of *qubits* representing integers or real numbers.

This is why, in order to get to the heart of the matter, we introduce in this last subsection some practical notations that we will use in the theoretical development

of our algorithms.

When one want to mention a state of 6 *qubits* representing the integer 53 in binary form, it's impractical to write it down:

$$|110101\rangle$$

Instead, we will represent it in decimal form:

$$|53\rangle_6$$

In a similar way, to represent the uniform superposition of states $|s\rangle_n$ we obtain when we apply H gates to each *qubit* of a n *qubits* register, we write:

$$|s\rangle_n = \frac{1}{\sqrt{2^n}} \sum_{k=1}^{2^n-1} |k\rangle_6$$

In this context, $|k\rangle_n$ represent the n *qubits* state corresponding to the binary form of the integer k.

In the same way, we'll simplify our expressions of for the states representing real numbers. Instead of writing the state representing 3.125 as:

$$|011.001\rangle$$

we will simply write:

$$|3.125\rangle_6$$

That's also why one will encounter in the next chapters some notations as:

$$|F(q)\rangle_m$$

where $F(q)$ is a real valued function of argument q. $|F(q)\rangle_m$ will correspond to the state of the m *qubits* binary sequence associated with the real value of $F(q)$.

Chapter 2

Useful Quantum Algorithms

Now that we've familiarized ourselves with the mathematical fundamentals of quantum algorithms, let's take a look at some of the methods that will be particularly useful in the development of our optimization algorithms.

2.1 Quantum Fourier Transform

The first method to be introduced in this chapter is the quantum equivalent of the classical discrete Fourier transform. Given a superposition of binary sequences formed by different *qubits*, the quantum Fourier transform associates another superposition of states with a phase that contains the input informations. Although the connection with our thesis topic is not very obvious, this algorithm thoroughly explained in [**4**] has surprising applications in quantum physics, particularly in the creation of a quantum state superposition, as we will see in the next section.

As a reminder, the discrete Fourier transform takes as input a vector $x \in \mathbb{C}^n$ and maps it to the output vector $y \in \mathbb{C}^n$ following the well-know formula:

$$y_k = \frac{1}{\sqrt{n}} \sum_{i=0}^{n-1} x_i e^{\frac{-j2\pi(i)k}{n}} \quad \forall k \in \{1, \ldots, n\}$$

with j being the imaginary unit.
Very similarly, the Quantum Fourier transform takes as input the state superposition of a 0-phase quantum register $|x\rangle_n$ and transform the superposed *qubit* values in it in the following way:

$$|x\rangle_n \xrightarrow{QFT} \frac{1}{\sqrt{2^n}} \sum_{k=0}^{2^n-1} |k\rangle_n \, e^{\frac{j2\pi kx}{2^n}}$$

In this context, x is the integer represented by the binary sequence of the register $|x\rangle_n$. Therefore, a quantum register $|x\rangle_n$ upon which we apply the QFT, follows

the following mapping:

$$|x_1, \ldots, x_n\rangle \longrightarrow \frac{\left(|0\rangle + e^{j2\pi \frac{x_n}{2}} |1\rangle\right) \otimes \left(|0\rangle + e^{j2\pi\left(\frac{x_{n-1}}{2} + \frac{x_n}{4}\right)} |1\rangle\right) \cdots \left(|0\rangle + e^{j2\pi\left(\frac{x_1}{2} + \frac{x_2}{4} + \cdots + \frac{x_n}{2^n}\right)} |1\rangle\right)}{2^{n/2}}$$

The quantum circuit needed to perform the QFT as explained above is the following:

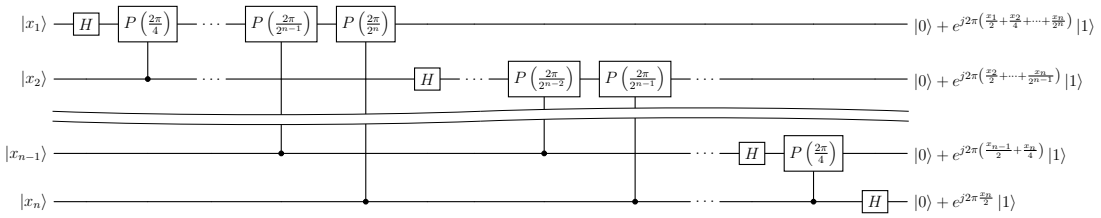

Figure 2.1: QFT Circuit

2.1.1 Inverse Quantum Fourier Transform

Now, we would like to introduced the inverse operation of the Quantum Fourier Transform: the QFT*. This is actually the a method we will use in our algorithms described in Chapter 3 and 4. The inverse QFT is an operation that takes a superposition of states as input and returns a quantification of the states' phase, in the same quantum register. In other words it acts like:

$$\frac{\left(|0\rangle + e^{j2\pi \frac{x_n}{2}} |1\rangle\right) \otimes \left(|0\rangle + e^{j2\pi\left(\frac{x_{n-1}}{2} + \frac{x_n}{4}\right)} |1\rangle\right) \cdots \left(|0\rangle + e^{j2\pi\left(\frac{x_1}{2} + \frac{x_2}{4} + \cdots + \frac{x_n}{2^n}\right)} |1\rangle\right)}{2^{n/2}}$$

$$\longrightarrow |x_1, \ldots, x_n\rangle$$

In other words, it is exactly the inverted mapping of the QFT:

$$\frac{1}{\sqrt{2^n}} \sum_{k=0}^{2^n-1} |k\rangle_n \, e^{\frac{j2\pi kx}{2^n}} \xrightarrow{QFT^*} |x\rangle_n$$

Therefore, its associated circuit is simply the QFT circuit in reverse and with the inverse gates:

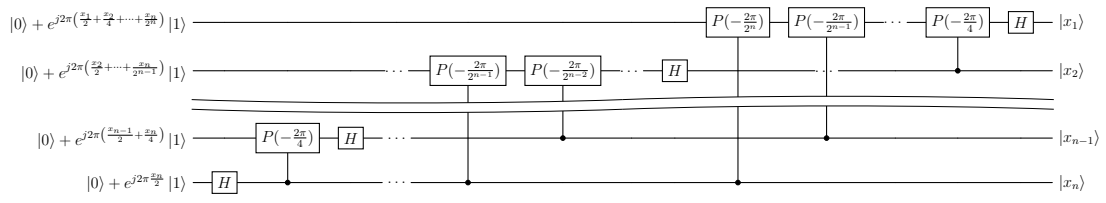

Figure 2.2: QFT* Circuit

Since the Quantum Fourier Transform (and its inverse) can be very useful for estimating the phase of a superposition of states, our wish is to turn it into a gate for use at the right moments in our circuits and on the desired selection of *qubits*. So, for simplicity's sake, to refer to the action of the QFT gate or the QFT* gate on *qubits* ranging from *qubit q* to *qubit p*, we will refer to them as such:

$$\text{gate for the QFT} = QFT_p^q$$
$$\text{gate for the inverse QFT} = (QFT_p^q)^*$$

2.2 Quantum Phase Routine

We will now introduce the subroutine phase. The purpose of this part of the circuit is to add integers or reals stored in a classical register to the phase of a quantum register. Such a routine is used in many quantum algorithms and is thouroughly explained in [**5**] or [**6**], for example.

2.2.1 U_G Gate

First, let's introduce the U_G gate. The purpose of this gate is to introduce an artificial phase (based on a parameter a) to a uniform state superposition $|s\rangle_m$ of the form:

$$|s\rangle_m = \frac{1}{\sqrt{2^m}} \sum_{k=0}^{2^m-1} |k\rangle_m$$

The desired output of the U_G gate applied to $|s\rangle$ is a state superposition of a form analogous to the one we would have obtained via a QFT:

$$|k\rangle \xrightarrow{U_G(a)} \frac{1}{\sqrt{2^m}} \sum_{k=0}^{2^m-1} |k\rangle_m e^{\frac{j2\pi ka}{2^m}} \tag{2.1}$$

Such a gate is represented by the following expression:

$$U_G(a) = \otimes_{i=0}^{m-1} P_i(a\frac{2\pi}{2^{m-i}})$$

with P_i being a phase gate applied on the qubit i. Indeed, applied on $|s\rangle_m$, we obtain the expression (2.1) again:

$$\begin{aligned} U_G(a)\,|s\rangle_m &= \frac{1}{\sqrt{2^m}} \sum_{k=0}^{2^m-1} \left(\prod_{i=0}^{m-1} e^{j2\pi a 2^{i-m} k_i} \right) |k\rangle_m \\ &= \frac{1}{\sqrt{2^m}} \sum_{k=0}^{2^m-1} e^{\frac{j2\pi a}{2^m} \sum_{i=0}^{m-1} 2^i k_i} \,|k\rangle_m \\ &= \frac{1}{\sqrt{2^m}} \sum_{k=0}^{2^m-1} e^{\frac{j2\pi a k}{2^m}} \,|k\rangle_m \end{aligned}$$

A very important result of this section also comes from the expression (2.1). Because of the behaviour of the inverse QFT described previously, we see that, for a integer parameter $a \leq 2^m - 1$:

$$\boxed{QFT^* U_G(a)\,|s\rangle_m = |a\rangle_m} \tag{2.2}$$

For your information, here is an example of a $U_G(3)$ gate on a quantum register of size $m = 5$:

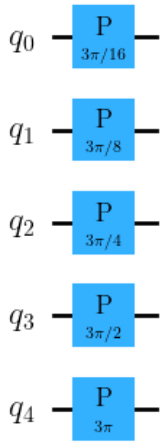

Figure 2.3: U_G Gate Circuit for $m = 5$

The second important property we want to address in this section is the additivity

of U_G gates: $U_G(a)U_G(b) = U_G(a+b)$. Indeed, it's easy to see that:

$$\begin{aligned} U_G(a)U_G(b) &= \left[\otimes_{i=1}^{m-1} P_i(\frac{2\pi a 2^i}{2^m})\right]\left[\otimes_{i=1}^{m-1} P_i(\frac{2\pi 2^i}{2^m})\right] \\ &= \otimes_{i=1}^{m-1} P_i(\frac{2\pi a 2^i}{2^m}) P_i(\frac{2\pi b 2^i}{2^m}) \end{aligned}$$

However, varying the phase once by an angle $\frac{2\pi a 2^i}{2^m}$ and then a second time by an angle $(\frac{2\pi a 2^i}{2^m})$ is equivalent to varying the phase only once by the sum of the angles $(\frac{2\pi(a+b)2^i}{2^m})$. Therefore:

$$\begin{aligned} U_G(a)U_G(b) &= \otimes_{i=1}^{m-1} P_i(\frac{2\pi a 2^i}{2^m}) P_i(\frac{2\pi b 2^i}{2^m}) \\ &= \otimes_{i=1}^{m-1} P_i(\frac{2\pi(a+b)2^i}{2^m}) \\ U_G(a)U_G(b) &= U_G(a+b) \end{aligned}$$

$$\boxed{U_G(a)U_G(b) = U_G(a+b)} \tag{2.3}$$

This second property is particularly useful, since it shows that by using two successive U_G gates, we can add two integer parameters in the phase of a superposition of states. This can come in handy when you want to perform addition with a condition on another *qubit*. For example, with a $U_G(a)$ gate controlled by the i *qubit* (denoted $C_i U_G(a)$) we can perform an addition of a in the phase of our quantum register in all states where the i *qubit* has the value 1. In the next sub-section, we'll take a closer look at the details of such operations.

2.2.2 Operation on real numbers

For the moment, we have been talking about adding integers together. On the other hand, many optimization functions have real values. Consequently, the aim of this subsection is to show a simple method for generalizing the previous results to real values.

Let's illustrate our motivation with a simple example. Let's say we have a quantum register of size $n = 2$, containing a given superposition of states. We also have a second quantum register of size m in which we wish to add:

- the real w in the states where the first *qubit* of the first register has the value 1

- the real v in the states where the second *qubit* of the first register has the value 1

In other words, we would like to our quantum circuit to output the conditioned addition of v and w in our real binary form introduced in Chapter 1, in the second quantum register. This circuit should work like this:

Conditioned Addition

- **Step 1:** Create a state superposition with two entangled quantum registers. The first one has to be set arbitrarily and the second one has to be set to 0 for all *qubits* and for all states: $|q_1 q_2\rangle |0000\rangle$ (if $n = 2$, and $m = 4$).
- **Step 2: For** each state $|q_1 q_2\rangle |0000\rangle$:
  - **If** $q_1 = 1 \longrightarrow$ Add v (in binary) to the second register.
  - **If** $q_2 = 1 \longrightarrow$ Add w (in binary) to the second register.

For example, for a simple case where $v = 0.75$ and $w = -1.25$, the following input-output pairs should be witnessed after the use of our quantum circuit:

Input	**Output**
$\|00\rangle \|0000\rangle$	$\|00\rangle \|0\rangle_4 = \|00\rangle \|00.00\rangle$
$\|01\rangle \|0000\rangle$	$\|00\rangle \|0.75\rangle_4 = \|01\rangle \|00.11\rangle$
$\|01\rangle \|0000\rangle$	$\|00\rangle \|-1.25\rangle_4 = \|10\rangle \|10.11\rangle$
$\|11\rangle \|0000\rangle$	$\|00\rangle \|-0.5\rangle_4 = \|11\rangle \|11.10\rangle$

Table 2.1: Inputs, Outputs Pairs of a Reals Adding Circuit

Thanks to the previous sections, we know that it's enough to create a uniform superposition of 4 states in the first register by means of two H gates, and then to use a QFT^* gate preceded by two controlled U_G gates:

- $C_1U_G(0.75)$: This gate adds 0.75 in binary form to the phase of the second quantum register, only to states where the first *qubit* of the first register has the value 1.
- $C_2U_G(-1.25)$: This gate adds -1.25 in binary form to the phase of the second quantum register, only to states where the second *qubit* of the first register has the value 1.

However, we stated in previous sections that U_G gates followed by an inverse QFT only work with integers. This drawback can be overcome with the following trick.

We transform the reals to be added into their binary formulations, then transform them back into integers corresponding to the binary formulations previously found. For example, for v and w, we obtain:

	Real Values	**Binary Real Values**	**Corresponding Integers**
v	0.75	0011	3
w	-1.25	1011	11

Table 2.2: Table of the Trick Transformation

For future developments, we will denote this trick transformation by $g(w)$. Therefore the actual two CU_G gates we will apply are: $C_1U_G(g(0.75))$ and $C_2U_G(g(-1.25))$ The resulting circuit of the conditionned addition is as follows:

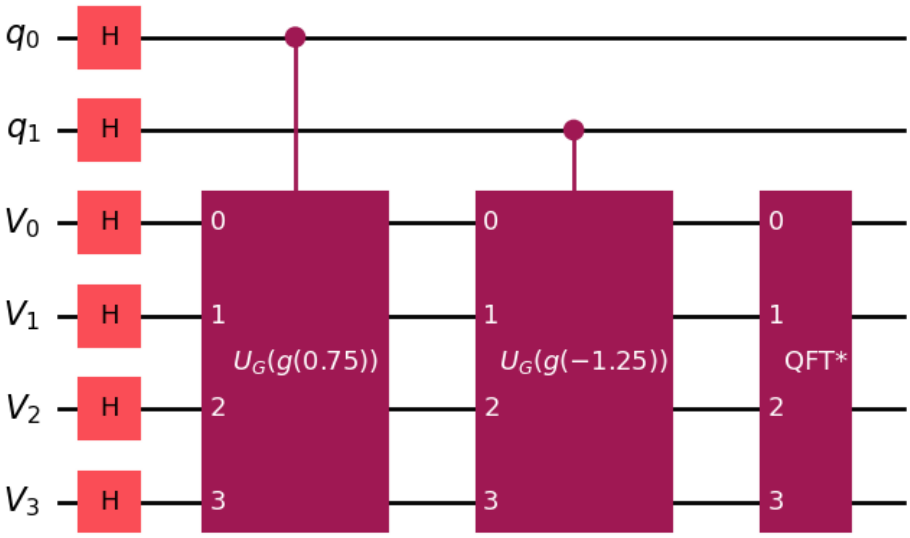

Figure 2.4: Circuit for the conditionned addition of real numbers

To check that the circuit is working correctly, we can run it a large number of times (100000 for example) and then take a measurement at the end of the circuit. This will enable us to obtain the empirical distribution of states contained in our 2 registers. Below, one will find the histogram of the empirical distribution of the output states:

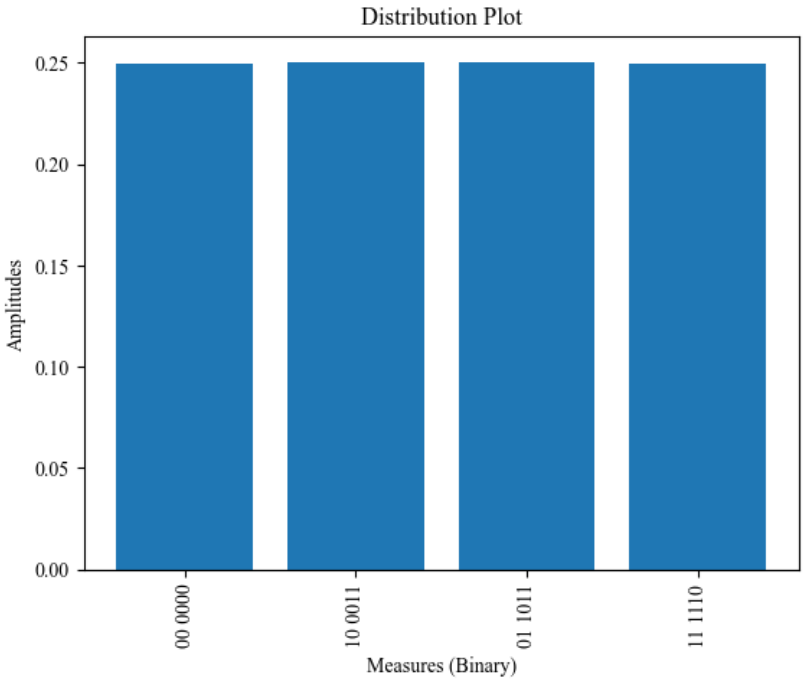

Figure 2.5: Distribution of the outputs of the conditionned addition

As expected, the output is an uniform distribution of states that has the exact same *qubits* values that we had in the table 2.1.

2.3 Grover Search

In this section, we will explore Grover's algorithm, a quantum algorithm designed by Lov K. Grover in [**7**] for unstructured search problems as, for example, the searching of a particular number in a list. Grover's algorithm significantly improves efficiency compared to classical algorithms. Specifically, Grover's algorithm performs a number of operations proportional to the square root of the number of operations required by classical algorithms for the same task. In other words, classical algorithms need at least a quadratic number of operations compared to Grover's algorithm. This quadratic speedup makes Grover's algorithm, along with its variations and foundational techniques, widely applicable, especially in discrete optmization.

The typical problem solved by Grover's algorithm is as follows. Let be a quantum superposition of N states $|\Sigma\rangle_n$. Grover's algorithm finds and measures whether the superposition contains one or more particular states (called marked states).

For example, let's say we have a superposition of 5 states $|10101\rangle$, $|10000\rangle$, $|01010\rangle$, $|00000\rangle$ and $|00111\rangle$. Grover's can help us to efficiently find the states whose first *qubit* has the value 1 with a high probability. In this this case, Grover's algorithm will output, with a high probability, either $|10101\rangle$ or $|10000\rangle$.

To describe it more formally, the Grover's algorithm is an amplification method. It means that it amplifies the amplitudes of the marked states present in $|\Sigma\rangle_n$ so that the circuit *qubits* have a higher chance to be measured as one of the marked states. But how does one can even amplify the amplitude of a state we don't exactly know? To give the answer to this question we will first show what kind of operation can achieve this trick and, then, we will formally describe the quantum circuit representing these operations.

2.3.1 State Amplification

State amplification is based on two main steps, the recognition and the diffusion of the marked states.
The first one, the recognition of marked states. Using a quantum operator O we call *Grover's Oracle*, we will flip the phase of the solution states. In other words, the phases of the solution states, after application of the oracle, will change from 0° to 180°. If we take our example from the introduction to the section, the oracle that recognizes states with a value of 1 on their first *qubit* should act as follows:

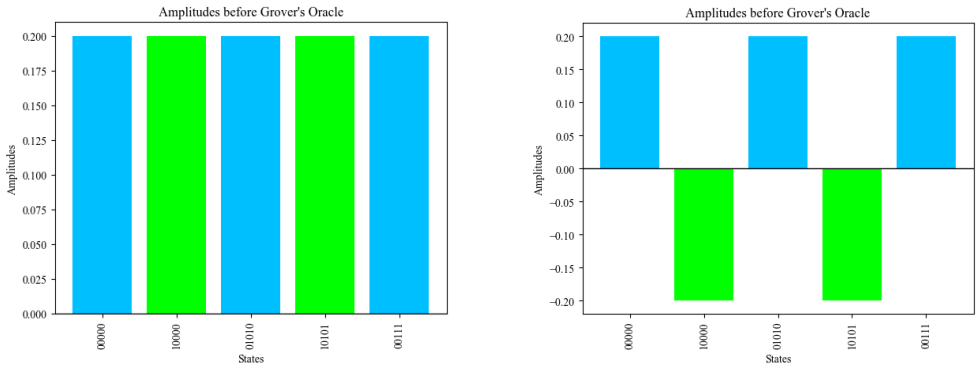

Figure 2.6: Effect of the Grover's Oracle

To help with the vizualization, the solution states bars are green where the other states have blue bars.

The design of such an oracle completely depends on the condition we have on the marked states and therefore have to be designed especially for the case studied.

However, it should always have the following effect:

$$O\left|x\right\rangle = \begin{cases} -\left|x\right\rangle & \text{if } \left|x\right\rangle \text{ is a marked state,} \\ \left|x\right\rangle & \text{else.} \end{cases}$$

with $\left|x\right\rangle \in \left|\Sigma\right\rangle_n$ being a particular state of the $\left|\Sigma\right\rangle_n$ superposition.
In our previous example, O can be designed with the help of an additional *qubit*, initialized with an H gate, whose only purpose will be to help us flip the marked state. Therefore, O will be controlled Z gate $C_1Z(6)$. This gate checks if the first *qubit* of each state has the value 1 and, if so, flip the entire state via a the Z operation on the additional sixth *qubit*. Note that this *qubit* is used only for the circuit and will not be measured in the process nor shown in our distribution graphs.

So, the circuit associated with the Grover's oracle O is given by:

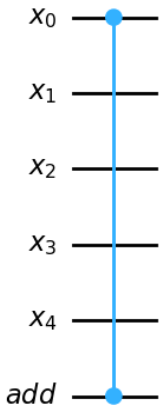

Figure 2.7: Oracle Circuit for our Example

The second step for the amplification of the marked states is the diffusion. It can be seen as an orthogonal symmetry whose axis of symmetry is the average of the amplitudes of the states, after the application of Grover's oracle.

For our example, here is the effect of the symmetry:

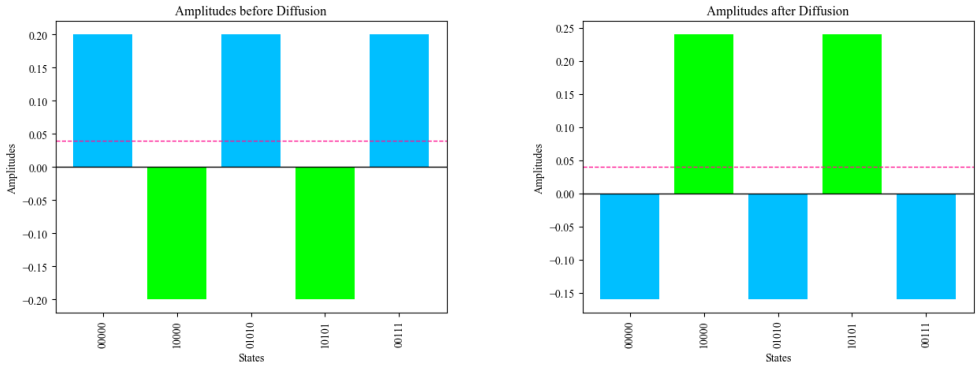

Figure 2.8: Effect of Diffusion

The quantum cicuit of such an operator is a bit more tricky to find and is the combination of three sub-operator. Let's say that the operator associated with the quantum circuit that creates the superposition of our problem and that initializes the additional *qubit* with an H gate is called A. Then:

$$A\left|0\right\rangle_n\left|0\right\rangle = \left|\Sigma\right\rangle_n\left|s\right\rangle$$

with $\left|s\right\rangle = \frac{\left|0\right\rangle+\left|1\right\rangle}{\sqrt{2}}$. Therefore, after the application of the Grover's oracle, we have the superposition $O\left|\Sigma\right\rangle_n\left|s\right\rangle$ where the phases of the marked states only are flipped. If we apply A^* (the inverse of the circuit that creates our initial state superposition), we get a new superposition where all of the marked states went back to the unique state $\left|0\right\rangle_n$ and where the marked states are transformed into some non-null states $\left|y\right\rangle_n$. Then we apply an operator called the diffusor D which works the following way:

$$D\left|x\right\rangle_n = \begin{cases} \left|x\right\rangle_n & \text{if } \left|x\right\rangle_n = \left|0\right\rangle_n, \\ -\left|x\right\rangle_n & \text{else.} \end{cases}$$

The quantum circuit that performs this operation D on the states is:

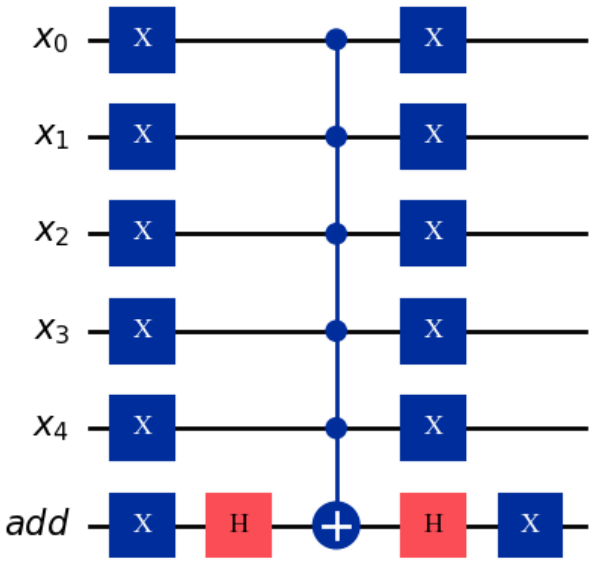

Figure 2.9: Diffusor Operator D

After the application of D, the $|y\rangle_n$ states corresponding to the transformed marked states have their phase flipped again and a non-null amplitude while the $|0\rangle_n$ states corresponding to the marked states remain the same.

Finally, if we apply the operator A again to this new states superposition, we will recover the states of our problem with the difference that the $|y\rangle_n$ states will have had the effect of significantly increasing the amplitude of the solution states and also decreasing the amplitude of the other states.

To summarize, the diffusion operator is ADA^* and corresponds to the following circuit:

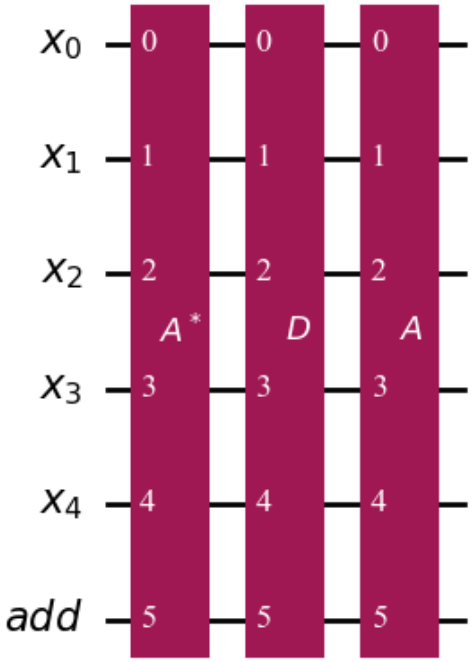

Figure 2.10: Complete Diffusion Circuit

Another important detail we notice on the figure 2.8 is that the resulting amplitudes of the solution states are higher (in absolute value) than the amplitude of the other states. Therefore, a measurement of the *qubits* value has a higher chance to output a solution state. However, in some cases, the probability of measuring a can still be very low. This is why we will often perform state amplification k times. We then say that we are performing k iterations of Grover's algorithm.

It is important to note that k must be chosen carefully. A value of k that is too large may have the opposite effect to that intended, and reduce the amplitude of the marked states. Numerous articles like [**8**] present the formula for choosing Grover's optimal number of iterations:

$$k = \left\lfloor \frac{4\sqrt{N}}{\pi\sqrt{M}} \right\rfloor$$

with N being the number of states in the superposition and M being the number of marked states in the superposition.
In short, Grover's algorithm works as follows:

Grover's Algorithm

- **Step 0:** Compute k the optimal number of Grover's iterations with the formula:
$$k = \left\lfloor \frac{4\sqrt{N}}{\pi\sqrt{M}} \right\rfloor$$
- **Step 1:** Create an operator A that creates the states superposition you want to search in. A should also initialize an additional *qubit* with an H gate:
$$A\left|0\right\rangle_n \left|0\right\rangle = \left|\Sigma\right\rangle_n \left|s\right\rangle$$
- **Step 2:** Perform k iteration of the Grover's iterate ADA^*O:
$$(ADA^*O)^k \left|\Sigma\right\rangle_n \left|s\right\rangle$$
- **Step 3:** Measure the *qubits* at the end of the circuit.

With an optimal number of iterations, the measurement of the *qubits* has the highest possible chance to be a marked state.

2.4 Grover Adaptive Search

On the strength of our latest postulates, we would now like to tackle one of the cornerstones of discrete optimization: the search for a minimum in a list. In this section, we will be looking for the state associated with the binary representation of the minimal number in a states superposition, via manipulations of Grover's algorithm. Such a procedure is called Grover Adaptive Search and is described for integer numbers in [**5**], for example. In this section, we generalize this algorithm to the search for a real-valued minimum.

To better understand the problem, let's take another simple example.

Let's say we have a quantum register of $m = 6$ *qubits*, storing a superposition $\left|\Sigma\right\rangle_m$ of 5 states representing real numbers in their binary form (as a remainder of the Chapter 1, the first *qubit* is the sign *qubit*, the next 3 *qubits* are the integer part of the number and the last 2 *qubits* represent the decimal part of the number.

States	Real Value
$\|1011.01\rangle$	-1.25
$\|0011.01\rangle$	3.25
$\|0000.00\rangle$	0.0
$\|0010.00\rangle$	2.0
$\|0000.11\rangle$	0.75

Table 2.3: Recap of the state superposition

Our aim is to find the state associated with the minimum in this superposition (in our example the state $|11101\rangle$). Unfortunately, in this case it is impossible to use Grover's in the usual way because no oracle are able to detect the minimum of a superposition as it is not an information you can have looking at the value of an unique number. To do this, we need to use a Grover's algorithm sucession in a certain way: the Grover's Adaptive Search. Here's how it works:

GAS for Minimum

- **Step 1:** Randomly measure a state $|x\rangle_m \in |\Sigma\rangle_m$ and set its value as the current minimum $\bar{x} = x$.
- **Step 2:** Update the state superposition to $|\Sigma\rangle_m \longleftarrow |\Sigma - \bar{x}\rangle_m$
- **Step 3:** Create a Grover's O oracle that flips the phase of states representing negative values
- **Step 4:**
  While a ***termination condition*** is not met:
  1. Choose k ***randomly***.
  2. Run Grover's algorithm k times and with O as the Grover's oracle and measure the output $x - \bar{x}$.
  3. **If** $x - \bar{x} < 0$
     - Update the current minimum: $\bar{x} \longleftarrow x$
     - Update the superposition $|\Sigma\rangle_m \longleftarrow |\Sigma - \bar{x}\rangle_m$
- **Step 5:** Return $\bar{x}$ as the problable minimum of the superposition

As one can see, we have intentionally left two terms vague: *termination condition* and *randomly*. We'll explain what we mean by these, but first let's explain the essence of the procedure.
In above algorithm, we iteratively search for the minimum $\bar{x}$ of this list. To do this,

we subtract a randomly chosen value (which we assign to $\bar{x}$) in the superposition from all its other states. In this way, all new states corresponding to negative numbers were those whose x value was less than $\bar{x}$. Indeed:

$$\begin{aligned} & x - \bar{x} < 0 \\ \Leftrightarrow\ & x < \bar{x} \end{aligned}$$

Once this manipulation has been carried out in the superposition, we run Grover's algorithm with an oracle O capable of detecting negative states. This oracle is extremely simple, since the binary representation of the reals we use is such that a number is negative if and only if its first *qubit* is equal to 1. This oracle is therefore the same as the one presented in 2.8, a controlled Z gate.

The state measured (of value x) at the end of the grover is therefore more likely to be associated with a new minimum. If it is the case, we update $\bar{x}$ the superposition again with:

$$\begin{aligned} \bar{x} &\longleftarrow x \\ |\Sigma\rangle_m &\longleftarrow |\Sigma - \bar{x}\rangle_m \end{aligned}$$

These operations are then repeated until a new minimum is found a given number of times. And that's exactly what we will use as a our ***termination condition***. Since Grover's algorithm greatly increases the chance of measuring a new minimal state, we can assume that if we don't find one for a significant number of times, then we've already found the absolute minimum.

Now we need to work out how to choose ***randomly*** k, the number of Grover's iterations. As we explained in the previous section, Grover requires a particular number of iterations, depending on the number of solution states. However, as in this case we never know how many of the updated states represent negative values, we have to choose it randomly. Numerous methods exist, based on the random choice of k among small values. In general, low values will always amplify, but suboptimally. On the other hand, as Grover is used successively, a new minimum will always be found, albeit less quickly.

For the choosing of k at each loop, we will use the procedure presented in [**9**]:

Choice of k values for GAS

- **Step 1:** Set $K = 1$ and $K_{\max} = \left\lfloor \frac{4\sqrt{N}}{\pi} \right\rfloor$
- **Step 2:** Choose k randomly in $\{1, \ldots, \min(K, K_{\max})\}$
- **Step 3:**
  - **If** Grover's outputs a solution state update the new minimu and go back to Step 1 next time you use the procedure.
  - **Else** $K \longleftarrow \left\lceil \frac{5}{4}T \right\rceil$ and go back to Step 2

This procedure is based on a heuristic and provides a good compromise between a low number of iterations for lower complexity (as well as a limited but assured amplification of solution states) and the possibility of testing larger values of k.

Now, let's go back to our example with the superposition shown in table 2.6 and run the GAS with it.

Firstly, we measure a random state of the superposition. Let's say we measured $|001000\rangle$. Then, the provisional minimum is $\bar{x} = 2$. We then update the superposition by substracting $\bar{x} = 2$ to every state and we get the below superposition:

$\lvert x - \bar{x} \rangle$	$x - \bar{x}$
$\lvert 1100.11 \rangle$	-3.25
$\lvert 0001.01 \rangle$	1.25
$\lvert 1110.00 \rangle$	-2.0
$\lvert 0000.00 \rangle$	0.0
$\lvert 1110.11 \rangle$	-1.25

Table 2.4: State superposition after the first update of the GAS

We clearly see that only negative valued states initially contained lower values than $\bar{x}$. Now we have updated our states superposition, let's say the first run of Grover's allow us to measure the new state $|1110.11\rangle$. As this state is negative (-1.25), it means that its original value $-1.25 + \bar{x} = 0.75$ is lower than our current minimum $\bar{x} = 2.0$. We therefore update our provisional minimum $\bar{x} \leftarrow 0.75$ and our state superposition:

$\|x - \bar{x}\rangle$	$x - \bar{x}$
$\|1110.00\rangle$	-2.0
$\|0010.10\rangle$	2.5
$\|1110.11\rangle$	-1.25
$\|0001.01\rangle$	1.25
$\|0000.00\rangle$	0.0

Table 2.5: State superposition after the second update of the GAS

Let's say, by running Grover's a second time we measure the state $|0010.10\rangle$ which is positive (3.25). As it won't be a lower value than $\bar{x}$, we know that we failed to find a better value and we keep going.

Then, we run Grover's for the third time and we get the state $|1110.00\rangle$ which correspond to $x = -2.0 + \bar{x} = -1.25$. We know we found a better minimum and we then update $\bar{x} \leftarrow -1.25$ and the states superposition again:

$\|x - \bar{x}\rangle$	$x - \bar{x}$
$\|0000.00\rangle$	0
$\|0100.10\rangle$	4.5
$\|0001.01\rangle$	1.25
$\|0011.01\rangle$	3.25
$\|0020.00\rangle$	2.0

Table 2.6: State superposition after the thhird update of the GAS

Now, we can run Grover's any number of time we won't ever find a negative state, which means we finally found the global minimum of the list which is -1.25 as expected. In an realistic situation we will stop the running of the GAS when we exceed a Grover's iterations arbitrary threshold.

Chapter 3

Quantum Grid Search Algorithm

The grid search problem is a basic formulation of discrete function optimization. It has many applications in research (bioinformatics, telecommunications network design, airline scheduling, circuit design, and efficient resource allocation, etc.). In a problem with no particular exploitable geometry, it consists in evaluating the values of the function to be optimized on a set of points of the domain and finding the optimum. This type of problem therefore finds a highly inefficient solution, as its complexity increases exponentially with the number of input dimensions. Quantum algorithms, on the other hand, can bring a quadratic acceleration to this type of problem, and this is what we will be discussing in this chapter.

The code of this algorithm is available on the following GitHub.

3.1 Derivative-free Optimization

To show the full range of problems that the grid search architecture can solve, we will code it on the basis of a derivative-free optimization problem, as presented in [**10**]. In this formulation, the task is to optimize a function $f : \mathbb{R}^d \to \mathbb{R}; x \to f(x)$ on an optimization domain $x \in [0; 1]^d$:

$$\min_{x \in [0;1]^d} f(x) \tag{3.1}$$

To apply a grid search on it, we will need to perform a discretization of the problem domain. We will reduce the feasible domain to a grid of points uniformly distributed on the domain, and we will return the optimum of the grid as an estimaton of the true optimum of the problem:

Uniform Grid Search

- **Step 1:** Given an integer $p \geq 1$, consider the set $S(p)$ of points of the form:
$$x_{(i_1,\ldots,i_d)} = \left(\frac{i_1}{p}, \ldots, \frac{i_d}{p}\right) \tag{3.2}$$
with $i_j \in \{0, 1, \ldots, p\}$ for $j = 1, \ldots, d$.
- **Step 2:** Find:
$$\bar{x} = \arg \min_{x \in S(p)} \{f(x)\}. \tag{3.3}$$
- **Step 3:** Return the pair $(\bar{x}, f(\bar{x}))$ as the result.

To explain this method more clearly, here's a figure showing the discretization of the feasible domain for the special case $d = 2$ and $p = 2$:

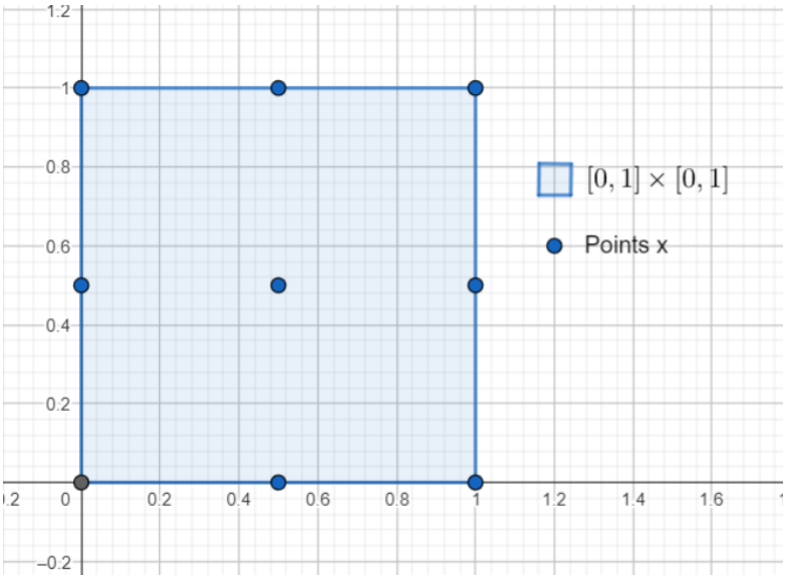

Figure 3.1: Feasible domain and discretization.

As explained above, the method involves discretizing the *feasible domain* (blue surface in 3.1) into $(p + 1)^d$ points (blue points in 3.1) on which the function will be computed and the optimum will be searched.

3.2 Quantum Formulation

From now on, we'll try to translate this problem into its quantum version. For simplicity's sake, we'll reduce the field of functions to be optimized to quadratic functions[1]. As a remainder the quadratic multivariate functions have the following form:

$$f(x) = x^T Ax + b^T x + c \tag{3.4}$$

with $A \in \mathbb{R}^{d\times d}, b \in \mathbb{R}^d, c \in \mathbb{R}$.
To formulate the problem in quantum optimization, we need to transform it into a binary optimization problem on *qubits*. In other words, we need to change the expression of the function so that these arguments are the binary variables $q \in \{0,1\}^{dn}$ required for the binary representation of the x arguments of the original problem. Mathematically, it means we need to go from the formulation:

$$\min_{x\in S(p)} f(x) = x^T Ax + b^T x + c \tag{3.5}$$

to a formulation:

$$\min_{q\in S'(p)} F(q) = q^T A'q + b'^T q + c' \tag{3.6}$$

where $S'(p)$ is the new discretized domain of q and A' b' and c' are the new coefficients of the form.
Before delving into the purely mathematical derivation of such a model, let's explain the concept with an example. Let's say we want to optimize the following quadratic function:

$$f(x) = x^T \begin{pmatrix} 1 & -2 \\ 1.5 & -0.5 \end{pmatrix} x + (2, -0.5)x + 2$$

on a discretized grid with $p = 3$.
We now want to find the equivalent function of the form (3.6). First; q should be a vector containing the binary variables needed to represent the binary formulations of the reals numbers stored in x.

As in this case, x is of length 2 and according to (3.2), q should be the concatenation of 2 binary sequences representing integers bounded by p.

As $p = 3$, the minimal numbre of qubits needed to represent all the possible integers is 2. Therefore, q will be of length $2\times 2 = 4$. So for example, if $x = (0.33\ldots, 0.66\ldots)$, the corresponding vector q will be $q = (0, 1, 1, 0)$.

[1]Later in the chapter, we'll show that our solution is generalizable to all multivariate polynomials.

Now we still need to find the coefficients A', b' and c' such that:

$$f(x) = F(q)$$

For the given example, those coefficients are:

$$A' = \begin{pmatrix} 0 & 0.22 & -0.89 & -0.44 \\ 0.22 & 0 & -0.44 & -0.22 \\ 0.67 & 0.33 & 0 & -0.11 \\ 0.33 & 0.17 & -0.11 & 0 \end{pmatrix}$$

$$b' = \begin{pmatrix} 1.78 \\ 0.78 \\ -0.56 \\ -0.22 \end{pmatrix}$$

$$c' = 2$$

We will get into details about how we got those results in the next sections. For now, it is interesting to note how those two function are strictly equivalent. On the discretized domain of the functions $F(q)$ and $f(x)$ strictly have the same values:

x	q	$f(x)$	$F(q)$
(0,0)	(0,0,0,0)	2.	2.
(0, 0.33)	(0,0,0,1)	1.78	1.78
(0, 0.66)	(0,0,1,0)	1.44	1.44
(0,1)	(0,0,1,1)	1.	1.
(0.33,0)	(0,1,0,0)	2.78	2.78
(0.33,0.33)	(0,1,0,1)	2.5	2.5
(0.33,0.66)	(0,1,1,0)	2.11	2.11
(0.33,1)	(0,1,1,1)	1.61	1.61
(0.66,0)	(1,0,0,0)	3.78	3.78
(0.66, 0.33)	(1,0,0,1)	3.44	3.44
(0.66,0.66)	(1,0,1,0)	3.	3.
(0.66,1)	(1,0,1,1)	2.44	2.44
(1,0)	(1,1,0,0)	5.	5.
(1,0.33)	(1,1,0,1)	4.61	4.61
(1,0.66)	(1,1,1,0)	4.11	4.11
(1, 1)	(1,1,1,1)	3.5	3.5

Table 3.1: Comparison of $f(x)$ and $F(q)$

Now that we have given a good example to get an intuition of the results we are looking for, in the next sections we will focus on the more rigorous mathematical development that will enable us to derive the quantum models with greater ease.

3.2.1 Dimension of the Quantum Formulation

First, let's look at the new dimension of q, A', b', c'. As we have already seen in (3.2), x is actually a vector of integer variables of the form $\frac{i_j}{p}$. Representing x in binary therefore requires d integer representation with n *qubits*. Therefore, $q \in \mathbb{R}^{dn}$ and hence $A' \in \mathbb{R}^{dn \times dn}$, $b' \in \mathbb{R}^{dn}$, $c' \in \mathbb{R}$.

As the integers we try to represent are positive only, n has to be chosen so that $i_j < 2^n$. Again and as seen in the example of the previous section, we can deduce from equation (3.2) that, to use the minimal number of qubits:

$$\begin{aligned} & 2^n - 1 = p \\ \Leftrightarrow \quad & 2^n = p + 1 \\ \Leftrightarrow \quad & n = \log_2(p+1) \end{aligned}$$

One has probably noticed that the above equation is only admissible if $p+1$ is a power of 2. Furthermore, this condition ensures that none of the *qubits* will be unused. Given the limited number of *qubits* available in quantum computers nowadays, reducing the number of *qubits* used is welcome. In the following, we'll consider this condition to be verified and use it in the implementation of our algorithm.

3.2.2 Quadratic Form of the Quantum Formulation

In this section, we will derive the expression of A', b' and c' knowing that:

$$q = (b_{i_1,1}, b_{i_1,2}, \ldots, b_{i_1,n}, b_{i_2,1}, \ldots, b_{i_2,n}, \ldots b_{i_d,n})^T \tag{3.7}$$

with $b_{i_j,k}$ being the *bit* value needed for the *n-bit* represention of the integer i_j, defined in (3.2). Here, we establish the analytic formula we used for our example in the introduction of the section *Quantum Formulation*. To do this, we'll simplify the equations (3.4) and (3.6) to their sum form. Then, we will identify the different terms in the two equations. First, let's derive (3.6) in its sum form:

$$F(q) = q^T A' q + b'^T q + c' = \sum_{i=1}^{nd} \sum_{j=1}^{nd} A'_{ij} q_i q_j + \sum_{i=1}^{nd} b'_i q_i + c' \tag{3.8}$$

Now, let's also derive (3.4) in its sum form:

$$\begin{aligned} f(x) &= x^T A x + b^T x + c \\ &= \frac{i^T}{p} A \frac{i}{p} + b^T \frac{i}{p} + c \\ &= i^T \frac{A}{p^2} i + \frac{b^T}{p} i + c \end{aligned}$$

with $i = px$. For further developments, we introduce two new notations Q and γ:

$$Q = \begin{pmatrix} b_{i_0,n} & \cdots & b_{i_0,1} \\ \vdots & \ddots & \vdots \\ b_{i_d,n} & \cdots & b_{i_d,1} \end{pmatrix}$$

$$\gamma = \begin{pmatrix} 2^{n-1} \\ 2^{n-2} \\ \vdots \\ 2 \\ 1 \end{pmatrix}$$

such that $i = Q\gamma$. These new elements allow us to continue our development by introducing q into our expression via the $b_{i_j,k}$ elements that compose it.

$$f(x) = i^T \frac{A}{p^2} i + \frac{b^T}{p} i + c$$

$$\Leftrightarrow f(x) = \gamma^T Q^T \frac{A}{p^2} Q\gamma + \frac{b^T}{p} iQ\gamma + c$$

$$= \sum_{j=1}^{d} \left(\sum_{k=1}^{n} 2^{n-k} Q_{jk} \right) \left(\sum_{l=1}^{d} \frac{A_{jl}}{p^2} \sum_{u=1}^{n} 2^{n-u} Q_{lu} \right) + \sum_{i=1}^{d} \frac{b_i}{p} \sum_{k=1}^{n} 2^{n-k} Q_{ik} + c$$

$$= \sum_{j=1}^{d} \sum_{k=1}^{n} \sum_{l=1}^{d} \sum_{u=1}^{n} 2^{2n-k-u} Q_{jk} Q_{lu} \frac{A_{jl}}{p^2} + \sum_{i=1}^{d} \sum_{k=1}^{n} 2^{n-k} \frac{b_i}{p} Q_{ik} + c$$

As one can see from the definition of q and Q: $Q_{ij} = q_{in+j}$. This concludes our development:

$$f(x) = \sum_{j=1}^{d} \sum_{k=1}^{n} \sum_{l=1}^{d} \sum_{u=1}^{n} 2^{2n-k-u} q_{jn+k} q_{ln+u} \frac{A_{jl}}{p^2} + \sum_{i=1}^{d} \sum_{k=1}^{n} 2^{n-k} \frac{b_i}{p} q_{in+k} + c \tag{3.9}$$

Finally, thanks to our developments (3.8) and (3.9), we can identify the terms of the expressions A', b' and c' by identification in the equation $F(q) = f(x)$. We get:

$$\boxed{\begin{aligned} A'_{jn+k,ln+u} &= 2^{2n-k-u}\frac{A_{jl}}{p^2} \\ b'_{in+k} &= 2^{n-k}\frac{b_i}{p} \\ c' &= c \end{aligned}}$$

3.2.3 Formulation Redundancy

We can see that this formulation hides a redundancy in its monomials. Indeed, since the vector q is composed of binary variables:

$$q_i = q_i^2$$

Indeed, when $q_i = 1$, $1 = 1^2$ and when $q_i = 0$, $0 = 0^2$.
As a result, monomials, due to the diagonal of A': $q_i A'_{ii} q_i$, can be encoded more efficiently in quantum via the input i vector b': $b'_i q_i$. Indeed, if we replace the diagonal of A' by 0 and update b' as follows: $b'_i \leftarrow b'_i + A'_{ii}$, $F(q)$ remains preserved:

$$\begin{aligned} F'(q) &= \sum_{\substack{i=1 \\ j=1 \\ i\neq j}}^{nd,nd} A'_{ij} q_i q_j + \sum_{i=1}^{nd} b'_i q_i + \sum_{i=1}^{nd} A'_{ii} q_i + c' \\ \Leftrightarrow F'(q) &= \sum_{\substack{i=1 \\ j=1 \\ i\neq j}}^{nd,nd} A'_{ij} q_i q_j + \sum_{i=1}^{nd} b'_i q_i + \sum_{i=1}^{nd} A'_{ii} q_i q_i + c' \\ &= \sum_{\substack{i=1 \\ j=1}}^{nd,nd} A'_{ij} q_i q_j + \sum_{i=1}^{nd} b'_i q_i + c' \\ \Rightarrow F'(q) &= F(q) \end{aligned}$$

As will be explained in a later chapter, creating the superposition of states $(q, F(q))$ requires as many control gates as the addition of the degrees of each monomial. As one can imagine, it can only be beneficial to make the above change.

3.3 Structure of the Quantum Algorithm

Now that we have the quantum formulation of the problem, we can get to the heart of the matter: the design of the algorithm itself. Below, one will find the overall structure of the algorithm based on the routines described in Chapter 2. Some problem-specific details are also provided for these routines.

Quantum Grid Search for the optmization of $F(q)$

- **Step 0:** Choose:
  - $p \geq 1$: the discretization parameter that sets the grid $S(p)$. As previously explained, $p+1$ has to be a power of 2 for simplicity.
  - n: the number of *qubits* used for the arguments q_i binary representations
  - m: the number of *qubits* used for the function values representation of $F(q)$
- **Step 1:** Create a quantum superposition of the states $(q, F(q))$
- **Step 2:** Perform a *Grover Adaptive Search* of the minimum of $F(q)$ on $S'(p)$.
- **Step 3:** Return the result of Step 2: $(q^*, F(q^*))$ as the most probable[a] minimum.

[a] As seen in the *Chapter* 2, The *Grover search* is a non-deterministic algorithm.

3.3.1 State superposition

In this subsection, we'll detail the method we use to generate the quantum superposition of $(q, F(q))$ states.

As explained above, the total number of *qubits* needed to generate these states is $dn + m$: the sum of the number of *qubits* needed to store the vector q and the binary representation of the $F(q)$ values.

As for initializing the first dn *qubits* representing the vector q, we simply subject these qubits to *H gates*. Indeed, since we need to find our minimum among all possible arguments in the discrete domain $S'(p)$, we need to have a superposition of 2^{dn} states corresponding to all possible q values.

Below are the graphs of an example with $d = 2$ and $n = 2$ of the corresponding circuit and the quasi-probabilities histogram of the states superposition:

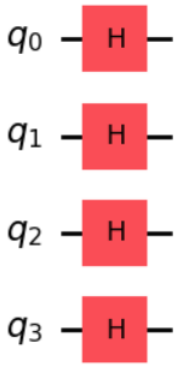

Figure 3.2: Circuit for $d = 2, n = 2$

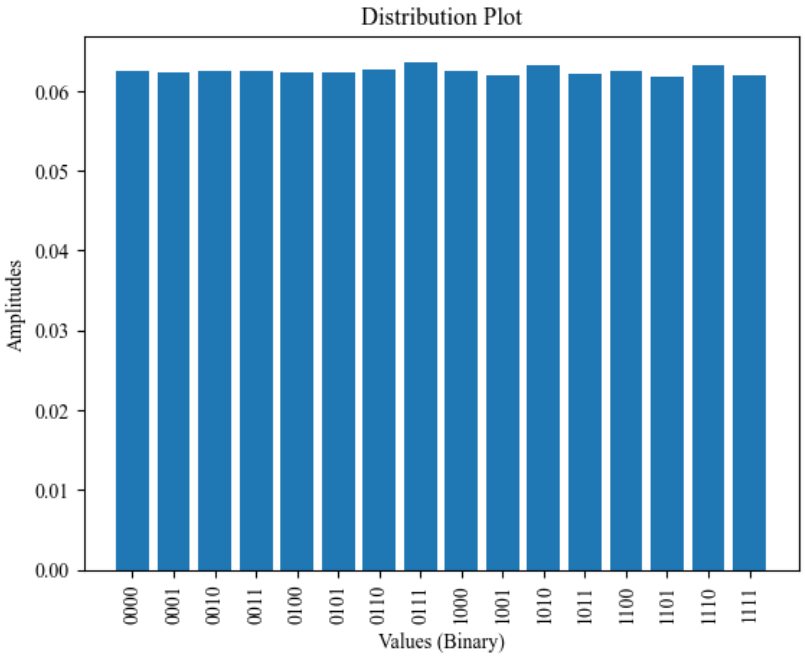

Figure 3.3: Empirical states distribution

As expected, we see from the measured distribution histogram that we see to have perfectly distributed superposition of all arguments q possible.

Now, we would like to convert all these argument values into their corresponding $F(q)$ values on a *m-qubits* quantum register, in a fast and efficient way[2]. In other

[2]We will describe what we mean by fast and efficient in the complexity analysis section.

words, we want to create an operator $U_{\text{pol}}(A', b', c')$ such that:

$$U_{\text{pol}}(A', b', c') \left|q\right\rangle_n \left|0\right\rangle_m = \left|q\right\rangle_{dn} \left|q^T A' q + b'^T q + c\right\rangle_m \tag{3.10}$$

One way to do so is to use to use the *phase subroutine* and the *inverse QFT* to add real numbers as explained in Chapter 2, section 2.2.2.

Such a method was already implemented in numerous articles (as in [**5**] and [**6**] for example), but for binary functions with integer coefficients.

Here, we will generalize it for real-valued functions with real-valued arguments. As already explained in section 2.2.2, the gate $U_G(a)$ is such that:

$$U_G(a + b) = U_G(a)U_G(b) \tag{3.11}$$

$\forall a, b \in \mathbb{R}$ and:

$$U_G(a) \left|s\right\rangle_m = QFT \left|a\right\rangle_m \tag{3.12}$$

$\forall a \in \mathbb{R}$ and with $\left|s\right\rangle_m = \otimes^m H \left|0\right\rangle_m$.

Therefore, one can easily see that:

$$\begin{aligned} U_G(q^T A' q + b'^T q + c) \left|s\right\rangle_m &= U_G(\sum_{i=1}^{dn} \sum_{j=1}^{dn} A'_{ij} q_i q_j + \sum_{i=1}^{dn} b'_i q_i + c) \left|s\right\rangle_m \\ &= \Pi_{i=1}^{dn} \Pi_{j=1}^{dn} U_G(A'_{ij} q_i q_j) \Pi_{i=1}^{dn} U_G(b'_i q_i) U_G(c') \left|s\right\rangle_m \end{aligned}$$

Let us have a closer look at $U_G(A'_{ij} q_i q_j)$. As a reminder, q_i hides the values of the i-th *qubit* of the *n-qubits* argument register, for each state in the superposition.

Similarly, q_j represent the values of the j-th *qubit* in the argument register, for each state in the superposition.

A legitimate question would be to ask what happens depending on the value of q_i and q_j in each state. If we are in a state where $q_i = 0$ or $q_j = 0$, then we have:

$$\begin{aligned} U_G(A'_{ij} q_i q_j) \left|s\right\rangle_m &= U_G(0) \left|s\right\rangle_m \\ &= \left|s\right\rangle_m \end{aligned}$$

However, if $q_i = 1$ and $q_j = 1$, then:

$$U_G(A'_{ij} q_i q_j) \left|s\right\rangle_m = U_G(A'_{ij}) \left|s\right\rangle_m$$

We see that this gate acts just like a controlled U_G gate with the coefficient A'_{ij} as an argument. The control *qubits* are obviously q_i and q_j. The exact same reasoning is possible for the gate $U_G(b'_i q_i)$. This previous gate is equivalent to a controlled U_G gate with the coefficient b'_i as an argument with q_i as the control *qubit*. So we can modify our previous development:

$$\begin{aligned} U_G(F(q))\,|q\rangle_{dn}\,|s\rangle_m &= U_G(\sum_{i=1}^{dn}\sum_{j=1}^{dn} A'_{ij}q_iq_j + \sum_{i=1}^{dn} b'_i q_i + c)\,|q\rangle_{dn}\,|s\rangle_m \\ &= [\Pi_{i=1}^{dn}\Pi_{j=1}^{dn} C_{i,j}U_G(A'_{ij})][\Pi_{i=1}^{dn} C_i U_G(b'_i)]U_G(c')\,|q\rangle_{dn}\,|s\rangle_m \end{aligned}$$

From from (3.12), we can also see that:

$$\begin{aligned} U_G(F(q))\,|q\rangle_{dn}\,|s\rangle_m &= |q\rangle_{dn}\,QFT\,|F(q)\rangle_m \\ \Leftrightarrow |q\rangle_{dn}\,QFT^* U_G(F(q))\,|s\rangle_m &= |q\rangle_{dn}\,|F(q)\rangle_m \end{aligned}$$

The combination of this last two results concludes this section. Below is a summary of the gate that create a states superposition of the arguments and their corresponding $F(q)$ values.

$$U_{\text{pol}}(A', b', c') = (QFT_{dn+1}^{m+dn})^* CU_G(q^t A' q + b'^T q + c)(\otimes_{i=dn+1}^{m+dn} H)$$

$$\text{with } U_G(q^t A' q + b'^T q + c) = [\Pi_{i=1}^{dn}\Pi_{j=1}^{dn} C_{i,j}U_G(A'_{ij})][\Pi_{i=1}^{dn} C_i U_G(b'_i)]U_G(c')$$

For the moment, these formulas probably seem rather complex and unintuitive. To better understand the latest developments, we'll illustrate them with an example. Let us say we want to optimize the function:

$$f(x) = x^T \begin{pmatrix} 0 & -2 \\ 0 & -0.5 \end{pmatrix} x + (0, -0.5)x + 1.5$$

on a discretized grid with $p = 3$. This grid is exactly the same as in Figure 3.1. The corresponding quantum formulation is the following:

$$F(q) = q^T \begin{pmatrix} 0 & 0 & -0.89 & -0.44 \\ 0 & 0 & -0.44 & -0.22 \\ 0 & 0 & 0 & -0.11 \\ 0 & 0 & -0.11 & 0 \end{pmatrix} q + \begin{pmatrix} 0 \\ 0 \\ -0.11 \\ -0.11 \end{pmatrix}^T q + 1.5$$

On the discretized grid $S'(3)$, $F(q)$ (or equvalently $f(x)$) gives the following values:

x	q	$F(q)$
(0,0)	(0,0,0,0)	1.5
(0, 0.33)	(0,0,0,1)	1.38
(0, 0.66)	(0,0,1,0)	1.38
(0,1)	(0,0,1,1)	1.5
(0.33,0)	(0,1,0,0)	1.5
(0.33,0.33)	(0,1,0,1)	1.17
(0.33,0.66)	(0,1,1,0)	0.94
(0.33,1)	(0,1,1,1)	0.83
(0.66,0)	(1,0,0,0)	1.5
(0.66, 0.33)	(1,0,0,1)	0.94
(0.66,0.66)	(1,0,1,0)	0.5
(0.66,1)	(1,0,1,1)	0.17
(1,0)	(1,1,0,0)	1.5
(1,0.33)	(1,1,0,1)	0.72
(1,0.66)	(1,1,1,0)	0.06
(1, 1)	(1,1,1,1)	-0.5

Table 3.2: Pairs $(q, F(q))$

If we want to create a quantum states superposition of the pairs $(q, F(q))$ with the routine we introduced in this section, we get the following quantum circuit:

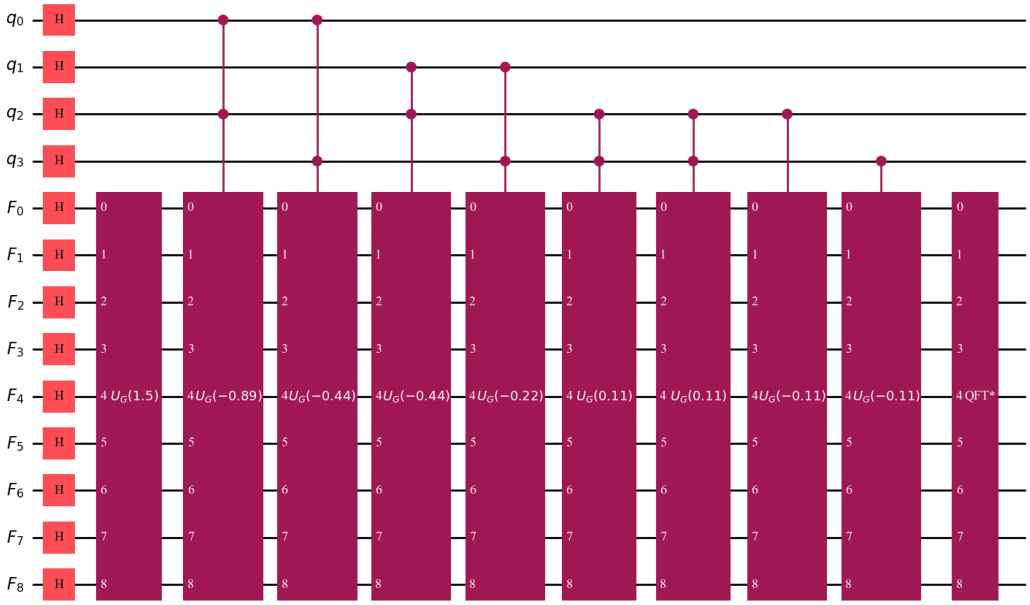

Figure 3.4: Quantum circuit for $(q, F(q))$ pairs superposition.

As you can see, this quantum circuit was built such that the quantum register for the function values $F(q)$ was of length $m = 9$.

To better understand what this circuit does to every states, let us go through the circuit with a state $q = |0, 1, 0, 1\rangle$, $|F(q)\rangle_9 = |0\rangle_9$ as an example.

The first gates are H-gates on the function value register. Therefore, after the first gates, the values of the qubits for our state is:

$$|F(q)\rangle_9 = \frac{1}{\sqrt{2^9}} \sum_{k=0}^{2^9-1} |k\rangle_9$$

with a 0-phase.

Then the state goes through a $U_G(1.5)$ gate. This gate is designed to add the constant c' to the F(q) register. As seen in chapter 2, such a gate outputs the following state:

$$|F(q)\rangle_9 = \frac{1}{\sqrt{2^9}} \sum_{k=0}^{2^9-1} e^{jkg(1.5)} |k\rangle_9$$

with $g(x)$ being the integer corresponding to the binary representation of real number x.

After this, the state passes through a series of multicontrolled U_G gates. These gates, each controlled by two *qubits* q_i and q_j, are designed to add A'_{ij} to the phase of states associated with $q = 0, 1, 0, 1$ if and only if $i = 1$ and $j = 1$. Because of this double condition, only the $U_G(-0.22)$ gate is active. Therefore, after those gates, the states superposition associated with $q = (0, 1, 0, 1)$ are:

$$|F(q)\rangle_9 = \frac{1}{\sqrt{2^9}} \sum_{k=0}^{2^9-1} e^{jk(g(1.5)-g(0.22))} |k\rangle_9$$

Finally, these states pass through two final, simply controlled gates. The purpose of these two gates is to add b'_i to the phase of the states concerned if the *qubit* $q_i = 1$. This is why only the $U_G(0.11)$ gate is activated. The states associated with $q = (0, 1, 0, 1)$ become:

$$|F(q)\rangle_9 = \frac{1}{\sqrt{2^9}} \sum_{k=0}^{2^9-1} e^{jk(g(1.5)-g(0.22)-g(0.11))} |k\rangle_9$$

At that point, the last gate for our example to go through is the inverse QFT. This gate has the objective to replace the states superposition stored in $|F(q)\rangle_9$ associated with $q = (0, 1, 0, 1)$ with a single state that is the binary representation of the current phase of $|F(q)\rangle_9$. The gate outputs the following state:

$$|F(q)\rangle_9 = |1.5 - 0.22 - 0.11\rangle_9 = |1.17\rangle_9$$

That's indeed the function value we were looking for as written in the table 3.2
Now, we can also check empirically this routine works for every states by running the circuit 10000 times and measuring each time the *qubits* values. Here are the result of such an expermiment:

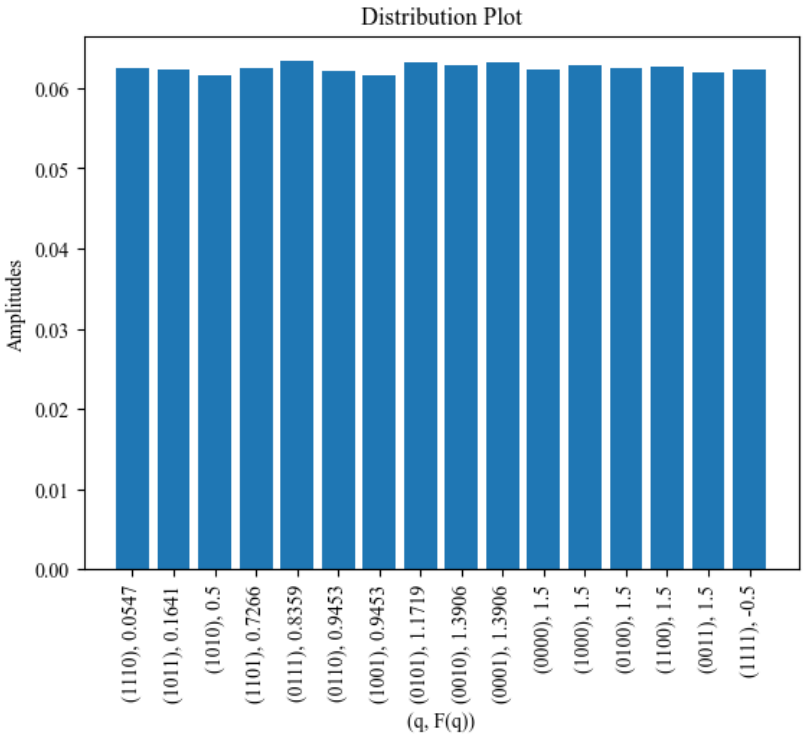

Figure 3.5: Distribution of the *qubits* values

As we could imagine, the circuit does create a equally distributed superposition of every $(q, F(q))$ pairs. Indeed, we found the superposition of all argument and value pairs we computed in table 3.2. Some small rounding errors, however, are present due to the fact our binary format for real numbers contains only 9 *qubits*.

3.3.2 Optimization of Higher Degree Polynomials

The state superposition method $(q, F(q))$ described in the previous section can of course be generalized to all multivariate polynomials of order n. However, we have not coded this possibility for simplicity's sake. In this sub-section, we outline how this can be done.

First of all, we need to find a binary quantum formulation from the classical formulation for a polynomial of order n, as we did in section 4.2. Once this has been achieved, we simply create a circuit similar to the one shown in figure 3.4, adding the U_G ports corresponding to the new higher-order monomials. As these monomials are of higher order, the corresponding U_G port will have as many control *qubits* as its order.

For example, let us say that we wish to optimize a multivariate polynomial of degree ≥ 4: $P(q)$. Let us also say that $P(q)$ has a degree 4 monomial of the form $-3.2q_0q_2q_4q_5$. Similarly to what has been done previously, the gate that will account for this monomial in the creation of the states superposition is $C_{0,2,4,5}U_G(-3.2)$ and correspond to the following circuit:

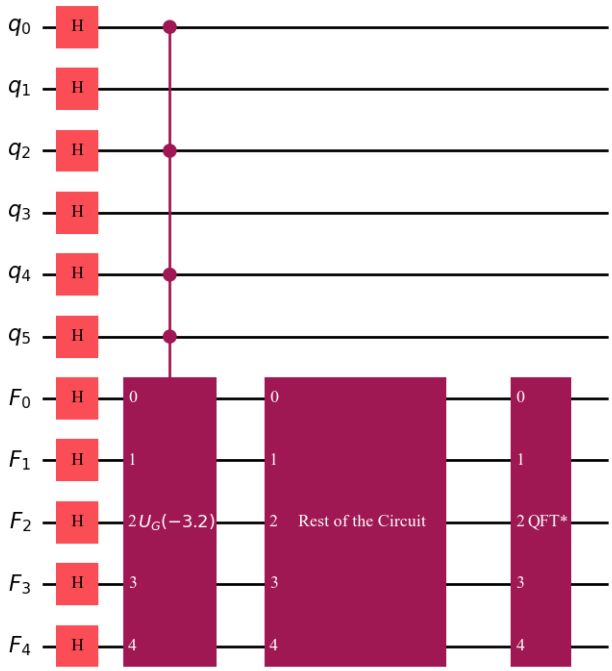

Figure 3.6: Quantum circuit for $(q, P(q))$ pairs superposition.

To produce this example, we used the following circuit parameters: $d = 2$, $m = 5$, $n = 3$.

3.4 Grover Adaptive Search for QGS

In this subsection, we explain how we used Grover's algorithm to find the state corresponding to the minimum in the states superposition previously obtained. As explained in Chapter 2, we are going to use a Grover Adaptive Search for minimum of the following structure:

GAS for Minimum

- **Step 0:**
  1. Let $A(c)$ be the unitary transformation creating the superposition $|q\rangle_{nd}|F(q)-c\rangle_m|s\rangle_1$, with $|s\rangle_1 = \frac{|0\rangle+|1\rangle}{\sqrt{2}}$.
  2. Let O be the unitary transformation that flips the phase of states where $F(q)-c$ is negative.
- **Step 1:**
  1. Set StillFindingMins = True
  2. Set $K_{\max} = \left\lceil \frac{\pi\sqrt{2^{nd}}}{4} \right\rceil$
  3. Choose randomly a pair $(\bar{q}, F(\bar{q}))$
  4. Create the superposition

$$|q\rangle_{nd}|F(q)-F(\bar{q})\rangle_m|s\rangle_1 = A(F(\bar{q}))|0\rangle_{nd}|0\rangle_m|0\rangle_1$$

- **Step 2:**
  1. **While** StillFindingLower:
     (a) Set StillFindingLower = False
     (b) Initialize $K = 1$
     (c) **While** $K \leq K_{\max}$:
     - Choose k randomly in $\{1, \ldots, K\}$
     - Measure $q, F(q)-F(\bar{q})$ by running Grover's k times and with O as the Grover's oracle.
     - **If** $F(q)-F(\bar{q}) < 0$:
       * Set StillFindingLower = True
       * Update $F(\bar{q}) = F(q)$
       * Update $\bar{q} = q$
       * Update the superposition

$$|q\rangle_{nd}|F(q)-F(\bar{q})\rangle_m|s\rangle_1 = A(F(\bar{q}))|0\rangle_{nd}|0\rangle_m|0\rangle_1$$

     - **Else** Set $K = \left\lceil \frac{5}{4}K \right\rceil$

Allow us to break these steps down for a better understanding. This central step of the algorithm consists in finding the minimum in the superposition of pairs (q,

F(q)). To do this, we randomly select a pair $(\bar{q}, F(\bar{q}))$ from our problem in step 1, to make it our provisional minimum. Next, we use the operator $A(F(\bar{q}))$ to generate the superposition $|q\rangle_{nd} |F(q) - F(\bar{q})\rangle_m |s\rangle_1$. This operator is the same as $U_{\text{pol}}(A', b', c')$, but with three differences:

- The first is that, before the inverse QFT, we add an $U_G(-F(\bar{q}))$ gate.
- The second is the addition of an H gate on the additional *qubit* needed for Grover's algorithm.
- The last difference is that A also includes the initialization of the argument register $|q\rangle_{dn}$ with H gates.

Therefore the operator $A(F(\bar{q}))$ is given by:

$$A(F(\bar{q})) = (QFT_{dn+1}^{dn+m})^* U_G(-F(\bar{q}))CU_G(F(q))(\otimes^{dn+m+1} H)$$

Below is the equivalent circuit to the operator $A(F(\bar{q}))$ with the parameters $d = 2$, $n = 2$, $m = 4$:

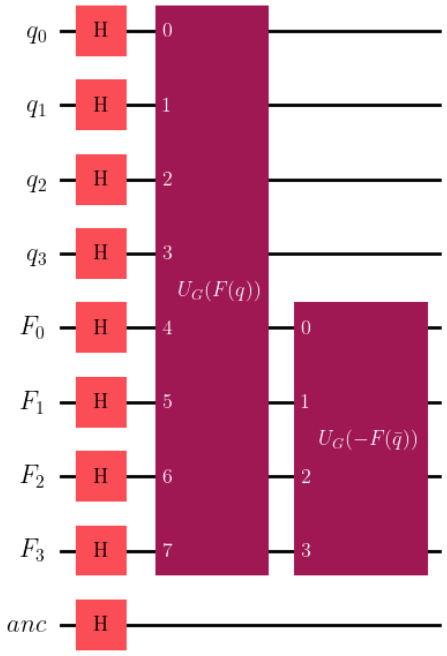

Figure 3.7: $A(F(\bar{q}))$ Quantum Circuit

The rest of the algorithm works in exactly the same way as described in Chapter 2, section 2.4, with the difference that the search is made over a larger quantum register

containing the sub-register $|q\rangle_{nd}$ to find the minimal argument corresponding to the minimum of $F(q)$.

3.5 Benchmark of the Algorithm

Now we have a working algorithm to minimize discrete real valued functions, we would like to analyze the efficiency of our algorithm both theoretically and empirically.

To do so, we will first compute the worst case complexity and compare it with the classical algorithm. Then we'll run our *Qiskit* simulator to assess its empirical convergence.

3.5.1 Worst Case Complexity

In quantum algorithms, complexity is calculated by counting the number of vertical layers of gates our circuit contains. Since all operations in a given layer are performed at the same time, this gives us a point of comparison with classical algorithms.

Complexity of the State Superposition

The creation of the states superposition $|q\rangle_{nd} |F(q) - F(\bar{q})\rangle |s\rangle_1$ requires an initialization by H gate, then a succession of U_G gates and finally an inverse QFT.

Gate H initialization requires a single gate H layer for all registers. Therefore, its complexity is: $\mathcal{O}(1)$.

Implementing our U_G gates requires a single layer of P quantum gates, as shown in figure 2.3. Consequently, if, in the worst-case scenario, the matrix A and the vector b are full, we'll need $\mathcal{O}((nd)^2 + nd + 2) = \mathcal{O}((nd)^2)$ U_G gates (one for each entry of A, b and c and one more for the subtraction of $F(\bar{q})$).

In 2000, Nielsen and Chuang analyzed the complexity of the inverse QFT circuit in their book [**4**], and found a complexity of $\mathcal{O}(m^2)$. Since our implementation is the same as theirs, we use the same complexity, with m being the number of *qubits* in the $|F(q) - F(\bar{q})\rangle$ register.

To sum up, the complexity of state superposition is:

$$\mathcal{O}((nd)^2 + nd + 3 + m^2) = \mathcal{O}((nd)^2 + m^2)$$

Complexity of the Grover Adaptive Search

The complexity of GAS is a little trickier to calculate because this algorithm doesn't have a fixed number of iterations. In addition, the number of certain GAS steps is chosen randomly. This is why we base our analysis on Grover's own analysis in [**7**], which states that Grover's algorithm has a complexity of $\mathcal{O}(\sqrt{2^{nd}})$ in our case. For the full GAS, the worst case complexity becomes:

$$\mathcal{O}(TK_{\max}\sqrt{2^{nd}})$$

where T is the number of provisional minima found before the algorithm returns. If we study the asymptotic complexity of the algorithm, i.e. as n and d increase, T and $K_{\max}$ become negligible in relation to $\sqrt{2^{nd}}$. The total complexity of the GAS is:

$$\mathcal{O}(\sqrt{2^{nd}})$$

Complexity of the QGS

The complexity of our total algorithm is obtained by summing the complexity of the two main blocks of the algorithm: state superposition and GAS:

$$\mathcal{O}((nd)^2 + \sqrt{2^{nd}})$$

As for large values of nd: $(nd)^2 < \sqrt{2^{nd}}$, we conclude that the worst case complexity of our algorithm is therefore:

$$\boxed{\mathcal{O}(\sqrt{2^{nd}})}$$

3.5.2 Comparison with Classical Algorithm

The classic Grid search algorithm is detailed in [**10**] and its complexity is given by:

$$\mathcal{O}((p+1)^d)$$

In fact, this algorithm amounts to calculating the valuers of the function f(x) on the $(p+1)^d$ point grid and finding the minimum.
Since we know that in our quantum formulation: $p = 2^n - 1$, we can write the complexity of the classical algorithm as follows:

$$\mathcal{O}(2^{nd})$$

We can see that, assymtotically, our solution produces a quadratic acceleration of the algorithm. If we vary the quantity nd, here's a comparison of the evolution of the number of iterations of the two versions:

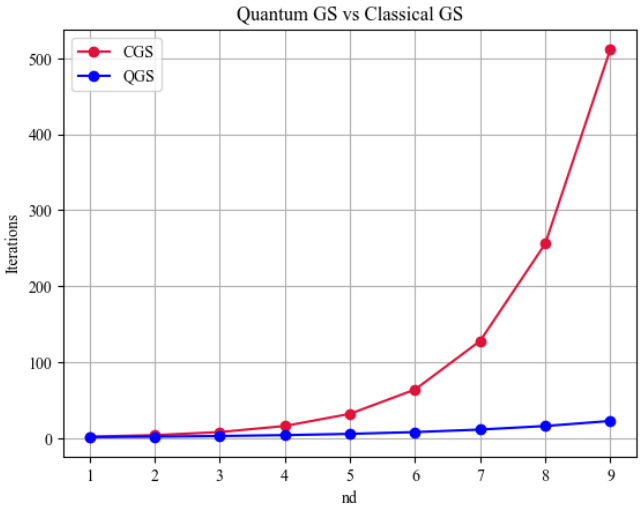

Figure 3.8: Comparison of the Classical and Quantum algorithms

Now that we have the same variables, we can compare the complexities of the classical algorithm with our quantum version.

3.5.3 Analysis of Empirical Convergence

In this section, we simulate our quantum algorithm on a *Qiskit qubits* simulator to analyze its empirical complexity. Indeed, the analysis in the previous sections gives an idea of the worst-case complexity. However, our algorithm is likely to converge sooner. This is why we're going to tabulate the number of iterations of Grover's algoritm at which we obtain the provisional minimums of the grid and then the global minimum.

To do this, we will use the following example and parameters:

$$\min_{x \in S(p)} x^T \begin{pmatrix} 0 & -2 \\ 0 & -0.5 \end{pmatrix} x + (0, -0.5)x + 1.5$$

and:

$$\begin{aligned} p &= 7 \\ m &= 9 \\ K_{\max} &= 5 \end{aligned}$$

Grover Iteration	$F(\bar{q})$
1	0.36
2	0.23
3	-0.21
5	-0.28
10	**-0.5**

Table 3.3: Iteration and Provisional Minimum Values

Here is the graph of the evolution of the provisional minimum $F(\bar{q})$ against the number of Grover iterations needed.

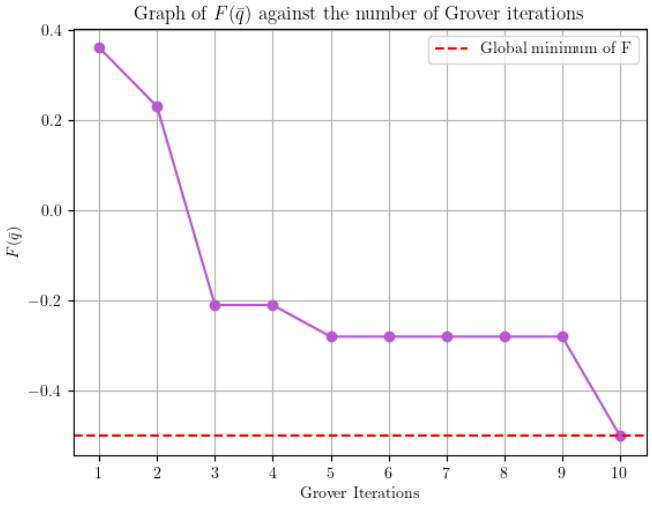

Figure 3.9: Convergence to the Global minimum

We observe that the global minimum is found in just 10 iterations of Grover, equivalent to 10 evaluations of F for the classical algorithm. Given that the admissible domain contains 64 points in total, this already represents a clear improvement.

Since the time required to simulate *qubits* increases exponentially, it is difficult to test our implementation for larger problems. Nevertheless, there's every reason to believe that our algorithm can offer a highly advantageous solution for higher-dimensional problems.

3.6 To go Further

There is still a great deal of room for improvement in this algorithm. The very architecture of our method could still be modified. The three main improvements

we would like to make are as follows:

- The first improvement is the one we have already discussed in section 3.3.2. It involves making our code compatible with the optimization of higher-degree multivariate polynomials.
- The second improvement would be another implementation of inverse QFT. In 2000, Richard Cleve and John Watrous [**11**] proved that it was possible to perform an exact Quantum Fourier Transform of m qubits in maximum $O(m(\log m)^2 \log(\log m))$. Although this enhancement does not account for the bulk of the final complexity, it could still be beneficial to use it.
- Finally, for optimization derivatives-free problems, we might want to implement our program recursively. Quantum Grid Searches would be performed in sequence to obtain the most accurate minimum in the fewest possible iterations.

Chapter 4

Quantum Weighted Max-SAT Algorithm

4.1 Description of the problem

In this chapter we will describe our quantum algorithm to solve the Quantum Weighted Max-SAT problem.

The Boolean satisfiability problem, also called SAT problem, is a well-know problem where, for given boolean variables, we search an assignment of values such that all the boolean expressions (we will call them clauses) of the problem are true.

The problem on which we focus on in this work is the Weighted Max-SAT problem. This is a variant of the SAT problem where each clause has a given weight associated, and where it is also possible to not find values for the variables satisfying all the clauses. The problem becomes then a maximisation problem over the sum of the weights of the satisfied clauses (for more information, see [**12**]).

The code of this algorithm is available on the following GitHub.

4.1.1 Conjunctive normal form

Using boolean algebra, it is possible to transform any logical expression in conjunctive normal form. The conjunctive normal form (CNF, see [**13**]) is a way to express logical expressions using only conjuctions, disjuctions and negations. In CNF, all clauses are in conjunction, and each clause is only composed of disjunctions of variables.

The CNF is a convenient form to express SAT problems and a lot of solvers are based on it, but it presents one inconvenient: sometimes expressing a problem in CNF grows exponentially with the size of the problem.

For the quantum Weighted Max-SAT problem, we will consider that we associated a weight to each of the single clauses.

4.1.2 Example of SAT problem: the 8-queens problem

Imagine a chess board where you want to place 8 queens such that they can't take each others. Let's assign to each square of the board a variable v_1 to v_{64} from the top left to the bottom right. We will assign the value "true" if a queen lays on the square, and "false" otherwise.

For each row i, $0 \leq i \leq 7$, we can assign the following condition:

$$\forall 1 \leq j \leq 8 : v_{8i+j} \Rightarrow \neg(v_{8i} \vee \cdots \vee v_{8i+j-1} \vee v_{8i+j+1} \vee \cdots \vee v_{8i+8})$$

which means that if a queen lays on a square of a row, then none of the other squares of the row could contain a queen. Of course this is not in CNF, but we can find an equivalent formulation that transposes the conditions above in CNF for each row i:

$$\begin{aligned}
& (v_{8i+1} \vee v_{8i+2} \vee v_{8i+3} \vee v_{8i+4} \vee v_{8i+5} \vee v_{8i+6} \vee v_{8i+7} \vee v_{8i+8}) \\
\wedge \quad & (\neg v_{8i+1} \vee \neg v_{8i+2}) \\
\wedge \quad & (\neg v_{8i+1} \vee \neg v_{8i+3}) \\
\wedge \quad & \cdots \\
\wedge \quad & (\neg v_{8i+1} \vee \neg v_{8i+8}) \\
\wedge \quad & (\neg v_{8i+2} \vee \neg v_{8i+3}) \\
\wedge \quad & \cdots \\
\wedge \quad & (\neg v_{8i+7} \vee \neg v_{8i+8})
\end{aligned}$$

The first clause ensures that there is at least one queen on the given line. All the following ones check that there are not 2 queens on 2 different squares of the line (the variables associated can not be true at the same time).

With this short example, we can see a drawback of the CNF formulation: if we have a generalized chess board of $n \times n$ squares, formulating the row conditions

takes n conjuncted boolean expressions if we are allowed to express them in any form, but when we want to express the same conditions in CNF, it is possible that this number increases. For example here, we have $1 + \frac{n(n-1)}{2}$ clauses for one single row, which is quadratic and not linear anymore in n.

The remaining clauses are the clauses expressing that two queens can not take each other on a diagonal or a colums. The number of these clauses also grows quadratically.

4.1.3 Mathematical definition of the Weighted Max-SAT problem using CNF

A clause is a disjuncion of variables or negated variables. Let c_i be the truthiness of the ith clause of the problem. For each i, there is a weight $w_i > 0$ associated to c_i. Let x be the variables of the problem. For each clause, we can define 2 sets S_i^+ and S_i^- such that S_i^- contains all the indexes of the variables negated in the ith clause, and S_i^+ contains the indexes of the non-negated variables in the ith clause. Let's also consider that there are n clauses and m variables.

Then we can formulate the Weighted Max-SAT problem as:

$$
\begin{aligned}
\text{maximize: } & \sum_{i=1}^{n} w_i c_i & \\
\text{subject to: } & c_i \leq \sum_{y \in S_i^+} x_y + \sum_{z \in S_i^-} (1 - x_z) & \forall i \in \{1, \cdots, n\} \\
& c_i \in \{0, 1\} & \forall i \in \{1, \cdots, n\} \\
& x_j \in \{0, 1\} & \forall j \in \{1, \cdots, m\}
\end{aligned}
$$

During this chapter, we will only deal with integer weights.

4.1.4 Example of Weighted Max-SAT problem: the superqueens problem

Imagine a new chess piece : the superqueen. The superqueen can move like a normal queen and a knight at the same time. This time, it is impossible to place 8 superqueens on a classical chess board so they can not take each others.

If we reformulate the preceding problem by adding weights to constraints, it is nevertheless possible to find the maximum number of superqueens we can place on

a board.

There are two kinds of clauses in the 8-queens problem. First there are the constraints ensuring that there is at least one queen by row, then there are the constraints ensuring that 2 queens can not take each others. Same sets of clauses can be created for the superqueens problem. Let's name the 2 set of clauses $\mathcal{C}_1$ and $\mathcal{C}_2$.

If we have a chess board of size $n \times n$, we can assign a weight of 1 to every clause in $\mathcal{C}_1$ and a weight of $n + 1$ to every clause in $\mathcal{C}_2$. This means that by violating a clause in $\mathcal{C}_2$ (i.e. two pieces can take each other), we lose more weight that by successfully placing all superqueens. By using a Weighted Max-SAT solver, we could thus find the maximum number of superqueens that we can place on a chess board.

4.2 Our algorithm

In this section, we will describe our implementation of the Weighted Max-SAT algorithm and the choices we made to tackle the problem. The main idea behind this algorithm is to benefit from the binary domain of the variables (either true or false) to assign one qubit by variable. We test each clause to see if there are true or false and we add the corresponding weight in a integer register. Finally, we perform a Grover Adaptive Search (GAS) on the integer register to find the maximal value in the superposition. When we find it, we can find one corresponding assignment of the variables that corresponds to the given maximal sum of weights.

The full algorithm can be described as following:

Quantum Weighted Max-SAT solver using Grover Adaptive Search

- **Step 0:** Choose a convenient number t such that if you do not find a better solution after t adaptive Grover iterations, you can consider that you have found the optimal solution. Consider an easy-to-compute initial optimal sum of weights score (for example: all variables set to true). Let's name this value v.
- **Step 1:** Let $K = 1$ and $K_{\max} = \left\lfloor \frac{4\sqrt{2^{n_{\text{vars}}}}}{\pi} \right\rfloor$.
- **Step 2:** Pick a random number k between 1 and $\min(K, K_{\max})$.
- **Step 3:** Build the following blocks:
  - **Block A:** A block containing a Quantum Weighted Max-SAT evaluation step (see below). Take the opposite from the sum of weights register, subtract v, add the sum of weights (so the maximization problem becomes a minimization problem easier to handle) and perform an inverse QFT transformation. For the purpose of this algorithm, you need to consider signed integers. This block is the one corresponding to the state superposition preparation.
  - **Block O:** A block containing the operation with the Z operation on the sign qubit of the integer registers. This is the oracle.
  - **Block D:** A block containing the Grover diffusor.
- **Step 4:** Build the following circuit described with blocks: $A \rightarrow (O \rightarrow A^{-1} \rightarrow D \rightarrow A)^k \rightarrow$ measurement of the score.
- **Step 5:**
  - **If** the score is higher than v, set v to this score. Go to Step 2.
  - **Else**, set $K_{\max} = \lceil 1.25 \cdot K_{\max} \rceil$. If you did not fail to find a new maximal value in the last t tries, go to Step 2. Otherwise stop the algorithm.

Quantum Weighted Max-SAT evaluation step

- **Step 0:** Let :
  - v be the number of variables of the CNF formulation of the problem.
  - c be the number of clauses of the problem.
  - r be the number of bits required to represent the sum off all the weights of all the clauses in the CNF formulated problem using a signed integer.

  The number of qubits required by the evaluation step is $v + r$.
- **Step 1:** Perform an initial Hadamard operation on the v registers representing qubits to create the superposition of all possible assignments of values to the variables.
- **Step 2:** Perform an initial QFT transformation on the r registers representing the sum of weights.
- **Step 3:** For each clause, consider the set $\mathcal{C}$ of variables in the clause. This is the control set of the clause. Let's also consider w the weight associated to the clause. Then test the truthiness of the clauses:
  - **Substep 1:** Invert the qubits corresponding to non-negated variables in the clause with an X operation.
  - **Substep 2:** For $i : 0 \to r-1$, if the rest of the division of w by 2^i is 1, apply a phase operation of angle $\lambda = \frac{\pi}{2^j}$ then multicontrolled phase operation with control set $\mathcal{C}$ and angle $\mu = \frac{-\pi}{2^j}$ on the jth qubit of the integer register for $j : 0 \to k-i-1$.
  - **Substep 3:** Invert back the qubits corresponding to non-negated variables in the clause with an X operation.
  - **Substep 4:** Repeat from substep 1 for the next clause.

In this pseudo-code, we can find the following bricks of the algorithm:

- **The clause verifiers:** we check if the clauses are true or false, and add to the sum of weights registers the correspoding weights.
- **The Grover Adaptive Search:** we perform a GAS to find the maximum sum of weights using the clauses verifiers during the preparation of the superposition states.

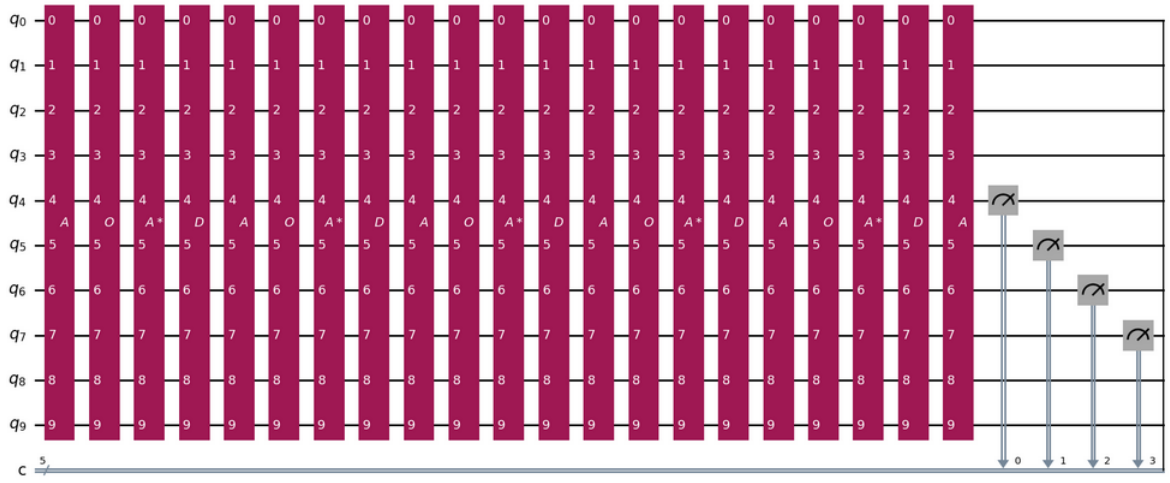

Figure 4.1: Example of an iteration of the GAS for Weighted Max-SAT

4.3 Implementation

At the opposite of classical algorithms that can tackle the same problem with different inputs all the time the same way (i.e. we don't need to modify the code of an algorithm if we just change some data), when dealing with quantum algorithms, we must create the right circuits given the data in input.

That means that for the same number of clauses and the same number of variables, we will get two different sets of circuits to build to solve our problem if the clauses are different.

The circuit corresponding to our evaluation step any search would look like this:

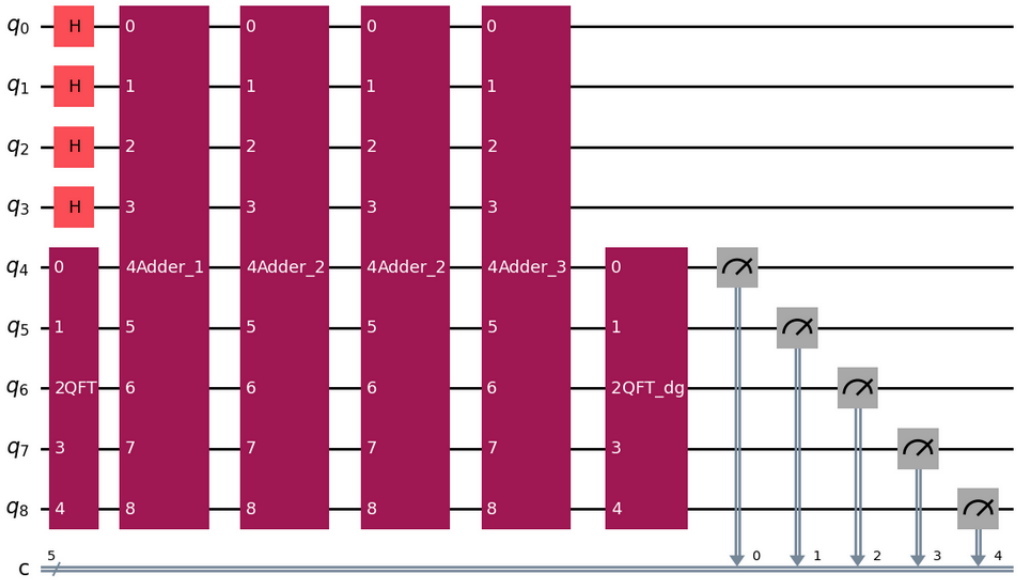

Figure 4.2: High-level Weighted Max-SAT block

The figure 4.2 pictures a problem where we have 4 clauses and 4 variables. We can see the 4 adder blocks corresponding to the 4 clauses, the first 4 qubits corresponding to the 4 variables, each one superposed using an Hadamard gate, the QFT block and the inverse-QFT block on the 4 last qubits of the circuit corresponding to the sum of the weights.

If we measure the 4 qubits as shown on the figure 4.2, we obtain a superposition of the possible sums of weights for each possible assignments of the different variables of the problem. For example, the circuit 4.2 corresponds to the problem for which we find the distribution of the sum of weights in figure 4.3. In this figure, we can see that the maximal score is 8 and that 62.5% of the assignments of values to the variables will lead to this score.

This means that if we run our algorithm, we will get with a high probability the value 8 as the optimum, and if we measure the variable registers, we will see one of the 9 out of 16 varariable assignments that yields a score of 8.

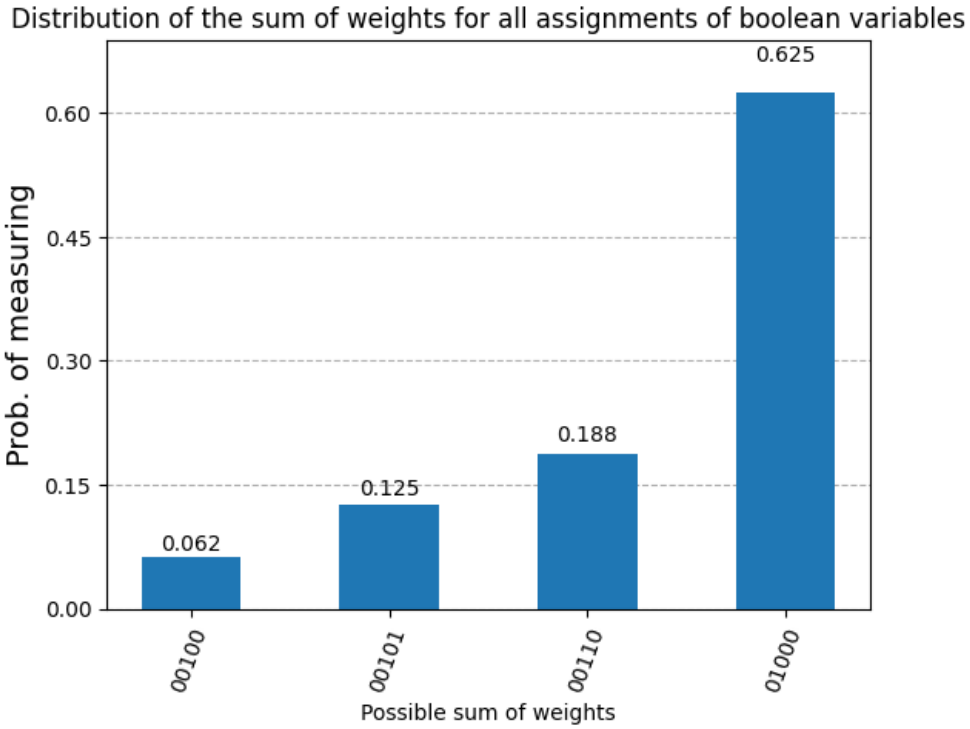

Figure 4.3: Distribution of the sum of the weights for a given instance

If we take a look inside an adder of the figure 4.2 like in figure 4.4, we can see that this is composed of two operations. First there are some X-gates at the start and the end of the block that corresponds to the variables in the clause that needs to be negated for the different QFT adding operations.

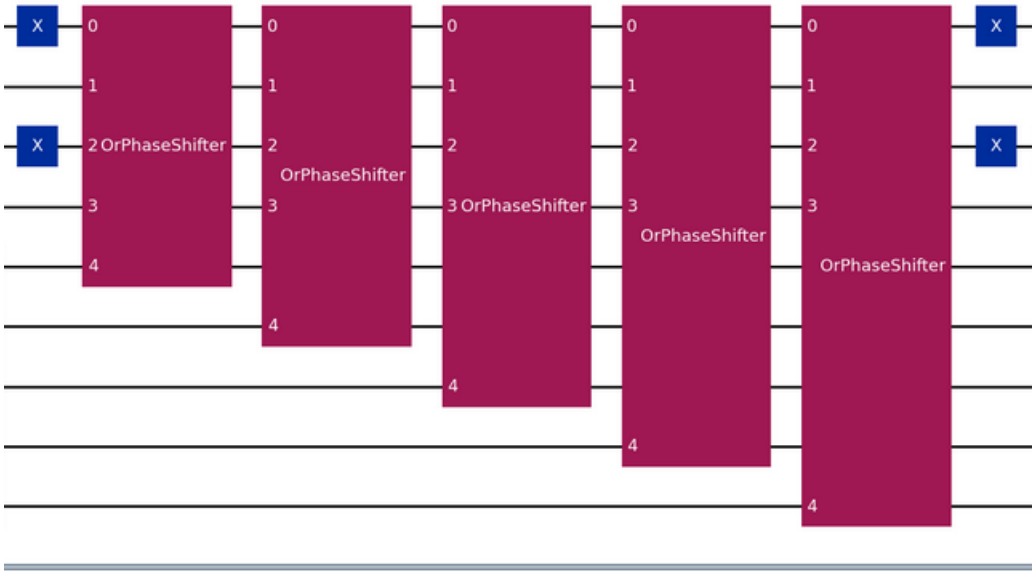

Figure 4.4: Decomposition of the adder block

A usual QFT adder performs multiple controlled phase operations from one integer register containing a state to another one. Here we adapted this to only perform the phase operations necessary for a given weight.

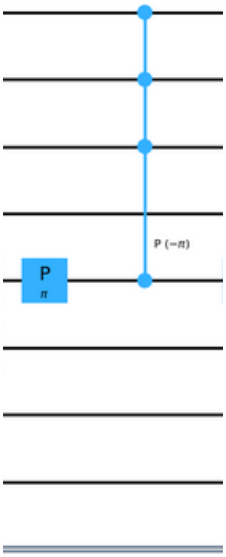

Figure 4.5: Controlled phase operation block in the adder

The phase operations are multicontrolled in our algorithm (see figure 4.5). The operation of adding a weight in the register should be done only if the clause is verified. The way we do this is by always applying the phase operation, then by reversing it if the conjunction of the negation of the variables in the clause in true (that means that the clause is false by the De Morgan's rule).

4.4 Complexity

In this section, we will focus on the complexity of our algorithm using the ideas presented before. In the following subsections, we will define n as the number of variables of our problem, and m as the number of clauses. We will also define k as the number of qubits needed to store the sum of all the weights of the problem.

This value can be obtained by:

$$k = 1 + \lfloor \log_2(\sum_{i=1}^{m} w_i) \rfloor$$

with w_i the weight of the ith clause. Usually, we will have $k \sim \mathcal{O}(\log m)$.

4.4.1 Complexity of the state preparation (adder block)

The implementation of the QFT that we use is quadratic in the number of qubits it transforms. We have for the QFT a complexity of:

$$\mathcal{O}(k^2) \sim \mathcal{O}(\log m)$$

In each adder block, the best way to evaluate and perform the addition with the lowest complexity is to evaluate the clause once with a multicontrolled X gate on an ancilla qubit, then use this ancilla qubit to perform the needed controlled phase operations for adding the weight into the adder. We learn in [**14**] that it is possible to create a multicontrolled X gate using only $\mathcal{O}(\log n)$ operations.

Then we need to perform $\mathcal{O}(k^2) \sim \mathcal{O}(\log m)$ operations maximum to add the weight. For each weigth, we perform

$$\mathcal{O}(\log m \log n)$$

operations.

We also have to perform X operations, but it is not meaningful in the complexity analysis since it is done twice in parallel by clause.

For one adder block, as we have m weights, we have a number of operations of:

$$\mathcal{O}(m \log m \log n)$$

4.4.2 Complexity of the Grover adaptive search

As presented in the chapter about QGS, a full GAS has a worst case complexity of

$$\mathcal{O}(TK_{\max}\sqrt{2^n})$$

where T is the number of provisional maxima found during the full search. As we perform potentially several adder blocks during an iteration, the global complexity of the algorithm is:

$$\mathcal{O}(TK_{\max}\sqrt{2^n} m \log m \log n) \sim \mathcal{O}(\sqrt{2^n} m \log m)$$

The algorithm is log-linear in the number of clauses, and exponential in the number of variables.

An NP-hard problem stays an NP-hard problem and it was expected to keep an exponential complexity despite the quantum improvements we purpose, although we are waiting a lot of improvements in practice.

4.5 Simulation and performance

A good metric to measure the complexity of a quantum circuit is its depth. As said before, quantum computers can only perform operations on one or two qubits. However, it is possible to run these operations in parallel as we can see on figure 4.6 where 5 operations are done at the same time. The depth depends highly on two things :

- The size of the circuit : the size of the circuit is the number of operations performed from the start to the end of a circuit.
- The number of controlled gates : using a controlled gate in the circuit implies an order depedency between different qubits, and sometimes some quantum registers will have to wait the operations of another one before performing the controlled X operation. We can already see that at the start of our circuit on figure 4.6.

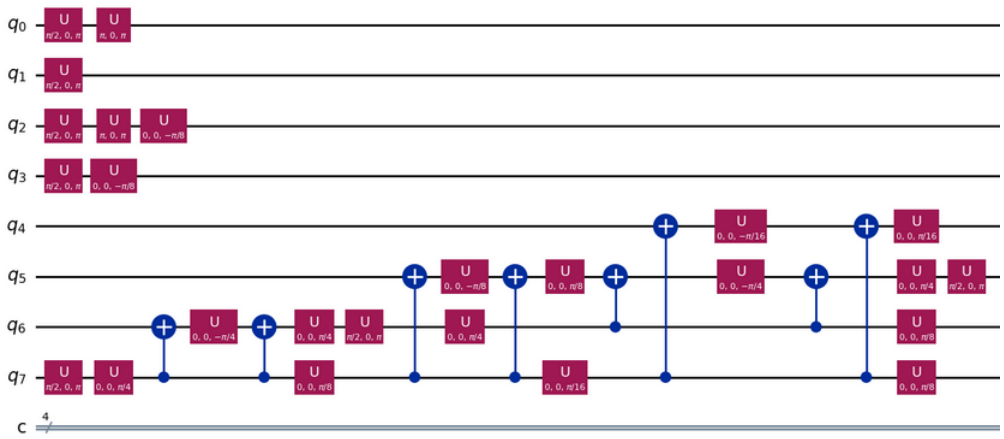

Figure 4.6: Start of the Weighted Max-SAT block, fully decomposed

In this section, we will provide an analysis of the total depth of the algorithm.

We will try to link the theorical complexity to our empirical results. Due to actual quantum computers performances and to the capacity of the simulators, we are not able to run instances as complex as classical solvers are able to solve. Futhermore, on small instances, the given dominant factors of the complexity may not be the effective dominant factors.

For every parameter that we will analyse, we will provide the results that we got by variating the number of clauses alone, then the number of variables alone, and see if the empirical results follow the theoretical ones. For every instance tested, we ran 5 times the algorithm and we provide a plot containing the mean of the results, as well as an indication of the variance if possible.

4.5.1 Compilation

When creating a quantum circuit, we tend to create it unoptimal in the sens that we want it understandable for a human. It exists different backends on which we can run quantum circuits, each one having its own specifications. For instance, a quantum computer in a city could be able to perform its controlled-X operations on different pairs of qubits than the one in another city.

It exists some compilation tools that rearrange the circuits to make them more optimal for a backend. We observed that compiling for the default fake backend (i.e. default simulator) decreases slightly the depth of the circuit by 10-15% while not fully loosing the time gained in the compilation process.

4.5.2 Depth

To check our theoretical results, we chose to run our algorithm 5 times on different created instances to generate the benchmarks.

As we can see on the figure 4.7, increasing the number of clauses seems to increase the average depth of the circuits, but this relation is not strict and it is difficult to interpret if the log-linear theorized relation holds or not.

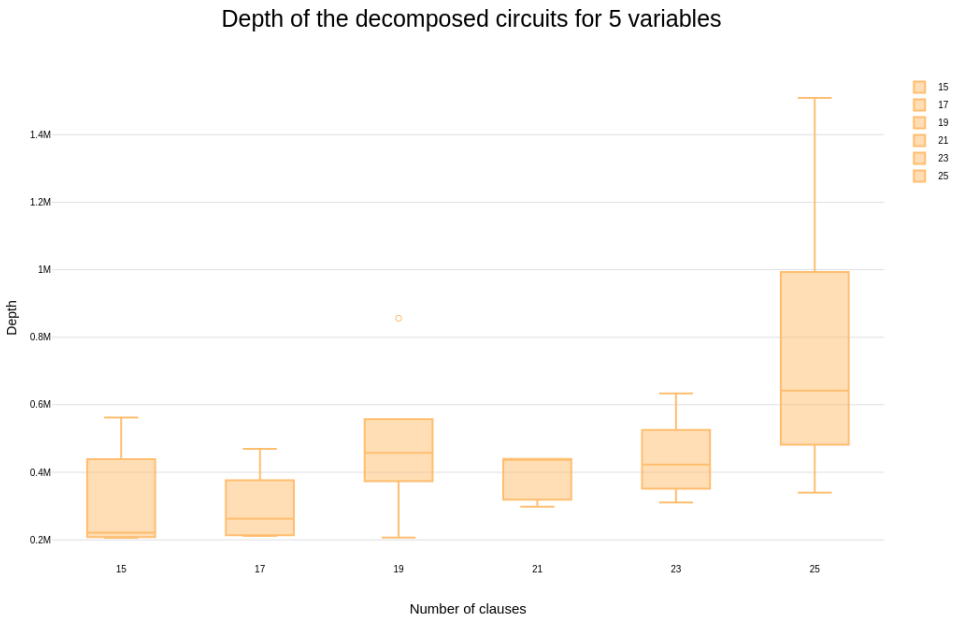

Figure 4.7: Sum of the depth of the circuits of the different iterations of the GAS, depending on the number of clauses

On figure 4.8, we can see that increasing the number of variables clearly increses exponentially the number of clauses as expected. We can also see that the variance of the observed depths tends to increse with the number of variables too.

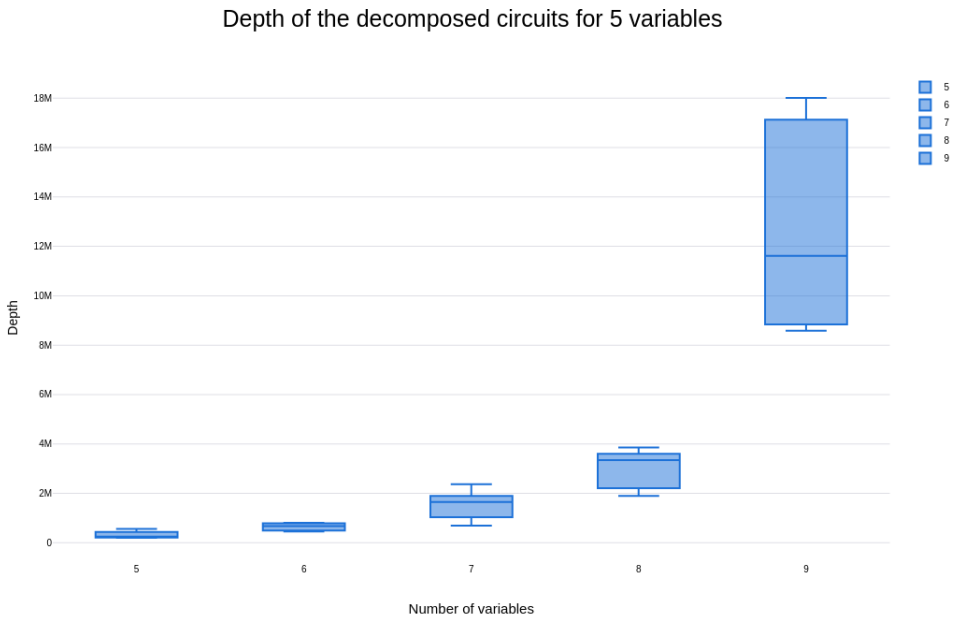

Figure 4.8: Sum of the depth of the circuits of the different iterations of the GAS, depending on the number of variables

4.5.3 Convergence of the optimal result

In every of our experiments, we observed a fast convergence of our algorithm, with a maximum of 2 intermediary steps before finding the optimal solution. Of course when the run was unlucky, we had to iterate more, thus it tends to increse the depth and computation time, but we got at the end the epected results.

4.6 To go further

The quantum implementation of the Weighted Max-SAT algorithm uses direct evaluation of clauses. This means that one could implement a framework where the clauses would not be anymore in CNF. Converting a problem into its CNF form is NP-hard [**15**], so if we have a given problem, we could gain a lot of time by not converting it. Another advantage would be that we can imagine some problems where weigths are assigned to clauses that are not in CNF, and transfer this information into the weights of CNF clauses could be difficult or would need approximations.

We could also use an alternative way to add the weights that is to add a qubit containing the truthness of a clause, and applying the controlled phase operation

from this qubit everytime instead of checking it for every phase operation with a multi-controlled manner. In the real life and at the moment, quantum computers can only perform operations on single qubits or perform single-controlled X-gate. The visual simplicity of a multi-controlled gate often hide a much higher complexity in the computation of this gate. Putting this additional qubit could make the code faster on real quantum computers, but it slows down the simulation. Indeed, every circuit corresponds to a unitary matrix of size $2^n \times 2^n$ if there are n qubits. The complexity of the simulation depends exponentially in n.

Chapter 5

Conclusion

In this thesis, we explored the significant potential of quantum algorithms to accelerate solutions for discrete optimization problems. We began with a code-oriented overview of foundational quantum computing principles and some essential quantum algorithms in Chapter 1 and 2, setting the stage for the advanced techniques discussed in the subsequent chapters.

Chapter 3 delved into the Quantum Grid Search (QGS) algorithm, a novel approach to derivative-free optimization or discrete optimization of real-valued functions. We demonstrated how this algorithm leverages quantum superposition and Grover's adaptive search to find the minimum of a quadratic polynomial function. The complexity analysis showed that the QGS algorithm achieves a quadratic speedup over its classical counterpart, with a worst-case complexity of $O(\sqrt{2^N})$ compared to the classical $O(2^N)$, with N being the number of feasible points in the minimization domain. This chapter highlighted the key steps in the algorithm, including creation of the state superposition and optimization of higher-degree polynomials, and provided a benchmark analysis to validate its theoretical efficiency.

Chapter 4 introduced the Quantum Weighted Max-SAT algorithm, which addresses the optimization of weighted satisfiability problems. We described the formulation of the problem in conjunctive normal form and presented our quantum solution, which also integrates Grover's search to achieve significant computational gains. The complexity analysis in this chapter indicated that this algorithm offers an improvement in speed over classical approaches to find an approximate of the optimal solution. Detailed steps and the implementation of the algorithm were provided, showcasing its ability to handle complex satisfiability problems efficiently.

The empirical convergence of both algorithms was simulated using Qiskit, providing practical insights into their performance. These simulations confirmed the theoreti-

cal speedups and demonstrated the algorithms' potential for handling larger, more complex problems that are infeasible with classical methods.

The algorithms developed and analyzed in this thesis represent progress in the field of quantum optimization. They offer potential solutions with theoretical quadratic speedups. However, given the current instability and early stage of quantum computers, further advancements in quantum hardware are necessary before we can fully validate and apply these results. Future research will be crucial to determine the practical utility of these algorithms as quantum computing technology continues to evolve.

Bibliography

[1] Daniel Koch, Laura Wessing, Paul M. Alsing (2019) *Introduction to Coding Quantum Algorithms: A Tutorial Series Using Qiskit*, arXiv:1903.04359 [quant-ph].

[2] *IBM Quantum* (2024) *IBM Quantum Learning*, `https://learning.quantum.ibm.com`, Accessed: 2024-01-18.

[3] Alyssa Noël (2022) *Classical Random Walks vs Quantum Walks*, Msc Thesis, UCLouvain - EPL.

[4] Michael A. Nielsen and Isaac L. Chuang (2010) *Quantum Computation and Quantum Information*, Cambridge University Press, 10th Anniversary Edition, pp. 216–234.

[5] Austin Gilliam, Stefan Woerner, Constantin Gonciulea (2021) *Grover Adaptive Search for Constrained Polynomial Binary Optimization*, *Quantum*, **5**, 428.

[6] Raphael Seidel et al. (2021) *Efficient Floating Point Arithmetic for Quantum Computers*, arXiv:2112.10537 [quant-ph].

[7] Lov K. Grover (1996) *A fast quantum mechanical algorithm for database search*, arXiv:quant-ph/9605043.

[8] Simanraj Sadana (2020) *Grover's search algorithm for n qubits with optimal number of iterations*, arXiv:2011.04051 [quant-ph].

[9] *IBM Quantum* (2024) *Fundamentals of Quantum Algorithms: Grover's Algorithm*, `https://learning.quantum.ibm.com/course/fundamentals-of-quantum-algorithms/grovers-algorithm`, Accessed: 2024-04-24.

[10] Geovani Nunes Grapiglia (2024) *Lecture 1 notes in LINMA2460: Nonlinear Programming*, Ecole Polytechnique de Louvain (UCL), February.

[11] R. Cleve, J. Watrous (2000) *Fast parallel circuits for the quantum Fourier transform*, *Proceedings 41st Annual Symposium on Foundations of Computer Science*, pp. 526–536, IEEE.

[12] `https://en.wikipedia.org/wiki/Maximum_satisfiability_problem`, Accessed: 2024-06-02.

[13] `https://en.wikipedia.org/wiki/Conjunctive_normal_form`, Accessed: 2024-06-02.

[14] Stefan Balauca, Andreea Arusoaie (2022) *Efficient Constructions for Simulating Multi Controlled Quantum Gates*, in *Computational Science – ICCS 2022*, Springer International Publishing, Cham, pp. 179–194.

[15] Grigori S. Tseitin (1968) *On the Complexity of Derivation in Propositional Calculus*, Structures in Constructive Mathematics and Mathematical Logic, Part II, Seminars in Mathematics (translated from Russian), pp. 115-125, Steklov Mathematical Institute.

[16] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. 4th ed., Pearson, 2020.

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**