



POLITECNICO DI TORINO

Master Degree Course in Computer Engineering

Master Degree Thesis

**Implementation and testing of
current quantum pattern
matching algorithms applied to
genomic sequencing**

Supervisor

Prof. Bartolomeo Montrucchio

Candidate

Raffaele PALMIERI

Company Supervisors at LINKS

Dr. Olivier Terzo

Dr. Andrea Scarabosio

Dr. Alberto Scionti

Carmine D'Amico

ACADEMIC YEAR 2018/2019

Summary

Quantum Computing is a new and mostly unexplored research field that promises to revolutionize the way of computing information for those problems that still need a lot of time and resources for classical computers to resolve them. An example of such problems is *Genome Sequencing and Alignment*, in which genomes are sequenced by means of sequencers machines, obtaining short reads that are then assembled together to obtain the whole genome. In particular, *Grover's algorithm* is a quantum search algorithm that could find application in the context of sequence alignment. In this thesis we analyzed the most important and recent *Quantum Pattern Matching* algorithms developed for *Genome Sequencing*, all based on the fundamental *Grover's algorithm*, implementing them in Python 3 by means of the *Qiskit* library (developed by IBM and giving free access to the IBM Quantum Computing cloud platform). So, we did some testing on the algorithms and did some final considerations about them and on the current and future status of Quantum Computing.

Contents

1	Introduction	7
1.1	Scope of the thesis	7
1.2	Organisation of the thesis	8
2	Why Quantum Computing?	9
2.1	A new way of computing information	9
2.1.1	Shor’s algorithm	10
2.1.2	Grover’s algorithm	10
2.1.3	Deutsch-Jozsa algorithm	10
2.1.4	Obstacles that separates us from quantum supremacy	11
2.2	Quantum Computing: basic concepts	12
2.2.1	Qubits and quantum states	13
2.2.2	Quantum gates and quantum circuits	17
2.3	Real quantum computers and quantum simulators	19
2.3.1	Technologies for real quantum computers	21
2.3.2	Quantum simulators	22
2.3.3	Quantum Assembly	23
2.4	Tools used	24
3	An overview of Genome Sequencing	26
3.1	An introduction to the problem	26
3.2	DNA Structure	26
3.3	Genome Sequencing	27
3.3.1	Sequencing methods: an historical perspective	28
3.3.2	From Sequencing to Reconstruction	29
3.4	DNA Alignment	30
3.4.1	Exact matching methods	31
3.4.2	Approximate matching methods	34
3.5	DNA Big Data	36

4	Quantum Pattern Matching algorithms	38
4.1	Quantum search applied to Pattern Matching	38
4.2	The fundamental Grover's Algorithm	38
4.2.1	The Oracle	39
4.2.2	The Algorithm	39
4.3	Quantum Associative Memory	44
4.3.1	Arbitrary amplitude distribution initialization	44
4.3.2	The Algorithm	46
4.4	Fast Quantum Search based on Hamming distance	47
4.5	Quantum indexed Bidirectional Associative Memory	48
4.5.1	Quantum Associative Memory with Distributed Queries	49
4.5.2	The QiBAM model	52
4.6	Quantum Pattern Recognition	53
5	Implementation and Testing	56
5.1	A brief introduction to the implementation and testing work	56
5.1.1	Tools used	56
5.1.2	How testing was done	56
5.1.3	Chosen algorithms	58
5.1.4	Decoherence and number quantum of gates	59
5.2	Quantum Associative Memory	60
5.3	Quantum indexed Bidirectional Associative Memory	63
5.4	Quantum Pattern Recognition	67
5.5	A brief summary about analyzed algorithms	71
5.6	An example of noisy quantum computation	71
5.6.1	Noise, number of gates and circuit depth	74
6	Conclusions and future developments	76
6.1	Present and Future of Quantum Computing	76
A	Quantum Associative Memory	78
A.1	Python code	78
A.2	Other tests	85
A.3	Verifying the validity of the implementation	87

B Quantum indexed Bidirectional Associative Memory	89
B.1 Python code	89
B.2 Other tests	97
B.3 Verifying the validity of the implementation	99
C Quantum Pattern Recognition	101
C.1 Python code	101
C.2 Other tests	111
Bibliography	115

Chapter 1

Introduction

1.1 Scope of the thesis

In this thesis we will talk about the basic concepts of *Quantum Computing* and its application to a specific field, that is *Genome Sequencing*. *Quantum Computing* is a new and potentially revolutionary way of computing information, that finds its strongest advantage in the concept of *Quantum Parallelism*. Thanks to the peculiar characteristics offered by Quantum Computing, a wide variety of problems requiring a lot of time to be resolved (even to the point of becoming practically unapproachable with classical computation) becomes approachable. *Genome Sequencing* is one of the fields that could benefit a lot from the application of Quantum Computing: it requires the computation of a very big amount of data coming from genomes belonging to a wide variety of living beings, so it's an important and interesting field for Quantum Computing application (for medical and biological reasons).

In order to talk in more depth about Quantum Computing applied to Genome Sequencing, we examined some *Quantum Pattern Matching* algorithms, choosing the most recent and, in our opinion, promising ones; we implemented these algorithms using the *Qiskit* library in Python 3, developed by IBM for free access to the *IBM Quantum Experience* cloud platform and giving us the chance for testing these algorithms and doing some considerations about the obtained results. In the end, some final considerations about the current and future status of Quantum Computing field will be given.

1.2 Organisation of the thesis

The thesis is organised in chapters as follows:

- in *Chapter 2* we will talk about the idea of Quantum Computing and the companies involved in this mostly unexplored research field; after that, we will give some basic concepts about Quantum Computing;
- in *Chapter 3* we will talk about the problem of Genome Sequencing, giving an historical perspective and mentioning the most famous classical algorithms for pattern matching used in the Genome Sequencing and Alignment field;
- in *Chapter 4* we will introduce the chosen Quantum Pattern Matching algorithms for implementation and testing, giving some mathematical insights about them;
- in *Chapter 5* we will introduce the obtained results, doing also an important consideration about *noise* in Quantum Computing;
- in *Chapter 6* we will give some final considerations about the future of Quantum Computing.

Chapter 2

Why Quantum Computing?

2.1 A new way of computing information

“Nature isn’t classical, dammit, and if you want to make a simulation of nature, you’d better make it quantum mechanical, and by golly it’s a wonderful problem, because it doesn’t look so easy.”

With these words, pronounced in 1981 during a conference co-organized by MIT and IBM [1], Richard Feynman encouraged the scientific community to exploit the peculiar characteristics of quantum mechanics in order to give birth to a new way of computing information. Today’s computing platforms are called *classical computers*, to distinguish them from *quantum computers*. The interest towards this unexplored branch of computer science is big and is growing: why is Quantum Computing so fascinating? To answer this question, we need to briefly give some examples of the potentially disruptive performance of a quantum computer, comparing it to a classical computer. We will also talk about the reasons not to be over-hyped about Quantum Computing: important physical obstacles are yet to be overcome if we want to build powerful quantum computers with a big number of qubits and capable of doing error correction. After that, we will introduce some basic concepts of Quantum Computing.

2.1.1 Shor's algorithm

Shor's algorithm is maybe the best example to give an idea of the potential of Quantum Computing. This algorithm is used to find the prime factorization of a number: given a number, it can find efficiently two prime integers which, when multiplied, give the original number. It's important to remark that many modern cryptography systems (like RSA) rely on the fact that, while computing the product of two very large prime numbers is quick, figuring out which very large prime numbers multiplied together yield to a certain product is time consuming: RSA picks prime numbers so large that it would take more than a quadrillion years on a classical computer to discover them. On the other hand, Shor's algorithm can run on a quantum computer in polynomial time, with $\mathcal{O}(d^3)$ complexity (where d is the number of decimal digits of the integer to factor), against the $\mathcal{O}(\exp(d^{\frac{1}{3}}))$ of the best-known classical algorithm [2].

2.1.2 Grover's algorithm

Grover's algorithm is an example of quantum algorithm used for unstructured searching of data. Suppose you have a list of N items: between these items, there is one that has a unique property and that we wish to locate. To find this item with classical computation, one would have to check an average of $N/2$ items, or N items in the worst case. On the other hand, with Grover's algorithm running on a quantum computer, we can find the desired item in \sqrt{N} steps: this is a quadratic speed-up that captures our interest. This algorithm will be discussed in more depth in later chapters of this thesis [2].

2.1.3 Deutsch-Jozsa algorithm

This algorithm is an example of how *Quantum Parallelism* can be exploited to resolve a problem better than a classical computer. It resolves the following problem: suppose we are given a function $f : \{0, \dots, 2^n - 1\} \rightarrow \{0,1\}$, which takes as input a number x from 0 to $2^n - 1$ and outputs 0 or 1. The function itself could be constant for all values of x or balanced, *i.e.* equal to 1 for exactly half of all the possible x and 0 for the other half. Our goal is to discover the nature of the function itself, *i.e.* if it is constant or balanced. *A quantum computer can solve this problem with one evaluation of the function f* , compared to the classical counterpart that needs $2^n/2 + 1$ evaluations [3].

2.1.4 Obstacles that separates us from quantum supremacy

Quantum supremacy is when a Quantum Computing algorithm offers better performance compared to the best possible algorithm on a classical computer, and the speed-up offered by the quantum algorithm is demonstrated on a real quantum computer. Attempts were made to demonstrate quantum supremacy: it's worth to mention the contribution that came from Google, that published a research paper in 2019 in which the efficiency of a quantum computer with 53 qubits in executing a task that on a classical computer would take approximately 10000 years is demonstrated [4][5]; a few weeks after Google's research was published, IBM had some criticism to do about it, saying that “*an ideal simulation of the same task can be performed on a classical system in 2.5 days and with far greater fidelity*” [6]. However, some problems are yet to be resolved: to be able to demonstrate quantum supremacy, we need better hardware. When talking about real quantum computers, we need two distinguish them in two categories:

- *Universal gate quantum computers*: they can be used for general purpose computation; an example is the *IBM Quantum Experience* platform, accessible via cloud, that offers up to 14 qubits for free; it's worth to notice that the most powerful quantum computer by IBM offers 53 qubits, but it's not freely accessible at the time of writing this thesis; Google also is involved in this field and has built a quantum computer with 72 qubits;
- *Quantum Annealing*: it's a metaheuristic for finding the global minimum of a given objective function over a given set of candidate solutions by a process using quantum fluctuations; Quantum Annealing is used mainly for problems where the search space is discrete (combinatorial optimization problems) with many local minima [7]; quantum annealers like those built by D-Wave offer a larger number of qubits (even 5000 qubits), but they are optimized for special tasks (they are an example of *Special Purpose Quantum Computing*).

Many ways are exploited in order to realize a qubit. For example, D-Wave uses superconducting qubits (also called *SQUID*, *Superconducting QUantum Interference Device*): a superconducting qubit is organized as a structure that encodes two states as tiny magnetic fields, which either point up or down [8]. Another way to physically realize qubits is with optical techniques, that is electromagnetic radiation: simple devices like mirrors and beam splitters can be used to do elementary manipulations of photons, but a major difficulty is encountered when we want to produce single photons on demand. A way to produce photons “every now and then” at random moments was found by scientists and waiting for such an event to happen is necessary. Alternative techniques were developed, like schemes based upon methods for trapping different types of atoms or based upon *Nuclear Magnetic Resonance*. As we can see, we need to develop and optimize physical techniques to realize qubits.

A first achievement that we desire to reach is to have more stable qubits in order to exploit the possibility of a qubit to be in *entanglement* with other qubits. We want the power to choose which qubits are involved in an entanglement and

which are not, but it's necessary to isolate our computation from outside influences. *Entanglement* is a property we will talk about in more depth in the next paragraph; here we will talk about *coherence* as a measure of how well a quantum system is isolated and the problem of *decoherence*. *Coherence* is a property of waves: if two waves are coherent with each other, they can in some senses “work together”. Coherence can be any correlation between the physical quantities of a wave, or a group of waves, and this correlation proves that they are “working together”. This is important in Quantum Computing as quantum mechanics helps us understand that the physical nature of a qubit can also be represented by a wave, and so, if we want different qubits to “work together”, they will need to be coherent; coherent waves have the same frequency and a constant phase difference. Unfortunately, there is a problem: if we start out with coherent qubits in a real quantum computer, they will not remain coherent forever. Qubits interact with the environment and this process causes the correlations they had due to coherence (which we could need to represent the results of our computation) to be lost, so the information from our computation degrades or is lost. This loss of information due to noise from the environment is also called *decoherence* [2].

Also, we need to address the noise problem: the theory of quantum error-correcting codes suggests that quantum noise is a practical problem that needs to be resolved. On the other hand, quantum noise is not a fundamental problem of principle: in particular, there is a *threshold theorem* for quantum computation that, roughly speaking, states that provided the level of noise in a quantum computer can be reduced below a certain constant “threshold” value, quantum error-correcting codes can be used to push it down even further, in exchange of a small overhead in the complexity of computation. This theorem makes some broad assumptions about the nature and magnitude of the noise occurring in a quantum computer and the architecture available for performing quantum computation but, provided those assumptions are satisfied, the effects of noise can be made essentially negligible for quantum information processing [3].

As we can see, Quantum Computing represents an exciting and mostly unexplored field of research, although we need not to be over-hyped: we have reasons to be optimistic about Quantum Computing, but important problems need to be addressed.

2.2 Quantum Computing: basic concepts

Now the basic concepts of Quantum Computing will be introduced: we will start from the concept of qubit, then we will be able to talk about superposition and measurement of qubits, the Bloch Sphere representation, quantum gates, quantum circuits, quantum states and entanglement.

2.2.1 Qubits and quantum states

The qubit is the quantum equivalent of a classic bit. A classic bit is a binary piece of information that can assume two possible values: 0 or 1. On the other hand, a qubit can be expressed as follows:

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle \quad (2.1)$$

We can see that *Dirac notation* is used to mathematically express the qubit: this notation is a common notation for quantum states, *i.e.* vectors in a complex *Hilbert space* [9]. A *Hilbert space* is a complex inner product space that is also a complete metric space with respect to the distance function induced by the inner product [10]. Dirac notation is also known as *bra-ket notation*. An example of complete bra-ket notation is the following:

$$\langle\phi|\psi\rangle \quad (2.2)$$

$|\psi\rangle$ represents the *ket* part of the notation and denotes a vector in an abstract (usually complex) vector space V , while $\langle\phi|$ represents the *bra* part of the notation and denotes a co-vector in the dual vector space V^ν . A complete bra-ket notation denotes an inner product. To denote a quantum state, only the ket part of the notation is used. For example, to represent the classical values of 0 and 1 in a quantum state, we can write the following:

$$|0\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \quad (2.3)$$

$$|1\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \quad (2.4)$$

The elements inside the column vector represent the *amplitudes* of the quantum state. The number of amplitudes inside a vector that represents a quantum state are 2^n , where n is the number of qubits representing the quantum state. In a more general form, we can write the following for a single qubit:

$$|\psi\rangle = \begin{pmatrix} \alpha \\ \beta \end{pmatrix} = \alpha|0\rangle + \beta|1\rangle \quad (2.5)$$

The coefficients α and β are the amplitudes, and their squared values represent *the probability of obtain 0 or 1 when we measure the qubit*. It's important to understand that the qubit assumes a value only when we decide to measure it: until then, it exists in a superposition of different states. This is the first important difference from a classic bit, that can be only 0 or 1. As already seen, we can express classic bits as qubits: $|1\rangle$ represents the classic 1 bit, with 100% probability of measuring 1, and $|0\rangle$ represents the classic 0 bit, with 100% probability of measuring 0. In Equation 2.5, $|0\rangle$ and $|1\rangle$ represent the *computational basis states*, and the sum of the squared absolute values of the amplitudes is equal to one: $|\alpha|^2 + |\beta|^2 = 1$. One

or more qubits form a *quantum state*, that can be represented as a column vector containing all the amplitudes associated with each computational basis state (as already seen). We can illustrate another example with a quantum state composed by two qubits:

$$|\psi\rangle = \begin{pmatrix} \alpha \\ \beta \\ \gamma \\ \delta \end{pmatrix} = \alpha|00\rangle + \beta|01\rangle + \gamma|10\rangle + \delta|11\rangle \quad (2.6)$$

A qubit that lives in a perfect superposition between $|0\rangle$ and $|1\rangle$ states can be expressed as follows:

$$\frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle \quad (2.7)$$

In this situation, the probability of measuring 0 or 1 are identical (50% for both). The best way to visualize graphically a qubit is the *Bloch Sphere* (Figure 2.1).

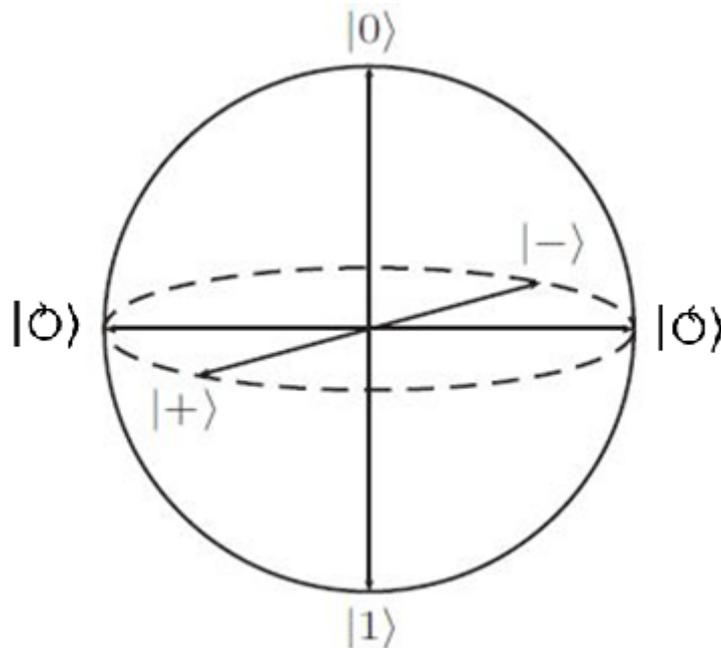


Figure 2.1. Bloch Sphere [3].

With the Bloch Sphere we can see the position of a single qubit, but it cannot be used in case of multi-qubit states. It is still useful to understand the nature of a qubit and the effects that quantum gates have on it. As already said, the states $|1\rangle$ and $|0\rangle$ are the quantum representation of the classic 1 and 0 and they are positioned on the z axis of the sphere (that has a radius equal to 1), but there

are other important basis states that need explanation, like the $|+\rangle$ and $|-\rangle$ states, that can be mathematically expressed as follows:

$$|+\rangle = \frac{|0\rangle + |1\rangle}{\sqrt{2}} \quad (2.8)$$

$$|-\rangle = \frac{|0\rangle - |1\rangle}{\sqrt{2}} \quad (2.9)$$

These states are positioned on opposite sides of the x axis of the sphere and they represent the perfect superposition between $|0\rangle$ and $|1\rangle$. The last two computational basis that is useful to explain are the $|\circlearrowleft\rangle$ and $|\circlearrowright\rangle$ basis state. They can be expressed as follows (i represents the imaginary unit):

$$|\circlearrowleft\rangle = \frac{|0\rangle + i|1\rangle}{\sqrt{2}} \quad (2.10)$$

$$|\circlearrowright\rangle = \frac{|0\rangle - i|1\rangle}{\sqrt{2}} \quad (2.11)$$

These states are positioned on opposite sides of the y axis of the sphere. It's worth to mention a set of interesting quantum states that are responsible for some interesting consequences in Quantum Computing. These states are called *Bell states*, or *EPR states*, or *EPR pairs*: they can be expressed as follows:

$$|\beta_{00}\rangle = \frac{|00\rangle + |11\rangle}{\sqrt{2}} \quad (2.12)$$

$$|\beta_{01}\rangle = \frac{|01\rangle + |10\rangle}{\sqrt{2}} \quad (2.13)$$

$$|\beta_{10}\rangle = \frac{|00\rangle - |11\rangle}{\sqrt{2}} \quad (2.14)$$

$$|\beta_{11}\rangle = \frac{|01\rangle - |10\rangle}{\sqrt{2}} \quad (2.15)$$

Let's take as example $|\beta_{00}\rangle$ to show an interesting phenomenon: upon measuring the first qubit, one obtains two possible results, *i.e.* 0 with probability 50%, leaving the post-measurement state $|\varphi'\rangle = |00\rangle$, and 1 with probability 50%, leaving $|\varphi'\rangle = |11\rangle$. *As a result, a measurement of the second qubit always gives the same result as the measurement of the first qubit.* That is, *the measurement outcomes are correlated.* Indeed, it turns out that other types of measurements can be performed on the Bell states, by first applying some operations to the first or second qubit, and that interesting correlations still exist between the result of a measurement on the first and second qubit. These correlations have been the subject of intense interest ever since a famous paper by Einstein, Podolsky and Rosen, in which they first pointed out the strange properties of states like the Bell states. *EPR's insights*

were taken up and greatly improved by John Bell, who proved an amazing result: *the measurement correlations in the Bell states are stronger than could ever exist between classical systems*. These results were the first intimation that quantum mechanics allows information processing beyond what is possible in the classical world and are an emblematic example of the entanglement phenomenon [3].

Now, suppose we are interested in a composite quantum state made up of two (or more) distinct states. To know how we should describe the composite state, we can rely on the following postulate that describes how the state space of a composite system is built up from the state spaces of the component systems.

“The state space of a composite physical system is the tensor product of the state spaces of the component physical systems. Moreover, if we have systems numbered 1 through n , and system number i is prepared in the state $|\psi_i\rangle$, then the joint state of the total system is $|\psi_1\rangle \otimes |\psi_2\rangle \otimes \dots \otimes |\psi_n\rangle$.”

The tensor product is a way of putting vector spaces together to form larger vector spaces. Suppose V and W are vector spaces of dimension m and n respectively. We also suppose that V and W are Hilbert spaces. Then $V \otimes W$ is an mn dimensional vector space. The elements of $V \otimes W$ are linear combinations of tensor products $|v\rangle \otimes |w\rangle$ of elements $|v\rangle$ of V and $|w\rangle$ of W [3]. For example, we can compute the tensor product of two generic quantum states as follows:

$$|\psi\rangle = \begin{pmatrix} \alpha \\ \beta \end{pmatrix} = \alpha|0\rangle + \beta|1\rangle \quad (2.16)$$

$$|\phi\rangle = \begin{pmatrix} \gamma \\ \delta \end{pmatrix} = \gamma|0\rangle + \delta|1\rangle \quad (2.17)$$

$$|\psi\rangle \otimes |\phi\rangle = \begin{pmatrix} \alpha \\ \beta \end{pmatrix} \otimes \begin{pmatrix} \gamma \\ \delta \end{pmatrix} = \begin{pmatrix} \alpha \begin{pmatrix} \gamma \\ \delta \end{pmatrix} \\ \beta \begin{pmatrix} \gamma \\ \delta \end{pmatrix} \end{pmatrix} = \begin{pmatrix} \alpha\gamma \\ \alpha\delta \\ \beta\gamma \\ \beta\delta \end{pmatrix} = \alpha\gamma|00\rangle + \alpha\delta|01\rangle + \beta\gamma|10\rangle + \beta\delta|11\rangle \quad (2.18)$$

Another example is given by computing the tensor product of states $|0\rangle$ and $|1\rangle$:

$$|0\rangle \otimes |1\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \otimes \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \begin{pmatrix} 0 \\ 1 \end{pmatrix} \\ 0 \begin{pmatrix} 0 \\ 1 \end{pmatrix} \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix} = |01\rangle \quad (2.19)$$

There are infinite points on the Bloch sphere, so paradoxically we could store the entire text of *The Divine Comedy* into a single qubit, but this is a misleading statement because of the behaviour of a qubit when it is measured: when observed, *it collapses* to 0 or 1. Nobody knows the reason why this happens and this behaviour is simply one of the fundamental postulates of quantum mechanics, but we can still manipulate qubits in order to exploit qubits for computation. We can do this with quantum gates [3].

2.2.2 Quantum gates and quantum circuits

As in classic computing, also in Quantum Computing we can use a variety of quantum gates to manipulate qubits. Since a quantum state can be represented as a vector, these gates can be represented as matrices and we can mathematically visualize the effects of these gates with the product between the matrix and the column vector representing the quantum state. We can start from single qubit gates and describe their effects using the Bloch Sphere as reference.

The *Hadamard gate* (H gate) rotates the initial qubit by 180 degrees around the x axis and then 90 degrees around the y axis. An example of applying Hadamard gate is given by the following relations:

$$H|0\rangle = |+\rangle \quad (2.20)$$

$$H|1\rangle = |-\rangle \quad (2.21)$$

The *Pauli gates* are the X , Y and Z gates and they rotate the qubit they act on by 180 degrees. The rotation of the X gate is around the x axis, the rotation of the Y gate is around the y axis and the rotation of the Z gate is around the z axis. It's worth considering that the X gate is the quantum analogue of the classic *NOT* gate, and you can see it when applying this gate on $|1\rangle$ or $|0\rangle$:

$$X|1\rangle = |0\rangle \quad (2.22)$$

$$X|0\rangle = |1\rangle \quad (2.23)$$

The *Phase gate* (S gate) and the $\pi/8$ gate (T gate) rotate the qubit around the z axis about 90 degrees and 45 degrees respectively (why the $\pi/8$ gate isn't simply called $\pi/4$ gate it's for historical reasons). There are also the “*dagger*” gates S^\dagger and T^\dagger that rotate the qubit around the z axis by the same amount of degrees of S and T gates, but in the opposite direction.

The gates mentioned so far are single qubit gates. There are also multi-qubit gates that take more than one qubit as input. An important multi-qubit gate is the *Controlled NOT gate* (also known as *CNOT*): it takes two qubits as input, with one that is called *control* and the other *target*. When the control qubit is equal to $|1\rangle$, the target is subjected to a X gate. It is represented by the symbols in Figure 2.2.

The upper wire represents the control qubit, while the lower wire represents the target qubit. The *CNOT* is also called *Quantum XOR*, since the target can be viewed as the result of a *XOR* between control and target. This is an important gate, since is the building block for other gates: an example is the *SWAP gate*, that realizes the swapping between two qubits and is composed by three *CNOT* gates as illustrated in Figure 2.3.



Figure 2.2. Controlled *NOT* gate [3].

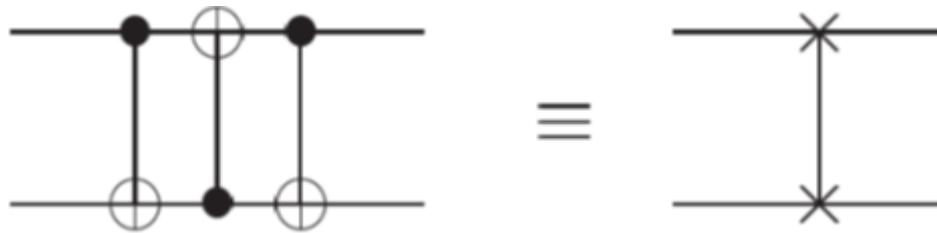


Figure 2.3. *SWAP* gate [3].

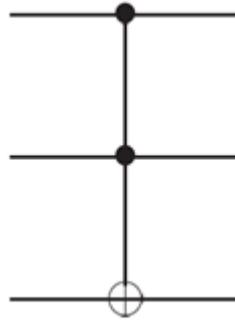
Another important multi-qubit gate that it's worth to mention is the *Toffoli gate* (also known as *Quantum AND*), that is a *CNOT* with two control qubits instead of only one. It is represented as illustrated in Figure 2.4.

There are other single-qubit and multi-qubit gates that we can summarize in Tables 2.1 and 2.2, together with the gates we have talked about so far: they contain their name, unitary matrix and circuit symbol. A square matrix U is *unitary* when the following relation is respected [11]:

$$UU^\dagger = I \tag{2.24}$$

It means that the matrix product between the square matrix U and its transpose and complex conjugate U^\dagger is equal to the identity matrix I . Tables 2.1 and 2.2 also contain some equations representing the existing relations between some of these gates.

These quantum gates are the building blocks for quantum circuits as classic gates are for classic logic circuits. An important characteristic of quantum gates (and of quantum computation) is that their effects are *reversible*: while classic gates are mostly irreversible (from the output of a *AND* gate we cannot recover the original inputs), with quantum gates we can reverse the computation. Applying the H gate to $|0\rangle$ gives us the $|+\rangle$ state and applying again the same gate gives us back the $|0\rangle$ state. This feature can be easily verified also with other quantum gates. Every quantum gate can be represented by a matrix that has the property to be unitary. For a generic single qubit gate U is possible to demonstrate that there exist real numbers α, β, γ and δ such that is possible to write the following

Figure 2.4. *Toffoli* gate [3].

(R gates are the *Rotation gates*, as shown in Table 2.1):

$$U = e^{i\alpha} R_z(\beta) R_y(\gamma) R_z(\delta) \quad (2.25)$$

In classical computing, a small set of gates (*AND*, *OR*, *NOT*) can be used to compute an arbitrary classical function, so we can refer to it as a *universal* set of gates for classical computation. As in classical computing, in Quantum Computing we can obtain a similar universality result for a set of quantum gates: we say that a set of quantum gates is *universal for quantum computation* if any unitary operation may be approximated to arbitrary accuracy by a quantum circuit involving only those gates. It's possible to demonstrate that H , $CNOT$, S and T gates form a universal set for quantum computation [3].

2.3 Real quantum computers and quantum simulators

Many companies are working on improving the technologies involved in the building of a real quantum computer. IBM has built a universal quantum computer with a 53 qubits capability. Another company is D-Wave, that builds special purpose quantum computers and has announced *Advantage*, a quantum computer with 5000 qubits. These computers have the size of a mainframe and they need to be kept in special conditions, like low temperatures. Although these companies are used to announce every little step towards a better Quantum Computing, the technologies are still young and need refinement. In this situation, quantum simulators are useful tools to simulate the behaviour of a quantum algorithm on a classic computer in order to have an idea of how the execution on a real quantum computer could be. It's worth to warn the reader that the term "*quantum simulator*" has two meanings: the first denotes a quantum machine that permits the study of quantum systems that are difficult to study in the laboratory and impossible to model with a supercomputer (so, they are special purpose devices designed to provide insight about specific physics problems); the second denotes a software running on a classic computer for simulating the behaviour of a quantum computer. In this thesis, when we refer to quantum simulators, we mean a software for Quantum Computing

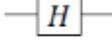
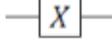
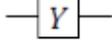
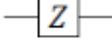
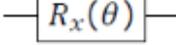
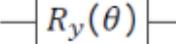
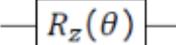
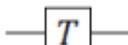
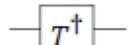
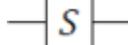
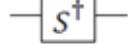
Name	Unitary	Circuit	Relations
Identity	$I = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$		
Hadamard	$H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$		
Pauli-X	$X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$		$X = iZY = HZH$
Pauli-Y	$Y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}$		$Y = iZH$
Pauli-Z	$Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$		$Z = iYH = HXH$
Rotation-X	$R_x(\theta) = \begin{pmatrix} \cos \frac{\theta}{2} & -i \sin \frac{\theta}{2} \\ -i \sin \frac{\theta}{2} & \cos \frac{\theta}{2} \end{pmatrix}$		
Rotation-Y	$R_y(\theta) = \begin{pmatrix} \cos \frac{\theta}{2} & -\sin \frac{\theta}{2} \\ \sin \frac{\theta}{2} & \cos \frac{\theta}{2} \end{pmatrix}$		
Rotation-Z	$R_z(\theta) = \begin{pmatrix} e^{-i\frac{\theta}{2}} & 0 \\ 0 & e^{i\frac{\theta}{2}} \end{pmatrix}$		
T	$T = \begin{pmatrix} 1 & 0 \\ 0 & e^{i\frac{\pi}{4}} \end{pmatrix}$		
T^\dagger	$T^\dagger = \begin{pmatrix} 1 & 0 \\ 0 & e^{-i\frac{\pi}{4}} \end{pmatrix}$		$TT^\dagger = I$
S	$S = \begin{pmatrix} 1 & 0 \\ 0 & i \end{pmatrix}$		$S = T^2$
S^\dagger	$S^\dagger = \begin{pmatrix} 1 & 0 \\ 0 & -i \end{pmatrix}$		$SS^\dagger = I$

Table 2.1. Common single-qubit gates.

simulation. Now we will talk more extensively about the technologies involved in real Quantum Computing and we will also talk about quantum simulators.

Name	Unitary	Circuit	Relations
<i>SWAP</i>	$SWAP = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$		$SWAP = CX_{01}CX_{10}CX_{01}$
Controlled- <i>NOT</i>	$CX = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$		$CX_{01} = H_0H_1CX_{10}H_1H_0X_1CX_{01} = CX_{01}X_1Z_0Z_1CX_{01} = CX_{01}Z_0$
Controlled-Phase	$CZ = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{pmatrix}$		$CZ = H_1CX_{01}H_1$
Toffoli	$CCX = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}$		$CCX_{012} = H_2CX_{12}T_2^\dagger CX_{02}T_2CX_{12}T_2^\dagger CX_{02}T_2T_1^\dagger H_2CX_{01}T_1^\dagger CX_{01}S_1T_0$
Controlled- <i>SWAP</i>	$CSWAP = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$		

Table 2.2. Common multi-qubit gates.

2.3.1 Technologies for real quantum computers

Ion trap qubits

Ion trap qubits technology uses radio-frequencies to confine ions. The ionic qubit corresponds to the two lowest electronic energy levels of an ion. Quantum gates are realized manipulating the ions with laser beams. A quantum algorithm starts with ions at grounded state, then single and two qubits gates are executed and finally the algorithm finishes with readout. During readout, the ion is hit with optical pulses: in this way, $|0\rangle$ state corresponds to a state that does not glow, while the $|1\rangle$ state becomes excited and corresponds to a state that does glow. A glowing ion is read as 1, while not glowing ion is read as 0. Ion trap qubits is the most remarkable technology for Quantum Computing: preparation and readout both can be done with fidelities of better than 0.999 where a fidelity of 1 signifies perfection [12].

Superconducting qubits

The idea behind *Superconducting qubits* technology is to create a network of connected artificial atoms, where each of them is realized as a nonlinear inductor-capacitor circuit. A superconductor enables an electric current in a circuit loop to circulate without resistance. A clockwise supercurrent generates a downward magnetic flux, while a counter clockwise supercurrent generates an upward magnetic

flux: superposing clockwise and counter clockwise supercurrents creates a superposition of up and down magnetic fields. This technology is used by both IBM and D-Wave for their quantum computers [12].

Photonic qubits

Photons can act as qubits and we can use various ways to represent two logical states (*e.g.*, presence or absence of the photon, polarization state of the photon, which path the photon is travelling, time of arrival of the photon). A way to prepare *photonic qubits* is *Spontaneous parametric down conversion*, in which a correlated pair of photons is produced simultaneously, but at random time (this means that when and where the photons are must be treated as being indeterminate). This indeterminacy determines errors in quantum computation that need fixing. This technology still needs a lot of refinement to be practical in Quantum Computing, but fortunately some improvements have been developed in the optical Quantum Computing field, with some small companies that are working in this segment (*e.g.* *Sparrow Quantum*, *Fathom Computing*, *Single Quantum*, *Quantum Opus* [13]) [12].

Topological qubits

Particles can be *bosons* or *fermions*. Fundamental matter (*e.g.*, electrons, quarks) are fermions, while bosons are responsible for fundamental quantum fields (*e.g.*, electromagnetism, nuclear forces). In two-dimensional space, a third type of particle can exist that is called *anyon*, that is a generalization of fermion and boson concepts. In a *topological quantum computer*, qubits are constructed from anyons. A big actor that is investing in this direction for Quantum Computing is Microsoft. Building a quantum computer with this technology requires semiconductor and superconductor components plus a magnetic field to produce a topological superconductor. Developing this technology still needs a big effort, which is compensated by the fact that topological Quantum Computing is expected to be especially robust against errors [12].

2.3.2 Quantum simulators

Though real quantum computers still need improvements in the technologies used to build them, we have seen that many big actors are researching in this field and they have been able to build interesting quantum computers that gave us the chance to experiment with running quantum algorithms on them and to carry on the research: we already mentioned IBM and Google with their state-of-the-art real quantum computers with 53 and 72 qubits respectively. Also, it's possible for any user to start experimenting with quantum computers via cloud for free: for example, IBM offers its *Quantum Experience* platform with freely accessible quantum computers up to 14 qubits and a quantum simulator up to 32 qubits. Speaking of quantum simulators, though we can experiment with real quantum computers, they still play an important role in testing quantum algorithms: we have to consider the fact that is not possible to access to certain data useful for testing purposes (like

the vector state of a quantum state) on a real quantum computer (any kind of measurement on qubits would cause collapsing). Another problem is that we would like to test a quantum algorithm that needs a number of qubits still unreachable for real hardware machines. Quantum simulators can simulate quantum states and quantum gates on a classical computer: realizing them could seem an easy task, since we can easily model interactions between quantum states and quantum gates in a mathematical way with vector and matrices, but it isn't if we take into account testing needs like observing the vector state or even simulate the noise, that in a real quantum computer would be present. It's important to remark that running quantum algorithms on simulators allow us to test them from an *accuracy* point of view: an evaluation from an execution time point of view would be possible only considering real quantum computers [14].

2.3.3 Quantum Assembly

Quantum assembly is the low-level language used by machines and simulators to apply quantum gates on qubits. Every actor working on the Quantum Computing field has its own version of quantum assembly. A common Quantum Assembly was proposed (*cQASM* [15]) to set a standard between quantum computers, so that every quantum computer could translate this intermediate QASM into its own executable QASM version (it is important to remark that it is just a proposal, not a totally accepted standard). An example of QASM language could be the following code, written in *OpenQASM* (IBM version of QASM).

```
OPENQASM 2.0;
include "qelib1.inc";

qreg q[12];
creg c[12];

h q[2];
cx q[1],q[2];
tdg q[2];
cx q[0],q[2];
t q[2];
cx q[1],q[2];
tdg q[2];
cx q[0],q[2];
t q[1];
t q[2];
cx q[0],q[1];
h q[2];
t q[0];
tdg q[1];
cx q[0],q[1];
```


- We can use a simple circuit composer online with a quantum register of 5 qubits (it is also possible to construct the desired circuit directly in OpenQASM and the online tool will draw the corresponding circuit);
- We can have access to real quantum computers up to 14 qubits and also to the IBM QASM simulator running on IBM servers;
- Using *Qiskit* (a library for Quantum Computing by IBM) we can write our programs in Python 3 and have access to the *IBM Quantum Experience* platform by simply using an access token, which will be given after creating an account on IBM website;
- It is also possible to use a local quantum simulator running on your local machine;
- Qiskit is an easy-to-use library that can count on good support by its developers and receives many updates (its documentation is also very good);
- Thanks to Qiskit, is easy to plot histograms of the results obtained by the execution of quantum algorithms and to draw the desired quantum circuit (using *matplotlib*, another Python library);
- It is possible to add a noise model when using quantum simulators, obtaining results that are closer to reality;

Together with Qiskit and the cloud Quantum Computing platform by IBM, *Jupyter Notebook* [17] was used as IDE to write our code from an Internet browser and run it; it was installed (together with Qiskit) using *Anaconda* [18] to create a new virtual environment where it was possible to install it.

Chapter 3

An overview of Genome Sequencing

3.1 An introduction to the problem

In this chapter we will talk about the problem of Genome Sequencing and some of the classical methods used to approach it; before talking about it, a few theoretical hints about DNA will be given in order to give a better understanding of the context.

3.2 DNA Structure

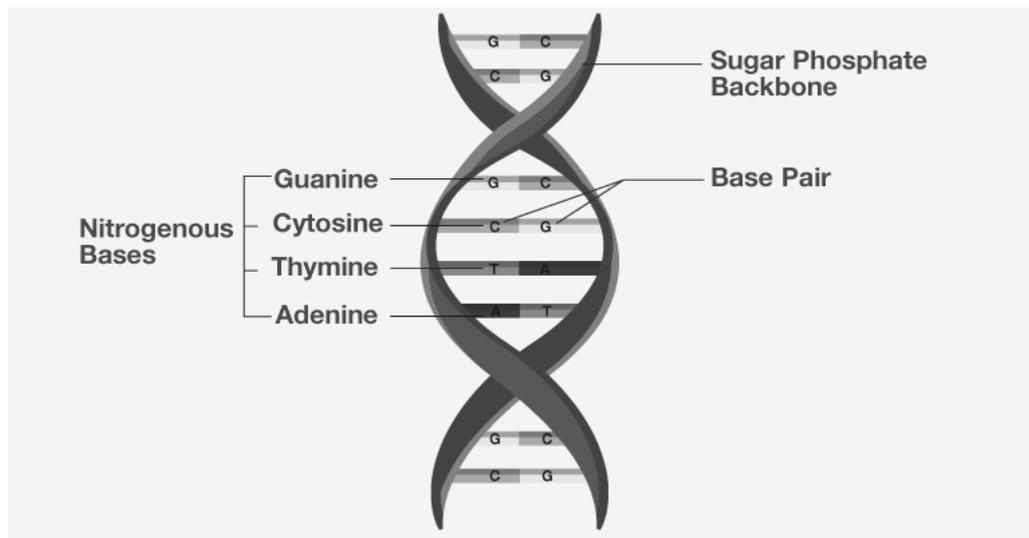


Figure 3.1. DNA Structure [19].

DNA stands for *Deoxyribonucleic Acid* and is a molecule with a double helix structure (as depicted Figure 3.1); it is made up by *nucleotides*, which are formed by three components: *sugar*, *phosphate groups*, and *nitrogen bases*. Sugar and

phosphate groups link the nucleotides together to form DNA strands, while *Adenine* (*A*), *Thymine* (*T*), *Guanine* (*G*) and *Cytosine* (*C*) are the four types of nitrogen bases. Inside the DNA structure the nitrogen bases are paired as follows: Adenine with Thymine (*A-T*) and Cytosine with Guanine (*C-G*). As we can see in Figure 3.1, these pairs link the two strands together. The order of the nitrogen bases encodes the genetic information inside DNA. Two important rules about DNA were discovered by Erwin Chargaff [20]; they are known as *Parity Rules*:

1. A double-stranded DNA molecule globally has percentage base pair equality: $\%A = \%T$ and $\%G = \%C$;
2. Both $\%A = \%T$ and $\%G = \%C$ are valid for each of the two DNA strands: this describes a global feature of the base composition in a single DNA strand.

As already said, DNA is responsible for transmitting the genetic information and it coils up to form a *chromosome*. *Chromosomes* are contained inside the *nucleus*. The necessary number of chromosomes to carry all the genetic information may vary with the considered species: *e.g.*, for human beings 46 chromosomes are needed.

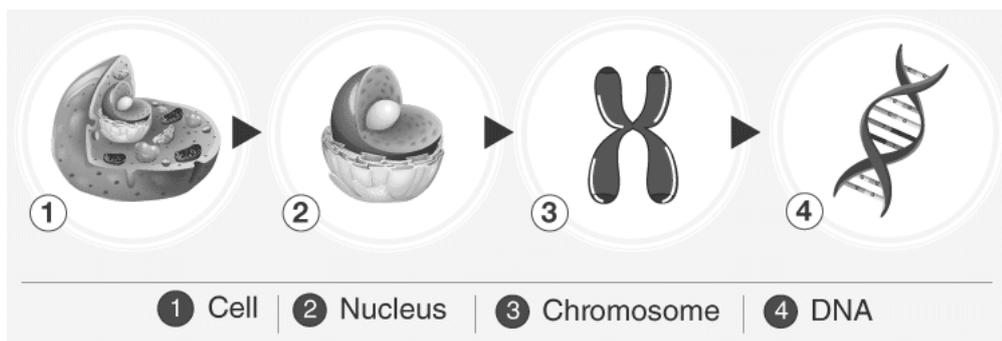


Figure 3.2. From DNA to cell [19].

3.3 Genome Sequencing

Genome sequencing is the process of determining the sequence of nucleotide bases (*A*, *T*, *C*, *G*) in a piece of DNA [21]. Sequencing an entire genome is not an easy task: human genome was completely sequenced only in 2003 with great international effort, ending a research project that started in 1990 (*Human Genome Project* [22]). Why would we want to sequence DNA? A reason could be a better treatment of human health: nowadays, researchers can compare long strings of DNA (even more than one million bases) from various individuals; with these comparisons, we could know an incredible amount of information about the role of inheritance in susceptibility to disease and in response to environmental influences [23]. Also, the ability to do these comparisons in the fastest, most accurate and cheapest way possible can give us more potential to diagnose diseases and propose therapies. Another field of application for genome sequencing is biology: we can use sequencing techniques in order to compare the genomes of different types of animals and organisms.

3.3.1 Sequencing methods: an historical perspective

DNA was first discovered by J. Watson and F. Crick in 1953, but at that time we weren't able to do sequencing on it: there were some strategies to infer the sequence of protein chains but they seemed not to apply well to DNA molecule, that is much longer and made of fewer units that were more similar to one another (which means that distinguish between them is harder). New strategies were needed to approach to the new problem. First attempts of sequencing were done on RNA (that is the acronym for *Ribonucleic acid*: like DNA, it is assembled as a chain of nucleotides, but unlike DNA it is more often found in nature as a single-strand folded onto itself, rather than a paired double-strand [25]), but the progress was slow: researchers were applying techniques used in analytical chemistry that could give the nucleotide composition, not the order. In 1965 Robert Holley, together with his colleagues, combined the above-mentioned techniques with selective ribonuclease treatments to obtain the first RNA fragments. At that time, also other researchers were working on the problem: Fred Sanger (together with its colleagues) developed a technique that was based on the detection of radiolabelled partial-digestion fragments after two-dimensional fractionation; it was with this technique that Walter Fiers obtained the first complete protein-coding gene sequence in 1972. From that moment, improvements followed and in 1975 Alan Coulson and Fred Sanger sequenced the first DNA genome using their “*plus and minus*” technique. However, also other two researchers, Allan Maxam and Walter Gilbert, were developing a new technique, called “*chemical cleavage*” technique, that was different from the “plus and minus” technique: it used chemicals to break the DNA chain at specific bases, instead of relying on DNA polymerase. Maxam and Gilbert's technique was the first to know large diffusion: this moment marked the birth of the so-called “*first generation*” genome sequencing.

Two years later, in 1977, an important improvement in the field came from the development of Sanger's “*chain-termination*” technique (or “*dideoxy technique*”): it had the advantages of being accurate, robust and easy to use, which determined its widespread diffusion and it was further improved in the following years. With these improvements, it was possible to develop and build the first genome sequencing machines, that could produce reads slightly less than one kilobase (*kb*) in length: this meant that researchers had to use auxiliary techniques together with these machines in order to analyse longer fragments.

In the following years, another technique was developed that became important for the birth of the second generation of DNA sequencers: it used a recently discovered luminescent method for measuring pyrophosphate synthesis. It was developed by Pal Nyren and colleagues and it had the advantages of being performed using natural nucleotides and observed in real time. This new technique was later licensed to *454 Life Sciences* (a biotechnology company founded by Jonathan Rothberg), that translated it into the first major successful commercial “*Next Generation Sequencing*” (*NGS*) technology. The sequencing machines produced by 454 had the main feature of greatly increasing the amount of DNA that could be sequenced in one run; they could produce reads around 400-500 base pairs (*bp*) long. The great sequencing power of these machines was the main reason behind the increased sequencing efforts that caused the completion of a single human's genome sequencing,

in a faster and cheaper way than previous sequencing methods.

After the success of 454 machines, other sequencing techniques were developed but it's worth to mention the contribution of *Solexa* (later acquired by *Illumina*) with its *Genome Analyzer* (*GA*) machines: they initially were able only of producing very short reads (up to 35 bp long) but also able to produce paired-end data (meaning that the sequence at both ends of each DNA cluster is recorded). This allowed to have better accuracy when mapping reads to reference sequences. After the standard Genome Analyzer version, *HiSeq* machine was developed, which had even bigger read length and depth. After HiSeq, *MiSeq* came, which had a lower throughput (but it was also cheaper) and longer read length.

Many actors appeared and disappeared on second generation genome sequencing machines market, in which the two major players were 454 and Solexa/Illumina. Together with these two companies, the third major role was played by *Applied Biosystems* (later known as *Life Technologies* after a merge with *Invitrogen*), that developed a sequencing technique by oligonucleotide ligation and detection (*SOLiD*); this technique wasn't able to produce the same read length and depth of Illumina machines, but had a better cost per base. The last technology of second generation that deserved attention was developed by Jonathan Rothberg after leaving 454: *Ion Torrent* was the first “*post-light sequencing*” technology, as it uses neither fluorescence nor luminescence.

All these improvements in the technologies and techniques used in genome sequencing allowed to change its cost and ease: the growth of power and capabilities of DNA sequencers had a very fast rate. Illumina machines were the most successful and gave the biggest contribution to second generation DNA sequencers.

Looking into the next generation genome sequencing, it's worth to mention the “*single molecule*” technology (*SMS*) developed at Stephen Quake's lab and commercially exploited by *Helicos BioSciences*. Helicos went into bankruptcy in 2012 but other companies continued to work on the next generation sequencers. A very diffused technology is the “*single molecule real time*” (*SMRT*) from *Pacific Biosciences*, available on *PacBio* machines) [24].

3.3.2 From Sequencing to Reconstruction

The sequencing process, executed through the sequencers we talked about in the previous paragraph, gives us a set of DNA reads in output that need to be assembled together in order to obtain the complete genome in one piece. There are two approaches to this matter:

- *de-novo assembly*: short reads are assembled in a full-length sequence without using a template;
- *reference-based assembly*: a previously assembled genome is used as a reference and sequenced reads are independently aligned against this reference sequence.

In terms of complexity and time requirements, de-novo assemblies are orders of magnitude slower and more memory intensive than mapping assemblies: this is mostly due to the fact that the assembly algorithm needs to compare every read with every other read (an operation that has a naive time complexity of $\mathcal{O}(n^2)$ [26]). Two types of algorithms are commonly used by de-novo assemblers:

- *Greedy algorithm assemblers*: they find local optima in alignments of smaller reads; early de novo sequence assemblers, such as *SEQAID* (1984) and *CAP* (1992) used greedy algorithms, such as *overlap-layout-consensus (OLC)* algorithms; these algorithms find overlap between all reads, use the overlap to determine a layout (or tiling) of the reads, and then produce a consensus sequence; some programs that used OLC algorithms featured filtration (to remove read pairs that will not overlap) and heuristic methods to increase speed of the analysis;
- *Graph method assemblers*: these assemblers were introduced at a *DIMACS workshop* in 1994 by Michael Waterman and Gene Myers; these methods represented an important step forward in sequence assembly, as they both use algorithms to reach a global optimum instead of a local optimum; the *De Bruijn graph* method has become the most popular in the age of next-generation sequencing [27].

3.4 DNA Alignment

Sequence alignment is useful in order to arrange sequences of DNA, RNA or protein to identify regions of similarity that may be of interest. There are two types of alignment:

- *Global alignment*: it attempts to align every residue in every sequence, which is most useful when the sequences in the query set are similar and of roughly equal size; it's worth to notice that such an approach could be time consuming and infeasible for very long sequences (*e.g.*, for two sequences of 100 residues there are more than 10 alternative alignments); a known and diffused algorithm for global alignment is the *Needleman-Wunsch algorithm* (named after their creators) [28];
- *Local alignment*: it is more useful for dissimilar sequences that are suspected to contain regions of similarity or similar sequence residues within their large sequence; for protein sequences, the most commonly used local alignment algorithm that allows gaps is described by Smith and Waterman (it is called the *Smith-Waterman algorithm*, after their names) [28].

These types of alignments realize an approximate matching, that differs from exact matching:

- *Exact matching*: Find all positions in a sequence (called *text*) where another sequence (*pattern*) occurs [29];

- *Approximate matching*: as exact matching but allows some differences between the pattern and its occurrence in the text [29].

Another categorization could be done if we distinguish between alignment between two sequences and more than two sequences:

- *Pairwise alignment*: it is the alignment between two sequences [29];
- *Multiple sequence alignment*: it is the simultaneous alignment of at least two sequences, usually displayed as a matrix with multiple rows, each row corresponding to one sequence [29].

In the following paragraphs, we will briefly talk about some of the most important methods for exact and approximate matching.

3.4.1 Exact matching methods

Naïve algorithm

Before talking about modern methods for exact matching, it's a good idea to focus our attention on the *naïve algorithm*. Figure 3.3 shows the Python code for this algorithm.

```
def naive(p, t):
    occurrences = []
    for i in range(len(t) - len(p) + 1): # Loop over alignments
        match = True
        for j in range(len(p)):         # Loop over characters
            if t[i+j] != p[j]:         # compare characters
                match = False          # mismatch; reject alignment
                break
        if match:
            occurrences.append(i)      # all chars matched; record
    return occurrences
```

Figure 3.3. Python code for the naïve algorithm [30].

The parameters of this function are the search pattern (p) and the reference string (t); they both are made up from characters belonging to a chosen alphabet Σ . The possible alignments are $N - M + 1$, where N and M are the length of reference string and search pattern respectively. The search pattern is shifted along the reference string and, at each shift, the comparisons between the characters of the search pattern and the corresponding characters of the reference string begin: if a matching occurs, the index is stored in the occurrences array. Clearly, this is the simplest matching method possible, as the least optimized: in the worst case, the number of comparisons is equal to $M(N - M + 1)A$, where A is the number of binary digits necessary to represent a character from the alphabet Σ .

Boyer-Moore algorithm

Boyer-Moore algorithm uses knowledge gained from character comparisons to skip future alignments that definitely won't match; it is based upon the following three rules:

- if we mismatch, use knowledge of the mismatched text character to skip alignments (*Bad character rule*);
- if we match some characters, use knowledge of the matched characters to skip alignments (*Good suffix rule*);
- Try alignments in one direction, then try character comparisons in opposite direction (useful for longer skips) [31].



Figure 3.4. Bad character rule [31].

In Figure 3.4 there is an example that illustrates how the bad character rule works; a pattern P is compared with the first set of characters from reference T (notice that the comparison order is opposite to the pattern shifting over reference order); when a character mismatch occurs (in the example the mismatching is at character labelled with b) characters are skipped until b matches its opposite in P (*Case a*) or P moves past b (*Case b*). Thanks to this optimization, we skipped 8 unnecessary alignments in the example above.

In Figure 3.5 another example illustrates the good suffix rule. Here, t is a substring of reference T that matched a suffix of search pattern P ; alignments are skipped until t matches opposite characters in P (*Case a*) or a prefix of P matches a suffix of t (*Case b*) or P moves past t , whichever happens first.

Thanks to the applications of all these optimizations together, this algorithm has the interesting feature of having a sub-linear average case (while it has a linear worst-case search time). Figure 3.6 illustrates an interesting comparison between naïve algorithm and Boyer-Moore algorithm over two references: human genome and all Shakespeare's works.



Figure 3.5. Good suffix rule [31].

	Naïve matching		Boyer-Moore		
	# character comparisons	wall clock time	# character comparisons	wall clock time	
P : "tomorrow" T : Shakespeare's complete works	5,906,125	2.90 s	785,855	1.54 s	17 matches $ T = 5.59$ M
P : 50 nt string from Alu repeat* T : Human reference (hg19) chromosome 1	307,013,905	137 s	32,495,111	55 s	336 matches $ T = 249$ M

Figure 3.6. Naïve algorithm vs. Boyer-Moore algorithm [31].

Knuth-Morris-Pratt algorithm

Let P and q be a search pattern and an integer variable respectively. The main idea behind this algorithm is to build a partial match table pm that, for each proper suffix of $P[0 : q]$, tells us the length of the longest match between this suffix and a proper prefix of $P[0 : q]$. Figure 3.7 illustrates an example of partial matching table for a given pattern. The algorithm progresses as follows, assuming that $P[0 : q - 1]$ matches $T[i - q : i - 1]$ (where i is another integer variable for algorithm iterations and T is the reference):

1. if $P[q] = T[i]$, then if $q < m$ (where m is the length of P) we extend the length of the match, otherwise we've found a match and set $q = pm[q - 1]$;
2. else if $P[q] \neq T[i]$, then if $q = 0$ we increment i , otherwise we shift the pattern by $pm[q - 1]$ and set $q = pm[q - 1]$.

The algorithm has a linear time complexity given by $\mathcal{O}(n + m)$ time, where m and n are the search pattern length and the reference length respectively [32].

P	C	G	A	G	A	C	G	A	G	A	T
q	0	1	2	3	4	5	6	7	8	9	10
pm[q]	0	0	0	0	0	1	2	3	4	5	0

Figure 3.7. Partial matching table [32].

3.4.2 Approximate matching methods

Needleman-Wunsch algorithm

Needleman-Wunsch algorithm is an algorithm for global alignment, and it can be summarized in five steps:

1. *consider all the possible pairs of residues from two sequences*: to do so, we can build two bidimensional matrices (traceback matrix for the sequences and score matrix for scores);
2. *initialize the score matrix*: it will be used to determine the relative score made by matching two characters in a sequence alignment; this alignment will then be used to determine the likelihood of one character being at the same position in the sequence as another character;
3. *assign gap penalties*: usually there are high chances of insertions and deletions (indels) in biological sequences but one large indel is more likely rather than multiple small indels in a given sequences; two kind of penalties are assigned to take the issue into consideration, called gap opening penalty (relatively higher) and gap extension penalty (relatively lower);
4. *calculate scores and fill the traceback matrix*;
5. *deduce the alignment from the traceback matrix*.

The last step of the algorithm is also called *traceback*. The traceback always begins with the last cell to be filled with the score (the bottom right cell). One moves according to the traceback value written in the cell. There are three possible moves: diagonally (toward the top-left corner of the matrix), up or left. The traceback is completed when the first cell of the matrix (top-left) is reached (“done” cell). Figure 3.8 shows an example of score matrix and traceback matrix, while Figure 3.9 shows the traceback performed on the completed traceback matrix [33].

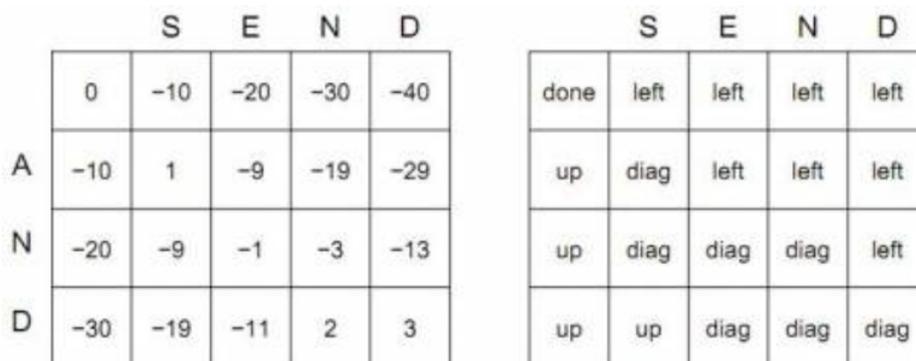


Figure 3.8. Score matrix and Traceback matrix [33].

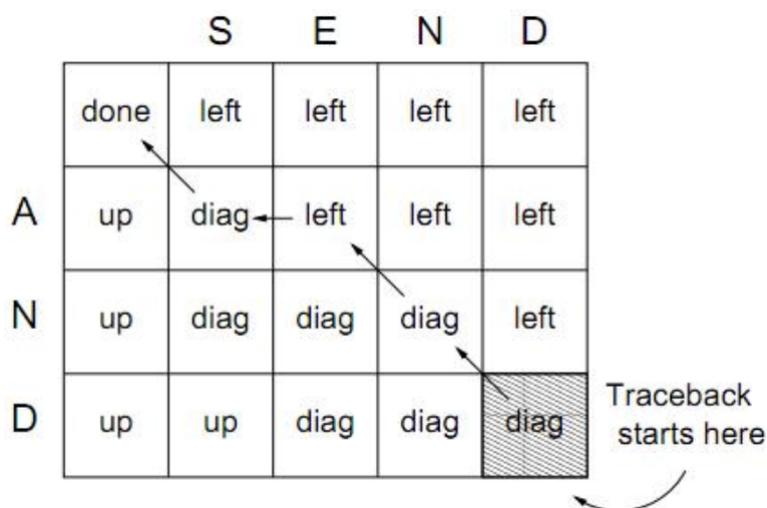


Figure 3.9. Traceback performed on the completed traceback matrix [33].

Smith-Waterman algorithm

Smith-Waterman algorithm is an algorithm for local alignment based on Needleman-Wunsch that implements a technique called *dynamic programming*. *Dynamic programming* computes optimal local alignments of two sequences: this means it identifies the two sub-sequences that are best preserved, *i.e.* their alignment shows the maximal similarity scoring. In order to find such a local alignment, the original Needleman-Wunsch algorithm is extended with an additional case “0”. This lower bound on the similarity score excludes “too bad” alignments that are eventually “not similar” ($score < 0$).

$$S_{i,j} = \max \begin{cases} S_{i-1,j-1} + s(a_i, b_j) \\ S_{i-1,j} + s(a_i, -) \\ S_{i,j-1} + s(-, b_j) \\ 0 \end{cases} = \max \begin{cases} S_{i-1,j-1} + 1 & a_i = b_j \\ S_{i-1,j-1} - 1 & a_i \neq b_j \\ S_{i-1,j} - 2 & b_j = - \\ S_{i,j-1} - 2 & a_i = - \\ 0 & \end{cases} \quad (3.1)$$

The dynamic programming approach tabularizes optimal sub-solutions in matrix S , where an entry $S_{i,j}$ represents the maximal similarity score for any local alignment of the (sub)prefixes $a_{x\dots i}$ with $b_{y\dots j}$, where $x, y > 0$ are so far unknown and have to be identified via traceback. The above example is valid for a matching score equal to +1, a mismatching score equal to -1 and a gap score equal to -2 [34].

S		A ₁	A ₂	C ₃	C ₄	T ₅	G ₆	G ₇	C ₈	A ₉	C ₁₀	T ₁₁	T ₁₂
	0	0	0	0	0	0	0	0	0	0	0	0	0
A ₁	0	1	1	0	0	0	0	0	0	1	0	0	0
A ₂	0	1	2	0	0	0	0	0	0	1	0	0	0
C ₃	0	0	0	3	1	0	0	0	1	0	2	0	0
T ₄	0	0	0	1	2	2	0	0	0	0	0	3	1
G ₅	0	0	0	0	0	1	3	1	0	0	0	1	2
T ₆	0	0	0	0	0	1	1	2	0	0	0	1	2
G ₇	0	0	0	0	0	0	2	2	1	0	0	0	0
C ₈	0	0	0	1	1	0	0	1	3	1	1	0	0
A ₉	0	1	1	0	0	0	0	0	1	4	2	0	0
C ₁₀	0	0	0	2	1	0	0	0	1	2	5	3	1
T ₁₁	0	0	0	0	1	2	0	0	0	0	3	6	4
A ₁₂	0	1	1	0	0	0	1	0	0	1	1	4	5
Score: 6													

Figure 3.10. Smith-Waterman score matrix [34].

3.5 DNA Big Data

It's worth to spend a few words about the size of genome data, taking human genome as example. The complete genetic information of a human being is stored inside 46 chromosomes, containing about 6×10^9 base pairs. In order to represent a DNA sequence on a computer, we need to be able to represent all 4 base pairs possibilities in a binary format: we can denote each base pair using a minimum of 2 bits, which yields 4 different bit combinations (00, 01, 10, and 11). Since each 2-bit combination would represent one DNA base pair, a single byte (or 8 bits) can represent 4 DNA base pairs. In order to represent the entire human genome in terms of bytes, we can perform the following calculations:

$$6 \times 10^9 \frac{\text{base pairs}}{\text{genome}} \times \frac{1 \text{ byte}}{4 \text{ base pairs}} = 1.5 \times 10^9 \text{ bytes or } 1.5 \text{ Gigabytes} \quad (3.2)$$

This simple example is useful to understand that the amount of data involved in sequence alignment could be big, so it shouldn't be surprising that we tried to optimize this task with various methods (as those previously seen in this chapter), but classical algorithms have their limitations and our desire is to speed up this

process. This is the main reason behind our interest in applying quantum computing in this context: when we will be able to build better quantum computers, they could be a true game changer [35].

Chapter 4

Quantum Pattern Matching algorithms

4.1 Quantum search applied to Pattern Matching

We have seen in the previous chapter that Genome Sequencing is an important problem for biological and medical reasons needing a lot of computational resources in order to be treated by classical algorithms running on classical computers: searching for matching patterns between two genomic sequences is time consuming and our desire is to find a speed-up that can make this problem more approachable.

In this chapter we will talk about Quantum Pattern Matching algorithms that could find an application in the context of Genome Sequencing. We will start talking about the fundamental Grover's algorithm, which is the main base all the other quantum search algorithms we will talk about in this thesis started from. After that, we will talk about some interesting quantum search algorithms, some of which have been developed in recent years.

4.2 The fundamental Grover's Algorithm

Grover's algorithm is a quantum algorithm used for database search. In order to understand its importance, we can illustrate the following example: let's suppose we are given a large list of N items and among them there is one particular item with unique characteristics that we desire to locate; we can call this item s . If we choose to use classical computation to search s , we would have to check on average $N/2$ items, and all of them in the worst case. On a quantum computer it's possible to find s in roughly \sqrt{N} steps running Grover's algorithm: this is an important speed-up in the search process that captures our attention. Before illustrating the algorithm's steps, we will talk about some preliminary concepts that are needed in order to fully understand what the algorithm does.

4.2.1 The Oracle

Let's keep focusing on the N -sized search space that was given to us: rather than search the elements directly, we concentrate on the indices of those elements, which are between 0 and $N - 1$. For convenience we assume $N = 2^n$, where n is the number of bits used to store an index. Also, we assume that our search problem has exactly M solutions, with $1 \leq M \leq N$. A particular instance of the search problem can conveniently be represented by a function f which takes as input an integer x in the range between 0 and $N - 1$. We assume that $f(x) = 1$ denotes that x is a solution to the search problem and $f(x) = 0$ denotes it isn't. In this situation we introduce a *quantum oracle*: it is defined as a black box that can recognize solutions to the search problem. Denoting with O the oracle operator, we can express its action on a quantum state $|x\rangle$ as follows:

$$|x\rangle \xrightarrow{O} (-1)^{f(x)}|x\rangle \quad (4.1)$$

We say that the oracle *marks* the solutions to the search problem by shifting the phase of the solution. For an N item search problem with M solutions, it turns out that we need to apply the search oracle only $\mathcal{O}(\sqrt{N/M})$ times in order to obtain a solution on a quantum computer. At this point, the reader could be confused about the nature of the oracle: from our explanation, it could seem that the oracle already knows the solutions to the search problem, but this is an incorrect statement. We have to distinguish between *knowing* the solution to a search problem and *being able to recognize* the solution to a search problem: the crucial point is that it is possible to do the latter without necessarily being able to do the former [3].

4.2.2 The Algorithm

At this point, we can start talking about how the algorithm works: let's suppose to start with a quantum register containing n qubits. The algorithm starts with the quantum register in the state $|0\rangle^{\otimes n}$ (this notation means that all n qubits in the quantum register are in the $|0\rangle$ state). We use Hadamard gates on all qubits to put the quantum register in a uniform superposition state, obtaining the following:

$$|\psi\rangle = \frac{1}{\sqrt{N}} \sum_{x=0}^{N-1} |x\rangle \quad (4.2)$$

Supposing $N = 4$, Figure 4.1 illustrates graphically the situation of linear superposition at this stage of the algorithm, with item at index 3 being the solution state.

At this point the algorithm iteratively applies a subroutine composed of the following two steps (t denotes a generic iteration of the algorithm):

1. oracle O is applied to the quantum state: $|\psi_{t'}\rangle = O|\psi_t\rangle$ (Figure 4.2);
2. operator G is applied to the quantum state: $|\psi_{t''}\rangle = G|\psi_{t'}\rangle$ (Figure 4.3).

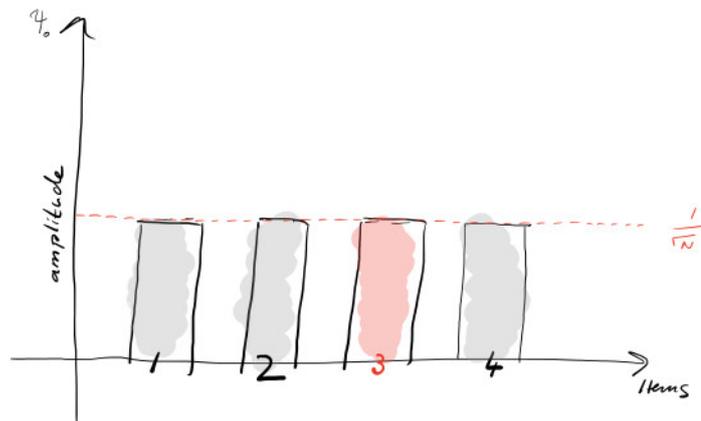
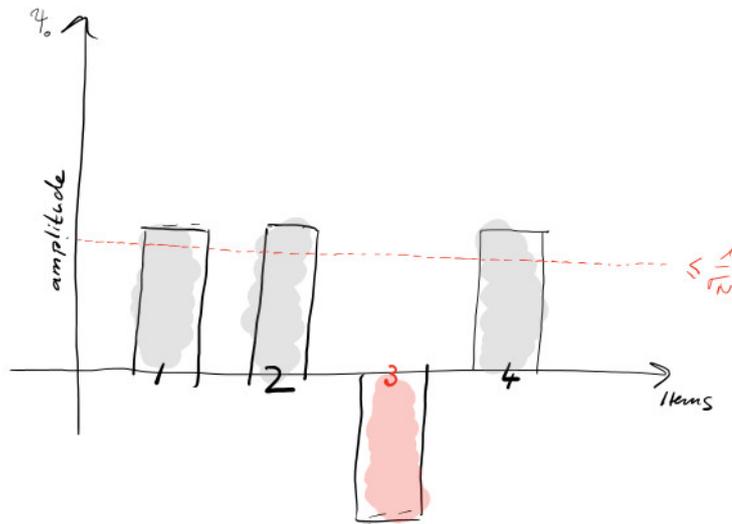


Figure 4.1. Uniform superposition of states [36].

Figure 4.2. Oracle O marks the solution state [36].

Looking at Figure 4.3, we can see that operator G has increased the amplitude of the solution state; at this point, it's necessary to explain the nature of this operator to fully understand how the algorithm is capable to find the solution state. Operator G implements the *inversion about mean*, that is another key concept of Grover's algorithm. To understand better, we can illustrate the following example: let's suppose that our search space is composed by eight possible states and that the quantum register (with a size of three qubits) is already in the situation of linear superposition:

$$\frac{1}{\sqrt{8}}(|000\rangle + |001\rangle + |010\rangle + |011\rangle + |100\rangle + |101\rangle + |110\rangle + |111\rangle) \quad (4.3)$$

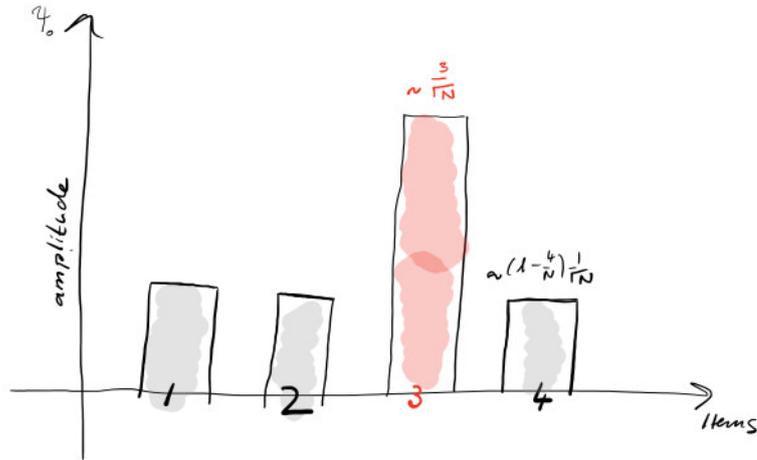


Figure 4.3. Operator G is applied [36].

Let's suppose that the solution to our search problem is the state $|110\rangle$. Applying the oracle operator O to the linear superposition, we obtain the following:

$$\frac{1}{\sqrt{8}}(|000\rangle + |001\rangle + |010\rangle + |011\rangle + |100\rangle + |101\rangle - |110\rangle + |111\rangle) \quad (4.4)$$

The oracle has marked the solution with a minus sign before $|110\rangle$. Now we are ready for the inversion about mean: this step will move all the chances of the other states down, while moving the chance for $|110\rangle$ up; it will do this by taking the average of the amplitudes out in front of each individual state. Since most of them are $1/\sqrt{8}$, this will be very close to $1/\sqrt{8}$ but not exactly. The average is calculated as follows:

$$\frac{\frac{1}{\sqrt{8}} + \frac{1}{\sqrt{8}} + \frac{1}{\sqrt{8}} + \frac{1}{\sqrt{8}} + \frac{1}{\sqrt{8}} + \frac{1}{\sqrt{8}} - \frac{1}{\sqrt{8}} + \frac{1}{\sqrt{8}}}{8} = \frac{6}{8\sqrt{8}} \quad (4.5)$$

The average is ≈ 0.265 compared to $1/\sqrt{8}$, which is ≈ 0.35 ; then, this average is taken and each state's amplitude is flipped over it. Twice 0.265 is 0.53 , so every state but the solution state changes its amplitude to the following:

$$2 \times \frac{6}{8\sqrt{8}} - \frac{1}{\sqrt{8}} \approx 0.53 - 0.35 = 0.18 \quad (4.6)$$

Then, the solution state changes its amplitude to the following:

$$2 \times \frac{6}{8\sqrt{8}} - \frac{-1}{\sqrt{8}} \approx 0.53 + 0.35 = 0.88 \quad (4.7)$$

So, our new state after one application of the oracle O and the G operator is the following:

$$\begin{aligned} \left(2 \times \frac{6}{\sqrt{8}} - \frac{1}{\sqrt{8}}\right) \times (|000\rangle + |001\rangle + |010\rangle + |011\rangle + |100\rangle + |101\rangle + |111\rangle) + \\ + \left(2 \times \frac{6}{\sqrt{8}} - \frac{-1}{\sqrt{8}}\right) \times |110\rangle \end{aligned} \quad (4.8)$$

Alternatively, we can write the following:

$$0.18 \times (|000\rangle + |001\rangle + |010\rangle + |011\rangle + |100\rangle + |101\rangle + |111\rangle) + 0.88 \times |110\rangle \quad (4.9)$$

Now, we can clearly see that, with just one iteration of the Grover's subroutine, we have $0.88^2 \approx 0.77$ or about 77% chance of obtaining the correct input $|110\rangle$ and $0.18^2 \approx 0.03$ or about 3% chance of obtaining each of the other inputs when we measure the qubits in the quantum register (since there are seven other inputs, we have about 21% chance of seeing one of the other outputs in total) [2]. This simple example highlights the efficiency of this quantum search algorithm. The optimal number of iterations of the Grover's subroutine would increase to the maximum the chance of obtaining solution states from the measurement of qubits. Which is the optimal number of iterations? It turns out that with M solutions to the search problem, roughly $\sqrt{N/M}$ iterations suffice, so Grover's algorithm has a $\mathcal{O}(\sqrt{N/M})$ time complexity: this is a quadratic improvement over the $\mathcal{O}(N/M)$ time complexity in the classical case. If we go further the optimal number of iterations, we would find ourselves straying from the correct solutions.

Now we will express the algorithm's steps from a more mathematical perspective. Grover's algorithm has two registers: n qubits in the first and one qubit in the second. We start by creating the linear superposition of all the $N = 2^n$ possible states in the first quantum register. This is done by applying the operator $H^{\otimes n}$ to the first register that starts from the state $|0\rangle^{\otimes n}$:

$$|\psi\rangle = H^{\otimes n}|0\rangle^{\otimes n} = (H|0\rangle)^{\otimes n} = \left(\frac{|0\rangle + |1\rangle}{\sqrt{2}}\right)^{\otimes n} = \frac{1}{\sqrt{N}} \sum_{x=0}^{N-1} |x\rangle \quad (4.10)$$

Now we define a function $f : \{0, \dots, N - 1\} \rightarrow \{0, 1\}$ capable of recognizing the solution to the search problem:

$$f(x) = \begin{cases} 1 & \text{if } x \text{ is the searched element } (s), \\ 0 & \text{otherwise} \end{cases} \quad (4.11)$$

Let's suppose the existence of a unitary operator O , called oracle:

$$O(|x\rangle|y\rangle) = |x\rangle|y \oplus f(x)\rangle \quad (4.12)$$

$|x\rangle$ is a state of the first register ($x \in \{0, \dots, 2^n - 1\}$), $|y\rangle$ is a state of the second register ($y \in \{0, 1\}$) and the \oplus symbol denotes the sum in modulo 2. Thanks to the sum in modulo 2, we can exploit the following:

$$1 \oplus f(x) = \begin{cases} 0 & \text{for } x = s \\ 1 & \text{for } x \neq s \end{cases} \quad (4.13)$$

So, we can easily check the following:

$$\begin{aligned} O(|x\rangle|-\rangle) &= \frac{O(|x\rangle|0\rangle) - O(|x\rangle|1\rangle)}{\sqrt{2}} = \\ &= \frac{|x\rangle|f(x)\rangle - |x\rangle|1 \oplus f(x)\rangle}{\sqrt{2}} = \\ &= (-1)^{f(x)}|x\rangle|-\rangle \end{aligned} \quad (4.14)$$

Let's apply the O operator on the initial superposition denoted by $|\psi\rangle$, with the second single-qubit register in the $|-\rangle$ state: we can see that the state of the second register doesn't change. To put the second register in the $|-\rangle$ state, we can start with $|1\rangle$ and apply a Hadamard gate:

$$|\psi_1\rangle|-\rangle = O(|\psi\rangle|-\rangle) = \frac{1}{\sqrt{N}} \sum_{x=0}^{N-1} (-1)^{f(x)}|x\rangle|-\rangle \quad (4.15)$$

As expected, the amplitude of the searched element is negative, while all other amplitudes are positive, so the solution state has been marked by the oracle.

Now we can see in more depth the G operator, that realizes the inversion about mean. Knowing that $|\psi\rangle$ represents the initial superposition state, the G operator can be expressed as follows (I represents the identity matrix):

$$G = 2|\psi\rangle\langle\psi| - I \quad (4.16)$$

We can rewrite $|\psi_1\rangle$ as follows:

$$|\psi_1\rangle = |\psi\rangle - \frac{2}{\sqrt{2^n}}|s\rangle \quad (4.17)$$

So, applying G to $|\psi_1\rangle$, we can obtain the following:

$$|\psi_G\rangle = (2|\psi\rangle\langle\psi| - I)|\psi_1\rangle = \frac{2^{n-2} - 1}{2^{n-2}}|\psi\rangle + \frac{2}{\sqrt{2^n}}|s\rangle \quad (4.18)$$

$|\psi_G\rangle$ is the state of the first register after applying the G operator. The second register is still in the $|-\rangle$ state: this realizes the inversion about mean [37].

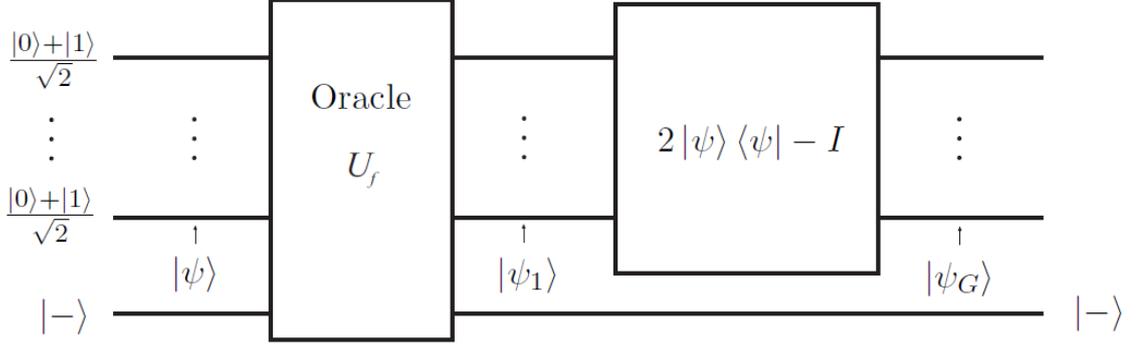


Figure 4.4. One Grover iteration [37].

Summarizing all the algorithm's steps in mathematical notation, we can write the following:

$$|\psi_{final}\rangle = \left(\prod_{t=1}^T GO \right) |\psi\rangle \quad (4.19)$$

T represents the optimal number of Grover's iterations. This number is calculated as follows (M represents the number of solution states) [3]:

$$T = \left\lceil \frac{\pi}{4} \sqrt{\frac{N}{M}} \right\rceil \quad (4.20)$$

The algorithms we will discuss about in the following paragraphs adopt the Grover's algorithm as main base, trying to reach better performance.

4.3 Quantum Associative Memory

The concept of *Quantum Associative Memory* (*QAM*) has been developed by D. Ventura and T. Martinez [38][39][40]. The same authors have written an important research paper where they explain how to initialize an arbitrary amplitude distribution for a quantum state (where arbitrary means that the distribution is not necessarily linear) [41]: this nonlinear initialization is used in the context of Quantum Associative Memory, so we will spend some words about it.

4.3.1 Arbitrary amplitude distribution initialization

Supposing we are given a set T of m examples of a function f , we want to produce the following as the quantum state of n qubits:

$$|\tilde{f}\rangle = \frac{1}{\sqrt{m}} \sum_{\bar{z} \in T} f(\bar{z}) |\bar{f}\rangle \quad (4.21)$$

We suppose that $f : \bar{z} \rightarrow s$ with $\bar{z} \in \{0, 1\}^n$ and $s \in \{-1, 1\}$. $|\tilde{f}\rangle$ represents a computational basis state. As we can see, the goal is to produce a superposition with only the amplitudes of the m states we are interested in being not zero (remember that the number of all the possible states is equal to 2^n). The algorithm proposed by Ventura and Martinez uses $2n + 1$ qubits for the initialization, organized in three registers x , g and c with n , $n - 1$ and 2 qubits respectively. The quantum state of all the three registers together can be expressed as $|x, g, c\rangle$. The algorithm's steps are illustrated in Figure 4.5.

1. Generate $|\tilde{f}\rangle = |x_1 \dots x_n, g_1 \dots g_{n-1}, c_1 c_2\rangle = |\bar{0}\rangle$
2. for $m \geq p \geq 1$
3. for $1 \leq j \leq n$
4. if $z_{pj} \neq z_{p+1j}$ (where $z_{m+1} = \{0\}^n$)
5. $\hat{F}_{c_2 x_j}^0 |\tilde{f}\rangle$
6. $\hat{F}_{c_2 c_1}^0 |\tilde{f}\rangle$
7. $\hat{S}_{c_1 c_2}^{s,p} |\tilde{f}\rangle$
8. $\hat{A}_{x_1 x_2 g_1}^{z_1 z_2} |\tilde{f}\rangle$
9. for $3 \leq k \leq n$
10. $\hat{A}_{x_k g_{k-2} g_{k-1}}^{z_k 1} |\tilde{f}\rangle$
11. $\hat{F}_{g_{n-1} c_1}^1 |\tilde{f}\rangle$
12. for $n \geq k \geq 3$
13. $\hat{A}_{x_k g_{k-2} g_{k-1}}^{z_k 1} |\tilde{f}\rangle$
14. $\hat{A}_{x_1 x_2 g_1}^{z_1 z_2} |\tilde{f}\rangle$
15. $\hat{F}_{c_2} |\tilde{f}\rangle$

Figure 4.5. Initialization algorithm by Ventura and Martinez [41].

The operators used in the algorithm are all unitary:

- \hat{F}_q is the *NOT* operator;
- $\hat{F}_{q_1 q_2}^1$ is the *CNOT* operator with q_1 as control and q_2 as target;
- $\hat{F}_{q_1 q_2}^0$ is the *CNOT* operator with *NOT*(q_1) as control and q_2 as target;
- $\hat{A}_{q_1 q_2 q_3}^{00}$ flips q_3 if q_1 and q_2 are both 0;
- $\hat{A}_{q_1 q_2 q_3}^{01}$ flips q_3 if q_1 is 0 and q_2 is 1;
- $\hat{A}_{q_1 q_2 q_3}^{10}$ flips q_3 if q_1 is 1 and q_2 is 0;
- $\hat{A}_{q_1 q_2 q_3}^{11}$ flips q_3 if q_1 and q_2 are both 1;
- Finally, $\hat{S}_{q_1 q_2}^{s,p}$ is a 2-qubit unitary operator that can be expressed as follows:

$$\hat{S}_{q_1 q_2}^{s,p} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \sqrt{\frac{p-1}{p}} & \frac{-s}{\sqrt{p}} \\ 0 & 0 & \frac{s}{\sqrt{p}} & \sqrt{\frac{p-1}{p}} \end{pmatrix} \quad (4.22)$$

Thanks to this initialization algorithm, we can obtain an arbitrary amplitude distribution.

4.3.2 The Algorithm

We are now ready to talk about the Quantum Associative Memory algorithm for pattern matching by Ventura and Martinez. Let's suppose we are given a set P of m binary patterns of length n that we want to store in a quantum register using superposition. In order to use the initialization algorithm seen before, we need a total of $2n + 1$ qubits. If we denote all the steps needed for initialization with the \hat{P} operator, we can express the initialization step as follows:

$$|\psi\rangle = \hat{P}|\bar{0}\rangle \quad (4.23)$$

In this way, we have correctly memorized the set P of patterns in our quantum register of length n . Now, suppose we know $n - 1$ bits of a pattern and we want to recall the entire pattern: in order to achieve this, we can simply apply the following:

$$|\psi'\rangle = GO_PGO_\tau|\psi\rangle \quad (4.24)$$

$$|\psi''\rangle = \left(\prod_{i=0}^T GO_\tau \right) |\psi'\rangle \quad (4.25)$$

τ denotes the target pattern, O_τ the corresponding oracle, O_P is the oracle that marks all the stored patterns and T the optimal number of Grover's iterations [40]. In this way, we were able to store up to 2^n patterns in $\mathcal{O}(mn)$ steps [39]. The only problem that still needs to be resolved is calculating the optimal number of Grover's iterations: we are able to do this calculation for a linear amplitude distribution, but this is not our case. This problem is approached by Ventura and Martinez on the base of a research paper by E. Bigham, O. Bigham, D. Biron, M. Grassl and D. A. Lidar [42][40]: they calculated T for an arbitrary amplitude distribution as follows:

$$T = \frac{\frac{\pi}{2} - \arctan\left(\frac{\bar{k}}{\bar{l}}\sqrt{\frac{r_0+r_1}{N-r_0-r_1}}\right)}{\arccos\left(1 - 2\frac{r_0+r_1}{N}\right)} \quad (4.26)$$

\bar{k} and \bar{l} represent the average of the amplitudes of marked states and non-marked states after applying Equation 4.24, while r_0 and r_1 are the number of marked states that do not correspond to stored patterns and the number of marked states that correspond to stored patterns respectively. N represents the size of search space. To obtain the correct number of Grover's iterations, T has to be rounded to the nearest integer. This algorithm can be used for pattern completion, where the input query is a pattern with some missing characters that we can represent with the ? symbol (for example $100??$ could be an input query).

4.4 Fast Quantum Search based on Hamming distance

Another interesting algorithm was developed by L. C. L. Hollenberg, explicitly proposed for the context of Protein Sequence Comparison [43]. The paper illustrates the following example: suppose that an entire genome is stored in a database D as a continuous list of N residues, $D = \{R_0, R_1, \dots, R_{N-1}\}$. Also, a sample sequence S composed by m residues is given, $S = \{r_0, r_1, \dots, r_{m-1}\}$. In the original paper an example is made by supposing that each residue denotes a letter from the amino acid alphabet (which has a size equal to 20), so $\lceil \log_2 20 \rceil = 5$ bits are needed for each residue, but we can generalize the algorithm for every alphabet size by considering $l = \lceil \log_2 |A| \rceil$ bits for each character, with A being the alphabet set and $|A|$ the number of alphabet's characters. So, each residue R_i and r_i from D and S is represented by bit strings of length l and we can write the following, with $B \in \{0, 1\}$ and $b \in \{0, 1\}$:

$$R_i = \prod_{\alpha=0}^l B_{i\alpha} \quad r_i = \prod_{\alpha=0}^l b_{i\alpha} \quad (4.27)$$

A couple of quantum registers Q_1 and Q_2 are used. The entire database is represented by a quantum superposition over the two registers as follows:

$$|\psi_D\rangle = \frac{1}{\sqrt{N-m+1}} \sum_{i=0}^{N-m} |\phi_i\rangle \otimes |i\rangle \quad (4.28)$$

All the consecutive sub-sequences in the database of length m are encoded in the first register Q_1 with size equal to lm as follows:

$$|\phi_i\rangle = \prod_{\alpha=i}^{i+m-1} \prod_{\beta=0}^l |B_{\alpha\beta}\rangle = \prod_{\alpha=0}^{lm-1} |q_{i\alpha}\rangle \quad (4.29)$$

This means that $N - m + 1$ sub-sequences of length m are extracted from D by moving along from the first position (allowing overlapping between the sub-sequences). The starting index of the sub-sequence is encoded by binary numbers $|i\rangle$ in the second register. Since $0 \leq i \leq N - m$, the second register Q_2 must have an adequate number of qubits to be able to encode the index of a sub-sequence, given by $\lceil \log_2(N - m + 1) \rceil$. We can express the sample sequence state $|S\rangle$ by its individual qubits, too:

$$|S\rangle = \prod_{\alpha=0}^{m-1} \prod_{\beta=0}^l |b_{\alpha\beta}\rangle = \prod_{\alpha=0}^{lm-1} |s_{\alpha}\rangle \quad (4.30)$$

Now, in order to complete the initialization process, we need to evolve the stored sub-sequences in *Hamming distances*, calculated by comparing with the sample sequence S . The *Hamming distance* between two strings of equal length is the

number of positions at which the corresponding symbols are different [44]. We can encode the Hamming distances in the quantum database in a surprisingly easy way applying a *CNOT* operation with respect to the sample sequence state:

$$|\psi_H\rangle = U_{CNOT}(s)|\psi_D\rangle = \frac{1}{\sqrt{N-m+1}} \sum_{i=0}^{N-m} |\bar{\phi}_i\rangle \otimes |i\rangle \quad (4.31)$$

$|\bar{\phi}_i\rangle$ encodes the Hamming distance of the i -th sub-sequence with respect to the sample sequence, and can be expressed by its individual qubits as follows:

$$|\bar{\phi}_i\rangle = \prod_{\alpha=0}^{lm-1} |\bar{q}_{i\alpha}\rangle \quad \text{with} \quad \bar{q}_{i\alpha} = \begin{cases} 1 & \text{if } q_{i\alpha} \neq s_\alpha, \\ 0 & \text{otherwise} \end{cases} \quad (4.32)$$

At this point, to obtain the Hamming distance for the i -th sub-sequence we can simply calculate $T_i = \sum_{\alpha=0}^{lm-1} \bar{q}_{i\alpha}$

Now that we have successfully initialized Q_1 and Q_2 , we can express the oracle O_S and the operator G s follows:

$$O_S = 1 - 2|S\rangle\langle S| \quad (4.33)$$

$$G = 1 - 2|\psi_H\rangle\langle\psi_H| \quad (4.34)$$

The oracle O_S will mark the solution states as follows:

$$O_S|\bar{\phi}_i\rangle = \begin{cases} -|\bar{\phi}_i\rangle & \text{if } T_i = 0, \\ |\bar{\phi}_i\rangle & \text{otherwise} \end{cases} \quad (4.35)$$

At the start of each search it is not known how many solutions exist, or if there exist matches at all, meaning that Grover’s algorithm cannot be used directly. However, an extension of Grover’s algorithm proposed by M. Boyer, G. Brassard, P. Hoyer and A. Tapp can be used, which performs a search with an a priori unknown number of solutions N_t , and finds a match (if it exists) in $\mathcal{O}(\sqrt{\frac{N}{N_t}})$ [45].

4.5 Quantum indexed Bidirectional Associative Memory

This algorithm has been proposed in 2019 by A. Sarkar, Z. Al-Ars, C. G. Almudever and K. Bertels [46]. It is presented by the authors as a “*novel quantum pattern matching algorithm specifically designed for the context of genome sequence reconstruction*” and has been called “*Quantum indexed Bidirectional Associative Memory*” (*QiBAM*). The algorithm is partially based on the previously illustrated algorithms and the research papers at [47][48] that introduced the model of *Quantum Associative Memory with distributed queries*, so we will briefly illustrate it before talking of the QiBAM algorithm in more depth.

4.5.1 Quantum Associative Memory with Distributed Queries

This model was first introduced by A. A. Ezhov, A. V. Nifanova and D. Ventura in 2000 [47] and then enhanced by a research paper in 2013 [48]. To introduce this new model, we can refer to the example made in [47].

Name	Code	Number	Code
A	00001	3	11
B	00010	0	00
C	00011	2	10
D	01010	1	01

Table 4.1. Phone-book database example [47].

The database in Table 4.1 contains some data organized as a phone-book, with a name and a number for each entry. The entries are ordered by name and unordered by number. While the *direct problem* of finding the number with given name is easy, the *inverse problem* of finding the name with a given number is difficult: the most efficient classical solution is random search which demands in the worst case $N - 1$ queries and on average $N/2$ queries. On the other hand, *using the Quantum Associative Memory approach*, we can setup a quantum state $|\psi\rangle$ which is the superposition of all the entries in the phone-book database:

$$|\psi\rangle = \frac{1}{\sqrt{P}} \sum_{x \in M} |x\rangle \quad (4.36)$$

P is the number of records in the database M that have been stored in memory, and $|x\rangle = |name, number\rangle$ is a memorized state which is composed by the two sets of qubits encoding *name* and *number*. At this point we can apply the oracle O and the operator G in Grover's iterations: we want to find the *name* with a given *number*. For a solution state $|s\rangle$ we can write the following:

$$O_{name}|x\rangle = \begin{cases} -|x\rangle & \text{if } |x\rangle = |s\rangle, \\ |x\rangle & \text{otherwise} \end{cases} \quad (4.37)$$

The operator G realizes the inversion about mean as follows (a_x denotes the amplitude of a state $|x\rangle$):

$$G : a_x \mapsto \frac{2}{P} \sum_{x \in M} a_x - a_x \quad (4.38)$$

The model described so far represents the Quantum Associative Memory model that we have already discussed in previous paragraphs, capable of doing pattern completion. However, this model does not actually take into account the distance between states but only uses information about the presence of some prescribed bit values in the memory state. We would like to introduce a metric into the quantum

search algorithm in the form of *distributed queries*. *Distributed query* means that a query, or stimulus to the system, has the form of a superposition, just as a quantum memory does:

$$|b^p\rangle = \sum_{x=0}^{2^d-1} b_x^p |x\rangle \quad (4.39)$$

The query includes in general *all* basis states (with d being the number of qubits encoding a phone number). The state $|p\rangle$ is the *center* of distribution. The introduction of distributed queries demands the modification of the phone-book memory in such a way that it has every possible basis state, despite the fact that most of them have no corresponding name. Such a memory can be represented by the example in Table 4.2.

Record	Name	Phone
1	not used	000
2	not used	001
3	Alice	010
4	not used	011
5	not used	100
6	not used	101
7	not used	110
8	Bob	111

Table 4.2. Example of Quantum Associative Memory with Distributed Queries [47].

At this point, we need to modify the oracle O such that it defines not a single query or a finite set of queries, but rather a *fuzzy query*. We already said that $|p\rangle$ is the center of the distribution described by Equation 4.39: we want the maximal value of that distribution to occur for some definite state $|x\rangle = |p\rangle$, while the amplitudes of the other basis states decrease monotonically with the Hamming distance with respect to $|p\rangle$ (from now on, we will refer to $|p\rangle$ as the *query center*). In order to satisfy this requirement, we can write the following binomial distribution:

$$|b_x^p|^2 = q^{|p-x|} (1-q)^{d-|p-x|} \quad (4.40)$$

$|p-x|$ denotes the Hamming distance between $|p\rangle$ and $|x\rangle$, while $0 < q < \frac{1}{2}$ is a number arbitrarily chosen in order to tune the width of the distribution. Introducing a distributed query with Hamming distance-dependent amplitudes for the basis states incorporates a metric into the model which permits comparison of the similarity of the stimulus and the retrieved memory. At this point we can express the oracle O_b and the operator G as follows [47]:

$$O_b = I - 2|b_p\rangle\langle b_p| \quad (4.41)$$

$$G : a_x \mapsto \begin{cases} \frac{2}{P} \sum_{x \in M} a_x - a_x & \text{if } x \in M, \\ -a_x & \text{otherwise.} \end{cases} \quad (4.42)$$

This model has been improved by a research paper in 2013 [48] that proposed the algorithm in Figure 4.6: the operators O and D correspond to the oracle O_b and the inversion about mean G that we have already discussed, while I_M is another oracle that marks *all* the stored patterns in memory M .

-
- 1: $|0_1 0_2 \dots 0_n\rangle \equiv |\bar{0}\rangle$; {Initialize the register}
 - 2: $|\Psi\rangle = A|\bar{0}\rangle = \frac{1}{\sqrt{N-m}} \sum_{x \notin M}^{N-1} |x\rangle$; {Learn the patterns using exclusion approach}
 - 3: Apply the operator oracle \mathcal{O} to the register;
 - 4: Apply the operator diffusion \mathcal{D} to the register;
 - 5: Apply operator \mathcal{I}_M to the register;
 - 6: Apply the operator diffusion \mathcal{D} to the register;
 - 7: **repeat**
 - 8: Apply the operator oracle \mathcal{O} to the register;
 - 9: Apply the operator diffusion \mathcal{D} to the register;
 - 10: $i = i + 1$;
 - 11: **until** $i > \Lambda - 2$
 - 12: Observe the system.
-

Figure 4.6. Quantum Associative Memory with Improved Distributed Queries [48].

I_M is defined as follows:

$$I_M = I - (1 - e^{i\pi})|\varphi\rangle\langle\varphi| \quad \text{with} \quad |\varphi\rangle\langle\varphi| = \sum_{x \in M} |x\rangle\langle x| \quad (4.43)$$

$$I_M : a_x \mapsto \begin{cases} -a_x & \text{if } |x\rangle \in M, \\ a_x & \text{otherwise.} \end{cases} \quad (4.44)$$

Figure 4.7 illustrates the steps of the model of Quantum Associative Memory with Distributed Queries (the “not improved” version) in order to compare it with the improved version: in this case, the I_M operator doesn’t appear.

-
- 1: $|0_1 0_2 \dots 0_n\rangle \equiv |\bar{0}\rangle$; {Initialize the register}
 - 2: $|\Psi\rangle = A|\bar{0}\rangle = \frac{1}{\sqrt{N-m}} \sum_{x \notin M}^{N-1} |x\rangle$; {Learn the patterns using exclusion approach}
 - 3: **repeat**
 - 4: Apply the operator oracle \mathcal{O} to the register;
 - 5: Apply the operator diffusion \mathcal{D} to the register;
 - 6: $i = i + 1$;
 - 7: **until** $i > \Lambda$
 - 8: Observe the system.
-

Figure 4.7. Quantum Associative Memory with Distributed Queries [48].

4.5.2 The QiBAM model

Now we are ready to talk about the Quantum index Bidirectional Associative Memory model: the scheme in Figure 4.8 helps us to understand the various steps. The algorithm tries to do an approximate matching between the reference string (called T) with length N and the pattern we are searching (called P) with length M :

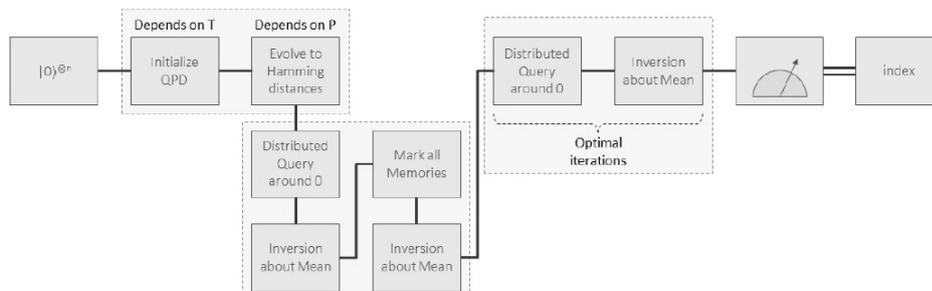


Figure 4.8. Diagram for QiBAM [46].

1. we start from a n -qubit quantum register prepared at $|0\rangle^{\otimes n}$ state;
2. the quantum register is initialized as follows, with $T_M(i)$ denoting the sub-sequence of length M from reference T at index i :

$$|\psi_0\rangle = \frac{1}{\sqrt{N-M+1}} \sum_{i=0}^{N-M} (|T_M(i)\rangle \otimes |i\rangle) \quad (4.45)$$

It's worth to notice that this initialization is the same of Equation 4.28 from the model proposed by Hollenberg [43];

3. the qubits encoding the sub-sequences are evolved into their Hamming distances with respect to the search pattern P . This step also is proposed by Hollenberg after the initialization step [43], so we can refer to Equations 4.31 and 4.32;
4. after the initialization of quantum database, a *distributed query with 0 as query center is performed*, with oracle O being the same of Equation 4.41 and $|b_p\rangle$ calculated with $p = 0$;
5. inversion about mean is performed, with G operator being the same as Equation 4.42;
6. operator I_M from Equations 4.43 and 4.44 to mark all the memory states is applied;
7. operator G is applied again;
8. Grover's iterations are performed, with Grover's subroutine being made up of applications of oracle O and inversion about mean G ;
9. finally, the qubits encoding the index are measured in order to obtain the final result.

4.6 Quantum Pattern Recognition

The final algorithm we will talk about was proposed in 2018 by K. Prousalis and N. Konofaos [49] and it was partially based on a previous research by R. Zhou and Q. Ding [50]. A *recurrence dot matrix approach* for sequence alignment is combined with a known quantum multi-pattern recognition method in order to improve the problem of sequence alignment. The *recurrence dot matrix method* is the most simple and qualitative one, though it is time-consuming in analyzing on a large scale: certain sequence features (such as insertions, deletions, repeats or inverted repeats of the elements of the sequence) can easily become identified by vision when a dot plot is formed for two sequences. A generic element a_{ij} of the matrix is marked with a dot if the corresponding characters i and j from two sequences s_1 and s_2 forming the dot matrix match. We can express this approach in mathematical notation:

$$a_{ij} = \begin{cases} 1 & \text{if } s_{1i} = s_{2j}, \\ 0 & \text{otherwise.} \end{cases} \quad (4.46)$$

In Figure 4.9 an example of recurrence dot matrix for two DNA sequences is defined. When a matching character occurs, the spot in the matrix is marked with a black square. This is a qualitative method to have an idea of the similarities between two given sequences: a black diagonal in the matrix denotes the presence of equal patterns between the two sequences. If the vertical sequence is much shorter than the horizontal one, we could use this method to spot a search pattern in a given reference: a full black diagonal means that at a certain index the search pattern could be found in the reference. However, this is only a qualitative method and we must take into account the fact that it could be time-consuming to build such a matrix classically. So, in order to obtain the recurrence dot matrix more efficiently than the classical way, a *spatial light modulator (SLM)* is used.

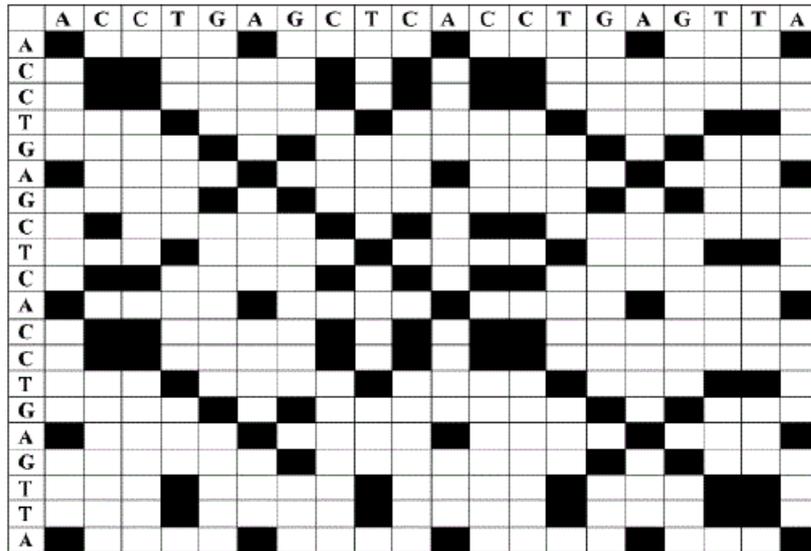


Figure 4.9. An example of recurrence dot matrix.

Now, we want to initialize a quantum register with the data coming from the recurrence matrix. We want to memorize all the diagonal binary patterns coming from the dot matrix with the corresponding indices. To understand better, we can illustrate an example in Figure 4.10. Together with the sequences, the corresponding indices are shown. Each diagonal of the matrix forms a binary pattern that can be considered as a “matching bit map”, with 1 for matching characters and 0 otherwise. The red framed area denotes the full-length diagonals that appear between the indices 0 and 3075 of the horizontal (*XL23808*) sequence and the overall vertical (*XLRHODOP*) sequence. For the remaining unframed areas (with shorter diagonal’s length) some extra paddings are added. Hence, the plot in Figure 4.10 has a total of 4750 diagonal data-sets, each of 1625 elements.

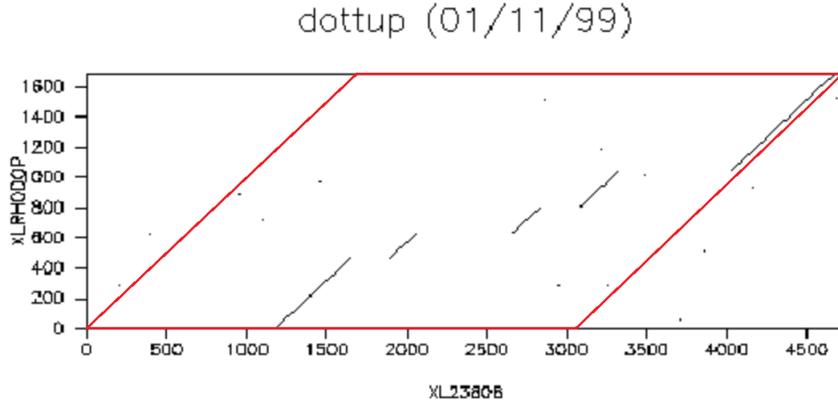


Figure 4.10. DNA sequence dot plot for the *XL23808* and *XLRHODOP* sequences [49].

Supposing that we can obtain N diagonal data-sets from the dot matrix, we can write our initial superposition state as follows:

$$|\psi\rangle = \frac{1}{\sqrt{N}} \sum_{i=0}^{N-1} |i\rangle \otimes |d_i\rangle \quad (4.47)$$

d_i denotes the diagonal at index i . For a correct initialization of the quantum register we can rely on the algorithm from [41] and already discussed in previous paragraphs. At this point we can perform queries using binary diagonal patterns as input. We can define a query set Q containing the M binary patterns we want to search. The oracle O_Q marking solution states is defined as follows:

$$O_Q = I - (1 - j) \sum_{d \in Q} |d\rangle \langle d| \quad (4.48)$$

j denotes the imaginary unit and I the identity matrix. The inversion about mean G is defined as follows:

$$G = (1 + j)|\psi\rangle \langle \psi| - jI \quad (4.49)$$

O_Q and G are applied inside Grover's iterations. The number of Grover's iterations is calculated as follows (the obtained result has to be rounded to the nearest integer):

$$T = \frac{\pi}{4} \sqrt{\frac{N}{M}} \quad (4.50)$$

We can summarize the algorithm's steps as follows:

1. form and encode $|\psi\rangle$ into a linear superposition state by learning the diagonal data-sets from recurrence dot matrix, together with their indices;
2. apply the oracle O_Q ;
3. apply the inversion about mean G ;
4. steps 2 and 3 are executed iteratively T times.

Chapter 5

Implementation and Testing

5.1 A brief introduction to the implementation and testing work

In this chapter we will illustrate the work that was done in order to test some of the algorithms explained in the previous chapter on the *IBM Quantum Experience* platform. First of all, we will briefly talk about the tools used, then we will talk about how we tested the chosen algorithms.

5.1.1 Tools used

We used *Jupyter Notebook* [17] as IDE for writing code in Python 3. Jupyter Notebook can be used from any Internet browser; in order to install it, we installed *Anaconda* [18] and setup a new Conda environment; in this environment we installed Python 3, Jupyter Notebook and other needed libraries. Among the installed libraries, the most important one is *Qiskit* [51], that is developed by IBM and gives us all the functions needed in order to setup and simulate (locally or remotely) or execute on a remote quantum computer the quantum circuits that we have built.

5.1.2 How testing was done

In order to test the algorithms, we have chosen the *HIV genome*. Some important considerations must be done in order to understand the testing results.

Judgment parameters

It's worth to warn the reader that *one of the adopted judgment parameters is the accuracy of the results*: we will run the chosen algorithms on quantum simulators, so judging them from a time execution point of view wouldn't make sense; also, another important parameter we must take into account is *the number of gates used to build the quantum circuit*, since higher is the number of gates, greater is the

probability of obtaining noisy results due to *decoherence* (we have briefly introduced the concept of *decoherence* in *Chapter 2*, and we will talk about it again in more depth later on in this chapter). The total required number of qubits also could be a judgment parameter: lower the number of qubits needed, the better (it's important to remember that today's real quantum computers are limited in the number of qubits they offer).

A brief note about HIV genome

The entire HIV genome has a total length that's greater than 9000 characters [52]: it's important to remark this because our test will be done on small pieces of the HIV genome, since doing testing on the entire HIV genome at once would require too many qubits (for example, the encoding of a character's index only would require at least 14 qubits).

It's worth to spend a few words on how the HIV genome for testing was obtained. We relied upon *ART* [60], that is a set of simulation tools to generate synthetic next-generation sequencing reads. *ART* simulates sequencing reads by mimicking real sequencing process with empirical error models or quality profiles summarized from large recalibrated sequencing data. *ART* is freely available to public and its binary packages are available for three major operating systems: Linux, Macintosh, and Windows. We used *ART-MountRainier-2016-06-05* (that is the latest version). We input the following commands from terminal to setup the system on a Linux PC and put ourselves in the folder containing the binaries.

```
wget https://www.niehs.nih.gov/research/resources/assets/  
      docs/artbinmountrainier2016.06.05linux64.tgz  
tar -xzf artbinmountrainier2016.06.05linux64.tgz  
cd art_bin_MountRainier/
```

At this point, we run the following command.

```
./art_illumina -ss HS25 -i ../..HIVepi/HIVgenome/HIVgenome.fa  
-p -l 150 -c 1000000 -m 200 -s 10 -o pair_dat
```

- `-p` specifies pair ended;
- `-c` specifies 1 million pair end reads;
- `-l` specifies length of 150 bp;
- `-m` specifies mean fragment size;
- `-s` specifies std deviation in fragment size;
- `-o` specifies the output prefix;

- `-ss` specifies the name of *Illumina* sequencing system of the built-in profile used for simulation (we used HS25 that stands for *HiSeq2500*).

The output of this process is a set of files with extensions `.fq` and `.aln` (`.fq` files contain the synthetic raw data, `.aln` files contain the sequences' original positions). At this point, we used these files as input for *Bowtie2* [60] in order to align the reads, obtaining a `.fa` file as output, containing the whole HIV genome; since this file can be opened as a common `.txt` file with *Notepad*, we easily converted the file in `.txt` format in order to use it easily in Python with the `open()`, `read()` and `close()` functions.

A brief note about the nature of the obtained results

We already said that we will judge the obtained results from an accuracy point of view and for the number of quantum gates needed by the algorithm. In order to understand better the obtained results that will be illustrated to the reader in this chapter, we must underline the fact that *the obtained results are statistical*: thanks to the Qiskit library, we can build a quantum circuit and specify the number of shots (*i.e.*, the number of times the algorithm is executed); the results obtained at every single shot are used to build an histogram where each column indicates the probability of obtaining the corresponding computational basis state after qubits measurement.

5.1.3 Chosen algorithms

We have chosen three algorithms for implementation and testing on the IBM Quantum Experience platform. The reasons we have chosen the following three algorithms are to be found in the fact that are the latest and better performing quantum algorithms for Pattern Matching available today. It's worth to warn the reader that we will test these algorithms but *we cannot do a direct comparison since the meaning of the results given by them is slightly different*.

- *Quantum Associative Memory with index measurement*: we implemented with Qiskit the *QAM* algorithm with a slight modification suggested by [46], in which the measurement is executed on the qubits encoding the index of a sub-sequence from the reference string; *given an incomplete search pattern with wildcard characters as input, we can obtain the index of a corresponding sub-sequence from the reference*; as already said, the algorithm tries to do *pattern completion*.
- *Quantum indexed Bidirectional Associative Memory*: it's the algorithm proposed by [46]; the obtained results are to be interpreted differently from the previous algorithm; *given a search pattern (this time wildcard characters are not used), the algorithm tries to find the indices of the sub-sequences that are most similar to the given input*; the meaning of the results is different from the previous algorithm and a direct comparison is not possible.

- *Quantum Pattern Recognition*: it’s the algorithm proposed by [49]; again, the results obtained with this algorithm are to be interpreted differently from the previous two algorithms; *given a recurrence dot matrix obtained from reference and search pattern by means of a spatial light modulator, a quantum database is built, in which the binary diagonals from the matrix (with the corresponding indices) are encoded; so, given a binary search pattern (that we could see as a “matching bit map”), the algorithm tries to find the indices of corresponding diagonals from the recurrence dot matrix (that correspond to the indices of the sub-sequences from the reference that match with the search pattern as specified by the diagonals of the recurrence dot matrix).*

5.1.4 Decoherence and number quantum of gates

The results that will be illustrated in this chapter are (for the most part) obtained by simulating the behaviour of an *ideal* quantum computer, so decoherence is not taken into account. However, decoherence is an important factor that affects the performance of a real quantum computer, so it’s worth to spend some words to talk about it in more depth.

To quantify decoherence, we can introduce two parameters T_1 and T_2 .

- T_1 helps to quantify how quickly the qubits experience energy loss due to environmental interaction (energy loss would result in a change in frequency, which would make coherent qubits experience decoherence);
- T_2 helps to quantify how quickly the qubits experience a phase change due to interaction with the environment (again, causing decoherence).

Energy relaxation is the loss of energy from the system, for example the process of a state with more energy decaying into another state with less energy. For example, T_1 measures the time for a state $|1\rangle$ with higher energy to become a state $|0\rangle$ with lower energy. Energy relaxation will always happen in a real quantum computer, and this process happens via exponential decay from the more energetic state to the less energetic state. In the illustrated example, we initially have 100% probability of being in state $|1\rangle$, but after some time t this probability has decreased exponentially to a value $e^{-\frac{t}{T_1}}$, where T_1 is a constant. Summarizing, we can write the following:

$$\text{Probability in state } |1\rangle = e^{-\frac{t}{T_1}} \quad (5.1)$$

So, T_1 is a measurement of how long energy relaxation takes to occur. As for T_2 , it affects only superposition states, since decoherence results from phase difference between two or more qubits that are coherent (that is, they are in superposition). Any environmental disturbance that causes phase changes can cause this sort of decoherence. Like T_1 , T_2 also is measured in terms of the exponential decay of our expected result over a period of time [2].

At this point, it should be clear to the reader that having bigger T_1 and T_2 is fundamental in order to have a more reliable quantum computation and that a quantum algorithm that executes its task with the lowest possible number of quantum gates (for executing the computation in the lowest time possible) is surely appreciable; on the other hand it should be noted that quantum computation needs time, particularly computations that require many steps (and many quantum gates). It is exactly the algorithms that require many steps which will enable Quantum Computing to be practically useful, so advancement in Quantum Computing hardware technology is fundamental.

5.2 Quantum Associative Memory

Python code for this algorithm is illustrated in *Appendix A*. In the example illustrated here, we tested the algorithm on the first 32 characters of the HIV genome, but other tests are illustrated in *Appendix A*. The following is the obtained output.

```
Reference genome: 32200222130033101311100020020100
Chosen pattern for testing: 2?3
Total number of qubits: 19
Number of ancilla qubits: 8
shots = 8192
Grover's algorithm had 1 iteration
Number of gates: 9295
Circuit depth: 7049
```

```
Tag: 00000 - Data: 322
Tag: 00001 - Data: 220
Tag: 00010 - Data: 200
Tag: 00011 - Data: 002
Tag: 00100 - Data: 022
Tag: 00101 - Data: 222
Tag: 00110 - Data: 221
Tag: 00111 - Data: 213
Tag: 01000 - Data: 130
Tag: 01001 - Data: 300
Tag: 01010 - Data: 003
Tag: 01011 - Data: 033
Tag: 01100 - Data: 331
Tag: 01101 - Data: 310
Tag: 01110 - Data: 101
Tag: 01111 - Data: 013
Tag: 10000 - Data: 131
Tag: 10001 - Data: 311
Tag: 10010 - Data: 111
Tag: 10011 - Data: 110
Tag: 10100 - Data: 100
```

Tag: 10101 – Data: 000
Tag: 10110 – Data: 002
Tag: 10111 – Data: 020
Tag: 11000 – Data: 200
Tag: 11001 – Data: 002
Tag: 11010 – Data: 020
Tag: 11011 – Data: 201
Tag: 11100 – Data: 010
Tag: 11101 – Data: 100

These data illustrate important information about this particular execution of the algorithm:

- the first 32 characters of HIV genome and the adopted search pattern with numeric encoding (0 for *A*, 1 for *C*, 2 for *G* and 3 for *T*);
- the total number of qubits (with ancilla qubits included in the calculation: they are used as support qubits by `mct()`, a Python function from Qiskit library that implements the multi-controlled *NOT*);
- the number of shots is the number of times the algorithm was executed in order to obtain a statistical result;
- the number of Grover’s iterations;
- the total number of gates used in the built quantum circuit;
- *the circuit depth*;
- the database obtained from the reference genome, with sub-sequences and corresponding indices in binary.

It’s worth to spend a few words about the number of gates and the circuit depth. The total number of gates applied in the quantum circuit does not correspond to the circuit depth: *the circuit depth is the length of the longest path from the input (or from a preparation) to the output (or a measurement gate), moving forward in time along qubit wires. The stopping points on the path are the gates, the allowed paths that must be considered can enter and exit those gates on any input or output, and the length is the number of jumps from each gate to the next gate along the path.* In alternative, we could say that *circuit depth is the maximum time from input to output assuming all gates take 1 unit of time* [53]. Our desire is to keep circuit depth as low as possible. Later on in this chapter, we will spend a few words about the existing correlation between circuit depth and noise in a real quantum computer.

Figure 5.1 shows the histogram obtained from the algorithm execution on HIV genome from 0 to 31 and search pattern equal to “2?3” over 8192 shots. We must warn the reader that the basis states reported on the *x* axis of the histogram are written in *Little Endian* notation (since Qiskit has adopted it). However, the output is correct: in the reference used for testing, only the sub-sequence at index 7 (in

binary 00111, that becomes 11100 in Little Endian) can satisfy the search pattern “ $2?3$ ”, so the peak in the histogram occurs at the right index.

In *Appendix A*, we illustrate other two examples of execution of the algorithm over other small pieces of HIV genome. In the first example, a small piece of genome from 1000 to 1031 is taken, with two possible sub-sequences satisfying the search pattern “ $2?3$ ” at indices 0 and 15. The algorithm correctly identifies them, with two peaks at the corresponding indices (look at Figure A.1). In the second example HIV genome from 2000 to 2031 is taken, with two possible sub-sequences satisfying the search pattern “ $2?3$ ” at indices 23 and 26. In this case also the algorithm correctly identifies them with two peaks at the corresponding indices (look at Figure A.2). In both examples, we have obtained a total number of gates and a circuit depth not so different from those obtained in the first example illustrated in this paragraph with HIV genome from 0 to 31. If we would increment the length of the reference genome and/or the length of the search pattern, the number of qubits would increase (making the simulation more difficult due to the higher computational resources needed), and also the number of gates (together with the circuit depth) would increase. However, the algorithm is always capable of identifying the correct solutions in all the proposed examples

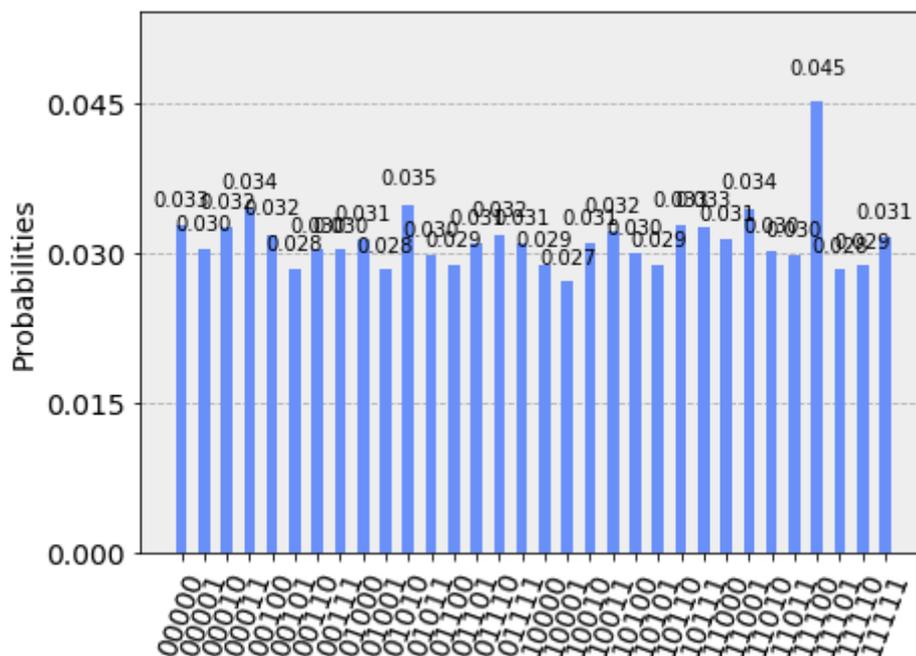


Figure 5.1. QAM execution: obtained probability histogram for HIV genome from 0 to 31 and search pattern equal to $2?3$ over 8192 shots.

5.3 Quantum indexed Bidirectional Associative Memory

Python code for this algorithm is illustrated in *Appendix B*. In this case also, we tested the algorithm on the first 32 characters of the HIV genome (other tests are illustrated in *Appendix B*). The following is the obtained output.

```
Reference genome: 32200222130033101311100020020100
Chosen pattern for testing: 203
Total number of qubits: 19
Number of ancilla qubits: 8
shots = 8192
Grover's algorithm had 2 iterations
Number of gates: 53801
Circuit depth: 42510

Tag: 00000 - Data: 322 - Hamming distance: 3
Tag: 00001 - Data: 220 - Hamming distance: 3
Tag: 00010 - Data: 200 - Hamming distance: 2
Tag: 00011 - Data: 002 - Hamming distance: 2
Tag: 00100 - Data: 022 - Hamming distance: 3
Tag: 00101 - Data: 222 - Hamming distance: 2
Tag: 00110 - Data: 221 - Hamming distance: 2
Tag: 00111 - Data: 213 - Hamming distance: 1
Tag: 01000 - Data: 130 - Hamming distance: 6
Tag: 01001 - Data: 300 - Hamming distance: 3
Tag: 01010 - Data: 003 - Hamming distance: 1
Tag: 01011 - Data: 033 - Hamming distance: 3
Tag: 01100 - Data: 331 - Hamming distance: 4
Tag: 01101 - Data: 310 - Hamming distance: 4
Tag: 01110 - Data: 101 - Hamming distance: 3
Tag: 01111 - Data: 013 - Hamming distance: 2
Tag: 10000 - Data: 131 - Hamming distance: 5
Tag: 10001 - Data: 311 - Hamming distance: 3
Tag: 10010 - Data: 111 - Hamming distance: 4
Tag: 10011 - Data: 110 - Hamming distance: 5
Tag: 10100 - Data: 100 - Hamming distance: 4
Tag: 10101 - Data: 000 - Hamming distance: 3
Tag: 10110 - Data: 002 - Hamming distance: 2
Tag: 10111 - Data: 020 - Hamming distance: 4
Tag: 11000 - Data: 200 - Hamming distance: 2
Tag: 11001 - Data: 002 - Hamming distance: 2
Tag: 11010 - Data: 020 - Hamming distance: 4
Tag: 11011 - Data: 201 - Hamming distance: 1
Tag: 11100 - Data: 010 - Hamming distance: 4
Tag: 11101 - Data: 100 - Hamming distance: 4
```

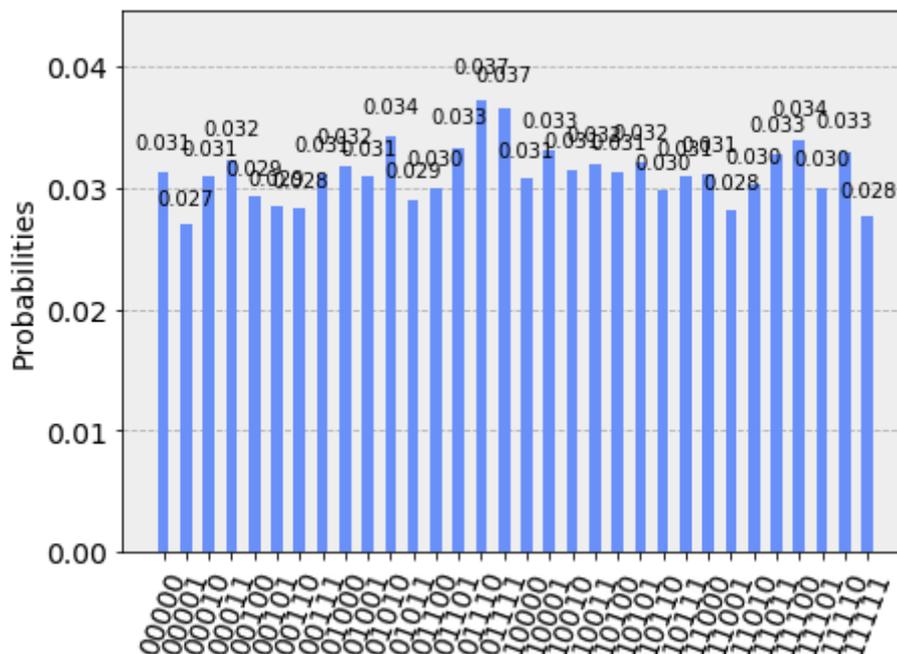


Figure 5.2. QiBAM execution: obtained probability histogram for HIV genome from 0 to 31 and search pattern equal to 203 over 8192 shots.

The obtained histogram is very different from the one obtained for QAM, and the reason is to be found in the goal that this algorithm wants to reach: it tries to find the indices of the sub-sequences most similar to the given search pattern. This similarity is evaluated by means of the Hamming distance with respect to the search pattern. Looking at the database obtained from the reference genome, the sub-sequences with lower Hamming distance are at index 7 (00111 in binary), at index 10 (01010 in binary) and at index 27 (11011 in binary). Looking at the histogram, the basis states with higher probability are the following (the indices are reported in binary and in decimal):

1. 01110 \rightarrow 14, *Hamming distance* = 3, 3.7%;
2. 11110 \rightarrow 30, *Spurious State*, 3.7%;
3. 00111 \rightarrow 7, *Hamming distance* = 1, 3.4%;
4. 01010 \rightarrow 10, *Hamming distance* = 1, 3.4%;
5. 11011 \rightarrow 27, *Hamming distance* = 1, 3.3%;
6. 01111 \rightarrow 15, *Hamming distance* = 2, 3.3%;
7. 10001 \rightarrow 17, *Hamming distance* = 3, 3.3%;
8. 10110 \rightarrow 22, *Hamming distance* = 2, 3.3%.

We already explained in the previous chapter that this algorithm, in its initialization process, allows the presence of spurious states in the quantum database:

that’s why in this execution of the algorithm a spurious state has appeared. We can also see that the algorithm fulfills its goal, although it is not perfect: not only a spurious state has appeared, but the highest probability came from a sub-sequence with Hamming distance equal to 3, while states with Hamming distance equal to 1 are immediately after.

In *Appendix B* we propose other two examples over other two small HIV genome pieces (taking *203* as search pattern in both cases). In the first example HIV genome from 1000 and 1031 is taken: looking at the output data (available in the appendix) there are some sub-sequences with Hamming distances equal to 0 and 1, so we would expect to have peaks in the final histogram at the corresponding indices (for example at indices 0, 15, 21, 23 and 26). Looking at Figure B.1, the highest peaks are (from highest to lowest) at indices 15, 30, 11, 0, 21 and 6. In this example also the identification of the solution states is not perfect: we would expect the indices denoting sub-sequences with zero Hamming distance to have the highest peaks, but only some of them are identified and in some cases they are surpassed by other states with higher Hamming distances or even spurious states (index 30 denotes in this example a spurious state). In the second example proposed in the appendix, HIV genome from 2000 and 2031 is taken: looking at the output data, the sub-sequence with the lowest Hamming distance (equal to 1) is at index 23. Other sub-sequences with lower Hamming distances are at indices 5, 6, 7, 12, 15, 19, 21, 22, 25 and 26 (all with Hamming distances equal to 2). Looking at the final histogram illustrated in Figure B.2, the highest peaks are (from higher to lower) at indices 3, 30, 29, 20, 6, 5, 7, 15. In this example also, the behaviour of the algorithm seems to be the same: the state denoting the index corresponding to the sub-sequence with the lowest Hamming distance is not even between the highest peaks, while some states with Hamming distance equal to 2 appear between the highest peaks, but surpassed by other states with higher Hamming distances or even spurious (like index 30).

In general, the algorithm seems not to behave always correctly. Also, we should highlight an interesting fact: the goal of the algorithm is to find the sub-sequences that are most similar to the adopted search pattern, but in order to do so it adopts Hamming distance as metric. We already said that Hamming distance is the number of places where two strings of equal length are different; *the algorithm calculates the Hamming distances between the sub-sequences from reference and the search pattern considering their binary encoding, but this does not correspond to a biological similarity but only to an encoding similarity, so biology is not taken into consideration in measuring similarity between sub-sequences and search pattern.*

It’s worth to spend a few words about the number of gates and the circuit depth that was output in the first proposed execution of this algorithm. The original implementation proposed in [46] by the authors for the *QX Simulator* (a quantum simulator by *QuTech*, another company involved in the Quantum Computing field) uses the *Quantum Shannon Decomposition* for decomposing arbitrary unitary matrices, as described in [54] by V. V. Shende, S. S. Bullock and I. L. Markov. In the Qiskit implementation proposed in *Appendix B*, we were able to use the needed unitary matrix implementing the `oracle()` function directly, without decomposing it. However, this means that the number of gates and the circuit depth that was output by the algorithm needs to be corrected, since Qiskit considers a given

arbitrary unitary matrix as a single gate.

We can mention the following theorem about Quantum Shannon Decomposition from [54].

“An arbitrary n -qubit operator can be implemented by a circuit containing three multiplexed rotations and four generic $(n - 1)$ -qubit operators, which can be viewed as co-factors of the original operator.”

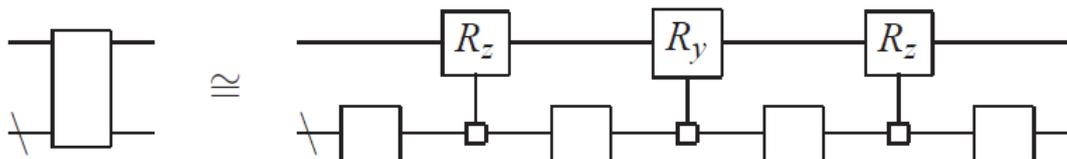


Figure 5.3. Quantum Shannon Decomposition [54].

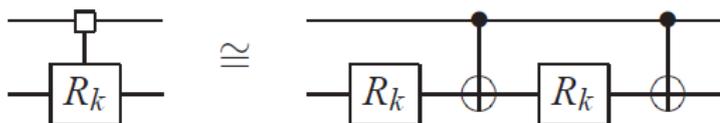


Figure 5.4. Corresponding circuit for multiplexed rotation gate [54].

Figure 5.3 shows the corresponding circuit for an arbitrary n -qubit operator, while Figure 5.4 shows the corresponding circuit for a multiplexed rotation gate (formed by two ordinary rotation gates and two $CNOT$ s). This theorem gives us a hint to calculate an estimation of the number of gates needed to implement the `oracle()` function that we applied directly in our implementation by means of its unitary matrix. In this particular execution of the QiBAM algorithm, the `oracle()` function applies an 4-qubit operator to the circuit, and the `oracle()` function was called 3 times in total (1 time before Grover’s iterations and 2 times inside them). According to the QSD theorem, to decompose an arbitrary 4-qubit gate we should need the following number of gates:

$$N_g^{4-qubit} = 3N_{mr} + 4(3N_{mr} + 4(3N_{mr} + 4N_{zyz})) \quad (5.2)$$

N_{mr} is the number of gates needed for a multiplexed rotation gate and is equal to 4 (2 $CNOT$ s gates plus 2 ordinary rotation gates); N_{zyz} is the number of gates needed to decompose an arbitrary single qubit gate by means of the ZYZ Decomposition [54] and is equal to 3. Doing the calculation, we will obtain $N_g^{4-qubit} = 444$. We can recalculate the total number of gates as follows:

$$Total\ number\ of\ gates = 53801 - 3 + 3N_g^{4-qubit} = 55130 \quad (5.3)$$

As for the circuit depth, we can assume that it can be estimated as the sum of the *CNOT* gates from the multiplexed rotation gates and the rotation gates due to the *ZYZ* Decomposition. We can estimate the circuit depth in the QSD for an arbitrary 4-qubit gate as follows:

$$\textit{Estimated circuit depth} = 3\frac{N_{mr}}{2} + 4\left(3\frac{N_{mr}}{2} + 4\left(3\frac{N_{mr}}{2} + 4N_{zyz}\right)\right) \quad (5.4)$$

Doing the calculation (and remembering that $N_{mr} = 4$ and $N_{zyz} = 3$) we obtain that, for an arbitrary 4-qubit gate, the estimated circuit depth is 318. So, we can estimate the total circuit depth for this particular QiBAM execution as follows:

$$\textit{Total estimated circuit depth} = 42510 - 3 + 3 \cdot 318 = 43461 \quad (5.5)$$

The other two example proposed in *Appendix B* have total number of gates and circuit depth not so different from those obtained in the example considering HIV genome from 0 to 31.

5.4 Quantum Pattern Recognition

Python code for this algorithm is illustrated in *Appendix C*. In this case, we tested the algorithm on the recurrence dot matrix formed by the first 32 characters of the HIV genome as horizontal sequence and the pattern *GAT* as vertical sequence. We searched for 4 binary patterns: 111, 110, 011 and 101 (other tests done by searching the same patterns but on recurrence dot matrices formed with different pieces of HIV genome are illustrated in *Appendix C*). The following is the obtained output.

```
|TGGAAGGGCTAATTCCTCCCAAAGAAGACAA
-----
G|-XX--XXX-----X--X----
A|---XX-----XX---X-----XXX-XX-X-XX
T|X-----X--XX---X-----
```

```
Total number of qubits: 18
qr = |t, x, g, c, a>
Size of t: 5
Size of x: 3
Size of g: 7
Size of c: 2
Size of a: 1
shots = 1024
Grover's algorithm had 2 iterations
Number of gates: 1577
Circuit depth: 685
```

Size of Learning Set: 32

Chosen patterns for testing: [1 1 1], [1 1 0], [0 1 1], [1 0 1]

```
0->00000: [0 0 0]
1->00001: [1 0 0]
2->00010: [1 1 0]
3->00011: [0 1 0]
4->00100: [0 0 0]
5->00101: [1 0 0]
6->00110: [1 0 0]
7->00111: [1 0 1]
8->01000: [0 0 0]
9->01001: [0 1 0]
10->01010: [0 1 1]
11->01011: [0 0 1]
12->01100: [0 0 0]
13->01101: [0 0 0]
14->01110: [0 1 0]
15->01111: [0 0 1]
16->10000: [0 0 0]
17->10001: [0 0 0]
18->10010: [0 0 0]
19->10011: [0 0 0]
20->10100: [0 1 0]
21->10101: [0 1 0]
22->10110: [0 1 0]
23->10111: [0 0 0]
24->11000: [1 1 0]
25->11001: [0 1 0]
26->11010: [0 0 0]
27->11011: [1 1 0]
28->11100: [0 0 0]
29->11101: [0 1 0]
30->11110: [0 1 0]
31->11111: [0 0 0]
```

In the output of the algorithm, we can see the built recurrence dot matrix. It's worth to warn the reader that in the original paper [49] this matrix is built by means of a *spatial light modulator* (that allows to obtain recurrence dot matrices quicker than the classical way, even for long sequences), but we hadn't such a tool for testing, so we decided to build a little matrix classically in order to do some testing. *Since we have built the recurrence dot matrix classically for the sake of testing, the time cost for doing it shouldn't be included in the total time cost of the algorithm.*

The quantum register ($|t, x, g, c, a\rangle$) is organized in different portions: t contains the index of a diagonal from the recurrence dot matrix; x contains the elements of the diagonal itself, organized as a linear array; g and c are used as

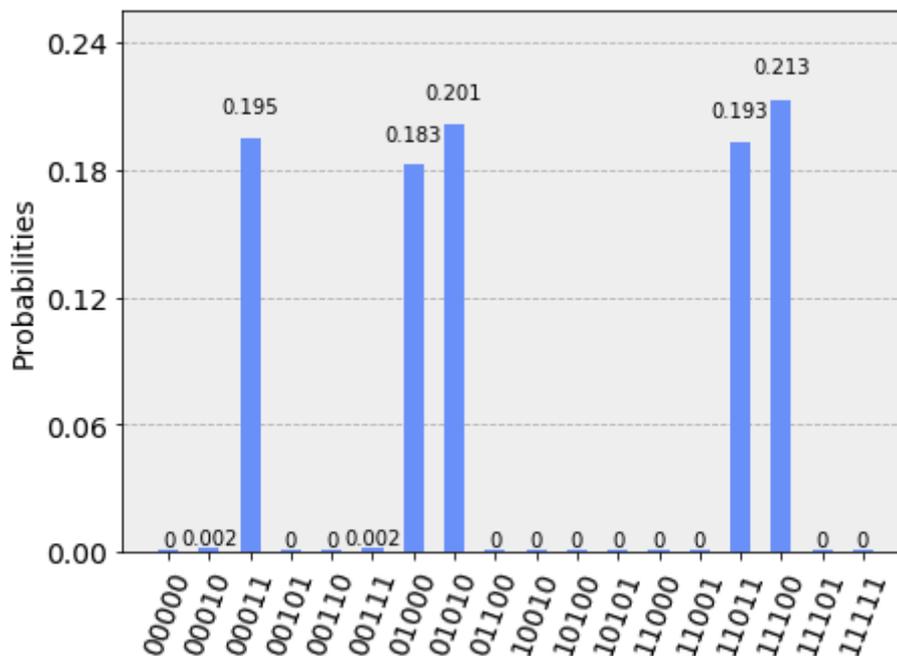


Figure 5.5. QPR execution: obtained probability histogram for recurrence dot matrix between HIV genome from 0 to 31 and pattern *GAT*, searching patterns *111*, *110*, *011* and *101* over 1024 shots.

support by the initialization phase; finally, a is used as support by `mct()`.

The **Learning Set** in the output data corresponds to the set formed by the extracted diagonals from the recurrence dot matrix together with their corresponding indices. Since the horizontal sequence length is 32, the size of the learning set is 32.

The results histogram in Figure 5.5 shows five peaks, while all the other basis states have practically zero probability: the five peaks correspond to 11000, 00010, 01010, 11011 and 00111; in decimal, they correspond to indices 24, 2, 10, 27 and 7. Looking at the learning set database, we can verify that the obtained indices correspond to the solutions.

Other two examples are proposed in *Appendix C*, taking into consideration HIV genome from 1000 to 1031 and from 2000 to 2031. In these two examples also, the adopted search patterns were *111*, *110*, *011* and *101*, with obtained histograms illustrated in Figure C.1 and C.2. In both cases the algorithm correctly identifies the solutions, while other non-solution states have practically zero probability, which means that the algorithm seems to work very well.

In this case also, we need to recalculate the total number of gates and the circuit depth as seen in the previous paragraph. In the first proposed execution of this algorithm, the `oracle()` and `inversionAboutMean()` functions build two unitary matrices acting on 3 qubits and 8 qubits respectively. Knowing this, we can calculate the number of gates needed to decompose arbitrary unitary 8-qubit and 3-qubit operators by means of QSD as follows (remembering that $N_{mr} = 4$ and

$N_{zyz} = 3$):

$$N_g^{8-qubit} = 3N_{mr} + 4(3N_{mr} + 4N_{zyz})))))) \quad (5.6)$$

$$N_g^{3-qubit} = 3N_{mr} + 4(3N_{mr} + 4N_{zyz}) \quad (5.7)$$

Doing the calculation, we obtain $N_g^{8-qubit} = 114684$ and $N_g^{3-qubit} = 108$. The two unitary matrices built by `oracle()` and `inversionAboutMean()` are applied inside Grover's iterations, which were equal to 2 in this execution of this algorithm, so we can say that the two unitary matrices are applied 2 times each. Knowing this, we can calculate the following:

$$Total\ number\ of\ gates = 1577 - 4 + 2N_g^{8-qubit} + 2N_g^{3-qubit} = 231157 \quad (5.8)$$

As we can see, the result is radically different (a lot higher) than the previous one. As for the circuit depth, we can estimate it as follows (doing the same assumptions that we made in the previous paragraph):

$$Estimated\ circuit\ depth_{8-qubit\ operator} = 3\frac{N_{mr}}{2} + 4\left(3\frac{N_{mr}}{2} + 4\left(3\frac{N_{mr}}{2} + 4\left(3\frac{N_{mr}}{2} + 4\left(3\frac{N_{mr}}{2} + 4\left(3\frac{N_{mr}}{2} + 4N_{zyz}\right)\right)\right)\right)\right) \quad (5.9)$$

$$Estimated\ circuit\ depth_{3-qubit\ operator} = 3\frac{N_{mr}}{2} + 4\left(3\frac{N_{mr}}{2} + 4N_{zyz}\right) \quad (5.10)$$

Doing the calculation, we obtain $Estimated\ circuit\ depth_{8-qubit\ operator} = 81918$ and $Estimated\ circuit\ depth_{3-qubit\ operator} = 78$. At this point we can estimate the total circuit depth as follows:

$$Total\ estimated\ circuit\ depth = 685 - 4 + 2 \cdot 81918 + 2 \cdot 78 = 164673 \quad (5.11)$$

The total estimated circuit depth also is a lot higher than what was output by the algorithm's Qiskit implementation. *In terms of needed gates, this is the most demanding algorithm between those examined in this thesis.* The examples proposed in *Appendix C* have a total number of gates and circuit depth not so different from those obtained in the example considering HIV genome from 0 to 31.

5.5 A brief summary about analyzed algorithms

In this paragraph we try to summarize the considerations we have done about the analyzed algorithms. *Quantum Associative Memory* and *Quantum Pattern Recognition* are the two algorithms that work better considering their initial goals: they always identify (in the considered examples) the right solutions. If we take into account the total number of gates and the circuit depth, QPR algorithm is, without any doubt, the most demanding in terms of resources: this is an important consideration that must be highlighted and that could denote a disadvantage of QPR (the next paragraph will talk in more depth about this argument, highlighting a correlation between noise, total number of gates and circuit depth in real quantum computation); also, we need to consider that QPR needs a *spatial light modulator* in order to obtain the recurrence dot matrix, while the other analyzed algorithms do not need any additional tool. QAM was the “lightest” algorithm in terms of resources, since it requires a much lower number of gates than the other two algorithms, and in our testing always performed well.

Quantum indexed Bidirectional Associative Memory was the algorithm that presented some problems during testing: its initial goal was to find the most similar sub-sequences to the search pattern from reference genome, considering Hamming distances between them, but we have seen that the algorithm not always gave us the correct results, with states having higher Hamming distances or even spurious surpassing solutions states. Also, we need to consider that calculating Hamming distance between binary encoding of sub-sequences from reference and search pattern gives us a similarity measure only from an encoding point of view, since biology is totally ignored in this process.

5.6 An example of noisy quantum computation

In this paragraph we will illustrate an example of quantum computation affected by noise; in order to do so, we have run the QAM algorithm on the first 16 characters of HIV genome, in order to lower the number of needed qubits below the threshold of 14 qubits; we have chosen this threshold because the highest number of qubits freely offered by the IBM Quantum Experience cloud platform for real quantum computation is 14, from the `ibmq_16_melbourne` backend. We have executed the computation on the remote `ibmq_qasm_simulator`, applying a noise model obtained from the available data about `ibmq_16_melbourne` (Python code is available at *Appendix A*).

```
Reference genome: 3220022213003310
Chosen pattern for testing: 2?
Total number of qubits: 13
Number of ancilla qubits: 5
shots = 1024
Grover's algorithm had 1 iteration
Number of gates: 3031
```

Circuit depth: 2275

Tag: 0000 - Data: 32
 Tag: 0001 - Data: 22
 Tag: 0010 - Data: 20
 Tag: 0011 - Data: 00
 Tag: 0100 - Data: 02
 Tag: 0101 - Data: 22
 Tag: 0110 - Data: 22
 Tag: 0111 - Data: 21
 Tag: 1000 - Data: 13
 Tag: 1001 - Data: 30
 Tag: 1010 - Data: 00
 Tag: 1011 - Data: 03
 Tag: 1100 - Data: 33
 Tag: 1101 - Data: 31
 Tag: 1110 - Data: 10

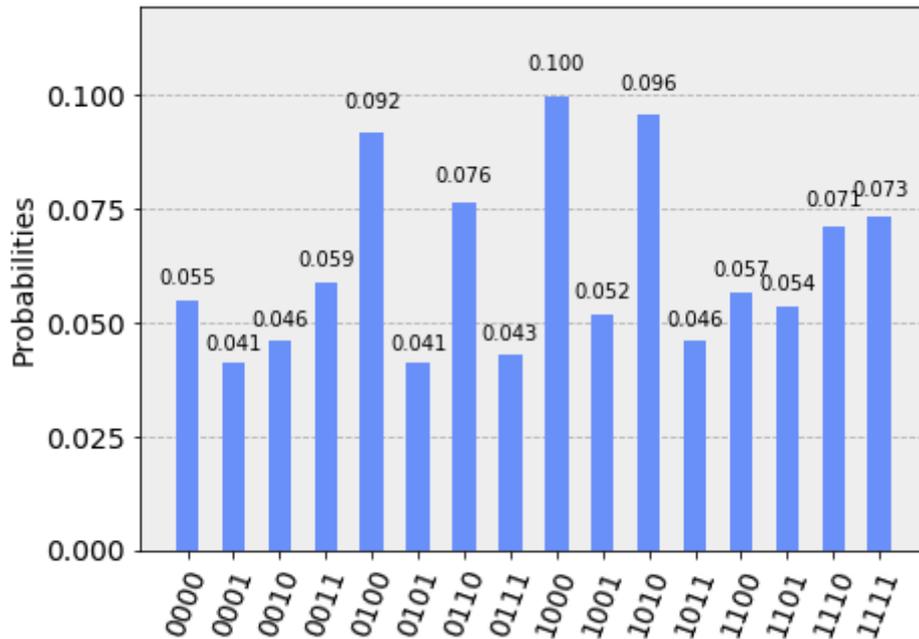


Figure 5.6. QAM: ideal computation for HIV genome from 0 to 15 and search pattern $2^?$ over 1024 shots.

Figure 5.6 shows the ideal computation of this algorithm, where the solution states are correctly identified (peaks at 0001, 0010, 0101, 0110 and 0111). Executing again the algorithm with the mentioned noise model applied gives us the histogram in Figure 5.7: the peaks are no more clearly distinguishable and the noise caused a loss of information. Figure 5.8 shows the comparison between the two histograms, with ideal computation in red and real (noisy) computation in orange. We will now talk in more depth about this phenomenon, in relation to the total number of gates and the circuit depth parameter (that in this execution were equal to 3031 and 2275 respectively).

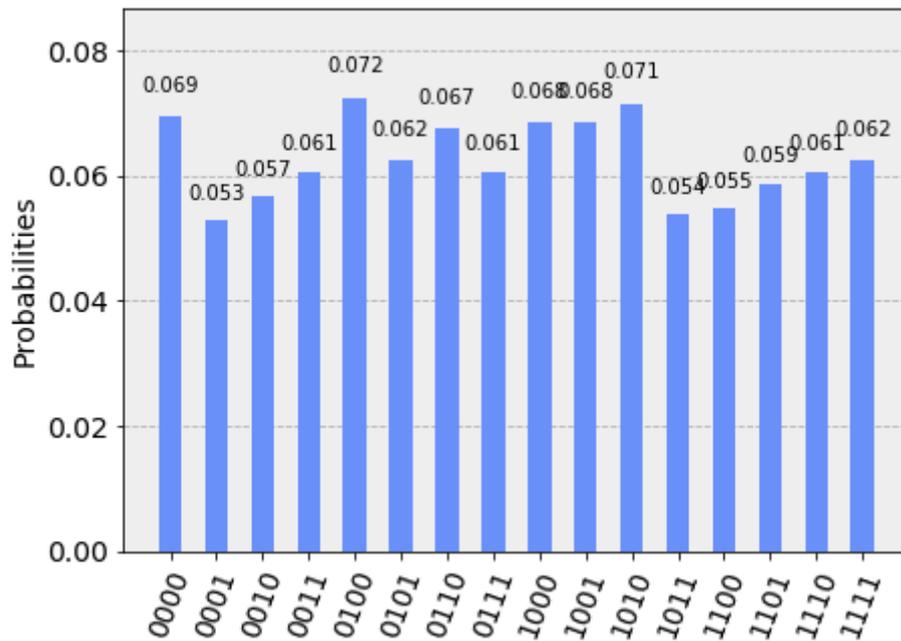


Figure 5.7. QAM: real computation (with noise) for HIV genome from 0 to 15 and search pattern 2^2 over 1024 shots.

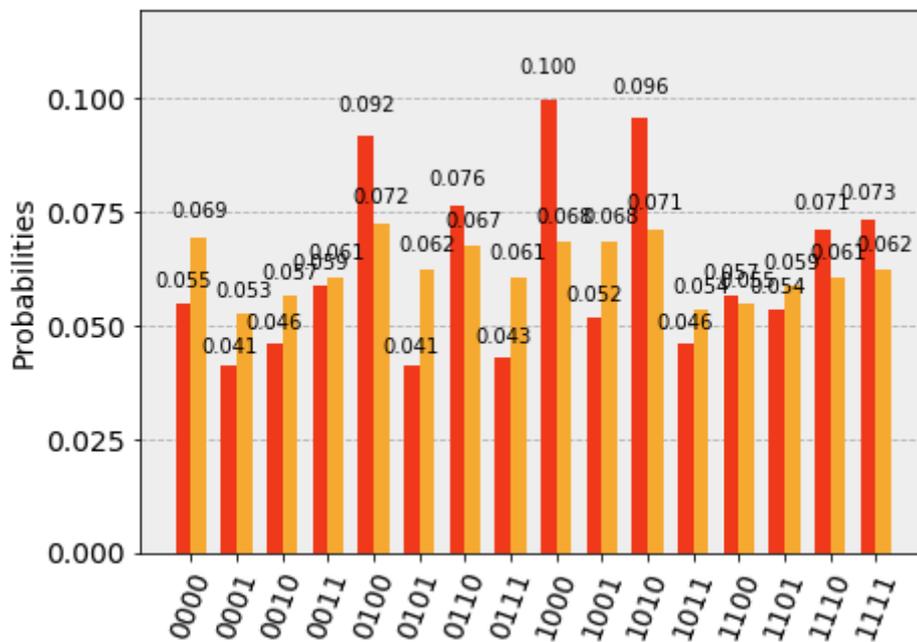


Figure 5.8. QAM: ideal computation vs. real computation comparison.

5.6.1 Noise, number of gates and circuit depth

Suppose we want to execute a generic quantum computation by means of a quantum circuit composed by 300 gates (a much lower number than those obtained in the testing of the algorithms in the previous paragraphs). From the public data available about *IBM Q16 Melbourne*, we can mention the following two parameters:

- Mean gate error: 2.14×10^{-3} ;
- Mean measure error: 2.68×10^{-2} .

If our circuit contains 300 gates, then the probability that at least one gate fails is given by the following:

$$\begin{aligned}
 P(\text{at least 1 gate fails}) &= 1 - P(\text{all gates succeed}) \\
 &= 1 - P(1 \text{ gate succeed})^{300} \\
 &= 1 - P(1 - 2.14 \times 10^{-3})^{300} \\
 &\approx 0.474
 \end{aligned} \tag{5.12}$$

This means that, *without accounting for decoherence errors*, our circuit fails nearly one time over two (47%). This is a first important piece of information that gives us an idea of the problem of noise in quantum computation.

Now, speaking about the errors due to decoherence, we can try to compute the time needed by the *IBM Q16 Melbourne* to finish the execution of our circuit. The basis gates set used by the *IBM Q16 Melbourne* to realize quantum circuits and translate more complicated gates into simpler ones is composed by $U1$, $U2$, $U3$ and $CNOT$ (more information about U gates can be found at [56]). From public data about the *IBM Q16 Melbourne* available at [57] and [58], we can obtain the required data to calculate the following:

- time needed for $U1$: $0ns$;
- time needed for $U2$: $100ns + 20ns = 120ns$;
- time needed for $U3$: $100ns + 20ns + 100ns + 20ns = 240ns$;
- rough average time needed for $CNOT$: $100ns + 20ns + 360ns + 20ns + 100ns + 20ns + 360ns + 20ns = 1000ns$.

Now, suppose we have a quantum circuit with a depth equal to 90 (a much lower number than those obtained in the testing of the algorithms in the previous paragraphs) with gates from *Qiskit* (i.e., I , X , Y , Z , H , S , S^\dagger , T , T^\dagger and $CNOT$). Since these gates will be translated by means of the gates in the *IBM Q16 Melbourne* basis gates set, we can say that the real circuit depth is approximately equal to 90. Now, let's suppose that this circuit depth is composed for 80% by $CNOT$ gates, 15% by $U3$ gates and 5% $U2$ gates: this means that we have 72 $CNOT$ gates, 14 $U3$ gates and 4 $U2$ gates. By doing the necessary calculation, we can

find that our circuit will take approximately $75.84\mu s$. Knowing that the coherence times for Melbourne are $T_1 = 71.5\mu s$ and $T_2 = 21.4\mu s$, we can infer that *we have a relatively low probability to execute the whole circuit without at least one error due to decoherence*. In conclusion, we could say that without accounting for decoherence errors we have approximately a 50% chance of executing successfully the circuit; on the other hand, with decoherence errors this probability would probably be greatly reduced [55].

It's worth to notice that the example made above stands for a total number of gates equal to 300 and a circuit depth of approximately 90 gates. The algorithms that we have seen in this thesis have both the total number of gates and circuit depth a lot higher than those in the example above: it shouldn't be surprising that trying the QAM algorithm with the *IBM Q16 Melbourne* noise model gave us a final results histogram greatly affected by noise and a resulting loss of information.

Chapter 6

Conclusions and future developments

6.1 Present and Future of Quantum Computing

In this thesis we talked about the current status of the research in the Quantum Computing field and we have seen a practical application of it to a very important field for medicine and biology, that is Genome Sequencing, implementing and testing some existing quantum algorithms developed for pattern matching in the genome alignment context. We have seen that Quantum Computing seems a promising field of research, that could possibly revolutionize various fields where it could be applied. Not only Genome Sequencing could benefit from Quantum Computing; we can mention the following potential applications [59]:

- *Artificial Intelligence*: AI is based on the principle of learning from experience, becoming more accurate as feedback is given; this feedback is based on calculating the probabilities for many possible choices, so AI is an ideal candidate for quantum computation;
- *Molecular Modeling*: the goal of precision modeling of molecular interactions is to find the optimal configurations for chemical reactions; this task is so complex that only the simplest molecules can be analyzed by today's digital computers; on the other hand, chemical reactions are quantum in nature as they form highly entangled quantum superposition states, so fully-developed quantum computers would not have any difficulty evaluating even the most complex processes;
- *Cryptography*: in *Chapter 2* we briefly talked about *Shor's algorithm* and the fact that it can factorize large numbers into primes much more efficiently than a classical computer; this means that with Quantum Computing today's methods for security could quickly become obsolete; it's worth to mention that in August 2015 the *NSA* began introducing a list of quantum-resistant cryptography methods that would resist quantum computers, and in April 2016 the *National Institute of Standards and Technology* began a public evaluation

process lasting four to six years; Quantum Computing gives us the chance to develop new *quantum encryption methods*;

- *Financial Modeling*: modern markets are some of the most complicated systems in existence; we have developed scientific and mathematical tools to approach them, but one major disadvantage affecting them is the absence of a controlled setting in which to run experiments; to solve this, investors and analysts have turned to Quantum Computing, since *the randomness inherent to quantum computers is congruent to the stochastic nature of financial markets*; investors would like to evaluate the distribution of outcomes under an extremely large number of scenarios generated at random;
- *Weather Forecasting*: some researchers have studied the existing relation between weather and the *GDP (Gross Domestic Product)* of a State, highlighting the fact that an important amount of *GDP* is directly or indirectly affected by weather, impacting food production, transportation and retail trade, among others; the ability to better predict the weather would have enormous benefit to many fields, not to mention more time to take cover from disasters; on the other hand, the equations governing weather processes contain many variables, making classical simulation lengthy; *quantum computers could help in building better climate models that could give us more insight into how humans are influencing the environment*; these models can help us in estimating future warming and determining what steps need to be taken now to prevent disasters;
- *Particle Physics*: models of particle physics are often extraordinarily complex, making pen-and-paper solutions not suitable and requiring much computing time for numerical simulation; this makes them ideal for Quantum Computing and researchers have already been taking advantage of this.

The above list is by no means exhaustive, and many more applications could be mentioned. However, we have seen that current real quantum computers are affected by some important limitations, like the maximum number of available qubits and the noise affecting computations involving many quantum gates. Many actors are involved in the Quantum Computing research, like Google, IBM and Microsoft (to mention the most important ones). Also, we have seen that anyone who wishes to experiment with Quantum Computing and real quantum computers can rely on cloud platforms, like *IBM Quantum Experience* (used in this thesis). Although current Quantum Computing is affected by the above mentioned limitations, it remains an exciting research field for physicists, engineers and computer scientists. Nobody can now make reliable predictions about the future of Quantum Computing, but the potential benefits that could come from it are a big push to the research and maybe, one day, Quantum Computing could become the revolution that will change the future of mankind.

Appendix A

Quantum Associative Memory

A.1 Python code

```
### Quantum Associative Memory by D. Ventura, T. Martinez
### Repository reference: https://gitlab.com/prince-ph0en1x/QaGs
    (by A. Sarkar)

## Importing libraries

%matplotlib inline
import qiskit
from qiskit import IBMQ
from qiskit import Aer
from qiskit import QuantumCircuit, ClassicalRegister,
    QuantumRegister, QiskitError
from qiskit.tools.visualization import circuit_drawer
from qiskit.tools.visualization import plot_histogram
from qiskit.providers.aer import noise

import random
import matplotlib
import math
from math import *

## Defining some auxiliary functions

def convertToNumEncoding(genome_str):
    bin_str = ""
    for i in range(0, len(genome_str)):
        if genome_str[i] == 'A':
            bin_str = bin_str + "0"
        elif genome_str[i] == 'C':
            bin_str = bin_str + "1"
        elif genome_str[i] == 'G':
```

```
        bin_str = bin_str + "2"
    elif genome_str[i] == 'T':
        bin_str = bin_str + "3"
    return bin_str

## Initializing global variables

# Alphabet set (0, 1, 2, 3) := {A, C, G, T} for DNA Nucleotide
bases
AS = {'00', '01', '10', '11'}
A = len(AS)

genome_file = open("HIVgenome.txt", "r")
RG = genome_file.read()
genome_file.close()

RG = convertToNumEncoding(RG)
RG = RG[0:32]
N = len(RG)

# Short Read search string
SR = "2?3"

M = len(SR)

Q_A = ceil(log2(A))

# Data Qubits
Q_D = Q_A * M

# Tag Qubits
Q_T = ceil(log2(N - M + 1))

# Ancilla Qubits
Q_anc = 8

# Ancilla qubits ids
anc = []

for qi in range(0, Q_anc):
    anc.append(Q_D + Q_T + qi)

# Total number of qubits
Q = Q_D + Q_T + Q_anc

## Initialization of IBM QX

IBMQ.enable_account('INSERT TOKEN HERE')
```

```
provider = IBMQ.get_provider()

# Choose a real device to simulate
device = provider.get_backend('ibmq_16_melbourne')
properties = device.properties()
coupling_map = device.configuration().coupling_map

# Generate an Aer noise model for device
noise_model = noise.device.basic_device_noise_model(properties)
basis_gates = noise_model.basis_gates

# Pick an available backend
# If this isn't available pick a backend whose name contains
# '_qasm_simulator' from the output above
backend = provider.get_backend('ibmq_qasm_simulator')

# Uncomment if you want to use local simulator
#backend= Aer.get_backend('qasm_simulator')

## Defining functions

def QAM():
    print('Reference genome:', RG)
    print('Chosen pattern for testing:', SR)
    print('Total number of qubits:', Q)
    print('Number of ancilla qubits:', Q_anc)
    qr = qiskit.QuantumRegister(Q)
    cr = qiskit.ClassicalRegister(Q_T)
    qc = qiskit.QuantumCircuit(qr, cr)

    # Patterns are stored
    generateInitialState(qc, qr)

    # Patterns are turned into Hamming Distances
    evolveToHammingDistances(qc, qr)

    # Marking the zero Hamming Distance states
    markZeroHammingDistance(qc, qr)

    inversionAboutMean(qc, qr)

    # Applying again this functions turns back the Hamming
    # Distances into the original patterns
    evolveToHammingDistances(qc, qr)

    # Marking the stored patterns
    markStoredPatterns(qc, qr)
```

```

# From patterns to Hamming Distances again
evolveToHammingDistances(qc, qr)

inversionAboutMean(qc, qr)

# Grover's iterations
it = 0
for i in range(0, 1):
    markZeroHammingDistance(qc, qr)
    inversionAboutMean(qc, qr)
    it = it + 1

print("Grover's algorithm had {} iterations.".format(int(it)))
finalGroverMeasurement(qc, qr, cr)
return qc

# Initialization as suggested by L. C. L. Hollenberg
def generateInitialState(qc, qr):
    for qi in range(0, Q_T):
        qc.h(qr[qi])

    control_qubits = []
    for ci in range(0, Q_T):
        control_qubits.append(qr[ci])

    ancilla_qubits = []
    for qi in anc:
        ancilla_qubits.append(qr[qi])

    for qi in range(0, N - M + 1):
        qis = format(qi, '0' + str(Q_T) + 'b')
        for qisi in range(0, Q_T):
            if qis[qisi] == '0':
                qc.x(qr[qisi])
        wMi = RG[qi:qi + M]
        print("Tag: {} - Data: {}".format(qis, wMi))
        for wisi in range(0, M):
            wisia = format(int(wMi[wisi]), '0' + str(Q_A) + 'b')
            for wisiai in range(0, Q_A):
                if wisia[wisiai] == '1':
                    qc.mct(control_qubits, qr[Q_T + wisi * Q_A +
                        wisiai], ancilla_qubits)
        for qisi in range(0, Q_T):
            if qis[qisi] == '0':
                qc.x(qr[qisi])
    return

# Hamming Distances

```

```

def evolveToHammingDistances(qc, qr):
    for pi in range(0, M):
        if SR[pi] == '?':
            continue
        ppi = format(int(SR[pi]), '0' + str(Q_A) + 'b')
        for ppil in range(0, Q_A):
            if ppi[ppil] == '1':
                qc.x(qr[Q_T + pi * Q_A + ppil])
    return

# Oracle to mark zero Hamming Distance
def markZeroHammingDistance(qc, qr):
    for qi in range(0, Q_D):
        qc.x(qr[Q_T + qi])

    control_qubits = []
    for mi in range(0, M):
        if SR[mi] != '?':
            for ai in range(0, Q_A):
                control_qubits.append(qr[Q_T + mi * Q_A + ai])

    ancilla_qubits = []
    for qi in anc:
        ancilla_qubits.append(qr[qi])

    target = control_qubits.pop()

    qc.h(target)
    qc.mct(control_qubits, target, ancilla_qubits)
    qc.h(target)

    for qi in range(0, Q_D):
        qc.x(qr[Q_T + qi])
    return

# Inversion about mean
def inversionAboutMean(qc, qr):
    for si in range(0, Q_D + Q_T):
        qc.h(qr[si])
        qc.x(qr[si])

    control_qubits = []
    for qi in range(1, Q_D + Q_T):
        control_qubits.append(qr[qi])

    ancilla_qubits = []
    for qi in anc:
        ancilla_qubits.append(qr[qi])

```

```

qc.h(qr[0])
qc.mct(control_qubits, qr[0], ancilla_qubits)
qc.h(qr[0])

for si in range(0, Q_D + Q_T):
    qc.x(qr[si])
    qc.h(qr[si])
return

# Oracle to mark stored patterns
def markStoredPatterns(qc, qr):
    control_qubits = []
    for qi in range(1, Q_D + Q_T):
        control_qubits.append(qr[qi])

    ancilla_qubits = []
    for qi in anc:
        ancilla_qubits.append(qr[qi])

    for qi in range(0, N - M + 1):
        qis = format(qi, '0' + str(Q_T) + 'b')
        for qisi in range(0, Q_T):
            if qis[qisi] == '0':
                qc.x(qr[qisi])
        wMi = RG[qi:qi + M]
        for wisi in range(0, M):
            wisia = format(int(wMi[wisi]), '0' + str(Q_A) + 'b')
            for wisiai in range(0, Q_A):
                if wisia[wisiai] == '0':
                    qc.x(qr[Q_T + wisi * Q_A + wisiai])

    qc.h(qr[0])
    qc.mct(control_qubits, qr[0], ancilla_qubits)
    qc.h(qr[0])

    for wisi in range(0,M):
        wisia = format(int(wMi[wisi]), '0' + str(Q_A) + 'b')
        for wisiai in range(0, Q_A):
            if wisia[wisiai] == '0':
                qc.x(qr[Q_T + wisi * Q_A + wisiai])
    for qisi in range(0, Q_T):
        if qis[qisi] == '0':
            qc.x(qr[qisi])
    return

# Final measurement
def finalGroverMeasurement(qc, qr, cr):

```

```
for qi in range(0, Q_T):
    qc.measure(qr[qi], cr[qi])
return

## Main function

if __name__ == '__main__':
    # Printing some data for testing
    qc = QAM()
    print("Circuit depth: {}".format(qc.depth()))

    # Total number of gates
    print("Number of gates: {}".format(len(qc.data)))
    gate_num = 1
    for item in qc.data:
        qb_list = ""
        for qb in item[1]:
            qb_list = qb_list + str(qb.index) + ", "
        qb_list = qb_list[:len(qb_list)-2]
        print("#{:}: {}, {}".format(gate_num, item[0].name,
            qb_list))
        gate_num = gate_num + 1

    # Drawing circuit
    #qc.draw()

    # Showing histogram
    # BE CAREFUL!
    # Qiskit uses a LSB ordering, meaning the first qubit is all
    # the way to the right!
    # For example, a state of |01> would mean the first qubit is
    # 1 and the second qubit is 0!
    sim = qiskit.execute(qc, backend=backend, shots=8192)
    result = sim.result()
    final = result.get_counts(qc)
    print(final)
    plot_histogram(final)

    # Showing histogram (noisy simulation)
    noisy_sim = qiskit.execute(qc, backend=backend, shots=8192,
        coupling_map=coupling_map,
        noise_model=noise_model,
        basis_gates=basis_gates)
    noisy_result = noisy_sim.result()
    noisy_final = noisy_result.get_counts(qc)
    print(noisy_final)
    plot_histogram(noisy_final)
```

```
# Ideal vs. noisy comparison
plot_histogram([final, noisy_final], color=['red', 'orange'])
```

A.2 Other tests

```
Reference genome: 20310200200133020310330303003010
Chosen pattern for testing: 2?3
Total number of qubits: 19
Number of ancilla qubits: 8
shots = 8192
Grover's algorithm had 1 iteration
Number of gates: 9719
Circuit depth: 7425
```

```
Tag: 00000 - Data: 203
Tag: 00001 - Data: 031
Tag: 00010 - Data: 310
Tag: 00011 - Data: 102
Tag: 00100 - Data: 020
Tag: 00101 - Data: 200
Tag: 00110 - Data: 002
Tag: 00111 - Data: 020
Tag: 01000 - Data: 200
Tag: 01001 - Data: 001
Tag: 01010 - Data: 013
Tag: 01011 - Data: 133
Tag: 01100 - Data: 330
Tag: 01101 - Data: 302
Tag: 01110 - Data: 020
Tag: 01111 - Data: 203
Tag: 10000 - Data: 031
Tag: 10001 - Data: 310
Tag: 10010 - Data: 103
Tag: 10011 - Data: 033
Tag: 10100 - Data: 330
Tag: 10101 - Data: 303
Tag: 10110 - Data: 030
Tag: 10111 - Data: 303
Tag: 11000 - Data: 030
Tag: 11001 - Data: 300
Tag: 11010 - Data: 003
Tag: 11011 - Data: 030
Tag: 11100 - Data: 301
Tag: 11101 - Data: 010
```

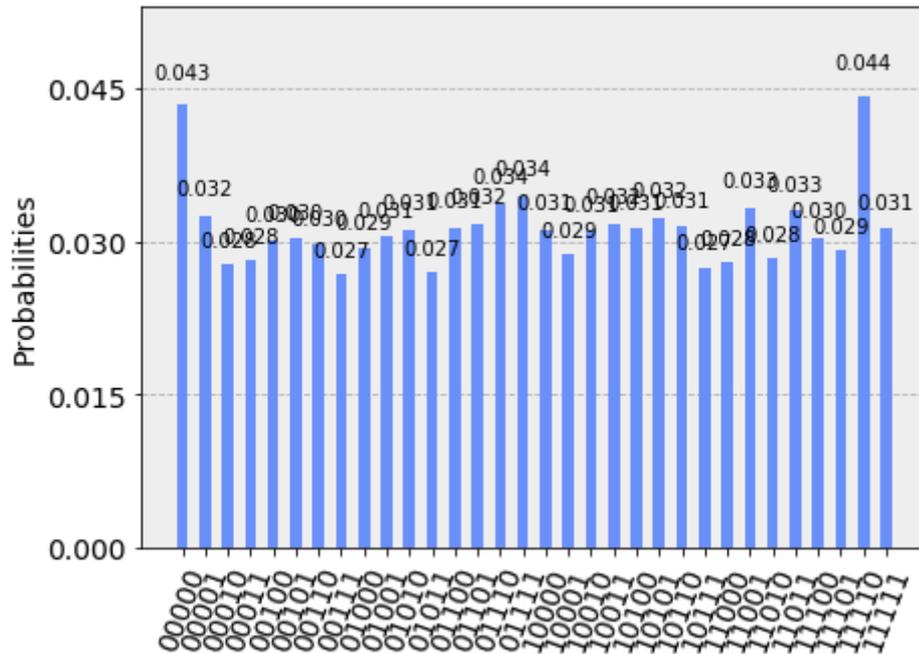


Figure A.1. Obtained probability histogram for HIV genome from 1000 to 1031 and search pattern equal to 2?3 over 8192 shots.

Reference genome: 33210222111130220000022213233220
 Chosen pattern for testing: 2?3
 Total number of qubits: 19
 Number of ancilla qubits: 8
 shots = 8192
 Grover's algorithm had 1 iteration
 Number of gates: 10143
 Circuit depth: 7804

Tag: 00000 - Data: 332
 Tag: 00001 - Data: 321
 Tag: 00010 - Data: 210
 Tag: 00011 - Data: 102
 Tag: 00100 - Data: 022
 Tag: 00101 - Data: 222
 Tag: 00110 - Data: 221
 Tag: 00111 - Data: 211
 Tag: 01000 - Data: 111
 Tag: 01001 - Data: 111
 Tag: 01010 - Data: 113
 Tag: 01011 - Data: 130
 Tag: 01100 - Data: 302
 Tag: 01101 - Data: 022
 Tag: 01110 - Data: 220
 Tag: 01111 - Data: 200
 Tag: 10000 - Data: 000

Tag: 10001 – Data: 000
 Tag: 10010 – Data: 000
 Tag: 10011 – Data: 002
 Tag: 10100 – Data: 022
 Tag: 10101 – Data: 222
 Tag: 10110 – Data: 221
 Tag: 10111 – Data: 213
 Tag: 11000 – Data: 132
 Tag: 11001 – Data: 323
 Tag: 11010 – Data: 233
 Tag: 11011 – Data: 332
 Tag: 11100 – Data: 322
 Tag: 11101 – Data: 220

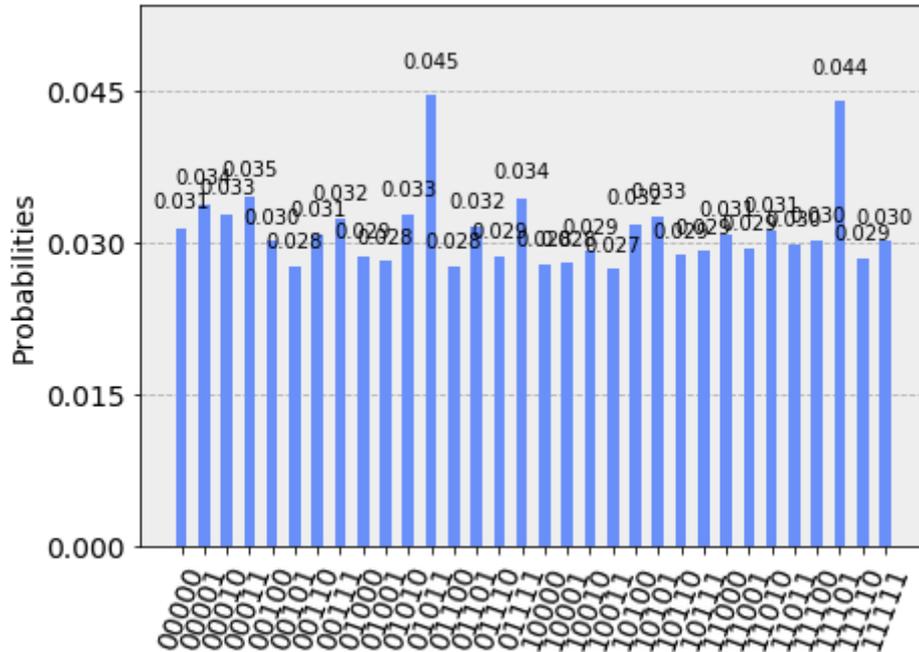


Figure A.2. Obtained probability histogram for HIV genome from 2000 to 2031 and search pattern equal to 2^{23} over 8192 shots.

A.3 Verifying the validity of the implementation

We implemented in Qiskit a QAM version proposed by [46] in which the measuring is done on the qubits encoding the index of a sub-sequence. In order to check if our implementation was valid, we compared the results of our implementation with those obtained from the execution of a compiled QASM code found on <https://gitlab.com/prince-ph0en1x/QaGs> (a repository by A. Sarkar), representing the execution of QAM on a default reference genome equal to 3020310213 with search pattern equal to 2^{23} and 13 Grover’s iterations. Figure A.3 shows the comparison between the obtained histograms: red columns denote the results obtained from our

implementation, while orange columns denote the results obtained from execution of compiled QASM code from <https://gitlab.com/prince-ph0enix/QaGs>. The two histograms practically coincide, highlighting the validity of our implementation.

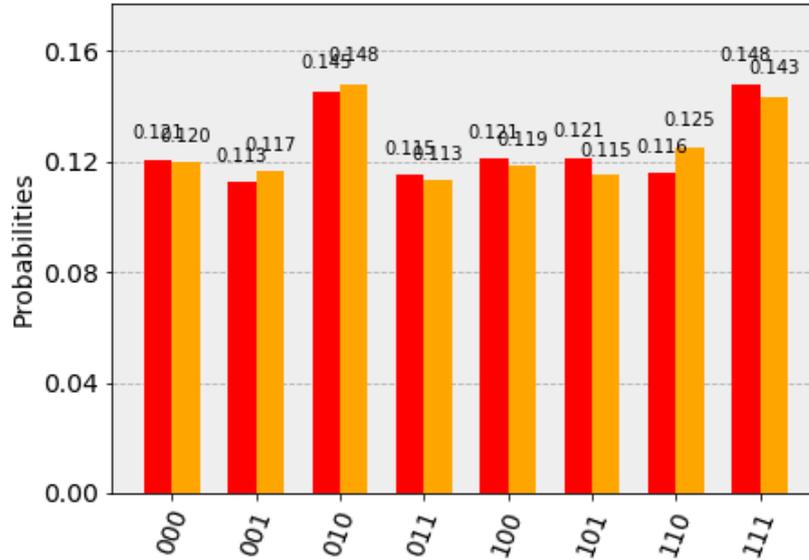


Figure A.3. Comparison of the results obtained from our Qiskit implementation of QAM and from execution of compiled QASM code from <https://gitlab.com/prince-ph0enix/QaGs>.

Appendix B

Quantum indexed Bidirectional Associative Memory

B.1 Python code

```
### Quantum indexed Bidirectional Associative Memory by A.
    Sarkar, Z. Al-Ars, C. G. Almudever, K. Bertels
### Repository reference: https://gitlab.com/prince-ph0en1x/QaGs
    (by A. Sarkar)

## Importing libraries

%matplotlib inline
import qiskit
from qiskit import IBMQ
from qiskit import Aer
from qiskit import QuantumCircuit, ClassicalRegister,
    QuantumRegister, QiskitError
from qiskit.quantum_info.operators import Operator
from qiskit.tools.visualization import circuit_drawer
from qiskit.tools.visualization import plot_histogram
from qiskit.providers.aer import noise
from qiskit.providers.aer.noise import NoiseModel, errors

import random
from math import *
import os
import re
import sys
import math
import matplotlib.pyplot as plt
import numpy as np

## Defining some auxiliary functions
```

```

def hamming_distance(str1, str2):
    count = sum(c1 != c2 for c1, c2 in zip(str1, str2))
    return count

def convertToNumEncoding(genome_str):
    bin_str = ""
    for i in range(0, len(genome_str)):
        if genome_str[i] == 'A':
            bin_str = bin_str + "0"
        elif genome_str[i] == 'C':
            bin_str = bin_str + "1"
        elif genome_str[i] == 'G':
            bin_str = bin_str + "2"
        elif genome_str[i] == 'T':
            bin_str = bin_str + "3"
    return bin_str

def convertReadToBin(read):
    bin_read = ""
    for i in range(0, len(read)):
        if read[i] == '0':
            bin_read = bin_read + "00"
        elif read[i] == '1':
            bin_read = bin_read + "01"
        elif read[i] == '2':
            bin_read = bin_read + "10"
        elif read[i] == '3':
            bin_read = bin_read + "11"
    return bin_read

## Initializing global variables

# Alphabet set {0, 1, 2, 3} := {A, C, G, T} for DNA Nucleotide
bases
AS = {'00', '01', '10', '11'}
A = len(AS)

##### Default Test #####
# Reference Genome: "AATTGTCTAGGCGACC"
#w = "0033231302212011"
#N = len(w)

# Short Read: "CA"
#p = "10"

# Distributed query around zero Hamming distance
#pb = "0000"

```

```
#####

##### Test on HIV genome #####
genome_file = open("HIVgenome.txt", "r")
w = genome_file.read()
genome_file.close()

w = convertToNumEncoding(w)
w = w[0:32]
N = len(w)

# Short read: "GA"
p = "203"

# Distributed query around zero Hamming distance
pb = "000000"
#####

# Short Read size
M = len(p)

# Number of qubits to encode one character
Q_A = ceil(log2(A))

# Number of data qubits
Q_D = Q_A * M

# Tag Qubits
Q_T = ceil(log2(N-M))

# Ancilla Qubits
Q_anc = 8

# Ancilla qubits ids
anc = []

for qi in range(0, Q_anc):
    anc.append(Q_D + Q_T + qi)

total_qubits = Q_D + Q_T + Q_anc

## Inizializing Oracle 0 Matrix

gamma = 0.25
SS = 2**Q_D
bp = np.empty([1, SS])

for i in range(0, SS):
```

```

i_binary = format(i, '0'+str(Q_D)+'b')
hd = hamming_distance(i_binary, pb)
bp[0][i] = sqrt((gamma**hd)*((1 - gamma)**(Q_D - hd)))

B0 = np.identity(SS) - 2*np.dot(np.conjugate(np.transpose(bp)),
    bp)
orc = Operator(B0)

qbsp = []
for qi in range (0, Q_D):
    qbsp.append(Q_T+qi)
qbsp.reverse()

## Initialization of IBM QX

IBMQ.enable_account('INSERT TOKEN HERE')
provider = IBMQ.get_provider()

# Pick an available backend
# If this isn't available pick a backend whose name contains
    '_qasm_simulator' from the output above
backend = provider.get_backend('ibmq_qasm_simulator')

# Uncomment if you want to use local simulator
#backend= Aer.get_backend('qasm_simulator')

## Defining functions

def QIBAM():
    print('Reference genome:', w)
    print('Chosen pattern for testing:', p)
    print('Total number of qubits:', total_qubits)
    print('Number of ancilla qubits:', Q_anc)
    qr = qiskit.QuantumRegister(total_qubits)
    cr = qiskit.ClassicalRegister(Q_T)
    qc = qiskit.QuantumCircuit(qr, cr)

    # Initialise
    generateInitialState(qc, qr)

    # Transform to Hamming distance
    evolveToHammingDistances(qc, qr)

    # Oracle call
    oracle(qc)

    # Inversion about mean
    inversionAboutMean(qc, qr)

```

```
# Memory Oracle
markStoredStates(qc, qr)

# Inversion about mean
inversionAboutMean(qc, qr)

it = 0
for r in range(0, 2):
    # Oracle call
    oracle(qc)

    # Inversion about mean
    inversionAboutMean(qc, qr)

    it = it + 1

print("Grover's algorithm had {} iterations.".format(int(it)))

# Measurement
finalGroverMeasurement(qc, qr, cr)

return qc

# Initialization as suggested by L. C. L. Hollenberg
def generateInitialState(qc, qr):
    for qi in range(0, Q_T):
        qc.h(qr[qi])

    control_qubits = []
    for ci in range(0, Q_T):
        control_qubits.append(qr[ci])

    ancilla_qubits = []
    for qi in anc:
        ancilla_qubits.append(qr[qi])

    for qi in range(0, N - M + 1):
        qis = format(qi, '0' + str(Q_T) + 'b')
        for qisi in range(0, Q_T):
            if qis[qisi] == '0':
                qc.x(qr[qisi])

        wMi = w[qi:qi + M]
        print("Tag: {} - Data: {} - Hamming distance:
              {}".format(qis, wMi,
                          hamming_distance(convertReadToBin(wMi),
                                              convertReadToBin(p))))
```

```

for wisi in range(0, M):
    wisia = format(int(wMi[wisi]), '0' + str(Q_A) + 'b')
    for wisiai in range(0, Q_A):
        if wisia[wisiai] == '1':
            qc.mct(control_qubits, qr[Q_T + wisi * Q_A +
                wisiai], ancilla_qubits)

    for qisi in range(0, Q_T):
        if qis[qisi] == '0':
            qc.x(qr[qisi])
wMi = p

for qi in range(N - M + 1, 2**Q_T):
    qis = format(qi, '0' + str(Q_T) + 'b')
    for qisi in range(0, Q_T):
        if qis[qisi] == '0':
            qc.x(qr[qisi])

    for wisi in range(0, M):
        wisia = format(int(wMi[wisi]), '0' + str(Q_A) + 'b')
        for wisiai in range(0, Q_A):
            if wisia[wisiai] == '0':
                qc.mct(control_qubits, qr[Q_T + wisi * Q_A +
                    wisiai], ancilla_qubits)

    for qisi in range(0, Q_T):
        if qis[qisi] == '0':
            qc.x(qr[qisi])
return

# Calculate Hamming Distance
def evolveToHammingDistances(qc, qr):
    for pi in range(0, M):
        ppi = format(int(p[pi]), '0' + str(Q_A) + 'b')
        for ppai in range(0, Q_A):
            if ppi[ppai] == '1':
                qc.x(qr[Q_T + pi * Q_A + ppai])
    return

# Oracle to mark zero Hamming distance
def oracle(qc):
    qc.unitary(orc, qbsp, label='orc')
    return

# Inversion about mean
def inversionAboutMean(qc, qr):
    for si in range(0, Q_D + Q_T):
        qc.h(qr[si])

```

```

        qc.x(qr[si])

    control_qubits = []
    for sj in range(0, Q_D + Q_T - 1):
        control_qubits.append(qr[sj])

    ancilla_qubits = []
    for qi in anc:
        ancilla_qubits.append(qr[qi])

    qc.h(qr[Q_D + Q_T - 1])
    qc.mct(control_qubits, qr[Q_D + Q_T - 1], ancilla_qubits)
    qc.h(qr[Q_D + Q_T - 1])

    for si in range(0, Q_D + Q_T):
        qc.x(qr[si])
        qc.h(qr[si])
    return

# Oracle to mark Memory States
def markStoredStates(qc, qr):
    control_qubits = []
    for qsi in range(0, Q_T + Q_D - 1):
        control_qubits.append(qr[qsi])

    ancilla_qubits = []
    for qi in anc:
        ancilla_qubits.append(qr[qi])

    for qi in range(0, N - M + 1):
        qis = format(qi, '0' + str(Q_T) + 'b')
        wMi = w[qi:qi + M]
        wt = qis
        for wisi in range(0, M):
            hd = int(format(int(wMi[wisi]), '0'+str(Q_A)+'b'), 2)
                ^ int(format(int(p[wisi]), '0'+str(Q_A)+'b'), 2)
            wisia = format(hd, '0' + str(Q_A) + 'b')
            wt = wt + wisia
        for qisi in range(0, Q_T + Q_D):
            if wt[qisi] == '0':
                qc.x(qr[qisi])

        qc.h(qr[Q_D + Q_T - 1])
        qc.mct(control_qubits, qr[Q_D + Q_T - 1],
            ancilla_qubits)
        qc.h(qr[Q_D + Q_T - 1])

```

```

        if wt[qisi] == '0':
            qc.x(qr[qisi])
    return

# Final measurement
def finalGroverMeasurement(qc, qr, cr):
    for qi in range(0, Q_T):
        qc.measure(qr[qi], cr[qi])
    return

## Main function

if __name__ == '__main__':
    # Printing some data for testing
    qc = QIBAM()
    print("Circuit depth: {}".format(qc.depth()))

    # Total number of gates
    print("Number of gates: {}".format(len(qc.data)))
    gate_num = 1
    for item in qc.data:
        qb_list = ""
        for qb in item[1]:
            qb_list = qb_list + str(qb.index) + ", "
        qb_list = qb_list[:len(qb_list)-2]
        print("#{:}: {}, {}".format(gate_num, item[0].name,
            qb_list))
        gate_num = gate_num + 1

    # Drawing circuit
    #qc.draw()

    # Showing histogram
    # BE CAREFUL!
    # Qiskit uses a LSB ordering, meaning the first qubit is all
    # the way to the right!
    # For example, a state of |01> would mean the first qubit is
    # 1 and the second qubit is 0!
    sim = qiskit.execute(qc, backend=backend, shots=8192)
    result = sim.result()
    final=result.get_counts(qc)
    print(final)
    plot_histogram(final)

```

B.2 Other tests

Reference genome: 20310200200133020310330303003010
Chosen pattern for testing: 203
Total number of qubits: 19
Number of ancilla qubits: 8
shots = 8192
Grover's algorithm had 2 iterations
Number of gates: 54245
Circuit depth: 42886

Tag: 00000 - Data: 203 - Hamming distance: 0
Tag: 00001 - Data: 031 - Hamming distance: 4
Tag: 00010 - Data: 310 - Hamming distance: 4
Tag: 00011 - Data: 102 - Hamming distance: 3
Tag: 00100 - Data: 020 - Hamming distance: 4
Tag: 00101 - Data: 200 - Hamming distance: 2
Tag: 00110 - Data: 002 - Hamming distance: 2
Tag: 00111 - Data: 020 - Hamming distance: 4
Tag: 01000 - Data: 200 - Hamming distance: 2
Tag: 01001 - Data: 001 - Hamming distance: 2
Tag: 01010 - Data: 013 - Hamming distance: 2
Tag: 01011 - Data: 133 - Hamming distance: 4
Tag: 01100 - Data: 330 - Hamming distance: 5
Tag: 01101 - Data: 302 - Hamming distance: 2
Tag: 01110 - Data: 020 - Hamming distance: 4
Tag: 01111 - Data: 203 - Hamming distance: 0
Tag: 10000 - Data: 031 - Hamming distance: 4
Tag: 10001 - Data: 310 - Hamming distance: 4
Tag: 10010 - Data: 103 - Hamming distance: 2
Tag: 10011 - Data: 033 - Hamming distance: 3
Tag: 10100 - Data: 330 - Hamming distance: 5
Tag: 10101 - Data: 303 - Hamming distance: 1
Tag: 10110 - Data: 030 - Hamming distance: 5
Tag: 10111 - Data: 303 - Hamming distance: 1
Tag: 11000 - Data: 030 - Hamming distance: 5
Tag: 11001 - Data: 300 - Hamming distance: 3
Tag: 11010 - Data: 003 - Hamming distance: 1
Tag: 11011 - Data: 030 - Hamming distance: 5
Tag: 11100 - Data: 301 - Hamming distance: 2
Tag: 11101 - Data: 010 - Hamming distance: 4

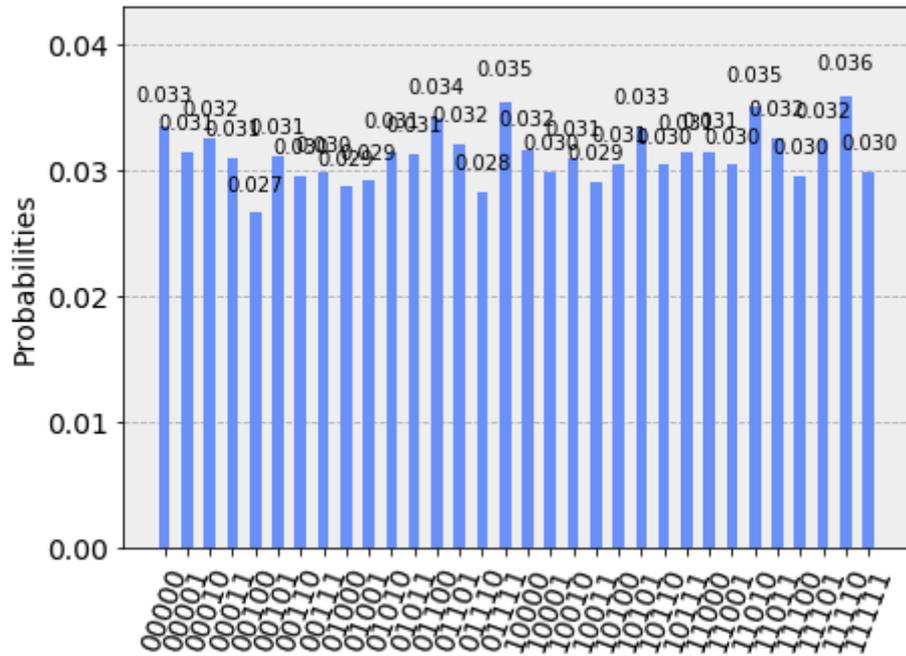


Figure B.1. Obtained probability histogram for HIV genome from 1000 to 1031 and search pattern equal to 203 over 8192 shots.

Reference genome: 33210222111130220000022213233220

Chosen pattern for testing: 203

Total number of qubits: 19

Number of ancilla qubits: 8

shots = 8192

Grover's algorithm had 2 iterations

Number of gates: 54689

Circuit depth: 43265

Tag: 00000 - Data: 332 - Hamming distance: 4
 Tag: 00001 - Data: 321 - Hamming distance: 3
 Tag: 00010 - Data: 210 - Hamming distance: 3
 Tag: 00011 - Data: 102 - Hamming distance: 3
 Tag: 00100 - Data: 022 - Hamming distance: 3
 Tag: 00101 - Data: 222 - Hamming distance: 2
 Tag: 00110 - Data: 221 - Hamming distance: 2
 Tag: 00111 - Data: 211 - Hamming distance: 2
 Tag: 01000 - Data: 111 - Hamming distance: 4
 Tag: 01001 - Data: 111 - Hamming distance: 4
 Tag: 01010 - Data: 113 - Hamming distance: 3
 Tag: 01011 - Data: 130 - Hamming distance: 6
 Tag: 01100 - Data: 302 - Hamming distance: 2
 Tag: 01101 - Data: 022 - Hamming distance: 3
 Tag: 01110 - Data: 220 - Hamming distance: 3
 Tag: 01111 - Data: 200 - Hamming distance: 2
 Tag: 10000 - Data: 000 - Hamming distance: 3

Tag: 10001 – Data: 000 – Hamming distance: 3
 Tag: 10010 – Data: 000 – Hamming distance: 3
 Tag: 10011 – Data: 002 – Hamming distance: 2
 Tag: 10100 – Data: 022 – Hamming distance: 3
 Tag: 10101 – Data: 222 – Hamming distance: 2
 Tag: 10110 – Data: 221 – Hamming distance: 2
 Tag: 10111 – Data: 213 – Hamming distance: 1
 Tag: 11000 – Data: 132 – Hamming distance: 5
 Tag: 11001 – Data: 323 – Hamming distance: 2
 Tag: 11010 – Data: 233 – Hamming distance: 2
 Tag: 11011 – Data: 332 – Hamming distance: 4
 Tag: 11100 – Data: 322 – Hamming distance: 3
 Tag: 11101 – Data: 220 – Hamming distance: 3

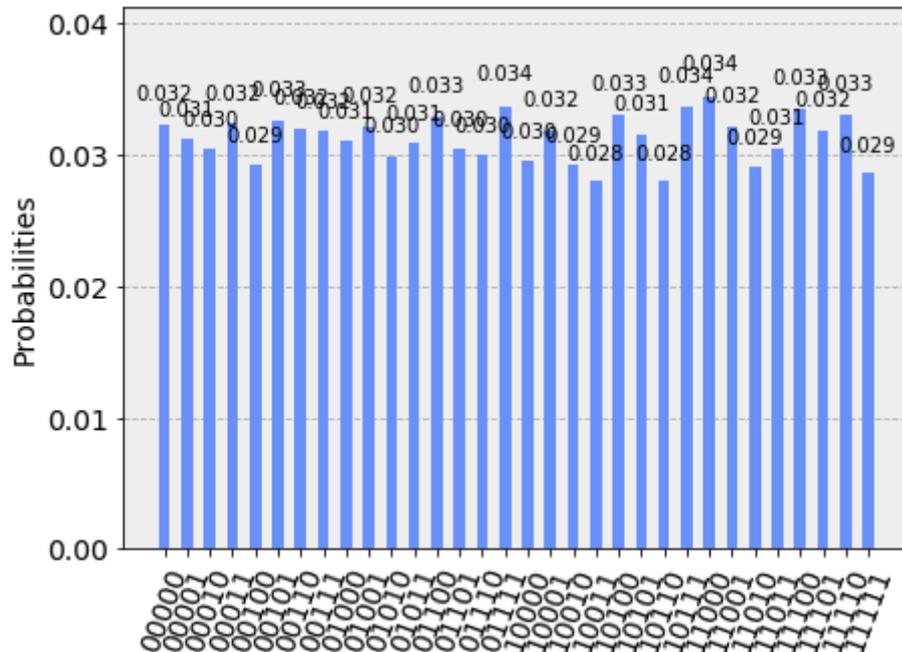


Figure B.2. Obtained probability histogram for HIV genome from 2000 to 2031 and search pattern equal to *203* over 8192 shots.

Please note: Number of gates and Circuit depth need to be recalculated as seen in Paragraph 5.3 (take a look where *Quantum Shannon Decomposition* is discussed).

B.3 Verifying the validity of the implementation

As for the QAM case, we have checked our Qiskit implementation of the QiBAM algorithm by comparing its results with those obtained from the execution of a compiled QASM code found on <https://gitlab.com/prince-ph0enix/QaGs>, representing an execution of QiBAM on a default reference genome *0033231302212011* with search pattern *10* and 3 Grover’s iterations. Figure B.3 shows the comparison

between the obtained histograms: red columns denote the results obtained from our implementation, while orange columns denote the results obtained from execution of compiled QASM code from <https://gitlab.com/prince-ph0enix/QaGs>.

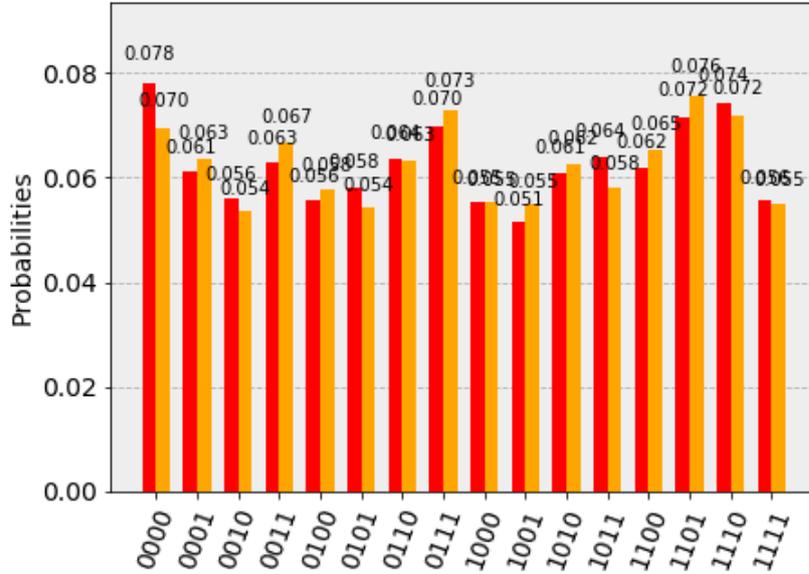


Figure B.3. Comparison of the results obtained from our Qiskit implementation of QiBAM and from execution of compiled QASM code from <https://gitlab.com/prince-ph0enix/QaGs>.

Tag	Input Database	Input Encoding	XOR with CA = 10	Hamming Distance	Estimate Amplification
0	AA	00	10	1	3
1	AT	03	13	3	1
2	TT	33	23	3	1
3	TG	32	22	2	2
4	GT	23	33	4	0
5	TC	31	21	2	2
6	CT	13	03	2	2
7	TA	30	20	1	3
8	AG	02	12	2	2
9	GG	22	32	3	1
10	GC	21	31	3	1
11	CG	12	02	1	3
12	GA	20	30	2	2
13	AC	01	11	2	2
14	CC	11	01	1	3

Figure B.4. Quantum database containing sub-sequences, corresponding indices and Hamming distances with respect to 10 from reference 0033231302212011 [46].

Figure B.4 shows the Hamming distances of the sub-sequences taken from the above mentioned reference with respect to the above mentioned search pattern. Looking at the table, sub-sequences at indices 0, 7, 11 and 14 have the lowest Hamming distances. The histograms in Figure B.3 have their peaks at the right indices, so this is a positive check of the validity of our implementation.

Appendix C

Quantum Pattern Recognition

C.1 Python code

```
### Original articles:
###
### (1) "Improving the Sequence Alignment Method by Quantum
      Multi-Pattern Recognition"
### Konstantinos Prousalis & Nikos Konofaos
### Published in: SETN '18 Proceedings of the 10th Hellenic
      Conference on Artificial Intelligence, Article No. 50
### Patras, Greece, July 09 - 12, 2018
###
### (2) "Quantum Pattern Recognition with Probability of 100%"
### Rigui Zhou & Qiulin Ding
### Published in: International Journal of Theoretical Physics,
      Springer
### Received: 3 August 2007, Accepted: 11 September 2007,
      Published online: 4 October 2007
###
### (3) "Initializing the amplitude distribution of a quantum
      state"
### Dan Ventura & Tony Martinez
### Revised 2nd November 1999

## Importing libraries

%matplotlib inline
import qiskit
from qiskit import IBMQ
from qiskit import Aer
from qiskit import QuantumCircuit, ClassicalRegister,
      QuantumRegister, QiskitError
from qiskit.quantum_info.operators import Operator
```

```
from qiskit.tools.visualization import circuit_drawer
from qiskit.tools.visualization import plot_histogram
from qiskit.tools.visualization import plot_state_city
from qiskit.providers.aer import noise

import random
from math import *
import math
import matplotlib
import numpy as np

## Initializing global variables

# Quantum register is organized like the following:
# |t, x, g, c, a>, with (t+x) having n qubits (index+pattern), g
    having (n-1) qubits and c having 2 qubits
# Also, ancilla qubits (a) as support for mct gate

genome_file = open("HIVgenome.txt", "r")
seq_x = genome_file.read()
genome_file.close()

seq_x = seq_x[0:32]
seq_y = "GAT"

Q_t = ceil(log2(len(seq_x)))
Q_x = len(seq_y)
Q_g = Q_t + Q_x - 1
Q_c = 2
Q_anc = 1
total_qubits = Q_t + Q_x + Q_g + Q_c + Q_anc

## Initialization of IBM QX

IBMQ.enable_account('INSERT TOKEN HERE')
provider = IBMQ.get_provider()

# Pick an available backend
# If this isn't available pick a backend whose name contains
    '_qasm_simulator' from the output above
backend = provider.get_backend('ibmq_qasm_simulator')

# Uncomment if you want to use local simulator
#backend= Aer.get_backend('qasm_simulator')

## Functions for recurrence dot matrix

def delta(x, y):
```

```
    return 0 if x == y else 1

def M(seq1, seq2, i, j, k):
    return sum(delta(x, y) for x, y in zip(seq1[i : i+k], seq2[j :
        j+k]))

def makeMatrix(seq1, seq2, k):
    n = len(seq1)
    m = len(seq2)
    return [[M(seq1, seq2, i, j, k) for j in range(m - k + 1)]
        for i in range(n - k + 1)]

def plotMatrix(M, t, seq1, seq2, nonblank = chr(0x25A0), blank =
    ' '):
    print(' |' + seq2)
    print('-' * (2 + len(seq2)))
    for label, row in zip(seq1, M):
        line = ''.join(nonblank if s < t else blank for s in row)
        print(label + '|' + line)
    return

def convertMatrix(M):
    for i in range(0, len(M)):
        for j in range(0, len(M[i])):
            if M[i][j] == 0:
                M[i][j] = 1
            elif M[i][j] == 1:
                M[i][j] = 0
    return M

def dotplot(seq1, seq2, k = 1, t = 1):
    if len(seq1) > len(seq2):
        raise Exception("Vertical sequence cannot be longer than
            horizontal sequence!")
    M = makeMatrix(seq1, seq2, k)
    plotMatrix(M, t, seq1, seq2)
    M = convertMatrix(M)
    return M

def getAllDiagonalsFromMatrix(M):
    D = np.array([])
    d_size = -1
    for i in range(0, len(M[0])):
        d = np.diag(M, k=i)

        if d_size == -1:
            d_size = len(d)
        D = d
```

```

        elif d_size > len(d):
            z = np.zeros((1, (d_size-len(d))), dtype=int)
            d = np.append(d, z)
            D = np.vstack((D, d))
        else:
            D = np.vstack((D, d))
    return D

def convertBinArrayToStr(array):
    string = ""
    for bin_digit in array:
        if bin_digit == 0:
            string = string + '0'
        elif bin_digit == 1:
            string = string + '1'
    return string

## Functions for Quantum Pattern Recognition

def generateInitialState(qc, qr, dot_matrix):
    D = getAllDiagonalsFromMatrix(dot_matrix)
    m = len(D)
    print("Size of Learning Set: {}".format(len(D)))
    idx = 0
    for d in D:
        print("{}->{:}: {}".format(idx, format(idx,
            '0'+str(Q_t)+'b'), d))
        idx = idx + 1

    z_values = convertBinArrayToStr(np.zeros(Q_t+Q_x))

    ancilla_qubits = []
    for qi in range(0, Q_anc):
        ancilla_qubits.append(qr[Q_t + Q_x + Q_g + Q_c + qi])

    for p in range(m, 0, -1):
        bin_diagonal = convertBinArrayToStr(D[len(D)-p])
        index = format((len(D)-p), '0' + str(Q_t) + 'b')
        instance = index + bin_diagonal
        #print("Instance #{:}, z={}".format(p, instance))
        for j in range(1, Q_t + Q_x + 1):
            if z_values[j-1] != instance[j-1]:
                #print("F_0 #{:} Applied to circuit with ctrl={:}
                    and target={}".format(j, Q_t+Q_x+Q_g+Q_c-1,
                    j-1))
                qc.x(qr[Q_t+Q_x+Q_g+Q_c-1])
                qc.cx(qr[Q_t+Q_x+Q_g+Q_c-1], qr[j-1])
                qc.x(qr[Q_t+Q_x+Q_g+Q_c-1])

```

```
z_values = instance

#print("F_0 Applied to circuit with ctrl={ } and
      target={ }".format(Q_t+Q_x+Q_g+Q_c-1,
      Q_t+Q_x+Q_g+Q_c-2))
qc.x(qr[Q_t+Q_x+Q_g+Q_c-1])
qc.cx(qr[Q_t+Q_x+Q_g+Q_c-1], qr[Q_t+Q_x+Q_g+Q_c-2])
qc.x(qr[Q_t+Q_x+Q_g+Q_c-1])

#print("S_{ },{ } Applied to circuit with ctrl={ } and
      target={ }".format(1, p, Q_t+Q_x+Q_g+Q_c-2,
      Q_t+Q_x+Q_g+Q_c-1))
theta = 2*np.arcsin(1/sqrt(p))
qc.cry(theta, qr[Q_t+Q_x+Q_g+Q_c-2],
      qr[Q_t+Q_x+Q_g+Q_c-1])

if instance[0]=='0' and instance[1]=='0':
    #print("A_00 #1 Applied to circuit with ctrl={ },{ } and
          target={ }".format(0, 1, Q_t+Q_x))
    qc.x(qr[0])
    qc.x(qr[1])
    qc.mct([qr[0], qr[1]], qr[Q_t+Q_x], ancilla_qubits)
    qc.x(qr[1])
    qc.x(qr[0])
elif instance[0]=='0' and instance[1]=='1':
    #print("A_01 #1 Applied to circuit with ctrl={ },{ } and
          target={ }".format(0, 1, Q_t+Q_x))
    qc.x(qr[0])
    qc.mct([qr[0], qr[1]], qr[Q_t+Q_x], ancilla_qubits)
    qc.x(qr[0])
elif instance[0]=='1' and instance[1]=='0':
    #print("A_10 #1 Applied to circuit with ctrl={ },{ } and
          target={ }".format(0, 1, Q_t+Q_x))
    qc.x(qr[1])
    qc.mct([qr[0], qr[1]], qr[Q_t+Q_x], ancilla_qubits)
    qc.x(qr[1])
elif instance[0]=='1' and instance[1]=='1':
    #print("A_11 #1 Applied to circuit with ctrl={ },{ } and
          target={ }".format(0, 1, Q_t+Q_x))
    qc.mct([qr[0], qr[1]], qr[Q_t+Q_x], ancilla_qubits)

for k in range(3, Q_t+Q_x+1):
    if instance[k-1]=='0':
        #print("A_01 #{} Applied to circuit with
              ctrl={ },{ } and target={ }".format(k-1, k-1,
              Q_t+Q_x+k-3, Q_t+Q_x+k-2))
        qc.x(qr[k-1])
```

```

        qc.mct([qr[k-1], qr[Q_t+Q_x+k-3]],
              qr[Q_t+Q_x+k-2], ancilla_qubits)
        qc.x(qr[k-1])
    elif instance[k-1]=='1':
        #print("A_11 #{} Applied to circuit with
              ctrl={},{} and target={}".format(k-1, k-1,
              Q_t+Q_x+k-3, Q_t+Q_x+k-2))
        qc.mct([qr[k-1], qr[Q_t+Q_x+k-3]],
              qr[Q_t+Q_x+k-2], ancilla_qubits)

#print("F_1 Applied to circuit with ctrl={} and
      target={}".format(Q_t+Q_x+Q_g-1, Q_t+Q_x+Q_g))
qc.cx(qr[Q_t+Q_x+Q_g-1], qr[Q_t+Q_x+Q_g])

for k in range(Q_t+Q_x, 2, -1):
    if instance[k-1]=='0':
        #print("A_01 #{} Applied to circuit with
              ctrl={},{} and target={}".format(k-1, k-1,
              Q_t+Q_x+k-3, Q_t+Q_x+k-2))
        qc.x(qr[k-1])
        qc.mct([qr[k-1], qr[Q_t+Q_x+k-3]],
              qr[Q_t+Q_x+k-2], ancilla_qubits)
        qc.x(qr[k-1])
    elif instance[k-1]=='1':
        #print("A_11 #{} Applied to circuit with
              ctrl={},{} and target={}".format(k-1, k-1,
              Q_t+Q_x+k-3, Q_t+Q_x+k-2))
        qc.mct([qr[k-1], qr[Q_t+Q_x+k-3]],
              qr[Q_t+Q_x+k-2], ancilla_qubits)

if instance[0]=='0' and instance[1]=='0':
    #print("A_00 #1 Applied to circuit with ctrl={},{} and
          target={}".format(0, 1, Q_t+Q_x))
    qc.x(qr[0])
    qc.x(qr[1])
    qc.mct([qr[0], qr[1]], qr[Q_t+Q_x], ancilla_qubits)
    qc.x(qr[1])
    qc.x(qr[0])
elif instance[0]=='0' and instance[1]=='1':
    #print("A_01 #1 Applied to circuit with ctrl={},{} and
          target={}".format(0, 1, Q_t+Q_x))
    qc.x(qr[0])
    qc.mct([qr[0], qr[1]], qr[Q_t+Q_x], ancilla_qubits)
    qc.x(qr[0])
elif instance[0]=='1' and instance[1]=='0':
    #print("A_10 #1 Applied to circuit with ctrl={},{} and
          target={}".format(0, 1, Q_t+Q_x))
    qc.x(qr[1])

```

```

        qc.mct([qr[0], qr[1]], qr[Q_t+Q_x], ancilla_qubits)
        qc.x(qr[1])
    elif instance[0]=='1' and instance[1]=='1':
        #print("A_11 #1 Applied to circuit with ctrl={},{} and
            target={}".format(0, 1, Q_t+Q_x))
        qc.mct([qr[0], qr[1]], qr[Q_t+Q_x], ancilla_qubits)
    #print("F Applied to circuit at
        qubit={}".format(Q_t+Q_x+Q_g+Q_c-1))
    qc.x(qr[Q_t+Q_x+Q_g+Q_c-1])
    return

def getIndices(mySet):
    indices = []
    for i in range(0, len(mySet)):
        tmp = ""
        for j in range(0, len(mySet[i])):
            tmp = tmp + str(int(mySet[i][j]))
        indices.append(int(tmp, 2))
    return indices

def oracle(query_set):
    I = np.identity(2**Q_x)
    b_sum = np.zeros((2**Q_x, 2**Q_x))

    indices = getIndices(query_set)
    for i in indices:
        vs = np.zeros((1, 2**Q_x))
        for j in range(0, 2**Q_x):
            if j == i:
                vs[0][j] = 1
        b_sum = b_sum + np.dot(np.conjugate(np.transpose(vs)), vs)

    U = I - (1-1j)*b_sum
    return U

def inversionAboutMean(dot_matrix):
    I = np.identity(2**(Q_t+Q_x))
    b_sum = np.zeros((2**(Q_t+Q_x), 1))

    D = getAllDiagonalsFromMatrix(dot_matrix)
    mySet = np.empty([len(D), Q_t+Q_x])

    for i in range(0, len(D)):
        bin_arr = np.concatenate((convertIntToBinArray(i, Q_t),
            D[i]))
        mySet[i] = bin_arr

    indices = getIndices(mySet)

```

```
for i in indices:
    vs = np.zeros((2**(Q_t+Q_x), 1))
    for j in range(0, 2**(Q_t+Q_x)):
        if j == i:
            vs[j][0] = 1
    b_sum = b_sum + vs

phi_zero = (1/sqrt(len(D))) * b_sum
phi_mtrx = np.dot(phi_zero,
    np.conjugate(np.transpose(phi_zero)))
U = (1 + 1j) * phi_mtrx - 1j * I
return U

def convertIntToBinArray(j, dim):
    if not isinstance(j, int):
        raise Exception("Number of bits must be an integer!")
    elif (j == 0 or j == 1) and dim < 1:
        raise Exception("More bits needed to convert j in
            binary!")
    elif j > 1 and dim <= log2(j):
        raise Exception("More bits needed to convert j in
            binary!")

    bin_arr = np.array([], dtype=int)
    j_bin = format(int(j), '0' + str(dim) + 'b')

    for k in j_bin:
        if k == '1':
            bin_arr = np.append(bin_arr, 1)
        elif k == '0':
            bin_arr = np.append(bin_arr, 0)
    return bin_arr

def QPR(dot_matrix):
    qr = qiskit.QuantumRegister(total_qubits)
    cr = qiskit.ClassicalRegister(Q_t)
    qc = qiskit.QuantumCircuit(qr, cr)

    print("Total number of qubits: {}".format(total_qubits))
    print("Size of t register: {}".format(Q_t))
    print("Size of x register: {}".format(Q_x))
    print("Size of g register: {}".format(Q_g))
    print("Size of c register: {}".format(Q_c))
    print("Number of ancilla qubits: {}".format(Q_anc))

    # A query set is manually defined
    query_set = np.array([[1,1,1],
```

```
        [0,1,1],
        [1,1,0],
        [1,0,1]])

O_mtrx = oracle(query_set)
U_phi_mtrx = inversionAboutMean(dot_matrix)
O = Operator(O_mtrx)
U_phi = Operator(U_phi_mtrx)

O_qubits = []
for qi in range(Q_x-1, -1, -1):
    O_qubits.append(Q_t + qi)

U_phi_qubits = []
for qi in range(Q_t+Q_x-1, -1, -1):
    U_phi_qubits.append(qi)

generateInitialState(qc, qr, dot_matrix)
#simulateStateVector(qc)

T = round((math.pi/4)*sqrt(len(dot_matrix[0])/len(query_set)))

it = 0
for i in range(0, T):
    print("Grover Iteration #{}".format(it+1))
    qc.unitary(O, O_qubits, label='O')
    #simulateStateVector(qc)

    qc.unitary(U_phi, U_phi_qubits, label='U_phi')
    #simulateStateVector(qc)
    it = it + 1

print("Grover's algorithm had {} iterations.".format(int(it)))
finalGroverMeasurement(qc, qr, cr)
return qc

def simulateStateVector(qc):
    result = qiskit.execute(qc,
        backend=Aer.get_backend('statevector_simulator')).result()
    state = result.get_statevector(qc)
    print("Number of states in vector: {}".format(len(state)))
    it = 0
    for item in state:
        bin_str = format(it, '0'+str(total_qubits)+'b')
        bin_str_rev = bin_str[len(bin_str):-1]
        if (item.real != 0 or item.imag != 0):
            print("{}->{}: {}".format(it,
                bin_str_rev[Q_t:Q_t+Q_x], item))
```

```
        it = it + 1
    return

# Final measurement
def finalGroverMeasurement(qc, qr, cr):
    for qi in range(0, Q_t):
        qc.measure(qr[qi], cr[qi])
    return

## Main function

if __name__ == '__main__':
    # Printing some data for testing
    M = dotplot(seq_y, seq_x)
    qc = QPR(M)
    print("Circuit depth: {}".format(qc.depth()))

    # Total number of gates
    print("Number of gates: {}".format(len(qc.data)))
    gate_num = 1
    for item in qc.data:
        qb_list = ""
        for qb in item[1]:
            qb_list = qb_list + str(qb.index) + ", "
        qb_list = qb_list[:len(qb_list)-2]
        print("#{:}: {}, {}".format(gate_num, item[0].name,
            qb_list))
        gate_num = gate_num + 1

    # Drawing circuit
    #qc.draw()

    # Showing histogram
    # BE CAREFUL!
    # Qiskit uses a LSB ordering, meaning the first qubit is all
    # the way to the right!
    # For example, a state of |01> would mean the first qubit is
    # 1 and the second qubit is 0!
    sim = qiskit.execute(qc, backend=backend, shots=1024)
    result = sim.result()
    final=result.get_counts(qc)
    print(final)
    plot_histogram(final)
```

C.2 Other tests

```

|GATCAGAAGAACTTAGATCATTATATAATACA
-----
G|X----X--X-----X-----
A|-X--X-XX-XX---X-X--X--X-X-XX-X-X
T|--X-----XX---X--XX-X-X--X---

```

```

Total number of qubits: 18
Size of t register: 5
Size of x register: 3
Size of g register: 7
Size of c register: 2
Size of a register: 1
Size of Learning Set: 32
shots = 1024
Grover's algorithm had 2 iterations
Number of gates: 1615
Circuit depth: 730

```

```

0->00000: [1 1 1]
1->00001: [0 0 0]
2->00010: [0 0 0]
3->00011: [0 1 0]
4->00100: [0 0 0]
5->00101: [1 1 0]
6->00110: [0 1 0]
7->00111: [0 0 0]
8->01000: [1 1 0]
9->01001: [0 1 0]
10->01010: [0 0 1]
11->01011: [0 0 1]
12->01100: [0 0 0]
13->01101: [0 1 0]
14->01110: [0 0 0]
15->01111: [1 1 1]
16->10000: [0 0 0]
17->10001: [0 0 0]
18->10010: [0 1 1]
19->10011: [0 0 1]
20->10100: [0 0 0]
21->10101: [0 1 1]
22->10110: [0 0 0]
23->10111: [0 1 1]
24->11000: [0 0 0]
25->11001: [0 1 0]
26->11010: [0 1 1]

```

27->11011: [0 0 0]
 28->11100: [0 1 0]
 29->11101: [0 0 0]
 30->11110: [0 1 0]
 31->11111: [0 0 0]

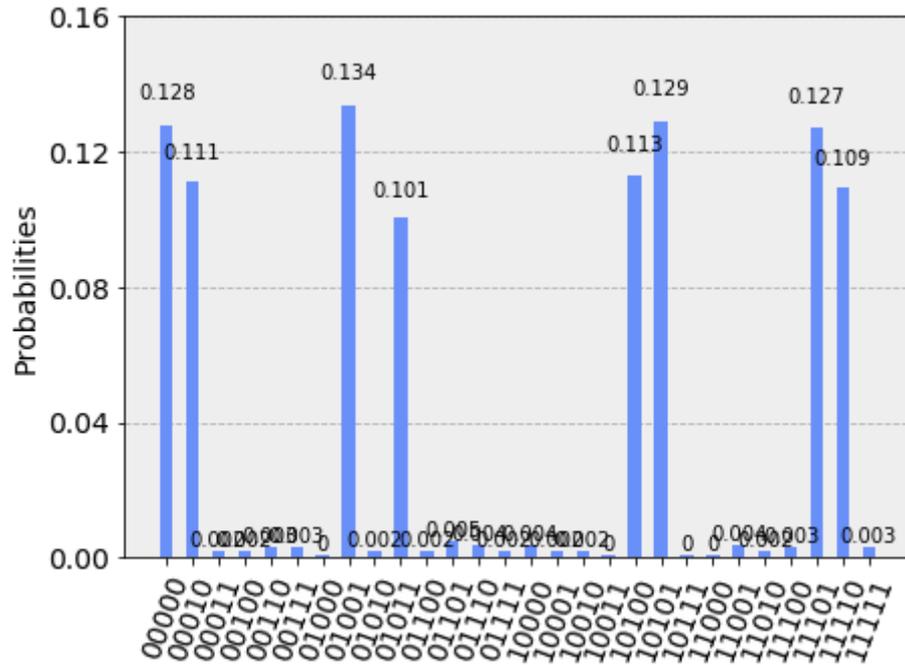


Figure C.1. Obtained probability histogram for recurrence dot matrix between HIV genome from 1000 to 1031 and pattern *GAT*, searching patterns *111*, *110*, *011* and *101* over 1024 shots.

```

|TTGCAGGGCCCCTAGGAAAAAGGGCTGTTGGA
-----
G|--X--XXX-----XX-----XXX--X--XX-
A|----X-----X--XXXXX-----X
T|XX-----X-----X-XX---

```

```

Total number of qubits: 18
Size of t register: 5
Size of x register: 3
Size of g register: 7
Size of c register: 2
Size of a register: 1
Size of Learning Set: 32
shots = 1024
Grover's algorithm had 2 iterations
Number of gates: 1571
Circuit depth: 682

```

```

0->00000: [0 0 0]
1->00001: [0 0 0]
2->00010: [1 0 0]
3->00011: [0 1 0]
4->00100: [0 0 0]
5->00101: [1 0 0]
6->00110: [1 0 0]
7->00111: [1 0 0]
8->01000: [0 0 0]
9->01001: [0 0 0]
10->01010: [0 0 1]
11->01011: [0 0 0]
12->01100: [0 1 0]
13->01101: [0 0 0]
14->01110: [1 0 0]
15->01111: [1 1 0]
16->10000: [0 1 0]
17->10001: [0 1 0]
18->10010: [0 1 0]
19->10011: [0 1 0]
20->10100: [0 0 0]
21->10101: [1 0 0]
22->10110: [1 0 0]
23->10111: [1 0 1]
24->11000: [0 0 0]
25->11001: [0 0 1]
26->11010: [1 0 1]
27->11011: [0 0 0]
28->11100: [0 0 0]
29->11101: [1 0 0]

```

30->11110: [1 1 0]
 31->11111: [0 0 0]

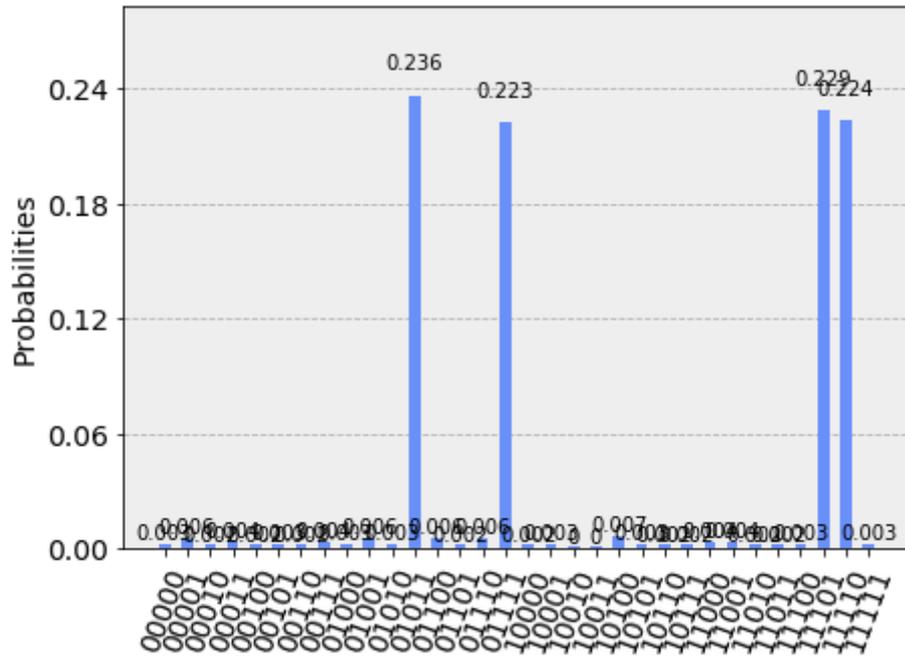


Figure C.2. Obtained probability histogram for recurrence dot matrix between HIV genome from 2000 to 2031 and pattern *GAT*, searching patterns *111*, *110*, *011* and *101* over 1024 shots.

Please note: Number of gates and Circuit depth need to be recalculated as seen in *Paragraph 5.3* (take a look where *Quantum Shannon Decomposition* is discussed).

Bibliography

- [1] Dario Gil, “The Dawn of Quantum Computing is Upon Us”, *IBM Think Blog*, May 2016, <https://www.ibm.com/blogs/think/2016/05/the-quantum-age-of-computing-is-here/>.
- [2] Dr. Christine Corbett Moran, “Mastering quantum computing with IBM QX”, *Packt Publishing*, 2019, ISBN: 978-1-78913-643-2.
- [3] Michael A. Nielsen, Isaac L. Chuang, “Quantum Computation and Quantum Information: 10th Anniversary Edition”, *Cambridge University Press*, 2010, ISBN: 978-1-107-00217-3.
- [4] Jon Porter, “Google confirms “quantum supremacy” breakthrough”, *The Verge*, October 2019, <https://www.theverge.com/2019/10/23/20928294/google-quantum-supremacy-sycamore-computer-qubit-milestone>.
- [5] “Quantum supremacy using a programmable superconducting processor”, *Nature*, Vol. 574, 2019, pp. 505-510, DOI [10.1038/s41586-019-1666-5](https://doi.org/10.1038/s41586-019-1666-5).
- [6] E. Pednault, J. Gunnels, D. Maslov, J. Gambetta, “On “Quantum Supremacy””, *IBM Research Blog*, October 2019, <https://www.ibm.com/blogs/research/2019/10/on-quantum-supremacy/>.
- [7] Quantum annealing, *Wikipedia*, https://en.wikipedia.org/wiki/Quantum_annealing.
- [8] “Introduction to the D-Wave Quantum Hardware”, *D-Wave Tutorials*, <https://www.dwavesys.com/tutorials/background-reading-series/introduction-d-wave-quantum-hardware>.
- [9] Bra-ket notation, *Wikipedia*, https://en.wikipedia.org/wiki/Bra%E2%80%93ket_notation.
- [10] Hilbert space, *Wikipedia*, https://en.wikipedia.org/wiki/Hilbert_space.
- [11] Unitary Matrix, *Wolfram MathWorld*, <http://mathworld.wolfram.com/UnitaryMatrix.html>.
- [12] Barry C. Sanders, “How to Build a Quantum Computer”, *IOP Publishing*, November 2017, ISBN: 978-0-7503-1536-4, DOI [10.1088/978-0-7503-1536-4](https://doi.org/10.1088/978-0-7503-1536-4).
- [13] “Quantum Computing - 73 Companies that are Changing the Computing Landscape”, *Medium*, August 2019, <https://medium.com/datadriveninvestor/quantum-computing-73-companies-that-are-changing-the-computing-landscape-f39ebf0ccfee>.
- [14] “Classical Simulators for Quantum Computers”, *Medium*, February 2019, <https://medium.com/qiskit/classical-simulators-for-quantum-computers-4b994dad4fa2>.
- [15] N. Khammassi, G. G. Guerreschi, I. Ashraf, J. W. Hogaboam, C. G. Almudever, K. Bertels, “cQASM v1.0: Towards a Common Quantum Assembly Language”, <https://arxiv.org/abs/1805.09607v1>.

-
- [16] Cloud-based Quantum Computing, *Wikipedia*, https://en.wikipedia.org/wiki/Cloud-based_quantum_computing.
- [17] Jupyter Notebook, <https://jupyter.org/>.
- [18] Anaconda, <https://www.anaconda.com/distribution/>.
- [19] “DNA: Structure, Function and Discovery”, *BIJU’S*, <https://byjus.com/biology/dna-structure/>.
- [20] Chargaff’s rules, *Wikipedia*, https://en.wikipedia.org/wiki/Chargaff%27s_rules.
- [21] DNA sequencing, *Khan Academy*, <https://www.khanacademy.org/science/high-school-biology/hs-molecular-genetics/hs-biotechnology/a/dna-sequencing>.
- [22] Human Genome Project, *Wikipedia*, https://en.wikipedia.org/wiki/Human_Genome_Project.
- [23] “DNA Sequencing Fact Sheet”, *National Human Genome Research Institute*, <https://www.genome.gov/about-genomics/fact-sheets/DNA-Sequencing-Fact-Sheet>.
- [24] J. M. Heather, B. Chain, “The sequence of sequencers: The history of sequencing DNA”, *Genomics*, Vol. 107, No. 1, January 2016, pp. 1-8, DOI [10.1016/j.ygeno.2015.11.003](https://doi.org/10.1016/j.ygeno.2015.11.003).
- [25] RNA, *Wikipedia*, <https://en.wikipedia.org/wiki/RNA>.
- [26] Sequence assembly, *Wikipedia*, https://en.wikipedia.org/wiki/Sequence_assembly.
- [27] De novo sequence assemblers, *Wikipedia*, https://en.wikipedia.org/wiki/De_novo_sequence_assemblers.
- [28] C. B. Abhilash, K. Rohitaksha, “A Comparative Study on Global and Local Alignment Algorithm Methods”, *International Journal of Emerging Technology and Advanced Engineering*, Vol. 4, No. 1, January 2014, <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.589.9514>.
- [29] “Sequence Analysis, Lecture notes”, *Bielefeld University, Faculty of Technology*, Winter 2015/2016, http://profs.scienze.univr.it/~liptak/FundBA/files/sequence_analysis_ws1516.pdf.
- [30] Ben Langmead, “Strings and Exact Matching”, *Johns Hopkins Whiting School of Engineering, Department of Computer Science*, http://www.cs.jhu.edu/~langmea/resources/lecture_notes/03_strings_exact_matching_v2.pdf.
- [31] Ben Langmead, “Strings, matching, Boyer-Moore”, *Johns Hopkins Whiting School of Engineering, Department of Computer Science*, http://www.cs.jhu.edu/~langmea/resources/lecture_notes/strings_matching_boyer_moore.pdf.
- [32] “Computational Biology, Exact String Matching”, *Stony Brook University*, <https://rob-p.github.io/CSE549F17/lectures/Lec03.pdf>.
- [33] Needleman-Wunsch Algorithm, *Bioinformatics Guide*, <https://bioinfoguide.com/index.php/algorithms-and-methods/10-needleman-wunsch-algorithm>.
- [34] Smith-Waterman Algorithm, *Freiburg RNA Tools, University of Freiburg, Department of Computer Science*, <http://rna.informatik.uni-freiburg.de/Teaching/index.jsp?toolName=Smith-Waterman>.
- [35] Yevgeniy Grigoryev, “How Much Information is Stored in the Human Genome?”, *BiteSizeBio*, <https://bitesizebio.com/8378/how-much->

- [information-is-stored-in-the-human-genome/](#).
- [36] “Grover’s algorithm and its Qiskit implementation”, *IBM Quantum Experience*, <https://quantum-computing.ibm.com/support/guides/quantum-algorithms-with-qiskit?page=5cc0d9fd86b50d00642353ca#>.
- [37] C. Lavor, L. R. U. Manssur, R. Portugal, “Grover’s Algorithm: Quantum Database Search”, <https://arxiv.org/abs/quant-ph/0301079v1>.
- [38] D. Ventura, T. Martinez, “Quantum associative memory with exponential capacity”, *Proceedings of the International Joint Conference on Neural Networks*, May 1998, pp. 509-13, DOI [10.1109/IJCNN.1998.682319](https://doi.org/10.1109/IJCNN.1998.682319).
- [39] D. Ventura, T. Martinez, “A Quantum Associative Memory Based on Grover’s Algorithm”, *Proceedings of the International Conference on Artificial Neural Networks and Genetic Algorithms*, 1999, pp. 22-27, DOI [10.1007/978-3-7091-6384-9_5](https://doi.org/10.1007/978-3-7091-6384-9_5).
- [40] D. Ventura, T. Martinez, “Quantum Associative Memory”, *Information Sciences*, Vol. 124, No. 1-4, May 2000, pp. 273-296, DOI [10.1016/S0020-0255\(99\)00101-2](https://doi.org/10.1016/S0020-0255(99)00101-2).
- [41] D. Ventura, T. Martinez, “Initializing the Amplitude Distribution of a Quantum State”, <https://arxiv.org/abs/quant-ph/9807054v1>.
- [42] E. Biham, O. Biham, D. Biron, M. Grassl and D. A. Lidar, “Grover’s Quantum Search Algorithm for an Arbitrary Initial Amplitude Distribution”, <https://arxiv.org/abs/quant-ph/9807027v2>.
- [43] L. C. L. Hollenberg, “Fast Quantum Search Algorithms in Protein Sequence Comparison: Quantum bioinformatics”, *Physical Review E*, Vol. 62, No. 5, November 2000, DOI [10.1103/PhysRevE.62.7532](https://doi.org/10.1103/PhysRevE.62.7532).
- [44] Hamming distance, *Wikipedia*, https://en.wikipedia.org/wiki/Hamming_distance.
- [45] M. Boyer, G. Brassard, P. Hoyer, A. Tapp, “Tight bounds on quantum searching”, *Fortschritte der Physik - Progress of Physics*, Vol. 46, No. 4-5, June 1998, pp. 493-505, DOI [10.1002/\(SICI\)1521-3978\(199806\)46:4/5<493::AID-PROP493>3.0.CO;2-P](https://doi.org/10.1002/(SICI)1521-3978(199806)46:4/5<493::AID-PROP493>3.0.CO;2-P).
- [46] A. Sarkar, Z. Al-Ars, C. G. Almudever, K. Bertels, “An algorithm for DNA read alignment on quantum accelerators”, <https://arxiv.org/abs/1909.05563v1>.
- [47] A. A. Ezhov, A. V. Nifanova, D. Ventura, “Quantum associative memory with distributed queries”, *Information Sciences*, Vol. 128, No. 3-4, October 2000, pp. 271-293, DOI [10.1016/S0020-0255\(00\)00057-8](https://doi.org/10.1016/S0020-0255(00)00057-8).
- [48] J.-P. Tchapel Njafa, S.G. Nana Engo, Paul Wofo, “Quantum associative memory with improved distributed queries”, *International Journal of Theoretical Physics*, Vol. 52 No. 6, June 2013, pp. 1787-1801, DOI [10.1007/s10773-012-1237-0](https://doi.org/10.1007/s10773-012-1237-0).
- [49] K. Prousalis, N. Konofaos, “Improving the Sequence Alignment Method by Quantum Multi-Pattern Recognition”, *Proceeding SETN ’18 Proceedings of the 10th Hellenic Conference on Artificial Intelligence*, Patras (Greece), July 9-12, 2018, Article No. 50, DOI [10.1145/3200947.3201041](https://doi.org/10.1145/3200947.3201041).
- [50] R. Zhou, Q. Ding, “Quantum Pattern Recognition with Probability of 100%”, *International Journal of Theoretical Physics*, Vol. 47, No. 5, May 2008, pp. 1278-1285, DOI [10.1007/s10773-007-9561-5](https://doi.org/10.1007/s10773-007-9561-5).
- [51] Qiskit, <https://qiskit.org/documentation/index.html>.

- [52] Structure and genome of HIV, *Wikipedia*, https://en.wikipedia.org/wiki/Structure_and_genome_of_HIV.
- [53] “How to calculate circuit depth properly?”, *StackExchange - Quantum Computing*, <https://quantumcomputing.stackexchange.com/questions/5769/how-to-calculate-circuit-depth-properly>.
- [54] V. V. Shende, S. S. Bullock, I. L. Markov, “Synthesis of Quantum Logic Circuits”, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 25, No. 6, June 2006, pp. 1000-1010, DOI [10.1109/T-CAD.2005.855930](https://doi.org/10.1109/T-CAD.2005.855930).
- [55] “What is maximum circuit depth and size IBM Q5 and Q16 could handle?”, *StackExchange - Quantum Computing*, <https://quantumcomputing.stackexchange.com/questions/5747/what-is-maximum-circuit-depth-and-size-ibm-q5-and-q16-could-handle>.
- [56] “Overview of Quantum Gates”, *IBM Quantum Experience*, <https://quantum-computing.ibm.com/support/guides/gate-overview?section=5d00d964853ef8003c6d6820#>
- [57] IBM Q 16 Melbourne V1.x.x, *GitHub*, <https://github.com/Qiskit/ibmq-device-information/tree/master/backends/melbourne/V1>.
- [58] IBM Q 16 Melbourne V1.x.x Version Log, *GitHub*, https://github.com/Qiskit/ibmq-device-information/blob/master/backends/melbourne/V1/version_log.md.
- [59] Mark Jackson, “6 Things Quantum Computers Will Be Incredibly Useful For”, *SingularityHub*, June 2017, <https://singularityhub.com/2017/06/25/6-things-quantum-computers-will-be-incredibly-useful-for/>.
- [60] ART, <https://www.niehs.nih.gov/research/resources/software/biostatistics/art/index.cfm>.
- [61] Bowtie 2, <http://bowtie-bio.sourceforge.net/bowtie2/index.shtml>.