ROMA TRE
UNIVERSITÀ DEGLI STUDI

Department of Mathematics and Physics
Master's Thesis in Computational Sciences
Academic Year: 2022/2023

Solving systems of equations through quantum computing

From Variational Quantum Algorithms to Quantum Annealing

Candidate:
Sara Galatro

Supervisor:
Prof. Marco Pedicini
Co-Supervisor:
Dott. Massimo Bernaschi

MSC AMS: **81P68**, **90C27**, **49R05**
KEYWORDS: **Quantum Computation**, **Combinatorial Optimization**, **Variational Methods**

The Road goes ever on and on
Down from the door where it began.
Now far ahead the Road has gone,
And I must follow, if I can,
Pursuing it with eager feet,
Until it joins some larger way
Where many paths and errands meet.
And whither then? I cannot say

J.R.R. Tolkien, "*The Fellowship of the Ring*"

Abstract

Quantum computing promises to become a powerful tool for solving classically expensive problems, but the current devices have limited capacity and service due to noise and decoherence. Two approaches, however, seem to help bridge this computational gap - Variational Quantum Algorithms and Quantum Annealing.
This thesis aims to test these approaches to solving systems of multivariate equations, which are the fundamental building blocks of every scientific field. Specifically, for gate-based computing, we discuss an ad-hoc-designed variational algorithm to solve linear systems and the QAOA algorithm to solve MQ problems. Furthermore, we test a quantum annealing iterative method to solve generic MQ problems on two different quantum devices to compare their performances.
Our results show that while there are still limitations due to quantum processors' hardware restrictions, these approaches may still be effective in solving multivariate equations with some more preprocessing. These findings encourage further research to expand the applicability of quantum computing beyond its current hardware limitations.

Contents

Introduction **15**

1 Classical Preliminaries **21**
1.1 Mathematics . . . 21
1.1.1 Notation: Vectors and Kets . . . 21
1.1.2 Finite Field . . . 22
1.1.3 Hilbert Space . . . 23
1.1.4 Graph Theory . . . 23
1.2 Computer Science . . . 25
1.2.1 Classical Computational Complexity . . . 25
1.2.2 Optimization Problems . . . 28
1.2.3 The Binary Quadratic Model . . . 29

2 Quantum Preliminaries **33**
2.1 Quantum Mechanics . . . 33
2.1.1 First Postulate . . . 33
2.1.2 Second Postulate . . . 34
2.1.3 Third Postulate . . . 36
2.1.4 Fourth Postulate . . . 40
2.2 Adiabatic Theorem . . . 42
2.3 Quantum Computing . . . 43

3 Gate-based Computing **47**
3.1 Quantum Circuits . . . 47
3.2 Transpiling Process . . . 50
3.3 Quantum Complexity Theory . . . 53

4 Variational Quantum Algorithms **55**
4.1 Theoretical background . . . 55
4.2 Workflow of a VQA . . . 56
4.3 Modules Analysis . . . 58
4.3.1 Reference States . . . 58
4.3.2 Ansatzes and Variational Forms . . . 59
4.3.3 Cost Functions . . . 63
4.3.4 Optimization Loops . . . 65

5 Quantum Annealing **69**
5.1 Physics background and algorithm description . . . 69
5.2 Minor Embedding . . . 72
5.3 Available QPUs' Topologies . . . 76

6 Experiments on the Gate Model: VQLS **77**
6.1 Procedure and Code Analysis . . . 77
6.1.1 Ansatz . . . 78
6.1.2 Cost Function . . . 79
6.1.3 Optimizer . . . 90
6.2 Experimental results . . . 91
6.2.1 Simulation of an ideal quantum computer . . . 91
6.2.2 Running on QPU and noise simulation . . . 96

7 Experiments on the Gate Model: QAOA **99**
7.1 Encoding the MQ problem into a BQM: Direct Encoding . . . 99
7.2 Experimental results . . . 101
7.2.1 Simulation of an ideal quantum computer . . . 101
7.2.2 Running on QPU and noise simulation . . . 104

8 Experiments with Quantum Annealing **107**
8.1 Encoding the MQ problem into a BQM: Truncated and Penalty Encoding . 107
8.1.1 Truncated Encoding . . . 107
8.1.2 Penalty Encoding . . . 109
8.1.3 Resources Comparison . . . 110
8.2 Experimental results . . . 111
8.2.1 First Tests: Solving MQ5 . . . 113
8.2.2 Solving MQ9 through an Iterative Method . . . 117

9 Conclusions and Future Work **123**
9.1 Experiments on the Gate-model . . . 123
9.2 Experiments with Quantum Annealing . . . 124
9.3 Analising Binary Quadratic Models through Complex Networks theory and the PyQubo library . . . 124

Appendix **125**

Bibliography **134**

Sitography **136**

List of Figures

1 Key applications of Variational Quantum Algorithms. 16

1.1 Examples of (a) a complete graph, (b) a complete bipartite graph, and (c) a random-generated weighted graph. 24
1.2 Diagram showing the relations between some classical complexity classes (assuming P ≠ NP). 27
1.3 The Merlin-Arthur protocol. 27

2.1 Bloch sphere representation of a qubit in state $|\psi\rangle$. 34

3.1 Simulation of a Toffoli gate using an ancilla 50
3.2 Steps of the transpilation process. 50
3.3 Transpiled Toffoli gate onto FakeLondonV2 backend 52
3.4 Transpiled Toffoli gate onto FakeRomeV2 backend 52
3.5 Diagram of the quantum computational complexity classes versus the classical ones. 53

4.1 Simplified workflow of a generic variational algorithm 57
4.2 Examples of 2-Local ansatzes 61
4.3 Example of a hardware efficient ansatz 61
4.4 Circuit example of a QAOA ansatz (not decomposed) 62
4.5 Circuit example of a QAOAnsatz (decomposed) 62
4.6 Example of a barren plateau 66

5.1 Visual representation of the quantum tunneling phenomenon 70
5.2 Quantum Annealing Evolution Diagram 71
5.3 Graphical representation of the cost function $\mathcal{H}(a, b) = 13a + 3ab - 7b$ . . . 72
5.4 Embedding of a triangular graph to show qubits chains usage. 73
5.5 Pegasus and Zephyr topologies. 76

6.1 Example of Hardware Efficient Ansatz built for VQSL with 8 qubits and $4 + 1$ layers. 78
6.2 Global cost evolution in VQLS with CG optimizer 93
6.3 Local cost evolution in VQLS with CG optimizer 93
6.4 Global cost evolution in VQLS with COBYLA optimizer 94
6.5 Local cost evolution in VQLS with COBYLA optimizer 94
6.6 Global cost evolution in VQLS with Powell optimizer 95
6.7 Local cost evolution in VQLS with Powell optimizer 95
6.8 QPUs available with a free plan on the IBM Quantum platform. 96
6.9 Global and local cost evolution of the noisy run on `FakeWashingtonV2` . . 98

7.1 QAOA outputs' probability as pie charts (noiseless simulation) 106
7.2 QAOA outputs' probability as pie charts (noisy simulation) 106

8.1 Scaling for different k in the truncated encoding 108
8.2 Illustration of the timing information and their relationships 114
8.3 Quantum Annealing: result analysis for the MQ5 on both Pegasus and Zephyr 116
8.4 Quantum Annealing: result analysis for the MQ9 on Pegasus 121
8.5 Quantum Annealing: result analysis for the MQ9 on Zephyr 122

List of Tables

4.1 Pauli operators to decompose any unitary **U** as a linear combination of tensor products. . . . 65

6.1 Different initialization techniques used to define $\boldsymbol{\theta}^{(0)}$. . . . 91
6.2 Accuracy results of the VQSL run with the CG method. . . . 91
6.3 Accuracy results of the VQSL run with COBYLA. . . . 92
6.4 Accuracy results of the VQSL run with the Powell method. . . . 92
6.5 Calculation to estimate the QPU usage for the VQLS algorithm. . . . 97

7.1 Additional numbers for the QAOA ideal simulation pie chart. . . . 104
7.2 Calculation to estimate the QPU usage for the QAOA algorithm. . . . 104
7.3 Additional numbers for the QAOA noisy simulation pie chart. . . . 105

8.1 Summary of Boolean operations and their penalty function implementation in a quantum annealer . . . 109
8.2 Logical and physical qubits used for the truncated embedding on a Pegasus device. . . . 110
8.3 Logical and physical qubits used for the truncated embedding on a Zephyr device. . . . 110
8.4 Logical and physical qubits used for the penalty embedding on a Pegasus device. . . . 110
8.5 Logical and physical qubits used for the penalty embedding on a Zephyr device. . . . 110
8.6 Timing comparison between the runs on different topologies . . . 114
8.7 Logical qubits q_i embedding comparison between the two topologies . . . 115

9.1 Software information: general Python packages installed. . . . 127
9.2 Software information for the gate model experiments. . . . 127
9.3 Software information for the quantum annealing experiments. . . . 127
9.4 Hardware information. . . . 128
9.5 Quantum hardware information for the gate model experiments. . . . 128
9.6 Quantum hardware information for the quantum annealing model experiments. . . . 128

List of source codes

3.1 Transpiling code example. . . . 51
6.1 Hardware Efficient Ansatz function implemented in the code. . . . 78
6.2 Function implementing the Hadamard test for β_{ij} with a fixed-hardware ansatz. . . . 81
6.3 Function that computes $\langle\psi|\psi\rangle$ trough Hadamard tests. . . . 82
6.4 Auxiliary function implementing the Hadamard test for $\mathbf{U}_b^\dagger\mathbf{A}_i\mathbf{V}$. . . . 84
6.5 Auxiliary function implementing the Hadamard test for $\mathbf{V}^\dagger\mathbf{A}_j^\dagger\mathbf{U}_b$. . . . 84
6.6 Function implementing the Hadamard tests for γ_{ij}. . . . 85
6.7 Function computing $|\langle b|\psi\rangle|^2$. . . . 85
6.8 Global cost function for the VQLS. . . . 86
6.9 Function implementing the Hadamard test for δ_{ij}^k. . . . 88
6.10 Local cost function for the VQLS. . . . 89
6.11 Code cell needed to run on QPU. . . . 96
7.1 Definition of a MQ problem with 5 variables. . . . 101
7.2 QUBO model associated to the MQ5 as Qiskit `QuadraticProgram`. . . . 102
7.3 Ising model associated to the MQ5. . . . 102
7.4 QAOA run using Qiskit. . . . 102
7.5 QAOA results for MQ5 in a noiseless simulation. . . . 103
7.6 QAOA run using a Qiskit fake backend. . . . 104
7.7 QAOA results for MQ5 in a noisy simulation. . . . 105
8.1 Initialization of the `dwave_runners` class. . . . 112
8.2 Definition of the D-Wave sampler, with automated embedding on the chosen topology. . . . 112
8.3 Computation of the chosen chain strength, either following the definition from the original article or using the definition suggested from the Ocean documentation. . . . 113
8.4 Function that runs **once** the sampling process for the given $\mathcal{H}$ on D-Wave devices. . . . 113
8.5 Results for the MQ5 problem solved through quantum annealing. The displayed results are valid for both topologies. . . . 114
8.6 Iterative method to solve general MQ problems. . . . 118
8.7 Results for the MQ9 problem solved through quantum annealing. . . . 120

Introduction

Over the last few decades, quantum computing has received more and more attention from the academic and industrial world, engaging researchers worldwide to test the potential of this new computational paradigm and its related devices. Though it is believed that quantum computers will yield a consistent advantage over their classical counterparts for diverse applications, the current hardware has many restrictions to face, such as quantum noise and fast decoherence time, and their capacity is minimal. Nonetheless, quantum algorithms and processors proliferated, trying to make the best out of the current situation, commonly called the **Noisy Intermediate-Scale Quantum (NISQ) era**.

These limitations are especially evident in the **gate-based** computing, a direct quantum analogue of classical computing, where the algorithm is devised as a sequence of quantum operations to apply to the given quantum system. In this model, **circuits** are used as models of computation, picturing the information moving through a network of wires and gates, representing the operations acting on the given information. Deutsch presented this model in one of his publications [Deu89], and it has since become the standard way to visualize quantum algorithms and their effects on a quantum system. The strength of this model is its universality [DBE95] [Deu85] and straightforward formulation, making it the go-to approach for many algorithm designs.

To mitigate the undesired consequences of using quantum hardware, a new class of hybrid algorithms called **Variational Quantum Algorithms (VQAs)** have been developed, which use a classical optimizer to minimize a cost function by training a parameterized quantum circuit. Although not exempt from difficulties, VQAs are now considered the most suitable strategy to reach a quantum advantage using the gate model[1], making them an exciting topic in diverse fields of application (see Fig. 1).

A completely different approach marks out the foundations of **Adiabatic Quantum Computing (AQC)**, an alternative to the conventional gate model based on the homonymous theorem of quantum mechanics [AL18]. Roughly, it consists of slowly varying the energy of a system through its Hamiltonian so as not to trigger a transition from one energy level to another. Even though AQC is a universal computational model [Far+00][Aha+08], its applicability is limited. First, the requirement of adiabatic evolution implicitly demands an a priori knowledge of the spectral gap, which is dependent on the specific instances of the initial and the final Hamiltonian. This information, though, could potentially provide the problem solution, making it an unsuitable expectation for any algorithm based on this method. Furthermore, AQC requires either complex Hamiltonian engineering for a direct analog implementation or a circuit with an elevated depth for a digitized implementation, limiting its feasibility and service.

As for analog implementations, in the nineties, another quantum computing model

[1] Since it is an active area of research, not everyone agrees with this statement. For another point of view, see [Kun+22][De +23].

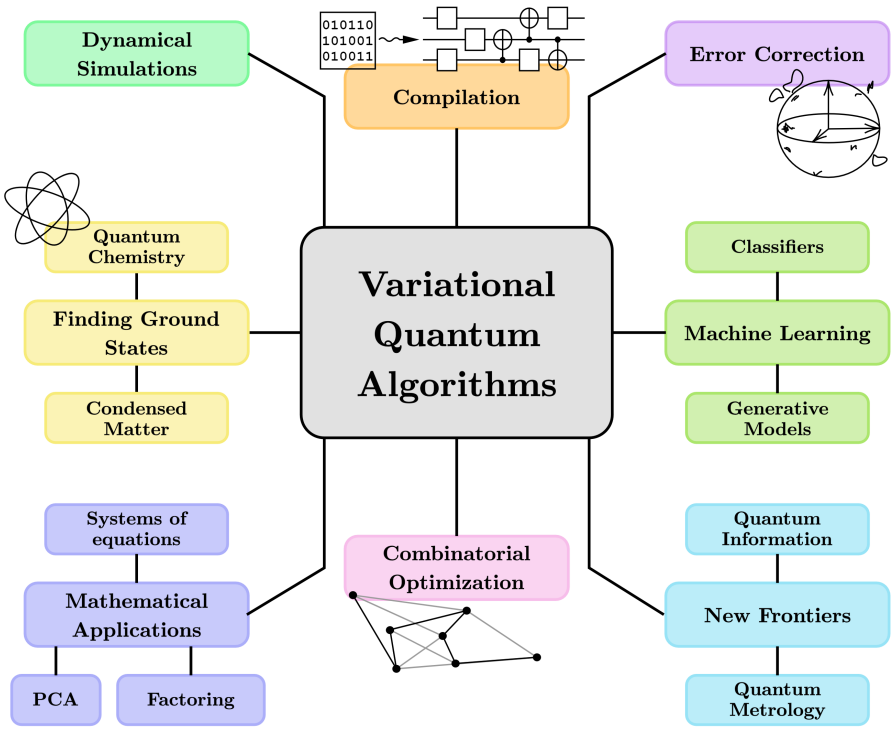


Fig. 1: Key applications of Variational Quantum Algorithms.

known as **Quantum annealing** was introduced [AHS93] [Fin+94] [KN98]. Mainly designed to solve optimization problems, it retraces the operation of the classical simulated annealing, substituting the artificial temperature with quantum fluctuations. Even though its realization relies on the adiabatic theorem too, which guarantees that the observed final state corresponds to the energetic ground state, this model considers physical noise and the possibility of non-adiabatic dynamics, making it a more robust model than AQC. However, due to the probabilistic nature of the run, a statistical sampling of computations is required to gather confidence in the observed result, thus mitigating the effects of a non-trivial error rate. Although more viable than AQC, quantum annealing is not as applicable. Indeed, through a quantum annealing routine, we can only solve problems that can be stated as optimization or sampling problems, taking away the universality of the quantum analog counterpart. Therefore, much interest has been given to reformulating complex problems to see if quantum annealing could help their resolution (see, for instance, [Luc14][DA17][GKD18]).

Given the highly fervent *quantum landscape*, we decided to test a few quantum algorithms and quantum devices on what we consider one of the core elements of mathematics, i.e. **systems of equations** and their resolution. In more detail, we discuss a total of three algorithms to solve different systems of equations, starting with **linear systems** of the form:

$$\mathbf{A}\mathbf{x} = \mathbf{b} \tag{1}$$

with $\mathbf{A}$ a $N \times N$ matrix, i.e., a system with the same number of equations and unknowns.

Linear systems of equations play a fundamental role in many scientific areas, such as machine learning and numerical solutions of PDEs. Significant attention has been devoted to developing quantum algorithms to compute their solutions in the past decade.

Specifically, by solving a problem called **Quantum Linear Systems Problem (QLSP)**, it is possible to find a solution of a given linear system: in QLSP, the goal is to prepare a quantum state $|x\rangle$ that is proportional to a vector $\mathbf{x}$ that satisfies equation (1). For a s-sparse matrix, the classical Conjugate Gradient method [HS+52] finds the solution vector in $\mathcal{O}(Ns\kappa)$. On the other hand, the HHL quantum algorithm proposed in [HHL09] runs in $\mathcal{O}(N \log N\kappa^2)$, granting a speedup over its classical counterpart. However, it will only be viable once fault-tolerant quantum hardware is developed. In the meantime, an interesting way to make use of the available NISQ devices, a variational approach, named **Variational Quantum Linear Solver (VQLS)**, has been proposed [Bra+23].

Another interesting class of problems is the class of systems of Boolean multivariate equations, specifically the ones with equations of degree 2. This last case is usually referred to as **Multivariate Quadratic (MQ) problem** and its resolution is known to be NP-Hard for generic systems [GJ79]. Polynomial systems over the binary field are a core structure in many diverse areas of applications like algebraic cryptoanalysis [Bar09]. Notably, the MQ problem has important applications in post-quantum cryptography since several post-quantum schemes base their security on its difficulty to be solved. Hence, inspired by [Ram+22], we decided to try and solve a few MQ problems with a gate-based algorithm known as **Quantum Approximate Optimization Algorithm (QAOA)** [FGG14].

The input to the MQ problem consists of m quadratic polynomials $p_1(\mathbf{x}), \ldots, p_m(\mathbf{x}) \in \mathbb{F}_q[\mathbf{x} = (x_1, \ldots, x_n)]$ in n variables $x_1, \ldots, x_n$ and coefficients in a finite field $\mathbb{F}_q$. The output of the problem is a vector $\mathbf{s} = (s_1, \ldots, s_n) \in \mathbb{F}_q^n$ for which $p_i(\mathbf{s}) = 0$ for every $1 \leq i \leq m$. We will tackle the Boolean variant of this problem, i.e., where $q = 2$. In this case, polynomials are called Boolean polynomials, and the corresponding unique map $f : \mathbb{F}_2^n \to \mathbb{F}_2$ is called a **Boolean function**. It is common to refer to the Boolean polynomial as the **Algebraic Normal Form (ANF)** of f which we indicate[2] with $f^{\mathbb{F}}$. In our experiments, we also need another representation called the **Numerical Normal Form (NNF)**, which we indicate with $f^{(\mathbb{Z})}$ and define as the following expression of f as a polynomial

$$f(x_1, \ldots, x_n) = \sum_{u \in \mathbb{F}_2^n} \lambda_u \left(\prod_{i=1}^{n} x_i^{u_i} \right) = \sum_{u \in \mathbb{F}_2^n} \lambda_u X^u$$

where $\lambda_u \in \mathbb{Z}$ and $u = (u_1, \ldots, u_n)$. In this representation, f is a Boolean function on $\mathbb{F}_2^n$, taking values in the integer ring $\mathbb{Z}$. Given a Boolean function, its ANF and NNF are unique. A Boolean polynomial can be converted from ANF to NNF, but, in general, this procedure causes an exponential increase in the number of terms, most of which have degree greater than two. However, this transformation is incompatible with current hardware, which can only simulate interactions with degrees of two at most. Hence, we also discuss a few techniques to encode a MQ problem into a Hamiltonian correctly and how to reduce its terms' interactions, reformulating our problem as a **Binary Quadratic Model (BQM)**.

Finally, following the leitmotif of this thesis, we focus on using quantum annealing to solve systems of Boolean multivariate quadratic equations. Precisely, we recreate the iterative method proposed in [Ram+22] and test it on two different annealer topologies to compare the results, considering that one of these topologies was not yet available when the article was published. Furthermore, we also discuss three encodings and their relative usage of quantum resources in a quantum annealer.

[2] The notation used when discussing the MQ problem follows the one defined in [Ram+22].

This thesis is structured as follows:

- **Preliminaries**
  In the first two chapters, we present a few preliminary notions to fix the notation and state the definitions and theorems on which subsequent chapters rely. In more detail, classical notions, such as the Dirac notation, Hilbert spaces, and Binary Quadratic Models, can be found in Chapter 1, while an overview of quantum mechanics and quantum computing is given in Chapter 2.

- **Gate-based Computing**
  In Chapter 3, we introduce the gate model and quantum circuits to discuss a few computational tricks and the so-called transpilation, i.e. the process that maps the designed algorithm to a real quantum processor. Furthermore, we also present a few "metrics" to estimate the feasibility of a circuit. Notions of quantum complexity theory close this chapter.

- **Variational Quantum Algorithms**
  In Chapter 4, we define the workflow of a variational quantum algorithm (VQA) and analyze each submodule, illustrating the options available and the guidelines that should be met when devising a VQA.

- **Quantum Annealing**
  In Chapter 5, we first introduce quantum annealing by describing the underlying physics and the algorithm execution. Furthermore, as for the gate model, we discuss the minor embedding, a heuristic process whose aim is to find an (approximately) optimal mapping of the problem BQM onto the given quantum processor.

- **Experiments on the Gate Model: VQLS**
  In Chapter 6, the procedure of the VQLS is presented with both a global and a local cost function and their relative estimation routines. Experiments ran on local simulators with and without noise and partially on real quantum hardware — the results, presented as plots and tables, concern three different optimizers and four diverse initialization techniques. The software used in this chapter is a Python library called **Qiskit**, which allows the use of **IBM**'s quantum computers through cloud computing.

- **Experiments on the Gate Model: QAOA**
  In Chapter 7, after describing a first direct encoding strategy for a MQ problem, we present the experimental results concerning an instance of MQ with five Boolean variables. As for the VQLS, experiments ran on local simulators with and without noise and partially on real quantum hardware. The software and hardware used in this chapter are, once more, the Python library called **Qiskit** and the **IBM**'s quantum computers through cloud computing.

- **Experiments with Quantum Annealing**
  In Chapter 8, we first discuss two alternative encodings for an MQ problem, their scaling compared to the direct approach (presented in Chapter 7), and the quantum resources they would require to run on a quantum annealer. Furthermore, we solve the very same MQ problem with five variables through quantum annealing, and we compare the performances of the two considered annealers' topologies. Finally, we implement an iterative method to solve an instance of an MQ with nine variables,

with a comparison between topologies once again. All the experiments of this chapter ran on real quantum hardware, and the results include energy histograms and pictures of the quantum processor with the given problem embedded in it. The software used in this chapter is a Python library called **Ocean**, which allows using **D-Wave**'s quantum annealers through cloud computing.

- **Conclusions and Future Work**
  In Chapter 9, we discuss the overall results and the encountered limitations. Furthermore, a few ideas for future projects are introduced.

Throughout Chapter 6, Chapter 7 and Chapter 8, code snippets illustrate the Python implementation of the main functions of every algorithm. The complete code of the three experiments, comprising Jupyter Notebooks and custom-made Python libraries, is available on the **GitHub repository "VQAs_and_QA"** [Gal]. Since the quantum annealing experiments run all on D-Wave quantum devices, an account on their platform https://cloud.dwavesys.com/leap is necessary to run the code.

An Appendix is also present with additional examples and information about the software and the hardware used.

Chapter 1

Classical Preliminaries

This chapter gives a few helpful **classical notions**, ranging from the mathematical definitions of Hilbert spaces and graphs (Section 1.1) to the introduction of optimization problems and the Binary Quadratic Model formulation (Section 1.2).

1.1 Mathematics

1.1.1 Notation: Vectors and Kets

Unless specified otherwise, we assume to be working in a complex vector space $\mathbb{C}^N$ where, more often than not, $N = 2^n$ for some $n > 0$.

When working in a classical environment, we denote column vectors with a bold letter (e.g., $\mathbf{x}$) and row vectors as their conjugate transpose (e.g., $\mathbf{x}^\dagger$). Furthermore, we indicate the components of a vector with x_i, where the subscript is its position in the vector and ranges from 1 to N. The inner product is displayed as $\langle \mathbf{x} | \mathbf{y} \rangle$, while the outer product is $\mathbf{x}\mathbf{y}^\dagger$.

On the other hand, when discussing quantum systems (which, as we will see, are also associated with complex vector spaces), we use the **Dirac notation**, also known as the **bra-ket notation**. In this scenario, a **ket** $|\psi\rangle$ is a column vector, and a **bra** $\langle\psi|$ is a row vector. Hence, the relationship between kets and bras is that of transposition and complex conjugation:

$$|\psi\rangle^\dagger = \begin{bmatrix} \psi_1 \\ \vdots \\ \psi_N \end{bmatrix}^\dagger \equiv \begin{bmatrix} \psi_1^*, \cdots, \psi_N^* \end{bmatrix} = \langle\psi|$$

With this notation, the inner product is again $\langle\psi|\varphi\rangle$ but without bold symbols, while the outer product is $|\psi\rangle\langle\varphi|$.

This notation is particularly convenient when, in Subsection 2.1.4, we introduce composite quantum systems defined through the tensor product. Suppose we have two complex vector spaces, V and W. The tensor product of two kets from these spaces say $|\psi\rangle \in V$ and $|\varphi\rangle \in W$, is a ket in $V \otimes W$ and it can be written using various notations:

$$|\psi\rangle \otimes |\varphi\rangle \qquad |\psi\rangle|\varphi\rangle \qquad |\psi\varphi\rangle$$

We use each of these notations based on what we want to focus on; for instance, we write $|\psi\rangle|\varphi\rangle$ when we want to characterize a space as its two parts, leaving the underlying tensor

product as an implicit operation. Furthermore, $V \otimes W$ is a vector space with dimension

$$\dim(V \otimes W) = \dim(V)\dim(W) \tag{1.1}$$

1.1.2 Finite Field

A **field** is a set F together with two binary operations on F called addition ($+$) and multiplication ($\cdot$). A **binary operation** on F is a mapping $F \times F \to F$, i.e., a correspondence that associates with each ordered pair of elements of F a uniquely determined element of F. These operations must satisfy the following properties:

- *associativity* of addition and multiplication for all $a, b, c \in F$:

$$a + (b + c) = (a + b) + c \qquad \text{and} \qquad a \cdot (b \cdot c) = (a \cdot b) \cdot c$$

- *commutativity* of addition and multiplication for all $a, b, c \in F$:

$$a + b = b + a \qquad \text{and} \qquad a \cdot b = b \cdot a$$

- *additive and multiplicative identity*, meaning that there exists two distinct elements 0 and 1 in F such that

$$a + 0 = a \qquad \text{and} \qquad a \cdot 1 = a$$

  for all $a \in F$;

- *additive inverses*, meaning that for every $a \in F$ there exists an element $-a \in F$ such that

$$a + (-a) = 0$$

- *multiplicative inverses*, meaning that for every $a \neq 0$ in F, there exists an element $a^{-1} \in F$ such that

$$a \cdot a^{-1} = 1$$

- *distributivity* of multiplication over addition for all $a, b, c \in F$:

$$a \cdot (b + c) = (a \cdot b) + (a \cdot c)$$

More succinctly, a field is a commutative ring where $0 \neq 1$ and all nonzero elements are invertible under multiplication.

A fundamental class in computer science is the one of **finite fields**, i.e., fields whose associated set contains a finite number of elements. The number of elements of a finite field is called its **order**. A finite field of order q exists if and only if $q = p^k$, where p is a prime number and k is a positive integer. Since all fields of order $q = p^k$ are isomorphic and a field cannot contain two different finite subfields with the same order, we can identify all finite fields with the same order and denote them $\mathbb{F}_q$. Specifically, in this thesis, we work with the **binary field** $\mathbb{F}_\mathbf{2}$, defined as the set $\{0, 1\}$ together with the addition and multiplication modulo 2. Furthermore, it is common to represent **Boolean variables** as binary variables, namely `True` as 1 and `False` as 0, making the addition and the multiplication correspond to the logical XOR and the logical AND respectively.

1.1.3 Hilbert Space

A **Hilbert space** $\mathbb{H}$ is a real or complex inner product space that is also a complete metric space with respect to the distance function induced by the inner product.

To say that a complex vector space $\mathbb{H}$ is a **complex inner product space** means that there is an inner product $\langle \mathbf{x}|\mathbf{y}\rangle$ associating a complex number to each pair of elements $\mathbf{x}, \mathbf{y} \in \mathbb{H}$ that satisfies the following properties:

- the inner product is conjugate symmetric, i.e.

$$\langle \mathbf{y}|\mathbf{x}\rangle = \langle \mathbf{x}|\mathbf{y}\rangle^*$$

  This implies that $\langle \mathbf{x}|\mathbf{x}\rangle \in \mathbb{R}$.

- the inner product is linear in its first argument and antilinear in its second, i.e., for all $a, b \in \mathbb{C}$

$$\begin{aligned}\langle a\mathbf{x}_1 + b\mathbf{x}_2|\mathbf{y}\rangle &= a\langle \mathbf{x}_1|\mathbf{y}\rangle + b\langle \mathbf{x}_2|\mathbf{y}\rangle \\ \langle \mathbf{x}|a\mathbf{y}_1 + b\mathbf{y}_2\rangle &= a^*\langle \mathbf{x}|\mathbf{y}_1\rangle + b^*\langle \mathbf{x}|\mathbf{y}_2\rangle\end{aligned}$$

- the inner product of an element with itself is positive definite, i.e.

$$\begin{aligned}\langle \mathbf{x}|\mathbf{x}\rangle &> 0 \qquad \text{if } \mathbf{x} \neq 0 \\ \langle \mathbf{x}|\mathbf{x}\rangle &= 0 \qquad \text{if } \mathbf{x} = 0\end{aligned}$$

The **norm** is the real-valued function $\|\cdot\| : \mathbb{H} \to \mathbb{R}$ such that

$$\|\mathbf{x}\| = \sqrt{\langle \mathbf{x}|\mathbf{x}\rangle}$$

while the **distance** d between two points $\mathbf{x}, \mathbf{y} \in \mathbb{H}$ is defined as

$$d(\mathbf{x}|\mathbf{y}) = \|\mathbf{x} - \mathbf{y}\| = \sqrt{\langle \mathbf{x} - \mathbf{y}|\mathbf{x} - \mathbf{y}\rangle}$$

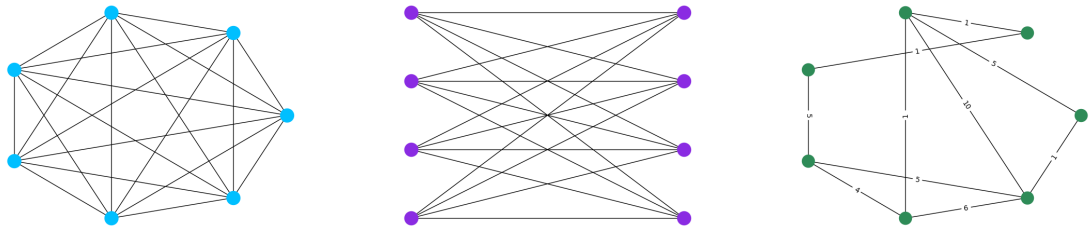
With such a distance, any inner product space is also a **metric space**.

Finally, a metric space $\mathbb{H}$ is **complete** if every Cauchy sequence[1] converges with respect to the associated norm to an element in the space.

1.1.4 Graph Theory

A **graph** G in an ordered pair $(V(G), E(G)) \equiv (V, E)$ consisting of a set V of **vertices** and a set E, disjoint from V, of **edges**, together with an **incidence function** ψ_G that associates with each edge of G an unordered pair of (not necessarily distinct) vertices of G. If e in an edge and u, v are vertices such that $\psi_G(e) = \{u, v\}$, then e is said to **join** u and v, and the vertices u, v are called the **ends** of e. Furthermore, two vertices that share an edge are said to be **adjacent**, as are two edges that are incident with a common vertex. Two distinct adjacent vertices are called **neighbors**. The set of neighbors of a vertex v in a graph G is denoted by $N_G(v)$.

[1] Roughly, given any small positive distance, a Cauchy sequence is a sequence whose elements are all but a finite number less than that given distance from each other.



(a) K_7 **(b)** $K_{4,4}$ **(c)** (G_{rand}, w)

Fig. 1.1: Examples of (a) a complete graph, (b) a complete bipartite graph, and (c) a random-generated weighted graph.

Concerning E, we can distinguish three kinds of edges: an edge with identical ends is called a **loop**, an edge with distinct ends is a **link**, and two or more links with the same pair of ends are said to be **parallel edges**. A graph with no loops or parallel edges is said to be **simple**.

It is common to study only some parts of the graph G, i.e., a subgraph of G. A graph F is called a **subgraph** of a graph G if $V(F) \subseteq V(G)$, $E(F) \subseteq E(G)$, and

$$\psi_F = \psi_G\Big|_{E(F)}$$

i.e., if ψ_F is the restriction of ψ_G to $E(F)$. Furthermore, an **induced subgraph** $G[S]$ is the graph whose vertex set is $S \subseteq V$ and whose edge consists of all the edges with both endpoints in S. That is, for any two vertices $u, v \in S$, u, and v are adjacent in $G[S]$ if and only if they are adjacent in G. The induced subgraph $G[S]$ may also be called the subgraph induced in G by S.

A first distinction between graphs is their connectivity. A graph is **connected** if, for every partition[2] of its vertex set into two nonempty sets V_1, V_2, there is an edge with one end in V_1 and one end in V_2; otherwise, the graph is **disconnected**.

Besides the overall connectivity, it is also possible to study the connectivity level of a single vertex and how these connections are distributed in the graph. The **degree** of a vertex v in a graph G, denoted by $d_G(v)$, is the number of edges incident with v, each loop counting as two edges. In particular, if G is simple, $d_G(v)$ is the number of neighbors of v in G. A vertex of degree zero is called an **isolated vertex**.

Among the many possibilities, certain types of graphs play a prominent role in graph theory - complete graphs and bipartite graphs.
A **complete graph** is a simple graph in which any two vertices are adjacent and is denoted as K_n, where n is the number of vertices (i.e., $n \coloneqq |V|$).
On the other hand, a graph is **bipartite** if its vertex set V can be partitioned in two subsets V_1, V_2 such that every edge has one end in V_1 and the other in V_2. We denote a generic bipartite graph with partition (V_1, V_2) by $G[V_1, V_2]$.
If $G[V_1, V_2]$ is simple and every vertex in V_1 is connected to every vertex in V_2, then G is called a **complete bipartite graph** and is identified as $K_{m,n}$, where $m \coloneqq |V_1|$ and $n \coloneqq |V_2|$.

[2] A family of sets $P = \{P_i\}_i$ is a partition of V if and only if $\varnothing \notin P$, $\bigcup_i P_i = V$, and $P_i \cap P_j = \varnothing$ for all $i \neq j$.

When graphs are used to model other problems, there is often the need to take into account additional factors, such as costs associated with edges. Such situations are modeled by **weighted graphs**, a tuple (G, w) where G is a graph and $w : E \to \mathbb{R}$ is the **weighting** that associates a real-valued weight to each edge $e \in G$.

1.2 Computer Science

1.2.1 Classical Computational Complexity

First, let us define the **Turing machine** (as in [Sip96]), a mathematical model of computation describing an abstract machine that manipulates symbols on a strip of tape according to a table of rules. It represents a general-purpose computer and provides a way to analyze and devise an algorithm without being tied to any particular formalism or physical restriction. We can formally define a Turing machine as a 7-tuple, $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$, where Q, Σ, Γ are all finite sets and

1. Q is the set of states in which the machine can be found;
2. Σ is the input alphabet[3] not containing the **blank symbol** $\sqcup$;
3. Γ is the tape alphabet, where $\sqcup \in \Gamma$ and $\Sigma \subseteq \Gamma$;
4. $\delta : Q \times \Gamma \to Q \times \Gamma \times \{l, r\}$ is the transition function, where l and r stands for "*left*" and "*right*", respectively;
5. $q_0 \in Q$ is the starting state;
6. $q_{accept} \in Q$ is the accept state;
7. $q_{reject} \in Q$ is the reject state, where $q_{reject} \neq q_{accept}$.

Furthermore, we represent with Σ^* the set containing the empty string and all finite-length strings that can be generated by concatenating arbitrary elements of Σ, allowing the use of the same element multiple times. Subsets of Σ^* are called **languages**.

A Turing machine $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$ computes as follows. Initially, M receives its input $w = w_1 w_2 \cdots w_n \in \Sigma^*$ on the leftmost n squares of the tape, and the rest of the tape is blank (i.e., filled with blank symbols). The head starts on the leftmost square of the tape. Note that Σ does not contain the blank symbol, so the first blank appearing on the tape marks the end of the input. Once M has started, the computation proceeds according to the rules described by the transition function. If M ever tries to move its head to the left off the left-hand end of the tape, the head stays in the same place for that move, even though the transition function indicates l. The computation continues until it enters either the accept or reject states, at which point it halts. If neither occurs, M goes on forever.

We also refer to a **non-deterministic Turing machine**, whose definition is similar to the deterministic case. The difference is in how the computation unrolls since the non-deterministic machine may proceed according to several possibilities. The transition function for a non-deterministic Turing machine has thus the following form

$$\delta : Q \times \Gamma \to \mathcal{P}(Q \times \Gamma \times \{l, r\})$$

[3] An alphabet, in the context of formal language theory, is a finite non-empty set of symbols $\{w_i\}_i$. Using these symbols, it is possible to create **words** (or **strings**) by concatenation. We denote the concatenation of the symbols/words $w_1, w_2, \ldots, w_n$ as $w_1 w_2 \cdots w_n$.

The computation of a non-deterministic Turing machine is a tree whose branches correspond to different possibilities for the machine. If some branch of the computation leads to the accept state, the machine accepts its input.

A special class of non-deterministic Turing machines is the **probabilistic Turing machine**, which chooses between the available transitions at each point according to some probability distribution. The formal definition is analogous, the only difference being the transition function. Indeed, the probabilistic Turing machine has two transition functions δ_1, δ_2, which are probabilistically applied to the configuration. This choice is made independently of all prior choices, making the selection process resemble a coin flip at each computation step.

A useful class of computational problems is one of **promise problems**, for which the input is assumed to be drawn from some subset of all possible input strings. In more detail, a promise problem is a pair $A = (A_{\text{yes}}, A_{\text{no}})$, where A_{yes}, $A_{\text{no}} \subseteq \Sigma^*$ are sets of strings satisfying $A_{\text{yes}} \cap A_{\text{no}} = \varnothing$. The strings contained in the sets A_{yes} and A_{no} are called the **yes-instances** and **no-instances** of the problem and have answers *yes* and *no*, respectively. Languages may be viewed as promise problems that obey the additional constraint

$$A_{\text{yes}} \cup A_{\text{no}} = \Sigma^*$$

Now that we have the necessary tools, we define the main classical computational complexity classes following [Wat08], where, for simplicity, promise problems are considered:

P	A promise problem $A = (A_{yes}, A_{no})$ is in P if and only if there exists a polynomial-time deterministic Turing machine M that accepts every string $x \in A_{yes}$ and rejects every string $x \in A_{no}$.
PSPACE	A promise problem $A = (A_{yes}, A_{no})$ is in PSPACE if and only if there exists a polynomial-space deterministic Turing machine M that accepts every string $x \in A_{yes}$ and rejects every string $x \in A_{no}$.
BPP	A promise problem $A = (A_{yes}, A_{no})$ is in BPP if and only if there exists a polynomial-time probabilistic Turing machine M that accepts every string $x \in A_{yes}$ with probability at least $2/3$, and accepts every string $x \in A_{no}$ with probability at most $1/3$.
NP	A promise problem $A = (A_{yes}, A_{no})$ is in NP if and only if there exists a polynomial-bounded function p and a polynomial-time deterministic Turing machine M with the following properties. For every string $x \in A_{yes}$, it holds that M accepts (x, y) for some string $y \in \Sigma^{p(\lvert x \rvert)}$, and for every string $x \in A_{no}$, it holds that M rejects (x, y) for all strings $y \in \Sigma^{p(\lvert x \rvert)}$. Alternatively, a promise problem is in NP if a non-deterministic Turing machine can solve it in polynomial time. It is common to identify $A \in$ NP as requiring A to be verifiable in polynomial time by a deterministic Turing machine.
NP− Complete	A promise problem $A = (A_{yes}, A_{no})$ is in NP − Complete if and only if $A \in$ NP and every other problem in NP is reducible[4] to A in polyomial time.
NP− Hard	A promise problem $A = (A_{yes}, A_{no})$ is in NP − Hard if and only if it can be reduced in polynomial time to every problem L in NP. They do not need solutions that are verifiable in polynomial time.

[4]Roughly, a reduction is an effective mapping between two languages L_1, L_2 which converts the instances in L_1 to instances in L_2.

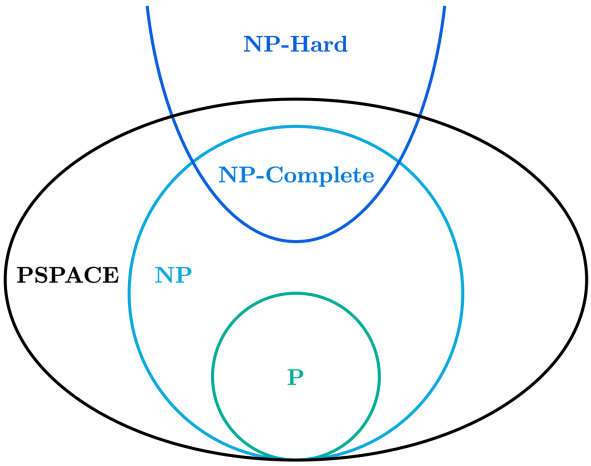


Fig. 1.2: Diagram showing the relations between some classical complexity classes (assuming $\mathsf{P} \neq \mathsf{NP}$).

The final classical complexity class that we need is the MA (Merlin-Arthur) class, based on the homonymous **protocol** in which Merlin, an omniscient wizard with unbounded computation resources, sends a message to King Arthur, who decides to accept it or not by running a probabilistic polynomial-time verification. However, Arthur knows Merlin is prone to prank him; hence, he does not trust him too much.

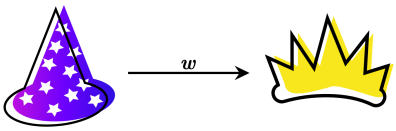


Fig. 1.3: The Merlin-Arthur protocol.

This is an example of an **interactive proof system**, an abstract machine that models computation as the exchange of messages between two parties: a **prover** and a **verifier**. The parties interact by exchanging messages to ascertain whether a given string belongs to a language or not. The prover possesses unlimited computational resources but cannot be trusted, while the verifier has bounded computation power but is assumed to be always honest. Messages are sent between the verifier and the prover until the verifier has an answer to the problem and has "convinced" itself that it is correct.

As for the complexity class, the idea is that a language L is in MA if, for all strings in the language, there is a polynomial-sized proof (sometimes also called **certificate** or **witness**) that Merlin can send Arthur to convince him of this fact with high probability, and for all strings not in the language there is no proof that convinces Arthur with high probability.
Formally, MA is the set of languages $L \subseteq \{0,1\}^*$ for which there exists a probabilistic polynomial-time Turing machine T such that, for all inputs x, if $x \in L$, then there exists a polynomial-sized certificate w so that $T(x,w)$ accepts with probability at least $2/3$; otherwise, if $x \notin L$, then for any polynomial-sized certificate $A(x,w)$ accepts with probability at most $1/3$.

1.2.2 Optimization Problems

Multivariate optimization problems are a common occurrence in many diverse scientific fields, from cosmology and astrophysics [HDR17], all the way through machine learning [Bis06] up to computational systems biology [Wan+06].

In the general case, such a problem is formulated through a **cost (or energy) function** $C(\theta_1, \ldots, \theta_n)$ that must be minimized with respect to the n variables $\boldsymbol{\theta} = \theta_1, \ldots, \theta_n$, which may be subject to some constraints. Within this context, the task is to find a set of values for these variables for which the function $C(\theta_1, \ldots, \theta_n)$ has the minimum value. This set of values is frequently referred to as a **configuration**, and the search space for the optimal one is usually restricted to feasible values. In fact, in many important optimization problems, the search space is a finite set, and the problem is said to be **combinatorial** in nature. Many combinatorial optimization problems can be encoded in a physical system by using its energy levels as a cost function. This way, the Hamiltonian $\mathcal{H}(\boldsymbol{\theta})$ is the new cost function with the minimum in its ground state. Hence, the original problem is reduced to finding the ground state of $\mathcal{H}(\boldsymbol{\theta})$.

Solving optimization problems is typically a complex task. A few deterministic algorithms solve specific optimization problems exactly, but they are pretty small in number and highly problem-specific. For NP or harder problems, only approximate results can be found, and these approximate algorithms are strictly problem-specific too[5].

Since exact algorithms are rare, it is common to use **heuristic algorithms**, i.e., algorithms based on intuitive moves that grant no guarantee on the accuracy or the run time for the worst-case instance. However, these algorithms are generally easy to formulate and based on stochastic iterative improvements. Furthermore, they have been (experimentally) proven to be quite effective in finding a solution for most instances of the intended problem.

One of the main challenges when executing an optimization process is the presence of **local minima**, in which the algorithm may get stuck, returning a false positive configuration and a poor approximation. A straightforward tactic is to repeat the optimization several times with a different initial configuration and choose the best output within all the runs. However, a much better plan would be to design some strategy to get out of the local minima. The idea is to introduce some **fluctuations (or noise)** in the process so that the movement is not always towards lower energy configurations, making it possible to get out of shallow local minima. This is achieved by defining a probability distribution that assigns the highest probabilities to move in the associated configuration to the ones with lower energies and vice versa. Initially, these fluctuations are strong (i.e., the probability of going to a higher energy configuration is relatively high). They are slowly reduced throughout the computation until they are completely turned off. This way, the landscape is explored more exhaustively, and it is more likely that the algorithm will end up in an energy minimum.

One famous classical algorithm exploits such a strategy - **Simulated Annealing**, proposed in [KGV83]. Here, a fluctuation is implemented by introducing an artificial **temperature** T into the problem such that the transition probability from a configuration $\boldsymbol{\theta}^{(i)}$ to a configuration $\boldsymbol{\theta}^{(j)}$ is given by

$$p = \min\{1, e^{-\Delta_{ij}/T}\}$$

where $\Delta_{ij} = E_j - E_i \equiv C(\boldsymbol{\theta}^{(j)}) - C(\boldsymbol{\theta}^{(i)})$, i.e. E_k denotes the cost or energy of the

[5] If a NP-complete problem is resolvable up to a certain degree of approximation through some polynomial algorithm, then that does not ensure that all other NP problems can be solved up to the said approximations in polynomial time using the said algorithm.

configuration C_k. Then, a corresponding Monte Carlo dynamic[6] is defined based on, for instance, a detailed balance, and the thermal relaxation of the system is simulated. During the simulation, the noise factor T is slowly reduced from a high initial value to zero following some annealing schedule. At the end of the simulation, the algorithm is expected to end in a configuration whose energy is a reasonable approximation of the global minimum. If the temperature is decreased as slowly as

$$T(t) \geq \frac{n}{\log t}$$

where t is the cooling time, and n is the system size, the global minimum is attained with certainty in the limit $t \to \infty$ [GG84]. In practice, a reasonably good approximation can be achieved even within a finite time and with a faster cooling rate.

1.2.3 The Binary Quadratic Model

Among all the possible cost functions, we rely on a specific class that encodes quadratic cost functions in a computational-wise structure - the **Binary Quadratic Model (BQM)**. In general, the BQM equation is of the form

$$\mathcal{H}(\boldsymbol{\theta}) = \sum_{i=1}^{n} a_i \theta_i + \sum_{i<j} b_{ij} \theta_i \theta_j + c \tag{1.2}$$

where n is the number of variables and $\theta_i \in \{-1, +1\}$ or $\theta_i \in \{0, 1\}$ for all $i = 1, \ldots, n$. Equation 1.2 represents both the models encoded in the BQM, i.e., the Ising model and the QUBO.

The Ising Model

The **Ising model** is a mathematical model that consists of discrete variables representing magnetic dipole moments of atomic *spins*, that can be either in the state $\uparrow$ or $\downarrow$. The spins are arranged in a graph, usually a lattice, allowing each spin to interact with its neighbors.

Consider a set Λ of lattice sites, each with a set of adjacent sites forming a d-dimensional lattice. For each site $k \in \Lambda$, a discrete variable $\sigma_k \in \{-1, +1\}$ represents the site's spin, where -1 is associated to $\downarrow$ and $+1$ to $\uparrow$. A **spin configuration** $\boldsymbol{\sigma} = \{\sigma_k\}_{k \in \Lambda}$ is the assignment of a spin value to each lattice site. For any two adjacent sites $i, j \in \Lambda$, there is an interaction J_{ij}, while an external magnetic field h_j interacts with each site $j \in \Lambda$. The following (classical) Hamiltonian gives the energy of a configuration $\boldsymbol{\sigma}$:

$$\mathcal{H}(\boldsymbol{\sigma}) = -\sum_{i \sim j} J_{ij} \sigma_i \sigma_j - \sum_{j} h_j \sigma_j \tag{1.3}$$

where the first sum is over pairs of adjacent spins, with each pair counted once. The relation $\sim$ identifies neighboring sites.

The $\mathsf{NP-Complete}$ problem associated with the Ising model is, in general, a decision problem regarding the sign of the ground state energy, i.e., "*does the ground state oh $\mathcal{H}$ have energy ≤ 0?*". Mathematically, because this decision form of the Ising model is $\mathsf{NP-Complete}$, a polynomial-time mapping exists for any other $\mathsf{NP-Complete}$ problem to be reduced to an Ising model.

[6] Monte Carlo methods may vary, but a common structure can be defined. First, we must specify a domain of possible inputs. Then, inputs are randomly generated from a probability distribution over the domain, and we perform a deterministic computation on them. Finally, the results are aggregated and analyzed. Further details in [Wei00].

The QUBO model

Quadratic unconstrained binary optimization (QUBO) is a combinatorial optimization problem defined on the binary vector space $\mathbb{F}_2^n$. Consider a real-valued upper triangular matrix $\mathbf{Q} \in \mathbb{R}^{n\times n}$, whose entries Q_{ij} define a weight for each pair of indices $i, j \in \{1, \dots, n\}$. We can then define a function $f_Q : \mathbb{F}_2^n \to \mathbb{R}$ that assigns a value to each binary vector through

$$f_Q(\mathbf{x}) = \mathbf{x}^T\mathbf{Q}\mathbf{x} = \sum_{i=1}^{n}\sum_{j=i}^{n} Q_{ij}x_ix_j \tag{1.4}$$

Intuitively, the weight Q_{ij} is added if $x_i = x_j = 1$ for all $i, j = 1, \dots, n$. When $i = j$, the values Q_{ii} are added if $x_i = 1$, as $x_ix_i = x_i$ for all $x_i \in \mathbb{F}_2$.
The QUBO problem consists of finding a binary vector $\mathbf{x}^*$ that is minimal with respect to f_Q, i.e.

$$\mathbf{x}^* = \underset{\mathbf{x}\in\mathbb{F}_2^n}{\operatorname{argmin}}\, \mathbf{x}^T\mathbf{Q}\mathbf{x} \tag{1.5}$$

Finding the optimal $\mathbf{x}^*$ in Eq. 1.5 is, in general, $\mathsf{NP}-\mathsf{Hard}$, as the number of candidates $|\mathbb{F}_2^n| = 2^n$ grows exponentially in n.

Relationship between the Ising and the QUBO models

It is possible to prove an equivalence between the Ising and the QUBO models.
For instance, suppose we have an Ising model $\mathcal{H}(\boldsymbol{\sigma})$ which we want to turn into a QUBO model. Applying the identity $\sigma_i \mapsto 2x_i - 1$ for all $i = 1, \dots, n$ yields an equivalent QUBO problem:

$$\begin{aligned}
\mathcal{H}(\mathbf{x}) &= \sum_{i\sim j} -J_{ij}(2x_i-1)(2x_j-1) + \sum_j h_j(2x_j-1) = \\
&= \sum_{i\sim j}(-4J_{ij}x_ix_j + 2J_{ij}x_i + 2J_{ij}x_j - J_{ij}) + \sum_j (2h_jx_j - h_j) \overset{(1)}{=} \\
&= \sum_{i\sim j}(-4J_{ij}x_ix_j) + \sum_{i\sim j} 2J_{ij}x_i + \sum_{i\sim j} 2J_{ij}x_j + \sum_j 2h_jx_j - \sum_{i\sim j} J_{ij} - \sum_j h_j = \\
&= \sum_{i\sim j}(-4J_{ij}x_ix_j) + \sum_{j\sim i} 2J_{ji}x_j + \sum_{i\sim j} 2J_{ij}x_j + \sum_j 2h_jx_j - \sum_{i\sim j} J_{ij} - \sum_j h_j \overset{(2)}{=} \\
&= \sum_{i\sim j}(-4J_{ij}x_ix_j) + \sum_j \sum_{k=j\sim i} 2J_{ki}x_j + \sum_j \sum_{i\sim k=j} 2J_{ik}x_j + \\
&\qquad + \sum_j 2h_jx_j - \sum_{i\sim j} J_{ij} - \sum_j h_j = \\
&= \sum_{i\sim j}(-4J_{ij}x_ix_j) + \sum_j \left(\sum_{i\sim k=j}(2J_{ki}+2J_{ik}) + 2h_j\right)x_j - \sum_{i\sim j} J_{ij} - \sum_j h_j \overset{(3)}{=} \\
&= \sum_{i=1}^{n}\sum_{j=1}^{i} Q_{ij}x_ix_j + c
\end{aligned} \tag{1.6}$$

where equality (1) follows by using $x_j = x_jx_j$, (2) from using $\sum_{i\sim j} = \sum_{j\sim i}$, and (3) from using $\sum_{k=j\sim i} = \sum_{i\sim k=j}$. Furthermore, in Eq. 1.6, we have implicitly defined the following

quantities:

$$Q_{ij} = \begin{cases} -4J_{ij} & \text{if } i \neq j \\ \sum_{i \sim k = j}(2J_{ki} + 2J_{ik}) + 2h_j & \text{if } i = j \end{cases}$$

$$c = -\sum_{i \sim j} J_{ij} - \sum_j h_j$$

where we use once again the fact that for a binary variable $x_j = x_j x_j$ for all $j = 1, \ldots, n$. However, the constant c, also known as **offset**, may be neglected during the optimization since it does not affect the position of the optimum value $\mathbf{x}^*$.

Vice versa, if we want to translate a QUBO model into an Ising model, we may use the following substitution:

$$x_i \mapsto \frac{\sigma_i + 1}{2}$$

Chapter 2

Quantum Preliminaries

Quantum mechanics is the mathematical framework for developing quantum theory and computing. In this chapter, we give a complete description of the basic **postulates of quantum mechanics** as in [NC10] (Section 2.1), followed by an introduction to the **Adiabatic Quantum theorem** (Section 2.2). Finally, the chapter ends with an **overview of quantum computing** (Section 2.3), retracing the milestones of its evolution.

2.1 Quantum Mechanics

2.1.1 First Postulate

The first postulate of quantum mechanics sets up the environment where quantum mechanics takes place: Hilbert spaces.

> **Postulate 1.**
> Associated to any isolated quantum system is a complex Hilbert space known as the **state space** of the system. The system is completely described by its **state vector**, a unit vector in its state space.

The simplest quantum mechanical system, and the one we will be most concerned with, is the **qubit**, the quantum information unit associated with a two-dimensional state space. Given that a Hilbert space is a vector space, numerous orthonormal bases can be defined to represent the state space. The most used one is referred to as the **standard** (or **computational**) **basis** $\mathcal{B} = \{|0\rangle, |1\rangle\}$, where $|0\rangle, |1\rangle$ are the quantum-equivalent of the classical states $0, 1$:

$$|0\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \qquad |1\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \tag{2.1}$$

When clear from the context, we refer to a state vector simply as a **state**.

Starting from Eq. 2.1, an arbitrary state vector in the state space can be written as

$$|\psi\rangle = \alpha_0 |0\rangle + \alpha_1 |1\rangle \tag{2.2}$$

where α_0 and α_1 are complex numbers. Since $|\psi\rangle$ must be a unit vector (i.e. $\langle\psi|\psi\rangle = 1$), we can infer the following **normalization condition** for state vectors:

$$|\alpha_0|^2 + |\alpha_1|^2 = 1 \tag{2.3}$$

Furthermore, we say that any linear combination of state vectors $\sum_i \alpha_i |\psi_i\rangle$ is a **superposition**, in which the basis state vectors $|\psi_i\rangle$ are weighted through their **amplitudes** α_i. For example, the state

$$\frac{|0\rangle - |1\rangle}{\sqrt{2}} \tag{2.4}$$

is a superposition of the states $|0\rangle$ and $|1\rangle$ with amplitude $1/\sqrt{2}$ and $-1/\sqrt{2}$, respectively.

It is sometimes useful to picture the qubits somehow. Because of Eq. 2.3, we can rewrite Eq. 2.2 as

$$|\psi\rangle = e^{i\gamma}\left(\cos(\theta/2)\,|0\rangle + e^{i\varphi}\sin(\theta/2)\,|1\rangle\right)$$

where θ, φ and γ are real numbers. In Subsection 2.1.3, we will prove that the factor $e^{i\gamma}$ can be ignored since it has no observable effects. Thus, we can write

$$|\psi\rangle = \cos(\theta/2)\,|0\rangle + e^{i\varphi}\sin(\theta/2)\,|1\rangle \tag{2.5}$$

The numbers θ and φ define a point on the unit three-dimensional sphere, as shown in Fig 2.1. This sphere is known as the **Bloch sphere**, and it provides a useful means of visualizing the state of a single qubit. However, this intuition is limited since no simple generalization of the Bloch sphere is known for multiple qubits.

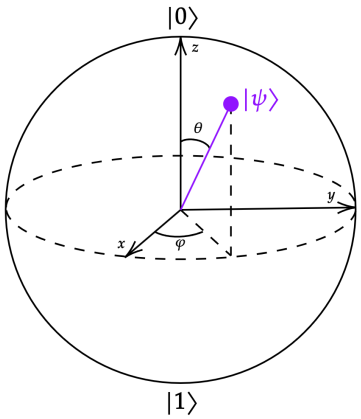


Fig. 2.1: Bloch sphere representation of a qubit in state $|\psi\rangle$.

2.1.2 Second Postulate

Having prepared the scenario with Postulate 1, we now turn to describe the time evolution of such systems.

Postulate 2.
The evolution of a closed quantum system is described by a **unitary transformation**[1]. That is, the state $|\psi\rangle$ of the system at time t_1 is related to the state $|\psi'\rangle$ of the system at time t_2 by a unitary operator $\mathbf{U}$ which depends only on the times t_1, t_2:

$$|\psi'\rangle = \mathbf{U}\,|\psi\rangle \tag{2.6}$$

Let's look at a few examples of unitary operators on a single qubit which are important in quantum computation and quantum information:

[1] A matrix $\mathbf{U}$ is a unitary transformation if its matrix inverse equals its conjugate transpose $\mathbf{U}^{-1} = \mathbf{U}^{\dagger}$.

- **Pauli operators** are a set of three 2×2 complex matrices that are Hermitian, involutory and unitary. They are denoted either as $\sigma^x, \sigma^y, \sigma^z$ or as $\mathbf{X}, \mathbf{Y}, \mathbf{Z}$, and defined as

$$\sigma^x = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \quad \sigma^y = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix} \quad \sigma^z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \tag{2.7}$$

  In quantum computation, the unitaries σ^x and σ^z are usually referred to as **bit flip**[2] and **phase flip**, respectively. This is because of their effect on qubits: the σ^x matrix changes $|0\rangle$ into $|1\rangle$, and $|1\rangle$ into $|0\rangle$, thus earning the name bit flip; while the σ^z matrix leaves $|0\rangle$ invariant, and takes $|1\rangle$ to -$|1\rangle$, with the extra factor of -1 added known as a **phase factor**, thus justifying the term phase flip. Furthermore, together with the identity matrix[3] $\mathbb{1}$ (sometimes considered as the zeroth Pauli matrix σ^0), the Pauli matrices form a basis for the real vector space of 2×2 Hermitian matrices.

- **Phase operators** are 2×2 matrices that implement a qubit rotation about the Z axis. The general phase matrix is represented as

$$\mathbf{P}(\lambda) = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\lambda} \end{bmatrix} \tag{2.8}$$

  with $\lambda \in \mathbb{R}$. Notorious phase operators are $\mathbf{Z} \equiv \mathbf{P}(\pi)$, $\mathbf{S} \coloneqq \mathbf{P}(\pi/2)$ and $\mathbf{T} \coloneqq \mathbf{P}(\pi/4)$.

- Another interesting unitary operator is the **Hadamard matrix**, defined as

$$\mathbf{H} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \tag{2.9}$$

  It maps the basis vector states to their **uniform superpositions**, i.e. superpositions such that $|\alpha_0|^2 = |\alpha_1|^2$. In more detail, for the computational basis, these uniform superpositions are called $|+\rangle$ and $|-\rangle$, and we have

$$|0\rangle \xrightarrow{\mathbf{H}} \frac{|0\rangle + |1\rangle}{\sqrt{2}} \qquad |1\rangle \xrightarrow{\mathbf{H}} \frac{|0\rangle - |1\rangle}{\sqrt{2}}$$

Postulate 2 requires that the system being described be closed; that is, it is not interacting in any way with other systems. In reality, all systems (except the Universe as a whole) interact at least somewhat with other systems. Nevertheless, interesting systems can be described as being closed and by unitary evolution to some good approximation.

It is also worth noticing that a more refined version of Postulate 2 can be given, describing the evolution of a quantum system in continuous time.

Postulate 2bis.
The time evolution of the state of a closed quantum system is described by the Schrödinger equation:

$$i\hbar \frac{d\,|\psi\rangle}{dt} \equiv i\hbar\,|\dot{\psi}\rangle = \mathcal{H}\,|\psi\rangle \tag{2.10}$$

where $\mathcal{H}$ is the fixed Hermitian operator[4] known as **Hamiltonian** of the system, representing the energy of the system, and $\hbar$ is **Planck's constant**, whose value is commonly incorporated in $\mathcal{H}$, effectively setting $\hbar = 1$.

[2] The bit-flip operation is frequently referred to as **NOT** since it resembles the logical NOT gate.
[3] When clear from the context, the identity matrix will have no subscript to identify its dimension.
[4] A **Hermitian matrix H** is a complex square matrix that is equal to its own conjugate transpose: $\mathbf{H} = \mathbf{H}^\dagger$.

Writing the solution to Schrödinger's equation (Eq. 2.10), the connection between Postulate 2 and Postulate 3 is evident:

$$|\psi(t_2)\rangle = \underbrace{\exp\left(\frac{-i\mathcal{H}(t_2 - t_1)}{\hbar}\right)}_{:=\mathbf{U}(t_1,t_2)} |\psi(t_1)\rangle$$

where $\mathbf{U}(t_1, t_2)$ is a unitary operator, just as we would expect from Postulate 2.

If we know the Hamiltonian of a system, then we understand its dynamics completely, at least in principle. In a more practical arrangement, figuring out the Hamiltonian needed to describe a particular physical system is a complicated problem requiring substantial input from experiments to be answered.

Because the Hamiltonian is a Hermitian operator, it has a spectral decomposition

$$\mathcal{H} = \sum_E E \, |E\rangle \langle E| \tag{2.11}$$

with eigenvalues E and corresponding normalized eigenvectors $|E\rangle$. The states $|E\rangle$ are referred to as **energy eigenstates**, while E is the **energy** of the state $|E\rangle$ (also known as **eigenenergy**). The lowest energy is known as the **ground state energy** for the system, and the corresponding energy eigenstate as **ground state**.

In quantum computation and quantum information, we often speak of applying a unitary operator to a particular quantum system, slightly contradicting what we said earlier about unitary operators describing the evolution of a closed quantum system. However, it is generally possible to incorporate a few external parameters, such as applications of unitary operators, in Eq. 2.10 by using a **time-varying Hamiltonian**. The system is not, therefore, closed, but it does evolve according to Schrödinger's equation with a time-varying Hamiltonian to some good approximation.

2.1.3 Third Postulate

Just as in any experiment, there must be a time when we interact with the system, making the system no longer closed and thus not necessarily subject to unitary time evolution. This is of extreme importance when we *measure* the quantum system, action whose effects are described in Postulate 3.

Postulate 3.
Quantum measurements are described by a collection $\{\mathbf{M}_m\}$ of **measurement operators**, which act on the state space of the system being measured. The index m refers to the measurement outcomes that may occur in the experiment. If the state of the quantum system is $|\psi\rangle$ immediately before the measurement, then the probability that result m occurs is

$$p_m = \langle\psi| \, \mathbf{M}_m^\dagger \mathbf{M}_m \, |\psi\rangle \tag{2.12}$$

and the (classical) state of the system after the measurement is

$$\frac{\mathbf{M}_m \, |\psi\rangle}{\sqrt{\langle\psi| \, \mathbf{M}_m^\dagger \mathbf{M}_m \, |\psi\rangle}} \tag{2.13}$$

The measurement operators satisfy the **completeness equation**, that is

$$\sum_m \mathbf{M}_m^\dagger \mathbf{M}_m = \mathbb{1} \tag{2.14}$$

which expresses the fact that probabilities sum to one:

$$1 = \sum_m p_m = \sum_m \langle\psi| \mathbf{M}_m^\dagger \mathbf{M}_m |\psi\rangle \tag{2.15}$$

Equation 2.15 being satisfied for all $|\psi\rangle$ is equivalent to the completeness equation (Eq. 2.14).

Measuring a quantum system has drastic consequences: performing a measurement causes the system to **collapse** on the measured classical state, turning into a classical system and losing all the system information that the quantum state held. Hence, measuring is neither a repeatable nor a reversible operation and must be carried out with the utmost care.

A simple but important example of measurements is the *measurement of a qubit in the standard basis*. This is a measurement on a single qubit with two outcomes defined by the two measurement operators $\mathbf{M}_0 = |0\rangle\langle 0|$ and $\mathbf{M}_1 = |1\rangle\langle 1|$. First, observe that each measurement operator is Hermitian and that $\mathbf{M}_0^2 = \mathbf{M}_0$, $\mathbf{M}_1^2 = \mathbf{M}_1$. Thus, the completeness relation of Eq. 2.14 is satisfied:

$$\mathbb{1} = \mathbf{M}_0^\dagger \mathbf{M}_0 + \mathbf{M}_1^\dagger \mathbf{M}_1 = \mathbf{M}_0 + \mathbf{M}_1$$

Suppose the state being measured is $|\psi\rangle = \alpha_0 |0\rangle + \alpha_1 |1\rangle$. Then, the probabilities of each outcome are

$$\begin{aligned} p_0 &= \langle\psi| \mathbf{M}_0^\dagger \mathbf{M}_0 |\psi\rangle = \langle\psi| \mathbf{M}_0 |\psi\rangle = |\alpha_0|^2 \\ p_1 &= \langle\psi| \mathbf{M}_1^\dagger \mathbf{M}_1 |\psi\rangle = \langle\psi| \mathbf{M}_1 |\psi\rangle = |\alpha_1|^2 \end{aligned}$$

which are perfectly consistent with the definition of a probability density function in light of Eq. 2.3, too. The state after measurement in the two cases is therefore

$$\frac{\mathbf{M}_0 |\psi\rangle}{|\alpha_0|} = \frac{\alpha_0}{|\alpha_0|} |0\rangle \equiv |0\rangle \qquad \frac{\mathbf{M}_1 |\psi\rangle}{|\alpha_1|} = \frac{\alpha_1}{|\alpha_1|} |1\rangle \equiv |1\rangle \tag{2.16}$$

Multipliers like $\alpha_0/|\alpha_0|$ in Eq. 2.16 are called **global phases** (or **global phase factors**), and they can be effectively ignored, so the two post-measurement states are effectively $|0\rangle$ and $|1\rangle$. In more detail, suppose to measure two states that differ for a global phase, say $|\psi\rangle$ and $e^{i\theta}|\psi\rangle$ with $\theta \in \mathbb{R}$, and that $\mathbf{M}_m$ is a measurement operator associated to some quantum measurement. The respective probabilities for outcome m occurring are $\langle\psi| \mathbf{M}_m^\dagger \mathbf{M}_m |\psi\rangle$ and $\langle\psi| e^{-i\theta} \mathbf{M}_m^\dagger \mathbf{M}_m e^{i\theta} |\psi\rangle = \langle\psi| \mathbf{M}_m^\dagger \mathbf{M}_m |\psi\rangle$. Furthermore, since the system's evolution is unitary and linear, the presence of a global phase factor does not influence the probabilities of the outcome. Therefore, from an observational point of view, these two states are identical, and we may ignore global phase factors.

Another kind of phase - the **relative phase** - has quite a different meaning. Consider the states

$$\frac{|0\rangle + |1\rangle}{\sqrt{2}} \qquad \text{and} \qquad \frac{|0\rangle - |1\rangle}{\sqrt{2}}$$

In the first state the amplitude of $|1\rangle$ is $1/\sqrt{2}$, while for the second state the amplitude is $-1/\sqrt{2}$. Even though the **magnitude** (i.e. $|\alpha_1|^2$) is the same, they differ in sign. More generally, we say that two amplitudes, α_i and α_j, differ by a relative phase if there is a real λ such that $\alpha_i = e^{i\lambda}\alpha_j$. Hence, relative phases may vary from amplitude to amplitude, making states which differ only by relative phases give rise to physically observable

differences in measurement statistics. Therefore, it is not possible to regard these states as physically equivalent, as we do with states differing by a global phase factor.

An important application of Postulate 3 is to the problem of distinguishing quantum states. To present the concept of **distinguishability** in quantum mechanics, let us make use of an abstract game involving two parties, Alice and Bob. Alice chooses a state $|\psi_i\rangle$, with $1 \leq i \leq n$, from some fixed set of states known to both parties. She gives the state $|\psi_i\rangle$ to Bob, whose task is to identify the index i of the state Alice has given him.

Suppose the states $\{|\psi_i\rangle\}_i$ are orthonormal. Then Bob can do a quantum measurement to distinguish these states using the following procedure. Define measurement operators $\mathbf{M}_i \coloneqq |\psi_i\rangle \langle\psi_i|$, one for each possible index i, and an additional measurement operator $\mathbf{M}_0$ defined as the positive square root of the positive operator $\mathbb{1} - \sum_i |\psi_i\rangle \langle\psi_i|$. These operators satisfy the completeness relation (Eq. 2.14), and if the state prepared is $|\psi_i\rangle$ then

$$p_i = \langle\psi_i| \mathbf{M}_i |\psi_i\rangle = 1$$

Since the result i occurs with certainty, it is possible to reliably distinguish the orthonormal states $\{|\psi_i\rangle\}_i$.

By contrast, if the states $\{|\psi_i\rangle\}_i$ are not orthonormal then we can prove that there is no quantum measurement capable of distinguishing the states. The idea is that Bob will do a measurement described by measurement operators $\mathbf{M}_j$ with outcome j. Depending on the outcome j, Bob tries to guess the index i by some rule $f(\cdot)$, i.e. $i = f(j)$. The key to why Bob cannot distinguish non-orthogonal states $|\psi_i\rangle$ and $|\psi_k\rangle$ is that $|\psi_k\rangle$ can be decomposed into a non-zero component parallel to $|\psi_i\rangle$, and a component orthogonal to $|\psi_i\rangle$. Suppose j is a measurement outcome such that $f(j) = i$; that is, Bob guesses the state was $|\psi_i\rangle$ when he measures j. Because of the component of $|\psi_k\rangle$ parallel to $|\psi_i\rangle$, there is a non-zero probability of getting outcome j when $|\psi_k\rangle$ is prepared. Therefore, Bob sometimes makes an error in identifying which state was prepared.
A more rigorous proof can be framed as follows. For practicality, we only consider a set with two non-orthonormal states, $|\psi_1\rangle$ and $|\psi_2\rangle$, and we suppose that a measurement to distinguish these states is possible. If the state $|\psi_1\rangle$ [or $|\psi_2\rangle$] is prepared, then the probability of measuring j such that $f(j) = 1$ [or $f(j) = 2$] must be 1. Defining $\mathbf{L}_i \coloneqq \sum_{j:f(j)=i} \mathbf{M}_j^\dagger \mathbf{M}_j$, these observations may be written as

$$\langle\psi_1| \mathbf{L}_1 |\psi_1\rangle = 1 \qquad \langle\psi_2| \mathbf{L}_2 |\psi_2\rangle = 1 \tag{2.17}$$

Since $\sum_i \mathbf{L_i} = \mathbb{1}$, it follows that $\sum_i \langle\psi_1| \mathbf{L}_i |\psi_1\rangle = 1$, and since $\langle\psi_1| \mathbf{L}_1 |\psi_1\rangle = 1$ we must have $\langle\psi_1| \mathbf{L}_2 |\psi_1\rangle = 0$, and thus $\sqrt{\mathbf{L}_2} |\psi_1\rangle = 0$. Suppose we decompose $|\psi_2\rangle = \alpha |\psi_1\rangle + \beta |\varphi\rangle$, where $|\varphi\rangle$ is orthonormal to $|\psi_1\rangle$, $|\alpha|^2 + |\beta|^2 = 1$, and $|\beta| < 1$ since $|\psi_1\rangle$ and $|\psi_2\rangle$ are not orthogonal. Then $\sqrt{\mathbf{L}_2} |\psi_2\rangle = \beta \sqrt{\mathbf{L}_2} |\varphi\rangle$, which implies a contradiction with Eq. 2.17, as

$$\langle\psi_2| \mathbf{L}_2 |\psi_2\rangle = |\beta|^2 \langle\varphi| \mathbf{L}_2 |\varphi\rangle \overset{(*)}{\leq} |\beta|^2 < 1$$

where the marked inequality follows from the observation that

$$\langle\varphi| \mathbf{L}_2 |\varphi\rangle \leq \sum_i \langle\varphi| \mathbf{L}_i |\varphi\rangle = \langle\varphi|\varphi\rangle = 1$$

To close this subsection, we present a particular class of measurements known as **projective measurements**, common in many quantum computation applications and quantum information. Indeed, projective measurements are equivalent to the general measurement postulate if we assume to subject the system to a unitary dynamic before the measurement[5]. The statement of the measurement postulate for projective measurements

[5]For detailed proof, see [NC10]

is superficially rather different from the general postulate, Postulate 3.

Postulate 3bis.
A projective measurement is described by an observable $\mathbf{M}$, a Hermitian operator on the state space of the observed system. The observable has a spectral decomposition

$$\mathbf{M} = \sum_m m\mathbf{P}_m \tag{2.18}$$

where $\mathbf{P}_m$ is the projector onto the eigenspace of $\mathbf{M}$ with eigenvalue m. The possible outcomes of the measurement correspond to the eigenvalues m of the observable. Upon measuring the state $|\psi\rangle$, the probability of getting the result m is

$$p_m = \langle\psi| \mathbf{P}_m |\psi\rangle$$

Given that outcome m occurred, the state of the quantum system immediately after the measurement is

$$\frac{\mathbf{P}_m |\psi\rangle}{\sqrt{p_m}}$$

Projective measurements are a special case of Postulate 3. Suppose the measurement operators in Postulate 3, in addition to satisfying the completeness relation in Eq. 2.14, also satisfy the conditions that $\mathbf{M}_m$ are orthogonal projectors, that is, the $\mathbf{M}_m$ are Hermitian, and $\mathbf{M}_m\mathbf{M}_{m'} = \delta_{m,m'}\mathbf{M}_m$. With these additional restrictions, Postulate 3 reduces to a projective measurement as just defined.

Projective measurements have many nice properties. In particular, it is straightforward to calculate average values for projective measurements. By definition, the average value of the measurement is

$$\begin{aligned}
\mathbb{E}[\mathbf{M}] &= \sum_m m p_m = \\
&= \sum_m m \langle\psi|\mathbf{P}_m|\psi\rangle = \\
&= \langle\psi| \left(\sum_m m\mathbf{P}_m\right) |\psi\rangle \\
&= \langle\psi| \mathbf{M} |\psi\rangle
\end{aligned}$$

This is a useful formula that simplifies many calculations. The average value of the observable $\mathbf{M}$ is often written $\langle\mathbf{M}\rangle \equiv \langle\psi| \mathbf{M} |\psi\rangle$. From this formula for the average follows a formula for the standard deviation associated to observations of $\mathbf{M}$:

$$\sigma^2(\mathbf{M}) = \langle(\mathbf{M} - \langle\mathbf{M}\rangle)^2\rangle = \langle\mathbf{M}^2\rangle - \langle\mathbf{M}\rangle^2$$

The standard deviation is a measure of the typical spread of the observed values upon measurement of $\mathbf{M}$. In particular, if we perform a large number of experiments in which the state $|\psi\rangle$ is prepared and the observable $\mathbf{M}$ is measured, then the standard deviation $\sigma(\mathbf{M})$ of the observed values is determined by the formula $\sigma(\mathbf{M}) = \sqrt{\langle\mathbf{M}^2\rangle - \langle\mathbf{M}\rangle^2}$.

We notice that a commonly used phrase is "*to measure in a basis* $\{|m\rangle\}_m$", where $\{|m\rangle\}_m$ is an orthonormal base. This means to perform the projective measurement with projectors $\mathbf{P}_m = |m\rangle\langle m|$.

2.1.4 Fourth Postulate

In quantum computation, we are mostly interested in **composite quantum systems** comprising two or more distinct quantum systems. The following postulate describes how the state space of a composite system is built up from the state spaces of the component systems.

Postulate 4.
The state space of a composite physical system is the *tensor product* of the state spaces of the component physical systems. Hence, it is still a Hilbert space. 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

$$|\psi_1\rangle \otimes |\psi_2\rangle \otimes \cdots \otimes |\psi_n\rangle$$

Since the most basic state space is a qubit, having a system of n-qubit leads to work in a Hilbert space of dimension $N = 2^n$, as per a generalized version of Eq. 1.1.

Suppose we have two qubits. If these were two classical bits, there would be four possible states - $00, 01, 10, 11$. Correspondingly, a two-qubit system has four computational basis states denoted $|00\rangle, |01\rangle, |10\rangle, |11\rangle$. Since we are still working in a Hilbert space, just of dimension 4, a pair of qubits can also exist in superpositions of these four states of the form

$$|\psi\rangle = \alpha_{00}|00\rangle + \alpha_{01}|01\rangle + \alpha_{10}|10\rangle + \alpha_{11}|11\rangle$$

where we assume that the normalization condition $\sum_{x\in\{0,1\}^2}|\alpha_x|^2 = 1$ is verified. Similarly to the one-qubit case, the measurement result $x \in \{00, 01, 10, 11\}$ occurs with probability $|\alpha_x|^2$, with the state of the qubits after the measurement being $|x\rangle$.

In a composite system, we may also measure just a subset of qubits instead of the whole system. For simplicity, consider a two-qubit system again and suppose to measure only the first qubit. Measuring the first qubit alone gives 0, for instance, with probability $|\alpha_{00}|^2 + |\alpha_{01}|^2$, leaving the post-measurement state

$$|\psi'\rangle = \frac{\alpha_{00}|00\rangle + \alpha_{01}|01\rangle}{\sqrt{|\alpha_{00}|^2 + |\alpha_{01}|^2}}$$

Operations acting on a composite system must still be unitary operators, as postulated in Postulate 2. If the operations apply to the system as a whole, there is not much we can say a priori. However, suppose they act independently on different subsystems. In that case, we can use the matrix tensor product to represent such evolution since the tensor product of unitaries is still a unitary operator. Suppose, for example, we have a three-qubit system and apply a Hadamard operator on the first qubit and a Pauli $\mathbf{X}$ on the third, leaving the second qubit unaltered. Then, the unitary operation acting on the whole system is

$$\mathbf{U} = \mathbf{H} \otimes \mathbb{1} \otimes \mathbf{X} \tag{2.19}$$

where we used the identity matrix to signal that no operator acts on the second qubit. The operator $\mathbf{U}$ of Eq. 2.19 acts on the system as follows:

$$\mathbf{U}|\psi_1\rangle|\psi_2\rangle|\psi_3\rangle = (\mathbf{H} \otimes \mathbb{1} \otimes \mathbf{X})|\psi_1\rangle|\psi_2\rangle|\psi_3\rangle = (\mathbf{H}|\psi_1\rangle) \otimes (|\psi_2\rangle) \otimes (\mathbf{X}|\psi_3\rangle) \tag{2.20}$$

This construction can be easily generalized to larger systems, and since the tensor product is also multiplicative, serial tensor products can still be applied qubit-wise as in Eq. 2.20, that is

$$(\mathbf{M}_1 \otimes \cdots \otimes \mathbf{M}_n)(\mathbf{N}_1 \otimes \cdots \otimes \mathbf{N}_n) = (\mathbf{M}_1\mathbf{N}_1) \otimes \cdots \otimes (\mathbf{M}_n\mathbf{N}_n) \tag{2.21}$$

for all unitary operators $\mathbf{M}_1, \ldots, \mathbf{M}_n, \mathbf{N}_1, \ldots, \mathbf{N}_n$. As in Eq. 2.21, adding a subscript to the operator is common practice to indicate the qubit they act upon.

Focusing on quantum computation, there are a few multi-qubit operations worth discussing algorithm-agnostically:

- the **SWAP** is a two-qubit operation that, as its name suggests, swaps the states of the qubits when differing, i.e.

  $$|01\rangle \to |10\rangle \qquad |10\rangle \to |01\rangle \qquad |00\rangle \to |00\rangle \qquad |11\rangle \to |11\rangle$$

  As for its matrix, we can represent the **SWAP** in the following way:

  $$\mathbf{SWAP} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- Suppose we have two systems, a qubit Q and some other quantum system T. Given a unitary operator $\mathbf{U}$ acting on T, we define the **controlled version of U** as

  $$\mathbf{CU} \coloneqq |0\rangle\langle 0| \otimes \mathbb{1}_T + |1\rangle\langle 1| \otimes \mathbf{U} = \begin{bmatrix} \mathbb{1}_T & 0 \\ 0 & \mathbf{U} \end{bmatrix} \tag{2.22}$$

  where $\mathbb{1}_T$ is the identity applied on the system T. In this scenario, Q is the **control-qubit** while T is the **target system**, onto which $\mathbf{U}$ acts if and only if the control-qubit is in state $|1\rangle$. Suppose T is also a single qubit, and we want to apply a controlled bit-flip. This operation is called **Controlled-NOT**, denoted as **CNOT** or **CX**, and its associate matrix is

  $$\mathbf{CNOT} = |0\rangle\langle 0| \otimes \mathbb{1} + |1\rangle\langle 1| \otimes \mathbf{NOT} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

- It is also possible to have composite systems as control qubits in a controlled operation. The most famous multi-controlled operation is the **Toffoli** unitary, also called **CCNOT** as it has two control qubits and, if both are in state $|1\rangle$, applies a $\mathbf{X}$ onto the target qubit.

  $$\mathbf{CCNOT} \equiv \mathbf{TOF} \coloneqq (|00\rangle\langle 00| + |01\rangle\langle 01| + |10\rangle\langle 10|) \otimes \mathbb{1} + |11\rangle\langle 11| \otimes \mathbf{NOT}$$
  $$\equiv \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

Finally, Postulate 4 enables us to define one of the most fascinating concepts associated with composite quantum systems - **entanglement**. Consider the two-qubit state

$$|\psi\rangle = \frac{|00\rangle + |11\rangle}{\sqrt{2}} \tag{2.23}$$

This state, known as the first **Bell state** $|\Phi^+\rangle$, is such that there are no single qubit states $|a\rangle, |b\rangle$ such that $|\psi\rangle = |a\rangle \otimes |b\rangle$. In general, a composite system that cannot be written as a product of states of its component systems is an **entangled state**.

Together with the one in Eq. 2.23, there are three other Bell states $|\Phi^-\rangle, |\Psi^+\rangle, |\Psi^-\rangle$ representing the most famous examples of entangled states. They are defined as follows:

$$|\Phi^-\rangle = \frac{|00\rangle - |11\rangle}{\sqrt{2}}$$
$$|\Psi^+\rangle = \frac{|01\rangle + |10\rangle}{\sqrt{2}}$$
$$|\Psi^-\rangle = \frac{|01\rangle - |10\rangle}{\sqrt{2}}$$

These four states form the so-called **Bell basis** for a composite space of dimension 4.

2.2 Adiabatic Theorem

We now state two validity conditions - one using the energy gap and the other using the computation time - to make a closed quantum system evolve adiabatically. These statements follow [SWL04], which also gives proof for each condition and treats the case of open quantum systems.

A closed quantum system evolves unitarily through a time-dependent Schrödinger equation

$$\mathcal{H}(t)\,|\psi(t)\rangle = i\,|\dot{\psi}(t)\rangle \tag{2.24}$$

where $\mathcal{H}(t)$ denotes the Hamiltonian and $|\psi(t)\rangle$ is a quantum state in a N-dimensional Hilbert space. We use units where $\hbar = 1$. For simplicity, we assume that the spectrum of $\mathcal{H}(t)$ is entirely discrete and nondegenerate. Thus, we can define an instantaneous basis of eigenenergies[6] by

$$\mathcal{H}(t)\,|n(t)\rangle = E_n(t)\,|n(t)\rangle$$

with the set of eigenvectors $|n(t)\rangle$ chosen to be orthonormal. In this simplest case, where each energy level corresponds to a unique eigenstate, we can define adiabaticity as the regime associated with an independent evolution of the instantaneous eigenvectors of $\mathcal{H}(t)$. This means that instantaneous eigenstates at one time evolve continuously to the corresponding eigenstates at later times and that their corresponding eigenenergies do not cross. In particular, if the system begins its evolution in a specific eigenstate $|n(0)\rangle$, then it will evolve to the instantaneous eigenstate $|n(t)\rangle$ at a later time t, without any transition to other energy levels.

Now, let

$$\Delta_{nk}(t) \coloneqq E_n(t) - E_k(t)$$

[6] **Instantaneous eigenstates** are quantum states that have a constant value for a specific observable at a particular instant in time.

be the energy gap between level n and k, and let τ be the total evolution time. A general validity condition for adiabatic behaviour can be stated as follows:

$$\max_{0\leq t\leq\tau}\left|\frac{\langle k|\dot{\mathcal{H}}|n\rangle}{\Delta_{nk}}\right| \ll \min_{0\leq t\leq\tau}|\Delta_{nk}| \tag{2.25}$$

Note that the left-hand side of Eq. 2.25 has dimensions of frequency and hence must be compared to the relevant physical frequency scale, which can be proved to be given by the gap Δ_{nk} [Mos01]. The interpretation of the adiabaticity condition in Eq. 2.25 is that for all pairs of energy levels, the expectation value of the time-rate-of-change of the Hamiltonian, in units of the gap, must be small compared to the gap.

A handy and equivalent alternative is to express the adiabaticity condition in terms of the total evolution time τ. We consider for simplicity a nondegenerate $\mathcal{H}(t)$ once again. Taking the initial state as the eigenvector $|m(0)\rangle$, the condition for adiabatic evolution can be stated as follows:

$$\tau \gg \frac{|\langle\dot{\mathcal{H}}\rangle|_{max}}{\Delta^2_{min}} \tag{2.26}$$

where

$$|\langle\dot{\mathcal{H}}\rangle|_{max} = \max_{0\leq s\leq 1}\left\{\left|\left\langle k(s)\left|\frac{d\mathcal{H}}{ds}\right|m(s)\right\rangle\right|\right\}$$

and

$$\Delta_{min} = \min_{0\leq s\leq 1}|\Delta_{mk}(s)|$$

with $s = t/\tau \in [0,1]$. Eq. 2.26 states that the total evolution time must be much larger than the norm of the Hamiltonian's time derivative divided by the energy gap's square. It gives an important validity condition for the adiabatic approximation, which has been used, e.g., to determine the running time required by adiabatic quantum algorithms.

2.3 Quantum Computing

The formalization of quantum mechanics quickly propagated through disciplines and, over the second half of last century, quantum computing turned from simulation to reality, with researchers trying to prove its advantages over classical computation: a first result was achieved by **David Deutsch** and **Richard Jozsa** in 1992, who proved that a quantum computer could solve a specific class of black box problems exponentially faster than any classical counterpart [DJ92]. Even though their problem has little to no use and it was designed ad-hoc to be easy to solve on a quantum computer and hard on a classical computer, their algorithm proved that quantum computing had a yet-to-be-discovered potential, marking the first turning point of this field. In this same category, we can also find the so-called **Simon's problem**, designed by Dan Simon in 1994, another black box problem whose resolution has an exponential speedup over its classical counterparts [Sim94]. Since this algorithm requires the black box to be built and computed optimally, its practical use is pretty limited. Still, its impact was gigantic, both theoretically and in practice: in fact, its formulation helped formalize the quantum complexity classes, which the reader can find in a few paragraphs, while the strategy used paved the way for Peter Shor's factorization algorithm, published later in 1994.

Shor's algorithm [Sho94] can be considered as the first quantum algorithm with a practical purpose: its original goal was to find the prime factors of a given number, but following alterations made it possible to solve also the discrete logarithm and the period

finding problems. Regardless of the designated problem, this algorithm guarantees to find a solution in **polynomial time**, obtaining a quantum speedup over the best classical algorithms. This significant reduction in time sparked curiosity and interest in quantum computers and quantum algorithms: considering that most of the cryptosystems in use today are based on the difficulty of solving said problems, Shor's algorithm represents a potential threat to modern protocols and new quantum-safe strategies and systems are still in work today. It is also important to underline that this threat is only theoretical and will continue to be so until the quantum hardware has enough qubits to run reliably for large numbers. For more information on the limitations regarding the implementation of this algorithm, the reader may refer to [Cai23] [GE21].

The final step in this first quantum revolution took place in 1996, with the formulation of a quantum database search algorithm also known as **Grover's algorithm** [Gro96]. Designed by Lov Grover, this algorithm only has a quadratic speedup over a classical search algorithm, which is not as drastic as Shor's. Still, contrary to its predecessors, it can be used in a much more comprehensive range of problems, proving that quantum computing could become a powerful tool in many different areas, such as solving optimization problems [BBW05], gaining a speedup in unstructured data search (especially in NP problems that have a brute force subroutine) [Amb04] or attacking symmetric-key cryptography, including collision attacks and pre-image attacks [BL17].

Over the last two decades, quantum engineering and quantum computing have proliferated, gathering more and more interest from researchers, organizations, and the general public. Given the theoretical supremacy that algorithms propose, it is natural to wonder why quantum computing is not the to-go approach to solving complex problems. What limits quantum computers right now is physics: qubits, unlike classical bits, are subject to **decoherence**, meaning that they collapse into classical states quite rapidly, and their life span decreases even more quickly once something acts on them, such as time evolution operators, entanglement, or other environmental factors. This makes working with qubits extremely difficult, slowing down the scalability of the available qubits in a quantum processor. Another significant limitation is the presence of **quantum noise**: a direct consequence of the Heisenberg uncertainty principle; this phenomenon identifies errors in elementary physical components due to unwanted or imperfect interactions. Since it is an intrinsic characteristic of quantum mechanics, building a quantum processor that does not show quantum noise is impossible, so one must rely on **quantum error correction** to get fault-tolerant hardware. We can then differentiate between a **logical qubit**, the abstract entity that behaves as one would ideally expect, and **physical qubits**, the real objects onto which quantum information is stored. Multiple physical qubits are needed to emulate an error-tolerant logical qubit, significantly increasing the capacity that a quantum processor should reach to allow a quantum advantage. There is no general rule to estimate the number of physical qubits required to emulate a logical qubit, as it depends on the implemented error-correction scheme and on the error rates of the physical qubits, but in the worst-case scenario, up to a thousand physical qubits could be necessary for just a logical qubit [Fow+12].

Quantum error correction is one of the main areas of interest, given its importance towards scalable processors, and over the last years, many diverse strategies have been proposed and/or implemented. We refer to [Cai+23] for a review on this topic.

Given the current state-of-art and the restrictions acting on it, we are now living in what the experts define as the **Noisy Intermediate-Scale Quantum (NISQ) era**: quantum computers are now a reality, but their computational capacity does not allow to gain a quantum advantage and greatly restricts the problems we can tackle with them.

Even so, new quantum algorithms can be defined and tested up to the available capacity, preparing for when quantum computers will be powerful enough to go beyond the NISQ era.

Different algorithms can take advantage of various hardware designs, and every quantum processor has a distinctive topology that exploits its components' chemical and physical properties. Nevertheless, we can classify them based on the underlying quantum paradigm, i.e., **gate-based computing**, presented in Chapter 3, and the **quantum annealing computing**, presented in Chapter 5.

Chapter 3

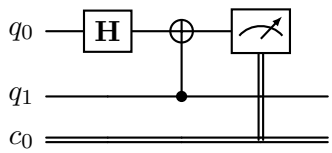
Gate-based Computing

In this chapter, first, we are going to describe the essential components of quantum algorithms and how we can arrange them to build a **quantum circuit** (Section 3.1) before moving on to describe the **transpilation process** (Section 3.2), a routine necessary to evaluate circuits on real quantum devices. Finally, we discuss the main computational classes in **quantum complexity theory** (Section 3.3), just as we do for classical computing in Section 1.2.

3.1 Quantum Circuits

Quantum analogue of logical circuits, **Quantum circuits** are the backbone of the gate-based quantum computing. They are mainly used to devise and analyze quantum algorithms through a straightforward and intuitive approach [Deu89], but they also represent a universal model for quantum computing used to discuss complexity classes and algorithms feasibility [Yao93][MW19].

A quantum circuit is defined starting from its **registers**, i.e., multiple lines that represent either qubits or classical bits. In the former case, we refer to **quantum registers**, and the corresponding lines are printed as single lines; in the latter, about **classical registers** with double lines. In these registers, every line identifies a single computational unit and we locate operations on them. Let us consider, for example, the following circuit:



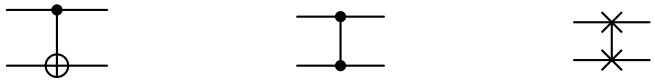
In this example, we have a quantum register composed of two qubits, whose default state is $|0\rangle$, and a classical register with one bit, onto which we save the measurement output. Before measuring, we apply a Hadamard gate **H** on the first qubit, followed by a Controlled-NOT **CX** with control on the second qubit (identified by $\bullet$) and target on the first (identified by $\oplus$).
Mathematically, we could represent this circuit through the following matrix operations:

$$\mathbf{CX}_{10} \cdot (\mathbf{H} \otimes \mathbb{1}) \, |00\rangle$$

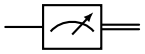
where the subscript "10" is to be read as "control-target". We want to point out that the order of the gates is *reversed* compared to the circuit layout, and this is because right matrix multiplication must be implemented since quantum states can be represented as column vectors.

We will not give a thorough presentation on the symbols of quantum circuits, but let us report the circuit representation of *elementary*[1] gates. For single-qubit operations (see Subsection 2.1.2), we have, respectively, Pauli's $\mathbf{X}, \mathbf{Y}, \mathbf{Z}$, Hadamard, and phase gates $\mathbf{S}$ and $\mathbf{T}$:

while for two-qubit operations (see Subsection 2.1.4) the elementary gates are Controlled-**NOT**, Controlled-**Z**, and the **SWAP** gate:



where we have conveniently assumed that the control qubit is always on the first line and the target on the second. Even though it is not technically a gate (i.e., a unitary operator), it is common to represent the measurement operation as its own gate in the circuit:



where the qubit collapse is represented through the single-line (quantum register) input and the double-line (classical register) output.

So far, quantum circuits have retraced the same buildup as classical ones, operating on quantum states through unitary gates instead of logical gates acting on bits. Even though the line of thought is similar to that of classical circuits, there are a few essential differences to consider.

First, working on superpositions allows us to run our algorithms on multiple inputs, shifting our focus to enhancing or deflating the probabilities of the measurable quantum states based on their correctness. Once the measurement has been performed, the quantum system (or at least the measured fraction) collapses into a classical bit-string, forcing us to re-initialize the whole system even if we want to modify a small portion of the code.

This is where a second critical distinction arises: let's suppose that, for debugging purposes, we want to measure a qubit and store a copy of said qubit to continue our computation once we have checked the measurement output. For arbitrary quantum states, though, it is not possible to create such a copy, as the following theorem states:

Theorem 3.1 (**No-Cloning Theorem**).
Suppose we have two quantum systems A and B with a common Hilbert space $\mathbb{H}_A = \mathbb{H}_B = \mathbb{H}$. Then there is no unitary operator $\mathbf{U}$ on $\mathbb{H} \otimes \mathbb{H}$ such that

$$\mathbf{U}(|\varphi\rangle_A |b\rangle_B) = e^{i\alpha(\varphi,b)} |\varphi\rangle_A |\varphi\rangle_B$$

for all normalised states $|\varphi\rangle_A$ and $|b\rangle_B$ in $\mathbb{H}$, and for some real number α depending on φ and b. The extra factor indicates that a quantum-mechanical state defines a normalized vector in Hilbert space only up to a phase factor.

[1] The term "*elementary*" refers both to the fact that these are the most simple gates to define and to a more general simulation property, as discussed in a few paragraphs.

Proof. Let's start by selecting an arbitrary pair of states $|\varphi\rangle_A, |\psi\rangle_A \in \mathbb{H}$. Since $\mathbf{U}$ must be unitary, we would have

$$\begin{aligned}\langle\varphi|\psi\rangle\langle b|b\rangle &\equiv \langle\varphi|_A\langle b|_B|\psi\rangle_A|b\rangle_B = \\ &= \langle\varphi|_A\langle b|_B U^\dagger U|\psi\rangle_A|b\rangle_B = \\ &= e^{-i(\alpha(\varphi,b)-\alpha(\psi,b))}\langle\varphi|_A\langle\varphi|_B|\psi\rangle_A|\psi\rangle_B \equiv \\ &\equiv e^{-i(\alpha(\varphi,b)-\alpha(\psi,b))}\langle\varphi|\psi\rangle^2\end{aligned}$$

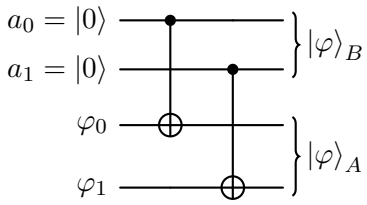
Since the quantum state $|b\rangle$ is normalized by definition, we get

$$|\langle\varphi|\psi\rangle|^2 = |\langle\varphi|\psi\rangle|$$

This implies that $|\langle\varphi|\psi\rangle| \in \{0, 1\}$. Hence by the Cauchy-Schwarz inequality either $|\varphi\rangle = e^{i\beta}|\psi\rangle$ or $|\varphi\rangle$ is perpendicular to $|\psi\rangle$. However, this is not true for two *arbitrary* states. Therefore, a single universal $\mathbf{U}$ cannot clone a general quantum state.

■

We remark that the previous statement concerns the impossibility of cloning an *arbitrary* state $|\varphi\rangle$. Indeed, if we were to clone any standard basis state, we could easily achieve our goal through a series of simple controlled-NOTs:



This does not contradict the No-Cloning theorem since this exact implementation could not create a copy of the $|+\rangle$ state, for instance. Therefore, even though the copy operator depends on the state we want to clone, we are nonetheless able to copy its information on some **ancilla qubits** (also called **work qubits**) to store it and use it for future calculations.

Besides using them as quantum information storage, ancillae are mainly utilized as **garbage qubits**, i.e., units storing some mid-computation state needed in later operations to not act on the original input qubit, which is "*thrown away*" in the end. But can we really get rid of these garbage registers? Discarding a register during computation is physically equivalent to measuring it, bringing unintentional consequences to the amplitudes of our state [NC10]. Hence, the correct procedure is to uncompute whatever operation we computed on the ancillae to restore them to their original state. This procedure is called **compute-uncompute**, and it reduces the number of auxiliary qubits we may need to implement our algorithm since they can be reused after every uncomputing. We report an example of this technique in Fig. 3.1, where we wanted to simulate a multi-controlled $\mathbf{Z}$ gate.

One may now wonder why we would need to simulate a gate instead of directly applying it to our quantum system. Due to hardware restrictions, only single-qubit operations or two-qubit interactions can be performed. These operators are called **elementary gate** and are used to *simulate* every other gate that acts on three or more qubits. As for classical computation with logical gates, we can define a set of **universal quantum gates** that can approximate any unitary transformation on a quantum computer to an arbitrary degree of accuracy. Many universal sets have been defined and documented

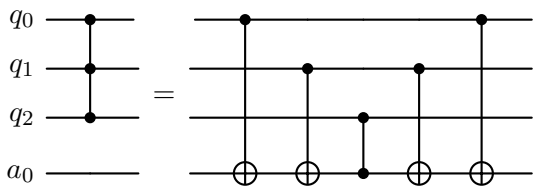


Fig. 3.1: On the left side, our **CCZ** gate as a single 3-qubit gate. On the right side, we find its decomposition through **CNOT** gates, a **CZ** gate, and a single ancilla. The idea is to store the information about the control qubits on the ancilla and then use this one qubit as the control for the phase flip, whose target is the original target qubit (in our case, q_2). Finally, we uncompute the controlled-NOTs we applied on the ancilla.

[Bar+95] [Bar95] [Aha03], with the most common being composed by single-qubit rotations $(\mathbf{R}_x(\theta), \mathbf{R}_y(\theta), \mathbf{R}_z(\theta))$, the phase shift gate $(\mathbf{P}(\varphi))$, and a specific two-qubit entangling gate, such as the Controlled-NOT gate (**CX**) [WW11]. Further details on universality and universal gates can be found in [NC10] [KSV02] and [Wat08].

3.2 Transpiling Process

Simulating quantum gates is also a fundamental step in actual implementation since every processor has its own set of native universal gates. Given the physical restrictions of a quantum system (e.g., decoherence, and noise), this simulation must be optimally carried out to obtain a final circuit that matches the topology of the quantum device. Thus, a rewriting process called **transpilation** is implemented for every circuit we submit to the quantum processor. Most subroutines used in this process are based on heuristics and have only experimental proof; nonetheless, their efficacy is well-tailored to the available hardware.

Fig. 3.2: Steps of the transpilation process.

To run our experiments using the gate model (see Chapter 6 and Chapter 7), we made use of IBM's quantum devices, and thus we wrote the code using their open-source software development kit named **Qiskit**. Hence, we rely on Qiskit's method `transpile` [IBMa] to take care of this mapping, of which we quickly outline the functioning:

1. **Translation stage**
   Given a quantum circuit and a quantum device, we must first ensure that the circuit's gates are all native to the processor. If not, the transpiler will decompose said gates using the processor's set of universal gates.

2. **Layout stage**
   We remark once again that the circuits we define when designing an algorithm are just *abstract objects* to sketch our idea. To implement this on an actual quantum

device, besides dealing with native gates, we must also find an injective map that will embed our qubits and their interactions onto physical qubits. The transpiler will try to find a perfect embedding first, starting with a trivial layout (every virtual qubit is mapped to the same-indexed physical qubit) and then searching for an isomorphic graph in the processor's layout. If such a perfect solution does not exist, the transpiler moves on to a heuristic strategy to find an optimal solution. The strategy used depends on the optimization level the user decides to implement. Still, the primary reference for this subroutine is the `SabreSwap` [LDX18], applied to a random initial layout.

3. **Routing stage**
   Since native gates are universal, transpiled circuits will only have gates acting on one or two qubits. However, hardware layout is never isomorphic to a complete graph with as many nodes as qubits in the processor, hence introducing the need to add as many **SWAP** gates as necessary to make sure the interacting qubit in the circuit can also interact in their physical rendition. Finding such a configuration, though, is $\mathsf{NP}-\mathsf{Hard}$, so we must rely on the `SabreSwap` heuristic once more to try and find the transpiled circuit that uses the fewest swaps.

4. **Optimization stage**
   Decomposing the given gates into native gate composition, though indispensable, greatly increases the number of gates in our transpiled circuit, making the computation more susceptible to decoherence and errors. The transpiler's solution to this is running an optimization routine to combine and eliminate gates, decreasing if possible[2] the number of gates in the transpiled circuit. Different gate optimizations are turned on with different `optimization_level` values when calling the transpiling method on a circuit, with higher values corresponding to a more complete optimization.

5. **Scheduling stage**
   After the circuit has been translated to the native gate basis, mapped to the device, and optimized, a scheduling phase can be applied to account for all the idle time in the circuit. One could think of this step as inserting delays into the circuit to account for idle time on the qubits between operations [IBMa] [Cór+21].

Let's now move on to a code example to focus on the transpilation output.

```
from qiskit import QuantumCircuit, transpile
from qiskit.providers.fake_provider import FakeLondonV2, FakeRomeV2

## Toffoli
ccx_circ = QuantumCircuit(3)
ccx_circ.ccx(0, 1, 2)

## transpiling on FakeLondonV2
ccx_london = transpile(ccx_circ, FakeLondonV2())

## transpiling on FakeRomeV2
ccx_rome = transpile(ccx_circ, FakeRomeV2())

print('Original depth:', ccx_circ.depth(),
```

[2] In some cases, the implemented routines are so effective that the final circuit has fewer gates than the input one; in other cases, not much can be done to optimize the circuit. Hence, running it on noisy devices may be challenging.

```
15      'Transpiled depth on FakeLondonV2 backend:', ccx_london.depth(),
16      'Transpiled depth on FakeRomeV2 backend:', ccx_rome.depth())
```

Listing 3.1: Transpiling code example.

After the necessary imports (lines [1-2]), we built a quantum circuit with a single Toffoli gate (lines [5-6]). Then we transpiled it onto two different quantum devices (also referred to as **backends**), namely `FakeLondonV2` and `FakeRomeV2`, both with five qubits. For the transpiled circuits, see Fig. 3.3 and Fig. 3.4. As the prefix preludes, these backends are not real quantum machines, but to discuss the transpilation, we do not need anything more than a fixed topology and set of native gates. Finally, we printed quite an interesting piece of information about the original circuit and the transpiled ones, i.e., their **depth**, a quantity defined as the length of the critical path in the circuit. More intuitively, circuit depth tells us the maximum number of gates between the input and the output, considering the serial dependency of the performed operations[3]. The deeper a circuit, the longer the computation will run and the more difficult it will be to contrast noise and decoherence. Hence, we must always factor this quantity into designing a quantum algorithm with multi-qubit operations.
Going back to our example, we can see that the original circuit, the single Toffoli gate, has a trivial depth of 1, while the transpiled circuits on `FakeLondonV2` and `FakeRomeV2` have a depth of 15 and 19 respectively. This number depends on the native gates set and backend layout. Still, it is not *fixed*: for more complex circuits, the depth of the circuit will change from transpilation to transpilation, given the heuristic and aleatory nature of its subroutine. The standard procedure is then to run the transpiling process multiple times and save the circuit with minimum depth to further optimize our computation.

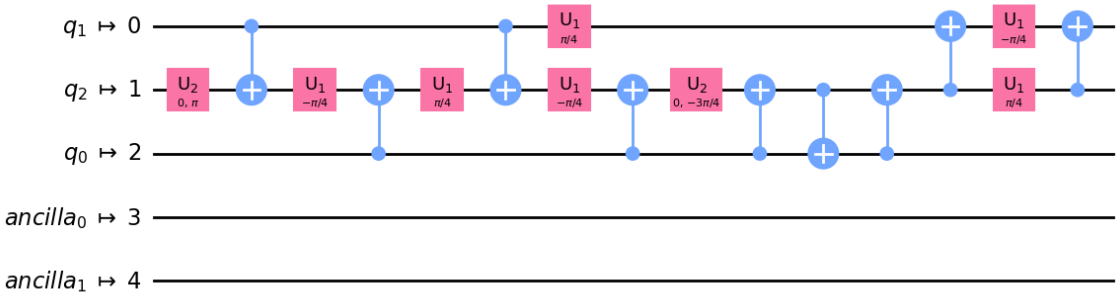


Fig. 3.3: Transpiled Toffoli gate on the FakeLondonV2 backend.

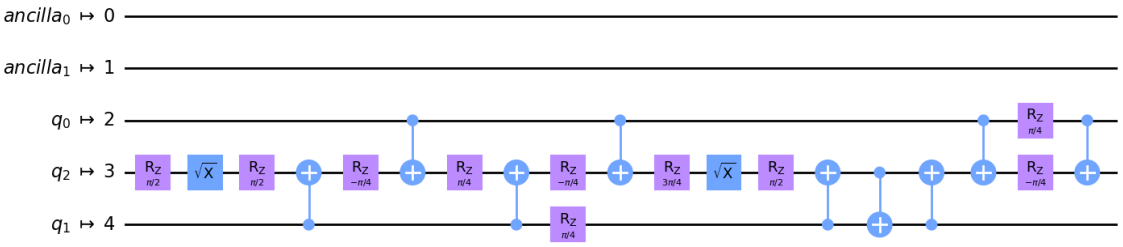


Fig. 3.4: Transpiled Toffoli gate on the FakeRomeV2 backend.

[3]See Appendix: Calculating the depth of a circuit for a few examples of different circuit depths.

Gate decomposition also affects the circuit **size** - the total number of gates in it. Along with circuit depth, it can be intended as a feasibility measure of a quantum circuit, as in what is referred to as **quantum circuit complexity**. As a general rule, the deeper the circuit and the larger the size, the more complicated it gets to implement the circuit.

This analysis focuses on how difficult implementing a circuit is with a *fixed-length input*, implicitly supposing we have already defined such a circuit. Hence, it does not fully represent what problems are computable or efficiently solvable on a quantum computer. As in classical computing, we can refer to **quantum complexity theory** to retrieve such information.

3.3 Quantum Complexity Theory

Concerning computability, it has been proven that any computational problem solvable by a classical computer is also solvable by a quantum computer [NC10], and vice versa. Hence, quantum computers do not provide any additional power over classical computers in terms of computability; they only provide efficiency.

Even though it is possible to define a **quantum Turing machine** to model the effects of a quantum computer [Ben80], quantum circuits have been proven to be a polynomially-equivalent alternative [Yao93][MW19], and have since turned into the most used model to discuss quantum complexity, too.

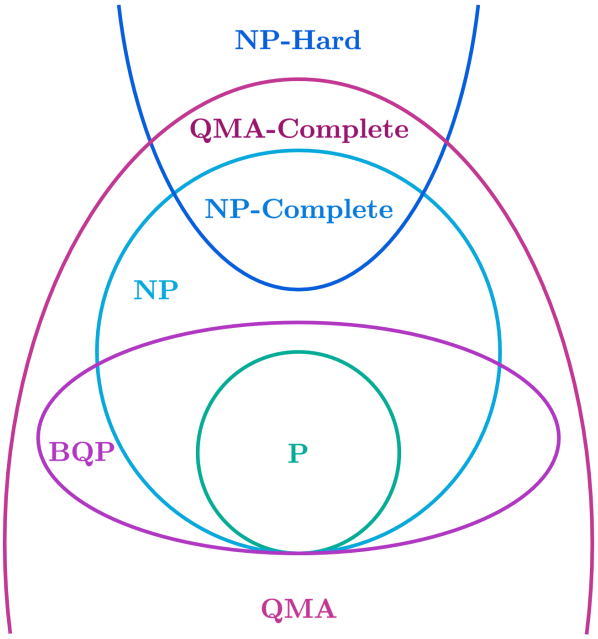


Fig. 3.5: Diagram of the quantum computational complexity classes versus the classical ones.

Therefore, following the blueprint of Subsection 1.2.1, we define the main **quantum complexity classes** using binary languages[4] and quantum circuits as in [Aar]:

BQP	Class of languages $L \subseteq \{0,1\}^*$ for which there exists a *uniform* family of polynomial-size quantum circuits $\{C_n\}$ over some basis of universal gates and a polynomial q so that for all n and inputs $x \in \{0,1\}^*$, if $x \in L$, then $C_n(\|x\rangle \|0\rangle^{q(n)})$ accepts with probability at least $2/3$. If not (i.e. $x \notin L$), then $C_n(\|x\rangle \|0\rangle^{q(n)})$ accepts with probability at most $1/3$. Since circuits must specify the input size a priori, we need a circuit for each input size n. By *uniform*, we mean that there is a classically efficient algorithm to produce C_n given n. BQP stands for "*Bounded*", "*Quantum*", and "*Probabilistic*".
QMA	Quantum analogue of the NP class, the **Quantum Merlin-Arthur** gathers the problems that can be verified by a polynomial-time quantum verifier (running on a quantum computer) with high probability. Formally, a language $L \subseteq \{0,1\}^*$ is in QMA if and only if there exist a polynomial-time quantum circuit A such that if $x \in L$, then there exists a witness state $\|\varphi\rangle$ so that $A(x, \|\varphi\rangle)$ accepts with probability at least $2/3$; otherwise, if $x \notin L$, then for any witness state $\|\varphi\rangle$, $A(x, \|\varphi\rangle)$ accepts with probability at most $1/3$. This must hold for all inputs x. This definition retraces the Merlin-Arthur protocol one given in Subsection 1.2.1, just with the verifier and the certificate turned quantum.
QMA−Hard	A language $L \subseteq \{0,1\}^*$ is in QMA − Hard if every language in QMA can be reduced to it. It is the quantum analogue of the NP − Hard class.
QMA−Complete	A language $L \subseteq \{0,1\}^*$ is in QMA − Complete if it is in QMA − Hard and in QMA . It is the quantum analogue of the NP − Complete class.

[4]The input of a quantum circuit is always a quantum state. However, it is possible to define a one-to-one correspondence between classical and quantum state by using each binary string as the label of the quantum state, e.g., $011 \mapsto |011\rangle$.

Chapter 4

Variational Quantum Algorithms

In this chapter, we will give an overview of Variational Quantum Algorithms (VQAs), starting from the **intuition** behind them (Section 4.1) and their **workflow** definition (Section 4.2). We will close this chapter by illustrating the main possible choices to implement the **modules** that make up these algorithms (Section 4.3).

4.1 Theoretical background

A common goal of variational algorithms is to find the eigenstate with maximum or minimum eigenvalue of a certain observable. More often than not, this is the energy of the quantum system, i.e., its Hamiltonian $\mathcal{H}$. Let us consider its spectral decomposition:

$$\mathcal{H} = \sum_{k=0}^{N-1} \lambda_k \left|\varphi_k\right\rangle \left\langle\varphi_k\right|$$

where $N = 2^n$ is assumed as the dimensionality of the Hilbert space, λ_k is the k-th eigenvalue or, physically, the k-th energy level, and $\left|\varphi_k\right\rangle$ is the corresponding eigenstate. We can then compute the expected energy of the system in the state $\left|\psi\right\rangle$ as

$$\begin{aligned}\left\langle\psi\right| \mathcal{H} \left|\psi\right\rangle &= \left\langle\psi\right| \left(\sum_{k=0}^{N-1} \lambda_k \left|\varphi_k\right\rangle \left\langle\varphi_k\right|\right) \left|\psi\right\rangle = \\ &= \sum_{k=0}^{N-1} \lambda_k \left\langle\psi|\varphi_k\right\rangle \left\langle\psi|\varphi_k\right\rangle = \\ &= \sum_{k=0}^{N-1} \lambda_k \left|\left\langle\psi|\varphi_k\right\rangle\right|^2\end{aligned}$$

Since we can always rearrange the eigenvalues in increasing order, we can assume $\lambda_0 \leq \lambda_k$ for every k.

Hence

$$\begin{aligned}\langle\psi|\,\mathcal{H}\,|\psi\rangle &= \sum_{k=0}^{N-1}\lambda_k\,|\langle\psi|\varphi_k\rangle|^2 \geq \\ &\geq \lambda_0 \underbrace{\sum_{k=0}^{N-1}|\langle\psi|\varphi_k\rangle|^2}_{=1} = \\ &= \lambda_0\end{aligned}$$

where the last equality follows from $\{|\varphi_k\rangle\}_k$ being an orthonormal basis for our space, making $|\langle\psi|\varphi_k\rangle|^2$ the probability of measuring $|\varphi_k\rangle$. We have, therefore, found a lower bound, namely the **ground state energy**, for our expected energy that holds regardless of the quantum system we are studying:

$$\langle\psi|\,\mathcal{H}\,|\psi\rangle \geq \lambda_0$$

Since the above argument applies to any quantum state $|\psi\rangle$ we can also consider **parameterized states** $|\psi(\boldsymbol{\theta})\rangle$ that depend on a parameter vector $\boldsymbol{\theta}$. We then define our cost function as $C(\boldsymbol{\theta}) \coloneqq \langle\psi(\boldsymbol{\theta})|\,\mathcal{H}\,|\psi(\boldsymbol{\theta})\rangle$, wishing to minimize it. Thus, the minimum will always satisfy

$$\min_{\boldsymbol{\theta}} C(\boldsymbol{\theta}) = \min_{\boldsymbol{\theta}} \langle\psi(\boldsymbol{\theta})|\,\mathcal{H}\,|\psi(\boldsymbol{\theta})\rangle \geq \lambda_0$$

Following this reasoning, our algorithm aims to find a vector $\boldsymbol{\theta}_{opt}$ such that $C(\boldsymbol{\theta})$ is the closest to λ_0.

We have just described the mathematical intuition behind the **variational method** of quantum mechanics, which allows us to calculate the optimal approximation of the ground state by minimizing the expectation value of the system Hamiltonian through a parameterized state. The reason why the variational method is stated in terms of energy minima is that it includes two crucial mathematical assumptions: a finite lower bound to the energy $\lambda_0 > -\infty$ needs to exist for physical reasons, even for $N \to \infty$. In contrast, upper bounds do not generally exist. However, from a mathematical point of view, the Hamiltonian has nothing special about it beyond these assumptions, so the method can be generalized to any other quantum observable that satisfies the same constraints.
The question now is: how do we exploit this method to define a variational algorithm?

4.2 Workflow of a VQA

Variational algorithms include several modules that can be combined and optimized based on algorithm, software, and hardware. This includes a **cost function** that describes a specific problem with a set of parameters, a parameterized circuit called **ansatz** to express the search space with these parameters, and an **optimizer** to explore the search space iteratively. During each iteration, the optimizer evaluates the cost function with the current parameters, and, based on the outcome, the algorithm may move on to a new iteration with updated parameters, or it may proceed to estimate the ground state.

Let us now quickly illustrate the workflow of VQAs:

1. **Initialization:** starting from the **default state** $|0^n\rangle$, VQAs initialize the quantum computer to some non-parameterized state $|\rho\rangle$ called **reference state**. This transformation is represented by the application of a unitary reference operator $\mathbf{U}_R$:

$$\mathbf{U}_R\,|0^n\rangle = |\rho\rangle$$

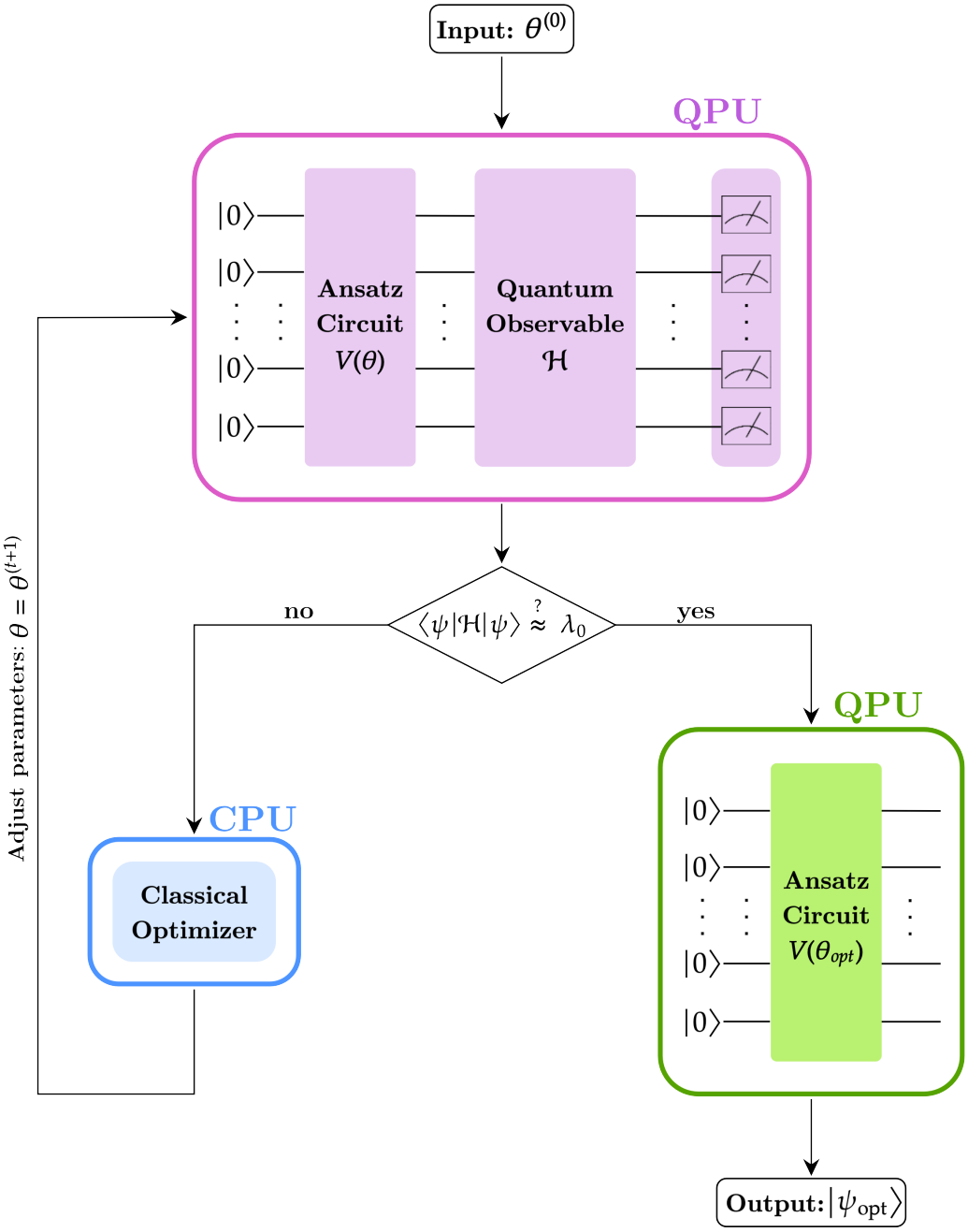


Fig. 4.1: Simplified workflow of a generic variational algorithm. Starting from an initialized vector of parameters $\boldsymbol{\theta}^{(\mathbf{0})}$, we prepare our reference state, which is usually $|0^n\rangle$, and feed it to the ansatz circuit alongside the parameters. The prepared state $|\psi\rangle$ is then forwarded to a circuit designed to measure some quantum observable, commonly assumed to be the Hamiltonian $\mathcal{H}$ of the system. At the end of this quantum run, the now classical output is sent to the CPU, where a classical optimizer generates a new vector of parameters for the QPU (Quantum Processor Unit) to test. Once the function cost (i.e., the expectation value of our Hamiltonian) drops below the wanted accuracy, the final vector of parameters $\boldsymbol{\theta}_{\mathbf{opt}}$ is used to build a final circuit ansatz, whose associated statevector $|\psi_{opt}\rangle$ is a solution to our problem.

2. **Ansatz preparation:** to move from our reference state to the parameterized state onto which the optimization will run, we define a **variational form** $\mathbf{V}(\boldsymbol{\theta})$ to represent a collection of parameterized states for our algorithm to explore. The combination of reference state and variational form is our **ansatz**. Ansatzes will ultimately take the form of parameterized quantum circuits, but we can also refer to them through unitary operations[1]:

$$\mathbf{U}_A(\boldsymbol{\theta}) \coloneqq \mathbf{V}(\boldsymbol{\theta})\mathbf{U}_R$$

All in all, we have

$$\mathbf{U}_A(\boldsymbol{\theta})\,|0^n\rangle = \mathbf{V}(\boldsymbol{\theta})\mathbf{U}_R\,|0^n\rangle = \mathbf{V}(\boldsymbol{\theta})\,|\rho\rangle = |\psi(\boldsymbol{\theta})\rangle$$

3. **Cost function evaluation:** we can encode our problem[2] into a cost function $C(\boldsymbol{\theta})$ comprising a quantum observable (usually the Hamiltonian $\mathcal{H}$) to leverage the variational method. Based on the problem one wants to solve, cost evaluation may be achieved using Qiskit's `primitives` or by implementing specifically designed strategies.

4. **Parameters optimisation :** evaluations are then passed to a classical computer, which then decides whether more iterations are needed. In the former case, a classical optimizer analyses the estimates and outputs a new set of parameters for the ansatz. If we have any prior knowledge about our optimal solution, we can set it as a starting point to bootstrap our optimization.

5. **Adjust ansatz parameters and re-run:** if our stopping criteria are not met, the new parameters chosen by the classical optimizers are used to prepare a new ansatz, and the process is repeated. When iterations halt, a final ansatz circuit is prepared using the current parameters $\boldsymbol{\theta}_{opt}$. The proposed solution state to our problem will then be

$$|\psi(\boldsymbol{\theta}_{opt})\rangle = \mathbf{U}_A(\boldsymbol{\theta}_{opt})\,|0^n\rangle$$

4.3 Modules Analysis

Having illustrated its workflow, we are now ready to explore its modules' different implementations and strategies.

4.3.1 Reference States

A **reference state** is the initial fixed start for our problem, prepared by applying a non-parameterized unitary $\mathbf{U}_R$ at the start of our circuit. If an educated guess or data from an existing optimal solution is known, using it as a starting point will likely make the algorithm converge faster.

The simplest reference state is the **default state**, where we use the qubits of our circuit without any alteration, i.e.

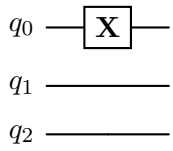
$$|\rho\rangle = |0^n\rangle \qquad \text{and} \qquad \mathbf{U}_R = \mathbb{1}$$

[1] It is common practice to refer to the ansatz including only the parameterized circuit, i.e., $\mathbf{U}_A(\boldsymbol{\theta}) \equiv \mathbf{V}(\boldsymbol{\theta})$, since the default state is generally used as the reference state.

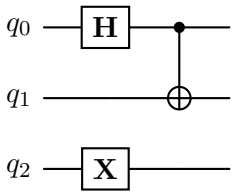
[2] If the problem is physical, this encoding will be straightforward; otherwise, a few techniques have been developed to map the problem into a quantum Hamiltonian. Further details and a few examples of these methods will be given in Chapter 6, Chapter 7, and Chapter 8.

Due to its simplicity, this state is quite versatile and a valid reference state in many diverse scenarios.

Another possibility is to build a **classical reference state** by modifying the string associated with our quantum state. For example, let's suppose we have a three-qubit system and want to start our optimization in the state $|100\rangle$ instead of the default one $|000\rangle$. To construct it, we should then apply a bit-flip (i.e., an $\mathbf{X}$ gate) on the first qubit, obtaining $\mathbf{U}_R = \mathbf{X} \otimes \mathbb{1} \otimes \mathbb{1}$ and the following circuit:

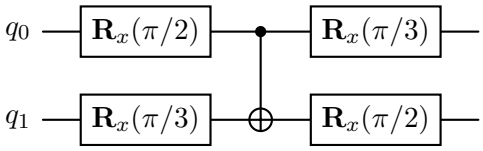


Similarly, we may wish to start with a **quantum reference state**, i.e. a more complex state that involves superposition and/or entanglement, such as $\frac{1}{\sqrt{2}}(|100\rangle + |111\rangle)$. There are many possible circuits to obtain this state from $|000\rangle$, with one approach being the following circuit:



Using the numerical-subscript notation introduced in Subsection 2.1.4, the unitary operator associated with the circuit will be $\mathbf{U}_R = \mathbf{X}_2\mathbf{C}\mathbf{X}_{01}\mathbf{H}_0$, which outputs the desired quantum state.

Since our goal is to tune the parameters of a variational form, a possible strategy is to start our circuit through a parameterized circuit with bound parameters. For instance, we could take a template circuit with tunable rotations and bind the angles of these gates:



One could also implement many **application-specific** reference states, like the one used in Quantum Machine Learning and, more specifically, in the context of a Variational Quantum Classifier (VQC). In VQC, the training data is encoded into a quantum state through a parameterized circuit known as **feature map**, where each parameter represents a data point from the training dataset. One of the most common choices for this kind of mapping is the `ZZFeatureMap`. For further information on Quantum Machine Learning techniques, see [CGD23].

4.3.2 Ansatzes and Variational Forms

At the heart of all variational algorithms lies the key idea of analyzing the differences between states, which are conveniently related through some well-behaved mapping (e.g., continuous, differentiable) from a set of parameters or variables. This collection of states is defined through **ansatz circuits**, which we now proceed to discuss.

A first approach could be to build our ansatz by hand as parameterized circuits, where gates are defined with tunable parameters, which will be bound later. By constructing our ansatz this way, it is easy to realize a significant issue in this approach: an n-qubit system has a vast number of distinct quantum states in the configuration space, so fully exploring it would require an unwieldy number of parameters. Since the runtime complexity of search algorithms grows exponentially with this dimensionality (a phenomenon known as **curse of dimensionality**), implementing these ansatzes is feasible only for small values of n, even though this does not guarantee optimality either.

To counter this drawback, it is common practice to impose some constraints on the variational form such that only relevant states are explored. Finding an efficient ansatz is an active area of research, but the two most common strategies are using either a heuristic-template ansatz or some problem-specific ansatz.

Heuristic ansatzes are primarily used when no information to help restrict the search space is known. In this class of ansatzes, one can find families of parameterized circuits with fewer parameters than in the previous case. However, there are some trade-offs to consider: reducing the search space may make the algorithm run faster, but it may exclude the actual solution to our problem, leading to a suboptimal solution. Furthermore, deeper circuits are affected by more noise, so one must experiment with the ansatz's connectivity, gates, and error rates. Therefore, when defining an ansatz, one must always consider this trade-off: the more parameters there are, the more likely the algorithm will output a precise result, but the longer it will take to run.

Let us now describe the most common designs of ansatzes [Cer+21a].

N-Local Circuits

These circuits consist of rotation and entanglement layers that are repeated alternatively one or more times as follows:

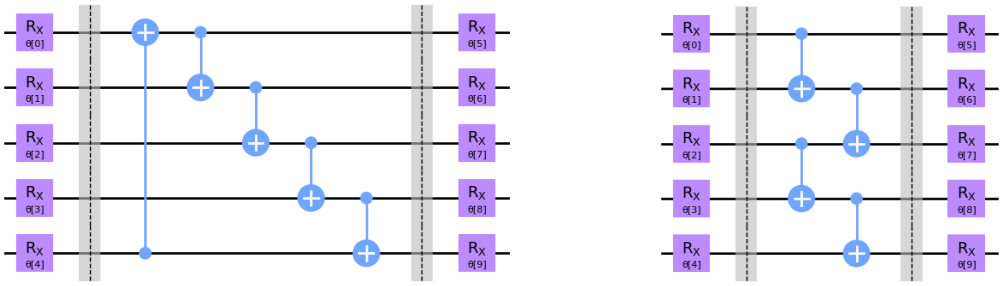
- each layer is formed by gates of size at most N, where N has to be lower than the number of qubits[3] n;
- for a rotation layer, the gates are stacked on top of each other, and any standard rotation can be used, such as $\mathbf{R}_x$ or $\mathbf{CR}_z$;
- for an entanglement layer, we can use gates like Toffoli or **CX** with an entanglement strategy;
- both types of layers can be parameterized or not, but at least one of them has to contain parameters;
- optionally, an extra rotation layer is added at the end of the circuit.

Since these circuits are mainly composed of simple, local gates, they can be efficiently implemented on a quantum computer using a small number of physical qubits. Furthermore, applying local entangling gates helps simulate complex quantum systems, making these ansatzes capture significant correlations between qubits. For these reasons, N-local circuits are one of the most widely used heuristic ansatzes in VQAs.

Specifically, the most commonly used type of N-local circuits is 2-local circuits with single-qubit rotations gates and 2-qubit entanglement gates. In Fig. 4.2, we report two

[3] We believe this notation may need some clarity: N is the go-to symbol when referring to the *dimension* of our Hilbert space (as we previously did ourselves too), but in this case N refers to the *locality* of our operations.

examples of five-qubits 2-local circuits using different entanglement strategies [IBMb]. The vertical dashed lines highlighted in grey are called **barriers** and are a Qiskit visualization tool. They aim to aid circuit representation by creating a clear separation between subcircuits. Hence, they do not have any repercussions on the computation.



(a) Circular entanglement

(b) Pairwise entanglement

Fig. 4.2: **(a)** A 2-local ansatz where every qubit is entangled with the following module n, i.e. $q_i \sim_{ent} q_{i+1 \mod n}$. This strategy is called **circular entanglement**. **(b)** A 2-local ansatz with two layers of entangling operations: a first layer where q_i is entangled with q_{i+1} for all even values of i, and then a second layer where q_i is entangled with q_{i+1} for all odd values of i. This strategy is called **pairwise entanglement**.

Hardware-Efficient Ansatz

The **hardware-efficient ansatz** [Kan+17] is a generic name used for ansatzes aimed at reducing the circuit depth needed to implement the ansatz on the quantum device. The unitaries forming this ansatz are determined from the connectivity and the native gates of the quantum hardware one wishes to use to reduce the transpilation overheard and qubits idle. An example is reported in Fig. 4.3.

One of the main advantages of this approach is its versatility, as it can encode symmetries [Gar+20] [OCG19] and bring correlated qubits closer for depth reduction [Tka+21]. Given their general setup, layered hardware efficient ansatzes are the go-to circuit design when the problem requires no specific layout.

The leading setback of choosing the hardware-efficient ansatz is that it may lead to trainability issues known as **barren plateaus**[4] when randomly initialized.

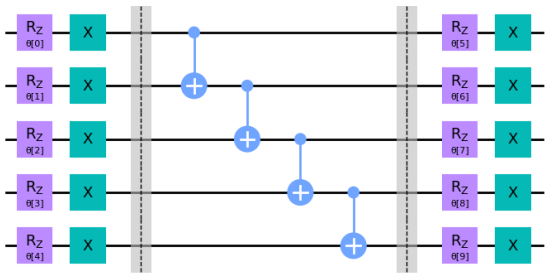


Fig. 4.3: Example of a hardware efficient ansatz called **EfficientSU2** with one repetition for each layer. The circuit consists of single-qubit operations spanned by $SU(2)$ and **CX** entanglements, where $SU(2)$ is the *special unitary group* of degree 2 comprising all 2×2 unitary matrices with determinant 1.

[4] Further details in Subsection 4.3.4

Quantum Alternating Operator Ansatz

The **Quantum Approximate Optimization Algorithm (QAOA)** [FGG14] was initially introduced to obtain approximate solutions for combinatorial optimization problems, but the ansatz used in it quickly became an independent entity itself. This ansatz shares the acronym with its origin article, and it is based on an alternating structure of phase-separation and mixing operations, hence the name **Quantum Alternating Operator Ansatz** [Had+19].

This ansatz is inspired by an approximated adiabatic transformation, where the order p of the approximation determines the precision of the solution[5]. The goal of this ansatz is to map an input state $|\psi_0\rangle$ to the ground state of a given problem Hamiltonian $\mathcal{H}_P$ by sequentially applying a problem unitary $e^{-i\gamma_l \mathcal{H}_P}$ and a mixer unitary $e^{-i\beta_l \mathcal{H}_M}$, where $\mathcal{H}_M$ is a Hermitian operator known as the **mixing Hamiltonian**. Hence, our ansatz will take the following form

$$\mathbf{V}(\boldsymbol{\gamma}, \boldsymbol{\beta}) = \prod_{l=1}^{p} e^{-i\beta_l \mathcal{H}_M} e^{-i\gamma_l \mathcal{H}_P}$$

where $\boldsymbol{\theta} = (\boldsymbol{\gamma}, \boldsymbol{\beta})$, and p is the number of repetitions (or layers in a circuit framework). Even so, decomposing these unitaries into native gates may result in a pretty deep circuit due to many-body terms in $\mathcal{H}_P$ and limited device connectivity.

One of the strengths of this ansatz is the fact that, for specific problems, the search space is reduced a lot from the whole Hilbert space, which could result in a better-performing algorithm[6].

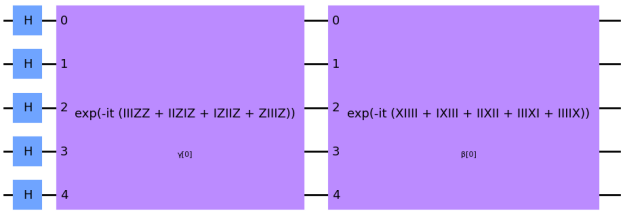


Fig. 4.4: Circuit example of a non-decomposed QAOA ansatz with $p = 1$ layers. The strings in the purple gates represent a Hamiltonian, defined as a linear combination of Pauli operators, identified by the corresponding letter.

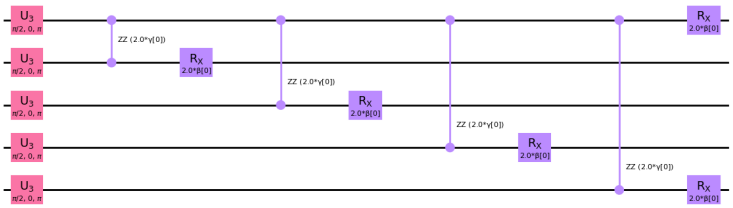


Fig. 4.5: Circuit example of a decomposed QAOAnsatz with $p = 1$ layers. The purple boxes in Fig. 4.4 have been decomposed into rotation gates $\mathbf{U}_3, \mathbf{R}_x$ and $\mathbf{R}_{zz}$, which are respectively a generic single-qubit rotation gate with 3 Euler angles, a single-qubit rotation about the x axis, and a parametric 2-qubit $\mathbf{Z} \otimes \mathbf{Z}$ interaction.

[5] The approximation inspiring this ansatz is known as the **Suzuki-Trotter expansion** [HS05], and it represents a general way of writing exponential operators in quantum simulations.

[6] This is not the case for a few specific instances or problems, such as [Bra+19] [De +23]

4.3.3 Cost Functions

Just as in classical machine learning, the **cost function** $C(\boldsymbol{\theta})$ maps values of the trainable parameters $\boldsymbol{\theta}$ to real numbers, and it defines a hyper-surface usually called the **cost landscape**. The optimizer's task is to explore such a landscape to find the global minima.

We could define the cost function as local or global based on prior knowledge of the landscape. **Local cost functions** make the optimizer search for a point that minimizes the cost function starting at an initial point $C(\boldsymbol{\theta}^{(0)})$ and move to different points based on what they observe in the region they are currently evaluating. This implies that the convergence of these algorithms will usually be fast but can be heavily dependent on the initial point. Furthermore, since optimizers working locally cannot see beyond the region they are evaluating, they can be especially vulnerable to local minima, reporting a wrong solution. On the other hand, **global cost functions** make the optimizer search for the point that minimizes the cost function over the whole domain. This makes the optimization less susceptible to local minima and somewhat independent of initialization, but also significantly slower to converge to a proposed solution.

Either way, when defining a cost function, there are a few desirable criteria that should be met:

- **Faithfullness:** the minimum of $C(\boldsymbol{\theta})$ must correspond to the solution of our problem;
- **Efficient estimation:** we must be able to efficiently estimate $C(\boldsymbol{\theta})$ by performing measurements on a quantum computer and, possibly, performing classical post-processing. It is implicitly assumed that the cost function should not be efficiently computable with a classical computer, as this would hinder the quantum advantage;
- **Operationally meaningfulness:** it is useful for the values of our cost function to get smaller as we approach the global minima, as to indicate a better solution quality;
- **Trainability:** it should be possible to efficiently optimize the parameters $\boldsymbol{\theta}$.

Furthermore, for a given VQA to be implementable on current NISQ hardware, the quantum circuits used to estimate $C(\boldsymbol{\theta})$ must keep the depth and the number of required ancillae small.

As previously stated, it is common practice to define our cost function as the expectation value of the problem Hamiltonian, but how can we estimate it in actual practice? Let's start by expressing the current state of our system $|\psi\rangle$ with respect to the basis of eigenstates of $\mathcal{H}$, i.e.

$$|\psi\rangle = \sum_{\lambda} a_\lambda |\lambda\rangle$$

It then follows:

$$\begin{aligned}
\langle\psi| \mathcal{H} |\psi\rangle &= \left(\sum_{\lambda'} a^*_{\lambda'} \langle\lambda'|\right) \mathcal{H} \left(\sum_{\lambda} a_\lambda |\lambda\rangle\right) = \\
&= \sum_{\lambda} \sum_{\lambda'} a^*_{\lambda'} a_\lambda \langle\lambda'| \mathcal{H} |\lambda\rangle = \\
&= \sum_{\lambda} \sum_{\lambda'} a^*_{\lambda'} a_\lambda \lambda \langle\lambda'|\lambda\rangle = \\
&= \sum_{\lambda} \sum_{\lambda'} a^*_{\lambda'} a_\lambda \lambda \delta_{\lambda'\lambda} =
\end{aligned}$$

$$= \sum_{\lambda} |a_\lambda|^2 \lambda =$$
$$= \sum_{\lambda} p_\lambda \lambda$$

where the last equality follows from $|a_\lambda|^2$ being the probability p_λ to measure $|\lambda\rangle$ when the system is in the state $|\psi\rangle$. Since we do not know the eigenvalues or the eigenstates of $\mathcal{H}$, we must first consider its diagonalization. Since $\mathcal{H}$ is hermitian, we can use the **spectral theorem** and write our Hamiltonian as

$$\mathcal{H} = \mathbf{U}^\dagger \mathbf{\Lambda} \mathbf{U}$$

where $\mathbf{U}$ is a unitary transformation and $\mathbf{\Lambda}$ is the diagonal eigenvalue matrix.

This implies that the expected value can be rewritten as:

$$\begin{aligned}
\langle\psi|\,\mathcal{H}\,|\psi\rangle &= \langle\psi|\,\mathbf{U}^\dagger\mathbf{\Lambda}\mathbf{U}\,|\psi\rangle \overset{(1)}{=} \\
&= \langle\psi|\,\mathbf{U}^\dagger \left(\sum_{j=0}^{2^n-1} |j\rangle\langle j|\right) \mathbf{\Lambda} \left(\sum_{k=0}^{2^n-1} |k\rangle\langle k|\right) \mathbf{U}\,|\psi\rangle = \\
&= \sum_{j=0}^{2^n-1}\sum_{k=0}^{2^n-1} \langle\psi|\,\mathbf{U}^\dagger\,|j\rangle\langle j|\,\mathbf{\Lambda}\,|k\rangle\langle k|\,\mathbf{U}\,|\psi\rangle \overset{(2)}{=} \\
&= \sum_{j=0}^{2^n-1} \langle\psi|\,\mathbf{U}^\dagger\,|j\rangle\langle j|\,\mathbf{\Lambda}\,|j\rangle\langle j|\,\mathbf{U}\,|\psi\rangle = \\
&= \sum_{j=0}^{2^n-1} |\langle j|\,\mathbf{U}\,|\psi\rangle|^2 \lambda_j \overset{(3)}{=} \\
&= \sum_{j=0}^{2^n-1} p_j \lambda_j
\end{aligned}$$

where (1) follows from those sums being the identity matrix decomposed as projective operators, (2) follows from $\langle j|\,\mathbf{\Lambda}\,|k\rangle = \lambda_j \delta_{jk}$ since $\mathbf{\Lambda}$ is a diagonal matrix and, finally, (3) follows from $|\langle j|\varphi\rangle|^2$ is the probability to measure $|j\rangle$ if the system is in the state $|\varphi\rangle = \mathbf{U}\,|\psi\rangle$. Since the probabilities are taken from the state $\mathbf{U}\,|\psi\rangle$, the matrix $\mathbf{U}$ is absolutely necessary, just like the eigenvalues λ_j. However, we assumed that this information is unknown since the goal of VQAs is to find them.

To find a way around this missing information, we are going to use a generalized version of the fact that Pauli operators are a basis of the Hilbert space of 2×2 Hermitian matrices[7]: any $2^n \times 2^n$ matrix can be written as a linear combination of 4^n tensor products of n Pauli matrices and identities, all of which are both Hermitian and unitary with known $\mathbf{U}$ and $\mathbf{\Lambda}$. Possible operators are illustrated in Table 4.1.

Let's rewrite $\mathcal{H}$ with respect to the Paulis and identities:

$$\mathcal{H} = \sum_{k_{n-1}=0}^{3} \cdots \sum_{k_0=0}^{3} w_{k_{n-1}\ldots k_0} \sigma_{k_{n-1}} \otimes \cdots \otimes \sigma_{k_0} = \sum_{k=0}^{4^n-1} w_k \hat{\mathbf{P}}_k$$

where $k = \sum_{l=0}^{n-1} 4^l k_l \equiv k_{n-1}\cdots k_0 \in \{0,1,2,3\}$, and $\hat{\mathbf{P}}_k \coloneqq \sigma_{k_{n-1}} \otimes \cdots \otimes \sigma_{k_0}$. Thus:

$$\langle\psi|\,\mathcal{H}\,|\psi\rangle = \sum_{k=0}^{4^n-1} w_k \sum_{j=0}^{2^n-1} |\langle j|\,\mathbf{V}_k\,|\psi\rangle|^2 \langle j|\,\mathbf{\Lambda}_k\,|j\rangle =$$

[7] Even more general, if we have a basis $\mathcal{B}$ for a vector space V and a basis $\mathcal{B}'$ for a vector space W, then $\{|b\rangle\,|b'\rangle : |b\rangle \in \mathcal{B}, |b'\rangle \in \mathcal{B}'\}$ is a basis for $V \otimes W$.

$$= \sum_{k=0}^{4^n-1} w_k \sum_{j=0}^{2^n-1} p_{kj} \lambda_{kj}$$

where $\mathbf{V}_k \coloneqq \mathbf{V}_{k_0} \otimes \cdots \otimes \mathbf{V}_{k_{n-1}}$ and $\mathbf{\Lambda}_k \coloneqq \mathbf{\Lambda}_{k_0} \otimes \cdots \otimes \mathbf{\Lambda}_{k_{n-1}}$, such that $\hat{\mathbf{P}}_k = \mathbf{V}^\dagger \mathbf{\Lambda}_k \mathbf{V}$.

This is the behind-the-scenes of our cost estimation, and we can either run it implicitly through Qiskit `Estimator` or explicitly by building some auxiliary shallow circuits with Pauli operations and identities.

Operator	σ	$\mathbf{U}$	$\mathbf{\Lambda}$
$\mathbb{1}$	$\sigma_0 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$	$\mathbf{U_0} = \mathbb{1} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$	$\mathbf{\Lambda}_0 = \mathbb{1} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$
$\mathbf{X}$	$\sigma_1 = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$	$\mathbf{U_1} = \mathbf{H} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$	$\mathbf{\Lambda}_1 = \sigma_3 = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$
$\mathbf{Y}$	$\sigma_2 = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}$	$\mathbf{U_2} = \mathbf{H}\mathbf{S}^\dagger = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & -i \\ 1 & i \end{bmatrix}$	$\mathbf{\Lambda}_2 = \sigma_3 = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$
$\mathbf{Z}$	$\sigma_3 = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$	$\mathbf{U_3} = \mathbb{1} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$	$\mathbf{\Lambda}_3 = \sigma_3 = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$

Table 4.1: Pauli operators to decompose any unitary $\mathbf{U}$ as a linear combination of tensor products.

4.3.4 Optimization Loops

As for any variational approach, the success of a variational quantum algorithm depends on the efficiency and reliability of the optimization method used. In addition to classical difficulties, when training a VQA, one can encounter new challenges, such as hardware noise and the presence of barren plateaus. This has led to the development of many quantum-aware classical optimizers, with the optimal choice still being an active topic of debate. We can differentiate between two classes of optimizers, namely the gradient-based and the gradient-free.

Gradient-based methods represent the most common approach to optimization. Their core strategy is to move iteratively in directions indicated by the gradient:

$$\boldsymbol{\theta}^{(n+1)} = \boldsymbol{\theta}^{(n)} - \eta \vec{\nabla} C(\boldsymbol{\theta})$$

where η is a small hyperparameter called **learning rate** that controls the size of the update. The naive version of this algorithm is called **Gradient Descent**, but many other versions have been developed to modulate this strategy to the problem landscape. For instance, if only statistical estimates are given, we can use a **Stochastic Gradient Descent(SGD)** optimizer, which uses these estimates to replace the actual gradient. One SGD method imported from machine learning is **Adam**, which adapts the size of the steps taken during the optimization to allow for more efficient and precise solutions than those obtained through basic SGD [KB14]. An alternative method inspired by the machine learning literature adapts the precision (i.e., the number of shots taken for each estimate), rather than the step size, at each iteration in an attempt to not squander the quantum

resources used [Küb+20]. The main disadvantages of this type of optimization are the convergence speed, which can be very slow, and there is no guarantee of achieving the optimal solution.

A different gradient-based approach uses the **Quantum Natural Gradient Descent** method [Sto+20]. Whereas standard gradient descent takes steps in the steepest direction in the Euclidean geometry of the parameter space, natural gradient descent works instead on a space with a metric that encodes the sensitivity of the quantum state to parameter variations. Using this metric typically accelerates the convergence, allowing a given level of precision to be attained with fewer iterations. This method has also been extended to incorporate the effects of noise [KB22].

On the other hand, **gradient-free methods** do not require gradient information and can be useful when computing the gradient is difficult, expensive, or too noisy. They also tend to be more robust in finding global optima, whereas gradient-based methods tend to converge to local optima. For instance, in our experiments, which will be presented in Chapters 6 and 7, we relied on two gradient-free methods known as **COBYLA** and **Powell**. However, gradient-free methods require higher computational resources, especially for problems with high-dimensional search spaces.

A middle ground between these two classes can be found in **Meta-Learning**. This optimizer trains a neural network to make a good update step based on the optimization history and current gradient with similar optimization problems [Wil+21]. Because the update steps taken are based on rules learned from similar cost functions, this meta-learning approach has significant potential to be highly efficient when used on a new instance of a common class of optimizations.

Barren Plateaus

The cost landscapes of VQAs are generally non-convex and complicated [HD21], making it difficult to obtain general guarantees about the computational expense of the optimizations. The optimizer navigates in this landscape, searching for the global minimum, but if it is relatively flat, finding the appropriate direction to explore can be challenging regardless of the implemented optimization method.

This scenario is referred to as **barren plateau**, signaling that the cost landscape becomes progressively flatter. Unluckily, for a broad range of parameterized circuits, the probability of encountering a barren plateau increases exponentially with the number of qubits, making this phenomenon a constraining bottleneck to implementing VQAs and other quantum machine learning algorithms [MKW21] [Lar+22] [McC+18].

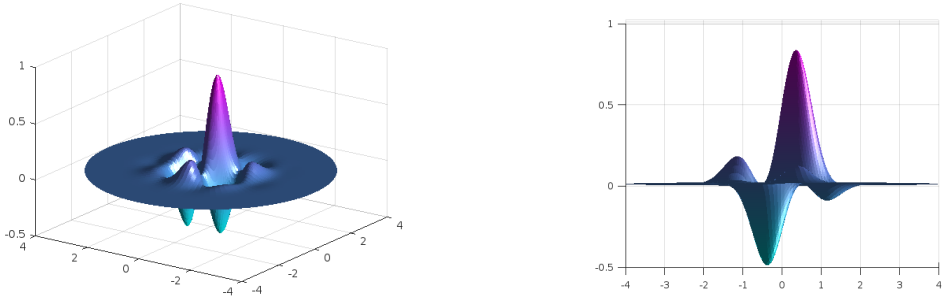


Fig. 4.6: Landscape example with a barren plateau from two different angles[8].

[8]Special thanks to my friend and colleague Pietro for helping me code this picture!

While this area is still under active research, a few precautions can be taken to improve optimization performance:

1. **Bootstrapping** can help the optimization loop avoid getting stuck in a parameter space where the gradient vanishes;
2. **Experimenting with hardware-efficient ansatz.** Since we use a noisy quantum device as a black-box oracle, the quality of those evaluations can affect the optimizer's performance. Hence, implementing a hardware-efficient ansatz may avoid producing vanishing gradients;
3. **Experimenting with error suppression and error mitigation**, reducing the impact of noise and making the optimization process more efficient;
4. **Experimenting with gradient-free optimizers.** Since these methods do not rely on gradient information, barren plateaus are less likely to affect them.
5. **Experimenting with local cost functions** since using only partial information about the circuit may avoid encountering barren plateaus, as proven in [Cer+21b].

Chapter 5

Quantum Annealing

In this chapter, we present the quantum annealing process, starting with an **overview** of its definition and underlying physics (Section 5.1). We then introduce the **Minor Embedding** algorithm, a heuristic process that maps the problem cost function onto a real QPU based on its specific topology (Section 5.2).

5.1 Physics background and algorithm description

In Subsection 1.2.2, we formally defined optimization problems. We also introduced simulated annealing, a heuristic algorithm that exploits thermal fluctuations to find a near-optimal solution to combinatorial problems whose landscape presents local minima points. However, there are two important limitations to consider:

1. Based on the cost function, the landscape may exhibit very high energy barriers around local minima, which does not correspond to a reasonably low cost. In the worst-case scenario, these barriers could even be proportional to the system size N and thus diverge. Hence, many unacceptable local minima might occur, trapping the algorithm for a very long and potentially diverging time.

2. Given that a classical system can only assume one state at a time and that the number of configurations grows exponentially ($\sim 2^n$) with the number of variables n, exploring the global minimum search will require, in the worst case[1], an exponential computational cost, erasing every possible advantage from this technique.

Quantum mechanics has, to some extent, some solutions to both these problems.

First, using quantum mechanics allows for an exploration that uses **quantum tunneling**, a phenomenon that enables optimization to pierce through energy barriers and lowers the relaxation time to the final ground state. This is especially useful in situations where the landscape has high and narrow energy peaks, precisely the ones where simulated annealing would get stuck.
Furthermore, a quantum system can assume multiple states at once (i.e., it can be a superposition of states) and, assuming the kinetic energy is high enough, this can lead to a delocalization of the state itself, which we can imagine as being able to *see* the whole landscape at the same time. Though a powerful tool, quantum mechanics does not resolve

[1] If the search space has a clean landscape with a gradient directly pointing to the global minimum, the algorithm may still lead to the optimal solution in a reasonable time, but that is quite a constricting hypothesis to make so we will keep focusing on the more general setting.

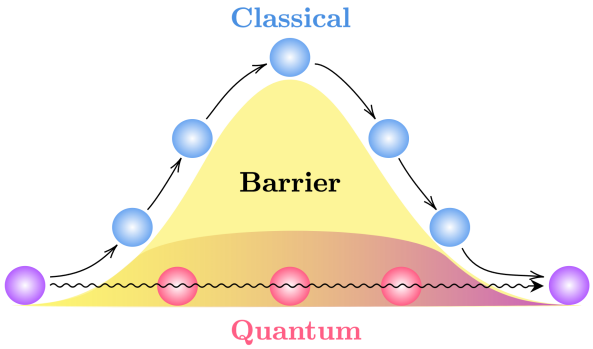


Fig. 5.1: Visual representation of the quantum tunneling phenomenon: instead of following the natural landscape, tunneling pierces barriers and allows faster exploration and relaxation to the searched ground state.

every issue of classical computation. Interestingly, the drawbacks of a quantum algorithm are inherently different from those of a classical algorithm, producing a proliferous still-open debate on the conditions that favour the former or the latter.

To realize a quantum annealing schedule, thermal fluctuations are replaced by quantum fluctuations [AHS93][Fin+94][KN98], which are introduced in the system description as an artificial quantum kinetic term[2] $\Gamma(t)\mathcal{H}_{kin}$ that does not commute with the classical Hamiltonian $\mathcal{H}_P$ representing the cost function. The coefficient $\Gamma(t)$ is the parameter that controls the quantum fluctuations. Hence, the total Hamiltonian of the system is

$$\mathcal{H} = \mathcal{H}_P + \Gamma(t)\mathcal{H}_{kin}$$

and the tunneling probability is approximately

$$p_{tunnel} = e^{-\sqrt{h}w/\Gamma(t)} \tag{5.1}$$

where h and w are respectively the height and the width of the barrier [DCS05]. For example, if our classical problem is formulated using a classical Ising model as in Eq. 1.3, the corresponding quantum Hamiltonian would be

$$\mathcal{H} = \underbrace{-\sum_{i>j} J_{ij}\sigma_i^z\sigma_j^z - \sum_{i=1}^{n} h_i\sigma_i^z}_{=\mathcal{H}_P} \underbrace{-\Gamma(t)\sum_{i=1}^{n}\sigma_i^x}_{=-\mathcal{H}_{kin}}$$

where σ_i^z, σ_i^x are now, respectively, the Pauli operations **Z** and **X** instead of monodimensional spin variables. Initially, $\Gamma(t)$ is kept very high so that the kinetic term dominates, and the ground state is a uniform superposition of all the classical configurations, whose easy realization makes it an excellent choice for a starting state. Following some anneal schedule, $\Gamma(t)$ slowly decreases to zero while the system remains in its instantaneous ground state. This last assertion holds if this evolution runs slowly enough to muster the adiabatic theorem of quantum mechanics (see Subsection 2.2) and, once $\Gamma(t)$ is brought to zero, it assures that the system will be found in the ground state of the final Hamiltonian, which will coincide with the problem Hamiltonian $\mathcal{H}_P$. Furthermore, when $\Gamma(t) \to 0$, the tunneling probability (Eq. 5.1) tends to zero, too, marking the end of the annealing and tunneling procedure.

[2] Choosing a right kinetic term to add to our problem Hamiltonian may improve the annealing results by increasing the gap between the ground state and the first excited state and hence avoiding unwanted energy jumps. See [MN07] for further details.

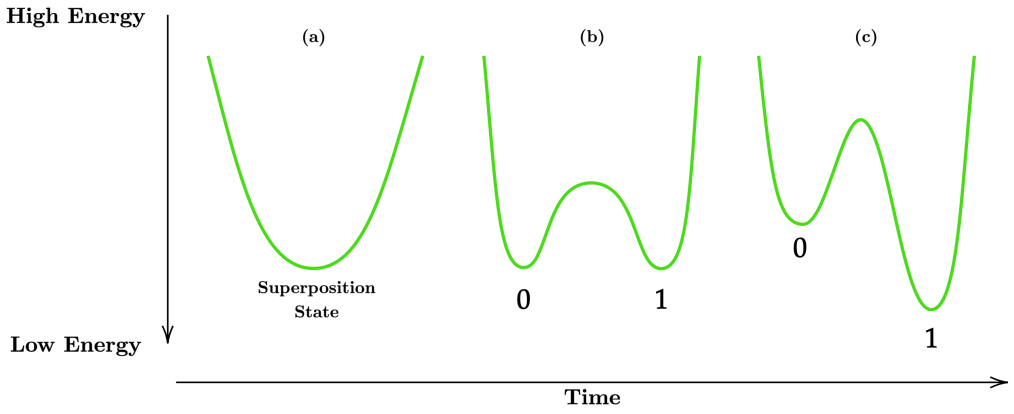


Fig. 5.2: Example of an energy diagram evolution associated with a one-qubit system with the standard eigenbasis $\{|0\rangle, |1\rangle\}$. (a) The system starts in the equal superposition of all classical states, which is the only minimum at this stage. (b) The quantum annealing process starts by raising the barrier and turning the energy diagram into a double-well potential landscape. Each low point corresponds to one of the eigenvalues. The qubit will end in one of these two wells at the end of the computation, but for now, the outcomes share the same probability (i.e. $1/2$). (c) Applying an external magnetic field to the qubit shifts the output probabilities in favour of the deeper well.

Defining the right evolution rate is thus critical to ensure adiabaticity in the quantum annealing schedule. More specifically, for a non-degenerate spectrum with a gap between the ground state and the first excited state, the adiabatic evolution is assured if the evolution time τ satisfies Eq. 2.26, which in our cases means

$$\tau \gg \frac{|\langle \dot{\mathcal{H}} \rangle|_{max}}{\Delta_{min}^2} \tag{5.2}$$

where

$$|\langle \dot{\mathcal{H}} \rangle|_{max} = \max_{0 \leq t \leq \tau} \left\{ \left| \left\langle \varphi_0(t) \left| \frac{d\mathcal{H}}{ds} \right| \varphi_1(t) \right\rangle \right| \right\}$$

and

$$\Delta_{min} = \min_{0 \leq t \leq \tau} |\Delta(t)|$$

with $s = t/\tau \in [0, 1]$. Furthermore, $|\varphi_0(t)\rangle$ and $|\varphi_1(t)\rangle$ are, respectively, the instantaneous ground state and the first excited state of the total Hamiltonian $\mathcal{H}$, and $\Delta(t)$ is the instantaneous gap between the ground state and the first excited state energies [SWL04]. It is important to remember that any quantum hardware may have unexpected noise acting on its system due to environmental interactions. In addition, a longer annealing time τ is believed to increase the effects of the environment on the system, leading to faster decoherence times and a worse success probability [Raj+23]. Since quantum annealing is inherently probabilistic, statistical sampling must always be executed to retrieve the ground state with fair certainty. Hence, even if adiabaticity is broken on a few runs, it may not hinder the search for the ground state much, considering that, even if the algorithm moves to an excited state (i.e., a local minimum), there is a positive probability to tunnel out of it.

Concerning the devices on which a quantum annealing procedure may run, it is possible

to simulate some of its dynamics through quantum Monte Carlo methods[3] to sample the ground state or a mixed state at a low enough temperature for a given set of parameters of the Hamiltonian [DC08]. However, for such Monte Carlo algorithms, there is no general bound on the success time τ as the one provided by the adiabatic theorem for real-time quantum annealing. Alternatively, one may use a **quantum annealer**, i.e., the analog version of the gate-model quantum computer, in which the system follows the real-time Schrödinger evolution. To execute our experiments using quantum annealing (see Chapter 8), we employ this latter option by running our code, written using the **Ocean** software development kit, on D-Wave's quantum annealers.

5.2 Minor Embedding

As of today[4], quantum annealers only allow cost functions formulated as a BQM, either a QUBO or an Ising model (Subsection 1.2.3). However, there is still another intermediate step that allows an algorithm to run on a quantum annealer - the **embedding** of the problem on the actual QPU topology.

To define such an embedding, the cost is foremost mapped to a graph in which the nodes represent the BQM variables (σ_i^z for Ising, q_i for QUBO), with each of their respective coefficients defined as the node **bias**, and the edges are the objective function's quadratic coefficients (J_{ij} for Ising, b_{ij} for QUBO). Let us make an example. Suppose we want to represent the following quadratic function graphically

$$\mathcal{H}(a, b) = 13a + 3ab - 7b$$

The corresponding graph has two nodes $V = \{a, b\}$ with biases 13 and -7, respectively. Furthermore, there is an edge $E = \{(a, b)\}$ with a weight of 3.

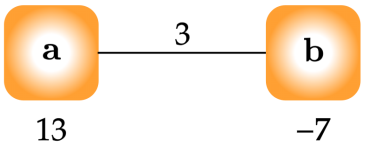


Fig. 5.3: Graphical representation of the cost function $\mathcal{H}(a, b) = 13a + 3ab - 7b$

Once we have this graphical representation, the cost is embedded into the qubits in the QPU through an algorithm known as minor embedding, where the QPU's topology is also processed as a graph. In general, a **minor embedding** of a graph H in another graph G is defined by a function $\varphi : V(H) \to 2^{V(G)}$ called **model** such that

- for each $x \in V(H)$, the subgraph induced by $\varphi(x)$ in G is connected;
- $\varphi(x)$ and $\varphi(y)$ are disjoint for all $x \neq y$ in $V(H)$;
- if x and y are adjacent in H, then there is at least one edge between $\varphi(x)$ and $\varphi(y)$ in G.

[3] In general, even for finite n, it is impossible to follow the entire evolution of a quantum system with a classical computer using polynomial resources, since it would require to keep track of all the amplitudes of all the basis vectors, whose number grows exponentially in n.

[4] Thesis submitted on March 5, 2024.

We refer to $\varphi(x)$ as the **vertex-model** of x and say that $\varphi(x)$ represents x in G.

A few issues may arise due to the QPUs' connectivity when the problem dimension increases. Since the topologies of the QPUs are not fully connected, it may not be possible to embed larger graphs with an elevated number of interactions with such a direct approach, mapping each variable to a single qubit. Hence, minor embedding creates a qubit **chain** to represent a single variable and, through a parameter called `chainstrength`, the algorithm ensures that the same value is assigned to every qubit in the chain by increasing the correlation between the physical qubits. In more detail, since chains represent one variable, we expect every qubit in the chain to share the same value. Hence, any configuration where the qubits in a chain assume different values has a penalization term factored in, reducing its sampling probability. Regardless of the designated chain strength, qubit chains may still **break**, i.e., the qubits in the chain may assume different values during the annealing schedule. However, it is important to analyze chain breaks considering graph connectivity and dimension to fully evaluate their consequences' severity. Thus, chain breaks are usually reported as a percentage, and if this value is not too high, the convergence to the ground state is not precluded.

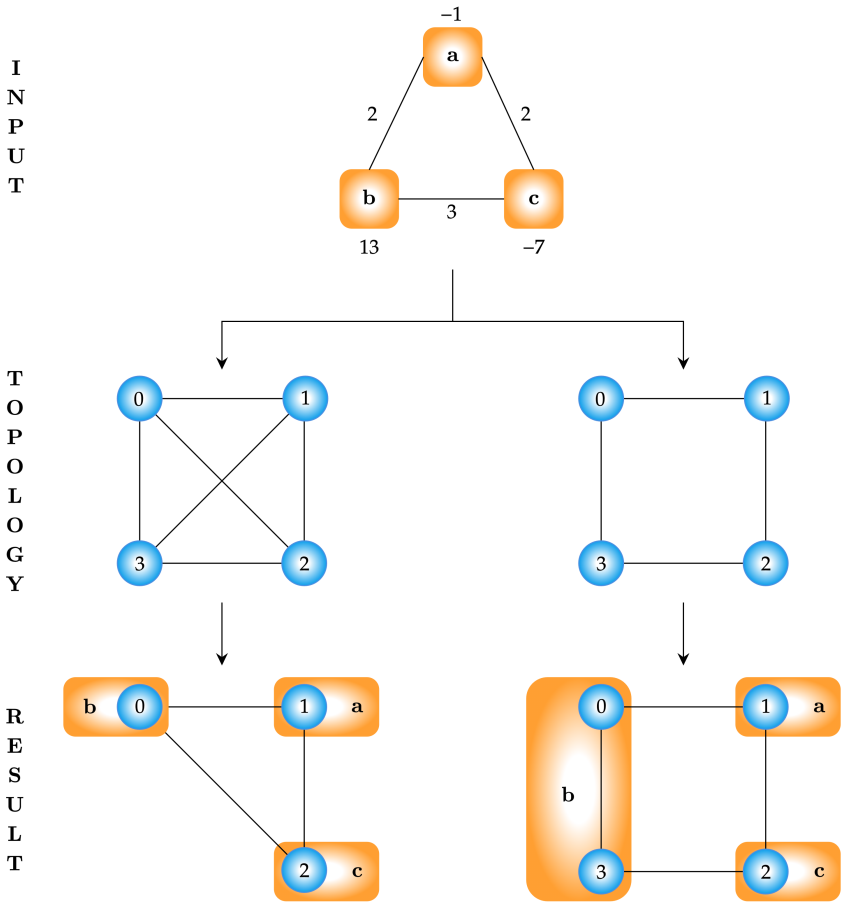


Fig. 5.4: Embedding of a triangular graph (associated to $\mathcal{H}(a,b,c) = -a+13b-7c+2ab+2ac-3bc$) into a fully connected topology (K_4) and a sparse four-node graph, which shows qubit chains usage. The orange squares and the blue circles identify the original variables and the qubits in the processor, respectively.

Due to the limitations of real-hardware connectivity, it is common to express the capacity of a QPU through the largest complete graph (or the largest complete bipartite graph) that can be embedded in the given topology. For this reason, complete graphs are referred to as **unit cells** when describing a QPU's layout.

The embedding process can be implemented manually, but the general approach is to use the automatic minor embedding provided by D-Wave [CMR14], whose pseudo-code is reported in Algorithm 1 and Algorithm 2. The idea behind this heuristic is to proceed iteratively by constructing a vertex-model for each vertex of H based on the locations of its neighbours' vertex-models.

Let us start by describing how to find a good vertex-model. Suppose we want a vertex-model $\varphi(y)$ for $y \in V(H)$, and y is adjacent to $x_1, \ldots, x_k$, which alreadu have vertex-models $\varphi(x_1), \ldots, \varphi(x_k)$. A good vertex-model might ensure that $\varphi(y)$ shares an edge with each $\varphi(x_1), \ldots, \varphi(x_k)$ while minimizing the number of vertices in $\varphi(y)$. Doing so leaves as much room as possible for other vertex-models yet to be determined. To this end, for each x_j, we compute the shortest-path distance from $\varphi(x_j)$ to every vertex g in the subgraph of G of vertices not in any vertex-model. We record such information as a cost $c(g, j)$. Then, we select the vertex g^* with the smallest total sum of distances $\sum_j c(g, j)$ and fix g^* as the root of the new vertex-model $\varphi(y)$. Finally, we identify the shortest path from g^* to each $\varphi(x_j)$ and take their union as $\varphi(y)$.

Since g^* seldom exists, i.e., no vertex in G has a path to each of $\varphi(x_1), \ldots, \varphi(x_k)$ of only yet to-be-used vertices, another strategy must be specified. To bypass this problem, we temporarily allow vertex-models to overlap so that the same vertex can represent multiple vertices of H in G. Then, a good vertex-model is defined to use a minimal number of vertices at which multiple vertex-models appear.

In the initial stage of the minor-embedding algorithm, we find a vertex-model for each vertex x based on the weighted shortest path distances to its neighbours. If none of its neighbours has vertex-models yet, a random vertex of G is selected as $\varphi(x)$. Once this stage is over, the algorithm tries to refine the vertex-models so that no vertex of G represents more than one vertex of H. This is achieved by iteratively going through the (ordered) vertices of H, removing a vertex-model from the embedding and reinserting a better one. In doing so, the shortest paths such that the overlap between vertex-models decreases are selected.

The algorithm stops when at most one vertex-model is represented at any vertex of G, or no further improvement has been achieved after a fixed number of iterations.

Algorithm 1: findMinorEmbedding(G, H)

Data: graph H with vertices $\{x_1, \dots, x_n\}$, graph G.
Result: vertex-models $\varphi(x_1), \dots, \varphi(x_n)$ of an H-minor in G, or "failure".

randomize the vertex order $x_1, \dots, x_n$
$stage \leftarrow 1$
for $i \in \{1, \dots, n\}$ **do**
  $\varphi(x_i) \leftarrow \{\}$
end
while $\max_{g \in V(G)} |\{|i : g \in \varphi(x_i)\}$ *or* $\sum_i |\varphi(x_i)|$ *is improving, or* $stage \leq 2$ **do**
  for $i \in \{1, \dots, n\}$ **do**
    for $g \in V(G)$ **do**
      $w(g) \leftarrow \mathrm{diam}(G)^{|\{j \neq i : g \in \varphi(x_j)\}|}$
    end
    $\varphi(x_i) \leftarrow$ `findMinimalVertexModel`$(G, w, \{\varphi(x_j) : x_j \sim x_i\})$
  end
  $stage \leftarrow state + 1$
end
if $|\{i : g \in \varphi(x_i)\}| \leq 1$ *for all* $g \in V(G)$ **then**
  return $\varphi(x_1), \dots, \varphi(x_n)$
else
  return "failure"
end

Algorithm 2: findMinimalVertexModel($G, w, \{\varphi(x_j)\}_j$)

Data: graph G with vertex weights w, neighbouring vertex-models $\{\varphi(x_j)\}_j$.
Result: vertex-models $\varphi(y)$ in G such that there is an edge between $\varphi(y)$ and each $\varphi(x_j)$.

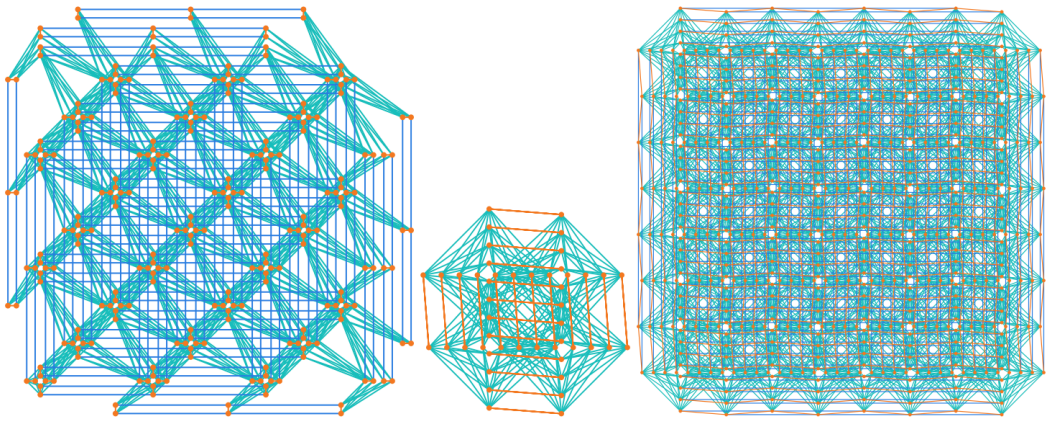
if *all* $\varphi(x_j)$ *are empty* **then**
  return random $\{g^*\}$
end
for $g \in V(G)$ *and all* j **do**
  if $\varphi(x_j)$ *is empty* **then**
    $c(g, j) \leftarrow 0$
  else if then
    $c(g, j) \leftarrow w(g)$
  else
    $c(g, j) \leftarrow w$-weighted shortest-path distance $(g, \varphi(x_j))$ excluding $w(\varphi(x_j))$
end
$g^* \leftarrow \mathrm{argmin}_g \sum_j c(g, j)$
return $\{g^*\} \cup \{$paths from g^* to each $\varphi(x_j)\}$

5.3 Available QPUs' Topologies

D-Wave is the leading company in the quantum annealing scene, and we rely on their cloud-based platform to execute our experiments. Couplers-interconnected qubits form the QPU, shaping their layout either as a **Pegasus** [D-Wa] or a **Zephyr** [D-Wb] topology. The QPU is, therefore, characterized by the total number of qubits, existing couplers, and the connectivity level they grant. These properties may vary slightly due to calibration processes that aim to bring the whole device to a consistent regime. Hence, a small percentage ($< 3\%$) of qubits and couplers may be removed from the accessible fabric. The resulting subgraph of the original topology is referred to as **working graph**.

At the moment[5], the most potent QPU is the **Advantage_system6.3**, which has over $5'000$ qubits arranged in a Pegasus topology. In this layout, unit cells have K_4 and $K_{6,6}$ as native complete subgraphs, while each qubit has an overall degree of 15. As per notation, P_n refers to the instances of Pegasus topologies, i.e., the number of unit cells that constitute the QPU. The Advantage QPU, for example, is a lattice of 16×16 unit cells, and its associated graph can therefore be denoted as P_{16}.

In 2022, D-Wave announced the experimental prototype **Advantage2_prototype1.1** with ~ 500 qubits arranged in a new topology layout called Zephyr. This prototype anticipates the Advantage2 system, which will feature over 7 000 qubits, higher connectivity, and lower error rates, thanks to successful quantum error mitigation techniques [Ami+23]. Since the final trimester of 2023, a new prototype has been expected to be released, and on January 23, 2024, it was announced that it would be available in the cloud service shortly after [D-Wc]. The new **Adavantage2_prototype2.2** was made available on February 12, 2024, featuring over 1 200 qubits and 10 000 couplers [D-Wd]. More in detail, the **Zephyr** topology [D-Wb] enables native K_4 and $K_{8,8}$ subgraphs with qubits of degree 20. As per notation, we use Z_n to refer to a $(2n+1) \times (2n+1)$ grid of unit cells.



(a) P_4 (b) Z_1 (c) Z_4

Fig. 5.5: Pegasus and Zephyr topologies. (a) A Pegasus graph with 264 qubits. (b) A Zephyr graph containing 32 qubits and four unit cells with several couplers. (c) A Zephyr graph with 576 qubits arranged in a 9×9 unit grid. Source:[D-We].

[5]Thesis submitted on March 5, 2024

Chapter 6

Experiments on the Gate Model: VQLS

In this chapter, we will present the workflow of the **Variational Quantum Linear Solver (VQLS)** and its **Qiskit implementation** (Section 6.1), analyzing every module of the algorithm separately. After that, we will discuss the **results** obtained from our experiments and the hardware limitations encountered (Section 6.2).

The code in this chapter is built from scratch and can be found as Jupyter Notebooks in my public repository on **GitHub** [Gal]. Additional information about the software and the hardware used can be found in Appendix: Used Hardware and Software Versions.

6.1 Procedure and Code Analysis

First, let us remark that the VQLS aims to solve **linear systems** of the form:

$$\mathbf{A}\mathbf{x} = \mathbf{b} \tag{6.1}$$

with $\mathbf{A}$ a $N \times N$ matrix, i.e., a system with the same number of equations and unknowns. Therefore, besides a starting point $\boldsymbol{\theta}^{(0)}$ for our optimizer, the VQLS requires two more inputs, namely the matrix $\mathbf{A}$ and a state $|b\rangle$ proportional to the vector $\mathbf{b}$. Given that our algorithm is associated with a circuit, we suppose that $|b\rangle$ can be prepared through an efficient gate sequence $\mathbf{U}_b$ and that $\mathbf{A}$ can be written as a linear combination of L unitaries[1]:

$$|b\rangle = \mathbf{U}_b |0\rangle \qquad\qquad \mathbf{A} = \sum_{l=0}^{L-1} c_l A_l \tag{6.2}$$

where the A_l are unitaries and the c_l are complex numbers. It is implicitly assumed that L is a polynomial function of the number of qubits n, that the A_l unitaries can be efficiently implemented with quantum circuits, and that the condition number κ of $\mathbf{A}$ is finite[2].

The goal of the algorithm is to find an optimal set of parameters $\boldsymbol{\theta}_{opt}$ such that $\mathbf{A}|x_{opt}\rangle \equiv \mathbf{A}|x(\boldsymbol{\theta}_{opt})\rangle$ is proportional to $|b\rangle$. For simplicity, we denote with a subscript

[1] In our experiments, we assumed the matrix to have the form in Eq. 6.2. Otherwise, it is possible to implement a Szegedy quantum random walk to decompose sparse matrices as a sum of unitaries [Bra+23].

[2] The condition number is used to measure the sensitivity of a function, i.e. how much the output value of the function can change for a small change in the input argument. For a matrix $\mathbf{A}$, $\kappa(A)$ is defined as the product of the two operator norms $\|\mathbf{A}^{-1}\|\|\mathbf{A}\|$.

only the optimal values found by the algorithm, and we omit the parameter and iteration dependency for intermediate steps, i.e., $|x\rangle \equiv |x(\boldsymbol{\theta})\rangle$.

Since VQLS is a variational algorithm, we must build an ansatz, define a cost function, and choose an optimizer. As reference state, we will use the default state $|0^n\rangle$ to consider as a general case as possible.

6.1.1 Ansatz

In the VQSL algorithm, $|x\rangle$ is prepared by making the state $|0^n\rangle$ evolve under the action of a trainable gate sequence (i.e., an ansatz) $\mathbf{V}(\theta)$. It is advised to choose either a QAOA ansatz or a Hardware Efficient one, so we decided to implement the latter to have more control over the circuits[3]. Specifically, we used only the **fixed-structure** ansatz, where the position of the gates in the circuit is indeed fixed[4].

Our ansatz is composed of rotations $\mathbf{R}_y(\theta_i)$, depending on our parameters, and of **CZ** gates, alternated in every layer we add. The number of layers is a hyperparameter chosen with heuristic strategies that aim to keep circuit depth as small as possible. In our code, we denote the number of layers as a variable `layers`, and it is set as the value we choose plus 1 since there is a layer zero with only rotations.

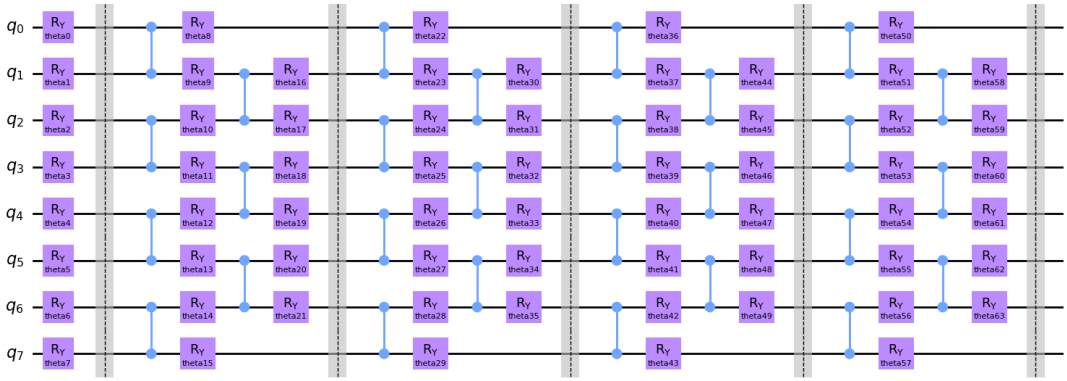


Fig. 6.1: Example of Hardware Efficient Ansatz built for VQSL with 8 qubits and $4+1$ layers.

The suggested implementations differ slightly in the layout of applied operations, which depends on the parity of the number of qubits. Regardless, this does not hinder our optimization.

We now report the Python code written to build such ansatzes:

```
def build_fixed_ansatz(n, layers, parameters):

    qc = QuantumCircuit(n)

    ## initial rotations on layer 0
    for i in range(n):
        qc.ry(parameters[0][i], i)

    ## layers (starting from 1)
    if n%2 == 0:
        for layer in range(1,layers):
```

[3] We followed the guidelines proposed in [Bra+23] for the layout of the layers.

[4] There also exists **variable-structure** ansatzes where one optimizes the gate angles and the gate placement in the circuit for every phase gate.

```
12
13              ## first CZ
14              for i in range(0,n-1,2):
15                  qc.cz(i,i+1)
16
17              ## Y rotations on all qubits
18              for i in range(n):
19                  qc.ry(parameters[layer][0][i], i)
20
21              ## other CZ
22              for i in range(1,n-2,2):
23                  qc.cz(i,i+1)
24
25              ## final Y rotations
26              for i in range(1,n-1):
27                  qc.ry(parameters[layer][1][i-1], i)
28
29      else:
30          for layer in range(1,layers):
31
32              ## first CZ
33              for i in range(0,n,2):
34                  ph_i = (i+1)%n
35                  qc.cz(i,ph_i)
36
37              ## Y rotations on all qubits
38              for i in range(n):
39                  qc.ry(parameters[layer][0][i], i)
40
41              ## other CZ
42              for i in range(1,n):
43                  ph_i = (i+1)%n
44                  qc.cz(i, ph_i)
45
46              ## final Y rotations
47              for i in range(n):
48                  qc.ry(parameters[layer][1][i], i)
49
50      return(qc)
```

Listing 6.1: Hardware Efficient Ansatz function implemented in the code.

The function `build_fixed_ansatz`[5] takes as input the number of qubits `n`, the number of layers `layers` (comprehensive of layer zero), and the parameters. This latter variable is implemented as `list[list]` object in Python, hence the multiple calls to access its values when appending the rotation gates (lines [27] or [49], for instance).

6.1.2 Cost Function

The idea behind the VQLS cost function is to estimate how much orthogonal $|\psi\rangle \coloneqq \mathbf{A}\,|x\rangle$ is to $|b\rangle$ and this can be achieved either through a global cost function or a local one.

Global Cost Function

Intuitively, we can define the global cost function as the overlap between the unnormalized projector $|\psi\rangle\langle\psi|$ and the subspace orthogonal to $|b\rangle$, i.e.

$$\hat{C}_G = \mathrm{Tr}\left(|\psi\rangle\langle\psi|(\mathbb{1} - |b\rangle\langle b|)\right) \equiv \langle x|\mathcal{H}_G|x\rangle \tag{6.3}$$

[5] The "*fixed*" refers to the structure of the ansatz.

which can be viewed as the expectation value of the following effective Hamiltonian[6]:

$$\mathcal{H}_G = \mathbf{A}^\dagger(\mathbb{1} - |b\rangle\langle b|)\mathbf{A}$$

It is important to notice that $\hat{C}_G$ is small either if $|\psi\rangle \propto |b\rangle$ or if the norm of $|\psi\rangle$ is small: the latter, though, does not represent a solution to our problem. Hence, we need to normalize the state to work with a cost function that will be minimized only when a solution is found. The modified cost function can then be defined as

$$C_G = \frac{\hat{C}_G}{\langle\psi|\psi\rangle} = 1 - |\langle b|\Psi\rangle|^2 \tag{6.4}$$

where $\Psi = |\psi\rangle / \sqrt{\langle\psi|\psi\rangle}$.

To estimate $\langle\psi|\psi\rangle$, let's consider

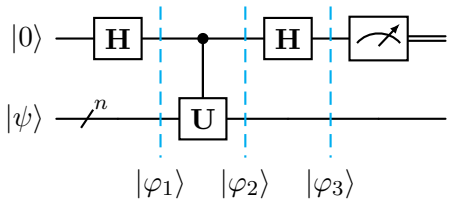
$$\langle\psi|\psi\rangle = \sum_{ij} c_i c_j^* \beta_{ij} \tag{6.5}$$

with

$$\beta_{ij} = \langle 0|\mathbf{V}^\dagger \mathbf{A}_j^\dagger \mathbf{A}_i \mathbf{V}|0\rangle \tag{6.6}$$

Our goal is then reduced to estimate the L^2 values[7] β_{ij}, which is achieved through a subroutine called **Hadamard Test**.

Let $\mathbf{U}$ be a unitary acting on n qubits, and $|\psi\rangle$ a n-qubit quantum state. We also assume to be able to apply the controlled version of the unitary $\mathbf{U}$. Then, the Hadamard test is a circuit that can be used to evaluate the value $\langle\psi|\,\mathbf{U}\,|\psi\rangle$. The circuit is pretty simple; it consists of a Hadamard gate applied on an auxiliary qubit, the controlled application of the unitary $\mathbf{U}$, and another Hadamard gate:



where the second register represents a bundle of n qubit lines. The initial operation leads to $|\varphi_1\rangle = (\mathbf{H}\otimes\mathbb{1})\,|0\rangle\,|\psi\rangle = |+\rangle\,|\psi\rangle$, then we have

$$|\varphi_2\rangle = \mathbf{CU}\,|+\rangle\,|\psi\rangle = \frac{1}{\sqrt{2}}(\,|0\rangle\,|\psi\rangle + |1\rangle\,\mathbf{U}\,|\psi\rangle\,)$$

and finally

$$\begin{aligned}
|\varphi_3\rangle &= (\mathbf{H}\otimes\mathbb{1})\frac{1}{\sqrt{2}}(\,|0\rangle\,|\psi\rangle + |1\rangle\,\mathbf{U}\,|\psi\rangle\,) = \\
&= \frac{1}{2}\Big(\,|0\rangle\,(|\psi\rangle + \mathbf{U}\,|\psi\rangle) + |1\rangle\,(|\psi\rangle - \mathbf{U}\,|\psi\rangle)\Big) = \\
&= \frac{1}{2}\Big(\,|0\rangle\,(\mathbb{1}+\mathbf{U})\,|\psi\rangle + |1\rangle\,(\mathbb{1}-\mathbf{U})\,|\psi\rangle\,\Big)
\end{aligned}$$

[6] An **effective Hamiltonian** is a Hamiltonian that acts in a reduced space and only describes a part of the eigenvalue spectrum of the true (more complete) Hamiltonian

[7] The number of values to estimate can be reduced in half with some precautions [Bra+23].

Hence, the probability to measure 1 in the first qubit is:

$$\begin{aligned} p_1 &= \left|\left|\frac{1}{2}(\mathbb{1} - \mathbf{U})\left|\psi\right\rangle\right|\right|_2^2 = \\ &= \frac{1}{4}\Big(\left\langle\psi\right| - \left\langle\psi\right|\mathbf{U}^\dagger\Big)\Big(\left|\psi\right\rangle - \mathbf{U}\left|\psi\right\rangle\Big) = \\ &= \frac{2 - \left\langle\psi\right|(\mathbf{U} + \mathbf{U}^\dagger)\left|\psi\right\rangle}{4} = \\ &= \frac{2 - 2\mathrm{Re}(\left\langle\psi\right|\mathbf{U}\left|\psi\right\rangle)}{4} \end{aligned}$$

Therefore we can compute $\mathrm{Re}(\left\langle\psi\right|\mathbf{U}\left|\psi\right\rangle)$ as

$$\mathrm{Re}(\left\langle\psi\right|\mathbf{U}\left|\psi\right\rangle) = 1 - 2p_1 \tag{6.7}$$

for any unitary operation $\mathbf{U}$. This procedure also allows us to compute the imaginary part of the expected value of $\mathbf{U}$ by just adding an $\mathbf{S}$ phase gate on the first qubit, in between the first Hadamard gate and the controlled unitary $\mathbf{CU}$. However, for the experiments run in this thesis, we used only unitary operations with no imaginary part, so the formula found in Eq. 6.7 will suffice.

Going back to our dot product $\langle\psi|\psi\rangle$, we are going to estimate the β_{ij} using $\mathbf{U} = \mathbf{V}^\dagger\mathbf{A}_j^\dagger\mathbf{A}_i\mathbf{V}$ to build our circuit for every $i, j \in \{0, L-1\}$.

```
def hadamard_test_fixed_beta(gates_ai, gates_aj, n, aux_index,
    parameters, layers, case):

    qc = QuantumCircuit(n+1, 1)
    qc_qubits = [i for i in range(1, n+1)]

    ## first hadamard
    qc.h(aux_index)

    ## phase gate
    if case == 'imaginary':
        S_dagger = Operator([[1,0],[0,-1j]])
        qc.unitary(S_dagger, aux_index, label='S^*')

    ## barrier
    qc.barrier()

    ## ansatz (V)
    qc.compose(build_fixed_ansatz(n=n, layers=layers,
                                  parameters=parameters),
               qubits=qc_qubits,inplace=True)

    ## barrier
    qc.barrier()

    ## Controlled-A_i
    index_count = 1
    CAi = QuantumCircuit(n+1, 1)
    for pauli in gates_ai:
        if pauli == 'Z':
            CAi.cz(aux_index, index_count)
        elif pauli == 'X':
            CAi.cx(aux_index, index_count)
        index_count += 1
```

```
34      qc.compose(CAi, inplace=True)
35
36      ## barrier
37      qc.barrier()
38
39      ## Controlled-A_j^dagger
40      index_count = 1
41      CAj = QuantumCircuit(n+1, 1)
42      for pauli in gates_aj:
43          if pauli == 'Z':
44              CAj.cz(aux_index, index_count)
45          elif pauli == 'X':
46              CAj.cx(aux_index, index_count)
47          index_count += 1
48      qc.compose(CAj.inverse(), inplace=True)      # A_j is unitary
49
50      ## barrier
51      qc.barrier()
52
53      ## final hadamard
54      qc.h(aux_index)
55
56      ## measure aux
57      qc.measure(aux_index, 0)
58
59      return(qc)
```

Listing 6.2: Function implementing the Hadamard test for β_{ij} with a fixed-hardware ansatz.

The first inputs to this function are the unitary operations $\mathbf{A}_i, \mathbf{A}_j$ which are assumed to be given as a product of **Pauli operators** or, Python equivalently, as a string of characters in the alphabet $\{\texttt{I}, \texttt{X}, \texttt{Y}, \texttt{Z}\}$. This means that if we are given $\mathbf{A}_i = \texttt{IIZ}$ then[8]

$$\mathbf{A}_i = \mathbf{Z} \otimes \mathbb{1} \otimes \mathbb{1}$$

Following our unitaries, we pass to the function five more inputs, namely the number of qubits `n`, the index of the auxiliary qubit to use as control (which is conventionally zero) as `aux_index`, the parameters, and finally, a string variable `case` to switch between real and imaginary part. As you can see in lines $[17-20]$, the ansatz $\mathbf{V}$ we build is not the associated controlled version. That is because we can make our system evolve to the state $|x\rangle$ and then equivalently estimate $\langle x| \mathbf{A}_j^\dagger \mathbf{A}_i |x\rangle$, potentially reducing the overhead cost and the circuit depth[9]. The Hadamard test is then built piece by piece, composing subcircuits into one final circuit `qc` that is returned as function output.

Since we have no way of knowing a priori the probabilities of our final statevector, we must rely on **quasi-distributions**, i.e., the distribution of the outputs based on the results of multiple circuit runs. Hence, we will execute every Hadamard test for a user-defined `num_shots` number of times and compute the quasi-distributions of the registered outputs.

```
1 def get_psi_dot_psi(n,layers,paulis_list,paulis_coeffs,parameters):
2
3     dotprod = 0
4     num_paulis = len(paulis_list)
```

[8] Qiskit uses the little-endian convention, so when converting a string of Pauli operators one must remember to reverse the order of the gates in the tensor product.

[9] This trick can be used for every gate in a Hadamard test that is applied both as itself and as its conjugate transpose, mimicking the compute-uncompute scheme.

```
5
6       for i in range(num_paulis):
7           for j in range(num_paulis):
8
9               num_shots = 100000
10              backend = Aer.get_backend('aer_simulator',
11                                        num_shots = num_shots)
12
13              cc = paulis_coeffs[i]*paulis_coeffs[j]
14
15              qreg = QuantumRegister(n+1, name='q')
16              creg = ClassicalRegister(1, name='c')
17              qc = QuantumCircuit(qreg, creg)
18
19              qc.compose(hadamard_test_fixed_beta(paulis_list[i],
20                                                  paulis_list[j],
21                                                  n, 0, parameters,
22                                                  layers, 'Real'),
23                         inplace=True)
24
25              transpiled_qc = transpile(qc, backend)
26              result = backend.run(transpiled_qc).result()
27
28              outputstate = result.get_counts()
29
30              prob1 = 0
31              if '1' in outputstate.keys():
32                  prob1 = float(outputstate['1'])/num_shots
33              beta_ij = 1-2*prob1
34
35              dotprod += cc*beta_ij
36
37      return(dotprod)
```

Listing 6.3: Function that computes $\langle\psi|\psi\rangle$ trough Hadamard tests.

The Python function in Listing 6.3 follows the steps we illustrated so far: for every possible combination of i and j, we build the associated Hadamard test (lines $[15-23]$), we execute its transpiled version (lines $[25-26]$), and finally we retrieve the information about the quasi-distributions by analyzing the frequency of the output `'1'` over the `num_shots` runs (lines $[30-33]$). After β_{ij} has been computed, it is added to the `dotprod` variable scaled by the product of coefficients `cc` defined in line [13]. Even though $c_i, c_j \in \mathbb{C}$, in our code, we only use real coefficients, so we omit the complex conjugate operation on c_j. There is one last element to discuss in this code snippet: the backend defined in [10]. We choose the local simulator `aer_simulator` to run the first experiments without worrying about queues or noise. In Section 6.2, we will leverage a real QPU and a noisy simulation to compare the results.

To compute the global cost function, though, we are still missing the term $|\langle b|\psi\rangle|^2$, which can be computed similarly to $\langle\psi|\psi\rangle$:

$$|\langle b|\psi\rangle|^2 = |\langle 0|\mathbf{U}_b^\dagger \mathbf{A}\mathbf{V}|0\rangle|^2 = \sum_{ij} c_i c_j^* \gamma_{ij} \tag{6.8}$$

with

$$\gamma_{ij} = \langle 0|\mathbf{U}_b^\dagger \mathbf{A}_i \mathbf{V}|0\rangle\langle 0|\mathbf{V}^\dagger \mathbf{A}_j^\dagger \mathbf{U}_b|0\rangle \tag{6.9}$$

We can achieve this either through a subroutine called the **Hadamard-Overlap Test** or a standard Hadamard Test, turning every operation into a controlled version of itself. Since

we will implement the latter circuit, we need to turn the ansatz $\mathbf{V}$, the A_l unitaries, and the operation $\mathbf{U}_b$ into the controlled version of themselves. Having built these circuits, we then proceed to compose two Hadamard tests, namely one for $\mathbf{U}_b^\dagger \mathbf{A}_i \mathbf{V}$ and one for $\mathbf{V}^\dagger \mathbf{A}_j^\dagger \mathbf{U}_b$.

```
def aux_hadamard_test_fixed_i(n, aux_index, CV, CA, CU, case):

    had_i = QuantumCircuit(n+1, 1)

    ## first hadamard
    had_i.h(aux_index)

    ## phase gate
    if case == 'imaginary':
        S_dagger = Operator([[1,0],[0,-1j]])
        had_i.unitary(S_dagger, aux_index, label='S^*')

    ## building
    had_i.compose(CV, inplace=True)
    had_i.compose(CA, inplace=True)
    had_i.compose(CU.inverse(), inplace=True)

    ## final hadamard
    had_i.h(aux_index)

    ## measure aux
    had_i.measure(aux_index, 0)

    return(had_i)
```

Listing 6.4: Auxiliary function implementing the Hadamard test for $\mathbf{U}_b^\dagger \mathbf{A}_i \mathbf{V}$.

```
def aux_hadamard_test_fixed_j(n, aux_index, CV, CA, CU, case):

    had_j = QuantumCircuit(n+1, 1)

    ## first hadamard
    had_j.h(aux_index)

    ## phase gate
    if case == 'imaginary':
        S_dagger = Operator([[1,0],[0,-1j]])
        had_j.unitary(S_dagger, aux_index, label='S^*')

    ## building
    had_j.compose(CU, inplace=True)
    had_j.compose(CA.inverse(), inplace=True)
    had_j.compose(CV.inverse(), inplace=True)

    ## final hadamard
    had_j.h(aux_index)

    ## measure aux
    had_j.measure(aux_index, 0)

    return(had_j)
```

Listing 6.5: Auxiliary function implementing the Hadamard test for $\mathbf{V}^\dagger \mathbf{A}_j^\dagger \mathbf{U}_b$.

We remark that to implement the conjugate transpose of one of the given operations, it suffices to compute its inverse since they are unitary. We automate this step using Qiskit's method `QuantumCircuit.inverse()`.

```
def hadamard_test_fixed_gamma(gates_ai,gates_aj,n,aux_index,parameters
    ,layers,case):

    ## Controlled-V
    CV = build_controlled_fixed_ansatz(n, layers,
                                       1, parameters)

    ## Controlled-Ub
    CUb = build_controlled_Ub_circuit(n)

    ## Controlled-A_i
    index_count = 1
    CAi = QuantumCircuit(n+1, 1)
    for pauli in gates_ai:
        if pauli == 'Z':
            CAi.cz(aux_index, index_count)
        elif pauli == 'X':
            CAi.cx(aux_index, index_count)
        index_count += 1

    ## Controlled-A_j^dagger
    index_count = 1
    CAj = QuantumCircuit(n+1, 1)
    for pauli in gates_aj:
        if pauli == 'Z':
            CAj.cz(aux_index, index_count)
        elif pauli == 'X':
            CAj.cx(aux_index, index_count)
        index_count += 1

    ## First hadamard test
    had_i = aux_hadamard_test_fixed_i(n, aux_index,
                                      CV, CAi, CUb,
                                      case)

    ## Second Hadamard test
    had_j = aux_hadamard_test_fixed_j(n, aux_index,
                                      CV, CAj, CUb,
                                      case)

    return(had_i, had_j)
```

Listing 6.6: Function implementing the Hadamard tests for γ_{ij}.

Hence, just as for $\langle\psi|\psi\rangle$, we define a function to compute $|\langle b|\psi\rangle|^2$ as the sum defined in Eq. 6.8 through the quasi-distributions obtained from running the Hadamard tests on a backend.

```
def get_b_dot_psi_global(n, layers, paulis, coeffs, parameters):

    dotprod = 0

    for i in range(len(paulis)):
        for j in range(len(paulis)):

            had_i, had_j = hadamard_test_fixed_gamma(paulis[i],
```

```
                                                    paulis[j],
                                                    n, 0, parameters,
                                                    layers, 'Real')

            num_shots = 100000
            backend = Aer.get_backend('aer_simulator',
                                      num_shots=num_shots)

            t_had_i = transpile(had_i, backend)
            t_had_j = transpile(had_j, backend)

            result_i = backend.run(t_had_i).result()
            result_j = backend.run(t_had_j).result()

            outputstate_i = result_i.get_counts()
            outputstate_j = result_j.get_counts()

            prob1_i = 0
            if '1' in outputstate_i.keys():
                prob1_i = float(outputstate_i['1'])/num_shots
            ph_i = 1-2*prob1_i

            prob1_j = 0
            if '1' in outputstate_j.keys():
                prob1_j = float(outputstate_j['1'])/num_shots
            ph_j = 1-2*prob1_j

            gamma_ij = ph_i*ph_j

            cc = coeffs[i]*coeffs[j]
            dotprod += cc*gamma_ij

    return(dotprod)
```

Listing 6.7: Function computing $|\langle b|\psi\rangle|^2$.

Finally, the global cost function can be defined as a Python function combining the components discussed in this subsection.

The main difference from previous code snippets is that the global cost function only takes *one* input, i.e., the parameters, and accesses the others as global variables. This is necessary as we will later pass this function to a SciPy optimizer that requires this kind of input management. Additionally, two more variables (i.e., `it` and `global_cost_plot`) are indicated as global to track the number of iterations in the optimizer run and plot the cost function's evolution.

```
def global_cost_function(parameters):

    ## variables
    global n, layers, paulis, coeffs, it, global_cost_plot

    parameters = separate_parameters(parameters, layers)

    ## psi_psi
    psi_psi = get_psi_dot_psi(n, layers, paulis,
                              coeffs, parameters)

    ## |b_psi|^2
    b_psi = get_b_dot_psi_global(n, layers, paulis,
                                 coeffs, parameters)

```

```
    ## estimate
    num = np.real(b_psi)
    den = np.real(psi_psi)
    estimate = 1 - float(num/den)

    print(f'Iteration: {it} \t Cost evaluation: {estimate}')
    global_cost_plot.append(estimate)

    it += 1

    return(estimate)
```

Listing 6.8: Global cost function for the VQLS.

This concludes the definition and evaluation of our global cost function.

Local Cost Function

Though very intuitive, the global cost function is more likely to exhibit barren plateaus issues, slowing the optimization routine. To try and help the optimization, we, therefore, define a local version of the cost function, both normalized and not, that still aims to estimate the orthogonal components of our prediction state $|\psi\rangle$:

$$\hat{C}_L = \langle x|\mathcal{H}_L|x\rangle \tag{6.10}$$

$$C_L = \frac{\hat{C}_L}{\langle\psi|\psi\rangle} \tag{6.11}$$

where $\mathcal{H}_L$ is the the effective Hamiltonian

$$\mathcal{H}_L = \mathbf{A}^\dagger \mathbf{U}_b \left(\mathbb{1} - \frac{1}{n} \sum_{k=1}^{n} |0_k\rangle\langle 0_k| \otimes \mathbb{1}_{\overline{k}} \right) \mathbf{U}_b^\dagger \mathbf{A} \tag{6.12}$$

where $|0_k\rangle$ is the zero state on qubit k and $\mathbb{1}_{\overline{k}}$ is the identity on all qubits except qubit k.

To evaluate this cost function, we use the Hadamard test subroutines once more, but we still need to pinpoint the correct operation $\mathbf{U}$ to use. With this goal in mind, let us expand our cost function. Given Eq. 6.12, we rewrite Eq. 6.10 as

$$\langle x|\mathcal{H}_L|x\rangle = \langle x|\mathbf{A}^\dagger \mathbf{U}_b \left(\mathbb{1} - \frac{1}{n} \sum_{k=1}^{n} \underbrace{|0_k\rangle\langle 0_k| \otimes \mathbb{1}_{\overline{k}}}_{:=\mathbf{P}_k} \right) \mathbf{U}_b^\dagger \mathbf{A}|x\rangle =$$

$$= \underbrace{\langle x|\mathbf{A}^\dagger}_{=\langle\psi|} \underbrace{\mathbf{U}_b \mathbf{U}_b^\dagger}_{=\mathbb{1}} \underbrace{\mathbf{A}|x\rangle}_{=|\psi\rangle} - \frac{1}{n} \sum_{k=1}^{n} \langle x|\mathbf{A}^\dagger \mathbf{U}_b \mathbf{P}_k \mathbf{U}_b^\dagger \mathbf{A}|x\rangle =$$

$$= \langle\psi|\psi\rangle - \frac{1}{n} \sum_{k=1}^{n} \langle\psi|\mathbf{U}_b \mathbf{P}_k \mathbf{U}_b^\dagger|\psi\rangle$$

Then Eq. 6.11 becomes

$$C_L = 1 - \frac{1}{n\langle\psi|\psi\rangle} \sum_{k=1}^{n} \langle\psi|\mathbf{U}_b \mathbf{P}_k \mathbf{U}_b^\dagger|\psi\rangle \tag{6.13}$$

We already know how to estimate $\langle\psi|\psi\rangle$; therefore, we only need to understand how to retrieve the addends. Since

$$|0_k\rangle\langle 0_k| = \frac{\mathbb{1}_k + \mathbf{Z}_k}{2}$$

we can express

$$\langle\psi|\mathbf{U}_b\mathbf{P}_k\mathbf{U}_b^\dagger|\psi\rangle = \langle\psi|\mathbf{U}_b\left(|0_k\rangle\langle 0_k| \otimes \mathbb{1}_{\overline{k}}\right)\mathbf{U}_b^\dagger|\psi\rangle =$$

$$= \langle\psi|\mathbf{U}_b\left(\frac{\mathbb{1}_k + \mathbf{Z}_k}{2} \otimes \mathbb{1}_{\overline{k}}\right)\mathbf{U}_b^\dagger|\psi\rangle =$$

$$= \frac{1}{2}\left(\langle\psi|\mathbf{U}_b\mathbf{U}_b^\dagger|\psi\rangle + \langle\psi|\mathbf{U}_b(\mathbf{Z}_k \otimes \mathbb{1}_{\overline{k}})\mathbf{U}_b^\dagger|\psi\rangle\right) =$$

$$= \frac{1}{2}\left(\langle\psi|\psi\rangle + \langle\psi|\mathbf{U}_b(\mathbf{Z}_k \otimes \mathbb{1}_{\overline{k}})\mathbf{U}_b^\dagger|\psi\rangle\right)$$

Going back to Eq. 6.13, we have

$$C_L = 1 - \frac{1}{2n\langle\psi|\psi\rangle}\left[\sum_{k=1}^{n}\langle\psi|\psi\rangle + \sum_{k=1}^{n}\langle\psi|\mathbf{U}_b(\mathbf{Z}_k \otimes \mathbb{1}_{\overline{k}})\mathbf{U}_b^\dagger|\psi\rangle\right] =$$

$$= \frac{1}{2} - \frac{1}{2n\langle\psi|\psi\rangle}\sum_{k=1}^{n}\langle\psi|\mathbf{U}_b(\mathbf{Z}_k \otimes \mathbb{1}_{\overline{k}})\mathbf{U}_b^\dagger|\psi\rangle =$$

$$= \frac{1}{2} - \frac{1}{2n\langle\psi|\psi\rangle}\sum_{k=1}^{n}\langle 0|\mathbf{V}^\dagger\mathbf{A}^\dagger\mathbf{U}_b(\mathbf{Z}_k \otimes \mathbb{1}_{\overline{k}})\mathbf{U}_b^\dagger\mathbf{A}\mathbf{V}|0\rangle =$$

$$= \frac{1}{2} - \frac{1}{2n\langle\psi|\psi\rangle}\sum_{k=1}^{n}\sum_{ij} c_i c_j^* \langle 0|\mathbf{V}^\dagger\mathbf{A}_j^\dagger\mathbf{U}_b(\mathbf{Z}_k \otimes \mathbb{1}_{\overline{k}})\mathbf{U}_b^\dagger\mathbf{A}_i\mathbf{V}|0\rangle$$

We can then apply a Hadamard Test to estimate the terms $\tilde{\delta}_{ij}^{(k)}$ for every $k = 1, \ldots, n$, such that

$$\tilde{\delta}_{ij}^{(k)} = \langle 0|\mathbf{V}^\dagger\mathbf{A}_j^\dagger\mathbf{U}_b(\mathbf{Z}_k \otimes \mathbb{1}_{\overline{k}})\mathbf{U}_b^\dagger\mathbf{A}_i\mathbf{V}|0\rangle \tag{6.14}$$

As a Python function, we can implement this Hadamard test by adding control only on the $\mathbf{A}_j^\dagger$,$\mathbf{A}_i$ and $\mathbf{CZ}_{0k} \equiv \mathbf{Z}_k \otimes \mathbb{1}_{\overline{k}}$ unitaries.

```
def hadamard_test_fixed_delta_k(k, gates_ai, gates_aj, n, aux_index,
    parameters, layers, case):

    qc = QuantumCircuit(n+1, 1)
    qc_qubits = [i for i in range(1, n+1)]

    ## first hadamard
    qc.h(aux_index)

    ## phase gate
    if case == 'imaginary':
        S_dagger = Operator([[1,0],[0,-1j]])
```

```
12          qc.unitary(S_dagger, aux_index, label='S^*')
13
14      ## ansatz (V)
15      qc.compose(build_fixed_ansatz(n, layers,
16                                    parameters),
17                 qubits=qc_qubits,inplace=True)
18
19      ## Controlled-A_i
20      index_count = 1
21      CAi = QuantumCircuit(n+1, 1)
22      for pauli in gates_ai:
23          if pauli == 'Z':
24              CAi.cz(aux_index, index_count)
25          elif pauli == 'X':
26              CAi.cx(aux_index, index_count)
27          index_count += 1
28      qc.compose(CAi, inplace=True)
29
30      ## Ub^dagger circuit
31      qc.compose(build_Ub_circuit(n).inverse(),
32                 qubits=qc_qubits, inplace=True)
33
34      ## Controlled-Z on qubit k
35      qc.cz(aux_index, k+1)
36
37      ## Ub circuit
38      qc.compose(build_Ub_circuit(n),
39                 qubits=qc_qubits, inplace=True)
40
41      ## Controlled-A_j^dagger
42      index_count = 1
43      CAj = QuantumCircuit(n+1, 1)
44      for pauli in gates_aj:
45          if pauli == 'Z':
46              CAj.cz(aux_index, index_count)
47          elif pauli == 'X':
48              CAj.cx(aux_index, index_count)
49          index_count += 1
50      qc.compose(CAj.inverse(), inplace=True)
51
52      ## final hadamard
53      qc.h(aux_index)
54
55      ## measure aux
56      qc.measure(aux_index, 0)
57
58      return(qc)
```

Listing 6.9: Function implementing the Hadamard test for δ_{ij}^k.

The complete local cost function is shown in the following listing.

```
1  def local_cost_function(parameters):
2
3      ## variables
4      global n, layers, paulis, coeffs, local_cost_plot, it
5
6      parameters = separate_parameters(parameters, layers)
7
8      ## psi_psi
9      psi_psi = get_psi_dot_psi(n, layers, paulis,
```

```
                                coeffs, parameters)

    ## local_b_psi
    local_b_psi = 0
    for k in range(n):
        ccdelta_k = get_b_dot_psi_local(k, n, layers,
                                        paulis, coeffs,
                                        parameters)
        local_b_psi += ccdelta_k

    ## estimate
    num = np.real(local_b_psi)
    den = np.real(2*n*psi_psi)
    estimate = .5 - float(num/den)

    ## saving cost for plotting
    local_cost_plot.append(estimate)

    ## printing
    print(f'Iteration {it} \t Cost evaluation: {estimate}')

    it += 1

    return(estimate)
```

Listing 6.10: Local cost function for the VQLS.

This concludes the evaluation of the local cost function.

6.1.3 Optimizer

To fulfill the algorithm definition, we must eventually choose an **optimizer**, i.e., a method that will determine how to update the set of parameters until convergence. Since we are coding in Qiskit, the function that will take care of the optimization process is SciPy's `minimize`, so our possibilities are restricted to the supported methods [The].

To cover different approaches, we will run our experiments with three different methods:

- **Conjugate Gradient (CG):** a gradient-based method designed to solve linear systems of equations where the associated matrix $\mathbf{A}$ is real, positive definite, and symmetric. It relies on the orthogonality of the residuals and the conjugacy[10] of the search directions [HS+52].
- **COBYLA:** a gradient-free method proposed for nonlinearly constrained optimization calculations that has progressively found its way into a larger field of applications. Each iteration forms linear approximations of the objective and constraint functions by interpolation at the vertices of a simplex[11], and a trust region bound restricts each change to the variables [Pow94].
- **Powell:** another gradient-free method designed for finding the minimum of a multivariable function by changing one parameter through conjugate directions [Pow64].

We want to make only one final remark before moving on to the experimental results: our optimization process's **starting point**. Unless some a priori information is known

[10] Two non-zero vectors $\mathbf{u}, \mathbf{v}$ are **conjugate** with respect to a matrix $\mathbf{A}$ if $\mathbf{u}^T\mathbf{A}\mathbf{v} = 0$.
[11] A **simplex** is the simplest possible polytope in any given dimension.

about the optimal set of parameters or the solution, no general rule will yield certain improvements to the optimization process. Since we are assuming no prior knowledge about the system, in the simulation stage, we try four different initialization techniques to compare[12], as reported in Table 6.1. Once we know the best strategy, we will set it as the default one.

	Zero Vector	**Uniform Vector**	**Normal Distribution**	**Random Vector**
$\boldsymbol{\theta}^{(0)}$	$\begin{pmatrix} 0 \\ \vdots \\ 0 \end{pmatrix}$	$\begin{pmatrix} 1/N \\ \vdots \\ 1/N \end{pmatrix}$	Obtained through `np.random.normal(0,1)`	Obtained through `np.random.rand()`

Table 6.1: Different initialization techniques used to define $\boldsymbol{\theta}^{(0)}$.

6.2 Experimental results

Let us define the following system of equations

$$\mathbf{A} = 0.55\mathbb{1} + 0.225\mathbf{Z}_2 + 0.225\mathbf{Z}_1 \qquad |b\rangle = \mathbf{H}^{\otimes 3} |0\rangle \tag{6.15}$$

Once the VQLS algorithm's optimization routine ends, we build one final ansatz to create the solution state $|x_{opt}\rangle$. We measure the accuracy ζ of the algorithm by taking the squared norm of $|b\rangle$ and the normalized version of $|\psi_{opt}\rangle = \mathbf{A}\,|x_{opt}\rangle$:

$$0 \leq \zeta \coloneqq |\langle b|\Psi_{opt}\rangle|^2 \leq 1 \tag{6.16}$$

where we used the same notation as in Eq.6.4.

6.2.1 Simulation of an ideal quantum computer

We start our simulated runs with the **conjugate gradient** method to see if the algorithm can exploit the landscape topography.

	Zero Vector	**Uniform Vector**	**Normal Distribution**	**Random Vector**
ζ_{global}	0.1250068	0.2405453	0.0414796	0.0138311
ζ_{local}	0.1249992	0.2405436	0.0414814	0.0138308

Table 6.2: Accuracy results of the VQSL run with the CG method.

Unfortunately, as we can see from the results reported in Table 6.2, the algorithm fails to find a vector proportional to $|b\rangle$. This is due to a precision loss in the optimization subroutine, so enlisting a pre-defined gradient-based method does not seem to be a viable strategy[13]. The optimization routine is displayed in Fig. 6.2 and in Fig. 6.3. The number of iterations for our optimizer is manually bound to 200 to plot the cost evolution, considering that the precision loss does not allow for convergence.

[12] Out of all the possible initialization strategies, these four were suggested in an e-mail exchange by Dr. Carlos Bravo-Prieto, one of the authors of [Bra+23].

[13] It is possible to employ a gradient-based optimization routine, but it must be tailored to the VQLS algorithm. Further information in [Bra+23].

Since the previous results highlight the training difficulty through a gradient-based method, we substitute the CG method with the gradient-free **COBYLA**, one of the optimizers suggested by the IBM documentation when implementing variational algorithms with no specific constraints or conditions to be met.

	Zero Vector	**Uniform Vector**	**Normal Distribution**	**Random Vector**
ζ_{global}	0.7486162	0.7765784	0.7145117	0.7466224
ζ_{local}	0.78634465	0.7876895	0.8834687	0.9157983

Table 6.3: Accuracy results of the VQSL run with COBYLA.

As we can see from Table 6.3, changing the optimizer greatly increases the accuracy of the estimation $|\psi_{opt}\rangle$, reaching 91% accuracy with a random initialization. As for the number of iterations to converge, the COBYLA optimizer finds a minimum in around 100 function evaluations, regardless of the locality of the cost function. This estimate is obtained by averaging over all the experiments run in these past months, and it is consistent with the cost evolution pictured in Fig. 6.4 and Fig. 6.5.

One final test is to try the optimizer used in [Bra+23], i.e., the **Powell** method[14], which we expect to top the performances of CG. So let's see if, for the problem defined in Eq. 6.1, it is better to use the COBYLA optimizer or the Powell one.

	Zero Vector	**Uniform Vector**	**Normal Distribution**	**Random Vector**
ζ_{global}	0.7300776	0.7290047	0.7145397	0.7502186
ζ_{local}	0.6307183	0.4820012	0.7673284	0.8091483

Table 6.4: Accuracy results of the VQSL run with the Powell method.

From the results displayed in Table 6.4, we see that the Powell method does perform better than CG but not well enough to compete with the COBYLA optimizer. Furthermore, the convergence of this method requires an average of 560 iterations with a cost evolution studded with heavy oscillations, as the ones pictured in Fig. 6.6 and Fig. 6.7.

A final remark on the result shown in this subsection is due. Given the probabilistic nature of measurements and the unpredictability of the optimization process, we ran every experiment multiple times to get a clearer picture of the overall scenario. The numerical results presented in this section were either averaged over the number of experiments (e.g., the number of iterations to convergence) or, quite simply, the best result obtained (e.g., the accuracy levels).

[14] They did use the Powell method in most of their experiments, but not on the one defined in Eq. 6.1. So we do not know a priori how this optimizer may perform in our experiments.

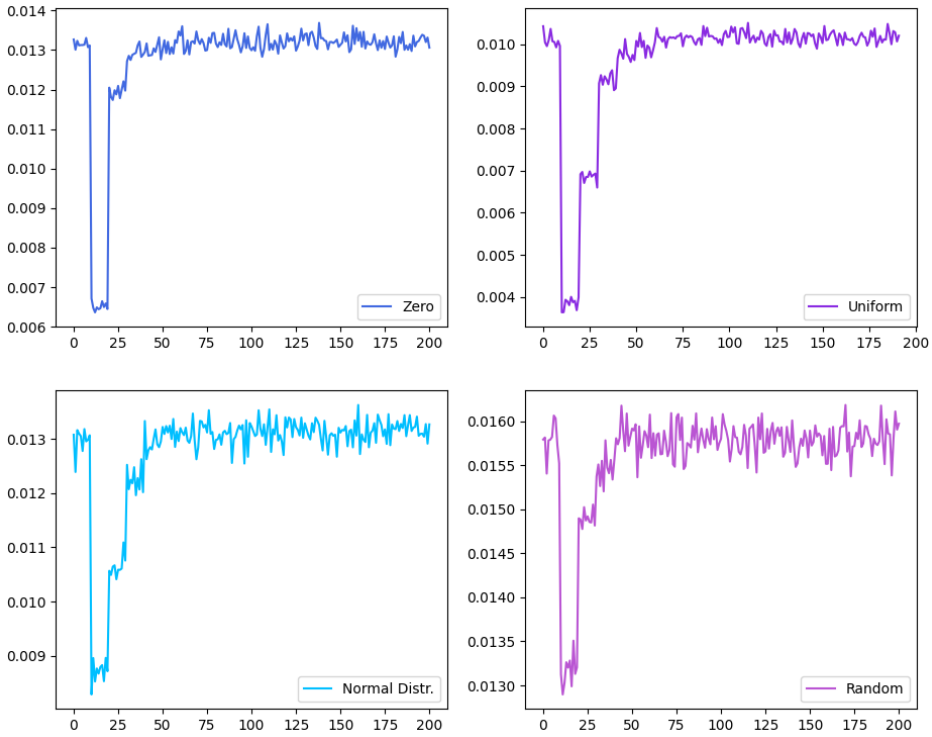


Fig. 6.2: Global cost evolution to the number of iterations for every initialization technique. The optimization process was run with the CG optimizer.

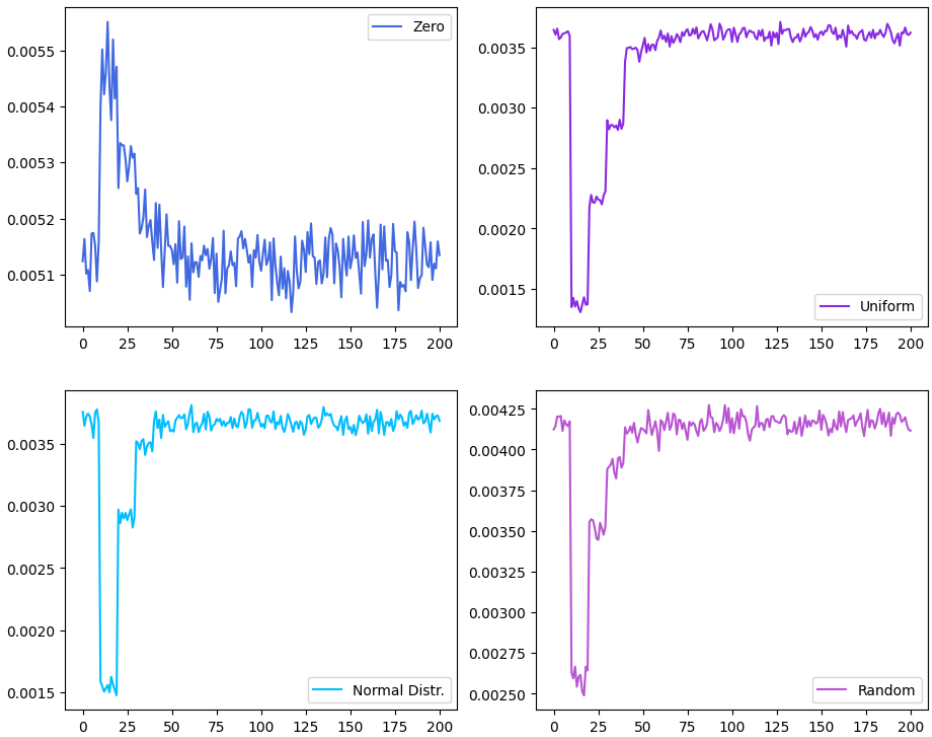


Fig. 6.3: Local cost evolution to the number of iterations for every initialization technique. The optimization process was run with the CG optimizer.

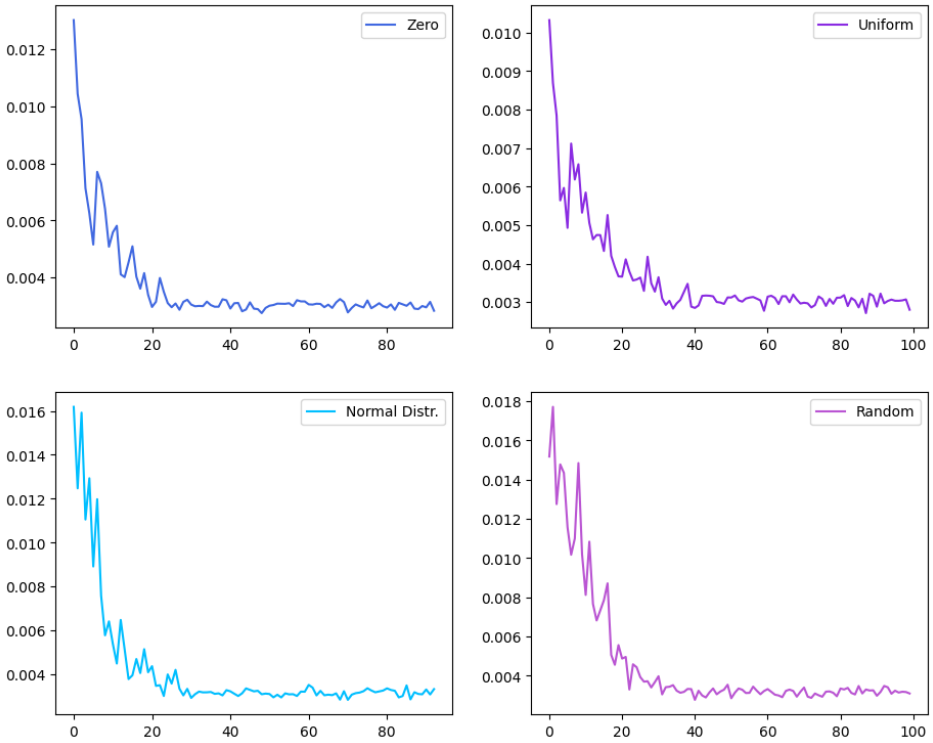


Fig. 6.4: Global cost evolution to the number of iterations for every initialization technique. The optimization process was run with the COBYLA optimizer.

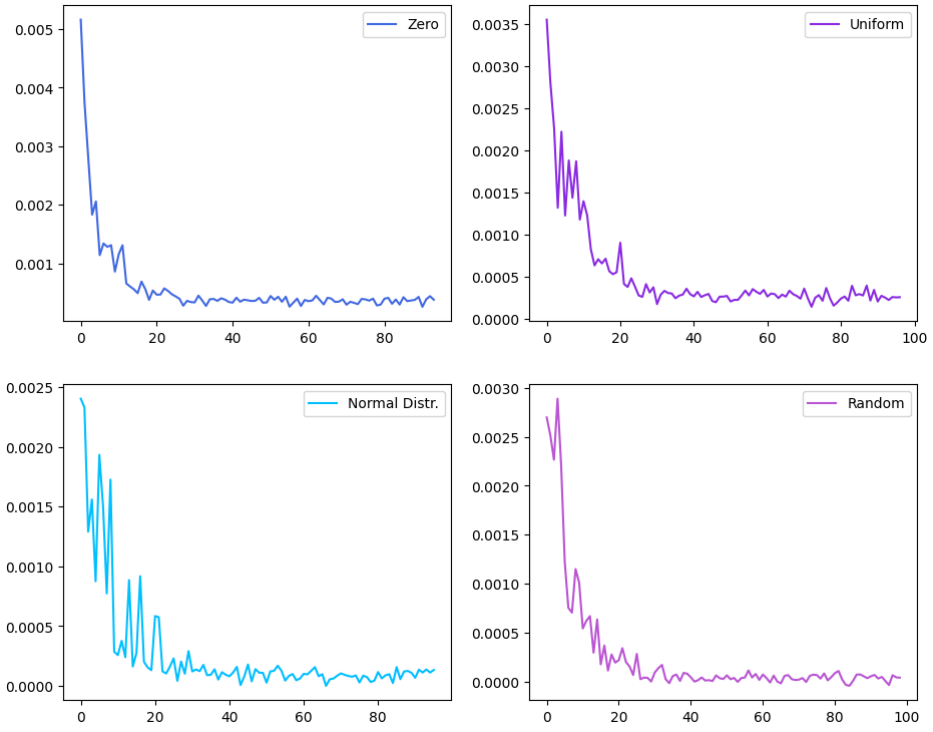


Fig. 6.5: Local cost evolution to the number of iterations for every initialization technique. The optimization process was run with the COBYLA optimizer.

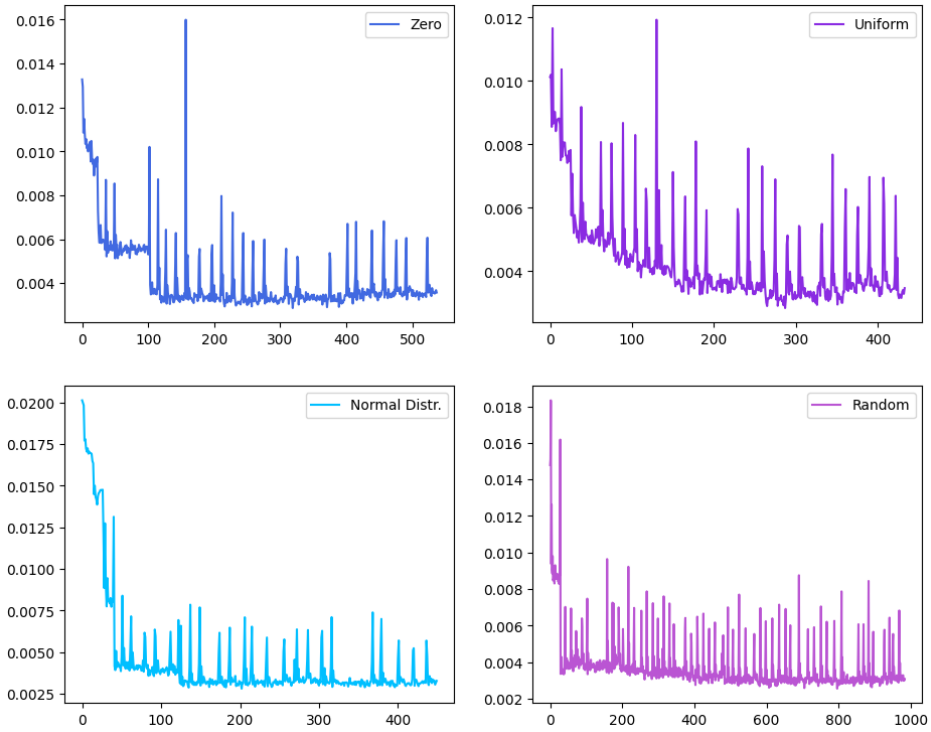


Fig. 6.6: Global cost evolution to the number of iterations for every initialization technique. The optimization process was run with the Powell optimizer.

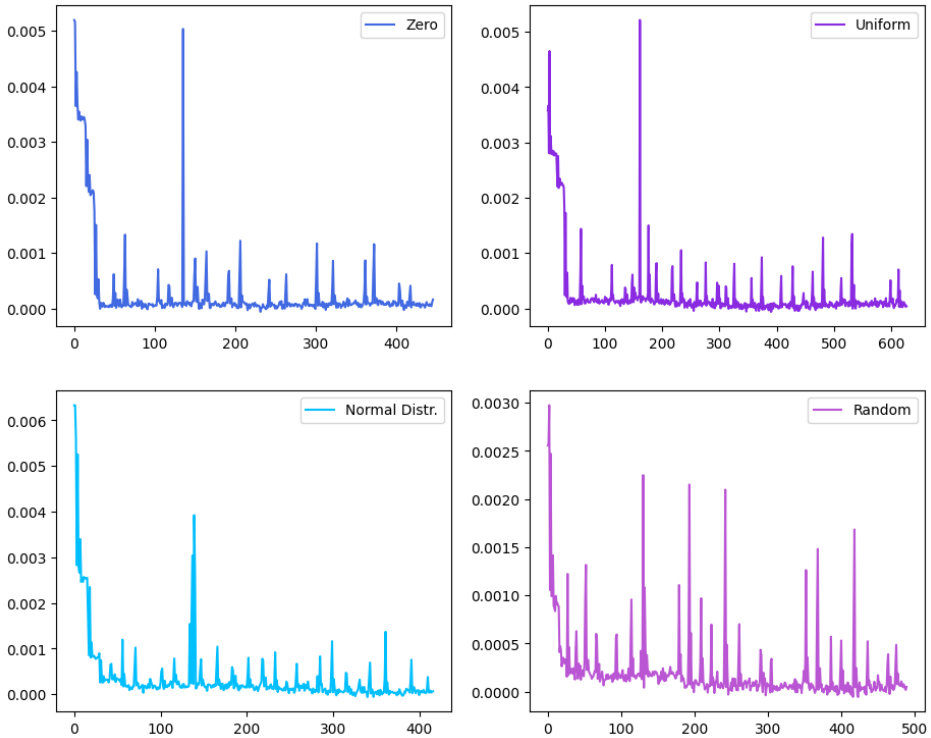


Fig. 6.7: Local cost evolution to the number of iterations for every initialization technique. The optimization process was run with the Powell optimizer

6.2.2 Running on QPU and noise simulation

Since quantum resources must not be used carelessly, we consistently set up the VQLS algorithm with the information gathered in Subsection 6.1. Hence, the optimization process will be handled by the COBYLA optimizer, and it will start in a randomly generated initial point $\boldsymbol{\theta}^{(0)}$.

To use one of the **QPUs** made remote-accessible by IBM, one must create an account on their platform https://quantum.ibm.com/. The free plan includes complete access to their documentation and their Quantum Lab, a cloud platform one can use to code without installing anything locally[15]. Furthermore, the free plan allows users to use three of their QPUs for a total of ten minutes per month.

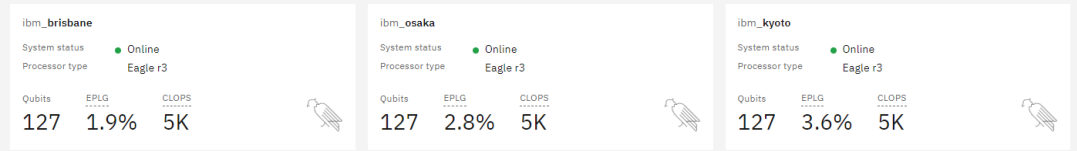


Fig. 6.8: QPUs available with a free plan on the IBM Quantum platform. Alongside the system name, it is also possible to access the information (e.g., error rate, status, number of queued jobs, etc) of a single QPU by clicking on it when on the platform. This screenshot was taken on February 19, 2024, at 12:38 a.m.

Having selected a QPU from the list in Fig. 6.8, we slightly modify our Jupyter Notebook by adding a code cell to redirect the computation to the quantum device.

```
1  from qiskit_ibm_runtime import QiskitRuntimeService, Options
2  from qiskit_aer.noise import NoiseModel
3  from qiskit.providers.fake_provider import FakeWashingtonV2
4
5  service = QiskitRuntimeService()
6
7  qpu = True
8
9  if qpu:
10     backend = service.backend('ibm_brisbane')        ## IBM QPU
11 else:
12     backend = FakeWashingtonV2()                     ## Fake backend
13     noise_model = NoiseModel.from_backend(backend)   ## + noise
14
15 options = Options()
16 options.simulator.set_backend(backend)
17 options.optimization_level = 3
18 options.resilience_level = 2
```

Listing 6.11: Code cell needed to run on QPU.

Lines [10,16] are the specific commands that set the QPU as the default device to run the computation. Lines $[15-18]$ determine the options defined through the homonym class, characterizing the computation on the QPU. The value `resilience_level` configures the level of noise mitigation we want the transpiler to apply to our circuits, with 0 being no mitigation and 3 being the heaviest mitigation. The `if_else` block in lines $[9-13]$ switches between the real QPU `ibm_brisbane` and the fake backend `FakeWashingtonV2` equipped with a noise model. Since the real device has 127 qubits, we chose the fake

[15] We did not use it for this thesis, but we thought it was worth mentioning the existence in this tool given its versatility and easy access.

backend to have the same number of qubits. Additionally, the noise model allows us to run a local simulation of a noisy computation by exchanging the boolean variable `qpu` as `False` in line [7].

One may wonder why such a model would be needed when a real QPU is available. To answer this question, let us estimate the QPU time we would need to run our implementation of VQLS on a quantum device. To have a basis to start from, we executed a Hadamard test of each kind and checked their execution time on the IBM Quantum platform: the Hadamard test defined in Listing 6.2 takes $\sim 3s$, the ones in Listings 6.4 and 6.5 take $\sim 4s$ each, and, finally, the one in Listing 6.9 takes $\sim 3s$ too. To estimate the term $\langle\psi|\psi\rangle$ we need L^2 Hadamard tests, one for each β_{ij} (see Eq. 6.5 and Eq. 6.6). For $|\langle b|\psi\rangle|^2$, we have to execute two different Hadamard tests for each γ_{ij}, which again are L^2 (see Eq. 6.8 and Eq. 6.9) . Finally, to compute the term in Eq. 6.14, we need once again L^2 Hadamard tests, but these have to be executed for each $k = 1, \ldots, L$. Since we do not use one of Qiskit's primitives, calculating the quasi-distributions of the outputs takes an approximate time of `num_shots` $= 4\,000$ times the execution time of the associated Hadamard test. These estimates are reported in Table 6.5.

Circuit	**Execution time of one circuit**	**Number of equivalent circuits to run**	**Total execution time per term**
Hadamard test for β_{ij}	$3s$	$L^2 = 9$	$4\,000 \cdot 9 \cdot 3s = 108\,000s$
Hadamard subtest for γ_{ij}	$4s$	$2L^2 = 18$	$4\,000 \cdot 18 \cdot 4s = 288\,000s$
Hadamard test for $\delta_{ij}^{(k)}$	$3s$	$L \cdot L^2 = 27$	$4\,000 \cdot 27 \cdot 3s = 324\,000s$

Table 6.5: Calculation to estimate the QPU usage for the VQLS algorithm.

If we decide to use the global cost function, we have an estimated overall QPU usage of $396\,000s$ or 110 hours. Similarly, if we implement the local cost function, the overall QPU usage is approximately 120 hours. We could reduce the `num_shots` variable, but it would hinder the precision of the quasi-distributions, making our efforts fruitless.

Since we cannot use a real QPU, we rely on the fake backend to simulate the noise effect on our computation and evaluate its impact on the output. We do not report other code snippets since the only difference is in the backend definition.

As shown in Fig. 6.9, noise does not affect the convergence speed, even though the trend is more irregular than the noiseless computation. As for accuracy, the level is not optimal, but noise does not prevent to obtain a good approximation:

$$\zeta_{global} = 0.6392252 \qquad \zeta_{local} = 0.8224284$$

If we go back to Table 6.3, we see that these approximations are not so far from the general accuracy trend, so playing with noise mitigation techniques and more tailoring of the circuits to the QPU might make our algorithm take the final leap to optimality.

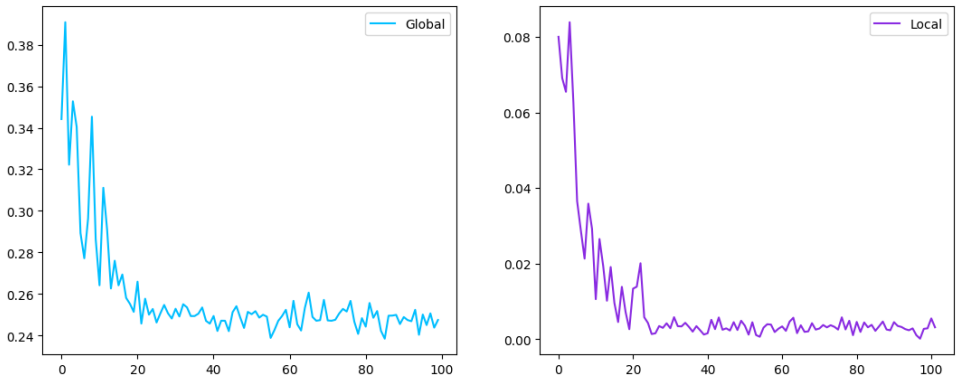


Fig. 6.9: Global and local cost evolution of the noisy run on `FakeWashingtonV2`. The optimization process starts in a randomly generated point and evolves through the COBYLA optimizer.

Chapter 7

Experiments on the Gate Model: QAOA

We start this chapter by discussing how to **encode** a MQ problem into a Hamiltonian and how to reduce its qubits' interactions, reformulating our problem as a Binary Quadratic Model (BQM) (Section 7.1). The **code** and the **results** of the performed experiments close this chapter and the discussion of the gate-based experiments (Section 7.2).

The code in this chapter can be found as Jupyter Notebooks in my public repository on **GitHub** [Gal]. Additional information about the software and the hardware used can be found in Appendix: Used Hardware and Software Versions.

7.1 Encoding the MQ problem into a BQM: Direct Encoding

We remark that the input to the MQ problem consists of m quadratic polynomials $p_1(\mathbf{x}), \ldots, p_m(\mathbf{x}) \in \mathbb{F}_q[\mathbf{x} = (x_1, \ldots, x_n)]$ in n variables $x_1, \ldots, x_n$ and coefficients in a finite field $\mathbb{F}_q$. The output of the problem is a vector $\mathbf{s} = (s_1, \ldots, s_n) \in \mathbb{F}_q^n$ for which $p_i(\mathbf{s}) = 0$ for every $1 \leq i \leq m$. We will tackle the Boolean variant of this problem, i.e., where $q = 2$. Furthermore, a Boolean polynomial is generally stated in its ANF but can be uniquely converted from ANF to NNF. However, this procedure causes an exponential increase in the number of terms, most of which have degree greater than two and cannot be simulated by current hardware. Hence, we need to devise a strategy to encode our problem efficiently in a two-body Hamiltonian.

A **direct approach** to encode our problem in a Hamiltonian is penalizing with positive energy each of the equations $p_i(\mathbf{x})$ that is not satisfied. The corresponding problem Hamiltonian can then be defined as

$$\mathcal{H}_P = \sum_{i=1}^{m} p_i(\mathbf{x}) \tag{7.1}$$

as it contributes with positive energy if the input bits for $p_i(\mathbf{x})$ do not result in a zero solution.

Typically, the polynomials $p_i(\mathbf{x})$ in Eq. 7.1 are given in ANF since bitwise operations are performed over $\mathbb{F}_2$. However, the quantum Hamiltonian we can encode in a quantum

device does not work with binary algebra, so each positive term only adds more energy to the final state. Therefore, each polynomial $p_i(\mathbf{x})$ has to be given in its NNF. This transformation can be obtained by recursively applying the following change[1]

$$(x_i + x_j) \mapsto x_i + x_j - 2x_i x_j$$

to the original ANF equations. As prospected, this transformation introduces multi-qubit interaction terms of degree greater than 2 that were not present in the original ANF.

For a general many-body Hamiltonian, its interactions can be reduced to two-body using perturbation theory by adding ancilla qubits [SC95]. However, suppose all the parts of the problem share the same basis. In that case, as is the case for a classical Hamiltonian such as ours, the reduction can be performed without perturbation theory [BOA13] [Bia08], and it is the strategy we implement. The resulting Hamiltonian has a different energy spectrum but equal ground state and energy, leaving the problem's solution unaltered.

This latter method consists of exchanging a two-qubit interaction by an ancilla, reducing the order of the interaction by one. A penalty function is then introduced to the Hamiltonian that adds energy when the value of the ancilla is not equal to the product of the original two qubits:

$$g(x_i, x_j, x_{ij}) = 3x_{ij} + x_i x_j - 2x_i x_{ij} - 2x_j x_{ij} \tag{7.2}$$

where x_{ij} is the label given to the ancillary qubit that is substituted. It is easy to see that $g(x_i, x_j, x_{ij}) = 0$ if $x_i x_j = a_{ij}$ and $g(x_i, x_j, x_{ij}) \geq 1$ otherwise. This keeps the ground state and energy unchanged. Furthermore, the same ancilla x_{ij} can be used for all the terms in the Hamiltonian where the term $x_i x_j$ appears by applying the substitution

$$\sum_K \alpha_{ijK} x_i x_j x_K \mapsto \sum_K \alpha_{ijK} x_{ij} x_K + (1 + \delta_{ij}) g(x_i, x_j, x_{ij})$$

where the index K is the product of multiple other variables in all terms where $x_i x_j$ is present and

$$\delta_{ij} \coloneqq \max\left(\sum_{K, \alpha_{ijK} > 0} \alpha_{ijK}, \sum_{K, \alpha_{ijK} < 0} -\alpha_{ijK} \right)$$

Using this transformation, we can build a **QUBO model** from the two-body Hamiltonian by defining a matrix $\mathbf{Q}$ using the terms' coefficients.

Since the number of qubits available in quantum devices is limited, it is important to estimate the required quantum resources. If a given n-qubit Hamiltonian has up to n-body terms, one would need $2^{\frac{n+2}{2}} - 2$ total qubits to reduce all possible combinations of qubit interactions to two-body terms for an even n (and $3 \cdot 2^{\frac{n-1}{2}} - 2$ for odd n) [Ram+22]. This is the case for a general conversion from ANF to NNF since an n term sum in ANF will generally require

$$\sum_{k=1}^{n} \binom{n}{k} = 2^n - 1$$

terms for the equivalent NNF polynomial. Therefore, an exponential amount of quantum resources is needed to encode the ground state into a two-body Hamiltonian following this direct embedding[2].

[1] With a slight abuse of notation, we write $+$ for both the addition over $\mathbb{F}_2$ or over another field or ring.
[2] This estimate refers to logical qubits; hence, the number of physical qubits is even higher.

7.2 Experimental results

The problem we solve in this section is the MQ problem with five variables[3] defined in Listing 7.1. The associated solution is known, but it is used solely for validation purposes[4]. We refer to this specific instance as MQ5 throughout the section.

```
4*x1*x2*x3*x4*x5 - 2*x1*x2*x3*x4 - 4*x1*x2*x3*x5 +
    2*x1*x2*x3 + 2*x1*x2*x5 - x1*x2 - 2*x1*x3*x4*x5 +
    2*x1*x3*x4 + 2*x1*x3*x5 - x1*x3 - x1*x5 - 2*x2*x3*x4*x5 +
    2*x2*x3*x5 - x2*x3 + x2*x4 - x2*x5 + x4*x5 - x4 + 1 = 0,

-2*x1*x2*x3*x4 + 2*x1*x2*x3*x5 + 2*x1*x2*x4*x5 - 2*x1*x2*x5 +
    2*x1*x3*x4 - x1*x3 - x1*x4 + x1 - 2*x2*x3*x5 + x2*x3 +
    x2*x5 - x3*x4 + x3*x5 - x4*x5 + x4 = 0,

-4*x1*x2*x3*x4*x5 + 2*x1*x2*x3*x4 + 2*x1*x2*x3*x5 -
    2*x1*x2*x3 + 2*x1*x2*x4*x5 - 2*x1*x2*x5 + x1*x2 +
    2*x1*x3*x4*x5 - 2*x1*x3*x5 + x1*x3 - x1*x4 + x1*x5 +
    x2*x3 - x2*x4 - x3*x4 + x4 = 0,

4*x1*x2*x3*x4*x5 - 2*x1*x2*x3*x5 - 2*x1*x2*x4*x5 - 2*x1*x3*x4
    + x1*x3 + x1*x4 - 2*x2*x3*x4*x5 + x2*x3 + x2*x4 + x2*x5
    - x2 + x3*x4 + x3*x5 - x3 + x4*x5 - x4 - x5 + 1 = 0,

4*x1*x2*x3*x4*x5 - 4*x1*x2*x3*x4 - 4*x1*x2*x3*x5 +
    4*x1*x2*x3 - 2*x1*x2*x4*x5 + 2*x1*x2*x4 + 2*x1*x2*x5 -
    2*x1*x2 - 2*x1*x3*x4*x5 + 2*x1*x3*x4 + 2*x1*x3*x5 -
    2*x1*x3 + 2*x1*x4*x5 - x1*x4 - x1*x5 + x1 - 2*x2*x3*x4*x5
    + 2*x2*x3*x4 + 2*x2*x3*x5 - 2*x2*x3 + 2*x2*x4*x5 - x2*x4 -
    2*x2*x5 + x2 - x3*x5 + x3 - x4*x5 + x5 = 0
```

Listing 7.1: Definition of a MQ problem with 5 variables.

To encode the MQ5 problem onto a QUBO model, we refer to the code available online on the article-associated GitHub repository [Ram] and proceed to discuss the newly-written code of these experiments.

7.2.1 Simulation of an ideal quantum computer

After the embedding, we obtain a matrix $\mathbf{Q}$ of dimension 10×10 (five original variables and five ancillae) and an offset that define our problem-associated QUBO. To implement the QAOA in Qiskit, we need to turn it into a `QuadraticProgram` Qiskit's object. After a few list manipulations to divide the linear[5] terms from the quadratic in $\mathbf{Q}$, we define the cost function to minimize through the method `minimize` of the `QuadraticProgram` class, obtaining the following output:

[3] Higher dimensions get prohibitive quite fast due to the required storage. For instance, the other core problem is the MQ9, which we could run neither locally due to memory limitations (it required 512. TiB) nor through other devices (like the university server) due to limited QPU access.

[4] For more complex instances, using such information is always recommended, but we decided to assume zero a priori knowledge to run all our experiments since this is the most interesting of the application cases.

[5] We have $x^2 = x$ since the problem is defined for Boolean variables.

```
Problem name: QUBO model from Article

Minimize
  8*a0*a2 - 6*a0*a3 - 6*a0*a4 - 18*a1*a2 - 30*x0*a0 + 2*x0*a1
  - 2*x0*a2 + 4*x0*a3 + 2*x0*a4 + 15*x0*x1 - 2*x0*x2 - 2*x0*x3
  - x0*x4 - 30*x1*a0 + 2*x1*a1 - 6*x1*a2 + 2*x1*a3 + 2*x1*a4
  - x1*x4 + 4*x2*a0 + 9*x2*a1 - 18*x2*a2 - 16*x2*a3 - 14*x2*a4
  + 8*x2*x3 + 7*x2*x4 + 2*x3*a0 - 26*x3*a1 - 16*x3*a3 + 13*x3*x4
  - 26*x4*a1 - 14*x4*a4 + 43*a0 + 39*a1 + 27*a2 + 23*a3 + 22*a4
  + 2*x0 + 2

Subject to
  No constraints

  Binary variables (10)
    x0 x1 x2 x3 x4 a0 a1 a2 a3 a4
```

Listing 7.2: QUBO model associated to the MQ5 as Qiskit `QuadraticProgram`.

Since the QAOA algorithm works only on spin variables, we turn the QUBO into an Ising model using the method `to_ising()`, which decomposes our cost function into a linear combination of Pauli operators:

```
offset: 41.5
operator:
SparsePauliOp(['IIIIIIIIIZ', 'IIIIZIIIII', 'IIIZIIIIII',
        'IIZIIIIIII', 'IZIIIIIIII', 'ZIIIIIIIII', 'IIIIIIIIZZ',
        'IIIIIIIIZI', 'IIIIIIIZIZ', 'IIIIIIIZII', 'IIIIIIZIIZ',
        'IIIIIIZIII', 'IIIIIIZZII', 'IIIIIZIIIZ', 'IIIIIZIIII',
        'IIIIIZIIZI', 'IIIIIZIZII', 'IIIIIZZIII', 'IIIIZIIIIZ',
        'IIIIZIIIZI', 'IIIIZIIZII', 'IIIIZIZIII', 'IIIZIIIIIZ',
        'IIIZIIIIZI', 'IIIZIIIZII', 'IIIZIIZIII', 'IIIZIZIIII',
        'IIZIIIIIIZ', 'IIZIIIIIZI', 'IIZIIIIZII', 'IIZIZIIIII',
        'IIZZIIIIII', 'IZIIIIIIIZ', 'IZIIIIIIZI', 'IZIIIIIZII',
        'IZIIIIZIII', 'IZIIZIIIII', 'ZIIIIIIIIZ', 'ZIIIIIIIZI',
        'ZIIIIIIZII', 'ZIIIIZIIII', 'ZIIIZIIIII'],
coeffs=[ 2.5 +0.j, -7.  +0.j, -5.25+0.j, -4.5 +0.j, -3.5 +0.j,
        -3.5 +0.j, 3.75+0.j,  4.  +0.j, -0.5 +0.j,  5.5 +0.j,
        -0.5 +0.j,  5.25+0.j, 2.  +0.j, -0.25+0.j,  5.5 +0.j,
        -0.25+0.j,  1.75+0.j,  3.25+0.j, -7.5 +0.j, -7.5 +0.j,
        1.  +0.j,  0.5 +0.j,  0.5 +0.j,  0.5 +0.j, 2.25+0.j,
        -6.5 +0.j, -6.5 +0.j, -0.5 +0.j, -1.5 +0.j, -4.5 +0.j,
        2.  +0.j, -4.5 +0.j,  1.  +0.j,  0.5 +0.j, -4.  +0.j,
        -4.  +0.j, -1.5 +0.j,  0.5 +0.j,  0.5 +0.j, -3.5 +0.j,
        -3.5 +0.j, -1.5 +0.j])
```

Listing 7.3: Ising model associated to the MQ5.

Since QAOA is a variational algorithm, it has the same workflow as the VQLS algorithm: after defining the ansatz, which in this case is necessarily the QAOA ansatz, a random vector of parameters is initialized, and the optimization loop starts. Since we are using the QAOA ansatz with $p = 1$, our circuit will only have two parameters, i.e., β and γ. To run the optimization routine, we rely once again on the **COBYLA** optimizer and Qiskit's method `compute_minimum_eigenvalue()`:

```
1  ## hyperparameters
2  p = 1
3  initial_point = [np.random.rand() for _ in range(2*p)]
4
```

```
    callback_data = []       ## to save parameters evolution
    cost = []                ## to save cost evolution

    ## QAOA class
    qaoa_mes = QAOA(sampler=Sampler(), optimizer=COBYLA(), reps=p,
                    initial_point=initial_point, callback=my_callback)

    ## minimizing
    qaoa_result = qaoa_mes.compute_minimum_eigenvalue(ising_model)
```

Listing 7.4: QAOA run using Qiskit.

The `qaoa_result` object holds all the information about the final eigenstate and the overall optimization process. In contrast, for a more detailed evolution of the cost or of the parameter, the custom callback function `my_callback` was defined.

Unlike the VQLS algorithm, in this case, we are only interested in the bit string representing our solution. This string can be retrieved from the `qaoa_result` attribute `best_measurement`, which also contains the associated eigenvalue, the cost evaluation, and the probability to measure such an output when the system is in this final eigenstate.

```
Best measurement:{'state': 51,
                  'bitstring': '0000110011',
                  'value': (-41.5+0j),
                  'probability': 0.0132117990543206}
```

Listing 7.5: QAOA results for MQ5 in a noiseless simulation.

It is of the utmost importance in a case like ours to remember that Qiskit works in little-endian convention. Hence, the output string must be reversed to have the values associated with the original variables in the first 5 positions[6]. Thus, the solution proposed by our algorithm is 11001, precisely the known solution of the MQ5 problem!

Since our algorithm works for MQ5, let us discuss the computation quality starting from the probability of measuring the output. From Listing 7.5, we know that this probability is approximately $1.3212 \cdot 10^{-2}$, which does not seem that high at first glance. To put things in perspective, consider the uniform superposition of all basis states $|S\rangle$ in the problem-associated Hilbert space $\mathbb{H}$, that is, the superposition in which every measurement result has the same probability of being the output:

$$|S\rangle = \frac{1}{\sqrt{2^n}} \sum_{k=1}^{2^n} |k\rangle$$

Since our system has $n = 10$ qubits between ancillae and original variables, $\mathbb{H}$ has a dimension of $2^{10} = 1024$. Hence, if we were to measure $|S\rangle$, we would get any eigenstate with probability

$$\tilde{p} \coloneqq p_j = \left| \frac{1}{\sqrt{2^{10}}} \right|^2 \approx 9.7656 \cdot 10^{-4}$$

In Fig. 7.1, we report how probabilities are distributed among eigenstates for two test runs for reference. Specifically, the second[7] test run is the one whose results are reported

[6] Trivially, one could leave the string in the given order and count backward, too, but we preferred this convention.

[7] The terminology "*first test run*" and "*second test run*" does not refer to a temporal order since numerous tests have been computed but not reported. When comparing the results, these terms are only used to differentiate between the runs.

in Listing 7.5. As we can see[8], over 60% of the outputs have a probability lower than $\tilde{p}$, leaving only about a third of states to test as a possible solution to our problem. For a more detailed report on the numbers in these pie charts, see Table 7.1.

Test run	Number of eigenstates with $p < 0.0009$	Number of eigenstates with $0.0009 < p < 0.001$	Number of eigenstates with $p > 0.001$
First	618	47	359
Second	745	7	272

Table 7.1: Additional numbers for the QAOA ideal simulation pie chart.

In closing this section, we note that the probability of measuring the solution-associated eigenstate might be incremented by increasing the number of ansatz repetitions p [FGG14]. However, this would deepen the circuit and boost the noise and decoherence problem. A bargain between a higher success probability and the depth of the circuit is thus necessary when designing a QAOA-based problem resolution.

7.2.2 Running on QPU and noise simulation

It is natural to wonder how quantum noise would affect these probability distributions. To answer this question, we first try to use one of the real QPUs available on the IBM Quantum Platform. Unluckily, as you can see from Table 7.2, this computation too would require longer than 10 minutes to end, also considering that the number of function evaluations ranges from 35 up to 1000[9].

Execution time of one function evaluation	Estimated number of function evaluations	Total execution time
$19s$	220	$19s \cdot 220 = 4180s \approx 70min$

Table 7.2: Calculation to estimate the QPU usage for the QAOA algorithm. The estimate of the execution time for one function evaluation was done using the `ibm_tokyo` real backend.

Unlike the VQLS estimate, we do not need to consider the repetitions separately to calculate the quasi-distributions since the solver takes care of it independently and with optimized scheduling. Regardless, an approximated hour-long computation exceeds our means, making the fake backend our to-go option in these experiments too.

```
from qiskit_ibm_runtime import QiskitRuntimeService, Options
from qiskit.providers.fake_provider import FakeWashingtonV2
from qiskit.primitives import BackendSampler
from qiskit_aer.noise import NoiseModel

service = QiskitRuntimeService()

backend = FakeWashingtonV2()                      ## fake backend
noise_model = NoiseModel.from_backend(backend)    ## + noise

options = Options()
options.resilience_level = 2
options.optimization_level = 3
options.simulator.set_backend(backend)
options_dict = options.__dict__
```

[8] All the executed experiments show the same trend: either their probability distribution has a $60-5-35$ representation or a $70-1-29$ representation. That is why we only report two test runs.

[9] This does not happen with alarming frequency, but it is possible for some really unlucky initialization.

```
16
17      sampler = BackendSampler(backend, options_dict) ## fake Sampler
```

Listing 7.6: QAOA run using a Qiskit fake backend.

The code is almost identical to the one in Listing 6.11, with the most important difference being in line [17], where we define a custom `Sampler` compatible with the noise model we are implementing. Listing 7.6 is the only addition we ought to make to run the noisy simulations, so let us move directly to the results.

```
Best measurement:{'state': 51,
                  'bitstring': '0000110011',
                  'value': (-41.5+0j),
                  'probability': 0.0029296875}
```

Listing 7.7: QAOA results for MQ5 in a noisy simulation.

Comparing these results with the noiseless ones in Listing 7.5, we see that the noise mitigation applied to this last simulation allows for the algorithm to find the correct eigenvalue and, thus, the solution to MQ5. What changes is the probability of finding such a string, which has indeed decreased from the previous experiments.

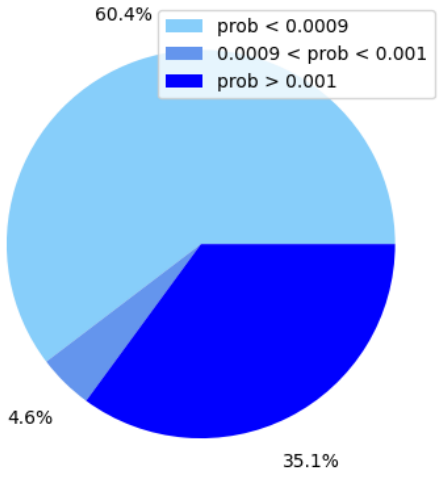
Illustrating the probabilities of the final eigenstate as a pie chart once more (Fig. 7.2), we see that the two major distributions we encountered in the noiseless simulation are now entirely comparable. Indeed, the differences between the elements in each section are now negligible, as reported in Table 7.3.

Test run	**Number of eigenstates with** $p < 0.0009$	**Number of eigenstates with** $0.0009 < p < 0.001$	**Number of eigenstates with** $p > 0.001$
First	488	280	256
Second	426	320	278

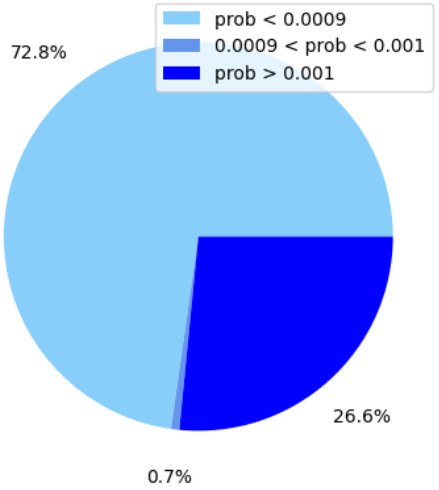
Table 7.3: Additional numbers for the QAOA noisy simulation pie chart.

Even though the number of interesting states to check is only halved, the algorithm is still able to output the correct string, allowing us to solve MQ5.

Analyzing these results, we encountered a peculiar effect: most of the values in the $p < 0.0009$ set are effectively **zero**. In contrast, in the noiseless simulation, the probabilities in this set were extremely small but positive nonetheless. Hence, the algorithm seems to obtain some benefit from a noisy computation.

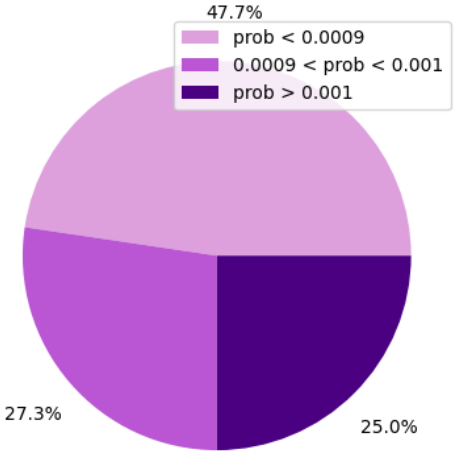


(a) First test run.

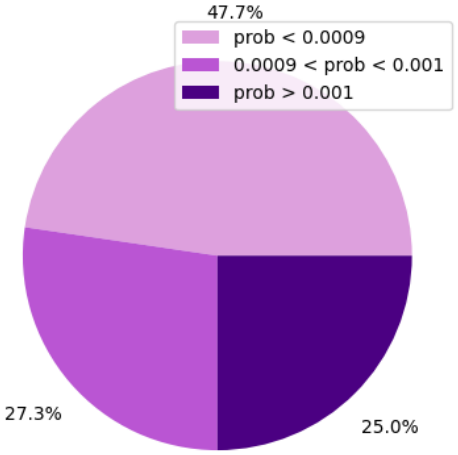


(b) Second test run.

Fig. 7.1: Pie charts representing the number of eigenstates divided by their measurement probability. The algorithm runs on a local simulator with no integrated noise model.



(a) First example run.



(b) Second example run.

Fig. 7.2: Pie charts representing the number of eigenstates divided by their measurement probability. The algorithm runs on a local simulator with an integrated noise model.

Chapter 8

Experiments with Quantum Annealing

Following [Ram+22], we foremost present two more possible encodings, namely the **truncated** and the **penalty** encoding, designed to map the problem Hamiltonian onto an annealer QPU (Section 8.1). Furthermore, quantum resources for these encoding strategies are calculated and compared between topologies (Subsection 8.1.3). Finally, we recreate the proposed **iterative procedure** to compare the results using the direct encoding and two different topologies. When the article was released, only the Pegasus topology was available. Hence, we tested the Zephyr layout to see how the topology impacts the computation (up to the current device capacity) (Subsections 8.2.2 and 8.2.1).

The code in this chapter can be found as Jupyter Notebooks and Python files in my public repository on **GitHub** [Gal]. Additional information about the software and the hardware used can be found in Appendix: Used Hardware and Software Versions.

8.1 Encoding the MQ problem into a BQM: Truncated and Penalty Encoding

In Section 7.1, we saw that the number of required (logical) quantum resources in the direct encoding scales exponentially with the number of original variables n. However, a diverse encoding technique can be defined to try and mitigate the scaling. Specifically, we report two more encodings, proposed in [Ram+22], and evaluate the quantum resources they would require.

Let us remark that the mathematical results of this section only apply to the number of logical qubits, as embedding the problem on a QPU may require qubit chains. The number of physical qubits is calculated in Subsection 8.1.3 by averaging on multiple runs of D-Wave's `minor_embedding` heuristic.

8.1.1 Truncated Encoding

The **truncated approach** relies on the idea of breaking the original polynomials $p_i(\mathbf{x})$ into smaller pieces with k-bounded length using ancillary variables. In more detail, a sum of n_i monomials $x_1 + \cdots + x_{n_i} = 0$ can be reduced to sums of up to k terms by adding

ancillae in the form

$$\begin{aligned} &x_1 + \cdots + x_{k-1} + a_1 = 0 \\ &a_1 + x_k + \cdots + x_{2k-2} + a_2 = 0 \\ &\cdots \\ &a_l + x_{n_i-k+1} + \cdots + x_{n_i} = 0 \end{aligned}$$

at the cost of expanding the number of equations to $\frac{n_i-2}{k-2}$ using $l = \frac{n_i-2}{k-2} - 1 = \frac{n_i-k}{k-2}$ ancillae (labeled a_i).

Since this encoding works on equations in the ANF representation, there are either single-qubit variables (i.e., monomials) or two-qubit interactions (i.e., terms like $x_i x_j$ with $i \neq j$). However, the polynomials must only display monomials. Thus, ancillae are introduced to substitute the two-qubit terms in the original ANF representation, adding the required penalty terms. In the worst-case scenario, $\binom{n}{2}$ ancillae are added to the polynomial. The maximum number of qubits needed to represent an MQ problem with m equations with n variables into a two-body Hamiltonian is then

$$\underbrace{\sum_{i=1}^{m} \left[\frac{n_i - 2}{k-2} \left(2^{\frac{k+2}{2}} - 2 - k \right) + \frac{n_i - k}{k-2} \right]}_{\substack{\text{variables added by doing} \\ ANF \mapsto NNF}} + \binom{n}{2} + n \tag{8.1}$$

where n_i is the number of monomials in each $p(\mathbf{x})$ and k (even) is the length of the partitions. The difference in scaling between truncated encodings for different values of k is reported in Fig. 8.1 alongside the direct encoding. While the latter requires fewer logical qubits for a small number of variables, its exponential scaling exceeds by far the scaling of the truncated approach. Furthermore, the scaling has been proven to be optimal for $k = 4$, a partition length that makes $2^{\frac{k+2}{2}}$ the dominant term, thus minimizing the total number of ancillae.

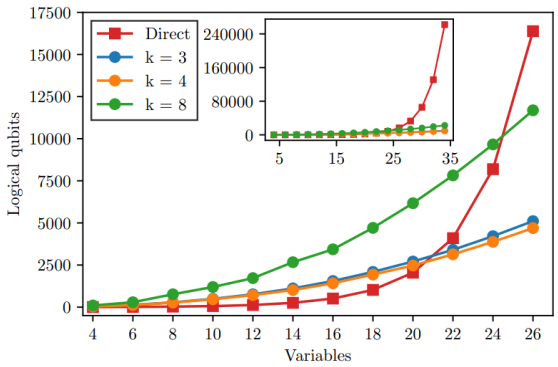


Fig. 8.1: Number of logical qubits needed to embed an MQ problem into the ground state of a Hamiltonian for the direct and truncated approaches. Source: [Ram+22].

Specifically, a 4-term sum only needs 2 extra ancillary variables to reduce it to up to two-body terms in the ANF↦NNF conversion. Fixing the value for k, the total number of qubits needed is

$$\frac{n^2}{2} + \frac{n}{2} - 4m + \frac{3}{2} \sum_{i=1}^{m} n_i$$

facing a polynomial scaling[1] with the number of parameters n, m. To obtain a scaling that only depends on the number of variables n, we use average values for m and n_i. Typically, the system will have as many equations as variables, each with an average number of monomials given by the total possible combinations of terms with two-body interactions, i.e.

$$n = m \qquad \text{and} \qquad n_i \sim \frac{n + \binom{n}{2}}{2} = \frac{n + n^2}{4}$$

These approximations yield a polynomial of degree 3 scaling in n, namely

$$\frac{3}{8}n^3 + \frac{7}{8}n^2 - \frac{7}{8}n$$

8.1.2 Penalty Encoding

An alternative encoding strategy is to model the equations in their ANF using logical quantum gates such as **CNOT** or Toffoli gates, which natively act over the $\mathbb{F}_2$ field, and then reproduce that circuit as an adiabatic evolution using penalty functions. In more detail, the MQ problem equations are modeled as Boolean operations on an output quantum register; that is, the actions of $+x_i$ and $+x_ix_j$ can be modeled to a **CNOT** and Toffoli gates targeting the output qubit and controlled by qubits $\{x_i\}$ and $\{x_i, x_j\}$ respectively. Then, a Hamiltonian is constructed with a ground state that follows the gate-by-gate implementation of the resulting circuit. The penalty functions needed to map the solution of an MQ problem into the ground state of a Hamiltonian are displayed in Table 8.1. The output auxiliary qubit z used in the penalty function of a given quantum gate is used as the target qubit x_t in the penalty function of the immediately following gate. These penalty functions contribute with positive energy if the state of the qubits involved does not match the logical Boolean operation that they map. Furthermore, the qubits used to initialize the output ancillae are penalized if they are in the $|1\rangle$ state since we assume the initial state to be the default one. The same reasoning is applied to the output ancillae where the final result of using each equation $p_i(\mathbf{x})$ is stored since we are interested in the solution with a zero output.

Gate	Boolean operation	Penalty function
NOT	$z = \bar{x}$	$2xz - x - z + 1$
Controlled-NOT	$z = x_cx_t$	$2x_cx_t - 2(x_c + x_t)z - 4(x_c + x_t)x_a + 4zx_a + x_c + x_t + z + 4x_a$
Toffoli	$z = x_{c1}x_{c2}x_t$	$-4x_{a1}x_{a2} + 4x_{a1}z - 4x_{a1}x_t - 2x_{a1}x_{c1} -$ $2x_{a2}x_{c2}2x_{a2}z + 2x_{a2}x_t + x_{c1}x_{c2} - 2x_tz + 4x_{a1} + 4x_{a2} + z + x_t$

Table 8.1: Summary of Boolean operations and their penalty function implementation in a quantum annealer as only constant, single, and two-qubit monomials appear in MQ problems. The result is saved in the qubit corresponding to the variable z, while the variable x corresponds to other qubits involved in the Boolean operation. The subscripts c, t, and a correspond to control, target, and ancilla.

The quantum resources needed to implement such encoding are governed by the number of monomials present in the equations of a given MQ problem, as they will dictate the number of gates to be implemented. As discussed for the truncated encoding, the average number of monomials scales as $\mathcal{O}(n^3)$, while the ancilla overhead needed for the

[1]This statement holds assuming that the total number of terms in the system of equations scales reasonably with the number of variables.

implementation of each **CNOT** or Toffoli gate does not change the overall scaling, as they require 1 or 2 ancillae respectively. Hence, the penalty and the truncated encoding scale similarly and outperform the direct encoding for large systems of equations.

8.1.3 Resources Comparison

Having discussed the theoretical scaling of logical qubits, we now turn to the physical quantum resources that these embeddings require. Since the minor embedding is a heuristic algorithm, we executed the embedding procedure four times and averaged the numbers of used physical qubits. The results are illustrated in Table 8.2 and Table 8.3 for the truncated embedding, and in Table 8.4 and Table 8.5 for the penalty.

To compute these averages, we defined a list of ANF problems with variables ranging from 4 to 18. However, we later restricted the maximum number of variables to 8 due to the limited qubit number available for the Zephyr device.

The results that were obtained align perfectly with our expectations. First, we can see that these alternative encodings lead to a higher number of total variables (i.e., original plus ancillae), thus making them more expensive in terms of used physical qubits. Furthermore, we can see that the impact of Zephyr's higher connectivity gets progressively more evident as the dimension of the problem grows. Therefore, even if the limitations of available quantum annealers are still quite restrictive, the ongoing developments seem to promise more advantages every day.

	4 bits	**6 bits**	**8 bits**	**10 bits**
logical_qubits	30.0	90.0	231.0	451.0
physical_qubits	56.5	222.5	733.5	1661.0

Table 8.2: Logical and physical qubits used for the truncated embedding on a Pegasus device.

	4 bits	**6 bits**	**8 bits**	**10 bits**
logical_qubits	30.00	90.0	231.00	451.0
physical_qubits	50.25	211.0	667.75	`NaN`

Table 8.3: Logical and physical qubits used for the truncated embedding on a Zephyr device.

	4 bits	**6 bits**	**8 bits**	**10 bits**
logical_qubits	61.0	150.00	345.00	645.00
physical_qubits	105.0	310.25	891.25	1805.25

Table 8.4: Logical and physical qubits used for the penalty embedding on a Pegasus device.

	4 bits	**6 bits**	**8 bits**	**10 bits**
logical_qubits	61.0	150.00	345.00	645.00
physical_qubits	88.5	279.5	769.0	`NaN`

Table 8.5: Logical and physical qubits used for the penalty embedding on a Zephyr device.

8.2 Experimental results

Let us make a few introductory statements before discussing the source code and the results related to the quantum annealing experiments.

To embed the MQ problems onto a QUBO model, we refer to the code available online on the article-associated GitHub repository [Ram] and proceed to discuss only the newly-written code of these experiments.

Given the embedding statistics in Subsection 8.1.3 and considering the small dimension of the tested problem, only direct encoding is used in the following experiments.

Furthermore, our code is designed as a Jupyter Notebook and an external Python file called `my_lib.py`, where a few auxiliary functions are defined. In more detail, an overall class called `dwave_runners` is built to implement the iterative procedure and the single runs on a D-Wave QPU. This class initialization is reported in Listing 8.1, and it takes as input the QUBO model `Q` previously computed in a dictionary format alongside its offset `offset`, the number of binary variables `bits`, and two strings to determine the QPU and the chain strength to use, respectively `topology` and `chosen_chainstrength`. The final two parameters that define this class are `numruns`, i.e., the number of samples for a single submitted problem (default is 1 000), and `T`, i.e., the annealing time for the experiments (default is $20\mu s$). After this general initialization, the QUBO model is transformed into a generic BQM[2] in line [31]. Finally, in line [32], the desired QPU (either the Zephyr **Advantage2_prototype2.2** or the Pegasus **Advantage_system6.3**) is selected, and the automated embedding is specified. The related source code of this process is reported in Listing 8.2.

As shown in Listing 8.1, two options have been provided for the chain strength, namely the *Article* or the *Ocean_Doc*. The latter is automatically computed by the Ocean function `uniform_torque_compensation` and is the standard definition in D-Wave documentation. However, the defined value is quite high, slowing the convergence to a solution. Therefore, the authors of [Ram+22] developed a heuristic method (see lines $[9-16]$ in Listing 8.3) that proved to be successful with many diverse instances. This chain strength can be seen as a penalization term for an extra constraint that forces the qubits on the chain to be equal, making it high enough thus not to alter the ground state (i.e., measure as few broken chains as possible) but not too high to introduce any issue. For more details on how to choose an optimal chain strength, see [Ale+23][3]. The chain strength is indeed calculated when the `single_run` method is called (see Listing 8.4), but the chosen strategy is defined a priori by the user when initializing the whole class.

[2] It is possible for the user to not manually turn the QUBO into a BQM, as Ocean has a few built-in conversions if one calls the methods designed for QUBO models. For instance, by calling the method `sample_qubo`, it is possible to use as input the QUBO model, as the method will take care of the conversion. However, since we need the BQM for other definitions, we preferred to instantiate the conversion manually and save its output in the `H` attribute.

[3] The additional information on this heuristic chain strength and the suggested article come from an e-mail exchange with one of the authors, specifically with Dr. Sergi Ramos-Calderer.

```
class dwave_runners:

    def __init__(self, Q, offset, bits, topology,
                 chosen_chainstrength, numruns = 1000, T=20):
        '''
            Initialize the variables for the class.

            Inputs:
                Q (dict): QUBO model.
                offset (float): offset of the QUBO model.
                bits (int): number of variables.
                topology (str): chosen topology,
                                either "Zephyr" or "Pegasus".
                chosen_chainstrength (str): chosen definition,
                                            either "Article"
                                            or "Ocean_Doc".
                numruns (int): number of sampling per run
                               (default = 1000).
                T (int): annealing time (default = 20).
        '''

        self.bits = bits
        self.numruns = numruns
        self.T = T
        self.topology = topology
        self.chosen_chainstrength = chosen_chainstrength
        self.energies_hist = []
        self.Q = Q
        self.offset = offset

        self.into_BQM()
        self.define_topology()
```

Listing 8.1: Initialization of the `dwave_runners` class.

```
    def define_topology(self):

        if self.topology == "Zephyr":
            self.sampler = EmbeddingComposite(
                            DWaveSampler(solver={'name':
                                    'Advantage2_prototype2.2'},
                                        )
                            )
            print('Running on Zephyr Topology.')

        elif self.topology == "Pegasus":
            self.sampler = EmbeddingComposite(
                            DWaveSampler(solver={'name':
                                    'Advantage_system6.3'},
                                        )
                            )
            print('Running on Pegasus Topology.')
```

Listing 8.2: Definition of the D-Wave sampler, with automated embedding on the chosen topology.

```
    def define_chainstrength(self):

        if self.chosen_chainstrength == 'Ocean_Doc':
            self.chain_strength = uniform_torque_compensation(self.H,
                                                              self.sampler)
            print(f'You chose the chainstrength from the
                        Ocean Documentation: {self.chain_strength}')

        elif self.chosen_chainstrength == 'Article':
            self.chain_strength = 0
            for i in self.Q.values():
                self.chain_strength += abs(i)
            self.chain_strength *= 3/len(self.Q)
            print(f'You chose the Article chainstrength:
                        {self.chain_strength}.')

        else:
            print('You can only choose Ocean_Doc or Article.')
```

Listing 8.3: Computation of the chosen chain strength, either following the definition from the original article or using the definition suggested from the Ocean documentation.

8.2.1 First Tests: Solving MQ5

We now turn to quantum annealing to try and solve the MQ5 problem defined in Listing 7.1. Given the small number of variables, direct encoding (see Section 7.1) is used.

Once the QUBO model is computed, we call for the class `dwave_runners` to define the computational environment and execute a single run of the sampling[4]. The method used in the source code is reported in Listing 8.4, where the chain strength is computed, and the sampling runs on the previously initialized `sampler`. The output `response` is a `SampleSet` Ocean object, which contains all the *numruns* samples configurations and energy levels and other information about the run, such as timing or embedding details.

```
    def single_run(self):

        self.define_chainstrength()
        response = self.sampler.sample(self.H,
                                       chain_strength=self.chain_strength,
                                       num_reads=self.numruns,
                                       annealing_time=self.T,
                                       answer_mode='histogram',
                                       label = f'mq_on_{self.topology}')
        print('Finished running the experiment!')

        return(response)
```

Listing 8.4: Function that runs **once** the sampling process for the given $\mathcal{H}$ on D-Wave devices.

This approach allows us to find the correct eigenstate in one single call with virtual certainty on both topologies.[5]

[4] This means that the whole sampling process is executed once, not that we only sample one single state.

[5] Over the last five months, we have run numerous experiments, and the solution has always been found in one iteration. Given the probabilistic nature of the algorithm, though, we reserve a small percentage of uncertainty.

```
Solution found: [1, 1, 0, 0, 1] with energy: 0.0
The solution found is one of the known solutions: True
```

Listing 8.5: Results for the MQ5 problem solved through quantum annealing. The displayed results are valid for both topologies.

Since the solution is found in a single iteration regardless of the topology, let us compare the details of those runs, starting with timing information. Table 8.6 reports all available timing-related values (in μs) for both topologies. Besides a few non-topology-related numbers that are trivially the same (e.g., the *qpu_anneal_time_per_sample*, which is set to $20\mu s$ by default), from the displayed timings, we can see how the Zephyr topology allows for an overall faster annealing process, reducing the sample time and the overall QPU usage time. In Fig. 8.2, a schematic representation of these timings and their relationships is reported.

	Zephyr	Pegasus	Differences
qpu_sampling_time	79000.00	98740.00	-19740.00
qpu_anneal_time_per_sample	20.00	20.00	0.00
qpu_readout_time_per_sample	38.45	58.20	-19.75
qpu_access_time	98227.21	114667.97	-16440.76
qpu_access_overhead_time	1030.79	1388.03	-357.24
qpu_programming_time	19227.21	15927.97	3299.24
qpu_delay_time_per_sample	20.55	20.54	0.01
post_processing_overhead_time	1.00	1.00	0.00
total_post_processing_time	1.00	1.00	0.00

Table 8.6: Timing comparison between the runs on different topologies. The unit measure for all values is the microsecond μs.

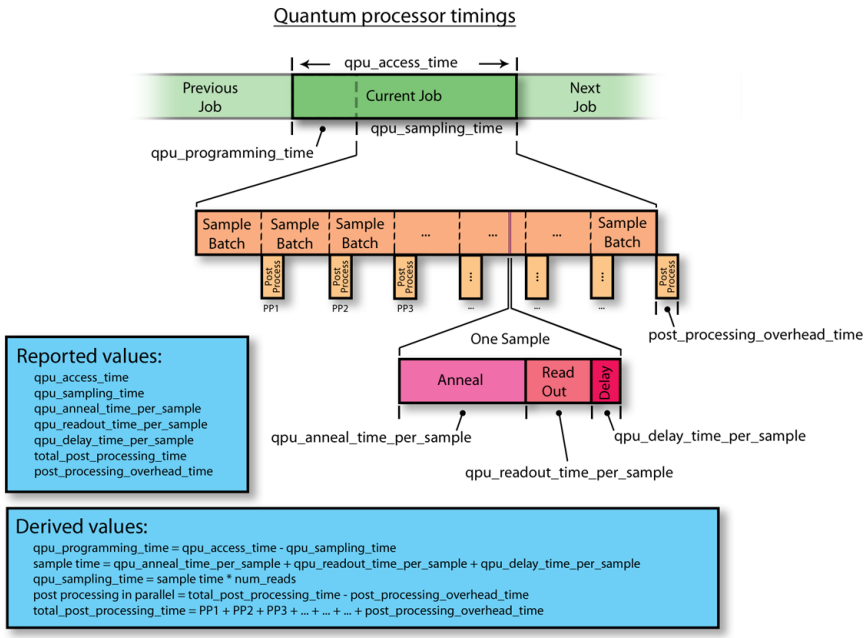


Fig. 8.2: Illustration of the timing information and their relationships. Source: [D-Wf].

Besides the timing information, we can analyze the embedding and the length of the used qubit chains. More in detail, the run on the Pegasus topology required 16 physical qubits, while the one on the Zephyr topology required 13 physical qubits. Table 8.7 lists the logical qubits and their representation on each topology. As we can see, in the Pegasus topology, 6 chains of length 2 are required compared to only 3 chains of analogous length in the Zephyr topology.

	q_0	q_1	q_2	q_3	q_4	q_5	q_6	q_7	q_8	q_9
Pegasus	2	2	2	1	2	1	1	2	2	1
Zephyr	2	1	1	1	1	2	2	1	1	1

Table 8.7: Logical qubits q_i embedding comparison between the two topologies. The numbers reported refer to the length of the used qubit chains in the embedding.

In Fig. 8.3 are reported the problem-associated graphs, the embedding on the QPU (both with highlighted chains and not), and the histograms displaying the energies associated with the samples. All the pictures were retrieved using the D-Wave tool `inspector`. The dots in the graph and the embedding represent the variables (treated as an Ising model[6]) and edges their interactions. The colour code is the following:

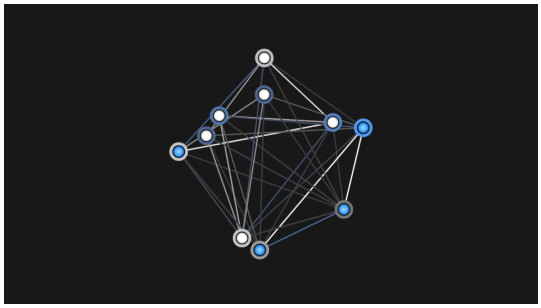
- the full dot represents the value associated with that variable in the given sample. White is for -1 and blue for $+1$;
- the hoop around the dot is for the variable's bias from the original problem formulation. Its colour depends on the sign and its intensity on the magnitude of such bias. Once again, blue is for positive biases, and white is for negative biases. Grey loops identify a nil bias, and hence, it is usually associated with the auxiliary qubits needed to build a chain;
- for the edges, the same convention as for the loops is followed;
- in the embedding pictures, the underlying topology is drawn in shades of black.

It is also interesting to confront the energy histograms[7], in which we can see that the Zephyr topology shows a smaller range of energies thanks to its higher connectivity and compactness.

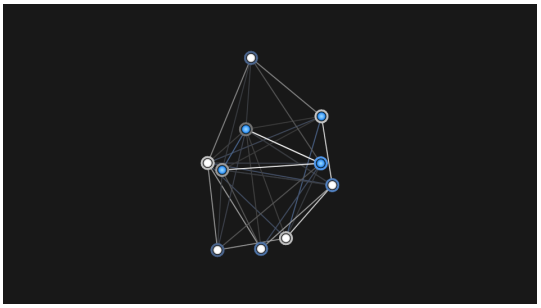
A final remark on these results is due. We report only a single run for both topologies as a specimen of the qualitative results tested over the last five months. Hence, even though they are not technically averages, they are completely replicable and a good representative of the trend we met in our experiments.

[6]For the embedding graph, the inspector only allows for the Ising model representation; hence, we decided to use it for the input graph too coherently.

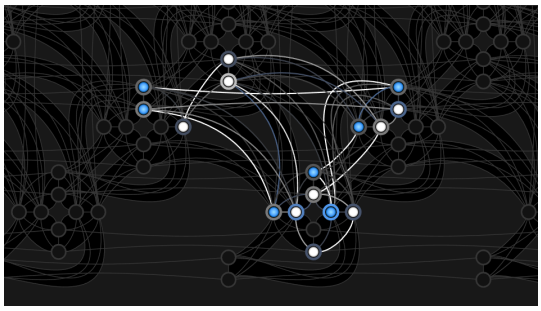
[7]Since the embedding may require additional physical qubits, the problem that the annealer solves is technically not the one we submitted (also called **source**) but an equivalent one (also called **target**). We can, therefore, choose which histogram we want to display: the source one (like in this case) or the target one.



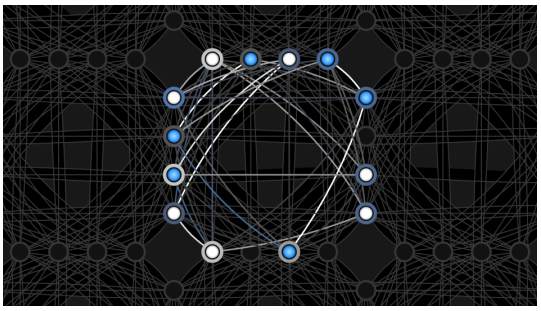
(a) Problem graph given as input to the Pegasus QPU.



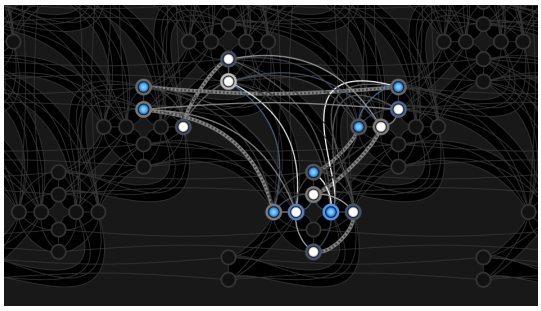
(b) Problem graph given as input to the Zephyr QPU.



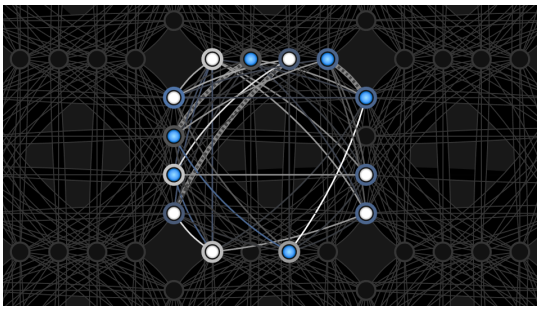
(c) Embedding on Pegasus topology.



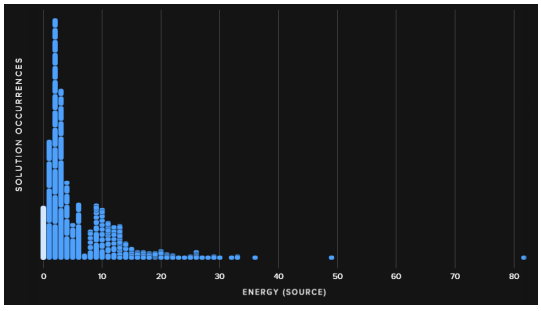
(d) Embedding on Zephyr topology.



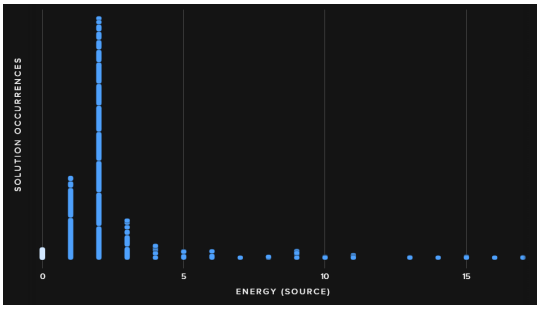
(e) Embedding on Pegasus topology with chains.



(f) Embedding on Zephyr topology with chains.



(g) Histogram for sampled energies by the Pegasus QPU.



(h) Histogram for sampled energies by the Zephyr QPU.

Fig. 8.3: Result analysis for the MQ5. The images on the left refer to the Pegasus QPU, while the ones on the right are to the Zephyr QPU. (a)-(b) Problem-associated graph to embed on QPU. (c)-(d) Embedding on the QPU. (e)-(f) Embedding on the QPU with highlighted qubit chains. (g)-(h) Histograms show the energy of all samples in the run, with the best sample being white. Looking at the energy axe, it is possible to notice how the Zephyr topology has a smaller range for the energy spectrum.

8.2.2 Solving MQ9 through Iterative Method

To test this quantum annealing approach, we can try to apply it to a problem with a higher dimension, namely the MQ problem with nine Boolean variables[8] and two possible solutions, namely $[1, 0, 0, 1, 0, 1, 0, 1, 1]$ or $[0, 0, 0, 1, 1, 0, 1, 1, 0]$. Concerning its final encoding, the reduction from a many-body Hamiltonian to a two-body Hamiltonian brings the number of variables from 9 to 46. Henceforth, this instance is referred to as MQ9 for brevity.

Due to hardware limitations, such as short annealing times or the quality of qubits, increasing the number of variables (and of the subsequent ancilla variables and physical qubits) may negatively impact the search for the ground state, yielding only sub-optimal results. To bypass this problem[9], in [Ram+22], the authors propose a heuristic iterative method that reduces the search subspace at every iteration based on the values of ancillae.

In more detail, we know from Section 7.1 that the rapid increase in the number of auxiliary qubits is due to the reduction from a multi-body Hamiltonian to a two-body Hamiltonian[10]. This means that the added ancillae represent products of other variables and, therefore, will have a stronger penalization than the original ones. Assuming no quantum state with zero energy[11] has been found, the idea is to **fix** a certain number of ancillae-associated values after every annealing protocol based on the low-energy configurations we sampled. If some qubits have the same result in all of the lowest energy states, we can assume that the Hamiltonian penalizes them more than others. Therefore, we can narrow the search subspace by substituting that value for the cost function variable and running a new annealing schedule on the reduced instance.

The amount of low-energy samples to check to fix the ancillae is a hyper-parameter (whose default value is set to 10), and it can be optimized from problem to problem. On the one hand, a too-low threshold might exclude the ground state from the reduced search space by fixing the ancillae to a wrong outcome. On the other hand, setting it too high may hinder fixing any ancillary variable, wasting the whole annealing schedule. The threshold can, therefore, be heuristically tuned by checking if the reduced subspace no longer contains the solution. This is achieved by comparing the lowest energy sample of the reduced Hamiltonian with its equivalent in the original Hamiltonian: if the former is lower than the latter, then we excluded the solution from the search subspace.

The source code[12] implementing the iterative method is reported in Listing 8.6. This method takes different values for the number of iterations and the threshold as optional inputs; otherwise, it requires only previously initialized attributes.

Since different iterative runs will end in different iterations, we only report the solution found, the success rates, and a little general information about the embedding on the first iteration.

[8] The definition of this problem is not described in this thesis as it is pretty long. It can be found in the GitHub repository, though, in the file "examples_nnf.py".

[9] This is intended up to the probabilistic nature of the quantum annealing procedure.

[10] In Section 8.1, we can see that this is a general trend for different encoding techniques, even though with varied scaling.

[11] Remark: we are taking polynomials on the Boolean field. Hence, if all the equations must be satisfied, the only acceptable energy value for the ground state is zero.

[12] The code snippet has been slightly modified (compared to the one present in the GitHub repository) to take into account the paper dimensions, but the core lines are intact.

```
    def iterative(self, iterations=5, treshold=10):

        '''
            Run on QPU multiple times, each iteration fixing
            ancillae that share the same value.

            Inputs:
                iterations (int): number of single runs (default=5).
                threshold (int): number of samples to compare
                                 to fix the ancillae (default=10).

            Outputs:
                solution (list): list of final found values.
                timing_info (dict): dictionary with total timings.
                physical_qubits (float): total number of the
                                         physical qubits used.
                it (int): final iteration run.
        '''

        ## general setup
        fixed = {}          # variable : value

        ###### iterating
        for it in range(iterations):

            ## info per run
            num_variables = self.H.num_variables
            print(f'Number of variables: {num_variables}')

            ## running on QPU
            response = self.single_it()
            record = response.record

            ## the samples are not ordered
            order = np.argsort(record['energy'])

            ## best outcomes
            best_sample = response.first.sample
            best_energy = response.first.energy

            ## info
            if it == 0:                 # first iteration
                #timings
                timing_info = response.info['timing']
                #embedding
                embedding_info = response.info['embedding_context']['
    embedding']
                physical_qubits, chains = self.get_info_on_embedding(
    embedding_info)
                show(response)
            else:                        # later iterations
                # timings
                for key in timing_info.keys():
                    timing_info[key] += response.info['timing'][key]
                # embedding
                ph_info = response.info['embedding_context']['
    embedding']
                ph_qubits, ph_chains = self.get_info_on_embedding(
    ph_info)
                physical_qubits += ph_qubits
```

```
57
58              ## energies histogram
59              ph_energy = []
60              for i in range(len(record.energy)):
61                  for j in range(record.num_occurrences[order[i]]):
62                      ph_energy.append(record.energy[order[i]])
63
64              self.energies_hist.append(ph_energy)
65
66              if best_energy == 0:
67                  print(f'Solution found with final iteration {it}.')
68                  #show(response)
69                  break
70
71              ## fixing variables
72              print("Fixing ancillae...")
73              for i in range(self.bits, num_variables):
74                  flag = True
75                  ph_value = record.sample[order[0]][i]
76
77                  # checking for shared values
78                  for k in range(min(threshold, len(record.sample))):
79                      if ph_value != record.sample[order[k]][i]:
80                          flag = False
81                          break
82                  if flag:
83                      fixed[i] = ph_value
84
85              ## updating Hamiltonian
86              for var in fixed.keys():
87                  ## checking if it's a new fix or not
88                  if var not in self.H.variables:
89                      continue
90                  else:
91                      self.H.fix_variable(var, fixed[var])
92
93              ## creating new QUBO model
94              self.Q, self.offset = self.H.to_qubo()
95
96          ###### reconstruncting
97          print('Reconstructing the final state...')
98          solution = []
99          for i in range(self.bits):
100             solution.append(best_sample[i])
101
102         print(f'Solution found: {solution}')
103
104         return(solution, timing_info, physical_qubits, it)
```

Listing 8.6: Iterative method to solve general MQ problems.

Starting with the final output of this procedure, we find that on both topologies, the iterative method finds a solution in about 3 iterations two times out of three. Suppose the solution is not found in such a few iterations. In that case, we might as well stop the computation and restart the whole procedure[13], to not squander quantum resources[14]. In the specific runs we report, the one executed on the Pegasus topology took three

[13] We tried running with over 20 iterations to look for a statistical upper bound, but with no luck.
[14] D-Wave too, just like IBM, gives access to its QPUs and hybrid solvers for a limited monthly time.

iterations. In contrast, the one on the Zephyr topology found the solution on the first annealing protocol.

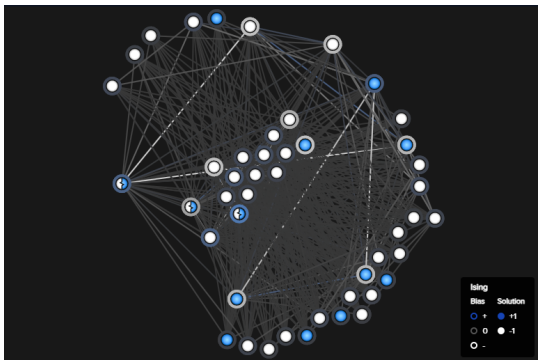
```
The iterative method on the Pegasus topology required a total of
391 physical qubits for 3 iterations.
It found a solution: True.
Found solution: [1, 0, 0, 1, 0, 1, 0, 1, 1]

The iterative method on the Zephyr topology required a total of
135 physical qubits for 1 iteration.
It found a solution: True.
Found solution: [0, 0, 0, 1, 1, 0, 1, 1, 0]
```

Listing 8.7: Results for the MQ9 problem solved through quantum annealing.

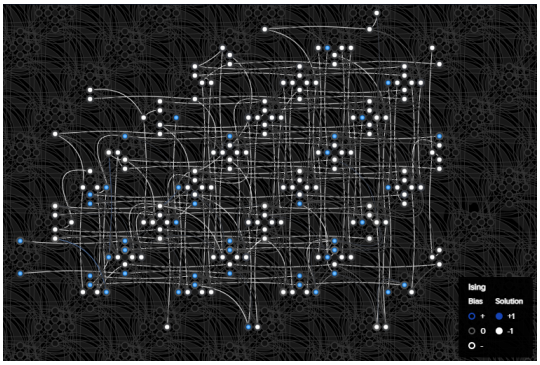
Concerning the first embedding[15], we find that to encode the 46 initial variables into the Pegasus topology, $\sim$ 185 physical qubits are required (187 for this specific instance). In contrast, the Zephyr topology only uses $\sim$ 140 physical qubits (135 for this specific instance). The graphs, embeddings, and energy histograms are reported in Fig. 8.4 for the Pegasus topology and in Fig. 8.5 for the Zephyr topology. Two main differences from the results of MQ5 must be discussed:

- running the annealing schedule with the article-defined chain strength causes a few chains to break in both topologies, but this does not hinder the algorithm's success. Conversely, using the documentation's chain strength erases the small percentage of chain breaks but makes the convergence to the ground state more difficult;
- such diverse energy values between histograms are due to the stochastic nature of the algorithm and to the two runs finding different ground states.

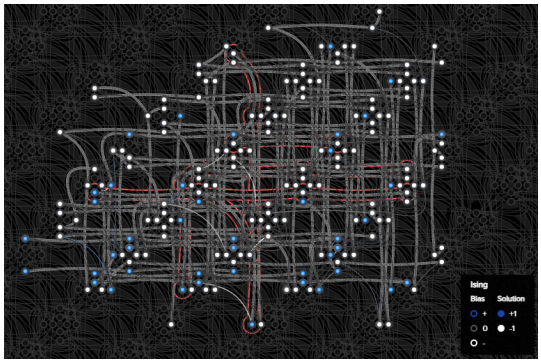
[15]The reported numbers are averaged on four runs for each topology.



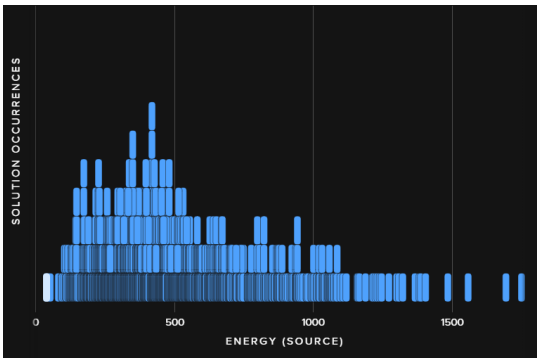
(a) Input problem graph for the Pegasus QPU.



(b) Embedding on Pegasus topology.

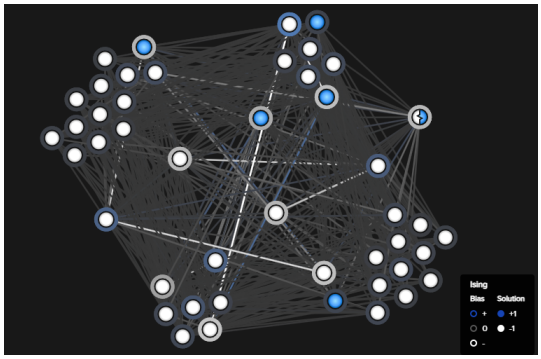


(c) Embedding on Pegasus topology with chains.

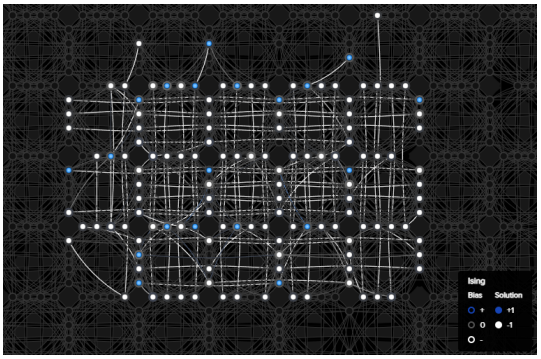


(d) Sampled energies by the Pegasus QPU.

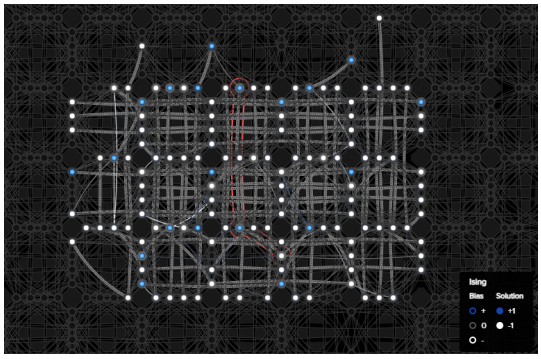
Fig. 8.4: Result analysis for the MQ9 run on Pegasus topology. (a) Problem-associated graph to embed on QPU. (b) Embedding on the QPU. (c) Embedding on the QPU with highlighted qubit chains. The red highlights broken chains for the displayed sample. (d) Histogram showing the energy of all samples in the run, with the best sample in white.



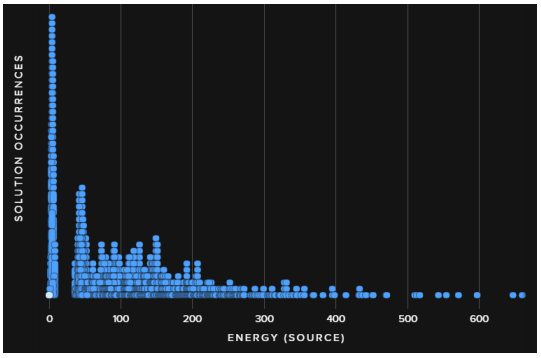
(a) Input problem graph for the Zephyr QPU.



(b) Embedding on Zephyr topology.



(c) Embedding on Zephyr topology with chains.



(d) Sampled energies by the Zephyr QPU.

Fig. 8.5: Result analysis for the MQ9 run on Zephyr topology. (a) Problem-associated graph to embed on QPU. (b) Embedding on the QPU. (c) Embedding on the QPU with highlighted qubit chains. The red highlights broken chains for the displayed sample. (d) Histogram showing the energy of all samples in the run, with the best sample in white.

Chapter 9

Conclusions and Future Work

We start this final chapter by reviewing the experimental results of the three implemented algorithms, dividing such discussions by their underlying quantum model. Ideas to complement or improve the experiments are also presented.

In closing this thesis, we propose a new method for analyzing a problem-associated landscape, hoping it will lead to exciting discoveries.

9.1 Experiments on the Gate-model

Using quantum circuits and gate-based computing, we tested two variational algorithms for solving linear systems of equations and MQ problems. Undoubtedly, these experiments are the ones that most display the effects of hardware limitations concerning both the number of available qubits and efficiency.

The VQLS (see Chapter 6), specifically, required multiple runs to find a solution with a decent accuracy level, even for a problem small like ours. Considering that the implemented code can and will be optimized, we expect the problem to be mitigated but not completely erased, as it is most likely due to a landscape studded of local minima. As shown in the cost function plots (Figs 6.2-6.7) and the associated accuracy tables (Tables 6.2-6.4), choosing a good initialization strategy is crucial to the correct algorithm execution. Hence, following the idea presented in [RSL22], we deem it interesting to explore the landscape through a **fast-and-slow algorithm**, which uses Bayesian Learning to identify a promising region in the parameter space to initialize later a local optimizer that, hopefully, converges faster to the global minimum.

On the other hand, the QAOA algorithm found a solution in every enactment, even with a relatively small probability (see Chapter 7). Even though we did not report the probabilities of the whole final eigenstate, we did encounter quite an interesting *bit* in the noisy simulation: most of the values in the $p < 0.0009$ are effectively **zero**. This effect may be due to some inference from the quantum noise or just a curious coincidence associated with this specific instance. Regardless, it is a phenomenon worth investigating with experiments on different and larger instances of the MQ problem.

It may also be useful to increase the probability of measuring the correct state. However, it would require a deeper circuit, which would restrict the physical implementation of the algorithm even more than it already is. An alternative would be to implement the so-called **Grover Adaptive Search** [GWG21], which could help develop shallower circuits to solve QUBO problems with a higher accuracy.

9.2 Experiments with Quantum Annealing

Our final leg of experiments was on solving MQ problems using quantum annealers (see Chapter 8). These results proved to be the most efficient and accurate of all three algorithms, encouraging us to find possible applications of this method.

Given the importance of MQ problems in post-quantum cryptography, the idea is to devise an algebraic attack for the **MinRank problem**, such as the one in [FLP08], to try gaining a quantum speedup over the classical implementation. It is likely, though, that the currently available quantum hardware will not have enough resources to implement such an attack directly. Hence, some realization strategies should be ideated, such as decomposing the problem into smaller, QPU-solvable problems or devising a hybrid algorithm to exploit the best of both worlds.

9.3 Analising Binary Quadratic Models through Complex Networks theory and the PyQubo library

Considering the limited capacity of current quantum hardware, finding a way to reduce the search space is crucial, and, indeed, many strategies have already been proposed and implemented (see [D-Wg] for instance). Since BQMs are easily mapped to a graph, we deem it interesting to analyze such representations as **complex networks** and see if the statistical information gathered on similarly defined problems can be used as a bootstrap for both variational algorithms and quantum annealing protocols, leveraging a meta-learning approach [Wil+21].

This idea is also partially due to initial experiments[1] about the direct encoding onto the annealers' QPUs. Using the Python library **PyQubo** [Rec], we tried encoding the given problem Hamiltonian using an automated mapping instead of the ad-hoc function defined in [Ram]. However, the automated mapping selected a whole different set of interacting qubits to reduce the Hamiltonian degree and led to different annealing computations and performances.

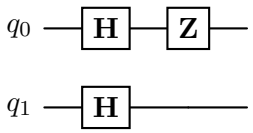
Therefore, alongside a complex network analysis, we will also keep experimenting with the PyQubo library to see if we can extrapolate more information on the problem topology.

[1] Not reported in this thesis.

Appendix

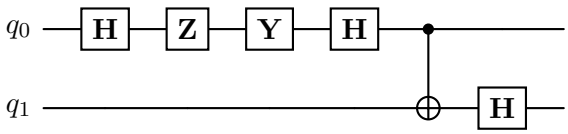
Calculating the depth of a circuit

In Section 3.1 we defined the **depth** of a circuit as the length of the critical path in the circuit, also saying that it represents the maximum number of gates that must be *serially* implemented. Since scheduling is crucial to the correct execution of our circuit when calculating the depth, we must always remember to consider *idle times*, and specifically the ones forced by other qubits dependency. Framing depth circuit as a running-time measure[2] makes it logical to use gates and the associated idle[3] as the **unit of measurement** to calculate our depth. We now present a few examples[4] to illustrate how this value is inferred from a circuit, starting with the following:



Our first example has a two-qubit quantum register onto which is mapped the unitary $(\mathbf{Z} \otimes \mathbb{1})(\mathbf{H} \otimes \mathbf{H})$. In this case, the Hadamard gates can be implemented in parallel, just as the phase flip and the identity. Hence, the depth of this circuit is the maximum number of gates on a single qubit line, i.e., 2.

Let's now consider another two-qubit quantum circuit, namely



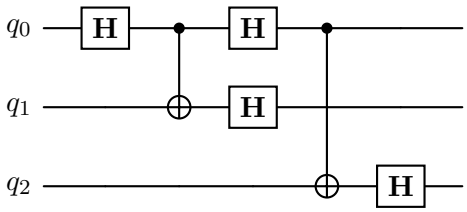
In this second example, five unitaries act on the first qubit q_0 (i.e. $\mathbf{CX}, \mathbf{H}, \mathbf{Y}, \mathbf{Z}$, and $\mathbf{H}$) while only two unitaries act on q_1 (i.e. $\mathbf{CX}$ and $\mathbf{H}$). One could easily be mistaken and say that this circuit has a depth of 5, not considering that to execute its two operations, the second register has to wait for the first qubit to reach the controlled gate. Hence, this circuit has a depth of 6: four idle-time units followed by the $\mathbf{CX}$ and the $\mathbf{H}$.

The previous line of reasoning can be easily generalized to more complex circuits defined in higher-dimension Hilbert spaces. Let us present a final example using the following three-qubit quantum register:

[2] Depth does not represent the actual execution time (in s) on a processor, but we can expect a deeper circuit to run longer than a shallower one.

[3] Following the matrix tensor vector notation, we can think of the idle of a given qubit as identity gates acting on it every time a non-trivial gate acts on a different qubit, leaving the unaltered one in wait.

[4] Inspired by [Gun].



The first qubit has no dependency on the others, hence its depth[5] is 4. The second qubit has a **CX** with control on q_0, meaning that we must add an idle unit in correspondence of the first Hadamard on q_0 to its own two gates. We underline that the Hadamard on this qubit has no dependency nor other limitations. Therefore, it can be computed in parallel with the second Hadamard on q_0. Finally, our last qubit has only two gates acting on it, but it has to wait for q_0 to end all previous computations, increasing its depth to 6. Hence, the depth of the circuit as a whole is 6.

[5]With some abuse of terminology, it is not uncommon to refer to the number of units on a line as its depth, leaving out of consideration the proper definition of gates or circuit.

Used Hardware and Software Versions

We report a few tables with all the information about the system used to run the experiments presented in this thesis, both related to the hardware and the software.

Software	Version
jupyter_client	7.4.9
jupyter_core	5.1.3
jupyter-events	0.6.3
jupyter_server	2.1.0
jupyter_server_terminals	0.4.4
jupyterlab-pygments	0.2.2
jupyterlab-widgets	3.0.3
Markdown	3.4.1
matplotlib	3.6.0
matplotlib-inline	0.1.6
minorminer	0.2.11
numpy	1.23.5
pandas	1.5.2
pyqubo	1.4.0
scipy	1.9.1
sympy	1.11.1
	System information
Python version	3.10.7
Python compiler	MSC v.1933 64 bit (AMD64)
Python build	tags/v3.10.7:6cc6b13, Sep 5 2022 14:08:36

Table 9.1: Software information: general Python packages installed.

Software	Version
qiskit	0.44.1
qiskit-terra	0.25.1
qiskit_aer	0.12.2
qiskit_algorithms	0.2.1
qiskit_optimization	0.6.0
qiskit_ibm_provider	0.7.2
qiskit_ibm_runtime	0.15.1

Table 9.2: Software information for the gate model experiments.

Software	Version
dimod	0.12.6
dwave-cloud-client	0.10.6
dwave-drivers	0.4.4
dwave-greedy	0.3.0
dwave-hybrid	0.6.10
dwave-inspector	0.4.2
dwave-inspectorapp	0.3.1
dwave-neal	0.6.0
dwave-networkx	0.8.14
dwave-ocean-sdk	6.4.1
dwave-preprocessing	0.5.4
dwave-samplers	1.0.0
dwave-system	1.19.0
dwave-tabu	0.5.0
dwavebinarycsp	0.2.0

Table 9.3: Software information for the quantum annealing experiments.

Hardware	System information
OS	Windows 11
GPU	NVIDIA GeForce GTX 1660 Ti
CPU	Intel® Core™ i7-9750H CPU @ 2.60GHz
RAM	8,00 GB (7,88 GB available)

Table 9.4: Hardware information.

Hardware	#Qubits	Quantum Volume[6]	Processor type
`ibm_brisbane`	127	128	Eagle $r3$
`FakeWashingtonV2`	127	/	/

Table 9.5: Quantum hardware information for the gate model experiments.

Hardware	Topology	#Qubits	#Couplers	Native Complete Subgraphs
`Advantage`	Pegasus	5 000+	35 000+	K_4, $K_{6,6}$
`Advantage Prototype`	Zephyr	1 200+	10 000+	K_4, $K_{8,8}$

Table 9.6: Quantum hardware information for the quantum annealing model experiments.

[6] Quantum volume is a metric that measures the capabilities and error rates of a quantum computer. It expresses the maximum size of square quantum circuits that can be implemented successfully by the quantum device.

Bibliography

[Deu89] David Elieser Deutsch. "Quantum computational networks". In: *Proceedings of the royal society of London. A. mathematical and physical sciences* 425.1868 (1989), pp. 73–90.

[DBE95] David Elieser Deutsch, Adriano Barenco, and Artur Ekert. "Universality in quantum computation". In: *Proceedings of the Royal Society of London. Series A: Mathematical and Physical Sciences* 449.1937 (1995), pp. 669–677.

[Deu85] David Deutsch. "Quantum theory, the Church–Turing principle and the universal quantum computer". In: *Proceedings of the Royal Society of London. A. Mathematical and Physical Sciences* 400.1818 (1985), pp. 97–117.

[Kun+22] Vyacheslav Kungurtsev et al. "Iteration complexity of variational quantum algorithms". In: *arXiv preprint arXiv:2209.10615* (2022).

[De +23] Giacomo De Palma et al. "Limitations of variational quantum algorithms: a quantum optimal transport approach". In: *PRX Quantum* 4.1 (2023), p. 010309.

[AL18] Tameem Albash and Daniel A Lidar. "Adiabatic quantum computation". In: *Reviews of Modern Physics* 90.1 (2018), p. 015002.

[Far+00] Edward Farhi et al. "Quantum computation by adiabatic evolution". In: *arXiv preprint quant-ph/0001106* (2000).

[Aha+08] Dorit Aharonov et al. "Adiabatic quantum computation is equivalent to standard quantum computation". In: *SIAM review* 50.4 (2008), pp. 755–787.

[AHS93] Patricia Amara, D Hsu, and John E Straub. "Global energy minimum searches using an approximate solution of the imaginary time Schrödinger equation". In: *The Journal of Physical Chemistry* 97.25 (1993), pp. 6715–6721.

[Fin+94] Aleta Berk Finnila et al. "Quantum annealing: A new method for minimizing multidimensional functions". In: *Chemical physics letters* 219.5-6 (1994), pp. 343–348.

[KN98] Tadashi Kadowaki and Hidetoshi Nishimori. "Quantum annealing in the transverse Ising model". In: *Physical Review E* 58.5 (1998), p. 5355.

[Luc14] Andrew Lucas. "Ising formulations of many NP problems". In: *Frontiers in physics* 2 (2014), p. 5.

[DA17] Raouf Dridi and Hedayat Alghassi. "Prime factorization using quantum annealing and computational algebraic geometry". In: *Scientific reports* 7.1 (2017), p. 43048.

[GKD18] Fred Glover, Gary Kochenberger, and Yu Du. "A tutorial on formulating and using QUBO models". In: *arXiv preprint arXiv:1811.11538* (2018).

[HS+52] Magnus R Hestenes, Eduard Stiefel, et al. “Methods of conjugate gradients for solving linear systems”. In: *Journal of research of the National Bureau of Standards* 49.6 (1952), pp. 409–436.

[HHL09] Aram W Harrow, Avinatan Hassidim, and Seth Lloyd. “Quantum algorithm for linear systems of equations”. In: *Physical review letters* 103.15 (2009), p. 150502.

[Bra+23] Carlos Bravo-Prieto et al. “Variational quantum linear solver”. In: *Quantum* 7 (2023), p. 1188.

[GJ79] Michael R Garey and David S Johnson. “Computers and intractability”. In: *A Guide to the* (1979).

[Bar09] Gregory Bard. *Algebraic cryptanalysis*. Springer Science & Business Media, 2009.

[Ram+22] Sergi Ramos-Calderer et al. “Solving systems of Boolean multivariate equations with quantum annealing”. In: *Physical Review Research* 4.1 (2022), p. 013096.

[FGG14] Edward Farhi, Jeffrey Goldstone, and Sam Gutmann. “A quantum approximate optimization algorithm”. In: *arXiv preprint arXiv:1411.4028* (2014).

[Sip96] Michael Sipser. “Introduction to the Theory of Computation”. In: *ACM Sigact News* 27.1 (1996), pp. 27–29.

[Wat08] John Watrous. “Quantum computational complexity”. In: *arXiv:0804.3401* (2008).

[HDR17] Salah Haggag, Fatma Desokey, and Moutaz Ramadan. “A cosmological inflationary model using optimal control”. In: *Gravitation and Cosmology* 23 (2017), pp. 236–239.

[Bis06] Christopher Bishop. “Pattern recognition and machine learning”. In: *Springer google schola* 2 (2006), pp. 5–43.

[Wan+06] Yong Wang et al. “Inferring gene regulatory networks from multiple microarray datasets”. In: *Bioinformatics* 22.19 (2006), pp. 2413–2420.

[KGV83] Scott Kirkpatrick, C Daniel Gelatt Jr, and Mario P Vecchi. “Optimization by simulated annealing”. In: *science* 220.4598 (1983), pp. 671–680.

[Wei00] Stefan Weinzierl. “Introduction to Monte Carlo methods”. In: *arXiv preprint hep-ph/0006269* (2000).

[GG84] Stuart Geman and Donald Geman. “Stochastic relaxation, Gibbs distributions, and the Bayesian restoration of images”. In: *IEEE Transactions on pattern analysis and machine intelligence* 6 (1984), pp. 721–741.

[NC10] Michael A Nielsen and Isaac L Chuang. *Quantum computation and quantum information*. Cambridge university press, 2010.

[SWL04] Marcelo S Sarandy, L-A Wu, and Daniel A Lidar. “Consistency of the Adiabatic Theorem”. In: *Quantum Information Processing* 3 (2004), pp. 331–349.

[Mos01] Ali Mostafazadeh. *Dynamical invariants, adiabatic approximation and the geometric phase*. Nova Science Publishers, 2001.

[DJ92] David Deutsch and Richard Jozsa. “Rapid solution of problems by quantum computation”. In: *Proceedings of the Royal Society of London. Series A: Mathematical and Physical Sciences* 439.1907 (1992), pp. 553–558.

[Sim94] Daniel R. Simon. "On the power of quantum computation". In: *Proceedings 35th Annual Symposium on Foundations of Computer Science* (1994), pp. 116–123.

[Sho94] Peter W Shor. "Algorithms for quantum computation: discrete logarithms and factoring". In: *Proceedings 35th annual symposium on foundations of computer science*. Ieee. 1994, pp. 124–134.

[Cai23] Jin-Yi Cai. "Shor's Algorithm Does Not Factor Large Integers in the Presence of Noise". In: *arXiv preprint arXiv:2306.10072* (2023).

[GE21] Craig Gidney and Martin Ekerå. "How to factor 2048 bit RSA integers in 8 hours using 20 million noisy qubits". In: *Quantum* 5 (2021), p. 433.

[Gro96] Lov K Grover. "A fast quantum mechanical algorithm for database search". In: *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*. 1996, pp. 212–219.

[BBW05] William P Baritompa, David W Bulger, and Graham R Wood. "Grover's quantum algorithm applied to global optimization". In: *SIAM Journal on Optimization* 15.4 (2005), pp. 1170–1184.

[Amb04] Andris Ambainis. "Quantum search algorithms". In: *ACM SIGACT News* 35.2 (2004), pp. 22–35.

[BL17] Daniel J Bernstein and Tanja Lange. "Post-quantum cryptography". In: *Nature* 549.7671 (2017), pp. 188–194.

[Fow+12] Austin G Fowler et al. "Surface codes: Towards practical large-scale quantum computation". In: *Physical Review A* 86.3 (2012), p. 032324.

[Cai+23] Zhenyu Cai et al. "Quantum error mitigation". In: *Reviews of Modern Physics* 95.4 (2023), p. 045005.

[Yao93] A Chi-Chih Yao. "Quantum circuit complexity". In: *Proceedings of 1993 IEEE 34th Annual Foundations of Computer Science*. IEEE. 1993, pp. 352–361.

[MW19] Abel Molina and John Watrous. "Revisiting the simulation of quantum Turing machines by quantum circuits". In: *Proceedings of the Royal Society A* 475.2226 (2019), p. 20180767.

[Bar+95] Adriano Barenco et al. "Elementary gates for quantum computation". In: *Physical review A* 52.5 (1995), p. 3457.

[Bar95] Adriano Barenco. "A universal two-bit gate for quantum computation". In: *Proceedings of the Royal Society of London. Series A: Mathematical and Physical Sciences* 449.1937 (1995), pp. 679–683.

[Aha03] Dorit Aharonov. "A simple proof that Toffoli and Hadamard are quantum universal". In: *arXiv preprint quant-ph/0301040* (2003).

[WW11] Colin P Williams and Colin P Williams. "Quantum gates". In: *Explorations in Quantum Computing* (2011), pp. 51–122.

[KSV02] Alexei Yu Kitaev, Alexander Shen, and Mikhail N Vyalyi. *Classical and quantum computation*. 47. American Mathematical Soc., 2002.

[LDX18] Gushu Li, Yufei Ding, and Yuan Xie. "Tackling the Qubit Mapping Problem for NISQ-Era Quantum Devices. arXiv". In: *arXiv preprint arXiv:1809.02573* (2018).

[Cór+21] Antonio D Córcoles et al. “Exploiting dynamic quantum circuits in a quantum algorithm with superconducting qubits”. In: *Physical Review Letters* 127.10 (2021), p. 100501.

[Ben80] Paul Benioff. “The computer as a physical system: A microscopic quantum mechanical Hamiltonian model of computers as represented by Turing machines”. In: *Journal of statistical physics* 22 (1980), pp. 563–591.

[CGD23] Elias F Combarro, Samuel González-Castillo, and Alberto Di Meglio. *A Practical Guide to Quantum Machine Learning and Quantum Optimization: Hands-on Approach to Modern Quantum Algorithms*. Packt Publishing Ltd, 2023.

[Cer+21a] Marco Cerezo et al. “Variational quantum algorithms”. In: *Nature Reviews Physics* 3.9 (2021), pp. 625–644.

[Kan+17] Abhinav Kandala et al. “Hardware-efficient variational quantum eigensolver for small molecules and quantum magnets”. In: *nature* 549.7671 (2017), pp. 242–246.

[Gar+20] Bryan T Gard et al. “Efficient symmetry-preserving state preparation circuits for the variational quantum eigensolver algorithm”. In: *npj Quantum Information* 6.1 (2020), p. 10.

[OCG19] Matthew Otten, Cristian L Cortes, and Stephen K Gray. “Noise-resilient quantum dynamics using symmetry-preserving ansatzes”. In: *arXiv preprint arXiv:1910.06284* (2019).

[Tka+21] Nikolay V Tkachenko et al. “Correlation-informed permutation of qubits for reducing ansatz depth in the variational quantum eigensolver”. In: *PRX Quantum* 2.2 (2021), p. 020337.

[Had+19] Stuart Hadfield et al. “From the quantum approximate optimization algorithm to a quantum alternating operator ansatz”. In: *Algorithms* 12.2 (2019), p. 34.

[HS05] Naomichi Hatano and Masuo Suzuki. “Finding exponential product formulas of higher orders”. In: *Quantum annealing and other optimization methods*. Springer, 2005, pp. 37–68.

[Bra+19] Sergey Bravyi et al. “Obstacles to state preparation and variational optimization from symmetry protection. arXiv”. In: *arXiv preprint arXiv:1910.08980* (2019).

[KB14] Diederik P Kingma and Jimmy Ba. “Adam: A method for stochastic optimization”. In: *arXiv preprint arXiv:1412.6980* (2014).

[Küb+20] Jonas M Kübler et al. “An adaptive optimizer for measurement-frugal variational algorithms”. In: *Quantum* 4 (2020), p. 263.

[Sto+20] James Stokes et al. “Quantum natural gradient”. In: *Quantum* 4 (2020), p. 269.

[KB22] Bálint Koczor and Simon C Benjamin. “Quantum natural gradient generalized to noisy and nonunitary circuits”. In: *Physical Review A* 106.6 (2022), p. 062416.

[Wil+21] Max Wilson et al. “Optimizing quantum heuristics with meta-learning”. In: *Quantum Machine Intelligence* 3 (2021), pp. 1–14.

[HD21] Patrick Huembeli and Alexandre Dauphin. “Characterizing the loss landscape of variational quantum circuits”. In: *Quantum Science and Technology* 6.2 (2021), p. 025011.

[MKW21] Carlos Ortiz Marrero, Mária Kieferová, and Nathan Wiebe. “Entanglement-induced barren plateaus”. In: *PRX Quantum* 2.4 (2021), p. 040316.

[Lar+22] Martin Larocca et al. “Diagnosing barren plateaus with tools from quantum optimal control”. In: *Quantum* 6 (2022), p. 824.

[McC+18] Jarrod R McClean et al. “Barren plateaus in quantum neural network training landscapes”. In: *Nature communications* 9.1 (2018), p. 4812.

[Cer+21b] Marco Cerezo et al. “Cost function dependent barren plateaus in shallow parametrized quantum circuits”. In: *Nature communications* 12.1 (2021), p. 1791.

[MN07] Satoshi Morita and Hidetoshi Nishimori. “Convergence of quantum annealing with real-time Schrödinger dynamics”. In: *Journal of the Physical Society of Japan* 76.6 (2007), p. 064002.

[DCS05] Arnab Das, Bikas K Chakrabarti, and Robin B Stinchcombe. “Quantum annealing in a kinetically constrained system”. In: *Physical Review E* 72.2 (2005), p. 026701.

[Raj+23] Atanu Rajak et al. “Quantum annealing: An overview”. In: *Philosophical Transactions of the Royal Society A* 381.2241 (2023), p. 20210417.

[DC08] Arnab Das and Bikas K Chakrabarti. “Colloquium: Quantum Annealing and Analog Quantum Computation”. In: *Reviews of Modern Physics* 80.3 (2008), p. 1061.

[CMR14] Jun Cai, William G Macready, and Aidan Roy. “A practical heuristic for finding graph minors”. In: *arXiv preprint arXiv:1406.2741* (2014).

[Ami+23] Mohammad H Amin et al. “Quantum error mitigation in quantum annealing”. In: *arXiv preprint arXiv:2311.01306* (2023).

[Pow94] Michael JD Powell. *A direct search optimization method that models the objective and constraint functions by linear interpolation*. Springer, 1994.

[Pow64] Michael JD Powell. “An efficient method for finding the minimum of a function of several variables without calculating derivatives”. In: *The computer journal* 7.2 (1964), pp. 155–162.

[SC95] Jun John Sakurai and Eugene D Commins. *Modern quantum mechanics, revised edition*. 1995.

[BOA13] Ryan Babbush, Bryan O’Gorman, and Alán Aspuru-Guzik. “Resource efficient gadgets for compiling adiabatic quantum optimization problems”. In: *Annalen der Physik* 525.10-11 (2013), pp. 877–888.

[Bia08] JD Biamonte. “Nonperturbative k-body to two-body commuting conversion Hamiltonians and embedding problem instances into Ising spins”. In: *Physical Review A* 77.5 (2008), p. 052331.

[Ale+23] Edoardo Alessandroni et al. “Alleviating the quantum Big-M problem”. In: *arXiv preprint arXiv:2307.10379* (2023).

[RSL22] Ali Rad, Alireza Seif, and Norbert M Linke. “Surviving the barren plateau in variational quantum circuits with Bayesian learning initialization”. In: *arXiv preprint arXiv:2203.02464* (2022).

[GWG21] Austin Gilliam, Stefan Woerner, and Constantin Gonciulea. "Grover adaptive search for constrained polynomial binary optimization". In: *Quantum* 5 (2021), p. 428.

[FLP08] Jean-Charles Faugere, Françoise Levy-dit-Vehel, and Ludovic Perret. "Cryptanalysis of minrank". In: *Annual International Cryptology Conference*. Springer. 2008, pp. 280–296.

[ST06] Giuseppe E Santoro and Erio Tosatti. "Optimization using quantum mechanics: quantum annealing through adiabatic evolution". In: *Journal of Physics A: Mathematical and General* 39.36 (2006), R393.

[LR21] Richard J Lipton and Kenneth W Regan. *Introduction to quantum algorithms via linear algebra*. MIT Press, 2021.

[De 19] Ronald De Wolf. "Quantum computing: Lecture notes". In: *arXiv preprint arXiv:1907.09415* (2019).

[Klu23] Grant Kluber. "Trotterization in Quantum Theory". In: *arXiv:2310.13296* (2023).

[Jia+18] Shuxian Jiang et al. "Quantum annealing for prime factorization". In: *Scientific reports* 8.1 (2018), p. 17667.

[LP17] David A Levin and Yuval Peres. *Markov chains and mixing times*. Vol. 107. American Mathematical Soc., 2017.

[BH02] Endre Boros and Peter L Hammer. "Pseudo-boolean optimization". In: *Discrete applied mathematics* 123.1-3 (2002), pp. 155–225.

Sitography

[Gal] Galatro, Sara. *VQAs_and_QA - GitHub repository*. Last updated March 4, 2024. URL: https://github.com/sara-galatro/VQAs_and_QA.

[IBMa] IBM Quantum. *Qiskit - Transpiler*. Last accessed February 8, 2024. URL: https://docs.quantum.ibm.com/api/qiskit/transpiler.

[Aar] Aaronson, Scott. *Quantum Complexity Theory - Lecture Notes*. As taught in Fall 2010. URL: https://ocw.mit.edu/courses/6-845-quantum-complexity-theory-fall-2010/resources/lecture-notes/.

[IBMb] IBM Quantum. *IBM Quantum Documentation - TwoLocal*. Last accessed February 12, 2024. URL: https://docs.quantum.ibm.com/api/qiskit/qiskit.circuit.library.TwoLocal.

[D-Wa] D-Wave Systems Inc. *14-1026 Next-Generation Topology of D-Wave Quantum Processors*. Published February 25, 2019. URL: https://www.dwavesys.com/media/jwwj5z3z/14-1026a-c_next-generation-topology-of-dw-quantum-processors.pdf.

[D-Wb] D-Wave Systems Inc. *14-1056 Zephyr Topology of D-Wave Quantum Processors*. Published September 22, 2021. URL: https://www.dwavesys.com/media/2uznec4s/14-1056a-a_zephyr_topology_of_d-wave_quantum_processors.pdf.

[D-Wc] D-Wave Systems Inc. *D-Wave Announces 1,200+ Qubit Advantage2™ Prototype in New, Lower-Noise Fabrication Stack, Demonstrating 20x Faster Time-to-Solution on Important Class of Hard Optimization Problems*. Published January 23, 2024. URL: https://www.dwavesys.com/company/newsroom/press-release/d-wave-announces-1-200-qubit-advantage2-prototype-in-new-lower-noise-fabrication-stack-demonstrating-20x-faster-time-to-solution-on-important-class-of-hard-optimization-problems/.

[D-Wd] D-Wave Systems Inc. *D-Wave Announces Availability of 1,200+ Qubit Advantage2™ Prototype in the Leap™ Quantum Cloud Service, Making its Most Performant System Available to Customers Today*. Published February 12, 2024. URL: https://www.dwavesys.com/company/newsroom/press-release/d-wave-announces-availability-of-1-200-qubit-advantage2-prototype/.

[D-We] D-Wave Systems Inc. *The D-Wave Advantage2 Prototype*. Published June 9, 2022. URL: https://www.dwavesys.com/media/eixhdtpa/14-1063a-a_the_d-wave_advantage2_prototype-4.pdf.

[The] The SciPy community. *SciPy Manual - optimize.minimize*. Last accessed February 18, 2024. URL: https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.minimize.html.

[Ram] Ramos-Calderer, Sergi and Lin, Ruge. *Solving systems of Boolean multivariate equations with quantum annealing - Code*. Last accessed February 18, 2024. URL: https://github.com/qiboteam/mq-problem-quantum-annealing.

[D-Wf] D-Wave Systems Inc. *Operation and Timing*. Last accessed February 27, 2024. URL: https://docs.dwavesys.com/docs/latest/c_qpu_timing.html#.

[D-Wg] D-Wave Systems Inc. *QPU Solvers: Decomposing Large Problems*. Last accessed February 29, 2024. URL: https://docs.dwavesys.com/docs/latest/handbook_decomposing.html.

[Rec] Recruit Communications Co. *PyQubo Documentation*. Last accessed March 5, 2024. URL: https://pyqubo.readthedocs.io/en/latest/getting_started.html.

[Gun] Gunzi, Arnaldo. *How to calculate the depth of a quantum circuit in Qiskit?* Published on September 8, 2020. URL: https://medium.com/arnaldo-gunzi-quantum/how-to-calculate-the-depth-of-a-quantum-circuit-in-qiskit-868505abc104.

[Qis23] Qiskit contributors. *Qiskit: An Open-source Framework for Quantum Computing*. 2023. DOI: 10.5281/zenodo.2573505. URL: https://www.ibm.com/quantum/qiskit.

[D-W23] D-Wave Systems Inc. *D-Wave Ocean Software Documentation*. 2023. URL: https://docs.ocean.dwavesys.com/en/stable/.

[D-Wh] D-Wave Systems Inc. *D-Wave Leap Platform*. URL: https://cloud.dwavesys.com/leap/login/?next=/leap/.

[IBMc] IBM Quantum. *IBM Quantum Platform*. URL: https://quantum.ibm.com/.

[IBMd] IBM Quantum. *IBM Quantum Learning - Variational algorithm design*. Last accessed February 14, 2024. URL: https://learning.quantum.ibm.com/course/variational-algorithm-design.

[D-Wi] D-Wave Systems Inc. *Advantage2: Next-generation Experimental Prototype*. Published June 16, 2022. URL: https://www.dwavesys.com/company/newsroom/press-release/ahead-of-the-game-d-wave-delivers-prototype-of-next-generation-advantage2-annealing-quantum-computer/.

Ringraziamenti

Vorrei ringraziare innanzitutto il *Professor Marco Pedicini*, il mio relatore, che da un anno e mezzo a questa parte mi ha seguita ed aiutata nel mio percorso accademico. Dagli esami ai progetti esterni e fino a questa tesi, il suo sapere, i suoi incoraggiamenti e la sua voglia di fare sono stati per me fondamentali. Mi ha spronata a fare sempre meglio, spingendomi al di là delle mie paure, e io le sono veramente grata per tutte le opportunità che mi ha dato. La ringrazio anche per aver saputo come gestire i miei momenti di ansia, ma, soprattutto, per aver creduto in me e nelle mie capacità.

Un ringraziamento speciale va anche al *Dottor Massimo Bernaschi*, co-relatore di questa tesi, che ha accettato di prendermi come tirocinante e di aiutarmi in questo progetto di tesi. La sua guida e la sua conoscenza sono state per me un'ispirazione continua ed hanno reso il lavorare con lei a questo progetto un vero e proprio piacere. É andata proprio come avevamo detto a quel primo colloquio, sa? É stata dura, ho imprecato abbastanza contro il codice, ma mi sono divertita davvero tanto.

Se sono riuscita ad arrivare fino a qui è anche grazie agli sforzi e al sostegno dei miei incredibili genitori Jeannette e Sergio, *mamma e papà*. Mi avete ascoltata parlare di tutti gli esami e di questa tesi per non so quante ore, pur non capendo nulla di quello che stessi dicendo, ma contenti di vedermi e di sentirmi orgogliosa e appassionata a questa materia così inaspettata. Grazie per accompagnarmi sempre col sorriso in questa vita. Questa tesi è mia quanto vostra, ve la dedico tutta.

Ringrazio di cuore anche i miei nonni di Roma, *Nonna Ester e Nonno Lucio*, che, seppur distanti, non hanno mai mancato di farmi sentire il loro affetto e la loro vicinanza.

Se si parla di ringraziamenti, è impossibile non parlare di voi, *Deianira, Antonia, Alessia e Michele*, gli amici di una vita. Ci conosciamo ormai da più di dieci anni e ci siamo sempre stati gli uni per gli altri: tra traslochi, cambiamenti e festeggiamenti, so che non importerà mai davvero quanto siamo distanti o impegnati perché tanto, alla fine, saremo sempre a distanza di una pizza. Grazie per essere il mio punto fisso in questa vita che cambia così velocemente e per ricordarmi sempre di vivere con leggerezza.

Infine, ringrazio i miei compagni di viaggio, i ragazzi del gruppo di *Ubuntu*. Tra le serate giochi, le ore passate in video-chiamata a studiare e i pranzi assieme, mi avete dimostrato che anche io posso trovare un mio posto nel mondo e per questo non vi ringrazierò mai abbastanza. Sono stati cinque anni davvero fuori dal comune, talmente tanto pieni e frenetici che ci sembra di conoscerci da una vita intera. E forse, è proprio per tutti questi momenti folli che questi anni sono stati così magici e divertenti. Al futuro ragazzi e ragazze, spero di poter continuare a brindare con voi in tutti i traguardi che raggiungeremo.

Sapete, avevo anche pensato di fare una menzione speciale a *Matteo*, ma poi ho realizzato che, in fondo, "*we actually don't need it*"!

Voglio chiudere questa tesi con una frase che, al momento, sento particolarmente mia. Il soundtrack di *Hazbin Hotel* è stato in loop per svariate ore nell'ultimo mese, accompagnando in sottofondo tutta la scrittura della tesi. Come potevo non citarne neanche una canzone?

The stage is wrecked, the crowd is gone
But by God, Charlie!
The show, it must go on![7]

[7] Citazione presa dalla canzone "*Finale*" dal soundtrack ufficiale della serie *Hazbin Hotel*. Un titolo accurato per questo *finale* universitario, non trovate?